Computational Complexity:

A Conceptual Perspective

Oded Goldreich

Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Rehovot, Israel.

Working draft (January 2006)

to Dana

II

Preface

The strive for efficiency is ancient and universal, as time and other resources are always in shortage. Thus, the question of which tasks can be performed efficiently is central to the human experience.

A key step towards the systematic study of the aforementioned question is a rigorous definition of the notion of a task and of procedures for solving tasks. These definitions were provided by computability theory, which emerged in the 1930's. This theory focuses on computational tasks, and considers automated procedures (i.e., computing devices and algorithms) that may solve such tasks.

In focusing attention on computational tasks and algorithms, computability theory has set the stage for the study of the computational resources (like time) that are required by such algorithms. When this study focuses on the resources that are necessary for *any* algorithm that solves a particular task (or class of tasks), the study becomes part of the theory of Computational Complexity (also known as Complexity Theory).[1]

Complexity Theory is a central field of the theoretical foundations of Computer Science. It is concerned with the study of the *intrinsic complexity of computational tasks*. That is, a typical Complexity theoretic study looks at the computational resources required to solve a a computational task (or a class of such tasks), rather than at a specific algorithm or algorithmic scheme. Actually, research in Complexity Theory tends to *start with and focus on the computational resources themselves*, and addresses the effect of limiting these resources on the class of tasks that can be solved. Thus, Computational Complexity is the study of the what can be achieved within limited time (and/or other limited natural computational resources).

The (half-century) history of Complexity Theory has witnessed two main research efforts (or directions). The first direction is aimed towards actually establishing concrete lower bounds on the complexity of problems, via an analysis of the evolution of the process of computation. Thus, in a sense, the heart of this direction is a "low-level" analysis of computation. Most research in circuit complexity and in proof complexity falls within this category. In contrast, a second

[1]In contrast, when the focus is on the design and analysis of specific algorithms (rather than on the intrinsic complexity of the task), the study becomes part of a related subfield that may be called Algorithmic Design and Analysis. Furthermore, Algorithmic Design and Analysis tends to be sub-divided according to the domain of mathematics, science and engineering in which the computational tasks arise. In contrast, Complexity Theory typically maintains a unity of the study of tasks solveable within certain resources (regardless of the origins of these tasks).

research effort is aimed at exploring the connections among computational problems and notions, without being able to provide absolute statements regarding the individual problems or notions. This effort may be viewed as a "high-level" study of computation. The theory of NP-completeness as well as the studies of approximation, probabilistic proof systems, pseudorandomness and cryptography all fall within this category.

The current book focuses on the latter effort (or direction). We list several reasons for our decision to focus on the "high-level" direction. The first is the great *conceptual significance* of the known results; that is, many known results (as well as open problems) in this direction have an extremely appealing conceptual message, which can also be appreciated by non-experts. Furthermore, these conceptual aspects may be explained without entering into excessive technical detail. Consequently, the "high-level" direction is more suitable for an exposition in a book of the current nature. Finally, there is a subjective reason: the "high-level" direction is within our own expertise, while this cannot be said about the "low-level" direction.

The last paragraph brings us to a discussion of the nature of the current book, which is captured by the subtitle (i.e., "a conceptual perspective"). Our main thesis is that complexity theory is extremely rich in conceptual content, and that this contents should be explicitly communicated in expositions and courses on the subject. The desire to provide a corresponding textbook is indeed the motivation for writing the current book and its main governing principle.

This book offers a conceptual perspective on complexity theory, and the presentation is designed to highlight this perspective. It is intended mainly for students that wish to learn complexity theory and for educators that intend to teach a course on complexity theory. The book is also intended to promote interest in complexity theory and make it acccessible to general readers with adequate background (which is mainly being comfortable with abstract discussions, definitions and proofs). We expect most readers to have a basic knowledge of algorithms, or at least be fairly comfortable with the notion of an algorithm.

The book focuses on several sub-areas of complexity theory (see the following organization and chapter summaries). In each case, the exposition starts from the intuitive questions addresses by the sub-area, as embodied in the concepts that it studies. The exposition discusses the fundamental importance of these questions, the choices made in the actual formulation of these questions and notions, the approaches that underly the answers, and the ideas that are embedded in these answers. Our view is that these ("non-technical") aspects are the core of the field, and the presentation attempts to reflect this view.

We note that being guided by the conceptual contents of the material leads, in some cases, to technical simplifications. Indeed, for many of the results presented in this book, the presentation of the proof is different (and arguablly easier to understand) than the standard presentations.

Organization and Chapter Summaries

This book consists of ten chapters and seven appendices. The chapters constitute the core of this book and are written in a style adequate for a textbook, whereas the appendices provide additional perspective and are written in the style of a survey article. The relative length and ordering of the chapters (and appendices) does not relfect their relative importance, but rather an attempt at the best logical order (i.e., minimizing the number of forward pointers).

Following are brief summaries of the book's chapters and appendices. Theses summaries are more detailed than those provided in Section 1.1.3 but less detailed than the summaries provided at the begining of each chapter.

Chapter 1: Introduction and Preliminaries. The introduction provides a high-level overview of some of the content of complexity theory as well as discusses some of the characteristic features of this field. The preliminaries provide the relevant background on *computability theory*, which is the setting in which complexity questions are being studied. Most importantly, central notions such as search and decision problems, algorithms that solve such problems, and their complexity are defined. In addition, this part presents the basic notions underlying non-uniform models of computation (like Boolean circuits).

Chapter 2: P, NP and NP-completeness. The P-vs-NP Question can be phrased as asking whether or not finding solutions is harder than checking the correctness of solutions. An alternative formulation in terms of decision problems asks whether or not discovering proofs is harder than verifying their correctness; that is, is proving harder than verifying. It is widely believed that the answer to the two equivalent formulation is that finding (resp., discovering) is harder than checking (resp., verifying); that is, that P is different from NP. At present, when faced with a hard problem in NP, we can only hope to prove that it is not in P assuming that NP is different from P. This is where the theory of NP-completeness, which is based on the notion of a reduction, comes into the picture. In general, one computational problem is reducible to another problem if it is possible to efficiently solve the former when provided with an (efficient) algorithm for solving the latter. A problem (in NP) is NP-complete if any problem in NP is reducible to it. Amazingly enough, NP-complete problems exist, and furthermore hundreds of natural computational problems arising in many different areas of mathematics and science are NP-complete.

Chapter 3: Variations on P and NP. Non-uniform polynomial-time (P/poly) captures efficient computations that are carried out by devices that handle specific input lengths. The basic formalism ignores the complexity of constructing such devices (i.e., a uniformity condition), but a finer formalism (based on "machines that take advice") allows to quantify the amount of non-uniformity. The Polynomial-time Hierarchy (PH) generalizes NP by considering statements expressed by a quantified Boolean formula with a fixed number of alternations of existential and universal quantifiers. It is widely believed that each quantifier alternation adds expressive power to the class of such formulae. The two different classes are related by showing that if NP is contained in P/poly then the Polynomial-time Hierarchy collapses to its second level.

Chapter 4: More Resources, More Power? When using "nice" functions to determine the algorithm's resources, it is indeed the case that more resources allow for more tasks to be performed. However, when "ugly" functions are used for the same purpose, increasing the resources may have no effect. By nice functions we mean functions that can be computed without exceeding the amount of resources that they specify. Thus, we get results asserting, for example, that there are functions computable in cubic-time but not in quadratic-time. In case of non-uniform models of computation, the issue of "nicity" does not arise, and it is easy to establish seperations results.

Chapter 5: Space Complexity. This chapter is devoted to the study of the space complexity of computations, while focusing on two rather extreme cases. The first case is that of algorithms having logarithmic space complexity, which seem a proper and natural subset of the set of polynomial-time algorithms. The second case is that of algorithms having polynomial space complexity, which in turn can solve almost all computational problems considered in this work. Among the results presented in this chapter are a log-space algorithm for exploring (undirected) graphs, a non-deterministic log-space procedure for recognizing directed graphs that are not strongly connected, and complete problems for $\mathcal{NL}$ and $\mathcal{PSPACE}$ (under log-space and poly-time reductions, respectively).

Chapter 6: Randomness and Counting. Various failure types of probabilistic polynomial-time algorithms yield complexity classes such as $\mathcal{BPP}$, $\mathcal{RP}$, and $\mathcal{ZPP}$. The results presented include $\mathcal{BPP} \subset \mathcal{P}/\text{poly}$ and $\mathcal{BPP} \subseteq \Sigma_2$. Turning to counting problems (i.e., counting the number of solutions for NP-type problem), we distinguish between exact counting and approximate counting (in the sense of relative approximation). While any problem in $\mathcal{PH}$ is reducible to the exact counting class $\#\mathcal{P}$, approximate counting (for $\#\mathcal{P}$) is reducible in probabilistic polynomial-time to $\mathcal{NP}$. Additional related topics include $\#\mathcal{P}$-completeness, the complexity of searching for unique solutions, and the relation between approximate counting and generating almost uniformly distributed solutions.

Chapter 7: The Bright Side of Hardness. It turns out that hard problem can be "put to work" to our benefit, most notably in cryptography. One key issue that arises in this context is bridging the gap between "occasional" hardness (e.g., worst-case hardness or mild average-case hardness) and "typical" hardness (i.e., inapproximability). We consider two conjectures that are related to $\mathcal{P} \neq \mathcal{NP}$. The first conjecture is that there are problems that are solvable in exponential-time but are not solvable by (non-uniform) families of small (say polynomial-size) circuits. We show that these types of worst-case conjectures can be transformed into average-case hardness results that yield non-trivial derandomizations of $\mathcal{BPP}$ (and even $\mathcal{BPP} = \mathcal{P}$). The second conjecture is that there are problems in NP for which it is easy to generate (solved) instances that are hard to solve for other people. This conjecture is captured in the notion of *one-way functions*, which are functions that are easy to evaluate but hard to invert (in an average-case sense). We show that functions that are hard to invert in a relatively mild average-case sense yield functions that are hard to invert almost everywhere, and that the latter yield predicates that are very hard to approximate (called *hard-core predicates*). The latter are useful for the construction of general-purpose pseudorandom generators as well as for a host of cryptographic applications.

Chapter 8: Pseudorandom Generators. A fresh view at the *question of randomness* was taken in the theory of computing: It has been postulated that a distribution is pseudorandom if it cannot be told apart from the uniform distribution by an efficient procedure. The paradigm, originally associating efficient procedures with polynomial-time algorithms, has been applied also with respect to a variety of limited classes of such distinguishing procedures. Starting with the general paradigm, we present the case of general-purpose pseudorandom generators (running in polynomial-time and withstanding any polynomial-time distinguisher), as well as derandomization-aimed generators (used towards the derandomization of complexity classes such as $\mathcal{BPP}$), generators withstanding space-bounded distinguishers, and some special-purpose generators.

Chapter 9: Probabilistic Proof Systems. Randomized and interactive verification procedures, giving rise to *interactive proof systems*, seem much more powerful than their deterministic counterparts. In particular, interactive proof systems exist for any set in $\mathcal{PSPACE} \supseteq \mathrm{co}\mathcal{NP}$ (e.g., for the set of unsatisfied propositional formulae), whereas it is widely believed that some sets in $\mathrm{co}\mathcal{NP}$ do *not* have NP-proof systems. Interactive proofs allow the introduction of *zero-knowledge proofs*, which are interactive proofs that yield nothing (to the verifier) beyond the fact that the assertion is indeed valid. Under reasonable complexity assumptions, every set in $\mathcal{NP}$ has a zero-knowledge proof system. (This result has many applications in cryptography.) A third type of probabilistic proof systems is the model of PCPs, standing for *probabilistically checkable proofs*. These are (redundant) NP-proofs that offers a trade-off between the number of (randomly) examined locations in the resulting proof and the confidence in its validity. It particular, a small constant error probability can be obtained by reading a constant number of bits in the redundant NP-proof. The PCP Theorem asserts that NP-proofs can be efficiently transformed into PCPs. The study of PCPs is closely related to the study of the complexity of approximation problems.

Chapter 10: Relaxing the Requirement. In light of the apparent infeasibility of solving numerous natural computational problems, it is natural to seek relaxations of these problems that are both useful for applications and allows for feasible solving procedures. Two such types of relaxation are provided by the notions of approximation and average-case complexity. The notion of approximation refers to the computational problems themselves; that is, for each problem instance we extend the set of admissible solutions. In the context of search problems this means settling for solutions that have a value that is "sufficiently close" to the value of the optimal solution, whereas in the context of decision problems this means settling for procedures that distinguish yes-instances from instances that are "far" from any yes-instance. Turning to average-case complexity, we note that a systemtaic study of this notion requires the development of a non-trivial conceptual framework. A major aspect of this framework is limiting the class of distributions in a way that, on one hand, allows for various types of natural distributions and, on the other hand, prevents the collapse of average-case complexity to worst-case complexity.

Appendix A: Glossary of Complexity Classes. The glossary provides self-contained definitions of most complexity classes mentioned in the book. The glossary is partitioned into two parts, dealing seperately with complexity classes that are defined in terms of algorithms and their resources (i.e., time and space complexity of Turing machines) and complexity classes defined in terms of non-uniform circuit (and referring to their size and depth). The following classes are defined: $\mathcal{P}$, $\mathcal{NP}$, co$\mathcal{NP}$, $\mathcal{BPP}$, $\mathcal{RP}$, co$\mathcal{RP}$, $\#\mathcal{P}$, $\mathcal{PH}$, $\mathcal{E}$, $\mathcal{EXP}$, $\mathcal{NEXP}$, $\mathcal{L}$, $\mathcal{NL}$, $\mathcal{RL}$, $\mathcal{PSPACE}$, $\mathcal{P}$/poly, $\mathcal{NC}^k$ and $\mathcal{AC}^k$.

Appendix B: On the Quest for Lower Bounds. This appendix surveys some attempts at proving lower bounds on the complexity of natural computational problems. The first part, devoted to Circuit Complexity, reviews lower bounds for the *size* of (restricted) circuits that solve natural computational problems. This represents a program whose long-term goal is proving that $\mathcal{P} \neq \mathcal{NP}$. The second part, devoated to Proof Complexity, reviews lower bounds on the length of (restricted) propositional proofs of natural tautologies. This represents a program whose long-term goal is proving that $\mathcal{NP} \neq$ co$\mathcal{NP}$.

Appendix C: On the Foundations of Modern Cryptography. This appendix surveys the foundations of cryptography, which are the paradigms, approaches and techniques used to conceptualize, define and provide solutions to natural security concerns. It presents some of these conceptual tools as well as some of the fundamental results obtained using them. The appendix augments the partial treatment of one-way functions, pseudorandom generators and zero-knowledge proofs, which is included in Chapters 7–9. Using these basic tools, the appendix provides a treatment of basic cryptographic applications such as Encryption, Signatures, and General Cryptographic Protocols.

Appendix D: Probabilistic Preliminaries and Advanced Topics in Randomization. The probabilistic preliminaries include conventions regarding random variables and overviews of three useful inequalities (i.e., Markov Inequality, Chebyshev's Inequality, and Chernoff Bound). The advanced topics include constructions and lemmas regarding families of hashing functions, a study of the sample and randomness complexity of estimating the average value of an arbitrary function, and a study of randomness extraction (i.e., procedures for extracting almost perfect randomness from sources of weak or defected randomness).

Appendix E: Explicit Constructions. Complexity theory provides a clear perspective on the intuitive notion of an explicit construction. This perspective is demonstrated with respect to error correcting codes and expander graphs. On the topic of codes, the appendix focuses on various computational aspects, containing a review of several popular constructions as well as a construction of a binary code of constant rate and constant relative distance. Also included are a brief review of the notions of locally testable and locally decodable codes, and a useful upper-bound on the number of codewords that are close to any single word. Turning to expander graphs, the appendix conatins a review of two standard definitions of expanders, two levels of explicitness, two properties of expanders that are related to (single-step and multi-step) random walks, and two explicit constructions of expander graphs.

Appendix F: Some Omitted Proofs. This appendix contains some proofs that were not included in the main text (for a variety of reasons) and still are beneficial as alternatives to the original and/or standard presentations. Included are proofs that $\mathcal{PH}$ is reducible to $\#\mathcal{P}$ via randomized Karp-reductions, and that $\mathcal{IP}(f) \subseteq \mathcal{AM}(O(f)) \subseteq \mathcal{AM}(f)$, for any function f such that $f(n) \in \{2, ..., \mathrm{poly}(n)\}$.

Appendix G: Some Computational Problems. This appendix includes definitions of most specific computational problems that are reffered to in the main text. In particular, it contains a brief introduction to graph algorithms, boolean formulae and finite fields.

Acknowledgments

My perspective on complexity theory was most influenced by Shimon Even and Leonid Levin. In fact, it was hard not to be influenced by these two remarkable and highly opinioned researchers (especially for somebody like me who was fortunate to spend a lot of time with them).[2]

Shimon Even viewed complexity theory as the study of the limitations of algorithms, a study concerned with natural computational resources and natural computational tasks. Complexity theory was there to guide the engineer and to address the deepest questions that bother an intellectually curious computer scientist. That is, complexity theory emerged out of the real world (but of course, as any dicipline, it is developed according to its intrinsic logic).

Leonid Levin emphasized the general principles that underly complexity theory, rejecting any "model-dependent effects" as well as the historical ties between complexity theory and automata and formal language theory. (Indeed, the historical links could not be denied, but they cannot justify coupling two fundamentally different theories, when such a coupling promotes a wrong perspective on complexity theory.)

I believe that this book (as well as much of my work) presents an odd hybrid of the different exposition approaches of Shimon Even and Leonid Levin (cf. [67] versus [145]). The desire to be intuitive and natural is Shimon's, while the refusal to be concrete and specific is Leonid's. I hope that this book proves that these are not as contradictory as one assumes.

I wish to acknowledge the influece of numerous other collegues on my professional perspectives and attitudes. These include Shafi Goldwasser, Dick Karp, Silvio Micali, and Avi Wigderson.

Lastly, I wish to thank many collegues for their comments and advise regarding earlier versions of this text. A partial list includes Omer Reingold, Dana Ron, Ronen Shaltiel, Amir Shpilka, Madhu Sudan, Salil Vadhan, and Avi Wigderson.

Relation to previous texts of mine. Some of the text of this book has been adapted from previous texts of mine. In particular, Chapters 8 and 9 were written based on my surveys [86, Chap. 3] and [86, Chap. 2], respectively; but the exposition has been extensively revised to fit the significantly different aims of the current

[2] Shimon Even was my graduate studies adviser (at the Technion, 1980-83); whereas I had a lot of meetings with Leonid Levin during my post-doctoral period (at MIT, 1983-86).

book. Similarly, Chapter 7 and Appendix C were written based on my survey [86, Chap. 1] and books [87, 88]; but, again, the previous texts are very different in many ways. In contrast, Appendix B was adapted with relatively little modifications from a section of an article by Avi Wigderson and myself [103].

Contents

List of Figures

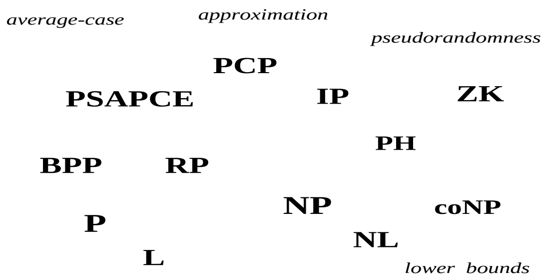
Chapter 1

Introduction and Preliminaries

You can start by putting the DO NOT DISTURB *sign.*

Cay, in *Desert Hearts* (1985).

The current chapter consists of two parts. The first part provides a high-level introduction to complexity theory. This introduction is much more detailed than the laconic statements made in the preface, but is quite sparse when compared to the richness of the field. In addition, the introduction contains several important comments regarding the contents, approach and style of the current book.



The second part of this chapter provides the necessary preliminaries to the rest of the book. It includes a discussion of computational tasks and computational models, as well as natural complexity measures associated with the latter. More specifically, this part recalls the basic notions and results of computability theory (including the definition of Turing machines, some undecidability results, the notion of universal machines, and the definition of oracle machines). In addition, this part presents the basic notions underlying non-uniform models of computation (like Boolean circuits).

1.1 Introduction

This section consists of two parts: the first part refers to the area itself, whereas the second part refers to the current book. The first part provides a brief overview of Complexity Theory (Section 1.1.1) as well as some reflections about its characteristics (Section 1.1.2). The second part describes the contents of this book (Section 1.1.3), the considerations underlying the choice of topics as well as the way the are presented (Section 1.1.4), and various notations and conventions (Section 1.1.5).

1.1.1 A brief overview of Complexity Theory

Out of the tough came forth sweetness[1]

Judges, 14:14

Complexity Theory is concerned with the study of the *intrinsic complexity* of computational tasks. Its "final" goals include the determination of the complexity of any well-defined task. Additional "final" goals include obtaining an understanding of the relations between various computational phenomena (e.g., relating one fact regarding computational complexity to another). Indeed, we may say that the former type of goals is concerned with *absolute* answers regarding specific computational phenomena, whereas the latter type is concerned with questions regarding the *relation* between computational phenomena.

Interestingly, the current success of Complexity Theory in coping with the latter type of goals has been more significant. In fact, the failure to resolve questions of the "absolute" type, led to the flourishing of methods for coping with questions of the "relative" type. Putting aside for a moment the frustration caused by the failure, we must admit that there is something fascinating in the success: in some sense, establishing relations between phenomena is more revealing than making statements about each phenomenon. Indeed, the first example that comes to mind is the theory of NP-completeness. Let us consider this theory, for a moment, from the perspective of these two types of goals.

Complexity theory has failed to determine the intrinsic complexity of tasks such as finding a satisfying assignment to a given (satisfiable) propositional formula or finding a 3-coloring of a given (3-colorable) graph. But it has established that these two seemingly different computational tasks are in some sense the same (or, more precisely, are computationally equivalent). The author finds this success amazing and exciting, and hopes that the reader shares his feeling. The same feeling of wonder and excitement is generated by many of the other discoveries of Complexity theory. Indeed, the reader is invited to join a fast tour of some of the other questions and answers that make up the field of Complexity theory.

We will indeed start with the "P versus NP Question". Our daily experience is that it is harder to solve a problem than it is to check the correctness of a solution

[1] The quote is commonly used to mean that benefit arose out of misfortune.

(e.g., think of either a puzzle or a research problem). Is this experience merely a coincidence or does it represent a fundamental fact of life (or a property of the world)? Could you imagine a world in which solving any problem is not significantly harder than checking a solution to it? Would the term "solving a problem" not lose its meaning in such a hypothetical (and impossible in our opinion) world? The denial of the plausibility of such a hypothetical world (in which "solving" is not harder than "checking") is what "P different from NP" actually means, where P represents tasks that are efficiently solvable and NP represents tasks for which solutions can be efficiently checked.

The mathematically (or theoretically) inclined reader may also consider the task of proving theorems versus the task of verifying the validity of proofs. Indeed, finding proofs is a special type of the aforementioned task of "solving a problem" (and verifying the validity of proofs is a corresponding case of checking correctness). Again, "P different from NP" means that there are theorems that are harder to prove than to be convinced of their correctness when presented with a proof. This means that the notion of a proof is meaningful (i.e., that proofs do help when trying to be convinced of the correctness of assertions). Here NP represents sets of assertions that can be efficiently verified with the help of adequate proofs, and P represents sets of assertions that can be efficiently verified from scratch (i.e., without proofs).

In light of the foregoing discussion it is clear that the P-versus-NP Question is a fundamental scientific question of far-reaching consequences. The fact that this question seems beyond our current reach led to the development of the theory of NP-completeness. Loosely speaking, this theory identifies a set of computational problems that are as hard as NP. That is, the fate of the P-versus-NP Question lies with each of these problems: if any of these problems is easy to solve then so are all problems in NP. Thus, showing that a problem is NP-complete provides evidence to its intractability (assuming, of course, "P different than NP"). Indeed, demonstrating NP-completeness of computational tasks is a central tool in indicating hardness of natural computational problems, and it has been used extensively both in computer science and in other disciplines. NP-completeness indicates not only the conjectured intractability of a problem but rather also its "richness" in the sense that the problem is rich enough to "encode" any other problem in NP. The use of the term "encoding" is justified by the exact meaning of NP-completeness, which in turn is based on establishing relations between different computational problems (without referring to their "absolute" complexity).

The foregoing discussion of the P-versus-NP Question also hints to *the importance of representation*, a phenomenon that is central to complexity theory. In general, complexity theory is concerned with problems the solutions of which are implicit in the problem's statement (or rather in the instance). That is, the problem (or rather its instance) contains all necessary information, and one merely needs to process this information in order to supply the answer.[2] Thus, complexity theory is

[2]In contrast, in other disciplines, solving a problem may require gathering information that is not available in the problem's statement. This information may either be available from auxiliary (past) records or be obtained by conducting new experiments.

concerned with manipulation of information, and its transformation from one representation (in which the information is given) to another representation (which is the one desired). Indeed, a solution to a computational problem is merely a different representation of the information given; that is, a representation in which the answer is explicit rather than implicit. For example, the answer to the question of whether or not a given Boolean formula is satisfiable is implicit in the formula itself (but the task is to make the answer explicit). Thus, complexity theory clarifies a central issue regarding representation; that is, the distinction between what is explicit and what is implicit in a representation. Furthermore, it even suggests a quantification of the level of non-explicitness.

In general, complexity theory provides new viewpoints on various phenomena that were considered also by past thinkers. Examples include the aforementioned concepts of proofs and representation as well as concepts like randomness, knowledge, interaction, secrecy and learning. We next discuss some of these concepts and the perspective offered by complexity theory.

The concept of *randomness* has puzzled thinkers for ages. Their perspective can be described as ontological: they asked "what is randomness" and wondered whether it exist at all (or is the world deterministic). The perspective of complexity theory is behavioristic: it is based on defining objects as equivalent if they cannot be told apart by any efficient procedure. That is, a coin toss is (defined to be) "random" (even if one believes that the universe is deterministic) if it is infeasible to predict the coin's outcome. Likewise, a string (or a distribution of strings) is "random" if it is infeasible to distinguish it from the uniform distribution (regardless of whether or not one can generate the latter). Interestingly, randomness (or rather pseudorandomness) defined this way is efficiently expandable; that is, under a reasonable complexity assumption (to be discussed next), short pseudorandom strings can be deterministically expanded into long pseudorandom strings. Indeed, it turns out that randomness is intimately related to intractability. Firstly, note that the very definition of pseudorandomness refers to intractability (i.e., the infeasibility of distinguishing a pseudorandomness object from a uniformly distributed object). Secondly, as hinted above, a complexity assumption that refers to the existence of functions that are easy to evaluate but hard to invert (called *one-way functions*) imply the existence of deterministic programs (called *pseudorandom generators*) that stretch short random seeds into long pseudorandom sequences. In fact, it turns out that the existence of pseudorandom generators is equivalent to the existence of one-way functions.

Complexity theory offers its own perspective on the concept of *knowledge* (and distinguishes it from information). It views knowledge as the result of a hard computation. Thus, whatever can be efficiently done by anyone is not considered knowledge. In particular, the result of an easy computation applied to publicly available information is not considered knowledge. In contrast, the value of a hard to compute function applied to publicly available information is knowledge, and if somebody provides you with such a value then it has provided you with knowledge. This discussion is related to the notion of *zero-knowledge* interactions, which are interactions in which no knowledge is gained. Such interactions may still be useful, because they may assert the *correctness* of specific data that was provided beforehand.

The foregoing paragraph has explicitly referred to *interaction*. It has pointed one possible motivation for interaction: gaining knowledge. It turns out that interaction may help in a variety of other contexts. For example, it may be easier to verify an assertion when allowed to interact with a prover rather than when reading a proof. Put differently, interaction with some teacher may be more beneficial than reading any book. We comment that the added power of such *interactive proofs* is rooted in their being randomized (i.e., the verification procedure is randomized), because if the verifier's questions can be determined beforehand then the prover may just provide the transcript of the interaction as a traditional written proof.

Another concept related to knowledge is that of *secrecy*: knowledge is something that one party has while another party does not have (and cannot feasibly obtain by itself) – thus, in some sense knowledge is a secret. In general, complexity theory is related to *Cryptography*, where the latter is broadly defined as the study of systems that are easy to use but hard to abuse. Typically, such systems involve secrets, randomness and interaction as well as a complexity gap between the ease of proper usage and the infeasibility of causing the system to deviate from its prescribed behavior. Thus, much of Cryptography is based on complexity theoretic assumptions and its results are typically transformations of relatively simple computational primitives (e.g., one-way functions) into more complex cryptographic applications (e.g., a secure encryption scheme).

We have already mentioned the context of *learning* when referring to learning from a teacher versus learning from a book. Recall that complexity theory provides evidence to the advantage of the former. This is in the context of gaining knowledge about publicly available information. In contrast, computational learning theory is concerned with learning objects that are only partially available to the learner (i.e., learning a function based on its value at a few random locations or even at locations chosen by the learner). Complexity theory sheds light on the intrinsic limitations of learning (in this sense).

Complexity theory deals with a variety of computational tasks. We have already mentioned two fundamental types of tasks: *searching for solutions* (or "finding solutions") and *making decisions* (e.g., regarding the validity of assertion). We have also hinted that in some cases these two types of tasks can be related. Now we consider two additional types of tasks: *counting the number of solutions* and *generating random solutions*. Clearly, both the latter tasks are at least as hard as finding arbitrary solutions to the corresponding problem, but it turns out that for some natural problems they are not significantly harder. Specifically, under some natural conditions on the problem, approximately counting the number of solutions and generating an approximately random solution is not significantly harder than finding an arbitrary solution.

Having mentioned the notion of *approximation*, we note that the study of the complexity of finding approximate solutions has also received a lot of attention. One type of approximation problems refers to an objective function defined on the set of potential solutions. Rather than finding a solution that attains the optimal value, the approximation task consists of finding a solution that attains an "almost optimal" value, where the notion of "almost optimal" may be understood in different ways giving rise to different levels of approximation. Interestingly, in many cases even a very relaxed level of approximation is as difficult to achieve as the original (exact) search problem (i.e., finding an approximate solution is as hard as finding an optimal solution). Surprisingly, these hardness of approximation results are related to the study of *probabilistically checkable proofs*, which are proofs that allow for ultra-fast probabilistic verification. Amazingly, every proof can be efficiently transformed into one that allows for probabilistic verification based on probing a *constant* number of bits (in the alleged proof). Turning back to approximation problems, we note that in other cases a reasonable level of approximation is easier to achieve than solving the original (exact) search problem.

Approximation is a natural relaxation of various computational problems. Another natural relaxation is the study of *average-case complexity*, where the "average" is taken over some "simple" distributions (representing a model of the problem's instances that may occur in practice). We stress that, although it was not stated explicitly, the entire discussion so far has referred to "worst-case" analysis of algorithms. We mention that worst-case complexity is a more robust notion than average-case complexity. For starters, one avoids the controversial question of what are the instances that are "important in practice" and correspondingly the selection of the class of distributions for which average-case analysis is to be conducted. Nevertheless, a relatively robust theory of average-case complexity has been suggested, albeit it is far less developed than the theory of worst-case complexity.

In view of the central role of randomness in complexity theory (as evident, say, in the study of pseudorandomness, probabilistic proof systems, and cryptography), one may wonder as to whether the randomness needed for the various applications can be obtained in real-life. One specific question, which received a lot of attention, is the possibility of "purifying" randomness (or "extracting good randomness from bad sources"). That is, can we use "defected" sources of randomness in order to implement almost perfect sources of randomness. The answer depends, of course, on the model of such defected sources. This study turned out to be related to complexity theory, where the most tight connection is between some type of *randomness extractors* and some type of pseudorandom generators.

So far we have focused on the time complexity of computational tasks, while relying on the natural association of efficiency with time. However, time is not the only resource one should care about. Another important resource is *space*: the amount of (temporary) memory consumed by the computation. The study of space complexity has uncovered several fascinating phenomena, which seem to indicate a fundamental difference between space complexity and time complexity. For example, in the context of space complexity, verifying proofs of validity of assertions (of any specific type) has the same complexity as verifying proofs of invalidity for the same type of assertions.

In case the reader feels dizzy, it is no wonder. We took an ultra-fast air-tour of some mountain tops, and dizziness is to be expected. Needless to say, the rest of the book is in a totally different style. We will climb some of these mountains by foot, step by step, and will stop to look around and reflect.

Absolute Results (a.k.a. Lower-Bounds). As stated up-front, absolute results are not known for many of the "big questions" of complexity theory (most notably the P-versus-NP Question). However, several highly non-trivial absolute results have been proved. For example, it was shown that using negation can speed-up the computation of monotone functions (which do not require negation for their mere computation). In addition, many promising techniques were introduced and employed with the aim of providing a low-level analysis of the progress of computation. However, as stated in the preface, the focus of this book is elsewhere.

1.1.2 Characteristics of Complexity Theory

We are successful because we use the right level of abstraction

Avi Wigderson (1996)

Using the "right level of abstraction" seems to be a main characteristic of the Theory of Computation at large. The right level of abstraction means abstracting away second-order details, which tend to be context-dependent, while using definitions that reflect the main issues (rather than abstracting them away too). Indeed, using the right level of abstraction calls for an extensive exercising of good judgment, and one indication for having chosen the right questions is the result of their study.

One major choice of the theory of computation, which is currently taken for granted, is the *choice of a model of computation and corresponding complexity measures and classes*. Two extreme choices that were avoided are a too realistic model and a too abstract model. On the one hand, the main model of computation used in complexity theory does not try to reflect (or mirror) the specific operation of real-life computers used at a specific point in time. Such a choice would have made it very hard to develop complexity theory as we know it and to uncover the fundamental relations discussed in this book: The mass of details would have obscured the view. On the other hand, avoiding any reference to a concrete model (like in the case of recursive function theory) does not encourage the introduction and study of natural measures of complexity. Indeed, as we shall see in Section 1.2.3, the choice was (and is) to use a simple model of computation (which does not mirror real-life computers), while avoiding any effects that are specific to that model (by keeping a eye on a host of variants and alternative models). The freedom from the specifics of the basic model is obtained by considering complexity classes that are invariant under a change of model (as long as the alternative model is "reasonable").

Another major choice is to use *asymptotic analysis*. Specifically, we consider the complexity of an algorithm as a function of its input length, and study the asymptotic behavior of this function. It turns out that structure that is hidden by concrete quantities appears at the limit. Furthermore, depending on the case, we classify functions according to different criteria. For example, in case of time complexity we consider classes of functions that are closed under multiplication, whereas in case of space complexity we consider closure under addition. In each case, the choice is governed by the nature of the complexity measure being considered. Indeed, one could have developed a theory without using these conventions, but this would have resulted in a far more cumbersome theory. For example, rather than saying that finding a satisfying assignment for a given formula is polynomial-time reducible to deciding satisfiability other of formulae, one could have said stated the exact functional dependence of the complexity of the search problem on the complexity of the decision problem.

Both aforementioned choices are common to other branches of the theory of computation. What makes complexity theory unique is its commitment to the most basic question of the theory of computation: *what can be efficiently computed*? This commitment is reflected by the area's *primary focus* on the class of efficient procedures, regardless of their goal. This focus is responsible for the area's failures and successes: It has failed to obtain a definite answer regarding the power and limitations of efficient procedures, but it has obtained many connections between computational phenomena that reflect what efficient procedures can or cannot do.

1.1.3 Contents of this book

This book consists of ten chapters and seven appendices. The chapters constitute the core of this book and are written in a style adequate for a textbook, whereas the appendices provide additional perspective and are written in the style of a survey article.

Section 1.2 and Chapter 2 are a prerequisite to the rest of the book. Technically, notions and results that appear in these parts are extensively used in the rest of the book. More importantly, the former parts are the conceptual framework that shapes the field and provides a good perspective on the questions and answers provided. Section 1.2 and Chapter 2 provide the very basic material that must be understood by anybody having an interest in complexity theory. The rest of the book covers more advanced material. Although some advanced chapters refer to material in other advanced chapters, the relation between these chapters is not a fundamental one. Thus, one may choose to read and/or teach an arbitrary subset of the advanced chapters and do it in an arbitrary order, provided one is willing to follow the relevant references to parts of other chapters. Needless to say, we recommend following the order presented in this book.

Section 1.2: Preliminaries. This section provides the relevant background on computability theory, which is the basis for the rest of this book (as well as for complexity theory at large). Most importantly, it contains a discussion of central notions such as search and decision problems, algorithms that solve such problems, and their complexity. In addition, this section presents non-uniform models of computation (like Boolean circuits).

Chapter 2: P, NP and NP-completeness. This chapter presents the P-vs-NP Question both in terms of search problems and in terms of decision problems. The second main topic of this chapter is the theory of NP-completeness. The chapter also provides a treatment of the general notion of a (polynomial-time) reduction, with special emphasis on self-reducibility. Additional topics include the existence of problems in NP that are neither NP-complete nor in P, optimal search algorithms, the class coNP, and promise problems.

Chapter 3: Variations on P and NP. This chapter provides a treatment of non-uniform polynomial-time (P/poly) and of the Polynomial-time Hierarchy (PH). Each of the two classes is defined in two equivalent ways (e.g., P/poly is defined both in terms of circuits and in terms of "machines that take advice"). Id addition, it is shown that if NP is contained in P/poly then the Polynomial-time Hierarchy collapses to its second level.

Chapter 4: More Resources, More Power? This chapter provides separation results asserting, for example, that there are functions computable in cubic-time but not in quadratic-time. These results depend on using bounding functions that can be computed without exceeding the amount of resources that they specify. In contrast, if this condition is not satisfied then increasing the resources may have no effect.

Chapter 5: Space Complexity. Among the results presented in this chapter are a log-space algorithm for exploring (undirected) graphs, a non-deterministic log-space procedure for recognizing directed graphs that are not strongly connected, and complete problems for $\mathcal{NL}$ and $\mathcal{PSPACE}$ (under log-space and poly-time reductions, respectively).

Chapter 6: Randomness and Counting. This chapter focuses on randomized complexity classes (i.e., $\mathcal{BPP}$, $\mathcal{RP}$, and $\mathcal{ZPP}$) and the counting class $\#\mathcal{P}$. Various failure types of probabilistic polynomial-time algorithms yield complexity classes such as $\mathcal{BPP}$, $\mathcal{RP}$, and $\mathcal{ZPP}$. Turning to counting problems we distinguish between exact counting and approximate counting (in the sense of relative approximation). The results presented in this chapter include $\mathcal{BPP} \subseteq \mathcal{P}/\mathrm{poly} \cap \Sigma_2$, the $\#\mathcal{P}$-completeness of the permanent, the connection between approximate counting and uniform generation of solutions, and the randomized reductions of approximate counting to $\mathcal{NP}$ and of $\mathcal{NP}$ to solving problems with unique solutions.

Chapter 7: The Bright Side of Hardness. We consider two conjectures that are related to $\mathcal{P} \neq \mathcal{NP}$. The first conjecture is that there are problems in $\mathcal{E}$ that are not solvable by (non-uniform) families of small (say polynomial-size) circuits, whereas the second conjecture is equivalent to the notion of *one-way functions*. Most of this chapter is devoted to converting these conjectures into tools that can be used for non-trivial dearndomizations of $\mathcal{BPP}$ and for a host of cryptographic applications.

Chapter 8: Pseudorandom Generators. The starting point of this chapter is the definition of a distribution as pseudorandom if it cannot be told apart from the uniform distribution by an efficient procedure. It presents the definition of general-purpose pseudorandom generators (running in polynomial-time and withstanding any polynomial-time distinguisher) as a special case of a general paradigm. The chapter also contains a presentation of other instatiations of the latter paradigm, including generators aimed at derandomizating complexity classes such as $\mathcal{BPP}$, generators withstanding space-bounded distinguishers, and some special-purpose generators.

Chapter 9: Probabilistic Proof Systems. This chapter provides an introduction to three types of probabilistic proof systems: *interactive proofs*, *zero-knowledge proofs*, and *probabilistic checkable proofs*. These proof systems share a common (untraditional) feature – they carry a probability of error; yet, this probability is explicitly bounded and can be reduced by successive application of the proof system. The gain in allowing this untraditional relaxation is substantial, as they enable the construction of proof systems with properties that seem impossible to achieve via traditional proof systems.

Chapter 10: Relaxing the Requirement. This chapter provides a treatment of two types of approximation problems and a theory of average-case (or rather typical-case) complexity. The traditional type of approximation problems refers to search problems and consists of a relaxation of standard optimization problems. The second type is known as "property testing" and consists of a relaxation of standard decision problems. The theory of average-case complexity involves several non-trivial definitional choices (e.g., an adequate choice of the class of distributions).

Appendix A: Glossary of Complexity Classes. The glossary provides self-contained definitions of most complexity classes mentioned in the book.

Appendix B: On the Quest for Lower Bounds. The first part, devoted to Circuit Complexity, reviews lower bounds for the *size* of (restricted) circuits that solve natural computational problems. The second part, devoted to Proof Complexity, reviews lower bounds on the length of (restricted) propositional proofs of natural tautologies.

Appendix C: On the Foundations of Modern Cryptography. The first part of this appendix augments the partial treatment of one-way functions, pseudorandom generators and zero-knowledge proofs, which is included in Chapters 7–9. Using these basic tools, the second part provides a treatment of basic cryptographic applications such as Encryption, Signatures, and General Cryptographic Protocols.

Appendix D: Probabilistic Preliminaries and Advanced Topics in Randomization. The probabilistic preliminaries include conventions regarding random variables and overviews of three useful inequalities (i.e., Markov Inequality, Chebyshev's Inequality, and Chernoff Bound). The advanced topics include constructions families of *hashing* functions and variants of the Leftover Hashing Lemma, and overviews of *samplers* and *extractors* (or, more generally, the problem of randomness extraction).

Appendix E: Explicit Constructions. This appendix focuses on various computational aspects of error correcting codes and expander graphs. On the topic of codes, the appendix contains a review of the Hadamard code, Reed-Solomon codes, Reed-Muller codes, and a construction of a binary code of constant rate and constant relative distance. Also included are a brief review of the notions of locally testable and locally decodable codes, and a list decoding bound. On the topic of expander graphs, the appendix contains a review of standard definitions and properties as well as a presentation of the Margulis-Gabber-Galil and the Zig-Zag constructions.

Appendix F: Some Omitted Proofs. This appendix contains some proofs that are beneficial as alternatives to the original and/or standard presentations. Included are proofs that $\mathcal{PH}$ is reducible to $\#\mathcal{P}$ via randomized Karp-reductions, and that $\mathcal{IP}(f) \subseteq \mathcal{AM}(O(f)) \subseteq \mathcal{AM}(f)$, for any function f such that $f(n) \in \{2, ..., \text{poly}(n)\}$.

Appendix G: Some Computational Problems. This appendix contains a brief introduction to graph algorithms, Boolean formulae, and finite fields.

Bibliography. As stated in Section 1.1.4, we tried to keep the bibliographic list as short as possible (and still reached a couple of hundreds of entries). As a result many relevant references were omitted. We tried, however, not to omit references to key papers in an area. In general, our choice of references was biased in favour of textbooks and survey articles.

Absent from this book. As stated in the preface, the current book does not provide a uniform cover of the various areas of complexity theory. Notable omissions include the areas of *circuit complexity* (cf. [43, 218]) and *proof complexity* (cf. [24]), which are briefly reviewed in Appendix B. Additional topics that are commonly covered in complexity theory courses but omitted here include the study of *branching programs* and *decision trees* (cf. [219]), *parallel computation* [130], and *communication complexity* [138]. Finally, we mention a two areas that we consider related to complexity theory, although this view is not very common. These areas are *distributed computing* [15] and *computational learning theory* [132].

1.1.4 Approach and style of this book

Explanations and motivations are merely methods for introducing superficial redundancy. Since the role of redundancy is to ensure error-correction, one better leave the implementation of these mechanisms to experts, trusting them to use the best error-correcting codes known at the time.

Leonid A Levin (1984)

Although people say that Levin is an extremist, the foregoing quote perfectly represent a common opinion regarding the presetation of scientific work. According to this opinion, the most important aspect of a scientific work is the technical result that it achieves and the rest is redundency introduced for the sake of "error correction" and/or comfort. It is further believed that, like in a work of art, the interpretation of the work should be left with the reader (or viewer or listener).

The author disagrees with the aforementioned opinions, and argues that there is a fundamental difference between art and science, and that this difference refers exactly to the meaning of a piece of work. Science is concerned with meaning (and not with form), and in its quest for truth and/or understanding it follows philosophy (and not art). The author holds the opinion that the most important aspects of a scientific work are the intuitive question that it addresses, the reason that it addresses this question, the way it phrases the question, the approach that underlies its answer, and the ideas that are embedded in the answer. Following this view, it is important to communicate these aspects of the work, and the current book is written accordingly.

These issues are even more acute when it comes to complexity theory, because this field is extremely rich in conceptual content. Unfortunately, this content is rarely communicated (explicitly) in books and/or extensive surveys of the area.[3] The annoying (and quite amazing) consequences are students that have only a vague understanding of the *meaning* and general relevance of the fundamental notions and results that they were taught. The author's view is that these consequences are easy to avoid by taking the time to explicitly discuss the *meaning* of definitions and results. A related issue is using the "right" definitions (i.e., those that reflect better the fundamental nature of the notion being defined) and teaching things in the (conceptually) "right" order.

1.1.4.1 Focus on conceptual issues

In accordance with the foregoing, this book starts from the intuitive questions addressed by complexity theory, explains the fundamental importance of these questions, the specific ways that they are phrased, the approaches that underly the answers, and the ideas that are embedded in these answers. Thus, a significant

[3]It is tempting to speculate on the reasons for this phenomenon. One speculation is that communicating the conceptual content of complexity theory involves making bold philosophical assertions that are technically straightforward, whereas this combination does not fit the character of most researchers in complexity theory.

portion of the text is devoted to motivating discussions that refer to the concepts and ideas that underly the definitions and results.

The material is organized around conceptual themes, which reflect fundamental notions and/or general questions. Specific computational problems are rarely referred to, with exceptions that are used either for sake of clarity or because the specific problem happens to capture a general conceptual phenumenon. For example, in this book, complete problems are always secondary to the class for which they are complete.[4]

We tried to avoid the presentation of material that, in our opinion, is neither the "last word" on the subject nor represents the "right" way of approaching the subject. Thus, we do not always present the "best" known result.

1.1.4.2 On a few specific choices

Our technical presentation often differs from the standard one. In many cases this is due to conceptual considerations. At times, this leads to some technical simplifications. In this section we only discuss general themes and/or choices that have a global impact on much of the presentation.

Avoiding non-deterministic machines. We try to avoid non-deterministic machines as much as possible. As argued in several places (e.g., Section 2.1.4), we believe that these fictitious "machines" have a negative effect both from a conceptual and technical point of view. The conceptual damage caused by using non-deterministic machines is that it is unclear why one should care about what such machines can do. Needless to say, the reason to care is clear when noting that these fictitious "machines" offer a (convinient or rather slothful) way of phrasing fundamental issues. The technical damage caused by using non-deterministic machines is that they tend to confuse the students. Furthermore, they do not offer the best way to handle more advanced issues (e.g., counting classes).

In contrast, we use search problems as the basis for much of the presentation. Specifically, the class $\mathcal{PC}$ (see Definition 2.3), which consists of search problems having efficiently checkable solutions, plays a central role in our presentation. Indeed, defining this class is slightly more complicated than the standard definition of $\mathcal{NP}$ (based on non-deterministic machines), but the techincal benefits start accumulating as we proceed. Needless to say, the class $\mathcal{PC}$ is a fundamental class of computational problems and this fact serves as the main motivation to its presentation. (Indeed, the most conceptually appealing phrasing of the P-vs-NP Question consists of asking whethere every search problem in $\mathcal{PC}$ can be solved efficiently.)

[4] We admit that a very natural computational problem can give rise to a class of problems that are computationally equivalent to it, and that in such a case the class may be less interesting than the original problem. Furthermore, in some cases, the historical evolution actually went from a specific computational problem to a class of problems that are computationally equivalent to it. However, in all cases presented in this book, a retrospective evaluation suggests that the class is actually more important than the original problem (see, e.g., $\mathcal{NP}$ and $\#\mathcal{P}$).

Avoiding model-dependent effects. Our focus is on the notion of efficient computation. A rigorous definition of this notion seems to require reference to some concrete model of computation; however, all questions and answers considered in this book are invariant under the choice of such a concrete model, provided of course that the model is "reasonable" (which, needless to say, is a matter of intuition). Indeed, the foregoing text reflects the tension between the need to make rigorous definitions and the desire to be independent of technical choices, which are unavoidable when making such definitions. Furthermore, in constrast to common beliefs, the foregoing comments refer not only to time complexity but also to space complexity. However, in both cases, the claim of invariance may not hold for marginally small resources (e.g., linear-time or sub-logarithic space).

In contrast to the foregoing paragraph, in some cases we choose to be specific. The most notorious case is the association of efficiency with polynomial-time (see §1.2.3.4). Indeed, all the questions and answers regarding efficient computation can be phrased without referring to polynomial-time (i.e., by stating explicit functional relations between the complexities of the problems involved), but such a generalized treatment will be painful to follow.

1.1.4.3 On the presentation of technical material

In general, the more complex the issues are, the more levels of expositions we employ, starting from the most high-level exposition, and when necessary providing more than one level of details. In particular, whenever a proof is not very simple, we try to present the key ideas first and implementation details later. We also try to clearly indicate the passage from a high-level presentation to implementation details (e.g., by using phrases such as "details follow"). In some cases, especially in the case of advanced results, only proof sketches are provided and the implication is that the reader should be able to fill-up the missing details.

Few results are stated without proof. In some of these cases the proof idea or a proof overview is provided, but the reader is *not* supposed to be able to fill-up the highly non-trivial details. (We clearly indicate that this is the case in the text.) One notable example is the proof of the PCP Theorem (Theorem 9.16).

1.1.4.4 Organizational principles

Each of the main chapters starts with a high-level summary and ends with chapter notes and exercises. The latter are not aimed at testing or inspiring creativity, but are rather designed to help and verify the basic understanding of the main text.

As stated in the preface, this book focuses on the high-level approach to complexity theory and the low-level approach (i.e., lower bounds) is briefly reviewed in Appendix B. Other appendices contain material that is closely related to complexity theory but is not an integral part of it (e.g., the Foundations of Cryptography).[5]

[5]As further articulated in Section 7.1, we recommend not including a basic treatment of cryptography within a course on complexity theory. Indeed, cryptography may be claimed to be the most appealing application of complexity theory, but a superficial treatment of cryptography (from this perspective) is likely to be misleading and cause more harm than good.

In an attempt to keep the bibliographic list from becoming longer than an average chapter, we omitted many relevant references. One trick used towards this end is referring to lists of references in other texts, especially when these texts are cited anyhow. Indeed, our choices of references were biased in favour of textbooks and survey articles, because we believe that they provide the best way to further learn about a research direction and/or approach. We tried, however, not to omit references to key papers in an area. In some cases, when we needed a reference for a result of interest and could not resort to the aforementioned trick, we cited also less central work.

As a matter of policy, we tried to avoid credits in the main text. The few exceptions are either pointers to texts that provide details that we chose to omit or usage of terms (bearing researchers' names) that are too popular to avoid.

Teaching note: The text also includes some teaching notes, which are typeset as this one. Some of these notes express quite opinionated recommendations and/or justify various expositional choices made in the text.

1.1.4.5 Additional notes

The author's guess is that the text will be criticized for lengthy discussions of technically trivial issues. Indeed, most researchers dismiss various conceptual clarifications as being trivial and devote all their attention to the technically challenging parts of the material. The consequence is students that master the technical material but are confused about its meaning. In contrast, the author recommends not being embarrased of devoting time to conceptual clarifications, even if some students may view them as obvious.

The motivational discussions presented in the text do not necessarily represent the original motivation of the researchers that pioneered a specific study and/or contributed greatly to it. Instead, these discussions provide what the author considers to be a good motivation and/or perspective on the corresponding concepts.

1.1.5 Standard notations and other conventions

Following are some notations and conventions that are freely used in this book.

Standard asymptotic notation: When referring to integral functions, we use the standard asymptotic notation; that is, for $f, g : \mathbb{N} \to \mathbb{N}$, we write $f = O(g)$ (resp., $f = \Omega(g)$) if there exists a constant $c > 0$ such that $f(n) \le c \cdot g(n)$ (resp., $f(n) \ge c \cdot g(n)$) holds for all $n \in \mathbb{N}$. We usually denote by poly an unspecified polynomial, and write $f(n) = \text{poly}(n)$ instead of "there exists a polynomial p such that $f(n) \le p(n)$ for all $n \in \mathbb{N}$." We also use the notation $f = \widetilde{O}(g)$ that mean $f(n) = \text{poly}(\log n) \cdot g(n)$, and $f = o(g)$ (resp., $f = \omega(g)$) that mean $f(n) < c \cdot g(n)$ (resp., $f(n) > c \cdot g(n)$) for every constant $c > 0$ and all sufficiently large n.

Integrality issues: Typically, we ignore integrality issues. This means that we may assume that $\log_2 n$ is an integer rather than using a more combersome form as $\lfloor \log_2 n \rfloor$.

Standard combinatorial and graph theory terms and notation: For a natural number $n \in \mathbb{N}$, we denote $[n] \stackrel{\text{def}}{=} \{1, ..., n\}$. Many of the computational problems refer to finite (undirected) graphs. Such a graph, denoted $G = (V, E)$, consists of a set of vertices, denoted V, and a set of edges, denoted E, which are unordered pairs of vertices. By default, graphs are undirected, whereas directed graphs consists of vertices and directed edges, where a directed edge is an order pair of vertices. We also refer to other graph theoretic terms such as connectivity, being acyclic (i.e., having no simple cycles), being a tree (i.e., being connected and acyclic), k-colorability, etc. For further background on graphs and computational problems regarding graphs, the reader is referred to Section G.1.

Typographic conventions: We denote formally defined complexity classes by caligraphic letters (e.g., $\mathcal{NP}$), but we do so only after defining these classes. Furthermore, when we wish to maintain some ambiguity regarding the specific formulation of a class of problems we use Roman font (e.g., NP may denote either a class of search problems or a class of decision problems). Likewise, we denote formally defined computational problems by typewriter font (e.g., `SAT`). In contrast, generic problems and algorithms will be denoted by italic font.

1.2 Computational Tasks and Models

We start by introducing the general framework for a discussion of computational tasks (or problems), which refers to the representation of instances and to two types of tasks (i.e., searching for solutions and making decisions). Once the stage is set, we consider two types of models of computation: uniform models that correspond to the intuitive notion of an algorithm, and non-uniform models (e.g., Boolean circuits) that allow for a closer look at the way computation progresses.

The contents of Sections 1.2.1–1.2.3 corresponds to a traditional *Computability course*, and most of this material is taken for granted in the rest of the current book. In contrast, Section 1.2.4 presents basic preliminaries regarding non-uniform models of computation (i.e., various types of Boolean circuits), and these are only used lightly in the rest of the current book. Thus, whereas Sections 1.2.1–1.2.3 are absolute prerequisites for the rest of this book, Section 1.2.4 is not.

Teaching note: The author believes that there is no real need for a semester-long course in Computability (i.e., a course that focuses on what can be computed rather than on what can be computed efficiently). Instead, undergraduates should take a course in computational complexity, where the computability aspects will serve as a basis for the rest of the course. Specifically, the former aspects should occupy at most 25% of the course, and the focus should be on basic complexity (i.e., P, NP and NP-completeness) and on some more advanced material. Indeed, the current book may be used as a basis for such a course.

1.2.1 Representation

In Mathematics and related sciences, it is customary to discuss objects without specifying their representation. This is not possible in the theory of computation, where the representation of objects plays a central role. In a sense, a computation merely transforms one representation of an object to another representation of the same object. In particular, a computation designed to solve some problem merely transforms the problem instance to its solution, where the latter can be though of as a (possibly partial) representation of the instance. Indeed, the answer to any fully specified question is implicit in the question itself.

Computation refers to objects that are represented in some canonical way, where such canonical representation provides an "explicit" and "full" description of the corresponding object. We will consider only *finite* objects like sets, graphs, numbers, and functions (and keep distinguishing these types of objects although, actually, they are all equivalent).

Strings. We consider finite objects, each represented by a finite binary sequence, called a string. For a natural number n, we denote by $\{0,1\}^n$ the set of all strings of length n, hereafter referred to as n-bit strings. The set of all strings is denoted $\{0,1\}^*$; that is, $\{0,1\}^* = \cup_{n\in\mathbb{N}}\{0,1\}^n$. For $x\in\{0,1\}^*$, we denote by $|x|$ the length of x (i.e., $x\in\{0,1\}^{|x|}$), and often denote by x_i the i^{th} bit of x (i.e., $x = x_1x_2\cdots x_{|x|}$). For $x,y\in\{0,1\}^*$, we denote by xy the string resulting from concatenation of the strings x and y.

At times, we associate $\{0,1\}^*\times\{0,1\}^*$ with $\{0,1\}^*$; the reader should merely consider an adequate encoding (e.g., the pair $(x_1\cdots x_m, y_1\cdots y_n)\in\{0,1\}^*\times\{0,1\}^*$ may be encoded by the string $x_1x_1\cdots x_mx_m01y_1\cdots y_n \in \{0,1\}^*$). Likewise, we may represent sequences of strings (of fixed or varying length) as single strings. When we wish to emphasize that such a sequence (or some other object) is to be considered as a single object we use the notation $\langle\cdot\rangle$ (e.g., "the pair (x,y) is encoded as the string $\langle x,y\rangle$").

Numbers. Unless stated differently, natural numbers will be encoded by their binary expansion; that is, the string $b_{n-1}\cdots b_1b_0 \in \{0,1\}^n$ encodes the number $\sum_{i=0}^{n-1} b_i\cdot 2^i$. Rational numbers will be represented as pairs of natural numbers. In the rare cases in which one considers real numbers as part of the input to a computational problem, one actually mean rational approximations of these real numbers.

Special symbols. We denote the empty string by λ (i.e., $\lambda \in \{0,1\}^*$ and $|\lambda| = 0$), and the empty set by $\emptyset$. It will be convenient to use some special symbols that are not in $\{0,1\}^*$. One such symbol is $\perp$, which typically denotes an indication by some algorithm that something is wrong.

1.2.2 Computational Tasks

Two fundamental types of computational tasks are so-called search problems and decision problems. In both cases, the key notions are the problem's *instances* and the problem's specification.

Search problems. A search problem consists of a specification of a set of valid solutions (possibly an empty one) for each possible instance. That is, given an instance, one is required to find a corresponding solution (or to determine that no such solution exists). For example, consider the problem in which one is given a system of equations and is asked to find a valid solution. Needless to say, much of computer science is concerned with solving various search problems. Furthermore, search problems correspond to the daily notion of "solving a problem" and thus a discussion of the possibility and complexity of solving search problems corresponds to the natural concerns of most people. In the following definition of solving search problems, the potential solver is a function (which may be thought of as a solving strategy), and the sets of possible solutions associated with each of the various instances are "packed" into a single binary relation.

Definition 1.1 (solving a search problem) *Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$. A function $f : \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$* solves the search problem of R *if for every x it holds that $(x, f(x)) \in R$ if and only if $R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\}$ is not empty.*

Indeed, $R(x)$ denotes the set of valid solutions for the problem instance x, and it is required that whenever there exist valid solutions (i.e., $R(x)$ is not empty) the solver finds one. It is also required that the solver f never outputs a wrong solution (i.e., if $R(x) \neq \emptyset$ then $f(x) \in R(x)$), and it follows that if $R(x) = \emptyset$ then $f(x) = \perp$, which in turn means that f indicates that x has no solution. A special case of interest is the case that $|R(x)| = 1$ for every x, where R is essentially a (total) function, and solving the search problem of R means computing (or evaluating) the function R (or rather the function R' defined defined by $R'(x) \stackrel{\text{def}}{=} y$ where $R(x) = \{y\}$).

Decision problems. A decision problem consists of a specification of a subset of the possible instances. Given an instance, one is required to determine whether the instance is in the specified set. For example, consider the problem where one is given a natural number, and is asked to determine whether or not the number is a prime. One important case, which corresponds to the aforementioned search problems, is the case of the set of instances having a solution; indeed, being able to determine whether or not a solution exists is a prerequisite to being able to solve the corresponding search problem (as per Definition 1.1). In general, decision problems refer to the natural task of making binary decision, a task that is not uncommon in daily life. In any case, in the following definition of solving search problems, the potential solver is again a function (i.e., in this case it is a Boolean function that is supposed to indicate membership in the said set).

Definition 1.2 (solving a decision problem) *Let* $S \subseteq \{0,1\}^*$. *A function* $f : \{0,1\}^* \to \{0,1\}$ solves the decision problem of S (or decides membership in S) *if for every* x *it holds that* $f(x) = 1$ *if and only if* $x \in S$.

Indeed, if f solves the search problem of R then the Boolean function $f' : \{0,1\}^* \to \{0,1\}$ defined by $f'(x) \stackrel{\text{def}}{=} 1$ if and only if $f(x) \neq \perp$ solves the decision problem of $S \stackrel{\text{def}}{=} \{x : R(x) \neq \emptyset\}$. We often identify the decision problem of S with S itself, and identify S with its characteristic function (i.e., with $\chi_S : \{0,1\}^* \to \{0,1\}$ defined such that $\chi_(x) = 1$ if and only if $x \in S$).

Most people would consider search problems to be more natural than decision problems: typically, people seeks solutions more than they stop to wonder whether or not solutions exist. Definitely, search problems are not less important than decision problems, it is merely that their study tends to require more cumbersome formulations. This is the main reason that most expositions choose to focus on decision problems. The current book attempts to devote at least a significant amount of attention also to search problems.

Promise problems (an advanced comment). Many natural search and decision problems are captured more naturally by the terminology of promise problems, where the domain of possible instances is a subset of $\{0,1\}^*$ rather than $\{0,1\}^*$ itself. In particular, note that the natural formulation of many search and decision problems refers to instances of a certain types (e.g., a system of equations, a pair of numbers, a graph), whereas the natural representation of these objects uses only a strict subset of $\{0,1\}^*$. A nasty convention is to postulate that every string represents some legitimate object (i.e., each string that is not used in the natural representation of these objects is postulated as a representation of some fixed object). For the time being, we will ignore this issue, but we will re-visit it in Section 2.4.1.

1.2.3 Uniform Models (Algorithms)

We are all familiar with computers, and the ability of computer programs to manipulate data. But how does one capture *all* computational processes? Before being formal, we offer a loose description, capturing many artificial as well as natural processes, whereas the former are associated with computers and the latter are used to model (aspects of) the natural reality (be it physical, biological, or even social).

A *computation* is a process that modifies an environment via repeated applications of a predetermined rule. The key restriction is that this rule is *simple*: in each application it depends and affects only a (small) portion of the environment, called the active zone. We contrast the *a-priori bounded* size of the active zone (and of the modification rule) with the *a-priori unbounded* size of the entire environment. We note that, although each application of the rule has a very limited effect, the effect of many applications of the rule may be very complex. Put in other words, a computation may modify the relevant environment in a very complex way, although it is merely a process of repeatedly applying a simple rule.

As hinted, the notion of computation can be used to model some aspects of the natural reality. In this case, the process that takes place in the natural reality is the starting point of the study, and the goal of the study is to learn the (computation) rule that underlies this natural process. In a sense, the goal of Science at large can be phrased as learning (simple) rules that govern various aspects of reality (or rather one's abstraction of these aspects of reality).

Our focus, however, is on artificial computation rules designed by humans in order to achieve specific desired effects on the corresponding artificial environment. Thus, our starting point is a desired functionality, and our aim is to design computation rules that effect it. Such a computation rule is referred to as an algorithm. Loosely speaking, an algorithm corresponds to a computer program written in a high-level (abstract) programming language. Let us elaborate.

We are interested in the transformation of the environment affected by the computational process (or the algorithm). Throughout (most of) this book, we will assume that, when invoked on any finite initial environment, the computation halts after a finite number of steps. Typically, the initial environment to which the computation is applied encodes an input string, and the end environment (i.e., at termination of the computation) encodes an output string. We consider the mapping from inputs to outputs induced by the computation; that is, for each possible input x, we consider the output y obtained at the end of a computation initiated with input x, and say that the computation maps input x to output y. Thus, a computation rule (or an algorithm) determines a function (computed by it): this function is exactly the aforementioned mapping of inputs to outputs.

In the rest of this book (i.e., outside the current chapter), we will also consider the number of steps (i.e., applications of the rule) taken by the computation for each possible input. The latter function is called the time complexity of the computational process (or algorithm). While time complexity is defined per input, we will often considers it per input length, taking the maximum over all inputs of the same length.

In order to define computation (and computation time) rigorously, one needs to specify some model of computation; that is, provide a concrete definition of environments and a class of rules that may be applied to them. Such a model corresponds to an abstraction of a real computer (be it a PC, mainframe or network of computers). One simple abstract model that is commonly used is that of *Turing machines* (see, §1.2.3.1 below). Thus, specific algorithms are typically formalized by corresponding Turing machines (and their time complexity is represented by the time complexity of the corresponding Turing machines). We stress, however, that most results in the Theory of Computation hold regardless of the specific computational model used, as long as it is "reasonable" (i.e., satisfies the aforementioned simplicity condition and can perform some obviously simple computations).

What is being Computed? The above discussion has implicitly referred to computations and algorithms as means of computing functions. Specifically, an algorithm A computes the function $f_A : \{0,1\}^* \to \{0,1\}^*$ defined by $f_A(x) = y$ if, when invoked on input x, algorithm A halts with output y. However, computations can also be viewed as a means of "solving search problems" or "making decisions" (as in Definitions 1.1 and 1.2). Specifically, we will say that algorithm A solves the search problem of R (resp., decides membership in S) if f_A solves the search problem of R (resp., decides membership in S). In the rest of this exposition we associate the algorithm A with the function f_A computed by it; that is, we write $A(x)$ instead of $f_A(x)$. For sake of future reference, we summarize the foregoing discussion.

Definition 1.3 (solution by an algorithm) *We denote by $A(x)$ the output of algorithm A on input x. Algorithm A* solves the search problem *R* (resp., the decision problem S) *if A, viewed as a function, solves R* (resp., S).

Organization of the rest of this section: In §1.2.3.1 we provide a sketchy description of the model of Turing machines. This is done merely for sake of providing a concrete model that supports the study of computation and its complexity, whereas most of the material in this book will not depend on the specifics of this model. In §1.2.3.2 and §1.2.3.2 we discuss two fundamental properties of any reasonable model of computation: the existence of uncomputable functions and the existence of universal computations. The time (and space) complexity of computation is defined in §1.2.3.4. We also discuss oracle machines and restricted models of computation (in §1.2.3.5 and §1.2.3.6, respectively).

1.2.3.1 Turing machines

The model of Turing machines offer a relatively simple formulation of the notion of an algorithm. The fact that the model is very simple complicates the design of machines that solve desired problems, but it makes the analysis of such machines simpler. Since the focus of complexity theory is on the analysis of machines and not on their design, the choice of this model and the trade-off that it offers is a good one. We stress again that the model is merely used as a concrete formulation of the intuitive notion of an algorithm, whereas we actually care about the intuitive notion and not its formulation. In particular, all results mentioned in this book hold for any other "reasonable" formulation of the notion of an algorithm.

The model of Turing machines is not supposed to provide a good (or "tight") model of real-life computers (although a task can be solved by a real-life computer if and only if it can be solved by a Turing machine). Historically, the model of Turing machines was invented before modern computers were even built, and was meant to provide a concrete model of computation (as opposed to the abstract definition of "recursive functions" that defines a class of "computable" functions in terms of composition of such functions). Indeed, this concrete model clarified fundamental properties of computable functions and plays a key role in defining the complexity of computable functions.

The model of Turing machines was envisioned as an abstraction of the process of an algebraic computation carried out by a human using a sheet of paper. In such a process, at each time, the human looks at some location on the paper, and depending on what he/she sees and what he/she has in mind (which is little...), he/she modifies the contents of this location and moves its look to an adjacent location.

The actual model. Following is a high-level description of the model of Turing machines; the interested reader is referred to standard textbooks (e.g., [194]) for further details. Recall that we need to specify the set of possible environments, the set of machines (or computation rules), and the effect of applying such a rule on an environment.

- The main component in the environment of a Turing machine is an infinite sequence of cells, each capable of holding a single symbol (i.e., member of a finite set $\Sigma \supset \{0,1\}$). In addition, he environment contains the current location of the machine on this sequence, and the internal state of the machine (which is a member of a finite set Q). The aforementioned sequence of cells is called the tape, and its contents combined with the machine's location and internal state is called the instantaneous configuration of the machine.

- The Turing machine itself consists of a finite rule (i.e., a finite function), called the transition function, which is defined over the set of all possible symbol-state pairs. Specifically, the transition function is a mapping from $\Sigma \times Q$ to $\Sigma \times Q \times \{-1, 0, +1\}$, where $\{-1, +1, 0\}$ correspond to a movement instruction (which is either "left" or "right" or "stay", respectively). In addition, the machine's description specifies an initial state and a halting state, and the computation of the machine halts when the machine enters its halting state.[6]

  In contrast to the finite description of the machine, the tape has an a priori unbounded length (and is considered, for simplicity, as being infinite).

- A single computation step of such a Turing machine depends on its current location on the tape, on the contents of the corresponding cell and on the internal state of the machine. Based on the latter two elements, the transition function determines a new symbol-state pair as well as a movement instruction (i.e., "left" or "right" or "stay"). The machine modifies the contents of the said cell and its internal state accordingly, and moves as directed. That is, suppose that the machine is in state q and resides in a cell containing the symbol σ, and suppose that the transition function maps (σ, q) to (σ', q', D).

[6] Envisioning the tape as extending from left to right, we also use the convention by which if the machine tries to move left of the end of the tape then it is considered to have halted.

Then, the machine modifies the contents of the said cell to σ', modifies its internal state to q', and moves one cell in direction D. Figure 1.1 shows a single step of a Turing machine that, when in state 'b' and seeing a binary symbol σ, replaces σ with the symbol $\sigma + 2$, maintains its internal state, and moves one position to the right.[7]

Formally, we define the successive configuration function that maps each instantaneous configuration to the one resulting by letting the machine take a single step. This function modifies only the contents of one cell (i.e. at which the machine resides), the internal state of the machine and its location, as described above.

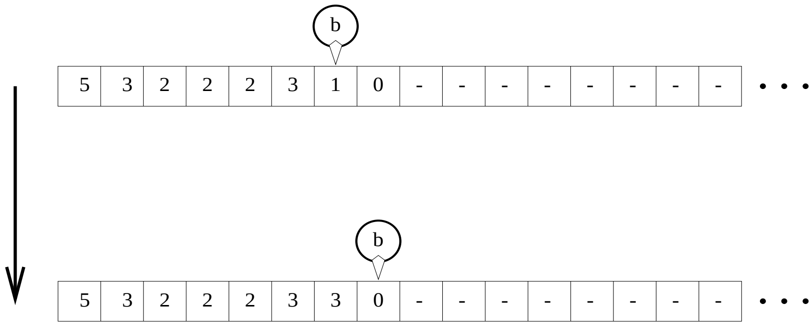


Figure 1.1: A single step by a Turing machine.

The initial environment (or configuration) of a Turing machine consists of the machine residing in the first (i.e., left-most) cell and being in its initial state. Typically, one also mandates that, in the initial configuration, a prefix of the tape's cells hold bit values, which concatenated together are considered the input, and the rest of the tape's cells hold a special symbol (which in Figure 1.1 is denoted by '-'). Once the machine halts, the output is defined as the contents of the cells that are to the left of its location (at termination time).[8] Thus, each machine defines a function mapping inputs to outputs, called the function computed by the machine.

Multi-tape Turing machines. We comment that in most expositions, one refers to the location of the "head of the machine" on the tape (rather than to the "location of the machine on the tape"). The standard terminology is more intuitive when extending the basic model, which refers to a single tape, to a model that supports a constant number of tapes. In the model of multi-tape machines, each step of the machine depends and effects the cells that are at the head location

[7] Figure 1.1 corresponds to a machine that, when in the initial state (i.e., 'a'), replaces the symbol σ by $\sigma + 4$, modifies its internal state to 'b', and moves one position to the right. Indeed, "marking" the leftmost cell (in order to allow for recognizing it in the future), is a common practice in the design of Turing machines.

[8] By an alternative convention, the machine halts while residing in the left-most cell, and the output is defined as the maximal prefix of the tape contents that contains only bit values.

of the machine on each tape. As we shall see in Chapter 5 (and in §1.2.3.4), the extension of the model to multi-tape Turing machines is crucial to the definition of space complexity. A less fundamental advantage of the model of multi-tape Turing machines is that it allows for an easier design of machines that compute desired functions.

Teaching note: We strongly recommend avoiding the standard practice of teaching the student to program with Turing machines. These exercises seem very painful and pointless. Instead, one should prove that a function can be computed by a Turing machine if and only if it is computable by a model that is closer to a real-life computer (see "sanity check" below). For starters, one should prove that a function can be computed by a single-tape Turing machine if and only if it is computable by a multi-tape (e.g., two-tape) Turing machine.

The Church-Turing Thesis: The entire point of the model of Turing machines is its simplicity. That is, in comparison to more "realistic" models of computation, it is simpler to formulate the model of Turing machines and to analyze machines in this model. The Church-Turing Thesis asserts that nothing is lost by considering the Turing machine model: *A function can be computed by some Turing machine if and only if it can be computed by some machine of any other* "reasonable and general" *model of computation*.

This is a thesis, rather than a theorem, because it refers to an intuitive notion that is left undefined on purpose (i.e., the notion of a *reasonable and general model of computation*). The model should be reasonable in the sense that it should refer to computation rules that are "simple" in some intuitive sense. On the other hand, the model should allow to compute functions that intuitively seem computable. At the very least the model should allow to emulate Turing machines (i.e., compute the function that given a description of a Turing machine and an instantaneous configuration returns the successive configuration).

A philosophical comment. The fact that a thesis is used to link an intuitive concept to a formal definition is common practice in any science (or, more broadly, in any attempt to reason rigorously about intuitive concepts). The moment an intuition is rigorously defined, it stops being an intuition, and becomes a definition and the question of the correspondence between the original intuition and the derived definition arises. This question can never be rigorously treated, because it relates to two objects, one being undefined. Thus, the question of correspondence between the intuition and the definition always transcends a rigorous treatment (i.e., it is always at the domain of the intuition).

A sanity check: Turing machines can emulate an abstract RAM. To gain confidence in the Church-Turing Thesis, one may attempt to define an abstract Random-Access Machine (RAM), and verify that it can be emulated by a Turing machine. An abstract RAM consists of an infinite number of memory cells, each capable of holding an integer, a finite number of similar registers, one designated as program counter, and a program consisting of instructions selected from a finite set. The set of possible instructions includes the following instructions:

- `reset`(r), where r is an index of a register, results in setting the value of register r to zero.
- `inc`(r), where r is an index of a register, results in incrementing the content of register r. Similarly `dec`(r) causes a decrement.
- `load`(r_1, r_2), where r_1 and r_2 are indices of registers, results in loading to register r_1 the contents of the memory location m, where m is the current contents of register r_2.
- `store`(r_1, r_2), stores the contents of register r_1 in the memory, analogously to `load`.
- `cond-goto`(r, ℓ), where r is an index of a register and ℓ does not exceed the program length, results in setting the program counter to $\ell - 1$ if the content of register r is non-negative.

The program counter is incremented after the execution of each instruction, and the next instruction to be executed by the machine is the one to which the program counter points (and the machine halts if the program counter exceeds the program's length). The input to the machine may be defined as the contents of the first n memory cells, where n is placed in a special input register. We note that the RAM model satisfies the Church-Turing Thesis, but in order to make it closer to a real-life computer we augment the model by instructions like `add`(r_1, r_2) (resp., `mult`(r_1, r_2)), which results in adding (resp., multiplying) the contents of registers r_1 and r_2 and placing the result in register r_1. We suggest proving that this abstract RAM can be emulated by a Turing machine.[9] (Hint: note that during the emulation, we only need to hold the input, the contents of all registers, and the contents of the memory cells that were accessed during the computation.)[10]

Observe that the abstract RAM model is more cumbersome than the Turing machine model. Furthermore, the question of which instructions to allow causes a vicious cycle, which we avoided by trusting the reader to consider only the standard instructions common in any real-life computer. (In general, we should only allow instructions that correspond to "simple" operations; i.e., operations that correspond to easily computable functions...)

[9]We emphasize this direction of the equivalence of the two models, because the RAM model is introduced in order to convince the reader that Turing machines are not too weak (as a model of general computation). The fact that they are not too strong seems self-evident. Thus, it seems pointless to prove that the RAM model can emulate Turing machines. Still, note that this is indeed the case, by using the RAM's memory cells to store the contents of the cells of the Turing machine's tape.

[10]Thus, at each time, the Turning machine's tape contains a list of the RAM's memory cells that were accessed so far as well as their current contents. When we emulate a RAM instruction, we first check whether the relevant RAM cell appears on this list, and augment the list by a corresponding entry or modify this entry as needed.

1.2.3.2 Uncomputable functions

Strictly speaking, the current subsection is not necessary for the rest of this book, but we feel that it provides a useful perspective.

In contrast to what every layman would think, we know that not all functions are computable. Indeed, an important message to be communicated to the world is that *not every well-defined task can be solved* by applying a "reasonable" procedure (i.e., a procedure that has a simple description that can be applied to any instance of the problem at hand). Furthermore, not only is it the case that there exist uncomputable functions, but it is rather that "most" functions are uncomputable. In fact, only relatively few functions are computable.

Theorem 1.4 (on the scarcity of computable functions): *The set of computable functions is countable, whereas the set of all functions* (from strings to string) *has cardinality* $\aleph$.

We stress that the theorem holds for any reasonable model of computation. In fact, it only relies on the postulate that each machine in the model has a finite description (i.e., can be described by a string).

Proof: Since each computable function is computable by a machine that has a finite description, there is a 1-1 correspondence between the set of computable functions and the set of strings (which in turn is in 1-1 correspondence to the natural numbers). On the other hand, there is a 1-1 correspondence between the set of Boolean functions (i.e., functions from strings to a bit) and the set of real number in $[0,1)$. This correspondence associates each real $r \in [0,1)$ to the function $f : \mathbb{N} \to \{0,1\}$ such that $f(i)$ is the $i^{\rm th}$ bit in the binary expansion of r. ■

The Halting Problem: In contrast to the preliminary discussion, at this point we consider also machines that may not halt on some inputs. (The functions computed by such machines are partial functions that are defined only on inputs on which the machine halts.) Again, we rely on the postulate that each machine in the model has a finite description, and denote the description of machine M by $\langle M\rangle \in \{0,1\}^*$. The halting function, $\mathtt{h} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$, is defined such that $\mathtt{h}(\langle M\rangle, x) \stackrel{\rm def}{=} 1$ if and only if M halts on input x. The following result goes beyond Theorem 1.4 by pointing to an explicit function (of natural interest) that is not computable.

Theorem 1.5 (undecidability of the halting problem): *The halting function is not computable.*

The term undecidability means that the corresponding decision problem cannot be solved by an algorithm. That is, Theorem 1.5 asserts that the decision problem associated with the set $\mathtt{h}^{-1}(1) = \{(\langle M\rangle, x) : \mathtt{h}(\langle M\rangle, x) = 1\}$ is not solvable by an algorithm (i.e., there exists no algorithm that, given a pair $(\langle M\rangle, x)$, decides whether or not M halts on input x). Actually, the following proof shows that there exists no algorithm that, given $\langle M\rangle$, decides whether or not M halts on input $\langle M\rangle$.

Proof: We will show that even the restriction of $\mathtt{h}$ to its "diagonal" (i.e., the function $\mathtt{d}(\langle M\rangle) \stackrel{\rm def}{=} \mathtt{h}(\langle M\rangle,\langle M\rangle))$ is not computable. Note that the value of $\mathtt{d}(\langle M\rangle)$ refers to the question of what happens when we feed M with its own description, which is indeed a "nasty" (but legitimate) thing to do. We will actually do worse: towards the contradiction, we will consider the value of $\mathtt{d}$ when evaluated at a (machine that is related to a) machine that supposedly computes $\mathtt{d}$.

We start by considering a related function, $\mathtt{d}'$, and showing that this function is uncomputable. The function $\mathtt{d}' : \{0,1\}^* \to \{0,1\}$ is defined such that $\mathtt{d}'(\langle M\rangle) \stackrel{\rm def}{=} 1$ if and only if M halts on input $\langle M\rangle$ with output 0. (That is, $\mathtt{d}'(\langle M\rangle) = 1$ if M halts on input $\langle M\rangle$ with a specific output, and $\mathtt{d}'(\langle M\rangle) = 0$ if either M does not halt on input $\langle M\rangle$ or its output does not equal the designated value.) Now, suppose, towards the contradiction, that $\mathtt{d}'$ is computable by some machine, denoted $M_{\mathtt{d}'}$. Note that machine $M_{\mathtt{d}'}$ is supposed to halt on every input, and so $M_{\mathtt{d}'}$ halts on input $\langle M_{\mathtt{d}'}\rangle$. But, by definition of $\mathtt{d}'$, it holds that $\mathtt{d}'(\langle M_{\mathtt{d}'}\rangle) = 1$ if and only if $M_{\mathtt{d}'}$ halts on input $\langle M_{\mathtt{d}'}\rangle$ with output 0 (i.e., if and only if $M_{\mathtt{d}'}(\langle M_{\mathtt{d}'}\rangle) = 0$). Thus, $M_{\mathtt{d}'}(\langle M_{\mathtt{d}'}\rangle) \neq \mathtt{d}'(\langle M_{\mathtt{d}'}\rangle)$ in contradiction to the hypothesis that $M_{\mathtt{d}'}$ computes $\mathtt{d}'$.

We next prove that $\mathtt{d}$ is uncomputable, and thus $\mathtt{h}$ is uncomputable (because $\mathtt{d}(z) = \mathtt{h}(z,z)$ for every z). To prove that $\mathtt{d}$ is uncomputable, we show that if $\mathtt{d}$ is computable then so is $\mathtt{d}'$ (which we already know not to be the case). Let A be an algorithm for computing $\mathtt{d}$ (i.e., $A(\langle M\rangle) = d(\langle M\rangle)$ for every machine M). Then we construct an algorithm for computing $\mathtt{d}'$, which given $\langle M'\rangle$, invokes A on $\langle M''\rangle$, where M'' is defined to operate as follows:

1. On input x, machine M'' emulates M' on input x.
2. If M' halts on input x with output 0 then M'' halts.
3. If M' halts on input x with an output different from 0 then M'' enters an infinite loop (and thus does not halt).
4. Otherwise (i.e., M' does not halt on input x), then machine M'' does not halt (because it just stays stuck in Step 1 forever).

Note that the mapping from $\langle M'\rangle$ to $\langle M''\rangle$ is easily computable (by augmenting M' with instructions to test its output and enter an infinite loop if necessary), and that $\mathtt{d}(\langle M''\rangle) = \mathtt{d}'(\langle M'\rangle)$, because M'' halts on x if and only if M'' halts on x with output 0. We thus derived an algorithm for computing $\mathtt{d}'$ (i.e., transform the input $\langle M'\rangle$ into $\langle M''\rangle$ and output $A(\langle M''\rangle))$, which contradicts the already established fact by which $\mathtt{d}'$ is uncomputable. ■

Turing-reductions. The core of the second part of the proof of Theorem 1.5 is an algorithm that solves one problem (i.e., computes $\mathtt{d}'$) by using as a subroutine an algorithm that solves another problem (i.e., computes $\mathtt{h}$). In fact, the first algorithm is actually an algorithmic scheme that refers to a "functionally specified" subroutine rather than to an actual (implementation of such a) subroutine, which may not exist. Such an algorithmic scheme is called a Turing-reduction (i.e., we have Turing-reduced the computation of $\mathtt{d}'$ to the computation of $\mathtt{d}$, which in turn Turing-reduces to $\mathtt{h}$). The "natural" ("positive") meaning of a Turing-reduction of

f' to f is that when given an algorithm for computing f we obtain an algorithm for computing f'. In contrast, the proof of Theorem 1.5 uses the "unnatural" ("negative") counter-positive: if (as we know) there exists no algorithm for computing $f' = \mathtt{d}'$ then there exists no algorithm for computing $f = \mathtt{h}$ (which is what we wanted to prove). Jumping ahead, we mention that resource-bounded Turing-reductions (e.g., polynomial-time reductions) play a central role in complexity theory itself, and again they are used mostly in a "negative" way. We will define such reductions and extensively use them in subsequent chapters.

Rice's Theorem. The undecidability of the halting problem (or rather the fact that the function $\mathtt{d}$ is uncomputable) is a special case of a more general phenomenon: Every non-trivial decision problem *regarding the function computed by a given Turing machine* has no algorithmic solution. We state this fact next, clarifying what is the aforementioned class of problems. (Again, we refer to Turing machines that may not halt on all inputs.)

Theorem 1.6 (Rice's Theorem): *Let $\mathcal{F}$ be a non-trivial subset*[11] *of the set of all computable partial functions, and let $S_{\mathcal{F}}$ be the set of strings that describe machines that compute functions in $\mathcal{F}$. Then deciding membership in $S_{\mathcal{F}}$ cannot be solved by an algorithm.*

Theorem 1.6 can be proved by a Turing-reduction from $\mathtt{d}$. We do not provide a proof because this is too remote from the main subject matter of the book. We stress that Theorems 1.5 and 1.6 hold for any reasonable model of computation (referring both to the potential solvers and to the machines the description of which is given as input to these solvers). Thus, Theorem 1.6 means that *no algorithm can determine any non-trivial property of the function computed by a given computer program* (written in any programming language). For example, *no algorithm can determine whether or not a given computer program halt on each possible input.* The relevance of this assertion to the project of program verification is obvious.

The Post Correspondence Problem. We mention that undecidability arises also outside of the domain of questions regarding computing devices (given as input). Specifically, we consider the Post Correspondence Problem in which the input consists of two (equal length) sequences of strings, $(\alpha_1, ..., \alpha_k)$ and $(\beta_1, ..., \beta_k)$, and the question is whether or not there exists a sequence of indices $i_1, ..., i_\ell \in \{1, ..., k\}$ such that $\alpha_{i_1} \cdots \alpha_{i_\ell} = \beta_{i_1} \cdots \beta_{i_\ell}$. (We stress that the length of this sequence is not bounded.)[12]

Theorem 1.7 *The Post Correspondence Problem is undecidable.*

Again, the omitted proof is by a Turing-reduction from $\mathtt{d}$ (or $\mathtt{h}$).

---

[11] The set S is called a non-trivial subset of U if both S and $U \setminus S$ are non-empty. Clearly, if $\mathcal{F}$ is a trivial set of computable functions then the corresponding decision problem can be solved by a "trivial" algorithm that outputs the corresponding constant bit.

[12] In contrast, the existence of an adequate sequence of a specified length can be determined in time that is exponential in this length.

1.2.3.3 Universal algorithms

So far we have used the postulate that, in any reasonable model of computation, each machine (or computation rule) has a finite description. Furthermore, we also used the fact that such model should allow for the easy modification of such descriptions such that the resulting machine computes an easily related function (see the proof of Theorem 1.5). Here we go one step further and postulate that the description of machines (in this model) is "effective" in the following natural sense: there exists an algorithm that, given a description of a machine (resp., computation rule) and a corresponding environment, determines the environment that results from performing a single step of this machine on this environment (resp. the effect of a single application of the computation rule). This algorithm can, in turn, be implemented in the said model of computation (assuming this model is general; see the Church-Turing Thesis). Successive applications of this algorithm leads to the notion of a universal machine, which (for concreteness) is formulated below in terms of Turing machines.

Definition 1.8 (universal machines): *A* universal Turing machine *is a Turing machine that on input a description of a machine M and an input x returns the value of $M(x)$ if M halts on x and otherwise does not halt.*

That is, a universal Turing machine computes the partial function $\mathtt{u}$ that is defined over pairs $(\langle M\rangle, x)$ such that M halts on input x, in which case it holds that $\mathtt{u}(\langle M\rangle, x) = M(x)$. We note that if M halts on all possible inputs then $\mathtt{u}(\langle M\rangle, x)$ is defined for every x. We stress that the mere fact that we have defined something does not mean that it exists. But, as hinted above and obvious to anyone who has written a computer program (and thought about what he/she was doing), universal Turing machines do exist.

Theorem 1.9 *There exists a universal Turing machine.*

Theorem 1.9 asserts that the partial function $\mathtt{u}$ is computable. In contrast, it can be shown that any extension of $\mathtt{u}$ to a total function is uncomputable. That is, for any total function $\hat{\mathtt{u}}$ that agrees with the partial function $\mathtt{u}$ on all the inputs on which the latter is defined, it holds that $\hat{\mathtt{u}}$ is uncomputable.[13]

Proof: Given a pair $(\langle M\rangle, x)$, we just emulate the computation of machine M on input x. This emulation is straightforward, because by the effectiveness of the description of M, we can iteratively determine the next instantaneous configuration of the computation of M on input x. If the said computation halts then we will obtain its output and can output it (and so, on input $(\langle M\rangle, x)$, our algorithm

[13] The claim is easy to prove for the total function $\hat{\mathtt{u}}$ that extends $\mathtt{u}$ and assigns the special symbol $\perp$ to inputs on which $\mathtt{u}$ is undefined (i.e., $\hat{\mathtt{u}}(\langle M\rangle, x) \stackrel{\text{def}}{=} \perp$ if $\mathtt{u}$ is not defined on $(\langle M\rangle, x)$ and $\hat{\mathtt{u}}(\langle M\rangle, x) \stackrel{\text{def}}{=} \mathtt{u}(\langle M\rangle, x)$ otherwise). In this case $\mathtt{h}(\langle M\rangle, x) = 1$ if and only if $\hat{\mathtt{u}}(\langle M\rangle, x) \neq \perp$, and so the halting function $\mathtt{h}$ is Turing-reducible to $\hat{\mathtt{u}}$. In the general case, we may adapt the proof of Theorem 1.5 by observing that, for a machine M that halts on every input, it holds that $\hat{\mathtt{u}}(\langle M\rangle, x) = \mathtt{u}(\langle M\rangle, x)$ for every x (and in particular for $x = \langle M\rangle$).

returns $M(x)$). Otherwise, we turn out emulating an infinite computation, which means that our algorithm does not halt on input $(\langle M\rangle, x)$. Thus, the foregoing emulation procedure constitutes a universal machine (i.e., yields an algorithm for computing $\mathtt{u}$). ■

As hinted above, the existence of universal machines is the fundamental fact underlying the paradigm of general-purpose computers. Indeed, a specific Turing machine (or algorithm) is a device that solves a specific problem. A priori, solving each problem would have required building a new physical device that allows for this problem to be solved in the physical world (rather than as a thought experiment). The existence of a universal machine asserts that it is enough to build one physical device; that is, a general purpose computer. Any specific problem can then be solved by writing a corresponding program for the general purpose computer. In other words, the existence of universal machines says that software can be viewed as (part of the) input.

In addition to their practical importance, the existence of universal machines (and their variants) has important consequences in the theories of computability and computational complexity. Here we merely note that Theorem 1.9 implies that many questions about the behavior of a universal machine on certain input types are undecidable. In particular, there is no algorithm that, given $X \stackrel{\rm def}{=} (\langle M\rangle, x)$, can tell whether or not a (fixed) universal machine halts on input X. Revisiting the proof of Theorem 1.7, it follows that the Post Correspondence Problem remains undecidable even if the sequences are restricted to have a specific length (i.e., k is fixed).

A detour: Kolmogorov Complexity. The existence of universal machines, which may be viewed as universal languages for writing effective and succinct descriptions of objects, plays a central role in Kolmogorov Complexity. Loosely speaking, the latter theory is concerned with the length of (effective) descriptions of objects, and views the minimum such length as the inherent "complexity" of the object; that is, "simple" objects (or phenomena) are those having short description (resp., short explanation), whereas "complex" objects have no short description. Needless to say, these (effective) descriptions have to refer to some fixed "language" (i.e., to a fixed machine that, given a succinct description of an object, produces its explicit description). Fixing any machine M, a string x is called a description of s with respect to M if $M(x) = s$. The complexity of s with respect to M, denoted $K_M(s)$, is the length of the shortest description of s with respect to M. Certainly, we want to fix M such that every string has a description with respect to M, and furthermore that this description is not "significantly" longer than the description with respect to a different machine M'. The following theorem make it natural to use a universal machine as the "point of reference" (i.e., the aforementioned M).

Theorem 1.10 (complexity w.r.t a universal machine): *Let U be a universal machine. Then, for every machine M', there exists a constant c such that $K_U(s) \leq K_{M'}(s) + c$ for every string s.*

The theorem follows by (setting $c = O(|\langle M'\rangle|)$ and) observing that if x is a description of s with respect to M' then $(\langle M'\rangle, x)$ is a description of s with respect to U. Here it is important to use an adequate encoding of pairs of strings (e.g., the pair $(\sigma_1 \cdots \sigma_k, \tau_1 \cdots \tau_\ell)$ is encoded by the string $\sigma_1\sigma_1 \cdots \sigma_k\sigma_k 01\tau_1 \cdots \tau_\ell$). Fixing any universal machine U, we define the Kolmogorov Complexity of a string s as $K(s) \stackrel{\text{def}}{=} K_U(s)$. The reader may easily verify the following facts:

1. $K(s) \leq |s| + O(1)$, for every s.

   (Hint: apply Theorem 1.10 to the machine that computes the identity mapping.)

2. There exist infinitely many strings s such that $K(s) \ll |s|$.

   (Hint: consider $s = 1^n$. Alternatively, consider any machine M such that $|M(x)| \gg |x|$.)

3. Some strings of length n have complexity at least n. Furthermore, for every n and i,
$$|\{s \in \{0,1\}^n : K(s) \leq n - i\}| < 2^{n-i+1}$$
   (Hint: different strings must have different descriptions with respect to U.)

It can be shown that *the function K is uncomputable*. The proof is related to the paradox captured by the following "description" of a natural number: `the largest natural number that can be described by an English sentence of up-to a thousand letters`. (The paradox amounts to observing that if the above number is well-defined then so is `the integer-successor of the largest natural number that can be described by an English sentence of up-to a thousand letters`.) Needless to say, the above sentence presupposes that any sentence is a legitimate description in some adequate sense (e.g., in the sense defined above). Specifically, the above sentence presupposes that we can determine the Kolmogorov Complexity of each natural number, and furthermore effectively produce the largest number that has Kolmogorov Complexity not exceeding some threshold. Indeed, the paradox provides a proof to the fact that the latter task cannot be performed (i.e., there exists no algorithm that given t produces the lexicographically last string s such that $K(s) \leq t$, because if such an algorithm A would have existed then $K(s) \leq O(|\langle A\rangle|) + \log t$ and $K(s0) < K(s) + O(1) < t$ in contradiction to the definition of s).

1.2.3.4 Time and space complexity

Fixing a model of computation (e.g., Turing machines) and focusing on algorithms that halt on each input, we consider the number of steps (i.e., applications of the computation rule) taken by the algorithm on each possible input. The latter function is called the time complexity of the algorithm (or machine); that is, $t_A : \{0,1\}^* \to \mathbb{N}$ is called the time complexity of algorithm A if, for every x, on input x algorithm A halts after exactly $t_A(x)$ steps.

We will be mostly interested in the dependence of the time complexity on the input length, when taking the maximum over all inputs of the relevant length. That is, for t_A as above, we will consider $T_A : \mathbb{N} \to \mathbb{N}$ defined by $T_A(n) \stackrel{\rm def}{=} \max_{x\in\{0,1\}^n}\{t_A(x)\}$. Abusing terminology, we sometimes refer to T_A as the time complexity of A.

The time complexity of a problem. As stated in the preface and in the introduction, complexity theory is typically unconcerned with the (time) complexity of a specific algorithm. It is rather concerned with the (time) complexity of a problem, assuming that this problem is solvable by an algorithm. Intuitively, the time complexity of such a problem is defined as the time complexity of the fastest algorithm that solves this problem (assuming that the latter term is well-defined).[14] More generally, we will be interested in upper and lower bounds on the (time) complexity of algorithms that solve the problem. However, the complexity of a problem may depend on the specific model of computation in which algorithms that solve it are implemented. The following Cobham-Edmonds Thesis asserts that the variation (in the time complexity) is not too big, and in particular is irrelevant for much of the current focus of complexity theory (e.g., for the P-vs-NP Question).

The Cobham-Edmonds Thesis. As stated above, the time complexity of a problem may depend on the model of computation. For example, deciding membership in the set $\{xx : x \in \{0,1\}^*\}$ can be done in linear-time on a two-tape Turing machine, but requires quadratic-time on a single-tape Turing machine.[15] On the other hand, any problem that has time complexity t in the model of multi-tape Turing machines, has complexity $O(t^2)$ in the model of single-tape Turing machines. The Cobham-Edmonds Thesis asserts that the time complexities in any two "reasonable and general" models of computation are polynomially related. That is, *a problem has time complexity t in some* "reasonable and general" *model of computation if and only if it has time complexity* poly(t) *in the model of* (single-tape) *Turing machines.*

Indeed, the Cobham-Edmonds Thesis strengthens the Church-Turing Thesis. It asserts not only that the class of solvable problems is invariant as far as "reasonable and general" models of computation are concerned but also that the time complexity (of the solvable problems) in such models be polynomially related.

[14] **Advanced comment:** As we shall see in Section 4.2.2 (cf. Theorem 4.8), the naive assumption that a "fastest algorithm" for solving a problem exists is not always justified. On the other hand, the assumption is justified in some important cases (see, e.g., Theorem 2.31).

[15] Proving the latter fact is quite non-trivial. One proof is by a "reduction" from a communication complexity problem [138, Sec. 12.2]. Intuitively, a single-tape Turing machine that decides membership in the aforementioned set can be viewed as a channel of communication between the two parts of the input. Focusing our attention on inputs of the form $y0^nz0^n$, for $y,z \in \{0,1\}^n$, each time the machine passes from the first part to the second part it carries $O(1)$ bits of information (in its internal state) while making at least n steps. The proof is completed by invoking the linear lower bound on the communication complexity of the (two-argument) identity function (i.e, $\mathtt{id}(y,z) = 1$ if $y = z$ and $\mathtt{id}(y,z) = 0$ otherwise, cf. [138, Chap. 1]).

Efficient algorithms. As hinted above, much of complexity theory is concerned with efficient algorithms. The latter are defined as polynomial-time algorithms (i.e., algorithms that have a time complexity that is bounded by a polynomial in the length of the input). By the Cobham-Edmonds Thesis, the choice of a "reasonable and general" model of computation is irrelevant to the definition of this class. The association of efficient algorithms with polynomial-time computation is grounded in the following two considerations:

- *Philosophical consideration*: Intuitively, efficient algorithms are those that can be implemented within a number of steps that is a moderately growing function of the input length. To allow for reading the entire input, at least linear time complexity should be allowed, whereas exponential time (as in "exhaustive search") must be avoided. Furthermore, a good definition of the class of efficient algorithms should be closed under natural composition of algorithms (as well as be robust with respect to reasonable models of computation and with respect to simple changes in the encoding of problems' instances).

  Selecting polynomials as the set of time-bounds for efficient algorithms satisfy all the above requirements: polynomials constitute a "closed" set of moderately growing functions, where "closure" means closure under addition, multiplication and functional composition. These closure properties guarantee the closure of the class of efficient algorithm under natural composition of algorithms (as well as its robustness with respect to any reasonable and general model of computation). Furthermore, polynomial-time algorithms can conduct computations that are definitely simple (although not totally trivial), and on the other hand they do not include naturally inefficient algorithms like exhaustive search.

- *Empirical consideration*: It is clear that algorithms that are considered efficient in practice have running-time that is bounded by a small polynomial (at least on the inputs that occur in practice). The question is whether any polynomial-time algorithm can be considered efficient in an intuitive sense. The belief, which is supported by past experience, is that every natural problem that can be solved in polynomial-time also has "reasonably efficient" algorithms.

We stress that the association of efficient algorithms with polynomial-time computation is not essential to most of the notions, results and questions of complexity theory. Any other class of algorithms that supports the aforementioned closure properties and allows to conduct some simple computations but not overly complex ones gives rise to a similar theory, albeit the formulation of such a theory may be much more complicated. Specifically, all results and questions treated in this book relate the complexity of different computational tasks (rather than provide absolute assertions about the complexity of some computational tasks). These relations can be stated explicitly, by stating how any upper-bound on the time complexity of one task gets translated to an upper-bound on the time complexity of another task. Such cumbersome statements will maintain the contents of the standard statements; they will merely be much more complicated. Thus, we follow the tradition of focusing on polynomial-time computations, stressing that this focus is both natural and provides the simplest way of addressing the fundamental issues underlying the nature of efficient computation.

Universal machines, revisited. Time complexity yields an important variant of the universal function $\mathtt{u}$ (presented in §1.2.3.3). Define $\mathtt{u}'(\langle M\rangle, x, t) \stackrel{\text{def}}{=} y$ if on input x machine M halts within t steps and outputs the string y, and $\mathtt{u}'(\langle M\rangle, x, t) \stackrel{\text{def}}{=} \perp$ if on input x machine M makes more than t steps. Unlike $\mathtt{u}$, the function $\mathtt{u}'$ is computable. Furthermore, $\mathtt{u}'$ is computable by a machine U' that on input $X = (\langle M\rangle, x, t)$ halts after $\text{poly}(t)$ steps. Indeed, machine U' is a variant of a universal machine (i.e., on input X, machine U' merely emulates M for t steps rather than forever). Note that the number of steps taken by U' depends on the specific model of computation (and that some overhead is unavoidable because emulating each step of M requires reading the relevant portion of the description of M).

Space complexity. Another natural measure of the "complexity" of an algorithm (or a task) is the amount of memory consumed by the computation. We refer to the memory used for storing some intermediate results of the computation. Since much of our focus will be on using memory that is sub-linear in the input length, it is important to use a model in which one can differentiate memory used for computation from memory used for storing the initial input or the final output. In the context of Turing machines, this is done by considering multi-tape Turing machines such that the input is presented on a special read-only tape (called the input tape), the output is written on a special write-only tape (called the output tape), and intermediate results are stored on a work-tape. Thus, the input and output tapes cannot be used for storing intermediate results. The space complexity of such a machine M is defined as a function s_M such that $s_M(x)$ is the number of cells of the work-tape scanned by M on input x.

1.2.3.5 Oracle machines

The notion of Turing-reductions, which was discussed in §1.2.3.2, is captured by the definition of oracle machines. Loosely speaking, an oracle machine is a machine that is augmented such that it may pose questions to the outside. (A rigorous formulation of this notion is provided below.) We consider the case in which these questions, called queries, are answered consistently by some function $f : \{0,1\}^* \to \{0,1\}^*$, called the oracle. That is, if the machine makes a query q then the answer it obtains is $f(q)$. In such a case, we say that the oracle machine is given access to the oracle f. For an oracle machine M, a string x and a function f, we denote by $M^f(x)$ the output of M on input x when given access to the oracle f. (Re-examining the second part of the proof of Theorem 1.5, observe that we have actually described an oracle machine that computes $\mathtt{h}$ when given access to the oracle $\mathtt{d}'$.)

The notion of an oracle machine extends the notion of a standard computing device (machine), and thus a rigorous formulation of the former extends a formal model of the latter. Specifically, extending the model of Turing machines, we derive the following model of oracle Turing machines.

Definition 1.11 (using an oracle): *An* oracle machine *is a Turing machine with an additional tape, called the* oracle tape*, and two special states, called* oracle invocation *and* oracle spoke*. The* computation of the oracle machine M on input x and access to the oracle $f : \{0,1\}^* \to \{0,1\}^*$ *is defined based on the successive configuration function. For configurations with state different from* oracle invocation *the next configuration is defined as usual. Let γ be a configuration in which the machine's state is* oracle invocation *and suppose that the actual contents of the oracle tape is q (i.e., q is the contents of the maximal prefix of the tape that holds bit values).*[16] *Then, the configuration following γ is identical to γ, except that the state is* oracle spoke*, and the actual contents of the oracle tape is $f(q)$. The string q is called M's* query *and $f(q)$ is called the* oracle's reply*.*

We stress that the running time of an oracle machine is the number of steps made during its computation, and that the oracle's reply on each query is obtained in a single step.

1.2.3.6 Restricted models

We mention that restricted models of computation are often mentioned in the context of a course on computability, but they will play no role in the current book. One such model is the model of finite automata, which in some variant coincides with Turing machines that have space complexity zero.

In our opinion, the most important motivation for the study of these restricted models of computation is that they provide simple models for some natural (or artificial) phenomena. This motivation, however, seems only remotely related to the study of the complexity of various computational tasks. Thus, in our opinion, the study of these restricted models (e.g., any of the lower levels of Chomsky's Hierarchy [116, Chap. 9]) should be decoupled from the study of computability theory (let alone the study of complexity theory).

1.2.4 Non-uniform Models (Circuits and Advice)

By a non-uniform model of computation we mean a model in which for each possible input length one considers a different computing device. That is, there is no "uniformity" requirement relating devices that correspond to different input lengths. Furthermore, this collection of devices is infinite by nature, and (in absence of a uniformity requirement) this collection may not even have a finite description. Nevertheless, each device in the collection has a finite description. In fact, the

[16] A common convention is that the oracle can be invoked only when the machine's head resides at the left-most cell of the oracle tape. We comment that, in the context of space complexity, one uses two oracle tapes: a write-only tape for the query and a read-only tape for the answer.

relationship between the size of the device (resp., the length of its description) and the length of the input that it handles will be of major concern. The hope is that the finiteness of all parameters (which refer to a single device in such a collection) will allow for the application of combinatorial techniques to analyze the limitations of certain settings of parameters.

In complexity theory, non-uniform models of computation are studied either towards the development of lower-bound techniques or as simplified upper-bounds on the ability of efficient algorithms. In both cases, the uniformity condition is eliminated in the interest of simplicity and with the hope (and belief) that nothing substantial is lost as far as the questions in focus are concerned.

We will focus on two related models of non-uniform computing devices: Boolean circuits (§1.2.4.1) and "machines that take advice" (§1.2.4.2). The former model is more adequate for the study of the evolution of computation (i.e., development of lower-bound techniques), whereas the latter is more adequate for modeling purposes (e.g., upper-bounding the ability of efficient algorithms).

1.2.4.1 Boolean Circuits

The most popular model of non-uniform computation is the one of Boolean circuits. Historically, this model was introduced for the purpose of describing the "logic operation" of real-life electronic circuits. Ironically, nowadays this model provides the stage for some of the most practically removed studies in complexity theory (which aim at developing methods that may eventually lead to an understanding of the inherent limitations of efficient algorithms).

A Boolean circuit is a directed acyclic graph *with labels on the vertices*, to be discussed shortly. For sake of simplicity, we disallow isolated vertices (i.e., vertices with no in-going or out-going edges), and thus the graph vertices are of three types: *sources*, *sinks*, and *internal vertices*.

1. Internal vertices are vertices having in-coming and out-going edges (i.e., they have in-degree and out-degree at least 1). In the context of Boolean circuits, internal vertices are called gates. Each gate is labeled by a Boolean operation, where the operations typically considered are $\wedge$, $\vee$ and $\neg$ (corresponding to `and`, `or` and `neg`). In addition, we require that gates labeled $\neg$ have in-degree 1. (The in-coming degree of $\wedge$-gates and $\vee$-gates may be any number greater than zero, and the same holds for the out-degree of any gate.)

2. The graph sources (i.e., vertices with no in-going edges) are called input terminals. Each input terminal is labeled by a natural number (which is to be thought of the index of an input variable). (For sake of defining formulae, we allow different input terminals to be labeled by the same number.)[17]

3. The graph sinks (i.e., vertices with no out-going edges) are called output terminals, and we require that they have in-degree 1. Each output terminal is

[17]This is not needed in case of general circuits, because we can just feed out-going edges of the same input terminal to many gates. Note, however, that this is not allowed in case of formulae, where all non-sinks are required to have out-degree 1.

labeled by a natural number such that if the circuit has m output terminals then they are labeled $1, 2, ..., m$. That is, we disallow different output terminals to be labeled by the same number, and insist that the labels of the output terminals are consecutive numbers. (Indeed, the labels of the output terminals will correspond to the indices of locations in the circuit's output.)

For sake of simplicity, we also mandate that the labels of the input terminals are consecutive numbers.[18]

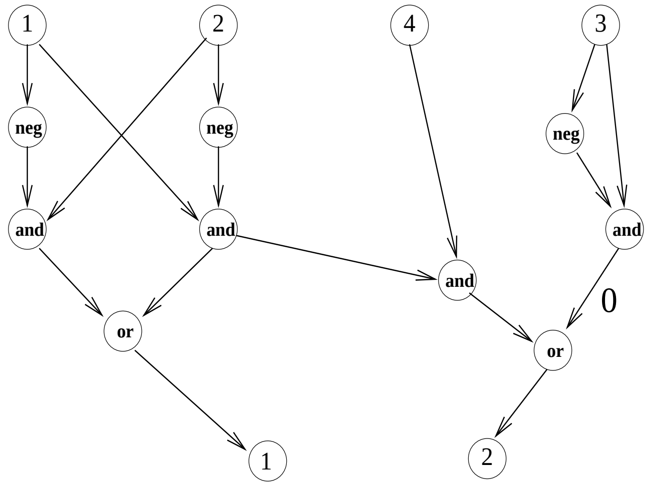


Figure 1.2: A circuit computing $f(x_1, x_2, x_3, x_4) = (x_1 \oplus x_2, x_1 \wedge \neg x_2 \wedge x_4)$.

A Boolean circuit with n different input labels and m output terminals induces (and indeed computes) a function from $\{0,1\}^n$ to $\{0,1\}^m$ defined as follows. For any fixed string $x \in \{0,1\}^n$, we iteratively define the value of vertices in the circuit such that the input terminals are assigned the corresponding bits in $x = x_1 \cdots x_n$ and the values of other vertices are determined in the natural manner. That is:

- An input terminal with label $i \in \{1, ..., n\}$ is assigned the i^{th} bit of x (i.e., the value x_i).
- If the children of a gate (of in-degree d) labeled $\wedge$ have values $v_1, v_2, ..., v_d$ then the gate is assigned the value $\wedge_{i=1}^{d} v_i$. The value of a gate labeled $\vee$ (or $\neg$) is determined analogously.

  Indeed, the hypothesis that the circuit is acyclic implies that the process of determining values for the circuit's vertices is well-defined: As long as the

[18] This convention slightly complicates the construction of circuits that ignore some of the input values. Specifically, we use artificial gadgets that have in-coming edges from the corresponding input terminals, and compute an adequate constant. To avoid having this constant as an output terminal, we feed it into an auxiliary gate such that the value of the latter is determined by the other in-going edge (e.g., a constant 1 fed into an $\vee$-gate). See example of dealing with x_3 in Figure 1.2.

value of some vertex is undetermined, there exists a vertex such that its value is undetermined but the values of all its children are determined. Thus, the process can make progress, and terminates when the values of all vertices (including the output terminals) are determined.

The value of the circuit on input x (i.e., the output computed by the circuit on input x) is $y = y_1 \cdots y_m$, where y_i is the value assigned by the above process to the output terminal labeled i. We note that *there exists a polynomial-time algorithm that, given a circuit C and a corresponding input x, outputs the value of C on input x*. This algorithm determines the values of the circuit's vertices, going from the circuit's input terminals to its output terminals.

We say that a family of circuits $(C_n)_{n\in\mathbb{N}}$ computes a function $f : \{0,1\}^* \to \{0,1\}^*$ if for every n the circuit C_n computes the restriction of f to strings of length n. In other words, for every $x \in \{0,1\}^*$, it must hold that $C_{|x|}(x) = f(x)$.

Bounded and unbounded fan-in. We will be most interested in circuits in which each gate has at most two in-coming edges. In this case, the types of (two-argument) Boolean operations that we allow is immaterial (as long as we consider a "full basis" of such operations; i.e., a set of operations that can implement any other two-argument Boolean operation). Such circuits are called circuits of bounded fan-in. In contrast, other studies are concerned with circuits of unbounded fan-in, where each gate may have an arbitrary number of in-going edges. Needless to say, in the case of circuits of unbounded fan-in, the choice of allowed Boolean operations is important and one focuses on operations that are "uniform" (across the number of operants; e.g., $\wedge$ and $\vee$).

Circuit size as a complexity measure. The size of a circuit is the number of its edges. When considering a family of circuits $(C_n)_{n\in\mathbb{N}}$ that computes a function $f : \{0,1\}^* \to \{0,1\}^*$, we are interested in the size of C_n as a function of n. Specifically, we say that this family has size complexity $s : \mathbb{N} \to \mathbb{N}$ if for every n the size of C_n is $s(n)$. The circuit complexity of a function f, denoted s_f, is the infimum of the size complexity of all families of circuits that compute f. Alternatively, for each n we may consider the size of the smallest circuit that computes the restriction of f to n-bit strings (denoted f_n), and set $s_f(n)$ accordingly. We stress that non-uniformity is implicit in this definition, because no conditions are made regarding the relation between the various circuits used to compute the function on different input lengths.

The circuit complexity of functions. We highlight some simple facts about the circuit complexity of functions. (These facts are in clear correspondence to facts regarding Kolmogorov Complexity mentioned in §1.2.3.3.)

1. Most importantly, any Boolean function can be computed by some family of circuits, and thus the circuit complexity of any function is well-defined. Furthermore, each function has at most exponential circuit complexity.

(Hint: $f_n : \{0,1\}^n \to \{0,1\}$ can be computed by a circuit of size $O(n2^n)$ that implements a look-up table.)

2. Some functions have polynomial circuit complexity. In particular, any function that has time complexity t (i.e., is computed by an algorithm of time complexity t) has circuit complexity $\text{poly}(t)$. Furthermore, the corresponding circuit family is uniform (in a natural sense to be discussion below).

   (Hint: consider a Turing machine that computes the function, and consider its computation on a generic n-bit long input. The corresponding computation can be emulated by a circuit that consists of $t(n)$ layers such that each layer represents an instantaneous configuration of the machine, and the relation between consecutive configurations is captured by ("uniform") local gadgets in the circuit. For further details see the proof of Theorem 2.20, which presents a similar emulation.)

3. Almost all Boolean functions have exponential circuit complexity. Specifically, the number of functions mapping $\{0,1\}^n$ to $\{0,1\}$ that can be computed by a circuit of size s is at most s^{2s}.

   (Hint: the number of circuits having v vertices and s edges is at most $\binom{v}{2}^s$.)

Note that the first fact implies that families of circuits can compute functions that are uncomputable by algorithms. Furthermore, this phenomenon occurs also when restricting attention to families of polynomial-size circuits. See further discussion in §1.2.4.2.

Uniform families. A family of polynomial-size circuits $(C_n)_{n\in\mathbb{N}}$ is called uniform if given n one can construct the circuit C_n in $\text{poly}(n)$-time. Note that *if a function is computable by a uniform family of polynomial-size circuits then it is computable by a polynomial-time algorithm.* The algorithm first constructs the adequate circuit (which can be done in polynomial-time by the uniformity hypothesis), and then evaluate this circuit on the given input (which can be done in time that is polynomial in the size of the circuit).

Note that limitations on the computing power of arbitrary families of polynomial-size circuits certainly hold for uniform families (of polynomial-size), which in turn yield limitations on the computing power of polynomial-time algorithms. Thus, lower bounds on the circuit complexity of functions yield analogous lower bounds on their time complexity. Furthermore, as is often the case in mathematics and Science, disposing of an auxiliary condition that is not well-understood (i.e., uniformity) may turn out fruitful. Indeed, this has occured in the study of limited classes of circuits, which is reviewed in §1.2.4.3 (and Appendix B).

1.2.4.2 Machines that take advice

General (non-uniform) circuit families and uniform circuit families are two extremes with respect to the "amounts of non-uniformity" in the computing device. Intuitively, in the former, non-uniformity is only bounded by the size of the device, whereas in the latter the amounts of non-uniformity is zero. Here we consider a model that allows to decouple the size of the computing device from the amount of non-uniformity, which may range from zero to the device's size. Specifically, we consider algorithms that "take a non-uniform advice" that depends only on the input length. The amount of non-uniformity will be defined to equal the length of the corresponding advice (as a function of the input length).

Definition 1.12 (taking advice): *We say that* algorithm A computes the function f using advice of length $\ell : \mathbb{N} \to \mathbb{N}$ *if there exists an infinite sequence* $(a_n)_{n\in\mathbb{N}}$ *such that*

1. *For every* $x \in \{0,1\}^*$*, it holds that* $A(a_{|x|}, x) = f(x)$.
2. *For every* $n \in \mathbb{N}$*, it holds that* $|a_n| = \ell(n)$.

The sequence $(a_n)_{n\in\mathbb{N}}$ *is called the* advice sequence.

Note that any function having circuit complexity s can be computed using advice of length $O(s \log s)$, where the log factor is due to the fact that a graph with v vertices and e edges can be described by a string of length $2e \log_2 v$. Note that the model of machines that use advice allows for some sharper bounds than the ones stated in §1.2.4.1: every function can be computed using advice of length ℓ such that $\ell(n) = 2^n$, and some uncomputable functions can be computed using advice of length 1.

Theorem 1.13 (the power of advice): *There exist functions that can be computed using one-bit advice but cannot be computed without advice.*

Proof: Taking any uncomputable Boolean function $f : \mathbb{N} \to \{0,1\}$, consider the function f' defined as $f'(x) = f(|x|)$. Note that f is Turing-reducible to f' (e.g., on input n make any n-bit query to f', and return the answer).[19] Thus, f' cannot be computed without advice. On the other hand, f' can be easily computed by using the advice sequence $(a_n)_{n\in\mathbb{N}}$ such that $a_n = f(n)$; that is, the algorithm merely outputs the advice bit (and indeed $a_{|x|} = f(|x|) = f'(x)$, for every $x \in \{0,1\}^*$).
■

1.2.4.3 Restricted models

As noted in §1.2.4.1, the model of Boolean circuits allows for the introduction of many natural subclasses of computing devices. Following is a laconic review of a few of these subclasses. For more detail, see Appendix B.2. We will refer to various types of Boolean formulae in the rest of this book, and thus suggest not to skip the following two paragraphs.

[19] Indeed, this Turing-reduction is not efficient (i.e., it runs in exponential time in $|n| = \log_2 n$), but this is immaterial in the current context.

Boolean formulae. In general Boolean circuits the non-sink vertices are allowed arbitrary out-degree. This means that the same intermediate value can be re-used (without being re-computed (while increasing the size complexity by only one unit)). Such "free" re-usage of intermediate values is disallowed in Boolean formulae, which corresponds to a Boolean expression over Boolean variables. Formally, a Boolean formula is a circuit in which all non-sink vertices have out-degree 1, which means that the underlying graph is a tree (and the formula as an expression can be read by traversing the tree scanning the leaves in order (see Section G.2)). Indeed, we have allowed different input terminals to be assigned the same label in order to allow formulae in which the same variable occurs multiple times. As in case of general circuits, one is interested in the size of these restricted circuits (i.e., the size complexity of families of formulae computing various functions). We mention that quadratic lower bounds are known for the formula size of simple functions (e.g., `parity`), whereas these functions have linear circuit complexity.

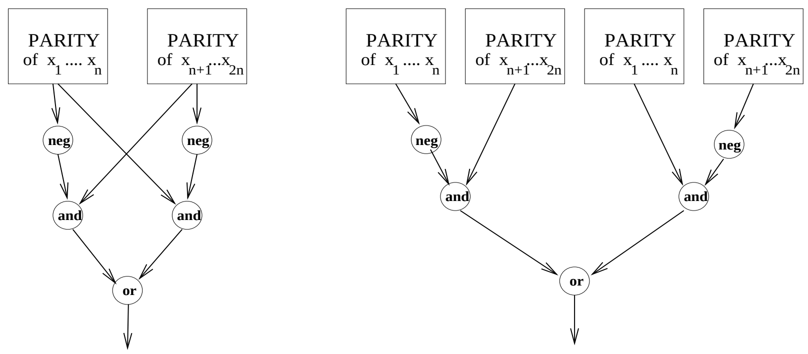


Figure 1.3: Recursive construction of parity circuits and formulae.

Formulae in CNF and DNF. A restricted type of Boolean formulae consists of formulae that are in conjunctive normal form (CNF). Such a formula consists of a conjunction of clauses, where each clause is a disjunction of literals each being either a variable or its negation. That is, such formulae are represented by layered circuits of unbounded fan-in in which the first layer consists of `neg`-gates that compute the negation of input variables, the second layer consist of `or`-gates that compute the logical-or of subsets of inputs and negated inputs, and the third layer consists of a single `and`-gate that computes the logical-and of the values computed in the second layer. Note that each Boolean function can be computed by a family of CNF formulae of exponential size, and that the size of CNF formulae may be exponentially larger than the size of ordinary formulae computing the same function (e.g., `parity`). For a constant k, a formula is said to be in k-CNF if its CNF has disjunctions of size at most k. An analogous restricted type of Boolean formulae refers to formulae that are in disjunctive normal form (DNF). Such a formula consists of a disjunction of a conjunctions of literals, and when each conjunction has at most k literals we say that the formula is in k-DNF.

Constant-depth circuits. Circuits have a "natural structure" (i.e., their structure as graphs). One natural parameter regarding this structure is the depth of a circuit, which is defined as the longest directed path from any source to any sink. Of special interest are constant-depth circuits of unbounded fan-in. We mention that sub-exponential lower bounds are known for the size of such circuits that compute a simple function (e.g., `parity`).

Monotone circuits. The circuit model also allows for the consideration of monotone computing devices: a monotone circuit is one having only monotone gates (e.g., gates computing $\wedge$ and $\vee$, but no negation gates (i.e., $\neg$-gates)). Needless to say, monotone circuits can only compute monotone functions, where a function f is called monotone if for any $x < y$ it holds that $f(x) \leq f(y)$ (where we refer to the lexicographic order on strings). One natural question is whether, as far as monotone functions are concerned, there is a substantial loss in using only monotone circuits. The answer is *yes*: there exist monotone functions that have polynomial circuit complexity but require sub-exponential size monotone circuits.

1.2.5 Complexity Classes

Complexity classes are sets of computational problems. Typically, such classes are defined by fixing three parameters:

1. A type of computational problems (see Section 1.2.2). Indeed, most classes refer to decision problems, but classes of search problems, promise problems, and other types of problems will also be considered.

2. A model of computation, which may be either uniform (see Section 1.2.3) or non-uniform (see Section 1.2.4).

3. A complexity measure and a function (or a set of functions), which put together limit the class of computations of the previous item; that is, we refer to the class of computations that have complexity not exceeding the specified function (or set of functions). For example, in §1.2.3.4, we mentioned time complexity and space complexity, which apply to any uniform model of computation. We also mentioned polynomial-time computations, which are computations in which the time complexity (as a function) does not exceed some polynomial (i.e., a member of the set of polynomial functions).

The most common complexity classes refer to decision problems, and are sometimes defined as classes of sets rather than classes of the corresponding decision problems. That is, one often says that a set $S \subseteq \{0,1\}^*$ is in the class $\mathcal{C}$ rather than saying that *the problem of deciding membership in S* is in the class $\mathcal{C}$. Likewise, one talks of classes of relations rather than classes of the corresponding search problems (i.e., saying that $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is in the class $\mathcal{C}$ means that *the search problem of R* is in the class $\mathcal{C}$).

Chapter Notes

It is quite remarkable that the theories of uniform and non-uniform computational devices have emerged in two single papers. We refer to Turing's paper [209], which introduced the model of Turing machines, and to Shannon's paper [188], which introduced Boolean circuits.

In addition to introducing the Turing machine model and arguing that it corresponds to the intuitive notion of computability, Turing's paper [209] introduces universal machines and contains proofs of undecidability (e.g., of the Halting Problem). Rice's Theorem is proven in [179], and the undecidability of the Post Correspondence Problem is proven in [168].

The formulation of machines that take advice (as well as the equivalence to the circuit model) originates in [128].

Chapter 2

P, NP and NP-Completeness

> *Forasmuch as many have taken in hand to set forth in order a declaration of those things which are most surely believed among us; Even as they delivered them unto us, who from the beginning were eyewitnesses, and ministers of the word; It seems good to me also, having had perfect understanding of all things from the very first, to write unto thee in order, most excellent Theophilus; That thou mightest know the certainty of those things, wherein thou hast been instructed.*
>
> Luke, 1:1–4

The main focus of this chapter is the P-vs-NP Question and the theory of NP-completeness. Additional topics covered in this chapter include the general notion of a polynomial-time reduction (with a special emphasis on self-reducibility), the existence of problems in NP that are neither NP-complete nor in P, the class coNP, optimal search algorithms, and promise problems.

Summary: Loosely speaking, the P-vs-NP Question refers to search problems for which the correctness of solutions can be efficiently checked (i.e., there is an efficient algorithm that given a solution to a given instance determines whether or not the solution is correct). Such search problems correspond to the class NP, and the question is whether or not all these search problems can be solved efficiently (i.e., is there an efficient algorithm that given an instance finds a correct solution). Thus, the P-vs-NP Question can be phrased as asking *whether or not finding solutions is harder than checking the correctness of solutions.*

An alternative formulation, in terms of decision problems, refers to assertions that have efficiently verifiable proofs (of relatively short length). Such sets of assertions correspond to the class NP, and the question is

whether or not proofs for such assertions can be found efficiently (i.e., is there an efficient algorithm that an assertion determines its validity and/or finds a proof for its validity). Thus, the P-vs-NP Question can be phrased as asking *whether or not discovering proofs is harder than verifying their correctness*; that is, is proving harder than verifying (i.e., are proofs valuable at all).

Indeed, it is widely believed that the answer to the two equivalent formulations is that finding (resp., discovering) is harder than checking (resp., verifying); that is, that *P is different than NP*. The fact that this natural conjecture is unsettled seems to be one of the big sources of frustration of complexity theory. The author's opinion, however, is that this feeling of frustration is out of place. In any case, at present, when faced with a hard problem in NP, we cannot expect to prove that the problem is not in P (unconditionally). The best we can expect is to prove that the said problem is not in P, assuming that NP is different from P. The contrapositive is proving that if the said problem is in P, then so is any problem in NP (i.e., NP equals P). This is where the theory of NP-completeness comes into the picture.

The theory of NP-completeness is based on the notion of a reduction, which is a relation between computational problems. Loosely speaking, one computational problem is reducible to another problem if it is possible to efficiently solve the former when provided with an (efficient) algorithm for solving the latter. Thus, the first problem is not harder to solve than the second one. A problem (in NP) is NP-complete if any problem in NP is reducible to it. Thus, the fate of the entire class NP (with respect to inclusion in P) rests with each individual NP-complete problem. In particular, showing that a problem is NP-complete implies that this problem is not in P unless NP equals P. Amazingly enough, NP-complete problems exist, and furthermore hundreds of natural computational problems arising in many different areas of mathematics and science are NP-complete.

Additional topics covered in this chapter include the existence of problems in NP that are neither NP-complete nor in P, optimal search algorithms, the class coNP, and promise problems.

Teaching note: Indeed, we suggest presenting the P-vs-NP Question both in terms of search problems and in terms of decision problems. Furthermore, in the latter case, we suggest to introduce NP while explicitly using the terminology of proof systems. As for the theory of NP-completeness, we suggest to highlight the mere existence of NP-complete sets.

Prerequisites: We assume familiarity with the notions of search and decision problems (see Section 1.2.2), algorithms (see Section 1.2.3) and their time complexity (see §1.2.3.4). We will also refer to the notion of an oracle machine (see §1.2.3.5).

Teaching note: This chapter has more teaching notes than any other chapter in this book. This reflects the author's concern regarding the way in which this fundamental material is often taught. Specifically, it is the author's impression that the material covered in this chapter is often taught in wrong ways, which fail to communicate its fundamental nature.

2.1 The P versus NP Question

Our daily experience is that it is harder to solve a problem than it is to check the correctness of a solution. Is this experience merely a coincidence or does it represent a fundamental fact of life (or property of the world)? This is the essence of the P versus NP Question, where *P represents search problems that are efficiently solvable and NP represents search problems for which solutions can be efficiently checked.* Another natural question captured by the P versus NP Question is whether proving theorems is harder that verifying the validity of these proofs. In other words, the question is whether deciding membership in a set is harder than being convinced of this membership by an adequate proof. In this case, *P represents decision problems that are efficiently solvable, whereas NP represents sets that have efficiently checkable proofs of membership.* These two meanings of the P versus NP Question are rigorously presented and discussed in Sections 2.1.1 and 2.1.2, respectively. The equivalence of the two formulations is shown in Section 2.1.3, and the common belief that P is different from NP is further discussed in Section 2.1.5.

Teaching note: Most students have heard of P and NP before, but we suspect that many have not obtained a good explanation of what the P vs NP Question actually represents. This unfortunate situation is due to using the standard technical definition of NP (which refers to the fictitious and confusing device called a non-deterministic polynomial-time machine). Instead, we advocate to use the more cumbersome definitions sketched above and elaborated below, which clearly capture the fundamental nature of NP.

The notion of efficient computation. Recall that we associate efficient computation with polynomial-time algorithms.[1] Furthermore, the latter class is independent of the specific model of computation, as long as the latter is "reasonable" (cf. the Cobham-Edmonds Thesis). Both issues are discussed in §1.2.3.4.

A note on the representation of problem instances. As noted in Section 1.2.2, many natural (search and decision) problems are captured more naturally by the terminology of promise problems (cf. Section 2.4.1), where the domain of possible instances is a subset of $\{0,1\}^*$ rather than $\{0,1\}^*$ itself. For example, computational problems in graph theory presume some simple encoding of graphs

[1] **Advanced comment:** In this chapter, we consider *deterministic* (polynomial-time) algorithms as the basic model of efficient computation. A more liberal view, which includes also *probabilistic* (polynomial-time) algorithms is presented in Chapter 6. We stress that the most important facts and questions that are addressed in the current chapter hold also with respect to probabilistic polynomial-time algorithms.

as strings, but this encoding is typically not onto (i.e., not all strings encode graphs and thus are possible instances). Typically, in these cases, the set of possible instances (e.g., encodings of graphs) is efficiently recognizable (i.e., membership in it can be decided in polynomial-time). Thus, artificially extending the set of instances to the set of all possible strings (and allowing trivial solutions for the corresponding dummy instances) does not change the complexity of the original problem. We further discuss this issue in Section 2.4.1.

2.1.1 The search version: finding versus checking

Teaching note: Complexity theorists are so accustomed to focus on decision problem that they seem to forget that search problems are at least as natural as decision problems. Furthermore, to many non-experts, search problems may seem even more natural than decision problems: Typically, people seeks solutions more than they pause to wonder whether or not solutions exist. Thus, we recommend starting with a formulation of the P-vs-NP Question in terms of search problems. Admittingly, the cost is more cumbersome formulations, but it is more than worthwhile.

Much of computer science is concerned with solving various search problems (as in Definition 1.1). Examples of such problems include finding solutions to systems of linear (or polynomial) equations, finding a spanning tree in a graph, finding a short traveling salesman tour in a metric space, and finding a scheduling of jobs to machines such that various given constraints are satisfied. Furthermore, search problems correspond to the daily notion of "solving problems" and thus are of natural general interest. In the current section, we will consider the question of *which search problems can be solved efficiently.*

One type of search problems that cannot be solved efficiently consists of search problems for which the solutions are too long in terms of the problem's instances. In such a case, merely typing the solution amounts to an activity that is deemed inefficient. Thus, we focus our attention on search problems that are not in this class. That is, we consider only search problems in which the length of the solution is bounded by a polynomial in the length of the instance. Recalling that search problems are associated with binary relations (see Definition 1.1), we focus our attention on polynomially bounded relations.

Definition 2.1 (polynomially bounded relations): *We say that $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is* polynomially-bounded *if there exists a polynomial p such that for every $(x,y) \in R$ it holds that $|y| \leq p(|x|)$.*

Recall that $(x,y) \in R$ means that y is a solution to the problem instance x, where R represents the problem itself. For such a relation R it makes sense to ask whether or not, given a problem instance x, one can efficiently find a solution y (i.e., y such that $(x,y) \in R$). The polynomial bound on the length of the solution (i.e., y) guarantees that a negative answer is not merely due to the length of the required solution.

2.1.1.1 The class P as a natural class of search problems

Recall that we are interested in the class of search problems that can be solved efficiently; that is, problems for which solutions (whenever they exist) can be found efficiently. Restricting our attention to polynomially bounded relations, we identify the corresponding fundamental class of search problem (or binary relation), denoted $\mathcal{PF}$ (standing for "Polynomial-time Find"). (The relationship between $\mathcal{PF}$ and the standard definition of P will be discussed in Sections 2.1.3 and 2.2.3.) The following definition refers to the formulation of solving search problems provided in Definition 1.1.

Definition 2.2 (efficiently solvable search problems):

- *The search problem of a polynomially bounded relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is* efficiently solvable *if there exists a polynomial time algorithm A such that, for every x, it holds that $A(x) \in R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\}$ if and only if $R(x)$ is not empty. Furthermore, if $R(x) = \emptyset$ then $A(x) = \perp$, indicating that x has no solution.*
- *We denote by $\mathcal{PF}$ the class of search problems that are efficiently solvable* (and correspond to polynomially bounded relations). *That is, $R \in \mathcal{PF}$ if R is polynomially bounded and there exists a polynomial time algorithm that given x finds y such that* $(x, y) \in R$ (or asserts that no such y exists).

Note that $R(x)$ denotes the set of valid solutions for the problem instance x. Thus, the solver A is required to find a valid solution (i.e., satisfy $A(x) \in R(x)$) whenever such a solution exists (i.e., $R(x)$ is not empty). On the other hand, if the instance x has no solution (i.e., $R(x) = \emptyset$) then clearly $A(x) \notin R(x)$. The extra condition (also made in Definition 1.1) requires that in this case $A(x) = \perp$. Thus, algorithm A always outputs a correct answer, which is a valid solution in the case that such a solution exists or an indication that no solution exists.

We have defined a fundamental class of problems, and we do know of many natural problems in this class (e.g., solving linear equations over the rationals, finding a perfect matching in a graph, etc). However, we must admit that we do not have a good understanding regarding the scope of this class (i.e., which problems it contains). This situation is quite common in complexity theory, and seems to be a consequence of the fact that complexity classes are defined in terms of the "external behavior" (of potential algorithms) rather than in terms of the "internal structure" (of the problem). Turning back to $\mathcal{PF}$, we note that while it contains many natural search problems there are also many natural search problems that are not known to be in $\mathcal{PF}$. A natural class containing a host of such problems is presented next.

2.1.1.2 The class NP as another natural class of search problems

Natural search problems have the property that valid solutions can be efficiently recognized. That is, given an instance x of the problem R and a candidate solution

y, one can efficiently determine whether or not y is a valid solution for x (with respect to the problem R; i.e., whether or not $y \in R(x)$). The class of all such search problems is a natural class *per se*, because it is not clear why one should care about a solution unless one can recognize a valid solution once given. Furthermore, this class is a natural domain of candidates for $\mathcal{PF}$, because the ability to efficiently recognize a valid solution seems to be a natural (albeit not absolute) prerequisite for a discussion regarding the complexity of finding such solutions.

We restrict our attention again to polynomially bounded relations, and consider the class of relations for which membership of pairs in the relation can be decided efficiently. We stress that we consider deciding membership of given pairs of the form (x, y) in a fixed relation R, and not deciding membership of x in the set $S_R \stackrel{\text{def}}{=} \{x : R(x) \neq \emptyset\}$. (The relationship between the following definition and the standard definition of NP will be discussed in Sections 2.1.3–2.1.4 and 2.2.3.)

Definition 2.3 (search problems with efficiently checkable solutions):

- *The search problem of a polynomially bounded relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ has* efficiently checkable solutions *if there exists a polynomial time algorithm A such that, for every x and y, it holds that $A(x, y) = 1$ if and only if $(x, y) \in R$.*
- *We denote by $\mathcal{PC}$* (standing for "Polynomial-time Check") *the class of search problems that correspond to polynomially-bounded binary relations that have efficiently checkable solutions. That is, $R \in \mathcal{PC}$ if the following two conditions hold:*
  1. *For some polynomial p, if $(x, y) \in R$ then $|y| \leq p(|x|)$.*
  2. *There exists a polynomial-time algorithm that given (x, y) determines whether or not $(x, y) \in R$.*

The class $\mathcal{PC}$ contains thousands of natural problems (e.g., finding a traveling salesman tour of length that does not exceed a given threshold, finding the prime factorization of a given composite, etc). In each of these natural problems, it is easy to see that the correctness of solutions can be checked efficiently (e.g., given a traveling salesman tour it is easy to compute its length and check whether or not it exceeds the given threshold).[2]

The class $\mathcal{PC}$ is the natural domain for the study of which problems are in $\mathcal{PF}$, because the ability to efficiently recognize a valid solution is a *natural* prerequisite for a discussion regarding the complexity of finding such solutions. We warn, however, that $\mathcal{PF}$ contains (unnatural) problems that are not in $\mathcal{PC}$ (see Exercise 2.1).

2.1.1.3 The P versus NP question in terms of search problems

Is it the case that every search problem in $\mathcal{PC}$ is in $\mathcal{PF}$? That is, if R has efficiently checkable solutions then is it necessarily the case that the search problem of R can

[2] In the traveling salesman problem (TSP), the instance is a matrix of distances between cities and a threshold, and the task is to find a tour that passes all cities and covers a total distance that does not exceed the threshold.

be solved efficiently? In other words, if it is *easy to check* whether or not a given solution for a given instance is correct then is it also *easy to find* a solution to a given instance?

If $\mathcal{PC} \subseteq \mathcal{PF}$ then this would mean that whenever solutions to given instances can be efficiently checked (for correctness) it is also the case that these solutions can be efficiently found (when given only the instance). This would mean that all reasonable search problems (i.e., all problems in $\mathcal{PC}$) are easy to solve. Needless to say, such a situation would contradict the intuitive feeling (and daily experience) that some reasonable search problems are hard to solve. Furthermore, in such a case, the notion of "solving a problem" will lose its meaning (because finding a solution will not be significantly more difficult than checking its validity).

On the other hand, if $\mathcal{PC} \setminus \mathcal{PF} \neq \emptyset$ then there exist reasonable search problems (i.e., some problems in $\mathcal{PC}$) that are hard to solve. This conforms with our basic intuition by which some reasonable problems are easy to solve whereas others are hard to solve. Furthermore, it reconfirms the intuitive gap between the notions of solving and checking (asserting that in some cases "solving" is significantly harder than "checking").

2.1.2 The decision version: proving versus verifying

As we shall see in the sequel, the study of search problems (e.g., the $\mathcal{PC}$-vs-$\mathcal{PF}$ Question) can be "reduced" to the study of decision problems. Since the latter problems have a less cumbersome terminology, complexity theory tends to focus on them (and maintains its relevance to the study of search problems via the aforementioned reduction). Thus, the study of decision problems provides a convenient way to study search problems. We wish to assert, however, that decision problems are interesting and natural *per se* (i.e., beyond their role in the study of search problems). After all, some people do care about the truth, and so determining whether a given object has some claimed property is an appealing computational problem. The P-vs-NP Question refers to the complexity of answering such questions for a wide and natural class of properties associated with the class NP. The latter class refers to properties that have "efficient proof systems" allowing for the verification of the claim that a given object has a predetermined property (i.e., is a member of a predetermined set). Jumping ahead, we mention that the P-vs-NP Question refers to the question of whether properties that have efficient proof systems can also be decided efficiently (without proofs). Let us clarify all these notions.

2.1.2.1 The class P as a natural class of decision problems

Properties of objects are modeled as subsets of the set of all possible objects (i.e., a property is associated with the set of objects having this property). Thus, we focus on deciding membership in sets (as in Definition 1.2). We consider the class of decision problems that are efficiently solvable, a class that is traditionally denoted $\mathcal{P}$ (standing for Polynomial-time). The following definition refers to the formulation of solving decision problems provided in Definition 1.2.

Definition 2.4 (efficiently solvable decision problems):

- *A decision problem $S \subseteq \{0,1\}^*$ is* efficiently solvable *if there exists a polynomial time algorithm A such that, for every x, it holds that $A(x) = 1$ if and only if $x \in S$.*
- *We denote by $\mathcal{P}$ the class of decision problems that are efficiently solvable.*

As in Definition 2.2, we have defined a fundamental class of problems, which contains many natural problems (e.g., determining whether or not a given graph is connected), but we do not have a good understanding regarding its scope (i.e., which problems this class contains). Specifically, there are many natural problems that are not known to reside in $\mathcal{P}$, and a natural class containing a host of such problems is presented next.

2.1.2.2 The class NP and NP-proof systems

(The origin of the notation NP will be discussed in Section 2.1.4.) We view NP as the class of decision problems that have efficiently verifiable proof systems. Loosely speaking, we say that a set S has a proof system if instances in S have valid proofs of membership (i.e., proofs accepted as valid by the system), whereas instances not in S have no valid proofs. Indeed, proofs are defined as strings that (when accompanying the instance) are accepted by the (efficient) verification procedure. We say that V is a verification procedure for membership in S if it satisfies the following two conditions:

1. Completeness: True assertions have valid proofs; that is, proofs accepted as valid by V. Bearing in mind that assertions refer to membership in S, this means that for every $x \in S$ there exists a string y such that $V(x, y) = 1$ (i.e., V accepts y as a valid proof for the membership of x in S).
2. Soundness: False assertions have no valid proofs. That is, for every $x \notin S$ and every string y it holds that $V(x, y) = 0$, which means that V rejects y as a proof for the membership of x in S.

We note that the soundness condition captures the "security" of the verification procedure; that is, its ability not to be fooled by anything into proclaiming a wrong assertion. The completeness condition captures the "viability" of the verification procedure; that is, its ability to be convinced of any valid assertion, when presented with an adequate proof. (We stress that, in general, proof systems are defined in terms of their verification procedures, which must satisfy adequate completeness and soundness conditions.) Our focus here is on efficient verification procedures that utilize relatively short proofs (i.e., proofs that are of length that is polynomially bounded by the length of the corresponding assertion).[3]

[3]**Advanced comment:** In continuation to Footnote 1, we note that in this chapter we consider *deterministic* (polynomial-time) verification procedures, and consequently the completeness and soundness conditions that we state here are error-less. In contrast, in Chapter 9, we will consider various types of probabilistic (polynomial-time) verification procedures as well as probabilistic

Definition 2.5 (efficiently verifiable proof systems):

- *A decision problem* $S \subseteq \{0,1\}^*$ *has an* efficiently verifiable proof system *if there exists a polynomial* p *and a polynomial-time* (verification) *algorithm* V *such that the following two conditions hold:*

  1. Completeness: *For every* $x \in S$*, there exists* y *of length at most* $p(|x|)$ *such that* $V(x,y) = 1$.

     (Such a string y is called an NP-witness for $x \in S$.)
  2. Soundness: *For every* $x \notin S$ *and every* y*, it holds that* $V(x,y) = 0$.

  Thus, $x \in S$ *if and only if there exists* y *of length at most* $p(|x|)$ *such that* $V(x,y) = 1$.

  In such a case, we say that S has an NP-proof system*, and refer to* V *as its* verification procedure (or as the proof system itself).

- *We denote by* $\mathcal{NP}$ *the class of decision problems that have efficiently verifiable proof systems.*

We note that the term *NP-witness* is commonly used, although in most cases V is not called a proof system (nor a verification procedure of such a system). In some cases, V (or the set of pairs accepted by V) is called a witness relation of S. We stress that the same set S may have many different NP-proof systems (see Exercise 2.2), and that in some cases the difference is not artificial (see Exercise 2.3).

Teaching note: Using Definition 2.5, it is typically easy to show that natural decision problems are in $\mathcal{NP}$. All that is needed is designing adequate NP-proofs of membership, which is typically quite straightforward and natural, because such decision problems are typically phrased as asking about the existence of a structure (or object) that can be easily verified as valid. For example, `SAT` is defined as the set of satisfiable Boolean formulae, which means asking about the existence of satisfying assignments. Indeed, we can efficiently check whether a given assignment satisfies a given formula, which means that we have (a verification procedure for) an NP-proof system for `SAT`.

We note that for any search problem R in $\mathcal{PC}$, the set of instances that have a solution with respect to R (i.e., $S_R \stackrel{\rm def}{=} \{x : R(x) \neq \emptyset\}$) is in $\mathcal{NP}$. (For any $R \in \mathcal{PC}$, consider the verification procedure V such that $V(x,y) \stackrel{\rm def}{=} 1$ if and only if $(x,y) \in R$, which in turn can be decided in $\mathrm{poly}(|x|)$-time.) Thus, *any search problem in* $\mathcal{PC}$ *can be viewed as a problem of searching for* (efficiently verifiable) *proofs* (i.e., NP-witnesses for the set of instances having solutions). Furthermore, any NP-proof system gives rise to a natural search problem in $\mathcal{PC}$; that is, the

completeness and soundness conditions. A common theme that underlies both treatments is that efficient verification is interpreted as meaning verification by a process that runs in time that is polynomial in the length of the assertion. In the current chapter, we use the equivalent formulation that considers the running time as a function of the total length of the assertion and the proof, but require that the latter has length that is polynomially bounded by the length of the assertion.

problem of searching for a valid proof (i.e., an NP-witness) for the given instance. (The verification procedure V yields the search problem that corresponds to $R = \{(x,y) : V(x,y)=1\}$.)

> **Teaching note:** The last paragraph suggests another easy way of showing that natural decision problems are in $\mathcal{NP}$: just thinking of the corresponding natural search problem. The point is that natural decision problems (in $\mathcal{NP}$) are phrased as referring to whether a solution exists (for the corresponding natural search problem). For example, in the case of SAT, the question is whether there exists a satisfying assignment to given Boolean formula, and the corresponding search problem is finding such an assignment. But in all these cases, it is easy to check the correctness of solutions; that is, the corresponding search problem is in $\mathcal{PC}$, which implies that the decision problem is in $\mathcal{NP}$.

Observe that $\mathcal{P} \subseteq \mathcal{NP}$ holds: A verification procedure for claims of membership in a set $S \in \mathcal{P}$ may just ignore the alleged NP-witness and run the decision procedure that is guaranteed by the hypothesis $S \in \mathcal{P}$; that is, $V(x,y) = A(x)$, where A is the aforementioned decision procedure. Indeed, the latter verification procedure is quite an abuse of the term (because it makes no use of the proof); however, it is a legitimate one. As we shall shortly see, the P-vs-NP Question refers to the question of whether such proof-oblivious verification procedures can be used for every set that has some efficiently verifiable proof system (i.e., whether $\mathcal{NP} \subseteq \mathcal{P}$).

2.1.2.3 The P versus NP question in terms of decision problems

Is it the case that NP-proofs are useless? That is, is it the case that for every efficiently verifiable proof system one can easily determine the validity of assertions without looking at the proof? If that were the case, then proofs would be meaningless, because they would have no fundamental advantage over directly determining the validity of the assertion. The conjecture $\mathcal{P} \neq \mathcal{NP}$ asserts that proofs are useful: there exists sets in $\mathcal{NP}$ that cannot be decided by a polynomial-time algorithm, and so for these sets obtaining a proof of membership (for some instances) is useful (because we cannot efficiently determine membership by ourselves).

In the foregoing paragraph we viewed $\mathcal{P} \neq \mathcal{NP}$ as asserting the advantage of obtaining proofs over deciding the truth by ourselves. That is, $\mathcal{P} \neq \mathcal{NP}$ asserts that (in some cases) verifying is easier than deciding. A slightly different perspective is that $\mathcal{P} \neq \mathcal{NP}$ asserts that finding proofs is harder than verifying their validity. This is the case, because for any set S having an NP-proof system, the ability to efficiently find proofs of membership in the set (i.e., finding an NP-witness of membership in S for any given $x \in S$), yields the ability to decide membership in S. Thus, for $S \in \mathcal{NP} \setminus \mathcal{P}$, it must be harder to find proofs of membership in S than to verify the validity of such proofs (which can be done in polynomial-time).

2.1.3 Equivalence of the two formulations

As hinted before, *the two formulations of the P-vs-NP Questions are equivalent.* That is, every search problem having efficiently checkable solutions is solvable in polynomial time (i.e., $\mathcal{PC} \subseteq \mathcal{PF}$) if and only if membership in any set that has an NP-proof system can be decided in polynomial time (i.e., $\mathcal{NP} \subseteq \mathcal{P}$). Recalling that $\mathcal{P} \subseteq \mathcal{NP}$, we prove

Theorem 2.6 *$\mathcal{PC} \subseteq \mathcal{PF}$ if and only if $\mathcal{P} = \mathcal{NP}$.*

Proof: Suppose, on the one hand, that the inclusion holds for the search version (i.e., $\mathcal{PC} \subseteq \mathcal{PF}$). We will show that this implies the existence of an efficient algorithm for finding NP-witnesses for any set in $\mathcal{NP}$, which in turn implies that this set is in $\mathcal{P}$. Specifically, let S be an arbitrary set in $\mathcal{NP}$, and V be the corresponding verification procedure (i.e., satisfying the conditions in Definition 2.5). Then $R \stackrel{\text{def}}{=} \{(x,y) : V(x,y) = 1\}$ is a polynomially bounded relation in $\mathcal{PC}$, and by the hypothesis its search problem is solvable in polynomial time (i.e., $R \in \mathcal{PC} \subseteq \mathcal{PF}$). Denoting by A the polynomial-time algorithm solving the search problem of R, we decide membership in S in the obvious way. That is, on input x, we output 1 if and only if $A(x) \neq \perp$, where the latter event holds if and only if $A(x) \in R(x)$, which in turn occurs if and only if $R(x) \neq \emptyset$ (equiv., $x \in S$). Thus, $\mathcal{NP} \subseteq \mathcal{P}$ (and $\mathcal{NP} = \mathcal{P}$) follows.

Suppose, on the other hand, that $\mathcal{NP} = \mathcal{P}$. We will show that this implies an efficient algorithm for determining whether a given string y' is a prefix of some solution to a given instance x of a search problem in $\mathcal{PC}$, which in turn yields an efficient algorithm for finding solutions. Specifically, let R be an arbitrary search problem in $\mathcal{PC}$. Then the set $S'_R \stackrel{\text{def}}{=} \{\langle x,y'\rangle : \exists y'' \text{ s.t. } (x,y'y'') \in R\}$ is in $\mathcal{NP}$ and hence in $\mathcal{P}$. This yields a polynomial-time algorithm for solving the search problem of R, by extending a prefix of a potential solution bit-by-bit (while using the decision procedure to determine whether or not the current prefix is valid). That is, on input x, we first check whether or not $(x,\lambda) \in S'_R$ and output $\perp$ (indicating $R(x) = \emptyset$) in case $(x,\lambda) \notin S'_R$. Next, we proceed in iterations, maintaining the invariant that $(x,y') \in S'_R$. In each iteration, we set $y' \leftarrow y'0$ if $(x,y'0) \in S'_R$ and $y' \leftarrow y'1$ if $(x,y'1) \in S'_R$. If none of these conditions hold (which happens after at most polynomially many iterations) then the current y' satisfies $(x,y') \in R$. Thus, $\mathcal{PC} \subseteq \mathcal{PF}$ follows. ■

Reflection: The first part of the proof of Theorem 2.6 associates with each set S in $\mathcal{NP}$ a natural relation R (in $\mathcal{PC}$). Specifically, R consists of all pairs (x,y) such that y is an NP-witness for membership of x in S. Thus, the search problem of R consists of finding such an NP-witness, when given x as input. Indeed, R is called the witness relation of S, and solving the search problem of R allows to decide membership in S. In the second part of the proof, we associate with each $R \in \mathcal{PC}$ a set S'_R (in $\mathcal{NP}$), but S'_R is not the natural set $S_R \stackrel{\text{def}}{=} \{x : \exists y \text{ s.t. } (x,y) \in R\}$ (which gives rise to R as its witness relation). Specifically, S'_R consists of strings that encode pairs (x,y') such that y' is a prefix of some string in $R(x) = \{y : (x,y) \in R\}$. Thus, deciding membership in S'_R allows to solve the search problem of R.

Conclusion: Theorem 2.6 justifies the traditional focus on the decision version of the P-vs-NP Question. Indeed, given that both formulations of the question are equivalent, we may just study the less cumbersome one.

2.1.4 The traditional definition of NP

Unfortunately, Definition 2.5 is not the commonly used definition of $\mathcal{NP}$. Traditionally, $\mathcal{NP}$ is defined as the class of sets that can be decided by a *fictitious* device called a non-deterministic polynomial-time machine (which explains the source of the notation NP). The reason that this class of fictitious devices is important is because it captures (indirectly) the definition of NP-proofs. Since the reader may come across the traditional definition of $\mathcal{NP}$ when studying different works, the author feels obliged to provide the traditional definition as well as a proof of its equivalence to Definition 2.5.

Definition 2.7 (non-deterministic polynomial-time Turing machines):

- *A* non-deterministic Turing machine *is define as in §1.2.3.1, except that the transition function maps symbol-state pairs to subsets of triples* (rather than to a single triple) *in* $\Sigma \times Q \times \{-1, 0, +1\}$*. Accordingly, the configuration following a specific instantaneous configuration may be one of several possibilities, each determine by a different possible triple. Thus, the* computations of a non-deterministic machine *on a* (fixed) *given input may result in different outputs.*

  In the context of decision problems one typically considers the question of whether or not there exists a computation that starting with a fixed input halts with output 1. We say that the non-deterministic machine M accept x *if there exists a computation of* M*, on input* x*, that halts with output 1. The* set accepted by a non-deterministic machine *is the set of inputs that are accepted by the machine.*

- *A* non-deterministic polynomial-time Turing machine *is defined as one that makes a number of steps that is polynomial in the length of the input. A set is in* $\mathcal{NP}$ *if there exists a non-deterministic polynomial-time machine that accepts it.*

We stress that Definition 2.7 makes no reference to the number (or fraction) of possible computations of the machine that yield a specific output (on a specific input).[4] Definition 2.7 only refers to whether or not computations leading to a certain output exist (for a specific input). The question of what does the mere existence of such possible computations mean in terms of real-life is not addressed,

[4]**Advanced comment:** In contrast, the definition of a probabilistic machine refers to this number (or, equivalently, to the probability that the machine produces a specific output, when the probability is essentially taken uniformly over all possible computations). Thus, a probabilistic machine refers to a natural model of computation that can be realized provided we can equip the machine with a source of randomness.

because the model of a non-deterministic machine is not meant to provide a reasonable model of a real-life computer. The model is meant to capture something completely different (i.e., it is meant to provide an elegant definition of the class $\mathcal{NP}$, while relying on the fact that Definition 2.7 is equivalent to Definition 2.5).

Teaching note: Whether or not Definition 2.7 is elegant is a matter of taste. For sure, many students find Definition 2.7 quite confusing, possibly because they assume that it represents some natural model of computation and allow themselves to be fooled by their intuition regarding such models. (Needless to say, the students' intuition regarding computation is irrelevant when applied to a fictitious model.)

Note that, unlike other definitions in this chapter, Definition 2.7 makes explicit reference to a specific model of computation. Still, a similar extension can be applied to other models of computation by considering adequate non-deterministic computation rules. Also note that, without loss of generality, we may assume that the transition function maps each possible symbol-state pair to exactly two triples (cf. Exercise 2.4).

Theorem 2.8 *Definition 2.5 is equivalent to Definition 2.7. That is, a set S has an NP-proof system if and only if there exists a non-deterministic polynomial-time machine that accepts S.*

Proof Sketch: Suppose, on one hand, that the set S has an NP-proof system, and let us denote the corresponding verification procedure by V. We construct a non-deterministic polynomial-time machine M that accepts S as follows. On input x, machine M makes an adequate $m = \text{poly}(|x|)$ number of non-deterministic steps producing (non-deterministically) a string $y \in \{0,1\}^m$, and then emulates $V(x, y)$. We stress that these non-deterministic steps may result in producing any m-bit string y. Recall that $x \in S$ if and only if there exists y of length at most $\text{poly}(|x|)$ such that $V(x, y) = 1$, which implies that M accepts S.

Suppose, on the other hand, that there exists a non-deterministic polynomial-time machine M that accepts S. Consider a deterministic machine M' that on input (x, y), where y has adequate length, emulates a computation of M on input x using y to determine the non-deterministic steps of M. That is, the i^{th} step of M on input x is determined by the i^{th} bit of y (which indicates which of the two possible moves to make at the current step). Note that $x \in S$ if and only if there exists y of length at most $\text{poly}(|x|)$ such that $M'(x, y) = 1$. Thus, M' gives rise to an NP-proof system for S. $\square$

2.1.5 In support of P different from NP

Intuition and concepts constitute... the elements of all our knowledge, so that neither concepts without an intuition in some way corresponding to them, nor intuition without concepts, can yield knowledge.

Immanuel Kant (1724–1804)

It is widely believed that P is different than NP; that is, that $\mathcal{PC}$ contains search problems that are not efficiently solvable, and that there are NP-proof systems for sets that cannot be decided efficiently. This belief is supported by both philosophical and empirical considerations.

- *Philosophical considerations*: Both formulations of the P-vs-NP Question refer to natural questions about which we have strong intuition. The notion of solving a (search) problem seems to presume that, at least in some cases (if not in general), finding a solution is significantly harder than checking whether a presented solution is correct. This translates to $\mathcal{PC} \setminus \mathcal{PF} \neq \emptyset$. Likewise, the notion of a proof seems presume that, at least in some cases (if not in general), the proof is useful in determining the validity of the assertion; that is, that deciding the validity of an assertion may be made significantly easier when provided with a proof. This translates to $\mathcal{P} \neq \mathcal{NP}$, which also implies that it is significantly harder to find proofs than to verify their correctness, which again coincides with the daily experience of researchers and students.

- *Empirical considerations*: The class NP (or rather $\mathcal{PC}$) contains thousands of different problems for which no efficient solving procedure is known. Many of these problems have arisen in vastly different disciplines, and were the subject of extensive research of numerous different communities of scientists and engineers. These essentially independent studies have all failed to provide efficient algorithms for solving these problems, a failure that is extremely hard to attribute to sheer coincidence or a stroke of bad luck.

Throughout the rest of this book, we will adopt the common belief that P is different from NP. At some places, we will explicitly use this conjecture (or even stronger assumptions), whereas in other places we will present results that are interesting (if and) only if $\mathcal{P} \neq \mathcal{NP}$ (e.g., the entire theory of NP-completeness becomes uninteresting if $\mathcal{P} = \mathcal{NP}$).

The $\mathcal{P} \neq \mathcal{NP}$ conjecture is indeed very appealing and intuitive. The fact that this natural conjecture is unsettled seems to be one of the sources of frustration of complexity theory. The author's opinion, however, is that this feeling of frustration is not in place. In contrast, the fact that complexity theory evolves around natural questions that are so difficult to resolve makes its study very exciting.

2.1.6 Two technical comments regarding NP

Recall that when defining $\mathcal{PC}$ (resp., $\mathcal{NP}$) we have explicitly confined our attention to search problems of polynomially bounded relations (resp., NP-witnesses of polynomial length). An alternative formulation may allow a binary relation R to be in $\mathcal{PC}$ (resp., $S \in \mathcal{NP}$) if membership of (x, y) in R can be decided in time that is polynomial in the length of x (resp., the verification of a candidate NP-witness y for membership of x in S is required to be performed in poly($|x|$)-time). Indeed, this mean that the validity of y can be determined without reading all of it (which means that some substring of y can be used as the effective y in the original definitions).

We comment that problems in $\mathcal{PC}$ (resp., $\mathcal{NP}$) can be solved in exponential-time (i.e., time $\exp(\text{poly}(|x|))$ for input x). This can be done by an exhaustive search among all possible candidate solutions (resp., all possible candidate NP-witnesses). Thus, $\mathcal{NP} \subseteq \mathcal{EXP}$, where $\mathcal{EXP}$ denote the class of decision problems that can be solved in exponential-time (i.e., time $\exp(\text{poly}(|x|))$ for input x).

2.2 Polynomial-time Reductions

We present the general notion of (polynomial-time) reductions among computational problems, and view the notion of a "Karp-reduction" as an important special case that suffices (and is more convenient) in many cases. Such reductions play a key role in the theory of NP-completeness, which is the topic of Section 2.3, but we stress that the fundamental nature of the notion of a reduction and highlight two specific applications (i.e., reducing search and optimization problems to decision problems). Furthermore, in the latter applications, it will be important to use the general notion of a reduction (i.e., "Cook-reduction" rather than "Karp-reduction").

Teaching note: We assume that many students have heard of reductions, but we fear that most have obtained a conceptually poor view of their fundamental nature. In particular, we fear that reductions are identified with the theory of NP-completeness, while in our opinion they only play a key role there. Furthermore, we believe it is important to show that natural search and optimization problems can be reduced to decision problems.

2.2.1 The general notion of a reduction

Reductions are procedures that use "functionally specified" subroutines. That is, the functionality of the subroutine is specified, but its operation remains unspecified and its running-time is counted at unit cost. Analogously to algorithms, which are modeled by Turing machines, reductions can be modeled as *oracle* (Turing) machines. A reduction solves one computational problem (which may be either a search or a decision problem) by using oracle (or subroutine) calls to another computational problem (which again may be either a search or a decision problem).

The notion of a general algorithmic reduction was discussed in §1.2.3.1 and §1.2.3.5. These reductions, called Turing-reductions (cf. §1.2.3.1) and modeled by oracle machines (cf. §1.2.3.5), made no reference to the time complexity of the main algorithm (i.e., the oracle machine). Here, we focus on efficient (i.e., polynomial-time) reductions, which are often called *Cook reductions*. That is, we consider oracle machines (as in Definition 1.11) that run in time polynomial in the length of their input. We stress that the running time of an oracle machine is the number of steps made during its computation, and that the oracle's reply on each query is obtained in a single step.

The key property of efficient reductions is that they allow for the transformation of efficient implementations of the subroutine into efficient implementations of the task reduced to it. That is, as we shall see, if one problem is Cook-reducible to another problem and the latter is polynomial-time solvable then so is the former. The most popular case is that of reducing decision problems to decision problems, but we will also consider reducing search problems to search problems and reducing search problems to decision problems. Note that when reducing to a decision problem, the oracle is determined as the single valid solver of the decision problem (i.e., the function $f : \{0,1\}^* \to \{0,1\}$ solves the decision problem of membership in S if, for every x, it holds that $f(x) = 1$ if $x \in S$ and $f(x) = 0$ otherwise). In contrast, when reducing to a search problem, there may be many different valid solvers (i.e., each function f that satisfies $(x, f(x)) \in R$ for every $(x, y) \in R$ is a valid solver of the search problem of R). We capture both cases below.

Definition 2.9 (Cook reduction): *A problem Π is* Cook-reducible *to a problem Π' if there exists a polynomial-time oracle machine M such that for every function f that solves Π' it holds that M^f solves Π, where $M^f(x)$ denotes the output of M on input x when given oracle access to f.*

Note that Π (resp., Π') may be either a search problem or a decision problem (or even a yet undefined type of a problem). At this point the reader should verify that if Π is Cook-reducible to Π' and Π' is solvable in polynomial-time then so is Π. (See Exercise 2.5 for other properties of Cook-reductions.) Also observe that the second part of the proof of Theorem 2.6 is actually a Cook-reduction of the search problem of any R in $\mathcal{PC}$ to a decision problem regarding a related set S'_R in $\mathcal{NP}$. Thus, that proof establishes that *any search problem in $\mathcal{PC}$ is Cook-reducible to some decision problem in $\mathcal{NP}$*. We shall see a tighter relation between search and decision problems in Section 2.2.3 (i.e., R will be reduced to $S_R = \{x : \exists y \text{ s.t. } (x,y) \in R\}$ rather than to $S'_R = \{(x, y') : \exists y'' \text{ s.t. } (x, y'y'') \in R\}$).

A Karp-reduction is a special case of a reduction (from a decision problem to a decision problem). Specifically, for decision problems S and S', we say that S is Karp-reducible to S' if there is a reduction of S to S' *that operates as follows*: On input x (an instance for S), the reduction computes x', makes query x' to the oracle S' (i.e., invokes the subroutine for S' on input x'), and answers whatever the latter returns. This reduction is often represented by the polynomial-time computable mapping of x to x'; that is, the standard definition of a Karp-reduction is actually as follows.

Definition 2.10 (Karp reduction): *A polynomial-time computable function f is called a* Karp-reduction of S to S' *if, for every x, it holds that $x \in S$ if and only if $f(x) \in S'$.*

Thus, syntactically speaking, a Karp-reduction is not a Cook-reduction, but it trivially gives rise to one (i.e., on input x, the oracle machine makes query $f(x)$, and returns the oracle answer). Being slightly inaccurate but essentially correct, we shall say that Karp-reductions are special cases of Cook-reductions. Needless to say, Karp-reductions constitute a very restricted case of Cook-reductions. Still, this restricted case suffices for many applications (e.g., most importantly for the theory of NP-completeness (when developed for decision problems)), but not for reducing a search problem to a decision problem. Furthermore, whereas each decision problem is Cook-reducible to its complement, some decision problems are *not* Karp-reducible to their complement (see Exercises 2.7 and 2.26).

We comment that Karp-reductions may (and should) be augmented in order to handle reductions of search problems to search problems. Such an augmented Karp-reduction of the search problem of R to the search problem of R' operates as follows: On input x (an instance for R), the reduction computes x', makes query x' to the oracle R' (i.e., invokes the subroutine for searching R' on input x') obtaining y' such that $(x',y') \in R'$, and uses y' to compute a solution y to x (i.e., $y \in R(x)$). Thus, such a reduction can be represented by two polynomial-time computable mappings, f and g, such that $(x, g(x,y')) \in R$ for any y' that solves $f(x)$ (i.e., for y' that satisfies $(f(x),y') \in R'$). (Indeed, in general, unlike in the case of decision problems, the reduction cannot just return y' as an answer to x.) This augmentation is called a Levin-reduction and, analogously to the case of a Karp-reduction, is often represented by the two polynomial-time computable mappings (i.e., of x to x', and of (x,y') to y).

Definition 2.11 (Levin reduction): *A pair of polynomial-time computable functions, f and g, is called a* Levin-reduction of R to R' *if for every $x \in S_R$ and $y' \in R'(f(x))$ it holds that $(x, g(x,y')) \in R$, where $S_R = \{x : \exists y \text{ s.t. } (x,y) \in R\}$ and $R'(x') = \{y' : (x',y') \in R'\}$. In addition, as an augmentation of a Karp-reduction, for every $x \notin S_R$ it must hold that $R'(f(x)) = \emptyset$.*

Indeed, the first function in a Levin-reduction (i.e., f) constitutes a Karp-reduction. As for the second function (i.e., g), it maps any solution y' for the reduced instance $f(x)$ to a solution for x (where this mapping may also depend on x).

Terminology: In the sequel, whenever we neglect to mention the type of a reduction, we refer to a Cook-reduction. Two additional terms, which will be particularly useful in the advanced chapters, are presented next.

- We say that two problems are computationally equivalent if they are reducible to one another. This means that the two problems are essentially as hard (or as easy).
- We say that a *class of problems, $\mathcal{C}$, is reducible to a problem* Π' *if every problem in $\mathcal{C}$, is reducible to* Π'. We say that the class $\mathcal{C}$ is reducible to the class $\mathcal{C}'$ if for every $\Pi \in \mathcal{C}$ there exists $\Pi' \in \mathcal{C}'$ such that Π is reducible to Π'.

2.2.2 Reducing optimization problems to search problems

Many search problems involve a set of potential solutions, per each problem instance, such that different solutions are assigned different "values" (resp., "costs").

In such a case, one is interested in finding a solution that has value exceeding some threshold (resp., cost below some threshold), or (even better) in finding a solution of maximum value (resp., minimum cost). For simplicity, let us focus on the case of a value which we wish to maximize. Still, there are two different objectives (i.e., exceeding a threshold and optimizing), giving rise to two different (auxiliary) search problems related to the same relation R. Specifically, for a binary relation R and a *value function* $f:\{0,1\}^*\times\{0,1\}^*\to\mathbb{R}$, we consider two search problems.

1. *Exceeding a threshold*: Given a pair (x,v) the task is to find $y\in R(x)$ such that $f(x,y)\geq v$, where $R(x)=\{y:(x,y)\in R\}$. That is, we are actually referring to the search problem of the relation

$$R_f \stackrel{\text{def}}{=} \{(\langle x,v\rangle,y):(x,y)\in R\wedge f(x,y)\geq v\}, \tag{2.1}$$

   where $\langle x,v\rangle$ denotes a string that encodes the pair (x,v).

2. *Maximization*: Given x the task is to find $y\in R(x)$ such that $f(x,y)=v_x$, where v_x is the maximum value of $f(x,y')$ over all $y'\in R(x)$. That is, we are actually referring to the search problem of the relation

$$R'_f \stackrel{\text{def}}{=} \{(x,y)\in R: f(x,y)=\max_{y'\in R(x)}\{f(x,y')\}\}. \tag{2.2}$$

Examples of value functions include the size of a clique in a graph, the amount of flow in a network (with link capacities), etc. The task may be to find a clique of size exceeding a given threshold in a given graph or to find a maximum-size clique in a given graph. Note that, in these examples, the "base" search problem (i.e., the relation R) is quite easy to solve, and the difficulty arises from the auxiliary condition on the value of a solution (presented in R_f and R'_f). Indeed, one may trivialize R (i.e., let $R(x)=\{0,1\}^{\mathrm{poly}(|x|)}$ for every x), and impose all necessary structure by the function f (see Exercise 2.8).

We confine ourselves to the case that f is polynomial-time computable, which in particular means that $f(x,y)$ can be represented by a rational number of length polynomial in $|x|+|y|$. We will show next that, in this case, the two aforementioned search problems (i.e., of R_f and R'_f) are computationally equivalent.

Theorem 2.12 *For any polynomial-time computable $f:\{0,1\}^*\times\{0,1\}^*\to\mathbb{R}$ and a polynomially bounded binary relation R, let R_f and R'_f be as in Eq.* (2.1) *and Eq.* (2.2)*, respectively. Then the search problems of R_f and R'_f are computationally equivalent.*

It follows that, *for $R\in\mathcal{PC}$ and polynomial-time computable f, both the R_f and R'_f are reducible to $\mathcal{NP}$.* We note, however, that, while $R_f\in\mathcal{PC}$ always holds, it is not necessarily the case that $R'_f\in\mathcal{PC}$. See further discussion following the proof.

Proof: The search problem of R_f is reduced to the search problem of R'_f by finding an optimal solution (for the given instance) and comparing its value to the given threshold value. That is, we construct an oracle machine that solves R_f by making a single query to R'_f. Specifically, on input (x, v), the machine issues the query x (to a solver for R'_f), obtaining the optimal solution y (or an indication $\perp$ that $R(x) = \emptyset$), computes $f(x, y)$, and returns y if $f(x, y) \geq v$. Otherwise (i.e., either $y = \perp$ or $f(x, y) < v$), the machine returns an indication that $R_f(x, v) = \emptyset$.

Turning to the opposite direction, we reduce the search problem of R_f to the search problem of R'_f by first finding the optimal value $v_x = \max_{y \in R(x)}\{f(x, y)\}$ (by binary search on its possible values), and next finding a solution of value (at least) v_x. In both steps, we use oracle calls to R_f. For simplicity, we assume that f assigns positive integer values, and let $\ell = \mathrm{poly}(|x|)$ be such that $f(x, y) \leq 2^\ell - 1$ for every $y \in R(x)$. Then, on input x, we first find $v_x = \max\{f(x, y) : y \in R(x)\}$, by making oracle calls of the form $\langle x, v\rangle$. The point is that $v_x < v$ if any only if $R_f(\langle x, v\rangle) = \emptyset$, which in turn is indicated by the oracle answer $\perp$ (to the query $\langle x, v\rangle$). Making ℓ queries, we determine v_x (see Exercise 2.9). Note that in case $R(x) = \emptyset$, all answers will indicate that $R_f(\langle x, v\rangle) = \emptyset$, which we treat as if $v_x = 0$. Finally, we make the query (x, v_x), and halt returning the oracle's answer. ■

Proof's digest. Note that the first direction uses the hypothesis that f is polynomial-time computable, whereas the opposite direction only used the fact that the optimal value lies in a finite space of exponential size that can be "efficiently searched". Whereas the first direction can be proved using a Levin-reduction, this seems impossible for the opposite direction (in general).

On the complexity of R_f and R'_f. We focus on the natural case in which $R \in \mathcal{PC}$ and f is polynomial-time computable. In this case, Theorem 2.12 implies that R_f and R'_f are computationally equivalent. A closer look reveals, however, that $R_f \in \mathcal{PC}$ always holds, whereas $R'_f \in \mathcal{PC}$ does *not* necessarily hold. That is, the problem of finding a solution (for a given instance) that exceeds a given threshold is in the class $\mathcal{PC}$, whereas the problem of finding an optimal solution is not necessarily in the class $\mathcal{PC}$. For example, the problem of finding a clique of a given size K in a given graph G is in $\mathcal{PC}$, whereas the problem of finding a maximum size clique in a given graph G is not known (and quite unlikely) to be in $\mathcal{PC}$ (although it is Cook-reducible to $\mathcal{PC}$). Indeed, the class of problems that are reducible to $\mathcal{PC}$ is a natural and interesting class (see the ending paragraph of Section 3.2.1); indeed, for every $R \in \mathcal{PC}$ and polynomial-time computable f, the former class contains R'_f.

2.2.3 Self-reducibility of search problems

The results presented in this section further justify the focus on decision problems. Loosely speaking, these results show that for many natural relations R, the question of whether or not the search problem of R is efficiently solvable (i.e., is in $\mathcal{PF}$) is determined by the question of whether or not the "decision problem implicit in R" (i.e., $S_R = \{x : \exists y \text{ s.t. } (x, y) \in R\}$) is efficiently solvable (i.e., is in $\mathcal{P}$). Note that the latter decision problem is easily reducible to the search problem of R, and so our focus is on the other direction. That is, we focus on relations R for which the search problem of R is reducible to the decision problem of S_R.

Teaching note: Our usage of the term self-reducibility differs from the traditional one. Traditionally, a decision problem is called self-reducible if it is Cook-reducible to itself via a reduction that on input x only makes queries that are smaller than x (according to some appropriate measure on the size of strings). Under some natural restrictions (i.e., the reduction takes the disjunction of the oracle answers) such reductions yield reductions of search to decision (as discussed in the main text).

Definition 2.13 (the decision implicit in a search and self-reducibility): *The* decision problem implicit the search problem of R *is deciding membership in the set* $S_R = \{x : R(x) \neq \emptyset\}$, *where* $R(x) = \{y : (x, y) \in R\}$. *The search problem of* R *is called* self-reducible *if it can be reduced to the decision problem of* S_R.

Note that the search problem of R and the problem of deciding membership in S_R refer to the same instances: The search problem requires finding an adequate solution (i.e., given x find $y \in R(x)$), whereas the decision problem refers to the question of whether such solutions exist (i.e., given x answer whether or not $R(x)$ is non-empty). Thus, S_R is really the "decision problem implicit in R," because it is a decision problem that one implicitly solves when solving the search problem of R. Indeed, *the decision problem of* S_R *is always reducible to the search problem for* R[5] (and if R is in $\mathcal{PC}$ then S_R is in $\mathcal{NP}$). It follows that *if a search problem* R *is self-reducible then it is computationally equivalent to the decision problem* S_R.

Self-reducibility means a reduction of the search problem to the decision problem implicit in it. We shall see that self-reducibility is a property of many natural search problems (including all NP-complete search problems). This justifies the relevance of decision problems to search problems in a stronger sense than established in Section 2.1.3: Recall that in Section 2.1.3 we showed that the fate of the search problem class $\mathcal{PC}$ (w.r.t $\mathcal{PF}$) is determined by the fate of the decision problem class $\mathcal{NP}$ (w.r.t $\mathcal{P}$). Here we show that, for many natural search problems in $\mathcal{PC}$ (i.e., self-reducible ones), the fate of such a problem R (w.r.t $\mathcal{PF}$) is determined by the fate of the decision problem S_R (w.r.t $\mathcal{P}$), where S_R is the decision problem implicit in R.

We now present a few search problems that are self-reducible. We start with SAT (see Section G.2), the set of satisfiable Boolean formulae (in CNF), and consider the search problem in which given a formula one should provide a truth assignment that satisfies it. The corresponding relation is denoted $R_{\mathtt{SAT}}$; that is, $(\phi, \tau) \in R_{\mathtt{SAT}}$ if τ is a satisfying assignment to the formulae ϕ. The decision problem implicit in $R_{\mathtt{SAT}}$ is indeed SAT. Note that $R_{\mathtt{SAT}}$ is in $\mathcal{PC}$ (i.e., it is polynomially-bounded and membership of (ϕ, τ) in $R_{\mathtt{SAT}}$ is easy to decide (by evaluating a Boolean expression)).

Proposition 2.14 ($R_{\mathtt{SAT}}$ is self-reducible): *The search problem of* $R_{\mathtt{SAT}}$ *is reducible to* SAT.

[5] For example, the reduction invoke the search oracle and answer 1 if and only if the oracle returns some string (rather than the "no solution" symbol).

Thus, the search problem of $R_{\mathtt{SAT}}$ is computationally equivalent to deciding membership in SAT. Hence, in studying the complexity of SAT, we also address the complexity of the search problem of $R_{\mathtt{SAT}}$.

Proof: We present an oracle machine that solves the search problem of $R_{\mathtt{SAT}}$ by making oracle calls to SAT. Given a formula ϕ, we find a satisfying assignment to ϕ (in case such an assignment exists) as follows. First, we query SAT on ϕ itself, and return an indication that there is no solution if the oracle answer is 0 (indicating $\phi \notin \mathtt{SAT}$). Otherwise, we let τ, initiated to the empty string, denote a prefix of a satisfying assignment of ϕ. We proceed in iterations, where in each iteration we extend τ by one bit. This is done as follows: First we derive a formula, denoted ϕ', by setting the first $|\tau|+1$ variables of ϕ according to the values $\tau 0$. We then query SAT on ϕ' (which means that we ask whether or not $\tau 0$ is a prefix of a satisfying assignment of ϕ). If the answer is positive then we set $\tau \leftarrow \tau 0$ else we set $\tau \leftarrow \tau 1$. This procedure relies on the fact that if τ is a prefix of a satisfying assignment of ϕ and $\tau 0$ is not a prefix of a satisfying assignment of ϕ then $\tau 1$ must be a prefix of a satisfying assignment of ϕ.

We wish to highlight a key point that has been blurred in the foregoing description. Recall that the formula ϕ' is obtained by replacing some variables by constants, which means that ϕ' *per se* contains Boolean variables as well as Boolean constants. However, the standard definition of SAT disallows Boolean constants in its instances.[6] Nevertheless, ϕ' can be simplified such that the resulting form contains no Boolean constants. This simplification is according to the straightforward boolean rules: That is, the constant `false` can be omitted from any clause, but if a clause contains only occurrences of the constant `false` then the entire formula simplifies to `false`. Likewise, if the constant `true` appears in a clause then the entire clause can be omitted, but if all clauses are omitted then the entire formula simplifies to `true`.) Needless to say, if the simplification process yields a Boolean constant then we may skip the query, and otherwise we just use the simplified form of ϕ' as our query. ■

Reductions analogous to the one used in the proof of Proposition 2.14 can be presented also for other search problems (and not only for NP-complete ones). Two such examples are searching for a 3-coloring of a given graph and searching for an isomorphism between a given pair of graphs (where the first problem is known to be NP-complete and the second problem is believed not to be NP-complete). In both cases, the reduction of the search problem to a decision problem involves extending a prefix of a valid solution by making suitable queries in order to decide which extension to use. Note, however, that in these cases the process of getting rid of constants (representing partial solutions) is more involved. For example, in the case of Graph 3-Colorability (resp., Graph Isomorphism) we need to enforce a partial coloring of a given graph (resp., a partial isomorphism between a given pair of graphs); see Exercises 2.10 and 2.11, respectively.

[6]While the problem seems rather technical at the current setting (as it merely amounts to whether or not the definition of SAT allows Boolean constants in its instances), it is far from being so technical in other cases (see Exercises 2.10 and 2.11).

Reflection: The proof of Proposition 2.14 (as well as the proofs of similar results) consists of two observations.

1. For every relation R in $\mathcal{PC}$, it holds that the search problem of R is reducible to the decision problem of $S'_R = \{(x,y') : \exists y'' \text{ s.t. } (x,y'y'') \in R\}$. Such a reduction is explicit in the proof of Theorem 2.6 and is implicit in the proof of Proposition 2.14.

2. For specific $R \in \mathcal{PC}$ (e.g., $S_{\mathtt{SAT}}$), deciding membership in S'_R is reducible to deciding membership in $S_R = \{x : \exists y \text{ s.t. } (x,y) \in R\}$. This is where the specific structure of SAT was used, allowing for a direct and natural transformation of instances of S'_R to instances of S_R.

   (We comment that if S_R is NP-complete then S'_R, which is always in $\mathcal{NP}$, is reducible to S_R by the mere fact that S_R is NP-complete; this comment is related to the following advanced comment.)

For an arbitrary $R \in \mathcal{PC}$, deciding membership in S'_R is not necessarily reducible to deciding membership in S_R. Furthermore, deciding membership in S'_R is not necessarily reducible to the search problem of R. (See Exercises 2.12 and 2.13.)

Teaching note: In the rest of this section, we assume that the students have heard of NP-completeness. Specifically, all we need the students to know is that a set S is $\mathcal{NP}$-complete if $S \in \mathcal{NP}$ and every set in $\mathcal{NP}$ is reducible to S. Yet, the teacher may prefer postponing the presentation of the following material to Section 2.3.1 (or even to a later stage).

Advanced comment: In general, self-reducibility is a property of the search problem and not of the decision problem implicit in it. Assuming that $\mathcal{P} \neq \mathcal{NP}$, there exists relations $R_1, R_2 \in \mathcal{PC}$ having the same implicit-decision problem (i.e., $\{x : R_1(x) \neq \emptyset\} = \{x : R_2(x) \neq \emptyset\}$) such that R_1 is self-reducible but R_2 is not (see Exercise 2.14). However, this phenomenon does not arise when NP-complete problems are involved; that is, *all search problems that refer to finding NP-witnesses for any NP-complete decision problem are self-reducible.*

Theorem 2.15 *For every R in $\mathcal{PC}$ such that S_R is $\mathcal{NP}$-complete, the search problem of R is reducible to deciding membership in S_R.*

In many cases, as in the proof of Proposition 2.14, the reduction of the search problem to the corresponding decision problem is quite natural. The following proof presents a generic reduction (which may be "unnatural" in some cases).

Proof: In order to reduce the search problem of R to deciding S_R, we compose the following two reductions:

1. A reduction of the search problem of R to deciding membership in $S'_R = \{(x,y') : \exists y'' \text{ s.t. } (x,y'y'') \in R\}$.

   As stated in the foregoing paragraph (titled "reflection"), such a reduction is implicit in the proof of Proposition 2.14 (as well as being explicit in the proof of Theorem 2.6).

2. A reduction of S'_R to S_R.

   This reduction exists by the hypothesis that S_R is $\mathcal{NP}$-complete and the fact that $S'_R \in \mathcal{NP}$. (Note that we do not assume that this reduction is a Karp-reduction, and furthermore it may be a "unnatural" reduction).

The theorem follows. ■

2.3 NP-Completeness

In light of the difficulty of settling the P-vs-NP Question, when faced with a hard problem H in NP, we cannot expect to prove that H is not in P (unconditionally). The best we can expect is to prove that H is not in P, assuming that NP is different from P. The contrapositive is proving that if H is in P, then so is any problem in NP (i.e., NP equals P). One possible way of proving such an assertion is showing that any problem in NP is polynomial-time reducible to H. This is the essence of the theory of NP-completeness.

Teaching note: Some students heard of NP-completeness before, but we suspect that many have missed important conceptual points. Specifically, we fear that they missed the point that the mere existence of NP-complete problems is amazing (let alone that these problems include natural ones such as SAT). We believe that this situation is a consequence of presenting the detailed proof of Cook's Theorem as the very first thing right after defining NP-completeness.

2.3.1 Definitions

The standard definition of NP-completeness refers to decision problems. Below we will also present a definition of NP-complete (or rather $\mathcal{PC}$-complete) search problems. In both cases, NP-completeness of a problem Π combines two conditions:

1. Π is in the class (i.e., Π being in $\mathcal{NP}$ or $\mathcal{PC}$, depending on whether Π is a decision or a search problem).

2. Each problem in the class is reducible to Π. This condition is called NP-hardness.

Although a perfectly good definition could have allowed arbitrary Cook-reductions (for establishing NP-hardness), it turns out that Karp-reductions (resp., Levin-reductions) suffice for establishing the NP-hardness of all natural NP-complete decision (resp., search) problems. Consequently, NP-completeness is usually defined using this restricted notion of a polynomial-time reduction.

Definition 2.16 (NP-completeness of decision problems, restricted notion): *A set S is $\mathcal{NP}$-complete if it is in $\mathcal{NP}$ and every set in $\mathcal{NP}$ is Karp-reducible to S.*

A set is $\mathcal{NP}$-hard if every set in $\mathcal{NP}$ is Karp-reducible to it. Indeed, there is no reason to insist on Karp-reductions (rather than using arbitrary Cook-reductions), except that the restricted notion suffices for all known demonstrations of NP-completeness and is easier to work with. An analogous definition applies to search problems.

Definition 2.17 (NP-completeness of search problems, restricted notion): *A binary relation R is $\mathcal{PC}$-complete if it is in $\mathcal{PC}$ and every relation in $\mathcal{PC}$ is Levin-reducible to R.*

In the sequel, we will sometimes abuse the terminology and refer to search problems as NP-complete (rather than $\mathcal{PC}$-complete). Likewise, we will say that a search problem is NP-hard (rather than $\mathcal{PC}$-hard) if every relation in $\mathcal{PC}$ is Levin-reducible to it.

We stress that the mere fact that we have defined something (i.e., NP-completeness) does not mean that this thing exists (i.e., that there exist objects that satisfy the property). *It is indeed remarkable that NP-complete problems do exist.* Such problems are "universal" in the sense that solving them allows to solve any other (reasonable) problem.

2.3.2 The existence of NP-complete problems

We suggest not to confuse the mere existence of NP-complete problems, which is remarkable by itself, with the even more remarkable existence of "natural" NP-complete problems. The following proof delivers the first message as well as focuses on the essence of NP-completeness, rather than on more complicated technical details. The essence of NP-completeness is that a single computational problem may "effectively encode" a wide class of seemingly unrelated problems.

Theorem 2.18 *There exist NP-complete relations and sets.*

Proof: The proof (as well as any other NP-completeness proof) is based on the observation that some decision problems in $\mathcal{NP}$ (resp., search problems in $\mathcal{PC}$) are "rich enough" to encode all decision problems in $\mathcal{NP}$ (resp., all search problems in $\mathcal{PC}$). This fact is most obvious for the "generic" decision and search problems, denoted $S_{\mathbf{u}}$ and $R_{\mathbf{u}}$ (and defined next), which are used to derive the simplest proof of the current theorem.

We consider the following relation $R_{\mathbf{u}}$ and the decision problem $S_{\mathbf{u}}$ implicit in $R_{\mathbf{u}}$ (i.e., $S_{\mathbf{u}} = \{\overline{x} : \exists y \text{ s.t. } (\overline{x}, y) \in R_{\mathbf{u}}\}$). Both problems refer to the same type of instances, which in turn have the form $\overline{x} = \langle M, x, 1^t \rangle$, where M is a description of a (deterministic) Turing machine, x is a string and t is a natural number. The number t is given in unary (rather than in binary) in order to allow various complexity measures, which depend on the instance length, to be polynomial in t (rather than poly-logarithmic in t).

> *The relation $R_{\mathbf{u}}$ consists of pairs $(\langle M, x, 1^t \rangle, y)$ such that M accepts the input pair (x, y) within t steps, where $|y| \leq t$.*

(Instead of requiring that $|y| \le t$, one may require that M is canonical in the sense that it reads its entire input before halting.) The corresponding set $S_{\mathtt{u}} \stackrel{\rm def}{=} \{\overline{x} : \exists y \text{ s.t. } (\overline{x}, y) \in R_{\mathtt{u}}\}$ consists of triples $\langle M, x, 1^t \rangle$ such that machine M accepts some input of the form $(x, \cdot)$ within t steps.

It is easy to see that $R_{\mathtt{u}}$ is in $\mathcal{PC}$ and that $S_{\mathtt{u}}$ is in $\mathcal{NP}$. Indeed, $R_{\mathtt{u}}$ is recognizable by a universal Turing machine, which on input $(\langle M, x, 1^t \rangle, y)$ emulates (t steps of) the computation of M on (x, y). (The fact that $S_{\mathtt{u}} \in \mathcal{NP}$ follows similarly.) We comment that u indeed stands for universal (machine), and the proof extends to any reasonable model of computation (which has adequate universal machines).

We now turn to show that $R_{\mathtt{u}}$ and $S_{\mathtt{u}}$ are NP-hard in the adequate sense (i.e., $R_{\mathtt{u}}$ is $\mathcal{PC}$-hard and $S_{\mathtt{u}}$ is $\mathcal{NP}$-hard). We first show that any set in $\mathcal{NP}$ is Karp-reducible to $S_{\mathtt{u}}$. Let S be a set in $\mathcal{NP}$ and let us denote its witness relation by R; that is, R is in $\mathcal{PC}$ and $x \in S$ if and only if there exists y such that $(x, y) \in R$. Let p_R be a polynomial bounding the length of solutions in R (i.e., $|y| \le p_R(|x|)$ for every $(x, y) \in R$), let M_R be a polynomial-time machine deciding membership (of alleged (x, y) pairs) in R, and let t_R be a polynomial bounding its running-time. Then, the desired Karp-reduction maps an instance x (for S) to the instance $\langle M_R, x, 1^{t_R(|x|+p_R(|x|))} \rangle$ (for $S_{\mathtt{u}}$); that is,

$$x \mapsto f(x) \stackrel{\rm def}{=} \langle M_R, x, 1^{t_R(|x|+p_R(|x|))} \rangle. \tag{2.3}$$

Note that this mapping can be computed in polynomial-time, and that $x \in S$ if and only if $f(x) = \langle M_R, x, 1^{t_R(|x|+p_R(|x|))} \rangle \in S_{\mathtt{u}}$. Details follow.

First, note that the mapping f does depend (of course) on S, and so it may depend on the fixed objects M_R, p_R and T_R (which depend on S). Thus, computing f on input x calls for printing the fixed string M_R, copying x, and printing a number of 1's that is a fixed polynomial in the length of x. Second, note that $x \in S$ if and only if there exists y such that $|y| \le p_R(|x|)$ and $(x, y) \in R$. Since M_R accepts $(x, y) \in R$ within $t_R(|x| + |y|)$ steps, it follows that $x \in S$ if and only if there exists y such that $|y| \le p_R(|x|)$ and M_R accepts (x, y) within $t_R(|x| + |y|)$ steps. It follows that $x \in S$ if and only if $f(x) \in S_{\mathtt{u}}$.

To reduce the search problem of any R in $\mathcal{PC}$ to the search problem of $R_{\mathtt{u}}$, we use essentially the same reduction. On input an instance x (for R), we make the query $\langle M_R, x, 1^{t_R(|x|+p_R(|x|))} \rangle$ to the search problem of $R_{\mathtt{u}}$ and return whatever the latter returns. Note that if $x \notin S$ then the answer will be "no solution", whereas for every x and y it holds that $(x, y) \in R$ if and only if $(\langle M_R, x, 1^{t_R(|x|+p_R(|x|))} \rangle, y) \in R_{\mathtt{u}}$. Thus, a Levin-reduction of R to $R_{\mathtt{u}}$ consists of the pair of functions (f, g), where f is the foregoing Karp-reduction and $g(x, y) = y$. Note that indeed, for every $(f(\overline{x}), y) \in R_{\mathtt{u}}$, it holds that $(x, g(x, y)) = (x, y) \in R$. ■

Advanced comment. Note that the role of 1^t in the definition of $R_{\mathtt{u}}$ is to allow placing $R_{\mathtt{u}}$ in $\mathcal{PC}$. In contrast, consider the relation $R'_{\mathtt{u}}$ that consists of pairs $(\langle M, x, t \rangle, y)$ such that M accepts xy within t steps. Then, as will become obvious in §4.2.1.2, membership in $R'_{\mathtt{u}}$ cannot be decided in polynomial time (even in the special case where x and y are fixed). Omitting t altogether from the problem instance yields a search problem that is not solvable at all. That is, consider the relation $R_H \stackrel{\text{def}}{=} \{(\langle M,x\rangle, y) : M(xy)=1\}$ (which is related to the halting problem). Indeed, the search problem of any relation (an in particular of any relation in $\mathcal{PC}$) is Karp-reducible to the search problem of R_H, but the latter is not solvable at all (i.e., there exists no algorithm that halts on every input and on input $\overline{x} = \langle M,x\rangle$ outputs y such that $(\overline{x},y) \in R_H$ if and only such a y exists).

Bounded Halting and Non-Halting

We note that the problem shown to be NP-complete in the proof of Theorem 2.18 is related to the following two problems, called `Bounded Halting` and `Bounded Non-Halting`. Fixing any programming language, the instance to each of these problems consists of a program π and a time bound t (presented in unary). The decision version of `Bounded Halting` (resp., `Bounded Non-Halting`) consists of determining whether or not there exists an input (of length at most t) on which the program π halts (resp., does *not* halt) in t steps, whereas the search problem consists of finding such an input.

Thus, the decision version of `Bounded Non-Halting` refers to a fundamental computational problem in the area of program verification; specifically, the question of whether a given program halts within a given time-bound on all inputs of a given length.[7] We mention the `Bounded Halting` problem because it is often referred to in the literature, but we believe that `Bounded Non-Halting` is more relevant to the project of program verification (because one seeks programs that halt on all inputs rather than programs that halt on some input).

It is easy to prove that both problems are NP-complete (see Exercise 2.18). The fact that `Bounded Non-Halting` is probably intractable (i.e., is intractable provided that $\mathcal{P} \neq \mathcal{NP}$) is even more relevant to the project of program verification than the fact that the Halting Problem is undecidable. The reason being that the latter problem (as well as other related undecidable problems) refers to arbitrarily long computations, whereas the former problem refers to computations of explicitly bounded number of steps. Specifically, `Bounded Non-Halting` is concerned with the existence of an input that causes the program to violate a certain condition (i.e., halting) within a given time-bound.

In light of the above, the common practice of bashing Bounded (Non-)Halting as an "unnatural" problem seems very odd at an age in which computer programs plays such a central role. (Nevertheless, we will use the term "natural" in this traditionally odd sense in the next title.)

[7] The length parameter need not equal the time-bound. Indeed, a more general version of the problem refers to two bounds, ℓ and t, and to whether the given program halts within t steps on each possible ℓ-bit input. It is easy to prove that the problem remains NP-complete also in the case that the instances are restricted to have parameters ℓ and t such that $t = p(\ell)$, for any fixed polynomial p (e.g., $p(n) = n^2$, rather than $p(n) = n$ as used in the main text).

2.3.3 Some natural NP-complete problems

Having established the mere existence of NP-complete problems, we now turn to prove the existence of NP-complete problems that do not (explicitly) refer to computation in the problem's definition. We stress that thousands of such problems are known (and a list of several hundreds can be found in [81]).

We will prove that deciding the satisfiability of propositional formulae is NP-complete (i.e., Cook's Theorem), and also present some combinatorial problems that are NP-complete. This presentation is aimed at providing a (small) sample of natural NP-completeness results as well as some tools towards proving NP-completeness of new problems of interest. We start by making a comment regarding the latter issue.

The reduction presented in the proof of Theorem 2.18 is called "generic" because it (explicitly) refers to any (generic) NP-problem. That is, we actually presented a scheme for design of reductions from any desired NP-problem to the single problem proved to be NP-complete. Indeed, in doing so, we have followed the definition of NP-completeness. However, once we know some NP-complete problems, a different route is open to us. We may establish the NP-completeness of a new problem by reducing a known NP-complete problem to the new problem. This alternative route is indeed a common practice, and it is based on the following simple proposition.

Proposition 2.19 *If an NP-complete problem* Π *is reducible to some problem* Π' *in NP then* Π' *is NP-complete. Furthermore, reducibility via Karp-reductions* (resp., Levin-reductions) *is preserved.*

Proof: The proof boils down to asserting the transitivity of reductions. That is, every problem in NP is reducible to Π, which in turn is reducible to Π'. Thus, by transitivity of reduction (see Exercise 2.6), every problem in NP is reducible to Π', which means that Π' is NP-hard and the proposition follows. ■

2.3.3.1 Circuit and formula satisfiability: CSAT and SAT

We consider two related computational problems, CSAT and SAT, which refer (in the decision version) to the satisfiability of circuits and formulae, respectively. (We refer the reader to the definition of Boolean circuits, formulae and CNF formulae that appear in §1.2.4.1.)

> **Teaching note:** We suggest to establish the NP-completeness of SAT by a reduction from the circuit satisfaction problem (CSAT), after establishing the NP-completeness of the latter. Doing so allows to decouple two important issues in the proof of the NP-completeness of SAT: the emulation of Turing machines by circuits, and the encoding of circuits by formulae with auxiliary variables.

CSAT. Recall that Boolean circuits are directed acyclic graphs with internal vertices, called gates, labeled by Boolean operations (of arity either 2 or 1), and external vertices called terminals that are associated with either inputs or outputs.

When setting the inputs of such a circuit, all internal nodes are assigned values in the natural way, and this yields a value to the output(s), called an evaluation of the circuit on the given input. The evaluation of circuit C on input z is denoted $C(z)$. We focus on circuits with a single output, and let CSAT denote the set of satisfiable Boolean circuits (i.e., a circuit C is in CSAT if there exists an input z such that $C(z) = 1$). We also consider the related relation $R_{\mathtt{CSAT}} = \{(C, z) : C(z) = 1\}$.

Theorem 2.20 (NP-completeness of CSAT): *The set* (resp., relation) CSAT (resp., $R_{\mathtt{CSAT}}$) *is $\mathcal{NP}$-complete* (resp., $\mathcal{PC}$-complete).

Proof: As usual it is easy to see that CSAT $\in \mathcal{NP}$ (resp., $R_{\mathtt{CSAT}} \in \mathcal{PC}$). Thus, we turn to showing that these problems are NP-hard. We will focus on the decision version (but also discuss the search version).

We will present (again, but for the last time in this book) a generic reduction, this time of any NP-problem to CSAT. The reduction is based on the observation, mentioned in §1.2.4.1, that the computation of polynomial-time algorithms can be emulated by polynomial-size circuits. In the current context, we wish to emulate the computation of a fixed machine M on input (x, y), *where x is fixed and y varies* (but $|y| = \text{poly}(|x|)$ and the total number of steps of $M(x, y)$ is polynomial in $|x| + |y|$). Thus, x will be "hard-wired" into the circuit, whereas y will serve as the input to the circuit. The circuit itself, denoted C_x, will consists of "layers" such that each layer represents an instantaneous configuration of the machine M, and the relation between consecutive configurations in a computation of this machine is captured by ("uniform") local gadgets in the circuit. The number of layers will depend on the polynomial that upper-bounds the running-time of M, and an additional gadget will be used to detect whether the last configuration is accepting. Thus, only the first layer of the circuit C_x will depend on x. The punch-line is that determining whether, for a given x, there exists a y ($|y| = \text{poly}(|x|)$) such that $M(x, y) = 1$ (in a given number of steps) reduces to the question of whether there exists a y such that $C_x(y) = 1$. Performing this reduction for any machine M_R that corresponds to any $R \in \mathcal{PC}$ (as in the proof of Theorem 2.18), we establish the fact that CSAT is NP-complete. Details follow.

Recall that we wish to reduce an arbitrary set $S \in \mathcal{NP}$ to CSAT. Let R, p_R, M_R and t_R be as in the proof of Theorem 2.18 (i.e., R is the witness relation of S, whereas p_R bounds the length of the NP-witnesses, M_R is the machine deciding membership in R, and t_R is its polynomial time-bound). Without loss of generality (and for simplicity), suppose that M_R is a one-tape Turing machine. We will construct a Karp-reduction that maps an instance x (for S) to a circuit, denoted $f(x) \stackrel{\text{def}}{=} C_x$, such that $C_x(y) = 1$ if and only if M_R accepts the input (x, y) within $t_R(|x| + p_R(|x|))$ steps. Thus, it will follow that $x \in S$ if and only if there exists $y \in \{0,1\}^{p_R(|x|)}$ such that $C_x(y) = 1$ (i.e., if and only if $C_x \in$ CSAT). The circuit C_x will depend on x as well as on M_R, p_R and t_R. (We stress that M_R, p_R and t_R are fixed, whereas x is varies and thus explicit in our notation.)

Before describing the circuit C_x, let us consider a possible computation of M_R on input (x, y), where x is fixed and y represents a generic string of length at most $p_R(|x|)$. Such a computation proceeds for $t = t_R(|x| + p_R(|x|))$ steps, and corresponds to a sequence of $t+1$ instantaneous configurations, each of length t. Each such configuration can be encoded by t pairs of symbols, where the first symbol in each pair indicates the contents of a cell and the second symbol indicates either a state of the machine or that the machine is not located in this cell. Thus, each pair is a member of $\Sigma \times (Q \cup \{\perp\})$, where Σ is the finite "work alphabet" of M_R, Q is its finite set of internal states, and $\perp$ is an indication that the machine is not present at a cell. The initial configuration includes xy as input, and the decision of $M_R(x,y)$ can be read from (the leftmost cell of) the last configuration.[8] With the exception of the first row, the values of the entries in each row are determined by the entries of the row just above it, where this determination reflects the transition function of M_R. Furthermore, the value of each entry in the said array is determined by the values of (up to) three entries that reside in the row above it (see Exercise 2.19). Thus, the aforementioned computation is represented by a $(t+1) \times t$ array, where each entry encodes one out of a constant number of possibilities, which in turn can be encoded by a constant-length bit string. See Figure 2.1.

(1,a)	(1,-)	(0,-)	(y_1,-)	(y_2,-)	(-,-)	(-,-)	(-,-)	(-,-)	(-,-)	initial configuration (with input $110y_1y_2$)
(3,-)	(1,b)	(0,-)	(y_1,-)	(y_2,-)	(-,-)	(-,-)	(-,-)	(-,-)	(-,-)	
(3,-)	(1,-)	(0,b)	(y_1,-)	(y_2,-)	(-,-)	(-,-)	(-,-)	(-,-)	(-,-)	
(3,-)	(1,c)	(0,-)								
(3,c)	(1,-)	(0,-)								
(1,-)	(1,f)	(0,-)								
										last configuration

Blank characters as well as the indication that the machine is not present in the cell are marked by a hyphen (-). The three arrows represent the determination of an entry by the three entries that reside above it. This machine accept the input if and only if it contains a zero.

Figure 2.1: An array representing ten computation steps on input $110y_1y_2$.

[8] We refer to the output convention presented in §1.2.3.1, by which the output is written in the leftmost cells and the machine halts at the cell to its right.

The circuit C_x has a structure that corresponds to the aforementioned array. Each entry in the array is represented by a *constant* number of gates such that when C_x is evaluated at y these gates will be assigned values that encode the contents of the said entry. In particular, the entries of the first row of the array are "encoded" by hard-wiring the reduction's input (i.e., x), and feeding the circuit's input (i.e., y) to the adequate input terminals. That is, the circuit has $p_R(|x|)$ ("real") input terminals, and the hard-wiring of constants to other gates that represent the first row is done by a simple gadget (as in Figure 1.2). Indeed, additional hard-wiring in the first row corresponds to the other fixed elements of the initial configuration (i.e., the blank symbols and the encoding of the initial state and the initial location; cf. Figure 2.1). The entries of subsequent rows will be "encoded" (or rather computed at evaluation time) by using *constant-size* circuits that determine the value of an entry based on the three relevant entries in the row above it. Recall that each entry is encoded by a constant number of gates, and thus these constant-size circuits merely compute the constant-size function described in Exercise 2.19. In addition, the circuit will have a few extra gates that check the values of the entries of the last row in order to determine whether or not it encodes an accepting configuration.[9] Note that the circuit C_x can be constructed in polynomial time from the string x, because we just need to encode x in an appropriate manner as well as generate a "highly uniform" grid-like circuit of size $O(t_R(|x|+p_R(|x|))^2)$.[10]

Although the foregoing construction of C_x capitalizes on various specific details of the (one-tape) Turing machine model, it can be adapted to any other "reasonable" model of efficient computation.[11] Alternatively, we recall the Cobham-Edmonds Thesis asserting that any decision problem that is solvable in polynomial-time (on some "reasonable" model) can be solved in polynomial-time by a (one-tape) Turing machine.

Turning back to the circuit C_x, we observe that indeed $C_x(y) = 1$ if and only if M_R accepts the input (x, y) while making at most $t = t_R(|x| + p_R(|x|))$ steps. Recalling that $S = \{x : \exists y \text{ s.t. } |y| \leq p_R(|x|) \land (x, y) \in R\}$ and that M_R decides membership in R in time t_R, we infer that $x \in S$ if and only if $f(x) = C_x \in \texttt{CSAT}$. Furthermore, $(x, y) \in R$ if and only if $(f(x), y) \in R_{\texttt{CSAT}}$. It follows that f is a Karp-reduction of S to $\texttt{CSAT}$, and, for $g(x, y) \stackrel{\text{def}}{=} y$ it holds that (f, g) is a Levin-reduction of R to $R_{\texttt{CSAT}}$. The theorem follows. ■

---

[9]In continuation to Footnote 8, we note that it suffices to check the values of the two leftmost entries of the last row. We assumed here that the circuit propagates a halting configuration to the last row. Alternatively, we may check for the existence of an accepting/halting configuration in the entire array, since this condition is quite simple.

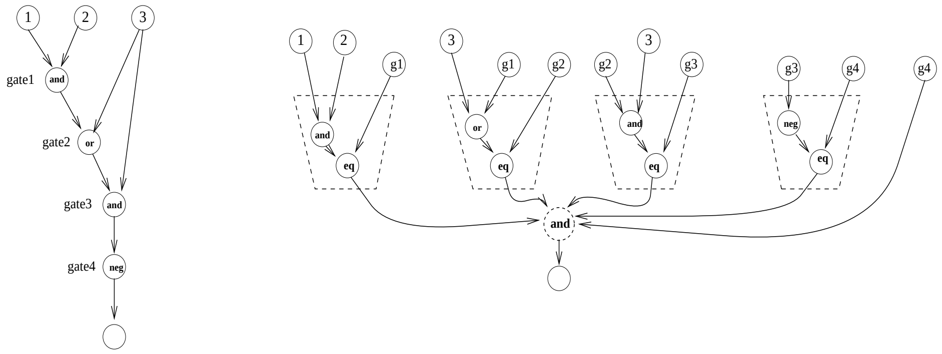
[10]**Advanced comment:** A more efficient construction, which generate almost-linear sized circuits (i.e., circuits of size $\widetilde{O}(t_R(|x| + p_R(|x|)))$) is known; see [167].

[11]**Advanced comment:** Note that it is actually inessential that each entry in each configuration is determined by a constant number of entries in the previous configuration. Any polynomial-time computable transformation of configurations will do since we can emulate such a transformation by a polynomial-size circuit. Indeed, this emulation will be based on presenting the said transformation in some concrete model of computation, which brings us to the next comment (invoking the Cobham-Edmonds Thesis).

SAT. Recall that Boolean formulae are special types of Boolean circuits (i.e., circuits having a tree structure).[12] We further restrict our attention to formulae given in conjunctive normal form (CNF). We denote by SAT the set of satisfiable CNF formulae (i.e., a CNF formula ϕ is in SAT if there exists an truth assignment τ such that $\phi(\tau) = 1$). We also consider the related relation $R_{\mathtt{SAT}} = \{(\phi, \tau) : \phi(\tau) = 1\}$.

Theorem 2.21 (NP-completeness of SAT): *The set* (resp., relation) SAT (resp., $R_{\mathtt{SAT}}$) *is* $\mathcal{NP}$*-complete* (resp., $\mathcal{PC}$-complete).

Proof: Since the set of possible instances of SAT is a subset of the set of instances of CSAT, it is clear that $\mathtt{SAT} \in \mathcal{NP}$ (resp., $R_{\mathtt{SAT}} \in \mathcal{PC}$). To prove that SAT is NP-hard, we reduce CSAT to SAT (and use Proposition 2.19). The reduction boils down to introducing auxiliary variables in order to "cut" the computation of an arbitrary ("deep") circuit into a conjunction of related computations of "shallow" circuits (i.e., depth-2 circuits) of unbounded fan-in, which in turn may be presented as a CNF formula. The aforementioned auxiliary variables hold the possible values of the internal gates of the input circuit, and the clauses of the CNF formula enforce the consistency of these values with the corresponding gate operation. For example, if $\mathtt{gate}_i$ and $\mathtt{gate}_j$ feed into $\mathtt{gate}_k$, which is a $\wedge$-gate, then the corresponding variables g_i, g_j, g_k should satisfy $g_k \equiv (g_i \wedge g_j)$ (which can be written as a 3CNF with four clauses). Details follow.



Using auxiliary variables (i.e., the g_i's) to "cut" a depth-5 circuit (into a CNF). The dashed regions will be replaced by equivalent CNF formulae. The dashed cycle representing an unbounded fan-in and*-gate is the conjunction of all constant-size circuits (which enforce the functionalities of the original gates) and the variable that represents the gate that feed the output terminal in the original circuit.*

Figure 2.2: The idea underlying the reduction of CSAT to SAT.

We start by Karp-reducing CSAT to SAT. Given a Boolean circuit C, with n input terminals and m gates, we first construct m *constant-size* formulae on $n + m$

[12]For an alternative definition, see Section G.2.

variables, where the first n variables correspond to the input terminals of the circuit, and the other m variables correspond to its gates. The i^{th} formula will depend on the variable that correspond to the i^{th} gate and the 1-2 variables that correspond to the vertices that feed into this gate (i.e., 2 vertices in case of $\wedge$-gate or $\vee$-gate and a single vertex in case of a $\neg$-gate, where these vertices may be either input terminals or other gates). This (constant-size) formula will be satisfied by a truth assignment if and only if this assignment matches the gate's functionality (i.e., feeding this gate with the corresponding values result in the corresponding output value). Note that these *constant-size* formulae can be written as constant-size CNF formulae (in fact, as 3CNF formulae).[13] Taking the conjunction of these m formulae as well as the variable associated with the gate that feeds into the output terminal, we obtain a formula ϕ in CNF (see Figure 2.2, where $n = 3$ and $m = 4$).

Note that ϕ can be constructed in polynomial-time from the circuit C; that is, the mapping of C to $\phi = f(C)$ is polynomial-time computable. We claim that C is in $\mathtt{CSAT}$ if and only if ϕ is in $\mathtt{SAT}$. Suppose that for some string s it holds that $C(s) = 1$. Then, assigning the i^{th} auxiliary variable the value that corresponds to the one assigned to the i^{th} gate of C when evaluated on s, we obtain (together with s) a truth assignment that satisfies ϕ (because such an assignment satisfies all m constant-size CNFs as well as the variable associated with the output of C). On the other hand, if τ satisfies ϕ then the first n bits in τ correspond to an input on which C evaluates to 1 (because the m constant-size CNFs guarantee that the variables of ϕ are assigned values that correspond to the evaluation of C on the first n bits of τ). Note that the latter mapping (of τ to its n-bit prefix) is the second mapping required by the definition of a Levin-reduction. Thus, we have established that f is a Karp-reduction of $\mathtt{CSAT}$ to $\mathtt{SAT}$, and that augmented with the second mapping it yields a Levin-reduction of $R_{\mathtt{CSAT}}$ to $R_{\mathtt{SAT}}$. ■

Comment. The fact that the second mapping required by the definition of a Levin-reduction is explicit in the proof of the validity of the corresponding Karp-reduction is a fairly common phenomenon. Actually, typical presentations of Karp-reductions provide two auxiliary polynomial-time computable mappings (in addition to the main mapping for instances from one problem (e.g., CSAT) to another (e.g., SAT)): The first auxiliary mapping is of solutions for the preimage instance (e.g., of CSAT) to solutions for the image instance of the reduction (e.g., of SAT), whereas the second mapping goes the other way around. (Note that only the main mapping and the second auxiliary mapping are required in the definition of a Levin-reduction.) For example, the proof of the validity of the Karp-reduction of $\mathtt{CSAT}$ to $\mathtt{SAT}$, denoted f, specified two mappings h and g such that $(C, s) \in R_{\mathtt{CSAT}}$ implies $(f(C), h(C, s)) \in R_{\mathtt{SAT}}$ and $(f(C), \tau) \in R_{\mathtt{SAT}}$ implies $(C, g(C, \tau)) \in R_{\mathtt{CSAT}}$. Specifically, in the proof of Theorem 2.21, we used $h(C, s) = (s, a_1, ..., a_m)$ where a_i is the value assigned to the i^{th} gate in the evaluation of $C(s)$, and $g(C, \tau)$ being

[13]Recall that any Boolean function can be written as a CNF (or even 3CNF) formula having size that is exponential in the length of its input, which in this case is a constant. Alternatively, note that the Boolean functions that we refer to here depends on 2-3 Boolean variables (and merely indicates whether or not the corresponding values respect the gate's functionality).

the n-bit prefix of τ. See Exercise 2.16.

3SAT. Note that the formulae resulting from the Karp-reduction presented in the proof of Theorem 2.21 are in conjunctive normal form (CNF) with each clause referring to at most three variables. Thus, the above reduction actually establishes the NP-completeness of 3SAT (i.e., SAT restricted to CNF formula with up to three variables per clause). Alternatively, one may Karp-reduce SAT (for CNF formula) to 3SAT (i.e., satisfiability of 3CNF formula), by replacing long clauses with conjunctions of three-variable clauses using auxiliary variables (see Exercise 2.20). Either way, we get the following result, where the furthermore part is proved by an additional reduction.

Proposition 2.22 *3SAT is NP-complete. Furthermore, the problem remains NP-complete also if we restrict the instances such that each variable appears in at most three clauses.*

Proof Sketch: The furthermore part is proved by reduction from 3SAT. We just replace each occurrence of each Boolean variable by a new copy of this variable, and add clauses to enforce that all these copies are assigned the same value. Specifically, replacing the variable z by copies $z^{(1)}, ..., z^{(m)}$, we add the clauses $z^{(i+1)} \vee \neg z^{(i)}$ for $i = 1..., m$ (where $m+1$ is understood as 1). □

2.3.3.2 Combinatorics and graph theory

Teaching note: The purpose of this subsection is to expose the students to a sample of NP-completeness results and proof techniques (i.e., the design of reductions among computational problems). The author believes that this traditional material is insightful, but one may skip it in the context of a complexity class.

We present just a few of the many appealing combinatorial problems that are known to be NP-complete. Throughout this section, we focus on the decision versions of the various problems, and adopt a more informal style. Specifically, we will present a typical decision problem as a problem of deciding whether a given instance, which belongs to a set of relevant instances, is a "yes-instance" or a "no-instance" (rather than referring to deciding membership of arbitrary strings in a set of yes-instances). For further discussion of this style and its rigorous formulation, see Section 2.4.1. We will also neglect to show that these decision problems are in NP.

We start with the set cover problem, in which an instance consists of a collection of finite sets $S_1, ..., S_m$ and an integer K and the question (for decision) is whether or not there exist (at most)[14] K sets that cover $\bigcup_{i=1}^m S_i$ (i.e., indices $i_1, ..., i_K$ such that $\bigcup_{j=1}^K S_{i_j} = \bigcup_{i=1}^m S_i$).

Proposition 2.23 `Set Cover` *is NP-complete.*

[14] Clearly, in case of `Set Cover`, the two formulations (i.e., asking for exactly K sets or at most K sets) are computationally equivalent.

Proof Sketch: We sketch a reduction of SAT to Set Cover. For a CNF formula ϕ with m clauses and n variables, we consider the sets $S_{1,\mathtt{t}}, S_{1,\mathtt{f}}, .., S_{n,\mathtt{t}}, S_{n,\mathtt{f}} \subseteq \{1, ..., m\}$ such that $S_{i,\mathtt{t}}$ (resp., $S_{i,\mathtt{f}}$) is the set of the indices of the clauses (of ϕ) that are satisfied by setting the i^{th} variable to `true` (resp., `false`). That is, if the i^{th} variable appears unnegated (resp., negated) in the j^{th} clause then $j \in S_{i,\mathtt{t}}$ (resp., $j \in S_{i,\mathtt{f}}$). Note that the union of these $2n$ sets equals $\{1, ..., m\}$. Now, on input ϕ, the reduction outputs the Set Cover instance $f(\phi) \stackrel{\text{def}}{=} ((S_1, .., S_{2n}), n)$, where $S_{2i-1} = S_{i,\mathtt{t}} \cup \{m+i\}$ and $S_{2i} = S_{i,\mathtt{f}} \cup \{m+i\}$ for $i = 1, ..., n$.

Note that f is computable in polynomial-time, and that if ϕ is satisfied by $\tau_1 \cdots \tau_n$ then the collection $\{S_{2i-\tau_i} : i = 1, ..., n\}$ covers $\{1, ..., m+n\}$. Thus, $\phi \in SAT$ implies that $f(\phi)$ is a positive `Set Cover` instance. On the other hand, each cover of $\{m+1, ..., m+n\} \subset \{1, ..., m+n\}$ must include either S_{2i-1} or S_{2i} for each i. Thus, a cover of $\{1, ..., m+n\}$ using n of the S_j's must contain, for every i, either S_{2i-1} or S_{2i} but not both. Setting τ_i accordingly (i.e., $\tau_i = 1$ if and only if S_{2i-1} is in the cover) implies that $\{S_{2i-\tau_i} : i = 1, ..., n\}$ covers $\{1, ..., m\}$, which in turn implies that $\tau_1 \cdots \tau_n$ satisfies ϕ. Thus, if $f(\phi)$ is a positive `Set Cover` instance then $\phi \in$ `SAT`. $\square$

Exact Cover and `3XC`. The exact cover problem is similar to the set cover problem, except that here the sets used in the cover are not allowed to intersect. That is, each element in the universe is covered by exactly one set in the cover. Restricting the set of instances to sequences of subsets each having exactly three elements, we get the restricted problem `3-Exact Cover` (`3XC`), where it is unnecessary to specify the number of sets to be used in the cover. The problem `3XC` is rather technical, but it is quite useful for demonstrating the NP-completeness of other problems (by reducing `3XC` to them).

Proposition 2.24 `3-Exact Cover` *is NP-complete.*

Indeed, it follows that the `Exact Cover` (in which sets of arbitrary size are allowed) is NP-complete. This follows both for the case that the number of sets in the desired cover is unspecified and for the various cases in which this number is bounded (i.e., upper-bounded or lower-bounded or both).

Proof Sketch: The reduction is obtained by composing three reductions. We first reduce a restricted case of `3SAT` (proved to be NP-complete in Proposition 2.22) to a restricted version of `Set Cover`, denoted `3SC`, in which each set has at most three elements (and an instance consists, as in the general case, of a sequence of finite sets as well as an integer K). Consider an instance ϕ of the aforementioned restricted case of `3SAT` (in which each *variable* appears in at most three clauses). If all three occurrences of a variable are of the same type (i.e., they are all negated or all non-negated) then this variable can be assigned a value that satisfies all clauses in which it appears, and so the variable and the clauses in which it appear can be omitted from the instance. In other words, we can reduce this restriction of `3SAT` to one in which each *literal* appears in at most two clauses.[15] Now, we reduce the new version of 3SAT to 3SC by using exactly the reduction presented in the proof of Proposition 2.23, and observing that the size of the generated sets is at most 3 (i.e., one more than the number of occurrences of the corresponding literal).

Next, we reduce 3SC to the following restricted case of Exact Cover, denoted 3XC', in which each set has at most three elements, an instance consists of a sequence of finite sets as well as an integer K, and the question is whether there exists an exact cover with at most K sets. The reduction maps an instance $((S_1, ..., S_m), K)$ of 3SC to the instance (C', K) such that C' is a collection of all subsets of each of the sets $S_1, ..., S_m$. Since each S_i has size at most 3, we introduce at most 7 non-empty subsets per each such set, and the reduction can be computed in polynomial-time. The reader may easily verify the validity of this reduction.

Finally, we reduce 3XC' to 3XC. Consider an instance $((S_1, ..., S_m), K)$ of 3XC', and suppose that $\bigcup_{i=1}^{m} S_i = [n]$. If $n > 3K$ then this is definitely a no-instance, which can be mapped to a dummy no-instance of 3XC, and so we assume that $x \stackrel{\text{def}}{=} 3K - n \geq 0$. Note that x represents the "excess" covering ability of an exact cover having K sets, each having three elements. Thus, we augment the set system with x new elements, denoted $n+1, ..., 3K$, and replace each S_i such that $|S_i| < 3$ by a sub-collection of 3-sets that cover S_i as well as arbitrary elements from $\{n+1, ..., 3K\}$. That is, in case $|S_i| = 2$, the set S_i is replaced by the sub-collection $(S_i \cup \{n+1\}, ..., S_i \cup \{3K\})$, whereas a singleton S_i is replaced by the sets $S_i \cup \{j_1, j_2\}$ for every $j_1 < j_2$ in $\{n+1, ..., 3K\}$. In addition, we add all possible 3-subsets of $\{n+1, ..., 3K\}$. This completes the description of the third reduction, the validity of which is left as an exercise. □

Vertex Cover, Independent Set, and Clique. Turning to graph theoretic problems (see Section G.1), we start with the Vertex Cover problem, which is a special case of the Set Cover problem. The instances consists of pairs (G, K), where $G = (V, E)$ is a simple graph and K is an integer, and the problem is whether or not there exists a set of (at most) K vertices that is incident to all graph edges (i.e., each edge in G has at least one endpoint in this set). Indeed, this instance of Vertex Cover can be viewed as an instance of Set Cover by considering the collection of sets $(S_v)_{v \in V}$, where S_v denotes the set of edges incident at vertex v (i.e., $S_v = \{e \in E : v \in e\}$). Thus, the NP-hardness of Set Cover follows from the NP-hardness of Vertex Cover. On the other hand, the Vertex Cover problem is computationally equivalent to the Independent Set and Clique problems (see Exercise 2.22), and thus it suffices to establish the NP-hardness of one of these problems.

Proposition 2.25 *The problems* Vertex Cover, Independent Set *and* Clique *are NP-complete.*

Proof Sketch: We show a reduction from 3SAT to Independent Set. On input a 3CNF formula ϕ with m clauses and n variables, we construct a graph with $7m$

[15] Actually, a closer look at the proof of Proposition 2.22 reveals the fact that the reduced instances satisfy this property anyhow.

vertices, denoted G_ϕ. The vertices are grouped in m cliques, each corresponding to one of the clauses and containing 7 vertices that correspond to the 7 truth assignments (to the 3 variables in the clause) that satisfy the clause. In addition to the internal edges in these m cliques, we add an edge between two such partial assignments that are inconsistent. That is, if a specific (satisfying) assignment to the variables of the i^{th} clause is inconsistent with some (satisfying) assignment to the variables of the j^{th} clause then we connect the corresponding vertices by an edge. Thus, on input ϕ, the reduction outputs the pair (G_ϕ, m).

Note that if ϕ is satisfiable by a truth assignment τ then there are no edges between the m vertices that correspond to the partial satisfying assignment (derived from τ). (We stress that any truth assignment to ϕ yields an independent set, but only a satisfying assignment guarantees that this independent set contains a vertex from each of the m cliques.) Thus, $\phi \in \texttt{SAT}$ implies that G_ϕ has an independent set of size m. On the other hand, an independent set of size m in G_ϕ must contain exactly one vertex in each of the m cliques, and thus induces a truth assignment that satisfies ϕ. (We stress that each independent set induces a consistent truth assignment to ϕ, because the partial assignments selected in the various cliques must be consistent, and that an independent set containing a vertex from a specific clique induces an assignment that satisfies the corresponding clause.) Thus, if G_ϕ has an independent set of size m then $\phi \in \texttt{SAT}$. $\square$

Graph 3-Colorability (`G3C`). In this problem the instances are graphs and the question is whether or not the graph can be colored using three colors (such that neighboring vertices are not assigned the same color).

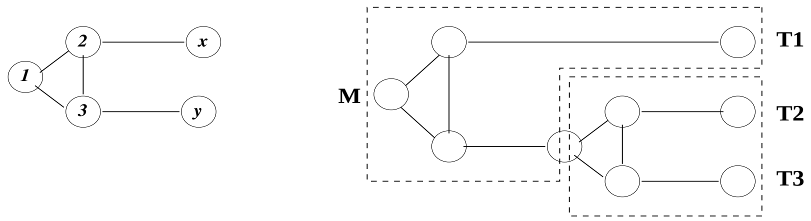
Proposition 2.26 `Graph 3-Colorability` *is NP-complete.*

Proof Sketch: We reduce `3SAT` to `G3C` by mapping a 3CNF formula ϕ to the graph G_ϕ, which consists of two designated vertices and gadgets per each variable and per each clause of ϕ.

- The two designated vertices are called `ground` and `false`, and are connected by an edge that ensures that they must be given different colors. We will refer to the color assigned to the vertex `ground` (resp., `false`) as color `ground` (resp., `false`). The third color will be called `true`.

- The gadget associated with variable x is a pair of vertices, associated with the two literals of x (i.e., x and $\neg x$). These vertices are connected by an edge, and each of them is also connected to the vertex `ground`. Thus, in a 3-coloring of G_ϕ one of the vertices associated with the variable is colored `true` and the other is colored `false`.

- The gadget associated with a clause C is depicted in Figure 2.3. It contains a master vertex, denoted **M**, and three terminal vertices, denoted **T1**, **T2** and **T3**. The master vertex is connected by edges to the vertices `ground` and `false`, and thus in a 3-coloring of G_ϕ the master vertex must be colored `true`. The gadget has the property that it is possible to color the terminals



In a generic 3-coloring of the sub-gadget it must hold that if $x = y$ then $x = y = 1$. Thus, if the three terminals of the gadget are assigned the same color, χ, then M is also assigned the color χ.

Figure 2.3: The reduction to G3C – the clause gadget and its sub-gadget.

with any combination of the colors `true` and `false`, except for coloring all terminals with `false`. The terminals of the gadget associated with clause C will be identified with the vertices that are associated with the corresponding literals appearing in C. See Figure 2.4.

Verifying the validity of the reduction is left as an exercise. □

2.3.4 NP sets that are neither in P nor NP-complete

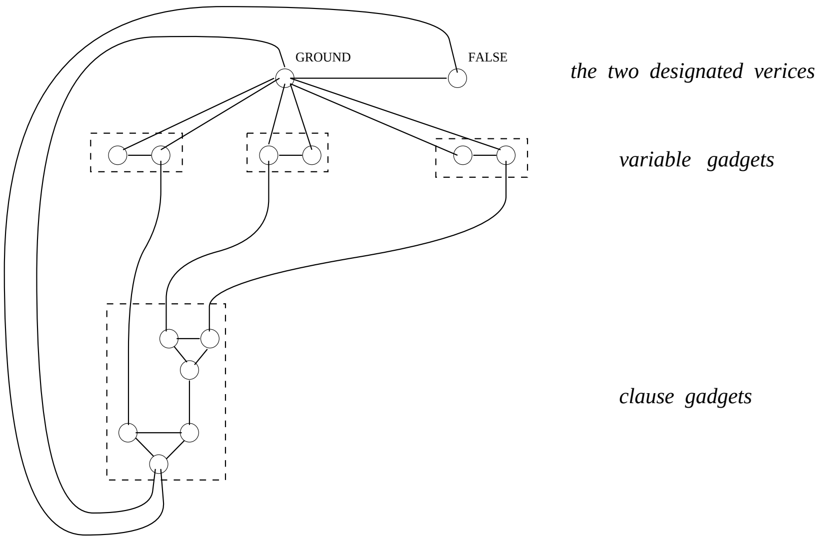
As stated in Section 2.3.3, thousands of problems have been shown to be NP-complete (cf., [81, Apdx.], which contains a list of more than three hundreds main entries). Things reached a situation in which people seem to expect any NP-set to be either NP-complete or in $\mathcal{P}$. This naive view is wrong: *Assuming $\mathcal{NP} \neq \mathcal{P}$, there exist sets in $\mathcal{NP}$ that are neither NP-complete nor in $\mathcal{P}$, where here NP-hardness allows also Cook-reductions.*

Theorem 2.27 *Assuming $\mathcal{NP} \neq \mathcal{P}$, there exist sets in $\mathcal{NP} \setminus \mathcal{P}$ such that some set in $\mathcal{NP}$ is not Cook-reducible to them.*

We mention that some natural problems (e.g., factoring) are conjectured to be neither solvable in polynomial-time nor NP-hard. One candidate class of such problems is $\mathcal{NP} \cap \mathrm{co}\mathcal{NP}$, where $\mathrm{co}\mathcal{NP} = \{\{0,1\}^* \setminus S : S \in \mathcal{NP}\}$ (see Section 2.4.3). Specifically, if $\Delta \stackrel{\rm def}{=} \mathcal{NP} \cap \mathrm{co}\mathcal{NP} \neq \mathcal{P}$ and $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$ then Δ is a class of sets that satisfy the conclusion of Theorem 2.27.[16] Below, the existence of sets satisfying this conclusion is proved based on the weaker assumption that $\mathcal{NP} \neq \mathcal{P}$ (which is actually a necessary condition for this conclusion).

Teaching note: We recommend either stating Theorem 2.27 without a proof or merely providing the proof idea.

[16]This implication is based on Theorem 2.33.



A single clause gadget and the relevant variables gadgets.

Figure 2.4: The reduction to G3C – connecting the gadgets.

Proof Sketch: The basic idea is modifying an arbitrary set in $\mathcal{NP} \setminus \mathcal{P}$ so as to fail all possible reductions (from $\mathcal{NP}$ to the modified set) as well as all possible polynomial-time decision procedures (for the modified set). Specifically, starting with $S \in \mathcal{NP} \setminus \mathcal{P}$, we derive $S' \subset S$ such that on one hand there is no polynomial-time reduction of S to S' while on the other hand $S' \in \mathcal{NP} \setminus \mathcal{P}$. The process of modifying S into S' proceeds in iterations, alternatively failing a potential reduction (by dropping sufficiently many strings from the rest of S) and failing a potential decision procedure (by including sufficiently many strings from the rest of S). Specifically, each potential reduction of S to S' can be failed by dropping finitely many elements from the current S', whereas each potential decision procedure can be failed by keeping finitely many elements of the current S'. These two assertions are based on the following two corresponding facts:

1. Any polynomial-time reduction (of any set not in $\mathcal{P}$) to a finite set (i.e., a finite subset of S) must fail, because only sets in $\mathcal{P}$ are Cook-reducible to a finite set. Thus, for any potential reduction (i.e., a polynomial-time oracle machine), there exists an input x on which this reduction fails. It follows that this failure is due to a finite set of queries (i.e., the set of all queries made by the reduction when invoked on an input that is smaller or equal to x). Thus, for every finite set $F \subset S$, any reduction of S to S' can be failed by dropping a finite number of elements from S' and without dropping elements of F.

2. For every finite set F, any polynomial-time decision procedure for $S \setminus F$ must fail, because S is (trivially) Cook-reducible to $S \setminus F$. Thus, for any potential

decision procedure (i.e., a polynomial-time algorithm), there exists an input x on which this procedure fails. It follows that this failure is due to a finite prefix of S (i.e., the set $\{z \in S : z \leq x\}$). Thus, for every finite set F, any polynomial-time decision procedure for $S \setminus F$ can be failed by keeping a finite subset of $S \setminus F$.

As stated, the process of modifying S into S' proceeds in iterations, alternatively failing a potential reduction (by dropping finitely many strings from the rest of S) and failing a potential decision procedure (by including finitely many strings from the rest of S). This can be done efficiently because *it is inessential to determine the first possible points of alternation* (in which sufficiently many strings were dropped (resp., included) to fail the next potential reduction (resp., decision procedure)). It suffices to guarantee that adequate points of alternation (albeit highly non-optimal ones) can be efficiently determined. Thus, S' is the intersection of S and some set in $\mathcal{P}$, which implies that $S' \in \mathcal{NP} \setminus \mathcal{P}$. Following are some comments regarding the implementation of the foregoing idea.

The foregoing plan calls for an ("effective") enumeration of all polynomial-time oracle machines (resp., polynomial-time algorithms). However, none of these sets can be enumerated (by an algorithm). Instead, we enumerate all corresponding machines along with all possible polynomials, and for each pair (M,p) we consider executions of machine M with time bound specified by the polynomial p. That is, we use the machine M_p obtained from the pair (M,p) by suspending the execution of M on input x after $p(|x|)$ steps. We stress that we do not know whether machine M runs in polynomial-time, but the computations of any polynomial-time machine is "covered" by some pair (M,p).

Let us clarify the process in which reductions and decision procedures are ruled out. We present a construction of a "filter" set F in $\mathcal{P}$ such that the final set S' will equal $S \cap F$. Recall that we need to select F such that each polynomial-time reduction of S to $S \cap F$ fails, and each polynomial-time procedure for deciding $S \cap F$ fails. The key observation is that for every finite F each polynomial-time reduction of S to $S \cap F$ fails, whereas for every co-finite F (i.e., finite $\{0,1\}^* \setminus F$) each polynomial-time procedure for deciding $S \cap F$ fails. Furthermore, each of these failures occur on some input, and this input is determined by finite portions of S and F. Thus, we alternate between failing possible reductions and decision procedures, while not trying to determine the "optimal" points of alternation but rather determining points of alternation in a way that allows for efficiently deciding membership in F. Specifically, we let $F = \{x : f(|x|) \equiv 0 \bmod 2\}$, where $f : \mathbb{N} \to \{0\} \cup \mathbb{N}$ is defined next such that $f(n)$ can be computed in time $\mathrm{poly}(n)$.

The value of $f(n)$ is defined by the the following experiment that consists of exactly n^3 computation steps (where cubic time is selected to allow for some non-trivial manipulations of data as conducted next). For $i = 0, 1, ...$, we scan all inputs in lexicographic order trying to find an input on which the $i + 1^{\text{st}}$ (modified) machine fails (where this machine is an oracle machine if i is even and a standard machine otherwise). In order to determine whether or not a failure occurs on a particular input, we may need to know the value of $f(n')$ for some n', which we just compute recursively (while counting the recursive steps in our total number of steps).[17] The point is that, when considering an input x, we may need the values of f only on $\{1, ..., p_{i+1}(|x|)\}$, where p_{i+1} is the polynomial bounding the running-time of the $i+1^{\text{st}}$ (modified) machine, and obtaining such a value takes at most $p_{i+1}(|x|)^3$ steps Also note that we may need to decide membership in $S \in \mathcal{NP}$, which we do by running the straightforward exponential-time algorithm (which tries all possible NP-witnesses). If we detect a failure of the $i+1^{\text{st}}$ machine, we increase i and proceed to the next iteration. When we reach the allowed number of steps (i.e., n^3 steps), we halt outputting the current value of i (i.e., the current i is output as the value of $f(n)$). Indeed, it is most likely that we will reach n^3 steps before examining inputs of length $3\log_2 n$, but this does not matter. What matters is that f is monotonically non-decreasing (because more steps allow to fail at least as many machines) as well as unbounded (see Exercise 2.27). Furthermore, by construction, $f(n)$ is computed in poly(n) time. □

Comment: The proof of Theorem 2.27 actually establishes that *for every $S \notin \mathcal{P}$ there exists $S' \notin \mathcal{P}$ such that S' is Karp-reducible to S but S is not Cook-reducible to S'*.[18] Thus, if $\mathcal{P} \neq \mathcal{NP}$ then there exists an infinite sequence of sets $S_1, S_2, ...$ in $\mathcal{NP} \setminus \mathcal{P}$ such that S_{i+1} is Karp-reducible to S_i but S_i is not Cook-reducible to S_{i+1}. That is, there exists an infinite hierarchy of problems (albeit unnatural ones), all in $\mathcal{NP}$, such that each problem is "easier" than the previous ones (in the sense that it can be reduced to the previous problems while these problems cannot be reduced to it).

2.4 Three relatively advanced topics

In this section we discuss three relatively advanced topics. The first topic, which was eluded to in previous sections, is the notion of promise problems (Section 2.4.1). Next we present an optimal search algorithm for NP (Section 2.4.2), and discuss the class (coNP) of sets that are complements of sets in NP.

Teaching note: These topics are typically not mentioned in a basic course on complexity. Still, pending on time constraints, we suggest discussing them at least at a high level.

2.4.1 Promise Problems

Promise problems are a natural generalization of (search and decision) problems, where one explicitly considers a set of legitimate instances (rather than considering any string as a legitimate instance). As noted before, this provides a more adequate formulation of natural computational problems (and indeed this formulation is used in all informal discussions). For example, in §2.3.3.2 we presented such problems

[17] We do not bother to present an efficient implementation of this process. That is, we may afford to recompute $f(n')$ every time we need it (rather than store it for later use).

[18] The said Karp-reduction (of S' to S) maps x to itself if $x \in F$ and otherwise maps x to a fixed no-instance of S.

using phrases like "given a graph and an integer..." (or "given a collection of sets..."). In other words, we assumed that the input instance has a certain format (or rather were "promised" that this is the case). Indeed, we claimed that in these cases the assumption can be removed without affecting the complexity of the problem, but we avoided providing a formal treatment of this issue, which is done next.

Teaching note: The notion of promise problems was originally introduced in the context of decision problems, and is typically used only in that context. However, we believe that it is as natural in the context of search problems, and present things accordingly.

2.4.1.1 Definitions

In the context of search problems, a promise problem is a relaxation in which one is only required to find solutions to instances in a predetermined set, called the promise. The requirement of efficient checking of solutions is adapted in an analogous manner.

Definition 2.28 (search problems with a promise): *A* search problem with a promise *consists of a binary relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ and a* promise *set P. Such a problem is also referred to as the* search problem R with promise P.

- *The search problem R with promise P is* solved by algorithm A *if for every $x \in P$ it holds that $(x, A(x)) \in R$ if $x \in S_R = \{x : R(x) \neq \emptyset\}$ and $A(x) = \perp$ otherwise, where $R(x) = \{y : (x,y) \in R\}$.*

  The time complexity of A on inputs in P *is defined as $T_{A|P}(n) \stackrel{\rm def}{=} \max_{x \in P \cap \{0,1\}^n} \{t_A(x)\}$, where $t_A(x)$ is the running time of $A(x)$ and $T_{A|P}(n) = 0$ if $P \cap \{0,1\}^n = \emptyset$.*

- *The search problem R with promise P is in the promise problem extension of $\mathcal{PF}$ if there exists a polynomial-time algorithm that solves this problem.*[19]

- *The search problem R with promise P is in the promise problem extension of $\mathcal{PC}$ if there exists a polynomial T and an algorithm A such that, for every $x \in P$ and $y \in \{0,1\}^*$, algorithm A makes at most $T(|x|)$ steps and it holds that $A(x,y) = 1$ if and only if $(x,y) \in R$.*

We stress that nothing is required of the solver in the case that the input violates the promise (i.e., $x \notin P$); in particular, in such a case the algorithm may halt with a wrong output. (Indeed, the standard formulation of search problems is obtained by considering the trivial promise $P = \{0,1\}^*$.)[20] In addition to the foregoing

[19]In this case it does not matter whether the time complexity of A is defined on inputs in P or on all possible strings. Suppose that A has polynomial time complexity T on inputs in P, then we can modify A to halt on any input x after at most $T(|x|)$ steps. This modification may only effects the output of A on inputs not in P (which is OK by us). The modification can be implemented in polynomial-time by computing $t = T(|x|)$ and emulating the execution of $A(x)$ for t steps. A similar comment applies to the definition of $\mathcal{PC}$, $\mathcal{P}$ and $\mathcal{NP}$.

[20]Here we refer to the formulation presented in Section 2.1.6.

motivation for promise problems, we mention one natural class of search problems with a promise. These are search problem in which the promise is that the instance has a solution (i.e., in terms of the foregoing notation $P = S_R$). We refer to such search problems by the name candid search problems.

Definition 2.29 (candid search problems): *An algorithm A solves the* candid search problem of the binary relation *R if for every $x \in S_R \stackrel{\rm def}{=} \{x : \exists y \text{ s.t. } (x,y) \in R\}$ it holds that $(x, A(x)) \in R$. The time complexity of such an algorithm is defined as $T_{A|S_R}(n) \stackrel{\rm def}{=} \max_{x \in P \cap \{0,1\}^n} \{t_A(x)\}$, where $t_A(x)$ is the running time of $A(x)$ and $T_{A|S_R}(n) = 0$ if $P \cap \{0,1\}^n = \emptyset$.*

Note that nothing is required when $x \notin S_R$: In particular, A may output a wrong solution (although no solutions exist) or run for more than $T_{A|S_R}(|x|)$ steps. Note that for $R \in \mathcal{PC}$, *if we "know" the time complexity of algorithm A* (e.g., if we can compute $T_{A|S_R}(n)$ in poly(n)-time), then we may modify A into an algorithm A' that solves the (general) search problem of R (i.e., halts on each input) in time $T_{A'}(n) = T_{A|S_R}(n) + \text{poly}(n)$. However, as we shall see in Section 2.4.2, the naive assumption by which we always know the running-time of an algorithm that we design is not necessarily valid.

Decision problems with a promise. In the context of decision problems, a promise problem is a relaxation in which one is only required to determine the status of instances that belong to a predetermined set, called the promise. The requirement of efficient verification is adapted in an analogous manner. In view of the standard usage of the term, we refer to *decision problems with a promise* by the name *promise problems.* Formally, promise problems refer to a three-way partition of the set of all strings into yes-instances, no-instances and instances that violate the promise. Standard decision problems are obtained as a special case by insisting that all inputs are allowed (i.e., the promise is trivial).

Definition 2.30 (promise problems): *A* promise problem *consists of a pair of non-intersecting sets of strings, denoted $(S_{\rm yes}, S_{\rm no})$, and $S_{\rm yes} \cup S_{\rm no}$ is called the* promise.

- *The promise problem $(S_{\rm yes}, S_{\rm no})$ is* solved by algorithm *A if for every $x \in S_{\rm yes}$ it holds that $A(x) = 1$ and for every $x \in S_{\rm no}$ it holds that $A(x) = 0$. The promise problem is in the promise problem extension of $\mathcal{P}$ if there exists a polynomial-time algorithm that solves it.*

- *The promise problem $(S_{\rm yes}, S_{\rm no})$ is in the promise problem extension of $\mathcal{NP}$ if there exists a polynomial p and a polynomial-time algorithm V such that the following two conditions hold:*

  1. Completeness: *For every $x \in S_{\rm yes}$, there exists y of length at most $p(|x|)$ such that $V(x,y) = 1$.*
  2. Soundness: *For every $x \in S_{\rm no}$ and every y, it holds that $V(x,y) = 0$.*

We stress that for algorithms of polynomial-time complexity, it does not matter whether the time complexity is defined only on inputs that satisfy the promise or on all strings (see Footnote 19). Thus, the extended classes $\mathcal{P}$ and $\mathcal{NP}$ (like $\mathcal{PF}$ and $\mathcal{PC}$) are invariant under this choice.

Reducibility among promise problems. The notion of a Cook-reduction extend naturally to promise problems, when postulating that a query that violates the promise (of the problem at the target of the reduction) may be answered arbitrarily.[21] The latter convention is consistent with the conceptual meaning of reductions and promise problems. Recall that reductions captures procedures that make subroutine calls to an arbitrary procedure that solves the reduced problem. But in case of promise problems, such a solver may behave arbitrarily on instances that violate the promise. Nevertheless, the main property of a reduction is preserved (see Exercise 2.28): *if the promise problem* Π *is Cook-reducible to a promise problem that is solvable in polynomial-time, then* Π *is solvable in polynomial-time.*

We warn that the extension of a complexity class to promise problems does not necessarily inherit the "structural" properties of the standard class. For example, in contrast to Theorem 2.33, there exists promise problems in $\mathcal{NP} \cap \mathrm{co}\mathcal{NP}$ such that every set in $\mathcal{NP}$ can be Cook-reduced to them: see Exercise 2.29. Needless to say, $\mathcal{NP} = \mathrm{co}\mathcal{NP}$ does not seem to follow from Exercise 2.29.

2.4.1.2 Discussion

The following discussion refers both to the decision and search versions of promise problems. Recall that promise problems offer the most direct way to capture natural computational problems.

Restricting a computational problem. In addition to the foregoing motivation to promise problems, we mention their use in formulating the notion of a restriction of a computational problem to a subset of the instances. Specifically, such a restriction means that the promise set of the restricted problem is a subset of the promise problem of the unrestricted problem. For example, when we say that 3SAT is a restriction of SAT, we refer to the fact that the set of allowed instances is now restricted to 3CNF formulae (rather than to arbitrary CNF formulae). In both cases, the natural computational problem is to determine satisfiability (or to find a satisfying assignment), but the set of instances (i.e., the promise set) is further restricted in the case of 3SAT. The fact that a restricted problem is never harder than the original problem is captured by the fact that the restricted problem is reducible to the original one (via the identity mapping).

[21] It follows that Karp-reductions among promise problems are not allowed to make queries that violate the promise. Specifically, we say that the promise problem $\Pi = (\Pi_{\mathrm{yes}}, \Pi_{\mathrm{no}})$ is Karp-reducible to the promise problem $\Pi' = (\Pi'_{\mathrm{yes}}, \Pi'_{\mathrm{no}})$ if there exists a polynomial-time mapping f such that for every $x \in \Pi_{\mathrm{yes}}$ (resp., $x \in \Pi_{\mathrm{no}}$) it holds that $f(x) \in \Pi'_{\mathrm{yes}}$ (resp., $f(x) \in \Pi'_{\mathrm{no}}$).

The standard convention of avoiding promise problems. Recall that, although promise problems provide a good framework for presenting natural computational problems, we managed to avoid this formulation in previous sections. This was done by relying on the fact that for all the (natural) problems considered in the previous sections, it is easy to decide whether or not a given instance satisfies the promise. For example, given a formula it is easy to decide whether or not it is in CNF (or 3CNF). Actually, the issue arises already when talking about formulae: What we are actually given is a string that is supposed to encode a formula (under some predetermined encoding scheme), and so the promise (which is easy to decide for natural encodings) is that the input string is a valid encoding of a formula. In any case, if the promise is efficiently recognizable (i.e., membership in it can be decided in polynomial-time) then we may avoid mentioning the promise by using one of the following two "nasty" conventions:

1. *Extending the set of instances to the set of all possible strings* (and allowing trivial solutions for the corresponding dummy instances). For example, in the case of a search problem, we may either define all instance that violate the promise to have no solution or define them to have a trivial solution (e.g., be a solution for themselves); that is, for a search problem R with promise P, we may consider the (standard) search problem of R where R is modified such that $R(x) = \emptyset$ for every $x \notin P$ (or, say, $R(x) = \{x\}$ for every $x \notin P$). In the case of a promise (decision) problem $(S_{\text{yes}}, S_{\text{no}})$, we may consider the problem of deciding membership in S_{yes}, which means that instances that violate the promise are considered as no-instances.

2. *Considering every string as a valid encoding of an object that satisfies the promise.* That is, fixing any string x_0 that satisfies the promise, we consider every string that violates the promise as if it were x_0. In the case of a search problem R with promise P, this means considering the (standard) search problem of R where R is modified such that $R(x) = R(x_0)$ for every $x \notin P$. Similarly, in the case of a promise (decision) problem $(S_{\text{yes}}, S_{\text{no}})$, we consider the problem of deciding membership in S_{yes}.

We stress that in the case that the promise is efficiently recognizable the aforementioned conventions (or modifications) do not effect the complexity of the relevant (search or decision) problem. That is, rather that considering the original promise problem we consider a (search or decision) problem (without a promise) that is computational equivalent to the original one. Thus, in some sense we loss nothing by studying the latter problem rather than the original one. On the other hand, even in case the two problems are computationally equivalent, it is useful to have a formulation that allows to distinguish between them (as we do distinguish between the different NP-complete problems although they are all computationally equivalent). This conceptual concern becomes of crucial importance in the case (to be discussed next) that the promise is not efficiently recognizable.

The foregoing transformations of promise problems into computationally equivalent standard (decision and search) problems does not necessarily preserve the complexity of the problem in the case that the promise is not efficiently recognizable. In this case, the terminology of promise problems is unavoidable. Consider, for example, the problem of deciding whether a Hamiltonian graph is 3-colorable. On the face of it, such a problem may have fundamentally different complexity than the problem of deciding whether a given graph is both Hamiltonian and 3-colorable.

The notion of a promise problem provides an adequate formulation for a variety of computational complexity notions and results. Examples include the notion of "unique solutions" (see Section 6.2.3) and the formulation of "gap problems" as capturing various approximation tasks (see Section 10.1).

2.4.1.3 The common convention

In spite of the foregoing opinions, we adopt the common convention of focusing on standard decision and search problems. That is, by default, all complexity classes refer to standard decision and search problems, and the exceptions in which we refer to promise problems are stated explicitly as such. Such exceptions appear in Sections 2.4.2, 6.1.2, 6.2.3, and 10.1.

2.4.2 Optimal search algorithms for NP

We refer to the candid search problem of any relation in $\mathcal{PC}$. Recall that $\mathcal{PC}$ is the class of search problems that allow for efficient checking of the correctness of candidate solutions (see Definition 2.3), and that the candid search problem is a search problem in which the solver is promised that the given instance has a solution (see Definition 2.29).

We claim the existence of an *optimal algorithm for solving any candid search problem of any relation in $\mathcal{PC}$*. Furthermore, we will explicitly present such an algorithm, and prove that it is optimal in a very strong sense: for any algorithm solving the candid search problem of $R \in \mathcal{PC}$, our algorithm solves the same problem in time that is at most a constant factor slower (ignoring a fixed additive polynomial term, which may be disregarded in case the problem is not solvable in polynomial-time). Needless to say, we do not know the time complexity of the aforementioned optimal algorithm (indeed if we knew it then we would have resolved the P-vs-NP Question). However, viewed differently, we "reduce" the P-vs-NP Question to determining the time complexity of a single explicitly presented algorithm (i.e., the optimal algorithm claimed in Theorem 2.31).

Theorem 2.31 *For every binary relation $R \in \mathcal{PC}$ there exists an algorithm A that satisfies the following:*

1. *A solves the candid search problem of R.*
2. *There exists a polynomial p such that for every algorithm A' that solves the candid search problem of R and for every $x \in S_R \stackrel{\text{def}}{=} \{x : R(x) \neq \emptyset\}$ it holds that $t_A(x) = O(t_{A'}(x) + p(|x|))$, where t_A* (resp., $t_{A'}$) *denotes the number of steps taken by A* (resp., A') *on input x.*

Interestingly, we establish the optimality of A without knowing what its (optimal) running-time is. Furthermore, the optimality claim is "point-wise" (i.e., it refers to any input) rather than "global" (i.e., referring to the (worst case) time complexity as a function of the input length).

We stress that the hidden constant in the O-notation depends only on A', but in the following proof the dependence is exponential in the length of the description of algorithm A' (and it is not known whether a better dependence can be achieved). Indeed, this dependence as well as the idea underlying it constitute one negative aspect of this otherwise amazing result. Another negative aspect is that the optimality of algorithm A refers only to inputs that have a solution (i.e., inputs in S_R). Finally, we note that the theorem as stated refers only to models of computation that have machines that can emulate a given number of steps of other machines with a constant overhead. We mention that in most natural models the overhead of such emulation is at most poly-logarithmic in the number of steps, in which case it holds that $t_A(x) = \widetilde{O}(t_{A'}(x) + p(|x|))$.

Proof Sketch: Fixing R, we let M be a polynomial-time algorithm that decides membership in R, and let p be a polynomial bounding the running-time of M. We present the following algorithm A that merely emulates all possible search algorithms "in parallel" and checks the result provided by each of them (using M), halting whenever it obtains a correct solution.

Since there are infinitely many possible algorithms, it may not be clear what we mean by the expression "emulating all possible algorithms in parallel." What we mean is emulating them at different "rates" such that the infinite sum of these rates converges to 1 (or to any other constant). Specifically, we will emulate the i^{th} possible algorithm at rate $1/(i+1)^2$, which means emulating a single step of this algorithm per $(i+1)^2$ emulation steps (performed for all algorithms). Note that a straightforward implementation of this idea may create a significant overhead, involved in switching frequently from the emulation of one machine to the emulation of another. Instead, we present an alternative implementation that proceeds in iterations. In the j^{th} iteration, for $i = 1, ..., 2^{j/2} - 1$, algorithm A emulates $2^j/(i+1)^2$ steps of the i^{th} machine. Each of these emulations is conducted in one chunk, and thus the overhead of switching between the various emulations is insignificant (in comparison to the total number of steps being emulated). In the case that some of these emulations halts with output y, algorithm A invokes M on input (x, y) and output y if and only if $M(x, y) = 1$. Furthermore, the verification of a solution provided by a candidate algorithm is also emulated at the expense of its step-count. (Put in other words, we augment each algorithm with a canonical procedure (i.e., M) that checks the validity of the solution offered by the algorithm.)

Clearly, whenever $A(x)$ outputs y (i.e., $y \neq \perp$) it must hold that $(x, y) \in R$. To show the optimality of A, we consider an arbitrary algorithm A' that solves the candid search problem of R. Our aim is to show that A is not much slower than A'. Intuitively, this is the case because the overhead of A results from emulating other algorithms (in addition to A'), but the total number of emulation steps wasted (due to these algorithms) is inversely proportional to the rate of algorithm A', which in turn is exponentially related to the length of the description of A'. The punch-line is that since A' is fixed, the length of its description is a constant. Details follow.

For every x, let us denote by $t'(x)$ the number of steps taken by A' on input x, where $t'(x)$ also accounts for the running time of $M(x,\cdot)$; that is, $t'(x) = t_{A'}(x) + p(|x|)$, where $t_{A'}(x)$ is the number of steps taken by $A'(x)$. Then, the emulation of $t'(x)$ steps of A' on input x is "covered" by the $j^{\rm th}$ iteration of A, provided that $2^j/(2^{|A'|+1})^2 \geq t'(x)$ where $|A'|$ denotes the length of the description of A'. (Indeed, we assume that the algorithms are emulated in lexicographic order, and note that there are at most $2^{|A'|+1}-2$ algorithms that precede A' in lexicographic order.) Thus, on input x, algorithm A halts after at most $j_{A'}(x)$ iterations, where $j_{A'}(x) = 2(|A'|+1) + \log_2(t_{A'}(x) + p(|x|))$, after emulating a total number of steps that is at most

$$t(x) \stackrel{\rm def}{=} \sum_{j=1}^{j_{A'}(x)} \sum_{i=1}^{2^{j/2}-1} \frac{2^j}{(i+1)^2} \;<\; 2^{j_{A'}(x)+1} \;=\; O\left(2^{2|A'|} \cdot (t_{A'}(x) + p(|x|))\right).$$

The question of how much time is required for emulating these many steps depends on the specific model of computation. In many models of computation, the emulation of t steps of one machine by another machine requires $\widetilde{O}(t)$ steps of the emulating machines, and in some models this emulation can even be performed with constant overhead. The theorem follows. $\square$

Comment: By construction, the foregoing algorithm A does not halt on input $x \notin S_R$. This can be easily rectified by letting A emulate a straightforward exhaustive search for a solution, and halt with output $\perp$ if this this exhaustive search indicates that there is no solution to the current input. This extra emulation will be done in parallel to all other emulations at a rate of one step for the extra emulation per each step of everything else.

2.4.3 The class coNP and its intersection with NP

By prepending the name of a complexity class (of decision problems) with the prefix "co" we mean the class of complement sets; that is,

$$\text{co}\mathcal{C} \stackrel{\rm def}{=} \{\{0,1\}^* \setminus S : S \in \mathcal{C}\}$$

Specifically, $\text{co}\mathcal{NP} = \{\{0,1\}^* \setminus S : S \in \mathcal{NP}\}$ is the class of sets that are complements of sets in $\mathcal{NP}$. Recalling that sets in $\mathcal{NP}$ are characterized by their witness relations such that $x \in S$ if and only if there exists an adequate NP-witness, it follows that their complement sets consists of all instances for which there are no NP-witness (i.e., $x \in \{0,1\}^* \setminus S$ if there is no NP-witness for x being in S).

Another perspective on $\text{co}\mathcal{NP}$ is obtained by considering the search problems in $\mathcal{PC}$. Recall that for such $R \in \mathcal{PC}$, the set of instances having a solution (i.e., $S_R = \{x : \exists y \text{ s.t. } (x,y) \in R\}$) is in $\mathcal{NP}$. It follows that the set of instances having no solution (i.e., $\{0,1\}^* \setminus S_R = \{x : \forall y\ (x,y) \notin R\}$) is in $\text{co}\mathcal{NP}$.

It is widely believed that $\mathcal{NP} \neq \text{co}\mathcal{NP}$ (which means that $\mathcal{NP}$ is not closed under complementation). Indeed, this conjecture implies $\mathcal{P} \neq \mathcal{NP}$ (because $\mathcal{P}$ is closed under complementation). The conjecture $\mathcal{NP} \neq \text{co}\mathcal{NP}$ means that some sets in co$\mathcal{NP}$ do not have NP-proof systems (because $\mathcal{NP}$ is the class of sets having NP-proof systems). As we will show next, under this conjecture, the complements of NP-complete sets do not have NP-proof systems; for example, there exists no NP-proof system for proving that a given formula is not satisfiable. We first establish this fact for NP-completeness in the standard sense (i.e., under Karp-reductions, as in Definition 2.16).

Proposition 2.32 *Suppose that $\mathcal{NP} \neq \text{co}\mathcal{NP}$ and let $S \in \mathcal{NP}$ such that every set in $\mathcal{NP}$ is Karp-reducible to S. Then $\overline{S} \stackrel{\text{def}}{=} \{0,1\}^* \setminus S$ is not in $\mathcal{NP}$.*

Proof: We first observe that the fact that every set in $\mathcal{NP}$ is Karp-reducible to S implies that every set in co$\mathcal{NP}$ is Karp-reducible to $\overline{S}$. We next claim that *if S' is in $\mathcal{NP}$ then every set that is Karp-reducible to S' is also in $\mathcal{NP}$.* Applying the claim to $S' = \overline{S}$, we conclude that $\overline{S} \in \mathcal{NP}$ implies co$\mathcal{NP} \subseteq \mathcal{NP}$, which in turn implies $\mathcal{NP} = \text{co}\mathcal{NP}$ in contradiction to the main hypothesis.

We now turn to prove the foregoing claim; that is, we prove that if S' has an NP-proof system and S'' is Karp-reducible to S' then S'' has an NP-proof system. Let V' be the verification procedure associated with S', and let f be a Karp-reduction of S'' to S'. Then, we define the verification procedure V'' (for membership in S'') by $V''(x,y) = V'(f(x),y)$. That is, any NP-witness that $f(x) \in S'$ serves as an NP-witness for $x \in S''$ (and these are the only NP-witnesses for $x \in S''$). This may not be a "natural" proof system (for S''), but it is definitely an NP-proof system for S''. ∎

Assuming that $\mathcal{NP} \neq \text{co}\mathcal{NP}$, Proposition 2.32 implies that sets in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ cannot be NP-complete with respect to Karp-reductions. In light of other limitations of Karp-reductions (see, e.g., Exercise 2.7), one may wonder whether or not the exclusion of NP-complete sets from the class $\mathcal{NP} \cap \text{co}\mathcal{NP}$ is due to the use of a restricted notion of reductions (i.e., Karp-reductions). The following theorem asserts that this is not the case: *some sets in $\mathcal{NP}$ cannot be reduced to sets in the intersection $\mathcal{NP} \cap \text{co}\mathcal{NP}$ even under general reductions* (i.e., Cook-reductions).

Theorem 2.33 *If every set in $\mathcal{NP}$ can be Cook-reduced to some set in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ then $\mathcal{NP} = \text{co}\mathcal{NP}$.*

In particular, assuming $\mathcal{NP} \neq \text{co}\mathcal{NP}$, no set in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ can be NP-complete, even when NP-completeness is defined with respect to Cook-reductions. Since $\mathcal{NP} \cap \text{co}\mathcal{NP}$ is conjectured to be a proper superset of $\mathcal{P}$, it follows (assuming $\mathcal{NP} \neq \text{co}\mathcal{NP}$) that there are decision problems in $\mathcal{NP}$ that are neither in $\mathcal{P}$ nor NP-hard (i.e., specifically, the decision problems in $(\mathcal{NP} \cap \text{co}\mathcal{NP}) \setminus \mathcal{P}$). We stress that Theorem 2.33 refers to standard decision problems and not to promise problems (see Section 2.4.1 and Exercise 2.29).

Proof: Analogously to the proof of Proposition 2.32 , the current proof boils down to proving that *if S is Cook-reducible to a set in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ then $S \in \mathcal{NP} \cap \text{co}\mathcal{NP}$.* Using the theorem's hypothesis, it follows that $\mathcal{NP} = \mathcal{NP} \cap \text{co}\mathcal{NP}$. Thus, $\mathcal{NP} \subseteq \text{co}\mathcal{NP}$ and $\mathcal{NP} = \text{co}\mathcal{NP}$ follows.

Fixing any S and $S' \in \mathcal{NP} \cap \text{co}\mathcal{NP}$ such that S is Cook-reducible to S', we prove that $S \in \mathcal{NP}$ (and the proof that $S \in \text{co}\mathcal{NP}$ is similar).[22] Let us denote by M the oracle machine reducing S to S'. That is, on input x, machine M makes queries and decides whether or not to accept x, and its decision is correct provided all queries are answered according to S'. To show that $S \in \mathcal{NP}$, we will present an NP-proof system for S. This proof system (or rather its verification procedure), denoted V, accepts a pair of the form $(x, ((z_1, \sigma_1, w_1), ..., (z_t, \sigma_t, w_t))$ if the following two conditions hold:

1. On input x, machine M accepts after making the queries $z_1, ..., z_t$, and obtaining the corresponding answers $\sigma_1, ..., \sigma_t$.

   That is, V check that, on input x, after obtaining the answers $\sigma_1, ..., \sigma_{i-1}$ to the first $i-1$ queries, the i^{th} query made by M equals z_i. In addition, V checks that M outputs 1 (indicating acceptance).

2. For every i, it holds that if $\sigma_i = 1$ then w_i is an NP-witness for $z_i \in S'$, whereas if $\sigma_i = 0$ then w_i is an NP-witness for $z_i \in \{0,1\}^* \setminus S'$.

   Thus, if this condition holds then it is the case that each σ_i indicates the correct status of z_i with respect to S' (i.e., $\sigma_i = 1$ if and only if $z_i \in S'$).

We stress that we use the fact that both S' and $\overline{S}' \stackrel{\text{def}}{=} \{0,1\}^* \setminus S$ have NP-proof systems, and refer to the corresponding NP-witnesses.

Note that V is indeed an NP-proof system for S. Firstly, the length of the corresponding witnesses is bounded by the running-time of the reduction (and the length of the NP-witnesses supplied for the various queries). Next note that V runs in polynomial time (i.e., verifying the first condition requires an emulation of the polynomial-time execution of M on input x when using the σ_i's to emulate the oracle, whereas verifying the second condition is done by invoking the relevant NP-proof systems). Finally, observe that $x \in S$ if and only if there exists a sequence $y \stackrel{\text{def}}{=} ((z_1, \sigma_1, w_1), ..., (z_t, \sigma_t, w_t))$ such that $V(x, y) = 1$. In particular, $V(x, y) = 1$ holds only if y contains a valid sequence of queries and answers made by $M(x)$ to the oracle S' and M accepts based on that sequence. ∎

The world view – a digest. Recall that on top of the $\mathcal{P} \neq \mathcal{NP}$ conjecture, we mentioned two other conjectures (which clearly imply $\mathcal{P} \neq \mathcal{NP}$):

1. The conjecture that $\mathcal{NP} \neq \text{co}\mathcal{NP}$ (equivalently, $\mathcal{NP} \cap \text{co}\mathcal{NP} \neq \mathcal{NP}$).

   This conjecture is equivalent to the conjecture that CNF formula have no short proofs of unsatisfiability (i.e., the set $\{0,1\}^* \setminus \texttt{SAT}$ has no NP-proof system).

[22] Alternatively, we show that $S \in \text{co}\mathcal{NP}$ by applying the following argument to $\overline{S} \stackrel{\text{def}}{=} \{0,1\}^* \setminus S$ and noting that $\overline{S}$ is Cook-reducible to S' (via S, or alternatively that $\overline{S}$ is Cook-reducible to $\{0,1\}^* \setminus S' \in \mathcal{NP} \cap \text{co}\mathcal{NP}$).

2. The conjecture that $\mathcal{NP} \cap \text{co}\mathcal{NP} \neq \mathcal{P}$.

   Notable candidates for the class $\mathcal{NP} \cap \text{co}\mathcal{NP} \neq \mathcal{P}$ include decision problems that are computationally equivalent to the integer factorization problem (i.e., the search problem (in $\mathcal{PC}$) in which, given a composite number, the task is to find its prime factors).

Combining these conjectures, we get the world view depicted in Figure 2.5, which also shows the class of co$\mathcal{NP}$-complete sets (defined next).

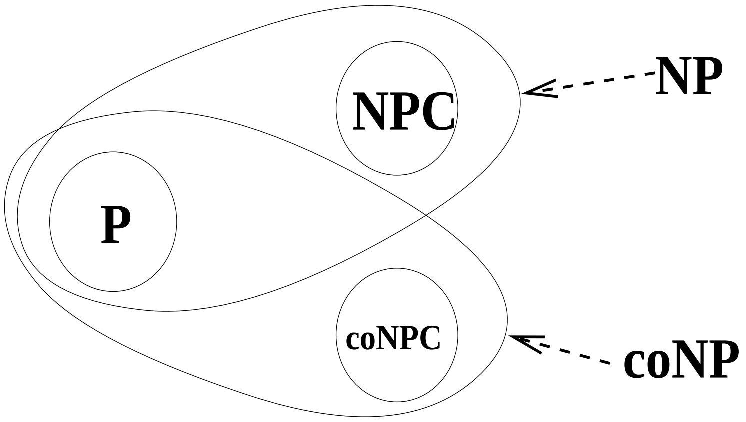


Figure 2.5: The world view under $\mathcal{P} \neq \text{co}\mathcal{NP} \cap \mathcal{NP} \neq \mathcal{NP}$.

Definition 2.34 *A set S is called* co$\mathcal{NP}$-hard *if every set in* co$\mathcal{NP}$ *is Karp-reducible to S. A set is called* co$\mathcal{NP}$-complete *if it is both in* co$\mathcal{NP}$ *and* co$\mathcal{NP}$*-hard.*

Indeed, insisting on Karp-reductions is essential for a distinction between $\mathcal{NP}$-hardness and co$\mathcal{NP}$-hardness.

Chapter Notes

The association of efficient computation with polynomial-time algorithms is attributed to the papers of Cobham [54] and Edmonds [66]. It is interesting to note that Cobham's starting point was his desire to present a philosophically sound concept of efficient algorithms, whereas Edmonds's starting point was his desire to articulate why certain algorithms are "good" in practice.

Many sources provide historical accounts of the developments that led to the formulation of the *P vs NP Problem* and to the development of the theory of NP-completeness (see, e.g., [81, Sec. 1.5] and [207]). Still, we feel that we should not refrain from offering our own impressions, which are based on the text of the original papers.

Nowadays, the theory of NP-completeness is commonly attributed to Cook [55], Karp [127], and Levin [142]. It seems that Cook's starting point was his interest in theorem proving procedures for propositional calculus [55, P. 151]. Trying to provide evidence to the difficulty of deciding whether or not a given formula is a tautology, he identified $\mathcal{NP}$ as a class containing "many apparently difficult problems" (cf, e.g., [55, P. 151]), and showed that any problem in $\mathcal{NP}$ is reducible to deciding membership in the set of 3DNF tautologies. In particular, Cook emphasizes the importance of the concept of polynomial-time reductions and of the complexity class $\mathcal{NP}$ (both explicitly defined for the first time in his paper). He also showed that `CLIQUE` is computationally equivalent to `SAT`, and envisioned a class of problems of the same nature. Karp's paper [127] can be viewed as fulfilling Cook's prophecy: Stimulated by Cook's work, Karp demonstrates that a "large number of classic difficult computational problems, arising in fields such as mathematical programming, graph theory, combinatorics, computational logic and switching theory, are [NP-]complete (and thus equivalent)" [127, P. 86]. Specifically, his list of twenty-one NP-complete problems includes Integer Linear Programming, Hamilton Circuit, Chromatic Number, Exact Set Cover, Steiner Tree, Knapsack, Job Scheduling, and Max Cut. Karp defines $\mathcal{NP}$ in terms of Definition 2.5, points to its relation to "backtrack search of polynomial bounded depth" and views it as the residence of a "wide range of important computational problems" (which are not in $\mathcal{P}$). Independently of these developments, while being in the USSR, Levin proved the existence of "universal search problems" (where universality meant NP-completeness [142]). The starting point of Levin's work was his interest in the "perebor" conjecture asserting the inherent need for brute-force in some search problems that have efficiently checkable solutions (i.e., problems in $\mathcal{PC}$). He emphasizes the implication of polynomial-time reductions on the relation between the time complexity of the related problem (for any growth rate of the time complexity), asserts the NP-completeness of six "classical search problems" and claims that the underlying method "provides a mean for readily obtaining" similar results for "many other important search problems."

It is interesting to note that although the works of Cook [55], Karp [127], and Levin [142] were received with different levels of enthusiasm, none of the contemporaries realized the depth of the discovery and the difficulty of the question posed (i.e., the P-vs-NP Question). This fact is evident in every account from the early 1970's, and may explain the frustration of the relevant generation of researchers, which expected the P-vs-NP Question to be resolved in their life-time (if not in a matter of years). Needless to say, the author's opinion is that there was absolutely no justification for these expectations, and that one should have actually expected quite the opposite.

We mention that the three "founding papers" of the theory of NP-completeness (i.e., Cook [55], Karp [127], and Levin [142]) use the three different types of reductions used in this chapter. Specifically, Cook uses the general notion of polynomial-time reduction [55], often referred to as Cook-reductions (Definition 2.9). The notion of Karp-reductions (Definition 2.10) originates from Karp's paper [127], whereas its augmentation to search problems (i.e., Definition 2.11) originates from Levin's paper [142]. It is worth noting that unlike Cook and Karp's works, which treat decision problems, Levin's work is stated in terms of search problems.

The reductions presented in §2.3.3.2 are not necessarily the original ones. Most notably, the reduction establishing the NP-hardness of the Independent Set problem (i.e., Proposition 2.25) is adapted from [70]. In contrast, the reductions presented in §2.3.3.1 are merely a re-interpretation of the original reduction as presented in [55]. The equivalence of the two definitions of $\mathcal{NP}$ (i.e., Theorem 2.8) was proved in [127].

The existence of NP-sets that are neither in P nor NP-complete (i.e., Theorem 2.27) was proven by Ladner [139], Theorem 2.33 was proven by Selman [185], and the existence of optimal search algorithms for NP-relations (i.e., Theorem 2.31) was proven by Levin [142]. (Interestingly, the latter result was proved in the same paper in which Levin presented the discovery of NP-completeness, independently of Cook and Karp.) Promise problems were explicitly introduced by Even, Selman and Yacobi [68]; see [90] for a survey of their numerous applications.

We mention that the standard reductions used to establish natural NP-completeness results have several additional properties or can be modified to have such properties. These properties include an efficient transformation of solutions in the direction of the reduction (see Exercise 2.16), the preservation of the number of solutions (see Exercise 2.17), being computable by a log-space algorithm (see Section 5.2.2), and being invertible in polynomial-time (see [35], which actually refers to a stronger notion).

Exercises

Exercise 2.1 ($\mathcal{PF}$ contains problems that are not in $\mathcal{PC}$) Show that $\mathcal{PF}$ contains some (unnatural) problems that are not in $\mathcal{PC}$.

Guideline: Consider the relation $R = \{(x,1) : x \in \{0,1\}^*\} \cup \{(x,0) : x \in S\}$, where S is some undecidable set. Note that R is the disjoint union of two binary relations, denoted R_1 and R_2, where R_1 is in $\mathcal{PF}$ whereas R_2 is not in $\mathcal{PC}$. Furthermore, for every x it holds that $R_1(x) \neq \emptyset$.

Exercise 2.2 Show that any $S \in \mathcal{NP}$ has many different NP-proof systems (i.e., verification procedures $V_1, V_2, ...$ such that $V_i(x,y) = 1$ does not imply $V_j(x,y) = 1$ for $i \neq j$).

Guideline: For V and p be as in Definition 2.5, define $V_i(x,y) = 1$ if $|y| = p(|x|) + 1$ and there exists a prefix y' of y such that $V(x,y) = 1$.

Exercise 2.3 Relying on the fact that primality is decidable in polynomial-time and assuming that there is no polynomial-time factorization algorithm, present two "natural but fundamentally different" NP-proof systems for the set of composite numbers.

Guideline: Consider the following verification procedures V_1 and V_2 for the set of composite numbers. Let $V_1(n,y) = 1$ if and only if $y = n$ and n is not a prime, and $V_2(n,m) = 1$ if and only if m is a non-trivial divisor of n. Show that valid proofs with respect to V_1 are easy to find, whereas valid proofs with respect to V_2 are hard to find.

Exercise 2.4 Regarding Definition 2.7, show that if S is accepted by some non-deterministic machine of time complexity t then it is accepted by a non-deterministic machine of time complexity $O(t)$ that has a transition function that maps each possible symbol-state pair to exactly two triples.

Exercise 2.5 Verify the following properties of Cook-reductions:

1. If Π is Cook-reducible to Π' and Π' is solvable in polynomial-time then so is Π.
2. Cook-reductions are transitive (i.e., if Π is Cook-reducible to Π' and Π' is Cook-reducible to Π'' then Π is Cook-reducible to Π'').
3. If Π is solvable in polynomial-time then it is Cook-reducible to any problem Π'.

In continuation to the last item, show that a problem Π is solvable in polynomial-time if and only if it is Cook-reducible to a trivial problem (e.g., deciding membership in the empty set).

Exercise 2.6 Show that Karp-reductions (and Levin-reductions) are transitive.

Exercise 2.7 Show that some decision problems are not Karp-reducible to their complement (e.g., the empty set is not Karp-reducible to $\{0,1\}^*$).
A popular exercise of dubious nature is to show that any decision problem in $\mathcal{P}$ is Karp-reducible to any *non-trivial* decision problem, where the decision problem regarding a set S is called non-trivial if $S \neq \emptyset$ and $S \neq \{0,1\}^*$. It follows that every non-trivial set in $\mathcal{P}$ is Karp-reducible to its complement.

Exercise 2.8 (reducing search problems to optimization problems) For every polynomially bounded relation R, present a polynomial-time computable function f such that the search problem of R is computationally equivalent to the search problem in which given (x, v) one has to find a $y \in \{0,1\}^{\mathrm{poly}(|x|)}$ such that $f(x, y) \geq v$.
(Hint: use a Boolean function.)

Exercise 2.9 (binary search) Show that using ℓ binary queries of the form "is $z \geq v$" it is possible to determine the value of an integer z that is a priori known to reside in the interval $[0, 2^\ell - 1]$.

Guideline: Consider a process that iteratively halves the interval in which z is known to reside in.

Exercise 2.10 Show that the standard search problem of Graph 3-Colorability is self-reducible, where this search problem consists of finding a 3-coloring for a given input graph.
(Hint: Iteratively extend the current prefix of a 3-coloring of the graph by making adequate oracle calls to the decision problem of Graph 3-Colorability. Specifically, encode the question

of whether or not $(\chi_1, ..., \chi_t) \in \{1,2,3\}^t$ is a prefix of a 3-coloring of the graph G as a query regarding the 3-colorability of an auxiliary graph G'.)[23]

Exercise 2.11 Show that the standard search problem of Graph Isomorphism is self-reducible, where this search problem consists of finding an isomorphism between a given pair of graphs.

(Hint: Iteratively extend the current prefix of an isomorphism between the two N-vertex graphs by making adequate oracle calls to the decision problem of Graph Isomorphism. Specifically, encode the question of whether or not $(\pi_1, ..., \pi_t) \in [N]^t$ is a prefix of an isomorphism between $G_1 = ([N], E_1)$ and $G_2 = ([N], E_2)$ as a query regarding isomorphism between two auxiliary graphs G_1' and G_2'.)[24]

Exercise 2.12 (NP problems that are not self-reducible) Assuming that $\mathcal{P} \neq \mathcal{NP}$, show that there exists a search problem R in $\mathcal{PC}$ that is not self-reducible (i.e., the search problem of R is not Cook-reducible to the decision problem S_R implicit in R). Prove that it follows that $S'_R = \{(x, y') : \exists y'' \text{ s.t. } (x, y'y'') \in R\}$ is not Cook-reducible to $S_R = \{x : \exists y \text{ s.t. } (x, y) \in R\}$. Furthermore, prove that deciding S'_R is not reducible to the search problem of R.

(Hint: Consider the relation $R = \{(x, 0x) : x \in \{0,1\}^*\} \cup \{(x, 1y) : (x, y) \in R'\}$, where R' is an arbitrary relation in $\mathcal{PC}$, and note that $S_R = \{0,1\}^*$.)

Exercise 2.13 In continuation to Exercise 2.12, present a natural search problem R in $\mathcal{PC}$ such that if factoring integers is intractable then S'_R is not reducible to S_R.

Guideline: Consider the relation R such that $(N, Q) \in R$ if the integer Q is a non-trivial divisor of the integer N. Use the fact that S_R is in $\mathcal{P}$, and show that deciding membership in S'_R is computationally equivalent to factoring integers.

Exercise 2.14 In continuation to Exercises 2.12 and 2.13, show that under suitable assumptions there exists relations $R_1, R_2 \in \mathcal{PC}$ having the same implicit-decision problem (i.e., $\{x : R_1(x) \neq \emptyset\} = \{x : R_2(x) \neq \emptyset\}$) such that R_1 is self-reducible but R_2 is not.

Exercise 2.15 Provide an alternative proof of Theorem 2.15 without referring to the set $S'_R = \{(x, y') : \exists y'' \text{ s.t. } (x, y'y'') \in R\}$. Hint: use Proposition 2.14.

Guideline: Reduce the search problem of R to the search problem of $R_{\texttt{SAT}}$, next reduce $R_{\texttt{SAT}}$ to $\texttt{SAT}$, and finally reduce $\texttt{SAT}$ to S_R. Justify the existence of each of these three reductions.

---

[23] Note that we merely need to check whether G has a 3-coloring in which the equalities and inequalities induced by $(\chi_1, ..., \chi_t)$ hold. This can be done by adequate gadgets (e.g., inequality is enforced by an edge between the corresponding vertices, whereas equality is enforced by an adequate subgraph that includes the relevant vertices as well as auxiliary vertices).

[24] This can be done by attaching adequate different gadgets to pairs of vertices that we wish to be mapped to one another (by the isomorphism). For example, we may connect the vertices in the i^{th} pair to an auxiliary star consisting of $(N + i)$ vertices.

Exercise 2.16 (additional properties of standard reductions) In continuation to the discussion in the main text, consider the following augmented form of Karp-reductions. Such a reduction of R to R' consists of three polynomial-time mappings (f, h, g) such that the following two conditions hold:

1. For every $(x, y) \in R$ it holds that $(f(x), h(x, y)) \in R'$.
2. For every $(f(x), y') \in R'$ it holds that $(x, g(x, y')) \in R$.

(We note that this definition is actually the one used by Levin in [142], except that he restricted h and g to only depend on their second argument.)
Prove that such a reduction implies both a Karp-reduction and a Levin-Reduction, and show that all reductions presented in this chapter satisfy this augmented requirement. Furthermore, prove that in all these cases the main mapping (i.e., f) is 1-1 and polynomial-time invertible.

Exercise 2.17 (parsimonious reductions) Let $R, R' \in \mathcal{PC}$ and let f be a Karp-reduction of $S_R = \{x : R(x) \neq \emptyset\}$ to $S_{R'} = \{x : R'(x) \neq \emptyset\}$. We say that g is parsimonious (with respect to R and R') if for every x it holds that $|R(x)| = |R'(g(x))|$. For each of the reductions, presented in this chapter, checked whether or not it is parsimonious. For the others, alternative reductions that are parsimonious can be found (cf. [81, Sec. 7.3]).

Exercise 2.18 Prove that `Bounded Halting` and `Bounded Non-Halting` are NP-complete, where the problems are defined as follows. The instance consists of a pair $(M, 1^t)$, where M is a Turing machine and t is an integer. The decision version of `Bounded Halting` (resp., `Bounded Non-Halting`) consists of determining whether or not there exists an input (of length at most t) on which M halts (resp., does *not* halt) in t steps, whereas the search problem consists of finding such an input.
(Hint: Either modify the proof of Theorem 2.18 or present a reduction of (say) the search problem of $R_{\mathbf{u}}$ to the search problem of Bounded (Non-)Halting. Indeed, the exercise is more straightforward in the case of Bounded Halting.)

Exercise 2.19 In the proof of Theorem 2.20, we claimed that the value of each entry in the "array of configurations" of a machine M is determined by the values of the three entries that reside in the row above it (as in Figure 2.1). Present a function $f_M : \Gamma^3 \to \Gamma$, where $\Gamma = \Sigma \times (Q \cup \{\perp\})$, that substantiates this claim.

Guideline: For example, for every $\sigma_1, \sigma_2, \sigma_3 \in Q$, it holds that $f_M((\sigma_1, \perp), (\sigma_2, \perp), (\sigma_3, \perp)) = (\sigma_2, \perp)$. More interestingly, if the transition function of M maps (σ, q) to $(\tau, p, +1)$ then, for every $\sigma_1, \sigma_2, \sigma_3 \in Q$, it holds that $f_M((\sigma, q), (\sigma_2, \perp), (\sigma_3, \perp)) = (\sigma_2, p)$ and $f_M((\sigma_1, \perp), (\sigma, q), (\sigma_3, \perp)) = (\tau, \perp)$.

Exercise 2.20 Present and analyze a reduction of `SAT` to `3SAT`.

Guideline: For a clause C, consider auxiliary variables such that the i^{th} variable indicates whether one of the first i literals is satisfied, and replace C by a 3CNF that uses the original variables of C as well as the auxiliary variables. For example, the clause $\vee_{i=1}^t x_i$ is replaced by the conjunction of 3CNFs that are logically equivalent to the formulae

$(y_2 \equiv (x_1 \vee x_2))$, $(y_i \equiv (y_{i-1} \vee x_i))$ for $i = 3, ..., t$, and y_t. We comment that this is not the standard reduction, but we find it more appealing conceptually.[25]

Exercise 2.21 (efficient solvability of 2SAT) In contrast to Exercise 2.20, prove that `2SAT` (i.e., the satisfiability of 2CNF formulae) is in $\mathcal{P}$.

Guideline: Consider the following "forcing process" for CNF formulae. If the formula contains a singleton clause (i.e., a clause having a single literal), then the corresponding variable is assigned the only value that satisfies the clause, and the formula is simplified accordingly (possibly yielding a constant, which is either `true` or `false`). The process is repeated until the formula is either a constant or contains only clauses of size at least 2. Clearly a formula ϕ is satisfiable if and only if the formula obtained from ϕ by the forcing process is satisfiable. The key fact (to be proved) is that a 2CNF formula is unsatisfiable if and only if there exists a variable such that any truth assignment to this variable yields a formula that the forcing process maps to the constant `false`.
(Extra hint: Applying the forcing process to a 2CNF formula we obtain a sub-formula of it; that is, each clause of the resulting formula is a clause (rather than a sub-clause) of the original formula.)

Exercise 2.22 The instance of the `Independent Set` problem consists of a pair (G, K), where G is a graph and K is an integer, and the question is whether or not the graph G contains an independent set (i.e., a set with no edges between its members) of size (at least) K. The `Clique` problem is analogous. Prove that both problems are computationally equivalent to the `Vertex Cover` problem.

Exercise 2.23 (Integer Linear Programming) Prove that the following problem is NP-complete. An instance of the problem is a systems of linear inequalities (say with integer constants), and the problem is to determine whether the system has an integer solution. For example, is there an integer solution to the system

$$\begin{array}{rcl} x + 2y - z & \geq & 3 \\ -3x - z & \geq & -5 \\ x & \geq & 0 \\ -x & \geq & -1 \end{array}$$

Guideline: Reduce from SAT. Specifically, consider an arithmetization of the input CNF by replacing $\vee$ with addition and $\neg x$ by $1-x$. Thus, each clause gives rize to an inequality (e.g., the clause $x \vee \neg y$ is replaced by the inequality $x + (1 - y) \geq 1$, which simplifies to $x - y \geq 2$). Enforce a 0-1 solution by introducing inequalities of the form $x \geq 0$ and $-x \geq -1$, for every variable x.

Exercise 2.24 (Maximum Satisfiability over GF(2)) Prove that the following problem is NP-complete. An instance of the problem consists of a systems of linear equations over GF(2) and an integer k, and the problem is to determine whether there exists an assignment that satisfies at least k equations. (Note that

[25] The standard reduction replaces the clause $\vee_{i=1}^{t} x_i$ by the conjunction of the 3CNFs $(x_1 \vee x_2 \vee z_2)$, $((\neg z_{i-1}) \vee x_i \vee z_i)$ for $i = 3, ..., t$, and $\neg z_t$.

the problem of determining whether there exists an assignment that satisfies all equations is in $\mathcal{P}$.)

Guideline: Reduce from 3SAT, using an arithetization similar to the one in Exercise 2.23. Specifically, replace each clause that contains $t \leq 3$ literals by 2^t linear GF(2) equations that correspond to the different subsets of these literals and assert that their sum (modulo 2) equals one; for example, the clause $x \vee \neg y$ is replaced by the equations $x+(1-y)=1$, $x=1$, $1-y=1$, and $0=1$. Note that if the original clause was satisfied by a value assignment $\overline{v}$ then exactly 2^{t-1} of the corresponding equations are satisfied by $\overline{v}$, whereas if the original clause was unsatisfied by $\overline{v}$ then none of the corresponding equations is satisfied by $\overline{v}$.

Exercise 2.25 Prove that a set S is Karp-reducible to some set in $\mathcal{NP}$ if and only if S is in $\mathcal{NP}$.
(Hint: For the non-trivial direction, see the proof of Proposition 2.32.)

Exercise 2.26 Recall that the empty set is not Karp-reducible to $\{0,1\}^*$, whereas any set is Cook-reducible to its complement. Thus our focus is on the Karp-reducibility of non-trivial sets to their complements, where a set is non-trivial if it is neither empty nor contains all strings.

1. Show that $\mathcal{P} \neq \mathcal{NP} = \text{co}\mathcal{NP}$ implies that some sets in $\mathcal{NP} \setminus \mathcal{P}$ are Karp-reducible to their complements.

   (Recall that any non-trivial set in $\mathcal{P}$ is Karp-reducible to its complement; see Exercise 2.7.)
2. Show that $\mathcal{NP} \neq \text{co}\mathcal{NP}$ implies that some (non-trivial) sets in $\mathcal{NP}$ cannot be Karp-reducible to their complements.

(Hint: Use NP-completeness in both parts, and Exercise 2.25 in the second part.)

Exercise 2.27 Referring to the proof of Theorem 2.27, prove that the function f is unbounded (i.e., for every i there exists an n such that n^3 steps of the process defined in the proof allow for failing the $i+1^{\text{st}}$ machine).

Guideline: Consider n' such that $f(n')=i$. Assuming, towards the contradiction that $f(n)=i$ for every $n>n'$, it follows that $F=F'' \cup F''$, where $F'=\{x : |x| \leq n' \wedge f(|x|) \equiv 0 \pmod 2\}$ is a finite set and $F''=\{x : |x|>n'\}$ if i is odd and $F''=\emptyset$ otherwise. In case i is odd, the $i+1^{\text{st}}$ machine tries to decide $S \cap F$ (which differs from S on finitely many strings), and must fail on some x. Derive a contradiction by showing that the number of steps that the process makes till reaching and considering the input x is at most $\exp(\text{poly}(|x|))$. A similar argument applies to the case that i is even, where we use the fact that F is finite and so the relevant reduction of S to $S \cap F$ must fail on some input x.

Exercise 2.28 Prove that if the promise problem Π is Cook-reducible to a promise problem that is solvable in polynomial-time, then Π is solvable in polynomial-time. Note that the solver may not halt on inputs that violate the promise.

Guideline: A polynomial-time solving a promise problem can be modified such that it halts on all inputs.

Exercise 2.29 (NP-complete promise problems in coNP (following [68])) Consider the promise problem `xSAT`, having instances that are pairs of CNF formulae. The yes-instances consists of pairs (ϕ_1, ϕ_2) such that ϕ_1 is satisfiable and ϕ_2 is unsatisfiable, whereas the no-instances consists of pairs such that ϕ_1 is unsatisfiable and ϕ_2 is satisfiable.

1. Show that `xSAT` is in the intersection of (the promise problem classes that are analogous to) $\mathcal{NP}$ and co$\mathcal{NP}$.

2. Prove that any promise problem in $\mathcal{NP}$ is Cook-reducible to `xSAT`. In designing the reduction, recall that queries that violate the promise may be answered arbitrarily.

   Guideline: Show a reduction of `SAT` to `xSAT`. Specifically, show that the search problem associated with `SAT` is Cook-reducible to `xSAT`, by following the ideas of the proof of Proposition 2.14. Actually, we need a more careful implementation of the search process. Suppose that we know (or assume) that τ is a prefix of a satisfying assignment to ϕ, and we wish to extend τ by one bit. Then, for each $\sigma \in \{0,1\}$, we construct a formula, denoted ϕ'_σ, by setting the first $|\tau|+1$ variables of ϕ according to the values $\tau\sigma$. We query the oracle about the pair (ϕ'_1, ϕ'_0), and extend τ accordingly (i.e., we extend τ by the value 1 if and only if the answer is positive). Note that if both ϕ'_1 and ϕ'_0 are satisfiable then it does not matter which bit we use in the extension, whereas if exactly one formula is satisfiable then the oracle answer is reliable.

3. Pinpoint the source of failure of the proof of Theorem 2.33 when applied to the reduction provided in the previous item.

Chapter 3

Variations on P and NP

In this chapter we consider variations on the complexity classes P and NP. We refer specifically to the non-uniform version of P, and to the Polynomial-time Hierarchy (which extends NP). These variations are motivated by relatively technical considerations, and the resulting classes are referred to quite frequently in the literature.

Summary: Non-uniform polynomial-time (P/poly) captures efficient computations that are carried out by devices that handle specific input lengths. The basic formalism ignore the complexity of constructing such devices (i.e., a uniformity condition). A finer formalism that allows to quantify the amount of non-uniformity refers to so called "machines that take advice."

The Polynomial-time Hierarchy (PH) generalizes NP by considering statements expressed by a quantified Boolean formula with a fixed number of alternations of existential and universal quantifiers. It is widely believed that each quantifier alternation adds expressive power to the class of such formulae.

The two different classes are related by showing that if NP is contained in P/poly then the Polynomial-time Hierarchy collapses to its second level. Assuming the latter collapse does not occur, this means that there is hope to resolve the P-vs-NP Question by showing that NP is not contained in "non-uniform P" (P/poly).

3.1 Non-uniform polynomial-time (P/poly)

In this section we consider two formulations of the notion of non-uniform polynomial-time, presenting two models of non-uniform (polynomial-time) computing devices. These models are derived by specializing the treatment of non-uniform computing devices, provided in Section 1.2.4, to the case of polynomially bounded complexities. It turns out that both models allow to solve the same class of computational problems, which is a strict superset of the class of problems solvable by polynomial-time algorithms.

The two models discussed below are the two models presented in Section 1.2.4 (i.e., Boolean circuits and "machines that take advice"). We will focus on the restriction of both models to the case of polynomial complexities, considering (non-uniform) polynomial-size circuits and polynomial-time algorithms that take (non-uniform) polynomially bounded advice.

The main motivation for considering non-uniform polynomial-size circuits is that their limitations imply analogous limitations on polynomial-time algorithms. The hope is that, as is often the case in mathematics and Science, disposing of an auxiliary condition that is not well-understood (i.e., uniformity) may turn out fruitful. In particular, it may facilitate a low-level analysis of the evolution of a computation, by using combinatorial techniques. This hope has materialized in the study of restricted classes of circuits (see Sections B.2.2 and B.2.3).

Polynomial-time algorithms that take polynomially bounded advice are useful in modeling auxiliary information available to possible efficient strategies that are of interest to us. Indeed, the typical cases are the modeling of adversaries in the context of cryptography and the modeling of arbitrary randomized algorithms in the context of derandomization. Furthermore, the model of polynomial-time algorithms that take advice allows for a quantitative study of the amount of non-uniformity, ranging from zero to polynomial.

3.1.1 Boolean Circuits

We refer the reader to §1.2.4.1 for a definition of (families of) Boolean circuits and the functions computed by them. For concreteness and simplicity, we assume throughout this section that all circuits has bounded fan-in. We highlight the following result stated in §1.2.4.1:

Theorem 3.1 (circuit evaluation): *There exists a polynomial-time algorithm that, given a circuit $C : \{0,1\}^n \to \{0,1\}^m$ and an n-bit long string x, returns $C(x)$.*

Recall that the algorithm works by performing the value-determining process described when defining the value of the circuit vertices on a given input.

Circuit size as a complexity measure. We recall the definitions of circuit complexity presented in to §1.2.4.1: The size of a circuit is defined as the number of edges, and the length of its description is almost linear in the latter; that is, a circuit of size s is commonly described by the list of its edges and the labels of its vertices, which means that its description length is $O(s \log s)$. We are interested in families of circuits that solve computational problems, and say that the circuit family $(C_n)_{n\in\mathbb{N}}$ computes the function $f : \{0,1\}^* \to \{0,1\}^*$ if for every $x \in \{0,1\}^*$ it holds that $C_{|x|}(x) = f(x)$. The size complexity of this family is the function $s : \mathbb{N} \to \mathbb{N}$ such that $s(n)$ is the size of C_n. The circuit complexity of a function f, denoted s_f, is the size complexity of the smallest family of circuits that computes f. An equivalent alternative follows.

Definition 3.2 (circuit complexity): *The* circuit complexity of $f : \{0,1\}^* \to \{0,1\}^*$ *is the function* $s_f : \mathbb{N} \to \mathbb{N}$ *such that* $s_f(n)$ *is the size of the smallest circuit that computes the restriction of* f *to* n*-bit strings.*

We stress that non-uniformity is implicit in this definition, because no conditions are made regarding the relation between the various circuits used to compute the function on different input lengths.

We will be interested in the class of problems that are solvable by families of polynomial-size circuits. That is, a problem is solvable by polynomial-size circuits if it can be solved by a function f that has polynomial circuit complexity (i.e., there exists a polynomial p such that $s_f(n) \le p(n)$, for every $n \in \mathbb{N}$).

A detour: uniform families. A family of *polynomial-size* circuits $(C_n)_n$ is called uniform if given n one can construct the circuit C_n in poly(n)-time. More generally:

Definition 3.3 (uniformity): *A family of circuits* $(C_n)_n$ *is called* uniform *if there exists an algorithm* A *that on input* n *outputs* C_n *within a number of steps that is polynomial in the size of* C_n.

We note that stronger notions of uniformity have been considered. For example, one may require the existence of a polynomial-time algorithm that on input n and v, returns the label of vertex v as well as the list of its children (or an indication that v is not a vertex in C_n). For further discussion see Section 5.2.3.

Proposition 3.4 *If a problem is solvable by a uniform family of polynomial-size circuits then it is solvable by a polynomial-time algorithm.*

As was hinted in §1.2.4.1, the converse holds as well. The latter fact follows easily from the proof of Theorem 2.20 (see also the proof of Theorem 3.6).

Proof: On input x, the algorithm operates in two stages. In the first stage, it invokes the algorithm guaranteed by uniformity condition, on input $n \stackrel{\text{def}}{=} |x|$, and obtains C_n. Next, it invokes the circuit evaluation algorithm (asserted in Theorem 3.1) on input C_n and x, and obtains $C_n(x)$. Since the size and the description length of C_n are polynomial in n, it follows that each stage of our algorithm runs in polynomial time (in $n = |x|$). Thus, the algorithm emulates the computation of $C_{|x|}(x)$, and does so in time polynomial in the length of its own input (i.e., x). ■

3.1.2 Machines that take advice

General (non-uniform) families of polynomial-size circuits and uniform families of polynomial-size circuits are two extremes with respect to the "amounts of non-uniformity" in the computing device. Intuitively, in the former, non-uniformity is only bounded by the size of the device, whereas in the latter the amounts of non-uniformity is zero. Here we consider a model that allows to decouple the size of the computing device from the amount of non-uniformity, which may indeed range from zero to the device's size. Specifically, we consider algorithms that "take a non-uniform advice" that depends only on the input length. The amount of non-uniformity will be defined to equal the length of the corresponding advice (as a function of the input length). Here we specialize Definition 1.12 to the case of polynomial-time algorithms.

Definition 3.5 (non-uniform polynomial-time and $\mathcal{P}/\text{poly}$): *We say that a function f is* computed in polynomial-time with advice of length $\ell : \mathbb{N} \rightarrow \mathbb{N}$ *if these exists a polynomial-time algorithm A and an infinite* advice sequence $(a_n)_{n\in\mathbb{N}}$ *such that*

1. *For every $x \in \{0,1\}^*$, it holds that $A(a_{|x|}, x) = f(x)$.*
2. *For every $n \in \mathbb{N}$, it holds that $|a_n| = \ell(n)$.*

We say that a computational problem can be solved in polynomial-time with advice of length ℓ if a function solving this problem can be computed within these resources. We denote by $\mathcal{P}/\ell$ the class of decision problems that can be solved in polynomial-time with advice of length ℓ, and by $\mathcal{P}/\text{poly}$ the union of $\mathcal{P}/p$ taken over all polynomials p.

Clearly, $\mathcal{P}/0 = \mathcal{P}$. But allowing some (non-empty) advice increases the power of the class (as we shall see below), and allowing advice of length comparable to the time complexity yields a formulation equivalent to circuit complexity (see Theorem 3.6). We highlight the greater flexibility available by the formalism of machines that take advice, which allows for separate specification of time complexity and advice length. (This comes at the expense of a more cumbersome formulation when we wish to focus on the case that both measures are equal.)

Relation to families of polynomial-size circuits. As hinted before, the class of problems solvable by polynomial-time algorithms with polynomially bounded advice equals the class of problems solvable by families of polynomial-size circuits. For concreteness, we state this fact for decision problems.

Theorem 3.6 *A decision problem is in $\mathcal{P}/\text{poly}$ if and only if it can solved by a family of polynomial-size circuits.*

More generally, for any function t, the following proof establishes that equivalence of the power of machines having time complexity t and taking advice of length t versus families of circuits of size polynomially related to t.

Proof: Suppose that a problem can be solved by a polynomial-time algorithm A using the polynomially bounded advice sequence $(a_n)_{n\in\mathbb{N}}$. Adapting the proof of Theorem 2.20, observe that the computation of $A(a_{|x|}, x)$ can be emulated by a circuit of $\text{poly}(|x|)$-size, which incorporates $a_{|x|}$ and is given x as input. That is, we construct a circuit C_n such that $C_n(x) = A(a_n, x)$ holds for every $x \in \{0,1\}^n$ (analogously to the way C_x was constructed in the proof of Theorem 2.20, where $C_x(y) = M_R(x, y)$).

On the other hand, given a family of polynomial-size circuits, we obtain a polynomial-time algorithm for emulating this family using advice that provide the description of the relevant circuits. Specifically, we use the evaluation algorithm asserted in Theorem 3.1, while using the circuit description as advice. We use the fact that a circuit of size s can be computed using advice of length $O(s \log s)$, where the log factor is due to the fact that a graph with v vertices and e edges can be described by a string of length $2e \log_2 v$. ■

Another perspective. A set S is called sparse if there exists a polynomial p such that for every n it holds that $|S \cap \{0,1\}^n| \leq p(n)$. We note that $\mathcal{P}/\mathrm{poly}$ equals the class of sets that are Cook-reducible to a sparse set (see Exercise 3.1). Thus, SAT is Cook-reducible to a sparse set if and only if $\mathcal{NP} \subset \mathcal{P}/\mathrm{poly}$. In contrast, SAT is Karp-reducible to a sparse set if and only if $\mathcal{NP} = \mathcal{P}$ (see Exercise 3.10).

The power of $\mathcal{P}/\mathrm{poly}$. In continuation to Theorem 1.13, we prove the following result.

Theorem 3.7 (the power of advice, revisited): *The class $\mathcal{P}/1 \subseteq \mathcal{P}/\mathrm{poly}$ contains $\mathcal{P}$ as well as some undecidable problems.*

Actually, $\mathcal{P}/1 \subset \mathcal{P}/\mathrm{poly}$. Furthermore, by using a counting argument, one can show that for any two polynomially bounded functions $\ell_1, \ell_2 : \mathbb{N} \to \mathbb{N}$ such that $\ell_2 - \ell_1 > 0$ is unbounded, it holds that $\mathcal{P}/\ell_1$ is strictly contained in $\mathcal{P}/\ell_2$; see Exercise 3.2.

Proof: Clearly, $\mathcal{P} = \mathcal{P}/0 \subseteq \mathcal{P}/1 \subseteq \mathcal{P}/\mathrm{poly}$. To prove that $\mathcal{P}/1$ contains some undecidable problems, we review the proof of Theorem 1.13. The latter proof established the existence of uncomputable Boolean function that only depend on their input length. That is, there exists an undecidable set $S \subset \{0,1\}^*$ such that for every pair of equal length strings (x, y) it holds that $x \in S$ if and only if $y \in S$. In other words, for every $x \in \{0,1\}^*$ it holds that $x \in S$ if and only if $1^{|x|} \in S$. But such a set is easily decidable in polynomial-time by a machine that takes one bit of advice; that is, consider the algorithm A and the advice sequence $(a_n)_{n \in \mathbb{N}}$ such that $a_n = 1$ if and only if $1^n \in S$ and $A(a, x) = a$ (for $a \in \{0,1\}$ and $x \in \{0,1\}^*$). Note that indeed $A(a_{|x|}, x) = 1$ if and only if $x \in S$. ■

3.2 The Polynomial-time Hierarchy (PH)

We start with an informal motivating discussion, which will be made formal in Section 3.2.1.

Sets in $\mathcal{NP}$ can be viewed as sets of valid assertions that can be expressed as quantified Boolean formulae using only existential quantifiers. That is, a set S is in $\mathcal{NP}$ if there is a Karp-reduction of S to the problem of deciding whether or not an existentially quantified Boolean formula is valid (i.e., an instance x is mapped by this reduction to a formula of the form $\exists y_1 \cdots \exists y_{m(x)} \phi_x(y_1, ..., y_{m(x)})$).

The conjectured intractability of $\mathcal{NP}$ seems due to the long sequence of existential quantifiers. Of course, if somebody else (i.e., a "prover") were to provide us with an adequate assignment (of the y_i's) whenever such an assignment exists then we would be in good shape. That is, we can efficiently verify proofs of validity of existentially quantified Boolean formulae.

But what if we want to verify the validity of a universally quantified Boolean formulae (i.e., formulae of the form $\forall y_1 \cdots \forall y_m \phi(y_1, ..., y_m)$). Here we seem to need the help of a totally different entity: we need a "refuter" that is guaranteed to provide us with a refutation whenever such exist, and we need to believe that if we were not presented with such a refutation then it is the case that no refutation exists (and hence the universally quantified formulae is valid). Indeed, this new setting (of a "refutation system") is fundamentally different from the setting of a proof system: In a proof system we are only convinced by proofs (to assertions) that we have verified by ourselves, whereas in the "refutation system" we trust the "refuter" to provide evidence against false assertions.[1] Furthermore, there seems to be no way of converting one setting (e.g., the proof system) into another (resp., the refutation system).

Taking an additional step, we may consider a more complicated system in which we use two agents: a "supporter" that tries to provide evidence in favor of an assertion and an "objector" that tries to refute it. These two agents conduct a debate (or an argument) in our presence, exchanging messages with the goal of making us (the referee) rule their way. The assertions that can be proven in this system take the form of general quantified formulae with alternating sequences of quantifiers, where the number of alternations equals the number of rounds of interaction in the said system. We stress that the exact length of each sequence of quantifiers of the same type does not matter, what matters is the number of alternations, denoted k.

The aforementioned system of alternations can be viewed as a two-party game, and we may ask ourselves which of the two parties has a k-move winning strategy. In general, we may consider any (0-1 zero-sum) two-party game, in which the game's position can be efficiently updated (by any given move) and evaluated. For such a fixed game, given an initial position, we may ask whether the first party has a (k-move) winning strategy. It seems that answering this question for some k does not help in answering it for $k+1$. We now turn to formalize the foregoing discussion.

3.2.1 Alternation of quantifiers

In the following definition, the aforementioned propositional formula ϕ_x is replaced by the input x itself. (Correspondingly, the combination of the Karp-reduction and a formula evaluation algorithm are replaced by the verification algorithm V (see Exercise 3.5).) This is done in order to make the comparison to the definition

[1] More formally, in proof systems the soundness condition relies only on the actions of the verifier, whereas completeness also relies on the prover using an adequate strategy. In contrast, in "refutation system" the soundness condition relies on the proper actions of the refuter, whereas completeness does not depend on the refuter's actions.

of $\mathcal{NP}$ more transparent (as well as to fit the standard presentations). We also replace a sequence of Boolean quantifiers of the same type by a single corresponding quantifier that quantifies over a string of corresponding length.

Definition 3.8 (the class Σ_k): *For a natural number k, a decision problem $S \subseteq \{0,1\}^*$ is in Σ_k if there exists a polynomial p and a polynomial time algorithm V such that $x \in S$ if and only if*

$$\exists y_1 \in \{0,1\}^{p(|x|)} \forall y_2 \in \{0,1\}^{p(|x|)} \exists y_3 \in \{0,1\}^{p(|x|)} \cdots Q_k y_k \in \{0,1\}^{p(|x|)} \quad \text{s.t. } V(x, y_1, ..., y_k) = 1$$

where Q_k is an existential quantifier if k is odd and is a universal quantifier otherwise.

Note that $\Sigma_1 = \mathcal{NP}$ and $\Sigma_0 = \mathcal{P}$. The Polynomial-time Hierarchy, denoted $\mathcal{PH}$, is the union of all the aforementioned classes (i.e., $\mathcal{PH} = \cup_k \Sigma_k$), and Σ_k is often referred to as the k^{th} level of $\mathcal{PH}$. The levels of the Polynomial-time Hierarchy can also be defined inductively, by defining Σ_{k+1} based on $\Pi_k \stackrel{\text{def}}{=} \text{co}\Sigma_k \stackrel{\text{def}}{=} \{\{0,1\}^* \setminus S : S \in \Sigma_k\}$.

Proposition 3.9 *The set S is in Σ_{k+1} if and only if there exists a polynomial p and a set $S' \in \Pi_k$ such that $S = \{x : \exists y \in \{0,1\}^{p(|x|)} \text{ s.t. } (x,y) \in S'\}$.*

Proof: Suppose that S is in Σ_{k+1} and let p and V be as in Definition 3.8. Then define S' as the set of pairs (x,y) such that $|y| = p(|x|)$ and

$$\forall y_1 \in \{0,1\}^{p(|x|)} \exists y_2 \in \{0,1\}^{p(|x|)} \cdots Q_k y_k \in \{0,1\}^{p(|x|)} \text{ s.t. } V(x, y, y_1, ..., y_k) = 1 .$$

Note that $x \in S$ if and only if there exists $y \in \{0,1\}^{p(|x|)}$ such that $(x,y) \in S'$, and that $S' \in \Pi_k$ (by replacing V with $1 - V$; see Exercise 3.4).

On the other hand, suppose that for some polynomial p and a set $S' \in \Pi_k$ it holds that $S = \{x : \exists y \in \{0,1\}^{p(|x|)} \text{ s.t. } (x,y) \in S'\}$. Then, for some p' and V', it holds that $(x,y) \in S'$ if and only if $|y| = p(|x|)$ and

$$\forall y_1 \in \{0,1\}^{p'(|x|)} \exists y_2 \in \{0,1\}^{p'(|x|)} \cdots Q_k y_k \in \{0,1\}^{p'(|x|)} \text{ s.t. } V'(x, y, y_1, ..., y_k) \neq 1$$

(see Exercise 3.4 again). By suitable encoding (of y and the y_i's as strings of length $\max(p(|x|), p'(|x|))$) and a trivial modification of V', we conclude that $S \in \Sigma_{k+1}$. ■

Determining the winner in k-move games. Definition 3.8 can be interpreted as capturing the complexity of determining the winner in certain *efficient two-party game*. Specifically, we refer to two-party games that satisfy the following three conditions:

1. The parties alternate in taking moves that effect the game's (global) position, where each move has a description length that is bounded by a polynomial in the length of the current position.

2. The current position can be updated in polynomial-time based on the previous position and the current party's move.[2]

3. The winner in each position can be determined in polynomial-time.

A set $S \in \Sigma_k$ can be viewed as the set of initial positions (in a suitable game) for which the first party has a k-move winning strategy. Specifically, $x \in S$ if starting at the initial position x, there exists move y_1 for the first party, such that for every response move y_2 of the second party, there exists move y_3 for the first party, etc, such that after k moves the parties reach a position in which the first party wins, where the final position as well as which party wins in it are determined by the predicate V (in Definition 3.8). That is, $V(x, y_1, ..., y_k) = 1$ if the position that is reached when starting from position x and taking the move sequence $y_1, ..., y_k$ is a winning position for the first party.

The collapsing effect of some equalities. Extending the intuition behind the $\mathcal{NP} \neq \text{co}\mathcal{NP}$ conjecture, it is conjectured that $\Sigma_k \neq \Pi_k$ for every $k \in \mathbb{N}$. The failure of this conjecture causes the collapse of the Polynomial-time Hierarchy to the corresponding level.

Proposition 3.10 *For every $k \geq 1$, if $\Sigma_k = \Pi_k$ then $\Sigma_{k+1} = \Sigma_k$, which in turn implies $\mathcal{PH} = \Sigma_k$.*

The converse also holds (i.e., $\mathcal{PH} = \Sigma_k$ implies $\Sigma_{k+1} = \Sigma_k$ and $\Sigma_k = \Pi_k$).

Proof: Assuming that $\Sigma_k = \Pi_k$, we consider any set S in Σ_{k+1}. By Proposition 3.9, there exists a polynomial p and a set $S' \in \Pi_k$ such that $S = \{x : \exists y \in \{0,1\}^{p(|x|)} \text{ s.t. } (x, y) \in S'\}$. Using the hypothesis, we infer that $S' \in \Sigma_k$, and it follows that $S \in \Sigma_k$ (by collapsing two adjacent existential quantifiers (relying on $k \geq 1$)).[3] We also note that $\Sigma_{k+1} = \Sigma_k$ (or, equivalently, $\Pi_{k+1} = \Pi_k$) implies $\Sigma_{k+2} = \Sigma_{k+1}$ (again by using Proposition 3.9), and thus $\Sigma_{k+1} = \Sigma_k$ implies $\mathcal{PH} = \Sigma_k$. ■

Decision problems that are Cook-reductions to NP. Recall that in Section 2.2.2 we defined two types of optimization problems and showed that under some natural conditions these two types are computationally equivalent (under Cook reductions). Specifically, one type of problems referred to finding a solution that have a value exceeding some given threshold, whereas the second type called for finding optimal solutions. In Section 2.3 we presented several problems of the first type, and proved that they are NP-complete. We note that corresponding

[2]Note that, since we consider a constant number of moves, the length of all possible final positions is bounded by a polynomial in the length of the initial position, and thus all items have an equivalent form in which one refers to the complexity as a function of the length of the initial position. The latter form allows for a smooth generalization to games with a polynomial number of moves (as in Section 5.4), where it is essential to state all complexities in terms of the length of the initial position.

[3]Needless to say, Proposition 3.10 does not seem to hold for $k = 0$.

versions of the second type are believed not to be in NP. For example, we discussed the problem of deciding whether or not a given graph G has a clique of a given size K, and showed that it is NP-complete. In contract, the problem of deciding whether or not K is the maximum clique size of the graph G is not known (and quite unlikely) to be in $\mathcal{NP}$, although it is Cook-reducible to $\mathcal{NP}$. Thus, the class of decision problems that are Cook-reducible to $\mathcal{NP}$ contains many natural problems that are unlikely to be in $\mathcal{NP}$. The Polynomial-time Hierarchy contains all these problems, because Σ_2 contains all problems that are Cook-reductions to $\mathcal{NP}$ (see Exercise 3.3).

Complete problems and a relation to AC0. We note that quantified Boolean formulae with a bounded number of quantifier alternation provide complete problems for the various levels of the Polynomial-time Hierarchy (see Exercise 3.5). We also note the correspondnace between these formulae and (highly uniform) constant-depth circuits of unbounded fan-in that get as input the truth-table of the underlying (quantifier-free) formula (see Exercise 3.6).

3.2.2 Non-deterministic oracle machines

The Polynomial-time Hierarchy is commonly defined in terms of non-deterministic polynomial-time (oracle) machines that are given oracle access to a set of the lower level of the same hierarchy. Such machines are defined by combining the definitions of non-deterministic (polynomial-time) machines (cf. Definition 2.7) and oracle machines (cf. Definition 1.11). Specifically, for an oracle $f : \{0,1\}^* \to \{0,1\}^*$, a non-deterministic machine M, and a string x, one considers the question of whether or not there exists an accepting (non-deterministic) computation of M on input x and access to the oracle f. The class of sets that can be accepted by non-deterministic polynomial-time (oracle) machines with access to f is denoted $\mathcal{NP}^f$. (We note that this notation makes sense because we can associate the class $\mathcal{NP}$ with a collection of machines that lends itself to be extended to oracle machines.) For any class of decision problems $\mathcal{C}$, we denote by $\mathcal{NP}^{\mathcal{C}}$ the union of $\mathcal{NP}^f$ taken over all decision problems f in $\mathcal{C}$. The following result provides an alternative definition of the Polynomial-time Hierarchy.

Proposition 3.11 *For every $k \geq 1$, it holds that $\Sigma_{k+1} = \mathcal{NP}^{\Sigma_k}$.*

Proof: The first direction is straightforward: For any $S \in \Sigma_{k+1}$, let $S' \in \Pi_k$ and p be as in Proposition 3.9; that is, $S = \{x : \exists y \in \{0,1\}^{p(|x|)} \text{ s.t. } (x,y) \in S'\}$. Consider a non-deterministic machine that, on input x, non-deterministically generates $y \in \{0,1\}^{p(|x|)}$ and accepts if and only if $(x,y) \in S'$. Thus, $S \in \mathcal{NP}^{\Pi_k} = \mathcal{NP}^{\Sigma_k}$, where equality holds by letting the oracle machine flip each (binary) answer that is provided by the oracle.[4]

[4]Do not get confused by the fact that the class of oracles may *not* be closed under complementation. From the point of view of the oracle machine, the oracle is merely a function, and the machine may do with its answer whatever it pleases (and in particular negate it).

For the opposite direction, suppose that the non-deterministic polynomial-time oracle machine M accepts S when given oracle access to $S' \in \Sigma_k$. Note that (unlike the machine constructed in the foregoing argument) machine M may issue several queries to S', and these queries may be determined based on previous oracle answers. To simplify the argument, we assume, without loss of generality, that at the very beginning of its execution M guesses (non-deterministic) all oracle answers and accepts only if the actual answers match its guesses. Thus, M's queries to the oracle are determined by its input, denoted x, and its non-deterministic choices, denoted y. We denote by $q^{(i)}(x,y)$ the $i^{\rm th}$ query made by M (on input x and non-deterministic choices y), and by $a^{(i)}(x,y)$ the corresponding (a priori) guessed answer (which is a bit in y). Thus, M accepts x if and only if there exists $y \in \{0,1\}^{\mathrm{poly}(|x|)}$ such that the following two conditions hold:

1. Machine M accepts x, on input x and non-deterministic choices y, provided that all oracle answers fit the correspondingly guessed answers $a^{(i)}(x,y)$.

2. All oracle answers fit the correspondingly guessed answers $a^{(i)}(x,y)$; that is, $a^{(i)}(x,y)=1$ if and only if $q^{(i)}(x,y) \in S'$, for every $i=1,...,q(x,y)$, where $q(x,y)=\mathrm{poly}(|x|)$ is the number of queries made by M.

Denoting the first event by $A(x,y)$, it follows that $x \in S$ if and only if

$$\exists y \left(A(x,y) \wedge \bigwedge_{i=1}^{q(x,y)} \left((a^{(i)}(x,y)=1) \Leftrightarrow (q^{(i)}(x,y) \in S') \right) \right).$$

Denoting the verification algorithm of S' by V', it holds that $x \in S$ if and only if

$$\exists y \left(A(x,y) \wedge \bigwedge_{i=1}^{q(x,y)} \left((a^{(i)}(x,y)=1) \Leftrightarrow \exists y_1^{(i)} \forall y_2^{(i)} \cdots Q_k y_k^{(i)}\, V'(q^{(i)}(x,y), y_1^{(i)}, ..., y_k^{(i)})=1 \right) \right).$$

The proof is completed by observing that the foregoing expression can be rearranged to fit the definition of Σ_{k+1}. Details follow.

Starting with the foregoing expression, we first pull all quantifiers outside, and obtain a quantified expression with $k+1$ alternations, starting with an existential quantifier.[5] (We get $k+1$ alternations rather than k, because $a^{(i)}(x,y)=0$ introduces an expression of the form $\neg \exists y_1^{(i)} \forall y_2^{(i)} \cdots Q_k y_k^{(i)}\, V'(q^{(i)}(x,y), y_1^{(i)}, ..., y_k^{(i)})=1$, which in turn is equivalent to the expression $\forall y_1^{(i)} \exists y_2^{(i)} \cdots \overline{Q}_k y_k^{(i)}\, \neg V'(q^{(i)}(x,y), y_1^{(i)}, ..., y_k^{(i)})=1$).) Once this is done, we may incorporate the computation of all the $q^{(i)}(x,y)$'s (and $a^{(i)}(x,y)$'s) as well as the polynomial number of invocations of V' (and other logical operations) into the new verification algorithm V. It follows that $S \in \Sigma_{k+1}$. ■

[5]For example, note that for predicates P_1 and P_2, the expression $\exists y\,(P_1(y) \Leftrightarrow \exists z\, P_2(y,z))$ is equivalent to the expression $\exists y\,((P_1(y) \wedge \exists z\, P_2(y,z)) \vee ((\neg P_1(y) \wedge \neg \exists z\, P_2(y,z)))$, which in turn is equivalent to the expression $\exists y \exists z' \forall z''\,((P_1(y) \wedge P_2(y,z')) \vee ((\neg P_1(y) \wedge \neg P_2(y,z'')))$. Note that pulling the quantifiers outside in $\wedge_{i=1}^t \exists y^{(i)} \forall z^{(i)} P(y^{(i)}, z^{(i)})$ yields an expression of the type $\exists y^{(1)},...,y^{(t)} \forall z^{(1)},...,z^{(t)} \wedge_{i=1}^t P(y^{(i)}, z^{(i)})$.

A general perspective – what does ${\mathcal{C}_1}^{\mathcal{C}_2}$ mean? By the foregoing discussion it should be clear that the class ${\mathcal{C}_1}^{\mathcal{C}_2}$ can be defined for two complexity classes $\mathcal{C}_1$ and $\mathcal{C}_2$, *provided that $\mathcal{C}_1$ is associated with a class of machines that extends naturally in a way that allows for oracle access.* Actually, the class ${\mathcal{C}_1}^{\mathcal{C}_2}$ *is not defined based on the class $\mathcal{C}_1$ but rather by analogy to it.* Specifically, suppose that $\mathcal{C}_1$ is the class of sets recognizable by machines of certain type (e.g., deterministic or non-deterministic) with certain resource bounds (e.g., time and/or space bounds). Then, we consider analogous oracle machines (i.e., of the same type and with the same resource bounds), and say that $S \in {\mathcal{C}_1}^{\mathcal{C}_2}$ if there exists an adequate oracle machine M_1 (i.e., of this type and resource bounds) and a set $S_2 \in \mathcal{C}_2$ such that $M_1^{S_2}$ accepts the set S.

Decision problems that are Cook-reductions to NP, revisited. Using the foregoing notation, the class of decision problems that are Cook-reductions to $\mathcal{NP}$ is denoted $\mathcal{P}^{\mathcal{NP}}$, and thus is a subset of $\mathcal{NP}^{\mathcal{NP}} = \Sigma_2$ (see Exercise 3.7). In contrast, recall that the class of decision problems that are Karp-reductions to $\mathcal{NP}$ equals $\mathcal{NP}$.

3.2.3 The P/poly-versus-NP Question and PH

As stated in Section 3.1, a main motivation for the definition of $\mathcal{P}$/poly is the hope that it can serve to separate $\mathcal{P}$ from $\mathcal{NP}$ (by showing that $\mathcal{NP}$ is not even contained in $\mathcal{P}$/poly, which is a (strict) superset of $\mathcal{P}$). In light of the fact that $\mathcal{P}$/poly extends far beyond $\mathcal{P}$ (and in particular contains undecidable problems), one may wonder if this approach does not run the risk of $\mathcal{NP}$ being in $\mathcal{P}$/poly (even if $\mathcal{P} \neq \mathcal{NP}$). Ideally, we would like to know that $\mathcal{NP} \subset \mathcal{P}$/poly may occur only if $\mathcal{P} = \mathcal{NP}$ (which means that the Polynomial-time Hierarchy collapses to its zero level). The following result may seem to get close, showing that $\mathcal{NP} \subset \mathcal{P}$/poly may occur only if the Polynomial-time Hierarchy collapses to its second level.

Theorem 3.12 *If $\mathcal{NP} \subset \mathcal{P}$/poly then $\Sigma_2 = \Pi_2$.*

Recall that $\Sigma_2 = \Pi_2$ implies $\mathcal{PH} = \Sigma_2$ (see Proposition 3.10). Thus, an unexpected behavior of the non-uniform complexity class $\mathcal{P}$/poly implies an unexpected behavior of in the world of uniform complexity (i.e., the ability to reduce any constant number of quantifier alternations into two).

Teaching note: We present the proof in terms of machines that take advice, although a presentation in terms of circuits may be somewhat less cumbersome (i.e., may use less symbols). We note that a presentation in terms of circuits tends to confuse some students, becuase these circuits appear both as strings (describing these circuits) and as computing devices (applied to other strings).

Proof: Starting with an arbitrary set $S \in \Pi_2$, we shall show that $S \in \Sigma_2$. By Proposition 3.9, there exists a polynomial p and a set $S' \in \mathcal{NP}$ such that $S = \{x : \forall y \in \{0,1\}^{p(|x|)}\ (x, y) \in S'\}$. Using the theorem's hypothesis, it follows that $S' \in \mathcal{P}/\text{poly}$, which means that there exists polynomially bounded advice that allow for deciding S' in polynomial-time. That is, there exists a polynomial-time algorithm A and an advice sequence $(a_n)_{n\in\mathbb{N}}$ such that $|a_n| = \text{poly}(n)$ and $A(a_n, \cdot)$ correctly decides the membership of n-bit long strings in S'. Thus, $x \in S$ if and only if there exists a "correct advice" $a \in \{0,1\}^{\text{poly}(|x|)}$ for $(|x| + p(|x|))$-bit long inputs such that for every $y \in \{0,1\}^{p(|x|)}$ it holds that $A(a,(x,y)) = 1$, where the correctness condition means that $A(a,\cdot)$ decides correctly the membership of $(|x| + p(|x|))$-bit long strings in S'. Note that the aforementioned conditions have the right form as they assert that *there exists a string a such that for all y and z some condition holds* (i.e., $A(a,(x,y)) = 1$ and $A(a,z) = \chi_{S'}(z)$, where $\chi_{S'}(z) = 1$ if $z \in S'$ and $\chi_{S'}(z) = 0$ otherwise).

The problem is that part of the latter condition (i.e., $A(a,z) = \chi_{S'}(z)$) is not checkable in polynomial-time. Specifically, verifying $A(a,z) = \chi_{S'}(z)$ requires the ability to decide membership in S', which is an arbitrary set in $\mathcal{NP}$. Still, we shall see that in some cases the condition $\forall z A(a,z) = \chi_{S'}(z)$ can be replaced by a condition of the form $\forall z B(a,z)$, where B is a polynomial-time computable predicate.

Suppose, for a moment, that S' is downwards self-reducible; that is, that deciding whether $z \in S'$ can be reduced to deciding membership in S' of shorter (than z) strings. Then, referring to the aforementioned algorithm A and polynomial p, we revise (or rather augment) the foregoing suggestion and assert that $x \in S$ if and only if there exists a sequence of poly$(|x|)$-bit long strings $a_1, ..., a_m$, where $m = (|x| + p(|x|))$, such that the following two conditions hold

1. For every $y \in \{0,1\}^{p(|x|)}$, it holds that $A(a_m,(x,y)) = 1$.

2. For $i = 1, ..., m$, for every $z \in \{0,1\}^i$ it holds that $A(a_i, z) = 1$ if and only if $z \in S'$. The point is that this condition can be expressed by a condition of the form for every $i = 1, ..., m$ and $z \in \{0,1\}^i$ it holds that $B(a_1, ..., a_i, z) = 1$, where B is a polynomial-time computable predicate.

   Specifically, using downwards self-reducible, we may obtain the value of $\chi_{S'}(z)$ from the values of $A(a_j, z')$ for some $j < i$ and $z' \in \{0,1\}^j$. Thus, we check the value of $A(a_i, z)$ against the value obtained by reducing the question of whether $z \in S'$ to the membership of shorter strings in S', where the latter decisions are made by relying on the hypothesis that we have correct advice for shorter strings. (Further details are given later.)

   In summary, we verify the correctness of the advice a_i, by relying on the correctness of the advice $a_1, ..., a_{i-1}$. (Needless to say, the correctness of the advice a_1 can be verified in a constant number of steps by using a brute force algorithm for deciding membership in S'.)

There is a minor problem with this revised suggestion: we have assumed that S' is downwards self-reducible, while S' may be an arbitrary set in $\mathcal{NP}$. The solution is to reduce S' to SAT, and rely on the fact that SAT is downwards self-reducible. The latter fact is implicit in the proof of Proposition 2.14; that is, $\phi \in$ SAT if and only if either $\phi_{x_1=0} \in$ SAT or $\phi_{x_1=1} \in$ SAT, where $\phi_{x_1=\sigma}$ denotes the formula obtained from ϕ by setting the first variable (denoted x_1) to σ and simplifying the formula in the obvious way (thus reducing its length).

For simplicity, in the rest of the proof, we rely on the fact that S' is Karp-reducible to SAT (rather than only Cook-reducible to it). Specifically, let f be a Karp-reduction of S' to SAT. Thus, $x \in S$ if and only if $\forall y \in \{0,1\}^{p(|x|)}\ \phi_{x,y} \in$ SAT, where $\phi_{x,y} \stackrel{\text{def}}{=} f(x,y)$. Using the hypothesis, we have SAT $\in \mathcal{P}$/poly, and thus there exists a polynomial-time algorithm A that solves SAT using advice of polynomial length. Now, we assert that $x \in S$ if and only if there exists a sequence of poly($|x|$)-bit long strings $a_1, ..., a_m$, where $m = \text{poly}(|x|)$ satisfies $m \geq \max_{y \in \{0,1\}^{p(|x|)}}\{|f(x,y)|\}$, such that the following two conditions hold:

1. For every $y \in \{0,1\}^{p(|x|)}$, it holds that $A(a_{|f(x,y)|}, f(x,y)) = 1$.
2. For $i = 1, ..., m$, for every $\phi \in \{0,1\}^i$ it holds that $A(a_i, \phi) = 1$ if and only if $\phi \in$ SAT. This condition is checked as follows.
   (a) For every (non-empty) formula ϕ, we denote by ϕ_σ the formula resulting from ϕ by setting its first variable to σ and making the obvious simplifications (i.e., omitting constants from clauses and omitting empty clauses). Thus, $|\phi_\sigma| < |\phi|$.
   (b) For every $i > 1$ and $\phi \in \{0,1\}^i$, we check that $A(a_i, \phi) = 1$ if and only if either $A(a_{|\phi_0|}, \phi_0) = 1$ or $A(a_{|\phi_1|}, \phi_1) = 1$.
   For $\phi \in \{0,1\}^1$, we check that $A(a_1, \phi) = 1$ if and only if ϕ is satisfiable (which can be determined in constant time). (Indeed, for any threshold $t = O(\log|x|)$, we could have distinguished the cases $i > t$ and $i \leq t$, where the latter can be handled in time poly(2^t).)

   To summarize, we determine whether or not ϕ is in SAT by computing $A(a_{|\phi_0|}, \phi_0)$ and $A(a_{|\phi_1|}, \phi_1)$. Thus, for $i > 1$, the condition checked is

$$\forall \phi \in \{0,1\}^i\ A(a_i, \phi) = \vee_{\sigma \in \{0,1\}} A(a_{|\phi_\sigma|}, \phi_\sigma).$$

Note that the correctness of the various advice follows by induction. That is, if the advice $a_1, ..., a_{i-1}$ are correct (for input lengths 1 through $i-1$) and, for $\phi^{(i)} \in \{0,1\}^i$, it holds that $A(a_i, \phi^{(i)}) = \vee_\sigma A(a_{|\phi_\sigma^{(i)}|}, \phi_\sigma^{(i)})$ then the advice a_i is correct (for input length i).

Finally, observe that the expression obtained for membership in S (i.e., $x \in S$ if and only if $\exists a_1, ..., a_m \forall y, \phi^{(1)}, ..., \phi^{(m)}$ it holds that $A(a_{|f(x,y)|}, f(x,y)) = 1$ and for $i = 2, ..., m$ $A(a_i, \phi^{(i)}) = \vee_\sigma A(a_{|\phi_\sigma^{(i)}|}, \phi_\sigma^{(i)})$ (and $A(a_1, \phi^{(1)})$ is correct)) is indeed of the Σ_2-form. Thus, $S \in \Sigma_2$ and it follows that $\Pi_2 \subseteq \Sigma_2$. Using Exercise 3.7, the theorem follows. ■

A technical perspective. The proof of Theorem 3.12 implies that *any set* $S' \in \mathcal{P}$/poly *that is downwards self-reducible is in* Σ_2. The point is that the existential quantifier selects advice that are checked for correctness using the universal quantifier, while relying on the downwards self-reducibility process.

Chapter Notes

The class $\mathcal{P}$/poly was defined by Karp and Lipton [128] as part of a general formulation of "machines which take advise" [128]. They also noted the equivalence to the traditional formulation of polynomial-size circuits as well as the effect of uniformity (Proposition 3.4).

The Polynomial-Time Hierarchy ($\mathcal{PH}$) was introduced by Stockmeyer [199]. A third equivalent formulation of $\mathcal{PH}$ (via "alternating machines") can be found in [49].

The implication of the failure of the conjecture that $\mathcal{NP}$ is not contained in $\mathcal{P}$/poly on the Polynomial-time Hierarchy (i.e., Theorem 3.12) was discovered by Karp and Lipton [128]. This interesting connection between non-uniform and uniform complexity provides the main motivation for presenting $\mathcal{P}$/poly and $\mathcal{PH}$ in the same chapter. We call the reader's attention to the proof of Theorem 3.12, and specifically to the *inspiring use of downwards self-reducibility.*

Exercises

Exercise 3.1 (sparse sets) A set $S \subset \{0,1\}^*$ is called sparse if there exists a polynomial p such that $|S \cap \{0,1\}^n| \leq p(n)$ for every n.

1. Prove that any sparse set is in $\mathcal{P}$/poly. Note that a sparse set may be undecidable.
2. Prove that a set is in $\mathcal{P}$/poly if and only if it is Cook-reducible to some sparse set.

Guideline: For the forward direction of Part 2, encode the advice sequence $(a_n)_{n\in\mathbb{N}}$ as a sparse set $\{(1^n, i, \sigma_{n,i}) : n \in \mathbb{N},\ i \leq |a_n|\}$, where $\sigma_{n,i}$ is the i^{th} bit of a_n. For the opposite direction, note that on input x the Cook-reduction makes queries of length at most poly($|x|$), and so all the relevant strings in the target of the reduction can be encoded in the n^{th} advice.

Exercise 3.2 (advise hierarchy) Prove that for any two functions $\ell, \delta : \mathbb{N} \to \mathbb{N}$ such that $\ell(n) < 2^{n-1}$ and δ is unbounded, it holds that $\mathcal{P}/\ell$ is strictly contained in $\mathcal{P}/(\ell + \delta)$.
(Hint: for every sequence $\overline{a} = (a_n)_{n\in\mathbb{N}}$ such that $|a_n| = \ell(n) + \delta(n)$, consider the set $S_{\overline{a}}$ where $x \in S$ if and only if x is the i^{th} string of length $|x|$ and $i \leq |a_{|x|}|$ and the i^{th} bit in $a_{|x|}$ is 1. For more details see Section 4.1.)

Exercise 3.3 Prove that Σ_2 contains all sets that are Cook-reducible to $\mathcal{NP}$.
(Hint: This is quite obvious when using the definition of Σ_2 as presented in Section 3.2.2; see Exercise 3.7. Alternatively, the fact can be proved by using some of the ideas of the proof of Theorem 2.33.)

Exercise 3.4 (the class Π_i) Prove that for any natural number k, a decision problem $S \subseteq \{0,1\}^*$ is in Π_k if there exists a polynomial p and a polynomial time algorithm V such that $x \in S$ if and only if

$$\forall y_1 \in \{0,1\}^{p(|x|)} \exists y_2 \in \{0,1\}^{p(|x|)} \forall y_3 \in \{0,1\}^{p(|x|)} \cdots Q_k y_k \in \{0,1\}^{p(|x|)} \quad \text{s.t. } V(x, y_1, ..., y_k) = 1$$

where Q_k is a universal quantifier if k is odd and is an existential quantifier otherwise.
(Recall that Π_k was defined as $\mathrm{co}\Sigma_k$, which in turn is defined as $\{\{0,1\}^* \setminus S : S \in \Sigma_k\}$.)

Exercise 3.5 (complete problems for $\mathcal{PH}$) A k-alternating quantified Boolean formula is a quantified Boolean formula with up to k alternations between existential and universal quantifiers, starting with an existential quantifier. For example, $\exists z_1 \exists z_2 \forall z_3 \phi(z_1, z_2, z_3)$ (where the z_i's are Boolean variables) is a 2-alternating quantified Boolean formula. Prove that the problem of deciding whether or not a k-alternating quantified Boolean formula is valid is Σ_k-complete. That is, denoting the aforementioned problem by kQBF, prove that kQBF is in Σ_k and every problem in Σ_k is Karp-reducible to kQBF.

Exercise 3.6 (on the relation between $\mathcal{PH}$ and $\mathcal{AC}^0$) Note that there is an obvious analogy between $\mathcal{PH}$ and constant depth circuits of unbounded fan-in, where existential (resp., universal) quantifiers are represented by "large" $\bigvee$ (rep., $\bigwedge$) gates. To articulate this relationship, consider the following definitions.

- A family of circuits $\{C_N\}$ is called highly uniform if there exists a polynomial-time algorithm that answers local queries regarding the structure of the relevant circuit. Specifically, on input (N, u, v), the algorithm determines the type of gates represented by the vertices u and v in C_N as well as whether there exists a directed edge from u to v. Note that this algorithm operates in time that polylogarithmic in the size of C_N.

  We focus on family of polynomial-size circuits, meaning that the size of C_N is polynomial in N, which in turn represents the number of inputs to C_N.

- Fixing a polynomial p, a *p-succinctly represented input* $Z \in \{0,1\}^N$ is a circuit c_Z of size at most $p(\log_2 N)$ such that for every $i \in [N]$ it holds that $c_Z(i)$ equals the i^{th} bit of Z.

- For a fixed family of highly uniform circuits $\{C_N\}$ and a fixed polynomial p, the problem of evaluating a succinctly represented input is defined as follows. Given p-succinct representation of an input $Z \in \{0,1\}^N$, determine whether or not $C_N(Z) = 1$.

For every k and every $S \in \Sigma_k$, show that there exists a family of highly uniform unbounded fan-in circuits of depth k and polynomial-size such that S is Karp-reducible to evaluating a succinctly represented input (with respect to that family of circuits). That is, the reduction should map an instance $x \in \{0,1\}^n$ to a p-succinct representation of some $Z = Z_x \in \{0,1\}^N$ such that $x \in S$ if and only if

$C_N(Z)=1$. (Note that Z is represented by a circuit c_Z of size at most $p(\log_2 N)$, and that it must hold that $|c_Z| \le \text{poly}(n)$ and thus $N \le \exp(\text{poly}(n))$.)[6]

Guideline: Let $S \in \Sigma_k$ and let V be the corresponding verification algorithm as in Definition 3.8. That is, $x \in S$ if and only if $\exists y_1 \forall y_2 \cdots Q_k y_k$, where each $y_i \in \{0,1\}^{\text{poly}(|x|)}$ such that $V(x, y_1, ..., y_k) = 1$. Then, for $m = \text{poly}(|x|)$ and $N = 2^{k \cdot m}$, consider the circuit $C_N(Z) = \bigvee_{i_1 \in [2^m]} \bigwedge_{i_2 \in [2^m]} \cdots Q'_{i_k \in [2^m]} Z_{i_1, i_2, \ldots, i_k}$, and the problem of evaluating it at the input consisting of the truth-table of $V(x, \cdots)$ (i.e., when setting $Z_{i_1,i_2,\ldots,i_k} = V(x, i_1, ..., i_k)$). Note that the size of C_N is $O(N)$.[7]

Exercise 3.7 Verify the following facts:

1. For every $k \ge 1$, it holds that $\Sigma_k \subseteq \mathcal{P}^{\Sigma_k} \subseteq \Sigma_{k+1}$.

   (Note that, for any complexity class $\mathcal{C}$, the class $\mathcal{P}^{\mathcal{C}}$ is the class of sets that are Cook-reducible to some set in $\mathcal{C}$. In particular, $\mathcal{P}^{\mathcal{P}} = \mathcal{P}$.)

2. For every $k \ge 1$, $\Pi_k \subseteq \mathcal{P}^{\Pi_k} \subseteq \Pi_{k+1}$.

   (Hint: For any complexity class $\mathcal{C}$, it holds that $\mathcal{P}^{\mathcal{C}} = \mathcal{P}^{\text{co}\mathcal{C}}$ and $\mathcal{P}^{\mathcal{C}} = \text{co}\mathcal{P}^{\mathcal{C}}$.)

3. For every $k \ge 1$, it holds that $\Sigma_k \subseteq \Pi_{k+1}$ and $\Pi_k \subseteq \Sigma_{k+1}$. Thus, $\mathcal{PH} = \cup_k \Pi_k$.

4. For every $k \ge 1$, if $\Sigma_k \subseteq \Pi_k$ (resp., $\Pi_k \subseteq \Sigma_k$) then $\Sigma_k = \Pi_k$.

   (Hint: For any $S \in \Pi_k$ (resp., $S \in \Sigma_k$), apply the hypothesis to $\{0,1\}^* \setminus S$.)

Exercise 3.8 In continuation to Exercise 3.5, prove that following claims:

1. SAT is computationally equivalent to 1QBF.

2. For every $k \ge 1$, it holds that $\mathcal{P}^{\Sigma_k} = \mathcal{P}^{\texttt{kQBF}}$ and $\Sigma_{k+1} = \mathcal{NP}^{\texttt{kQBF}}$.

   Guideline: Prove that if S is $\mathcal{C}$-complete then $\mathcal{P}^{\mathcal{C}} = \mathcal{P}^S$. Note that $\mathcal{P}^{\mathcal{C}} \subseteq \mathcal{P}^S$ uses the polynomial-time reductions of $\mathcal{C}$ to S, whereas $\mathcal{P}^S \subseteq \mathcal{P}^{\mathcal{C}}$ uses $S \in \mathcal{C}$.

Exercise 3.9 Referring to the notion of downwards self-reducibility (as defined in the proof of Theorem 3.12), prove that if $\mathcal{P} = \mathcal{NP}$ (resp., $\mathcal{NP} = \text{co}\mathcal{NP}$) then any set $S' \in \mathcal{P}/\text{poly}$ that is downwards self-reducible is in $\mathcal{P}$ (resp., in $\mathcal{NP}$).
(Hint: any set $S' \in \mathcal{P}/\text{poly}$ that is downwards self-reducible is in Σ_2.)

Exercise 3.10 In continuation to Part 2 of Exercise 3.1, we consider the class of sets that are Karp-reducible to a sparse set. It can be proved that this class contains SAT if and only if $\mathcal{P} = \mathcal{NP}$ (see [77]). Here, we only consider the special case in which the sparse set is contained in another sparse set that is in $\mathcal{P}$ (e.g., the latter set may be $\{1\}^*$, in which case the former set may be an arbitrary unary set). Specifically, we claim that

[6] Assuming $\mathcal{P} \ne \mathcal{NP}$, it cannot be that $N \le \text{poly}(n)$ (because circuit evaluation can be performed in time polynomial in the size of the circuit).

[7] Advanced comment: the limitations of $\mathcal{AC}^0$ circuits imply limitations on the functions of the truth-table of a *generic* V that such the aforementioned circuits C_N can compute. Unfortunately, these limitations do not seem to provide useful information on the limitations of functions that are confined to truth-tables that have succinct representation (as in the case of the actual V). This fundamental problem is "resolved" in the context of "relativization" by providing V with oracle access to an arbitrary input of length N (or so); cf. [79].

> *if* SAT *is Karp-reducible to a set* $S \subseteq G$ *such that* $G \in \mathcal{P}$ *and* G *is sparse then* SAT $\in \mathcal{P}$.

Using the hypothesis, we outline a polynomial-time procedure for solving the search problem of SAT, and leave the task of providing the details as an exercise. The procedure conducts a DFS on the tree of all possible partial truth assignment to the input formula, while truncating the search at nodes that are roots of sub-trees that were already proved to contain no satisfying assignment (at the leaves).[8]

Guideline: The key observation is that each internal node (which yields a formula derived from the initial formulae by instantiating the corresponding partial truth assignment) is mapped by the reduction either to a string not in G (in which case we conclude that the sub-tree contains no satisfying assignments and backtrack from this node) or to a string in G. In the latter case, unless we already know that this string is not in S, we start a scan of the sub-tree rooted at this node. but once we backtrack from this internal node, we know that the corresponding element of G is not in S, and we will never extend a node mapped to this element again. Also note that once we reach a leaf, we can check by ourselves whether or not it corresponds to a satisfying assignment to the initial formula. (Hint: When analyzing the forgoing procedure, note that on input an n-variable formulae ϕ the number of times we start to scan a sub-tree is at most $n \cdot |\cup_{i=1}^{\mathrm{poly}(|\phi|)} \{0,1\}^i \cap (G \setminus S)|$.)

[8]For an n-variable formulae, the leaves of the tree correspond to all possible n-bit long strings, and an internal node corresponding to τ is the parent of nodes corresponding to $\tau 0$ and $\tau 1$.

Chapter 4

More Resources, More Power?

More electricity, less toil.

The Israeli Electricity Company, 1960s

Is it indeed the case that the more resources one has, the more one can achieve? The answer may seem obvious, but the obvious answer (of yes) actually presumes that the worker knows how much resources are at his/her disposal. In this case, when allocated more resources, the worker (or computation) can indeed achieve more. But otherwise, nothing may be gained by adding resources.

In the context of computational complexity, an algorithm knows the amount of resources that it is allocated if it can determine this amount without exceeding the corresponding resources. This condition is satisfies in all natural cases, but it may not hold in general. The latter fact should not be that surprising: we already know that some functions are not computable and if these functions are used to determine resources then the algorithm may be in trouble. Needless to say, this discussion requires some formalization, which is provided in the current chapter.

Summary: When using "nice" functions to determine the algorithm's resources, it is indeed the case that more resources allow for more tasks to be performed. However, when "ugly" functions are used for the same purpose, increasing the resources may have no effect. By nice functions we mean functions that can be computed without exceeding the amount of resources that they specify (e.g., $t(n) = n^2$ or $t(n) = 2^n$). Naturally, "ugly" functions do not allow to present themselves in such nice forms.

The forgoing discussion refers to a uniform model of computation and to (natural) resources such as time and space complexities. Thus, we get results asserting, for example, that there are functions computable in cubic-time but not in quadratic-time. In case of non-uniform models

of computation, the issue of "nicety" does not arise, and it is easy to establish separations between levels of circuit complexity that differ by any unbounded amount.

Results that *separate* the class of problems solvable within one resource bound from the class of problems solvable within a larger resource bound are called hierarchy theorems. Results that indicate the non-existence of such separations, hence indicating a "gap" in the growth of computing power (or a "gap" in the existence of algorithms that utilize the added resources), are called gap theorems. A somewhat related phenomenon, called speed-up theorems, refers to the inability to define the complexity of some problems.

Caveat: Uniform complexity classes based on specific resource bounds (e.g., cubic-time) are model dependent. Furthermore, the tightness of separation results (i.e., how much more time is required to solve an additional computational problem) is also model dependent. Still the existence of such separations is a phenomenon common to all reasonable and general models of computation (as referred to in the Cobham-Edmonds Thesis). In the following presentation, we will explicitly differentiate model-specific effects from generic ones.

Organization: We will first demonstrate the "more resources yield more power" phenomenon in the context of non-uniform complexity. In this case the issue of "knowing" the amount of resources allocated to the computing device does not arise, because each device is tailored to the amount of resources allowed for the input length that it handles (see Section 4.1).

4.1 Non-uniform complexity hierarchies

The model of machines that use advice (cf. §1.2.4.2 and Section 3.1.2) offers a very convenient setting for separation results. We refer specifically, to classes of the form $\mathcal{P}/\ell$, where $\ell : \mathbb{N} \to \mathbb{N}$ is an arbitrary function (see Definition 3.5). Recall that every Boolean function is in $\mathcal{P}/2^n$, by virtue of a trivial algorithm that is given as advice the truth-table of the function restricted to the relevant input length. An analogous algorithm underlies the following separation result.

Theorem 4.1 *For any two functions $\ell', \delta : \mathbb{N} \to \mathbb{N}$ such that $\ell'(n) + \delta(n) \leq 2^n$ and δ is unbounded, it holds that $\mathcal{P}/\ell'$ is strictly contained in $\mathcal{P}/(\ell' + \delta)$.*

Proof: Let $\ell \stackrel{\text{def}}{=} \ell' + \delta$, and consider the algorithm A that given advice $a_n \in \{0,1\}^{\ell(n)}$ and input $i \in \{1, ..., 2^n\}$ (viewed as an n-bit long string), outputs the i^{th} bit of a_n if $i \leq |a_n|$ and zero otherwise. Clearly, for any $\overline{a} = (a_n)_{n\in\mathbb{N}}$ such that $|a_n| = \ell(n)$, it holds that the function $f_{\overline{a}}(x) \stackrel{\text{def}}{=} A(a_{|x|}, x)$ is in $\mathcal{P}/\ell$. Furthermore, different sequences $\overline{a}$ yield different functions $f_{\overline{a}}$. We claim that some of these functions $f_{\overline{a}}$ are not in $\mathcal{P}/\ell$, thus obtaining a separation.

The claim is proved by considering all possible (polynomial-time) algorithms A' and all possible sequences $\overline{a}' = (a'_n)_{n\in\mathbb{N}}$ such that $|a'_n| = \ell'(n)$. Fixing any algorithm A', we consider the number of n-bit long functions that are correctly computed by $A'(a'_n, \cdot)$. Clearly, the number of these functions is at most $2^{\ell'(n)}$, and thus A' may account for at most $2^{-\delta(n)}$ fraction of the functions $f_{\overline{a}}$ (even when restricted to n-bit strings). This consideration holds for every n and every possible A', and thus the measure of the set of functions that are computable by algorithms that take advice of length ℓ is zero.[1] ■

A somewhat less tight bound can be obtained by using the model of Boolean circuits. In this case some slackness is needed in order to account for the gap between the upper and lower bounds regarding the number of Boolean functions over $\{0,1\}^n$ that are computed by Boolean circuits of size $s < 2^n$. Specifically (see Exercise 4.1), an obvious lower-bound on this number is $2^{s/O(\log s)}$ whereas an obvious upper-bound is $\binom{s^2}{s} \approx 2^{2s\log_2 s}$. (Compare these to the lower-bound 2^s, and the upper-bound $2^{s+((\delta(n)-2)/2)}$ used in the proof of Theorem 4.1.)

4.2 Time Hierarchies and Gaps

In this section we show that in the "natural cases" increasing time-complexity allows for more problems to be solved, whereas in "pathological cases" it may happen that even a dramatic increase in the time-complexity provides no additional computing power. As hinted in the introductory comments to the current chapter, the "natural cases" correspond to time bounds that can be determined by the algorithm itself within the specified time complexity.

4.2.1 Time Hierarchies

Note that the non-uniform computing devices considered in the previous section were explicitly given the relevant resource bounds. Actually, they were given the resources themselves and did not need to monitor their usage of these resources. In contrast, when designing algorithms of time complexity $t : \mathbb{N} \to \mathbb{N}$, we need to make sure that the algorithm does not exceed the time bound. Furthermore, when invoked on input x, the algorithm is not given the time bound $t(|x|)$ explicitly, and a reasonable design methodology (for dealing with generic algorithms as we will in the proof of the following separation results) is to let the algorithm compute this bound (i.e., $t(|x|)$). This, in turn, requires to read the entire input (see Exercise 4.2) as well as to compute $t(n)$ using $O(t(n))$ (or so) time. The latter requirement motivates the following definition (which is related to the standard definition of "fully time constructibility" (cf. [116, Sec. 12.3])).

Definition 4.2 (time constructible functions): *A function $t : \mathbb{N} \to \mathbb{N}$ is called* time constructible *if there exists an algorithm that on input n outputs $t(n)$ using at most $t(n)$ steps.*

[1] It suffices to show that this measure is strictly less than one. This is easily done by considering, for every algorithm A', the performance of A' on inputs of length n such that $\delta(n) > 2|\langle A'\rangle| + 2$.

Equivalently, we may require that the mapping $1^n \mapsto t(n)$ be computable within time complexity t. We warn that the foregoing definition is model dependent; however, typically nice functions are computable even faster (e.g., in $\text{poly}(\log t(n))$ steps), in which case the model-dependency is irrelevant (for reasonable and general models of computation, as referred to in the Cobham-Edmonds Thesis). For example, in any reasonable and general model, functions like $t_1(n) = n^2$, $t_2(n) = 2^n$, and $t_3(n) = 2^{2^n}$ are computable in $\text{poly}(\log t_i(n))$ steps.

Likewise, for a fixed model of computation (to be understood from the context) and for any function $t : \mathbb{N} \to \mathbb{N}$, we *denote by* $\text{DTIME}(t)$ *the class of decision problems that are solvable in time complexity* t.

4.2.1.1 The Time Hierarchy Theorem

In the following theorem, we refer to the model of two-tape Turing machines. In this case we obtain quite a tight hierarchy, when referring to the relation between t_1 and t_2. We stress that using the Cobham-Edmonds Thesis, this results yields (possibly less tight) hierarchy theorems for any reasonable and general model of computation.

Teaching note: The standard statement of the following result asserts that *for any time computable function* t_2 *and every function* t_1 *such that* $t_2 = \omega(t_1 \log t_1)$ *and* $t_1(n) > n$ *it holds that* $\text{DTIME}(t_1)$ *is strictly contained in* $\text{DTIME}(t_2)$. We find the proof of the current version more intuitive, and comment on the proof of the standard version after proving the current one.

Theorem 4.3 (time hierarchy for two-tape Turing machines): *For any time computable function* t_1 *and every function* t_2 *such that* $t_2(n) \geq (\log t_1(n))^2 \cdot t_1(n)$ *and* $t_1(n) > n$ *it holds that* $\text{DTIME}(t_1)$ *is strictly contained in* $\text{DTIME}(t_2)$.

As will become clear from the proof, an analogous result holds for any model in which a universal machine can emulate t steps of another machine in $O(t \log t)$ time. Before proving Theorem 4.3, we derive the following corollary.

Corollary 4.4 (time hierarchy for any reasonable and general model): *For any reasonable and general model of computation there exists a positive polynomial* p *such that for any time-computable function* t_1 *and every function* t_2 *such that* $t_2 > p(t_1)$ *and* $t_1(n) > n$ *it holds that* $\text{DTIME}(t_1)$ *is strictly contained in* $\text{DTIME}(t_2)$.

Proof of Corollary 4.4: Letting DTIME_2 denote the classes that correspond to two-tape Turing machines, we have $\text{DTIME}(t_1) \subseteq \text{DTIME}_2(t_1')$ and $\text{DTIME}(t_2) \supseteq \text{DTIME}_2(t_2')$, where $t_1' = \text{poly}(t_1)$ and t_2' is defined such that $t_2(n) = \text{poly}(t_2'(n))$. The latter unspecified polynomials, hereafter denoted p_1 and p_2 respectively, are the ones guaranteed by the Cobham-Edmonds Thesis. Also, the hypothesis that t_1 is time-computable implies that $t_1' = p_1(t_1)$ is time-constructible with respect to the two-tape Turing machine model. Thus, for a suitable choice of the polynomial p, it holds that

$$t_2'(n) = p_2^{-1}(t_2(n)) > p_2^{-1}(p(t_1(n))) > p_2^{-1}(p(p_1^{-1}(t_1'(n)))) > (t_1'(n))^2 .$$

Invoking Theorem 4.3, we have $\text{DTIME}_2(t'_2) \supset \text{DTIME}_2(t'_1)$, and the corollary follows. ■

Proof of Theorem 4.3: The idea is to construct a Boolean function f such that all machines having time complexity t_1 fail to compute f. This is done by associating each possible machine M a different input x_M (e.g., $x_M = \langle M \rangle$), and making sure that $f(x_M) \neq M'(x)$, where $M'(x)$ denotes an emulation of $M(x)$ that is suspended after $t_1(|x|)$ steps. Actually, we are going to use a mapping μ of inputs to machines (i.e., $\mu(x_M) = M$), such that each machine is in the range of μ and μ is very easy to compute.

The issue is presenting an algorithm for computing f. This algorithm is straightforward: On input x, it computes $t = t_1(|x|)$, determines the machine $M = \mu(x)$ that corresponds to x (outputting a default value of no such machine exists), emulates $M(x)$ for t steps, and returns the value $1 - M'(x)$. The question is how much time is required for this emulation. We should bear in mind that the time complexity of our algorithm needs to be evaluated in the two-tape Turing machine model, whereas M itself is a two-tape Turing machine. The obvious approach is to implement our algorithm on a three-tape Turing-machine, using two tapes for the emulation itself and another tape for the emulation procedure. Using this approach, each step of M is emulated in $O(|\langle M \rangle|)$ steps (on the three-tape machine). Emulating t' steps of a three-tape machine on a two-tape machine requires $O(t' \log t')$ steps, and so the emulation costs $O(T_M(|x|) \log T_M(|x|))$, where $T_M(n) = O(|\langle M \rangle| \cdot t_1(n))$.

It turns out that the quality of the result we obtain depends on the mapping μ of inputs to machines. Using the naive (identity) mapping of the input x_M to the machine M (such that $x_M = \langle M \rangle$ is a description of M), we can only establish the theorem for $t_2(n) = \omega(n \cdot t_1(n) \log(n \cdot t_1(n)))$. The theorem follows by associating with machine M the input $x_M = 0^m 1 \langle M \rangle$, where $m = 2^{|\langle M \rangle|}$, because in this case $|\langle M \rangle| < \log_2 |x_M| = O(\log t_1(|x_M|))$. In other words, we may use the mapping μ such that $\mu(x) = M$ if $x = 0^{2^{|\langle M \rangle|}} 1 \langle M \rangle$ and $\mu(x)$ equals some fixed machine otherwise. ■

> **Teaching note:** Proving the standard version of Theorem 4.3 cannot be done by associating a sufficiently long input x_M with each machine M, because this does not allow to get rid from an additional unbounded factor multiplier of $t_1(n) \log t_1(n)$. Note that the latter factor needs to be computable (at the very least) and thus cannot be accounted for by the generic ω-notation that appears in the standard version (cf. [116, Thm. 12.9]). Instead, a different approach is taken (see Footnote 2).

[2] The function f is not defined with reference to $t_1(|x_M|)$ steps of $M(x_M)$, but rather with reference to the result of emulating $M(x_M)$ while using a total of $t_2(|x_M|)$ steps in the emulation process (i.e., in the algorithm used to compute f). This guarantees that f is in $\text{DTIME}(t_2)$, and pushes the problem to showing that f is not in $\text{DTIME}(t_1)$, where the problem is resolved by noting that each candidate machine having time complexity t_1 will have its executions fully emulated on any sufficiently long input. Thus, we merely need to associate with each M a disjoint set of infinitely many inputs and make sure that M errs on each of these inputs.

Comments. The proof of Theorem 4.3 associates with each potential machine an input and makes this machine err on this input. The aforementioned association is rather flexible: it should merely be efficiently computed (in the direction from the input to a possible machine) and should be sufficiently shrinking (in that direction). Specifically, we used the mapping μ such that $\mu(x) = M$ if $x = 0^{2^{|\langle M \rangle|}} 1\langle M \rangle$ and $\mu(x)$ equals some fixed machine otherwise. We comment that each machine can be made to err on infinitely many inputs by redefining μ such that $\mu(x) = M$ if $0^{2^{|\langle M \rangle|}} 1\langle M \rangle$ is a prefix of x (and $\mu(x)$ equals some fixed machine otherwise). We also comment that, in contrast to the proof of Theorem 4.3, the proof of Theorem 1.5 utilizes a rigid mapping of inputs to machines (i.e., there $\mu(x) = M$ if $x = \langle M \rangle$).

4.2.1.2 Impossibility of speed-up for universal computation

The Time Hierarchy Theorem (Theorem 4.3) implies that the computation of a universal machine cannot be significantly sped-up. That is, consider the function $\mathtt{u}'(\langle M \rangle, x, t) \stackrel{\text{def}}{=} y$ if on input x machine M halts within t steps and outputs the string y, and $\mathtt{u}'(\langle M \rangle, x, t) \stackrel{\text{def}}{=} \perp$ if on input x machine M makes more than t steps. Recall that the value of $\mathtt{u}'(\langle M \rangle, x, t)$ can be computed in $\widetilde{O}(|x| + |\langle M \rangle| \cdot t)$ steps. Theorem 4.3 implies that this value cannot be computed with significantly less steps.

Theorem 4.5 *There exists no two-tape Turing machine that, on input $\langle M \rangle, x$ and t, computes $\mathtt{u}'(\langle M \rangle, x, t)$ in $o((t + |x|) \cdot f(M)/\log^2(t + |x|))$ steps, where f is an arbitrary function.*

A similar result holds for any reasonable and general model of computation (cf., Corollary 4.4).

Proof: Suppose (towards the contradiction) that, for every M, given x and $t > |x|$, the value of $\mathtt{u}'(\langle M \rangle, x, t)$ can be computed in $o(t/\log^2 t)$ steps. Consider an arbitrary time computable t_1 (s.t. $t_i1(n) > n$) and an arbitrary set $S \in \text{DTIME}(t_2)$, where $t_2(n) = t_1(n) \cdot \log^2 t_1(n)$. Let M be a machine of time complexity t_2 that decides membership in S, and consider an algorithm that, on input x, first computes $t = t_1(|x|)$, and then computes (and outputs) the value $\mathtt{u}'(\langle M \rangle, x, t \log^2 t)$. By the time constructibility of t_1, the first computation can be implemented in t steps, and by the contradiction hypothesis the same holds for the second computation. Thus, S can be decided in $\text{DTIME}(t_1)$, implying that $\text{DTIME}(t_2) = \text{DTIME}(t_1)$, which in turn contradicts Theorem 4.3. ■

4.2.1.3 Hierarchy theorem for non-deterministic time

Analogously to DTIME, for a fixed model of computation (to be understood from the context) and for any function $t : \mathbb{N} \to \mathbb{N}$, we *denote by* $\text{NTIME}(t)$ *the class of sets that are accepted by some non-deterministic machine of time complexity* t. Alternatively, analogously to the definition of $\mathcal{NP}$, a set $S \subseteq \{0,1\}^*$ is in $\text{NTIME}(t)$ if there exists a *linear-time* algorithm V such that the two conditions hold

1. For every $x \in S$ there exists $y \in \{0,1\}^{t(|x|)}$ such that $V(x,y)=1$.

2. For every $x \notin S$ and every $y \in \{0,1\}^*$ it holds that $V(x,y)=0$.

We warn that the two formulations are not identical, but in sufficiently strong models (e.g., two-tape Turing machines) they are related up to logarithmic factors (see Exercise 4.3). The hierarchy theorem itself is similar to the one for deterministic time, except that here we require that $t_2(n) \geq (\log t_1(n+1))^2 \cdot t_1(n+1)$ (rather than $t_2(n) \geq (\log t_1(n))^2 \cdot t_1(n)$). That is:

Theorem 4.6 (non-deterministic time hierarchy for two-tape Turing machines): *For any time computable function t_1 and every function t_2 such that $t_2(n) \geq (\log t_1(n+1))^2 \cdot t_1(n+1)$ and $t_1(n) > n$ it holds that* $\textsc{Ntime}(t_1)$ *is strictly contained in* $\textsc{Ntime}(t_2)$.

Proof Sketch: We cannot just apply the proof of Theorem 4.3, because the Boolean function f defined there requires the ability to determine whether M accepts the input x_M in $t_1(|x_M|)$ steps. In the current context, M is a non-deterministic machine and so the only way we know how to determine this questions (both for a yes and no answers) is to try all the ($2^{t_1(|x_M|)}$) relevant executions. But this would put f in $\textsc{Dtime}(2^{t_1})$, rather than in $\textsc{Ntime}(\widetilde{O}(t_1))$, and so a different approach is needed.

We associate with each machine M, a large interval of strings (viewed as integers), denoted $I_M = [\ell_M, u_M]$, such that the various intervals do not intersect and such that it is easy to determine for each string x in which interval it falls. For each $x \in [\ell_M, u_M - 1]$, we define $f(x) = 1$ if and only if there exists a non-deterministic computation of M that accepts the input $x' \stackrel{\text{def}}{=} x+1$ in $t_1(|x'|)$ steps. As for $f(u_M)$, we define it as *zero* if and only if there exists a non-deterministic computation of M that accepts the input ℓ_M in $t_1(|\ell_M|)$ steps. This definition is coupled with a non-deterministic machine for accepting the set $\{x : f(x)=1\}$. On input $x \in [\ell_M, u_M - 1]$, this non-deterministic machine emulates a non-deterministic computation of M, which can be done in time $(\log t_1(|x+1|))^2 \cdot t_1(|x+1|)$. On input $x = u_M$, the machine just tries all $2^{t_1(|\ell_M|)}$ ($t_1(|\ell_M|)$-step) executions of $M(\ell_M)$, which can be done in time $t_1(|u_M|) \cdot t_1(|u_M|)$ provided the interval was chosen to be large enough (i.e., $t_1(|u_M|) \geq 2^{t_1(|\ell_M|)}$, which certainly holds when $|u_M| \geq 2^{t_1(|\ell_M|)}$).

Defining f this way guarantees that it is not in $\textsc{Ntime}(t_1)$. Suppose on the contrary, that some non-deterministic machine M of time complexity t_1 accepts the set $\{x : f(x)=1\}$. We define a Boolean function A_M such that $A_M(x)=1$ if and only if there exists a non-deterministic computation of M that accepts the input x, and note that $A_M(x) = f(x)$. Focusing on the interval $[\ell_M, u_M]$, we have $A_M(x) = f(x)$ for every $x \in [\ell_M, u_M]$, which (combined with the definition of f) implies that $A_M(x) = f(x) = A_M(x+1)$ for every $x \in [\ell_M, u_M - 1]$ and $A_M(u_M) = f(u_M) = 1 - A_M(\ell_M)$. Thus, we reached a contraction (i.e., $A_M(\ell_M) = \cdots = A_M(u_M) = 1 - A_M(\ell_M)$). $\square$

4.2.2 Time Gaps and Speed-Up

In contrast to Theorem 4.3, there exists functions $t : \mathbb{N} \to \mathbb{N}$ such that $\text{DTIME}(t) = \text{DTIME}(t^2)$ (or even $\text{DTIME}(t) = \text{DTIME}(2^t)$). Needless to say, these functions are not time-constructible (and thus the aforementioned fact does not contradict Theorem 4.3). The reason for this phenomenon is that, for such functions t, there exists not algorithms that have time complexity above t but below t^2 (resp., 2^t).

Theorem 4.7 (the time gap theorem): *For every non-decreasing computable function $g : \mathbb{N} \to \mathbb{N}$ there exists a non-decreasing computable function $t : \mathbb{N} \to \mathbb{N}$ such that* $\text{DTIME}(t) = \text{DTIME}(g(t))$.

The forgoing examples referred to $g(m) = m^2$ and $g(m) = 2^m$. Since we are mainly interested in dramatic gaps (i.e., super-polynomial functions g), the model of computation does not matter here (as long as it is reasonable and general).

Proof: Consider an enumeration of all possible algorithms (or machines) and let t_i denote the time complexity of the i^{th} algorithm. Recall that we cannot enumerate only machines that halt on every input, and that $t_i(n) = \infty$ if the i^{th} machine does not halt on some n-bit long input. The basic idea is to define t such that no t_i is "sandwiched" between t and $g(t)$, and thus no algorithm will have time complexity between t and $g(t)$.

Intuitively, if $t_i(n)$ is finite, then we may define t such that $t_i(n) < t(n)$, whereas if $t_i(n) = \infty$ then any finite value of $t(n)$ will do (because then $t_i(n) > g(t(n))$). The problem is that we want t to be computable, whereas we cannot tell whether or not $t_i(n)$ is finite. However, we do not really need to make the latter decision: for each candidate value v of $t(n)$, we should just determine whether or not $t_i(n) \in [v, g(v)]$, which can be decided by running the i^{th} machine for at most $g(v)$ steps (on each n-bit long string). Bearing in mind that we should deal with all possible machines, we obtain the following procedure for setting $t(n)$.

Let $\mu : \mathbb{N} \to \mathbb{N}$ be any unbounded and computable function (e.g., $\mu(n) = n$ will do). Starting with $v = n + 1$, we keep incrementing v until v satisfies, for every $i \in \{1, ..., \mu(n)\}$, either $t_i(n) < v$ or $t_i(n) > g(v)$. This condition can be verified by computing $\mu(n)$ and $g(v)$, and emulating the execution of of the $\mu(n)$ machines on each of the n-bit long strings for $g(v)$ steps. The procedure sets $g(n)$ to equal the first value v satisfying the aforementioned condition, and halts. To show that the procedure halts on every n, consider the set $H_n \subseteq \{1, ..., \mu(n)\}$ of indices of the relevant machines that halt on all inputs of length n. Then the procedure definitely halts before reaching the value $v = m + 2$, where $m = \max_{i \in H_n}\{t_i(n)\}$. (Indeed, the procedure may halt with $v \leq m$, provide that $g(v) < m$.)

For this function t, we claim that $\text{DTIME}(t) = \text{DTIME}(g(t))$. Indeed, let $S \in \text{DTIME}(g(t))$ and suppose that the i^{th} algorithm decides S in time at most $g(t)$; that is, for every n, it holds that $t_i(n) \leq g(t(n))$. Then (by the construction), for every n satisfying $\mu(n) \geq i$, it holds that $t_i(n) < t(n)$, and it follows that the i^{th} algorithm decides S in time at most t on all but finitely many inputs. Combining this algorithm with a "look-up table" machine that handles the exceptional inputs, the theorem follows. ■

Comment: The function t defined by the foregoing proof is computable in time that exceeds $g(t)$. Specifically, the presented procedure computes $t(n)$ (as well as $g(f(n))$) in time $\widetilde{O}(2^n \cdot g(t(n)) + T_g(t(n)))$, where $T_g(m)$ denotes the number of steps required to compute $g(m)$ on input m.

Speed-up Theorems. Theorem 4.7 can be viewed as asserting that some time complexity classes (i.e., $\text{DTIME}(g(t))$ in the theorem) collapse to lower classes (i.e., to $\text{DTIME}(t)$). A conceptually related phenomenon is of problems that have no optimal algorithm (not even in a very mild sense); that is, every algorithm for these ("pathological") problems can be drastically sped-up. It follows that the complexity of these problems can not be defined (i.e., as the complexity of the best algorithm solving this problem). The following drastic speed-up theorem should not be confused with the linear speed-up that is an artifact of the definition of a Turing machine (see Exercise 4.4).[3]

Theorem 4.8 (the time speed-up theorem): *For every computable* (and super-linear) *function g there exists a decidable set S such that if $S \in \text{DTIME}(t)$ then $S \in \text{DTIME}(t')$ for t' satisfying $g(t'(n)) < t(n)$.*

Taking $g(n) = n^2$ (or $g(n) = 2^n$), the theorem asserts that, for every t, if $S \in \text{DTIME}(t)$ then $S \in \text{DTIME}(\sqrt{t})$ (resp., $S \in \text{DTIME}(\log t)$). Note that Theorem 4.8 can be applied any number of times, which means that we cannot give a reasonable estimate to the complexity of deciding membership in S. In contrast, recall that in some important cases (see, e.g., Theorem 2.31), optimal algorithms for solving computational problems do exist. Specifically, algorithms solving (candid) search problems in NP cannot be speed-up (see Theorem 2.31), nor can the computation of a universal machine (see Theorem 4.5).

We refrain from presenting a proof of Theorem 4.8, but comment on the complexity of the sets involved in this proof. The proof (presented in [116, Sec. 12.6]) provides a construction of a set S in $\text{DTIME}(t') \setminus \text{DTIME}(t'')$ for any $t'(n) = h(n - O(1))$ and $t''(n) = h(n - \omega(1))$, where $h(n)$ denoted g iterated n times on 2 (i.e., $h(n) = g^{(n)}(2)$, where $g^{(i+1)}(m) = g(g^{(i)}(m))$ and $g^{(1)} = g$). The set S is constructed such that for every $i > 0$ there exists a $j > i$ and an algorithm that decides S in time t_i but not in time t_j, where $t_k(n) = h(n - k)$.

4.3 Space Hierarchies and Gaps

Hierarchy and Gap Theorems analogous to Theorem 4.3 and Theorem 4.7, respectively, are known for space complexity. In fact, since space-efficient emulation of space-bounded machines is simpler than time-efficient emulations of time-bounded machines, the results tend to be sharper. This is most conspicuous in the case of the separation result (stated next), which is optimal (in light of linear speed-up results; see Exercise 4.4).

[3]We note that this fact was implicitly addressed in the proof of Theorem 4.3, by allowing an emulation overhead that depends on the length of the description of the emulated machine.

Before stating the result, we need a few preliminaries. We refer the reader to §1.2.3.4 for a definition of space complexity (and to Chapter 5) for further discussion. As in case of time complexity, we consider a specific model of computation, but the results hold for any other reasonable and general model. Specifically, we consider three-tape Turing machines, because we designate two special tapes for input and output. For any function $s : \mathbb{N} \to \mathbb{N}$, we *denote by* $\mathrm{DSPACE}(s)$ *the class of decision problems that are solvable in space complexity* s. Analogously to Definition 4.2, we call a function $s : \mathbb{N} \to \mathbb{N}$ space constructible if there exists an algorithm that on input n outputs $s(n)$ using at most $s(n)$ cells of the work-tape. Actually, functions like $s_1(n) = \log n$, $s_2(n) = (\log n)^2$, and $s_3(n) = 2^n$ are computable using $\log s_i(n)$ space.

Theorem 4.9 (space hierarchy for three-tape Turing machines): *For any space computable function* s_2 *and every function* s_1 *such that* $s_2 = \omega(s_1)$ *and* $s_1(n) > \log n$ *it holds that* $\mathrm{DSPACE}(s_1)$ *is strictly contained in* $\mathrm{DSPACE}(s_2)$.

Theorem 4.9 is analogous to the traditional version of Theorem 4.3 (rather to the one we presented), and is proven using the alternative approach sketched in Footnote 2. The details are left as an exercise (see Exercise 4.5).

Chapter Notes

The material presented in this chapter predates the theory of NP-completeness and the dominant stature of the P-vs-NP Question. At these early days, the field (to be known as complexity theory) did not yet develope an independent identity and its perspectives were dominated by two classical theories: the theory of computability (and recursive function) and the theory of formal languages. Nevertheless, we believe that the results presented in this chapter are interesting for two reasons. Firstly, as stated upfront, these results address the natural question of under what conditions do more resources help. Secondly, their inapplicability to questions of the type of P-vs-NP, illustrates the non-generic flavor of these questions as referring to specific (and natural) aspects of the "complexity of computation".

The hierarchy theorems (e.g., Theorem 4.3) were proved by Hartmanis and Stearns [109]. Gap theorems (e.g., Theorem 4.7, often referred to as Borodin's Gap Theorem) were proven by Borodin [44]. A axiomatic treatment of complexity measures and corresponding speed-up theorems (e.g., Theorem 4.8, often referred to as Blum's Speed-up Theorem) are due to Blum [36].

Exercises

Exercise 4.1 For $s < 2^n$, prove that the number of Boolean functions over $\{0,1\}^n$ that are computed by Boolean circuits of size s is at least $2^{s/O(\log s)}$ and at most $\binom{s^2}{s}$.

Guideline: Any Bollean function $f : \{0,1\}^\ell \to \{0,1\}$ can be computed by a circuit of size $s = O(\ell \cdot 2^\ell)$. Thus, circuits of size s may compute $2^{2^\ell} > 2^{s/O(\log s)}$ different Boolean functions. On the other hand, the number of circuits of size s is less than $\binom{s^2}{s}$, where the upper-bound represents the number of possible choices of pair of gates that feed any gate in the circuit.

Exercise 4.2 Referring to a reasonable model of computation (and assuming that the input length is not given explicitly (e.g., as in Definition 10.10)), prove that any algorithm that has sub-linear time-complexity actually has constant time-complexity.
(Hint: Consider the question of whether or not there exists an infinite set of strings S such that when invoked on any input $x \in S$ the algorithm reads all of x. Note that if S is infinite then the algorithm cannot have sub-linear time-complexity, and prove that if S is finite then the algorithm has constant time-complexity.)

Exercise 4.3 Prove that the two definitions of NTIME, presented in §4.2.1.3, are related up to logarithmic factors. Note the importance of condition that V has linear (rather than polynomial) time-complexity.
(Hint: when emulating a non-deterministic machine by the verification procedure V, encode the non-deterministic choices in y while using sufficient padding such that t steps of the original machine can be emulated in $|y| = O(t \log t)$ steps of V.)

Exercise 4.4 Prove that any problem that can be solved by a two-tape Turing machine that has time complexity t can be solved by another two-tape Turing machine having time complexity t', where $t'(n) = O(n) + (t(n)/2)$.
(Hint: Consider a machine that uses a larger alphabet, capable of encoding a constant number of symbols of the original machine, and thus capable of emulating a constant number of steps of the original machine in a smaller constant number of steps. Note that the $O(n)$ term accounts to a preprocessing required to encode the input in the work-alphabet of the new machine, and that a similar result for one-tape Turing machine seems to require a $O(n^2)$ term.)
State and prove an analogous result for space complexity, when using the standard definition of space as recalled in Section 4.3. Note that this result does not hold with respect to "binary space complexity" as defined in Section 5.1.

Exercise 4.5 Prove Theorem 4.9. As a warm-up, assume that s_1 (rather than s_2) is space constructible.

Guideline: Note that providing a space-efficient emulation of one machine by another machine is easier than providing an analogous time-efficient emulation.

Chapter 5

Space Complexity

Open are the double doors of the horizon; unlocked are its bolts.

Philip Glass, Akhnaten, Prelude

Whereas the number of steps taken during a computation is the primary measure of its efficiency, the amount of temporary storage used by the computation is also a major concern. Furthermore, in some settings, space is even more scarce than time.

This chapter is devoted to the study of the space complexity of computations, while focusing on two rather extreme cases. The first case is that of algorithms having logarithmic space complexity. We view such algorithms as utilizing the naturally minimal amount of temporary storage, where the term "minimal" is used here in an intuitive (but somewhat inaccurate) sense, and note that logarithmic space complexity seems a more stringent requirement than polynomial time. The second case is that of algorithms having polynomial space complexity. Such algorithms can perform almost all computational tasks considered in this book (i.e., the class $\mathcal{PSPACE}$ contains almost all complexity classes considered in this book).

In addition to the intrinsic interest in space complexity, its study provides an interesting perspective on the study of time complexity. For example, in contrast to the common conjecture by which $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$, we shall see that analogous space complexity classes (e.g., $\mathcal{NL}$) are closed under complementation (e.g., $\mathcal{NL} = \mathrm{co}\mathcal{NL}$).

Summary: We first consider algorithms of logarithmic space complexity. Such algorithms may be used for solving various search and decision problems, providing reductions among such problems, and yielding a strong notion of uniformity for Boolean circuits. The highlight of this part is a log-space algorithm for exploring (undirected) graphs.

We then turn to non-deterministic machines, focusing on the complexity class $\mathcal{NL}$ that is captured by the problem of deciding directed connec-

tivity of (directed) graphs. The highlight of this part is a proof that $\mathcal{NL} = \text{co}\mathcal{NL}$.

We conclude with a short discussion of the class $\mathcal{PSPACE}$, proving that the set of satisfiable quantified Boolean formulae is $\mathcal{PSPACE}$-complete (under polynomial-time reductions). We mention the similarity between this proof and the proof that $\text{NSPACE}(s) \subseteq \text{DSPACE}(O(s^2))$.

We stress that, as in the case of time complexity, the main results presented in this chapter hold for any reasonable model of computation.[1] In fact, when properly defined, space complexity is even more robust than time complexity. Still, for sake of clarity, we often refer to the specific model of Turing machines.

Organization. Space complexity seems to behave quite differently from time complexity, and seems to require a different mind-set as well as auxiliary conventions. Some of these are discussed in Section 5.1. We then turn to the study of logarithmic space complexity (see Section 5.2) and the corresponding non-deterministic version (see Section 5.3). Finally, we consider polynomial space complexity (see Section 5.4).

5.1 General preliminaries

Space complexity is meant to measure the amount of temporary storage (i.e., computer's memory) used when performing a computational task. Since much of our focus will be on using an amount of memory that is sub-linear in the input length, it is important to use a model in which one can differentiate memory used for computation from memory used for storing the initial input or the final output. That is, we do not want to count the input and output themselves within the space of computation, and thus formulate that they are delivered on special devices that are not considered memory. On the other hand, we have to make sure that the input and output devices cannot be abused for providing work space (which is uncounted for). This leads to the convention by which the input device (e.g., a designated input-tape of a multi-tape Turing machine) is read-only, whereas the output device (e.g., a designated output-tape of a such machine) is write-only. Thus, space complexity accounts for the use of space on the other (storage) devices (e.g., the work-tapes of a multi-tape Turing machine)

Fixing a concrete model of computation (e.g., multi-tape Turing machines), we *denote by* $\text{DSPACE}(s)$ *the class of decision problems that are solvable in space complexity* s. The space complexity of search problems is defined analogously. Specifically, the *standard definition of space complexity* (see §1.2.3.4) *refers to the number of cells of the work-tape scanned by the machine on each input.* We prefer, however, an alternative definition, which provides a more accurate account of the

[1] The only exceptions appear in Exercises 5.3 and 5.14, which refer to the notion of a crossing sequence. The use of this notion in these proofs assumes that the machine scans its storage devices in a serial manner. We stress that the various notions of an instantaneous configuration do not assume such a machine model.

actual storage. Specifically, the binary space complexity of a computation *refers to the the number of bits that can be stored in these cells*, thus multiplying the number of cells by the logarithm of the finite set of work symbols of the machine.[2]

The difference between the two definitions is mostly immaterial, since it amounts to a constant factor and we will discard such factors. Nevertheless, aside from being conceptually right, the definition of binary space complexity will facilitate some technical details (because the number of possible configurations is explicitly upper-bounded in terms of binary space complexity whereas the relation to the standard definition depends on the machine in question). Towards such applications, one may also count the finite state of the machine in its space complexity. Furthermore, for sake of simplicity, we also assume that the machine does not scan the input-tape beyond the boundaries of the input, which are indicated by special symbols.

We stress that individual locations of the (read-only) input-tape (or device) may be read several times. This is essential for many algorithms that use a sublinear amount of space (because such algorithms may need to scan their input more than once while they cannot afford copying their input to their storage device). In contrast, rewriting on (the same loaction of) the write-only output-tape is inessential, and in fact can be eliminated at a relatively small cost (see Exercise 5.1).

5.1.1 On the minimal amount of useful computation space

Bearing in mind that one of our main objectives is identifying natural sub-classes of $\mathcal{P}$, we consider the question of what is the minimal amount of space that allows for meaningful computations. We note that regular sets [116, Chap. 2] are decidable by constant-space Turing machines and that this is all that the latter can decide (see, e.g., [116, Sec. 2.6]). It is tempting to say that sub-logarithmic space machines are not more useful than constant-space machines, because it *seems* impossible to allocate a sub-logarithmic amount of space. This wrong intuition is based on the presumption that the allocation of a non-constant amount of space requires explicitly computing the length of the input, which in turn requires logarithmic space. However, this presumption is wrong: the input itself (in case it is of a proper form) can be used to determine its length, whereas in case the input is not of the proper form then this fact may be detectable (within sub-logarithmic space). In fact, *for $\ell(n) = \log\log n$, the class $\textsc{Dspace}(O(\ell))$ is a proper superset of $\textsc{Dspace}(O(1))$*; see Exercise 5.2. In contrast to Exercise 5.2, double-logarithmic space is indeed the smallest amount of space that is more useful than constant space; that is, *for $\ell(n) = \log\log n$, it holds that $\textsc{Dspace}(o(\ell)) = \textsc{Dspace}(O(1))$*; see Exercise 5.3.

In spite of the fact that some non-trivial things can be done in sub-logarithmic space complexity, the lowest space complexity class that we shall study in depth is logarithmic space (see Section 5.2). As we shall see, this class is the natural habitat of several fundamental computational phenomena.

[2] We note that, unlike in the context of time-complexity, linear speed-up (as in Exercise 4.4) does not seem to represent an actual saving in space resources. Indeed, time can be sped-up by using stronger hardware (i.e., a Turing machine with a bigger work alphabet), but the actual space is not really affected by partitioning it into bigger chunks (i.e., using bigger cells).

A parenthetical comment (or a side lesson). Before proceeding let us highlight the fact that a naive presumption about generic algorithms (i.e., that the use of a non-constant amount of space requires explicitly computing the length of the input) could have led us to a wrong conclusion. This demonstrates the danger in making ("reasonably looking") presumptions about *arbitrary* algorithms, which in turn are the subject of a complexity lower-bound.

5.1.2 Time versus Space

Space complexity behaves very different from time complexity and indeed different paradigms are used in studying it. One notable example is provided by the context of algorithmic composition, discussed next.

5.1.2.1 Two composition lemmas

Unlike time, space can be re-used; but, on the other hand, intermediate results of a computation cannot be recorded for free. These two conflicting aspects are captured in the following composition lemma.

Lemma 5.1 (naive composition): *Let $f_1 : \{0,1\}^* \to \{0,1\}^*$ and $f_2 : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ be computable in space s_1 and s_2, respectively.*[3] *Then f defined by $f(x) \stackrel{\rm def}{=} f_2(x, f_1(x))$ is computable in space s such that*

$$s(n) = \max(s_1(n), s_2(n + \ell(n))) + \ell(n) + O(1)\,,$$

where $\ell(n) = \max_{x\in\{0,1\}^n}\{|f_1(x)|\}$.

That is, $f(x)$ is computed by first computing and storing $f_1(x)$, and then re-using the space (used in the first computation) when computing $f_2(x, f_1(x))$. The additional term of $\ell(n)$ is due to storing the intermediate result (i.e., $f_1(x)$). Lemma 5.1 is useful when ℓ is relatively small, but in many cases $\ell \gg \max(s_1, s_2)$. In these cases, the following composition lemma is more useful.

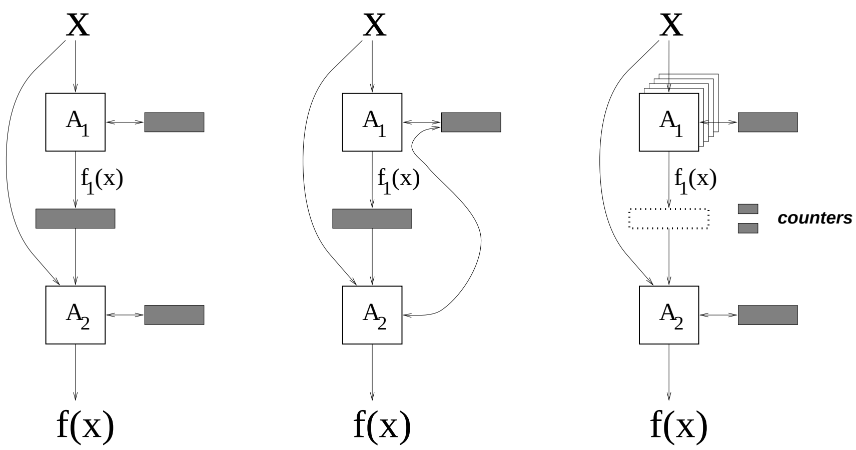
Lemma 5.2 (emulative composition): *Let f_1, f_2, s_1, s_2, ℓ and f be as in Lemma 5.1. Then f is computable in space s such that*

$$s(n) = s_1(n) + s_2(n + \ell(n)) + O(\log(n + \ell(n)))\,.$$

The alternative compositions are depicted in Figure 5.1 (which also shows the most straightforward composition of A_1 and A_2 that makes no attempt to economize space).

Proof: The idea is avoiding the storage of the temporary value of $f_1(x)$, by computing each of its bits ("on the fly") whenever it is needed for the computation of f_2. That is, we do not start by computing $f_1(x)$, but rather start by computing $f_2(x, f_1(x))$ although we do not have some of the bits of the relevant input. The

[3]Here (and throughout the chapter) we assume, for simplicity, that all complexity bounds are monotonically non-decreasing.



The leftmost figure shows the trivial composition (which just invokes A_1 and A_2 without attempt to economize storage), the middle figure shows the naive composition (of Lemma 5.1), and the rightmost figure shows the emulative composition (of Lemma 5.2). In all figures the filled rectangles represent designated storage spaces. The dotted rectangle represents a virtual storage device.

Figure 5.1: Algorithmic composition for space-bounded computation

missing bits will be computed (and re-computed) whenever we need them in the computation of $f_2(x, f_1(x))$. Details follow.

Let us assume, for simplicity, that algorithm A_1 never rewrites on (the same loaction of) its write-only output-tape. As shown in Exercise 5.1, this assumption can be justified at an additive cost of $O(\log \ell(n))$.[4]

Let A_1 and A_2 be the algorithms (for computing f_1 and f_2, respectively) guaranteed in the hypothesis. Then, on input $x \in \{0,1\}^n$, we invoke algorithm A_2 (for computing f_2). Algorithm A_2 is invoked on a virtual input, and so when emulating each of its steps we should provide it with the relevant bit. Thus, we should also keep track of the location of A_2 on the imaginary (virtual) input tape. Whenever A_2 seeks to read the i^{th} bit of its input, where $i \in [n + \ell(n)]$, we provide A_2 with this bit by reading it from x if $i \leq n$ and invoke $A_1(x)$ otherwise. When invoking $A_1(x)$ we provide it with a virtual output tape, which means that we get the bits of its output one-by-one and do not record them anywhere. Instead, we count until reaching the $(i-n)^{\text{th}}$ output bit, which we then pass to A_2 (as the i^{th} bit of $\langle x, f_1(x) \rangle$).

Note that while invoking $A_1(x)$, we suspend the execution of A_2 but keep its current configuration such that we can resume the execution (of A_2) once we get the desired bit. Thus, we need to allocate separate space for the computation of A_2

[4]Alternatively, the idea presented in Exercise 5.1 can be incorporated directly in the current proof.

and for the computation of A_1. In addition, we need to allocate separate storage for maintaining the aforementioned counters (i.e., the bit-location currently read by A_2 and the index of the bit currently produced in the current invocation of A_1). ■

Reflection. The algorithm presented in the proof of Lemma 5.2 is wasteful in terms of time: it re-computes $f_1(x)$ again and again (i.e., once per each access of A_2 to the second part of its input). Indeed, our aim was economizing on space and not on time (and the two goals may be conflicting (see, e.g., [56, Sec. 4.3])).

5.1.2.2 An obvious bound

The time complexity of an algorithm is essentially upper-bounded by an exponential in its space complexity. This is due to an upper-bound on the number of possible instantaneous "configurations" of the algorithm (as formulated in the proof of Theorem 5.3) and the fact that if the computation passes through the same configuration twice then it must loop forever.

Theorem 5.3 *If an algorithm A has binary space complexity s and halts on every input then it has time complexity t such that $t(n) = n \cdot 2^{s(n)+\log_2 s(n)}$.*

Note that for $s(n) = \Omega(\log n)$, the factor of n can be absorbed by $2^{O(s(n))}$, and so we may just write $t(n) = 2^{O(s(n))}$.

Proof: The proof refers to the notion of an *instantaneous configuration* (in a computation). Before starting, we warn the reader that this notion may be given different definitions, each tailored to the application at hand. All these definitions share the desire to specify *variable information* that together with some *fixed information* determines the next step of the computation being analyzed. In the current proof, we fix an algorithm A and an input x, and consider as variable the contents of the storage device (e.g., work-tape of a Turing machine as well as its finite state) and the machine's location on the input device and on the storage device. Thus, an instantaneous configuration of $A(x)$ consists of the latter three objects (i.e., the contents of the storage device and a pair of locations), and can be encoded by a binary string of length $\ell(|x|) = s(|x|) + \log_2 |x| + \log_2 s(|x|)$.

The key observation is that the computation $A(x)$ cannot pass through the same computation twice, because otherwise the computation $A(x)$ passes through this configuration infinitely many times, which means that it does not halt. Intuitively, the point is that the fixed information (i.e., A and x) together with the configuration, determines the next step of the computation. Thus, whatever happens (i steps) after the first time that the computation $A(x)$ passes through configuration γ, will also happen (i steps) after the second time that the computation $A(x)$ passes through γ.

By the forgoing observation, we infer that $t(|x|) < 2^{\ell(|x|)}$, and the theorem follows. ■

5.1.2.3 Subtleties regarding space-bounded reductions

Lemmas 5.1 and 5.2 suffice for the analysis of the affect of many-to-one reductions in the context of space-bounded computations. Specifically:

1. (In spirit of Lemma 5.1:)[5] If f is reducible to g via a many-to-one reduction that can be computed in space s_1, and g is computable in space s_2, then f is computable in space s such that $s(n) = \max(s_1(n), s_2(\ell(n))) + \ell(n)$, where $\ell(n)$ denotes the maximum length of the image of the reduction when applied to some n-bit string.

2. (In spirit of Lemma 5.2:) For f and g as in Item 1, it follows that f is computable in space s such that $s(n) = s_1(n) + s_2(\ell(n)) + O(\log \ell(n))$.

Note that by Theorem 5.3, it holds that $\ell(n) \leq 2^{s_1(n)+\log_2 s_1(n)} \cdot n$. We stress the fact that ℓ is not bounded by s_1 (as in the analogous case of time-bounded computation), but rather by $\exp(s_1)$.

Things gets much more complicated when we turn to general (space-bounded) reductions, especially when referring to such reductions that make a non-constant number of queries. A preliminary issue is defining the space complexity of general reductions (i.e., of oracle machines). In the standard definition, the length of the queries and answers is not counted in the space complexity, but the queries of the reduction (resp., answers given to it) are written on (resp., read from) a special device that is write-only (resp., read-only) for the reduction (and read-only (resp., write-only) for the invoked oracle). Note that these convention are analogous to the conventions regarding input and output (as well as fit the definitions of space-bounded many-to-one reductions (see Section 5.2.2)). The rest of the discussion, which is quite advanced and laconic (and is inessential to the rest of the chapter), concerns two additional issues.

Recall that the complexity of the algorithm resulting from the composition of an oracle machine and an actual algorithm depends on the length of the queries made by the oracle machine. The length of the first query is upper-bounded by an exponential function in the space complexity of the oracle machine, but the same does not necessarily hold for subsequent queries, *unless some conventions are added to enforce it.* For example, consider a reduction, that on input x and access to the oracle f such that $f(z) = 1^{2|z|}$, invokes the oracle $|x|$ times, where each time it uses as a query the answer obtained to the previous query. This reduction uses constant space, but produces queries that are exponentially longer than the input, whereas the first query of any constant-space reduction has length that is linear in its input. This problem can be resolved by placing explicit bounds on the length of the queries that space-bounded reductions are allowed to make; for example, we may bound the length of all queries by the obvious (exponential in the space complexity) bound that holds for the length of the first query.

With the aforementioned convention (or restriction) in place, let us consider the composition of *general* space-bounded reductions with a space-bounded implementation of the oracle. Specifically, we say that a reduction is (ℓ, ℓ')-restricted if,

[5] Here and in the next item, we refer to the case that $f(x) = g(f_1(x))$ rather than to the more general case where $f(x) = g(x, f_1(x))$. Consequently, s_2 is applied to $\ell(n)$ rather than to $n + \ell(n)$.

on input x, all oracle queries are of length at most $\ell(|x|)$ and the corresponding oracle answers are of length at most $\ell'(|x|)$. It turns out that naive composition (in the spirit of Lemma 5.1) remains valid, whereas the emulative composition of Lemma 5.2 breaks down (in the sense that it yield very weak results).

1. Following Lemma 5.1, we claim that *if* Π *can be computed in space* s_1 *when given* (ℓ, ℓ')*-restricted oracle access to* Π' *and* Π' *is solvable is space* s_2*, then* Π *is solvable in space* s *such that* $s(n) = s_1(n) + s_2(\ell(n)) + \ell(n) + \ell'(n)$. The claim is proved by using a naive emulation that allocates separate space for the reduction (i.e., oracle machine) itself, the emulation of its query and answer devices, and the algorithm solving Π'. Note that here we cannot re-use the space of the reduction when running the algorithm that solves Π', because the reduction's computation continues after the oracle answer is obtained.

   A related composition result is presented in Exercise 5.5. It yields $s(n) = 2s_1(n) + s_2(\ell(n)) + 2\ell'(n) + \log_2 \ell(n)$, which for $\ell(n) < 2^{2s_1(n)}$ means $s(n) = 4s_1(n) + s_2(\ell(n)) + 2\ell'(n)$.

2. Turning to the approach underlying Lemma 5.2, we get into more serious trouble. Recomputing the answer of the i^{th} query requires recomputing the query itself, which unlike in Lemma 5.2 is not the input to the reduction but rather depends on the answers to prior queries, which need to be recomputed as well. Thus, the space required for such an emulation may be linear in the number of queries. In fact, we should not expect any better, because any computation of space complexity s can be performed by a constant-space $(2s, 2s)$-restricted reduction to a problem that is solvable in constant-space (see Exercise 5.6).

An alternative notion of space-bounded reductions is discussed in §5.2.4.2. This notion is more cumbersome and more restricted, but it allows recursive composition with a smaller overhead than the two options explored above.

5.1.2.4 Complexity hierarchies and gaps

Recall that more space allows for more computation (see Theorem 4.9), provided that the space-bounding function is "nice" in an adequate sense. Actually, the proofs of space-complexity hierarchies and gaps are simpler than in the analogous proofs for time-complexity, because emulations are easier in the context of space-bounded algorithms (cf. Section 4.3).

5.1.2.5 Simultaneous time-space complexity

Recall that, for space complexity that is at least logarithmic, the time of a computation is always upper-bounded by an exponential function in the space complexity (see Theorem 5.3). Thus, polylogarithmic space complexity may extend beyond polynomial-time, and it make sense to define a class like $\mathcal{SC}$ that consists of all decision problems that may be solved by a polynomial-time algorithm of polylogarithmic space complexity. The class $\mathcal{SC}$ is indeed a natural sub-class of $\mathcal{P}$ (and contains the class $\mathcal{L}$, which is defined in Section 5.2.1).[6]

In general, one may define $\mathrm{DTiSp}(t, s)$ as the class of decision problems solvable by an algorithm that has time complexity t and space complexity s. Note that $\mathrm{DTiSp}(t, s) \subseteq \mathrm{Dtime}(t) \cap \mathrm{Dspace}(s)$ and that a strict containment may hold. We mention that $\mathrm{DTiSp}(\cdot, \cdot)$ provides the arena for the only known absolute (and highly non-trivial) lower-bound regarding $\mathcal{NP}$; see [75]. We also note that lower bounds on time-space trade-offs (see, e.g., [56, Sec. 4.3]) may be stated as referring to the classes $\mathrm{DTiSp}(\cdot, \cdot)$.

5.1.3 Circuit Evaluation

Recall that Theorem 3.1 asserts the existence of a polynomial-time algorithm that, given a circuit $C : \{0,1\}^n \to \{0,1\}^m$ and an n-bit long string x, returns $C(x)$. For circuits of bounded fan-in, the space complexity of such an algorithm can be made linear in the depth of the circuit and logarithmic in its size. This is obtained by the following DFS-type algorithm.

The algorithm (recursively) determines the value of a gate in the circuit by first determining the value of its first in-coming edge and next determining the value of the second in-coming edge. Thus, the recursive procedure, started at each output terminal of the circuit, needs only store the path that leads to the currently processed vertex as well as the temporary values computed for each ancestor. Note that this path is determined by indicating, for each vertex on the path, whether we currently process its first or second in-coming edge. In case we currently process the vertex's second in-coming edge, we need also store the value computed for its first in-coming edge.

The temporary storage used by the foregoing algorithm, on input (C, x), is thus $2d_C + O(\log |x| + \log |C(x)|)$, where d_C denotes the depth of C. The first term in the space-bound accounts for the core activity of the algorithm (i.e., the recursion), whereas the other terms account for the overhead involved in assigning the bits of x to the corresponding input terminals of C and in scanning all output terminals of C.

5.2 Logarithmic Space

Although Exercise 5.2 asserts that "there is life below log-space," logarithmic space seems to be the smallest amount of space that supports interesting computational phenomena. In particular, logarithmic space is required for merely maintaining an auxiliary counter that holds a position in the input, which seems required in many computations. On the other hand, logarithmic space suffices for solving many natural computational problems, for establishing reductions among many natural computational problems, and for a stringent notion of uniformity (of families of

[6] We also mention that $\mathcal{BPL} \subseteq \mathcal{SC}$, where $\mathcal{BPL}$ is defined in §6.1.4.1 and the result is proved in Section 8.5 (see Theorem 8.23).

Boolean circuits). Indeed, an important feature of logarithmic space computations is that they are a natural subclass of the polynomial-time computations (see Theorem 5.3).

5.2.1 The class L

Focusing on decision problems, we denote by $\mathcal{L}$ the class of decision problems that are solvable by algorithms of logarithmic space complexity; that is, $\mathcal{L} = \cup_c \mathrm{DSPACE}(\ell_c)$, where $\ell_c(n) \stackrel{\rm def}{=} c \log_2 n$. Note that, by Theorem 5.3, $\mathcal{L} \subseteq \mathcal{P}$. As hinted, many natural computational problems are in $\mathcal{L}$ (see Exercises 5.4 and 5.7 as well as Section 5.2.4). On the other hand, *it is widely believed that* $\mathcal{L} \neq \mathcal{P}$.

5.2.2 Log-Space Reductions

Another class of important log-space computations is the class of *logarithmic space reductions*. In light of the subtleties discussed in §5.1.2.3 we confine ourselves to the case of many-to-one reductions. Analogously to the definition of Karp-reductions (Definition 2.10), we say that f is a log-space many-to-one reduction of S to S' if f is log-space computable and, for every x, it holds that $x \in S$ if and only if $f(x) \in S'$. Clearly, if S is so reducible to $S' \in \mathcal{L}$ then $S \in \mathcal{L}$. Similarly, one can define a log-space variant of Levin-reductions (Definition 2.11). Both types of reductions are transitive (see Exercise 5.8). Note that Theorem 5.3 applies in this context and implies that these reductions run in polynomial-time. Thus, the notion of a log-space many-to-one reduction is a special case of a Karp-reduction.

We observe that all known Karp-reductions establishing NP-completeness results are actually log-space reductions. This is easily verifiable in the case of the reductions presented in Section 2.3.3 (as well as in Section 2.3.2). For example, consider the generic reduction to `CSAT` presented in the proof of Theorem 2.20: The constructed circuit is "highly uniform" and can be easily constructed in logarithmic-space (see Section 5.2.3). A degeneration of this reduction suffices for proving that every problem in $\mathcal{P}$ is log-space reducible to the problem of evaluating a given circuit on a given input. Note that the latter problem is in $\mathcal{P}$, and thus we may say that it is *P-complete under log-space reductions*.

Theorem 5.4 (The complexity of Circuit Evaluation): *Let* `CEVL` *denote the set of pairs* (C, α) *such that* C *is a Boolean circuit and* $C(\alpha) = 1$. *Then* `CEVL` *is in* $\mathcal{P}$ *and every problem in* $\mathcal{P}$ *is log-space Karp-reducible to* `CEVL`.

Proof Sketch: Recall that the observation underlying the proof of Theorem 2.20 (and used also in the proof of Theorem 3.6) is that the computation of a Turing machine can be emulated by a "highly uniform" circuit. In the proof of Theorem 2.20, we hardwired the input to the reduction (denoted x) into the circuit (denoted C_x) and introduced input terminals corresponding to the bits of the NP-witness (denoted y). In the current context we leave x as an input to the circuit, while noting that the auxiliary NP-witness does not exists (or has length zero). Thus, the reduction from $S \in \mathcal{P}$ to `CEVL` maps the instance x (for S) to the pair

$(C_{|x|}, x)$, where $C_{|x|}$ is a circuit that emulates the computation of the machine that decides membership in S (on any $|x|$-bit long input). □

The impact of P-completeness under log-space reductions. Indeed Theorem 5.4 implies that $\mathcal{L} \neq \mathcal{P}$ if any only if CEVL $\notin \mathcal{L}$. Other natural problems were proved to have the same property (i.e., being P-complete under log-space reductions; cf. [57]).

Log-space reductions are used to define completeness with respect to other classes that are assumed to extend beyond $\mathcal{L}$. This restriction of the power of the reduction is definitely needed when the class of interest is contained in $\mathcal{P}$ (e.g., $\mathcal{NL}$, see Section 5.3.2). In general, we say that a problem Π is $\mathcal{C}$-complete under log-space reductions if Π is in $\mathcal{C}$ and every problem in $\mathcal{C}$ is log-space (many-to-one) reducible to Π. In such a case, if $\Pi \in \mathcal{L}$ then $\mathcal{C} \subseteq \mathcal{L}$.

As in the case of polynomial-time reductions, we wish to stress that the relevance of log-space reductions extends beyond being a tool for defining complete problems.

5.2.3 Log-Space uniformity and stronger notions

Strengthening Definition 3.3, we say that a family of circuits $(C_n)_n$ is log-space uniform if there exists an algorithm A that on input n outputs C_n while using space that is logarithmic in the size of C_n. As implied by Theorem 5.5 (and implicitly proved in Theorem 5.4), *the computation of any polynomial-time algorithm can be emulated by a log-space uniform family of* (bounded fan-in) *polynomial-size circuits.* On the other hand, in continuation to Section 5.1.3, we note that *log-space uniform circuits of bounded fan-in and logarithmic depth can be emulated by an algorithm of logarithmic space complexity* (i.e., $\mathcal{NC}^1$ is in log-space; see Exercise 5.7).

As mentioned in Section 3.1.1, stronger notions of uniformity have been considered. Specifically, in analogy to the discussion in §E.2.1.2, we say that $(C_n)_n$ has a strongly explicit construction if there exists an algorithm that runs in polynomial-time and linear-space such that, on input n and v, the algorithm returns the label of vertex v in C_n as well as the list of its children (or an indication that v is not a vertex in C_n). Note that if $(C_n)_n$ has a strongly explicit construction then it is log-space uniform, because the length of the description of a vertex in C_n is logarithmic in the size of C_n. The proof of Theorem 5.4 actually establishes the following.

Theorem 5.5 (strongly uniform circuits emulating $\mathcal{P}$): *For every polynomial-time algorithm A there exists a strongly explicit construction of a family of polynomial-size circuits $(C_n)_n$ such that for every x it holds that $C_{|x|}(x) = A(x)$.*

5.2.4 Undirected Connectivity

Exploring a graph (e.g., towards determining its connectivity) is one of the most basic and ubiquitous computational tasks regarding graphs. The standard graph exploration algorithms (e.g., BFS and DFS) require temporary storage that is linear in the number of vertices. In contrast, the algorithm presented in this section uses temporary storage that is only logarithmic in the number of vertices. In addition to demonstrating the power of log-space computation, this algorithm (or rather its actual implementation) provides a taste of the type of issues arising in the design of sophisticated log-space algorithms.

The intuitive task of "exploring a graph" is captured by the task of *deciding whether a given graph is connected.*[7] In addition to the intrinsic interest in this natural computational problem, we mention that it is computationally equivalent (under log-space reductions) to numerous other computational problems (see, e.g., Exercise 5.12). We note that some seemingly related problems seem harder; for example, determining directed connectivity (in directed graphs) captures the essence of the class $\mathcal{NL}$ (see Section 5.3.2). In view of this situation, we emphasize the fact that the computational problem considered here refers to undirected graphs by calling it undirected connectivity.

Theorem 5.6 *Deciding undirected connectivity* (UCONN) *is in* $\mathcal{L}$

The algorithm is based on the fact that UCONN is easy in the special case that the graph consists of a collection of constant degree expanders (see Appendix E.2). In particular, if the graph has constant degree and logarithmic diameter then it can be explored using a logarithmic amount of space (which is used for determining a generic path from a fixed starting vertex).[8]

However, the input graph does not necessarily consist of a collection of constant degree expanders. The main idea is then to transform the input graph into one that does satisfy the aforementioned condition, while preserving the number of connected components of the graph. Needless to say, the key point is performing such a transformation in logarithmic space. The rest of this section is devoted to the description of such a transformation. We first present the basic approach and next turn to the highly non-trivial implementation details.

Teaching note: We recommend leaving the actual proof of Theorem 5.6 for advanced reading. The main reason is its heavy dependence on technical material that is beyond the scope of a course in complexity theory.

We first note that it is easy to transform the input graph $G_0 = (V_0, E_0)$ into a constant-degree graph G_1 that preserves the number of connected components in G_0. Specifically, each vertex $v \in V$ having degree $d(v)$ (in G_0) is represented by a cycle C_v of $d(v)$ vertices (in G_1), and each edge $\{u, v\} \in E_0$ is replaced by an edge having one end-point on the cycle C_v and the other end-point on the cycle C_u such that each vertex in G_1 has degree three (i.e., has two cycle edges and a single intra-cycle edge). This transformation can be performed using logarithmic space, and

[7] See Appendix G.1 for basic terminology.

[8] Indeed, this is analogous to the circuit evaluation algorithm of Section 5.1.3, where the circuit depth corresponds to the diameter and the bounded fan-in corresponds to the constant degree. For further details, see Exercise 5.9.

thus (relying on Lemma 5.2) we assume throughout the rest of the proof that the input graph has degree three. Our goal is to transform this graph into a collection of expanders, while maintaining the number of connected components. In fact, *we shall describe the transformation while pretending that the graph is connected, while noting that otherwise the transformation acts separately on each connected component.*

A couple of technicalities. For a constant integer $d > 2$ determined so as to satisfy some additional condition, we may assume that the input graph is actually d^2-regular (albeit is not necessarily simple). Furthermore, we shall assume that this graph is not bipartite. Both assumptions can be justified by augmenting the aforementioned construction of a 3-regular graph by adding $d^2 - 3$ self-loops to each vertex.

Prerequisites: Needless to say, the aforementioned transformation refers to the notion of an expander graph (as defined in §E.2.1.1). The transformation also relies on the *zig-zag product* defined in §E.2.2.2.

5.2.4.1 The basic approach

Recall that our goal is to transform G_1 into an expander. The transformation is gradual and consists of logarithmically many iterations, where in each iteration an adequate expansion parameter doubles while the graph becomes a constant factor larger and maintains the degree bound. The (expansion) parameter of interest is the gap between the relative second eigenvalue of the graph and 1 (see §E.2.1.1). A constant value of this parameter indicates that the graph is an expander. Initially, this parameter is lower-bounded by $1/O(n^2)$, where n is the size of the graph, and after logarithmically many iterations this parameter is lower-bounded by a constant (and the current graph is an expander).

The crux of the aforementioned gradual transformation is the transformation that takes place in each single iteration. This transformation combines the standard graph powering (to a constant power c) and the *zig-zag product* presented in §E.2.2.2. Specifically, for adequate positive integers d and c, we start with the d^2-regular graph $G_1 = (V_1, E_1)$, and go through a logarithmic number of iterations letting $G_{i+1} = G_i^c \circledz G$ for $i = 1, ..., t-1$, where G is a fixed d-regular graph with d^{2c} vertices. That is, in each iteration, we raise the current graph (i.e., G_i) to the power c and combine the resulting graph with the fixed graph G using the zig-zag product. Thus, G_i is a d^2-regular graph with $d^{(i-1)\cdot 2c} \cdot |V_1|$ vertices, where this invariant is preserved by definition of the zig-zag product.

The analysis of the improvement in the expansion parameter, denoted $\delta_2(\cdot) \stackrel{\text{def}}{=} 1 - \bar{\lambda}_2(\cdot)$, relies on Eq. (E.9). Recall that Eq. (E.9) implies that if $\bar{\lambda}_2(G) < 1/2$ then $1 - \bar{\lambda}_2(G' \circledz G) > (1 - \bar{\lambda}_2(G'))/3$. Thus, the fixed graph G is selected such that $\bar{\lambda}_2(G) < 1/2$, which requires a sufficiently large constant d. Thus, we have

$$\delta_2(G_{i+1}) \;=\; 1 - \bar{\lambda}_2(G_i^c \circledz G) \;>\; \frac{1 - \bar{\lambda}_2(G_i^c)}{3} \;=\; \frac{1 - \bar{\lambda}_2(G_i)^c}{3}$$

whereas, for sufficiently large constant c, it holds that $1-\bar{\lambda}_2(G_i)^c > \max(6\cdot(1-\bar{\lambda}_2(G_i)),1/2)$. It follows that that $\delta_2(G_{i+1}) > \max(2\delta_2(G_i),1/6)$. Thus, setting $t=O(\log|V_1|)$ and using $\delta_2(G_1)=1-\bar{\lambda}_2(G_1)=\Omega(|V_1|^{-2})$, we obtain $\delta_2(G_t)>1/6$ as desired.

One detail of crucial importance is the ability to transform G_1 into G_t via a log-space computation. Indeed, the transformation of G_i to G_{i+1} can be performed in logarithmic space (see Exercise 5.10), but we need to compose a logarithmic number of such transformations. Unfortunately, the standard composition lemmas for space-bounded algorithms involve overhead that we cannot afford.[9] Still, taking a closer look at the transformation of G_i to G_{i+1}, one may note that it is highly structured and in some sense it can be implemented in constant space and supports a stronger composition result that incurs only a constant amount of storage per iteration. The resulting implementation (of the iterative transformation of G_1 to G_t) and the underlying formalism will be the subject of §5.2.4.2. (An alternative implementation, provided in [177], can be obtained by unraveling the composition.)

5.2.4.2 The actual implementation

The space-efficient implementation of the iterative transformation outlined in §5.2.4.1 is based on the observation that we do not need to explicitly construct the various graphs but merely provide "oracle access" to them. This observation is crucial when applied to the intermediate graphs; rather than constructing G_{i+1}, when given G_i as input, we show how to provide oracle access to G_{i+1} (i.e., answer "neighborhood queries" regarding G_{i+1}) when given oracle access to G_i (i.e., an oracle that answers neighborhood queries regarding G_i). That is, we view G_i and G_{i+1} (or rather their incidence lists) as functions (to be evaluated) rather than as strings (to be printed), and show how to reduce the task of finding neighbors in G_{i+1} (i.e., evaluating the "incidence function" at a given vertex) to the task of finding neighbors in G_i.

A clarifying discussion. Note that here we are referring to oracle machines that access a finite oracle, which represents a *finite variable object* (which in turn is an instance of some computational problem). Such a machine provides access to a complex object by using its access to a more basic object, which is represented by the oracle. Specifically, such a machine get an input, which is a "query" regarding the complex object (i.e, the object that the machine tries to emulate), and produce an output (which is the answer to the query). Analogously, these machines make queries, which are queries regarding another object (i.e., the one represented in the oracle), and obtain corresponding answers.[10]

[9] Needless to say, we cannot afford the naive composition (of Lemma 5.1), since it causes an overhead linear in the size of the intermediate output. As for the emulative composition (of Lemma 5.2), it sums up the space complexities of the composed algorithms (not to mention adding another logarithmic term), which would result in a log-squared bound on the space complexity.

[10] Indeed, the current setting (in which the oracle represents a *finite variable object*, which in turn is an instance of some computational problem) is different from the standard setting, where the oracle represents a *fixed computational problem*. Still the mechanism (and/or operations) of these two types of oracle machines is the same: They both get an input (which here is a

Like in §5.1.2.3, queries are made via a special write-only device and the answers are read from a corresponding read-only device, where the use of these devices is not charged in the space complexity. With these conventions in place, we claim that neighborhoods in the d^2-regular graph G_{i+1} can be computed by a constant-space oracle machine that is given oracle access to the d^2-regular graph G_i. That is, letting $g_i : V_i \times [d^2] \to V_i \times [d^2]$ (resp., $g_{i+1} : V_{i+1} \times [d^2] \to V_{i+1} \times [d^2]$) denote the edge rotation function[11] of G_i (resp., G_{i+1}), we have:

Claim 5.7 *There exists a constant-space oracle machine that evaluates g_{i+1} when given oracle access to g_i, where the state of the machine is counted in the space complexity.*

Proof Sketch: We first show that the two basic operation that underly the definition of G_{i+1} (i.e., powering and zig-zag product with a constant graph) can be performed in constant-space.

The edge rotation function of the square of a graph can be evaluated at $(v, \langle j_1, j_2 \rangle)$, using a constant amount of space, by evaluating the edge rotation function of the original graph twice. First, by making the query (v, j_1) we obtain the edge rotation of (v, j_1), denoted (u, k_1), and next, making the query (u, j_2), we obtain (w, k_2) and output $(w, \langle k_2, k_1 \rangle)$. We stress that we only use the temporary storage to record k_1, whereas u is directly copied from the oracle answer device to the oracle query device. Accounting also for a constant number of states needed for the various stages of the foregoing activity, we conclude that graph squaring can be performed in constant-space. The argument extends to the task of raising the graph to any constant power.

Turning to the zig-zag product (of G' with a fixed G), we note that the corresponding edge rotation function can be evaluated in constant-space (given oracle access to the edge rotation function of G'). This follows directly from Eq. (E.7), noting that the latter calls for a single evaluation of the edge rotation function of G' and two simple modifications that only depend on the constant-size graph G. Again, using the fact that it suffices to copy vertex names from the input (or oracle answer device) to the oracle query device (or output), we conclude that the aforementioned activity can be performed using constant space.

The argument extends to a sequential composition of a constant number of operations of the aforementioned type (i.e., graph squaring and zig-zag product with a constant graph). $\square$

Recursive composition. Using Claim 5.7, we wish to obtain a log-space oracle machine that evaluates g_t by making oracle calls to g_1, where $t = O(\log |V_1|)$. Such

"query" regarding a variable object rather than an instance of a fixed computational problem), and produce an output (which here is the answer to the query rather than a "solution" for the given instance). Analogously, these machines make queries (which here are queries regarding another variable object rather than queries regarding another fixed computational problem), and obtain corresponding answers.

[11] Recall that the edge rotation function of a graph maps the pair (v, j) to the pair (u, k) if vertex u is the j^{th} neighbor of vertex v and v is the k^{th} neighbor of u (see §E.2.2.2).

an oracle machine will yield a log-space transformation of G_1 to G_t (by evaluating g_t at all possible values). It is tempting to hope that an adequate composition lemma, when applied to Claim 5.7, will yield the desired log-space oracle machine (reducing the evaluation of g_t to g_1). This is indeed the case, except that the adequate composition lemma is still to be developed (as we do next).

We first note that applying a naive composition (as in Lemma 5.1) amounts to an additive overhead of $O(\log |V_1|)$ *per each composition*. But we cannot afford more than an amortized constant additive overhead per composition. Applying the emulative composition (as in Lemma 5.2) amounts in multiplicative overhead (see §5.1.2.3), which is certainly unaffordable. The composition developed next is a variant of the naive composition, which is beneficial in the context of recursive calls. The basic idea is deviating from the paradigm that allocates separate input/output and query devices to each level in the recursion, and combining all these devices in a single ("global") device which will be used by all levels of the recursion. That is, rather than following the "structured programming" methodology of using (locally) designated space for passing information to the subroutine, we use the "bad programming" methodology of passing information through global variables. As usual, this notion is formulated by referring to the model of multi-tape Turing machine, but it can be formulated in any other reasonable model of computation.

Definition 5.8 (global-tape oracle machines): *A* global-tape oracle machine *is defined as an oracle machine* (cf. Definition 1.11)*, except that the input, output and oracle tapes are replaced by a single* global tape*. In addition, the machine has a constant number of work tapes, called the* local tapes*. The machine obtains its input from the global tape, writes each query on this very tape, obtains the corresponding answer from this tape, and writes its final output on this tape. The space complexity of such a machine is stated when referring separately to the use of the global tape and to the use of the local tapes.*

Clearly, any ordinary oracle machine can be converted into an equivalent global-tape oracle machine. The resulting machine uses a global tape of length at most $n + \ell + m$, where n denotes the length of the input, ℓ denote the length of the longest query or oracle answer, and m denotes the length of the output. However, combining the different tapes into one global tape seems to require holding separate pointers for each of the original tapes, which means that the local tape has to be used to store corresponding counters (in addition to the original work-tape). A key observation is that these counters can be avoided in the case that the original machine can be described as a sequence of transformations (of the input into the first query, and of the i^{th} answer to the $i+1^{\text{st}}$ query or the output), while maintaining auxiliary information on the work-tape. Indeed, the machine presented in the proof of Claim 5.7 has this form, and thus can be implemented by a global-tape oracle machine that uses a global-tape not longer than its input and a local-tape of constant length.

Claim 5.9 (Claim 5.7, revisited): *There exists a global-tape oracle machine that evaluates g_{i+1} when given oracle access to g_i, while using global tape of length $\log_2(d^2|V_{i+1}|)$ and a local tape of constant length.*

Proof Sketch: Following the proof of Claim 5.7, we merely indicate the exact use of the two tapes. For example, recall that the edge rotation function of the square of G_i is evaluated at $(v, \langle j_1, j_2 \rangle)$ by evaluating the edge rotation function of the original graph first at (v, j_1) and then at (u, j_2), where $(u, k_1) = g_i(v, j_1)$. This means the global-tape machine first reads $(v, \langle j_1, j_2 \rangle)$ from the global tape and replaces it by the query (v, j_1), while storing j_2 on the local tape. Thus, the machine merely deletes a constant number of bits from the global tape (and leaves its prefix intact). After invoking the oracle, the machine copies k_1 from the global tape (which currently holds (u, k_1)) to its local tape, and copies j_2 from its local tape to the global tape (such that it contains (u, j_2)). After invoking the oracle for the second time, the global tape contains $(w, k_2) = g_i(u, j_2)$, and the machine merely modifies it to $(w, \langle k_2, k_1 \rangle)$, which is the desired output.

Similarly, the edge rotation function of the zig-zag product of G' with a fixed G is evaluated at $(\langle u, i \rangle, \langle \alpha, \beta \rangle)$ by querying G' at $(u, E_\alpha(i))$ and outputting $(\langle v, E_\beta(j') \rangle, \langle \beta, \alpha \rangle)$, where (v, j') denotes the oracle answer (see Eq. (E.7)). This means that the global-tape oracle machine first copies α, β from the global-tape to the local-tape, transforms the contents of the global tape from $(\langle u, i \rangle, \langle \alpha, \beta \rangle)$ to $(u, E_\alpha(i))$, and makes an analogous transformation after the oracle is invoked. □

Composing global-tape oracle machines. In the proof of Claim 5.9, we implicitly used sequential composition of computations conducted by global-tape oracle machines.[12] In general, when sequentially composing such computations the global-tape and local-tape usage are the maximum among all composed computations; that is, the current formalism offers a tight bound on naive composition (as opposed to Lemma 5.1). Furthermore, global-tape oracle machines are beneficial in the context of recursive composition, as indicated by Lemma 5.10 (which relies on this model in a crucial way). The key observation is that all levels in the recursive composition may re-use the same global storage, and only the local storage gets added. Consequently we have the following composition lemma, where n denotes the length of the input to f_1 (and t may depend on n).

Lemma 5.10 (recursive composition in the global-tape model): *Suppose that, for every $i = 1, ..., t-1$, there exists a global-tape oracle machine that computes f_i by making oracle calls to f_{i+1} while using a global-tape of length $L(n)$ and a local-tape of length $l_i(n)$, which also accounts for the machine's state. Then f_1 can be computed by a standard oracle machine that makes calls to f_t and uses space $L(n) + \sum_{i=1}^{t-1} O(l_i(n))$.*

We shall apply this lemma with $f_i = g_{t+1-i}$, using the bounds $L(n) = O(n)$ and $l_i(n) = O(1)$ (as guaranteed by Claim 5.9), where in this application $n = \log_2(d^2|V_t|)$ is the length of the input to $f_1 = g_t$ and $t = O(\log|V_1|) = O(\log|V_t|)$.

Proof Sketch: We compute f_1 by allocating space for the emulation of the global-tape and the local-tapes of each level in the recursion. We emulate the recursive

[12] A similar composition took place in the proof of Claim 5.7, but here we assert a stronger feature of this specific computation.

computation by capitalizing on the fact that all recursive levels use the same global-tape (for making queries and receiving answers). In the actual recursion, each level may use the global-tape arbitrarily as long as when it returns control to the invoking machine the global-tape contains the right answer. Thus, the emulation may do the same, and emulate each recursive call by using the space allocated for the global-tape as well as the space designated for the local-tape of this level. □

Conclusion. Combining Claim 5.9 and Lemma 5.10, we conclude that the evaluation of $g_{O(\log |V_1|)}$ can be reduced to the evaluation of g_1 in space $O(\log |V_1|)$. Recalling that G_1 can be constructed in log-space (based on the input graph G_0), we infer that $G' = G_{O(\log |V_1|)}$ can be constructed in log-space. Theorem 5.6 follows by recalling that G' (which has constant degree and logarithmic diameter) can be tested for connectivity in log-space (see Exercise 5.9). Using a similar argument, we can test whether a given pair of vertices are connected in the input graph (see Exercise 5.11).

5.3 Non-Deterministic Space Complexity

The difference between space-complexity and time-complexity is quite striking in the context of non-deterministic computations. One aspect is the huge gap between the power of two formulation of non-deterministic space complexity (see Section 5.3.1), while in contrast the analogous formulations are equivalent in the context of time-complexity. Another aspect is the contrast between the results regarding (the standard model of) non-deterministic space-bounded computation (see Section 5.3.2) and the analogous questions in the context of time-complexity.

5.3.1 Two models

Recall that non-deterministic time-bounded computations were defined via two equivalent models. In the off-line model (underlying the definition of NP as a proof system (see Definition 2.5)) non-determinism is captured by reference to the existential choice of an auxiliary ("non-deterministic") input. In contrast, in the on-line model (underlying the traditional definition of NP (see Definition 2.7)) non-determinism is captured by reference to the non-deterministic choices of the machine itself. These models are equivalent because the latter on-line choices can be recorded (almost) for free (see the proof of Theorem 2.8). That is, while it is clear that the off-line model can emulate the on-line model (i.e., the off-line machine can emulate on-line choices by using its non-deterministic input), the emulation of the off-line model by the on-line model is enabled by the fact that an on-line machine may store (and re-use) a sequence of non-deterministic (on-line) choices.

The naive emulation of the off-line model on the on-line model is not possible in the context of space-bounded computation, because the number of non-deterministic choices is typically much larger than the space-bound. The models become equivalent only if the off-line model is restricted to access its non-deterministic input in a uni-directional manner. Let us formulate the two models and consider them more closely.

In the standard model, called the on-line model, the machine makes non-deterministic choices "on the fly" (or alternatively reads a non-deterministic input from a special read-only tape *that can be read only in a uni-directional way*). Thus, if the machine needs to refer to such a non-deterministic choice at a latter stage in its computation, then it must store the choice on its storage device (and be charged for it). In contrast, in the so-called off-line model the non-deterministic choices (or the bits of the non-deterministic input) are read from a read-only device (or tape) *that can be scanned in both directions like the main input.*

We denote by $\text{NSPACE}_{\text{on-line}}(s)$ (resp., $\text{NSPACE}_{\text{off-line}}(s)$) the class of sets that are acceptable by an on-line (resp., off-line) non-deterministic machine having space complexity s. We stress that, as in Definition 2.7, the set accepted by a non-deterministic machine M is the set of strings x such that there exists a computation of $M(x)$ that is accepting. Clearly, $\text{NSPACE}_{\text{on-line}}(s) \subseteq \text{NSPACE}_{\text{off-line}}(s)$. On the other hand, not only that $\text{NSPACE}_{\text{on-line}}(s) \neq \text{NSPACE}_{\text{off-line}}(s)$ but rather $\text{NSPACE}_{\text{on-line}}(s) = \text{NSPACE}_{\text{off-line}}(\Theta(\log s))$, provided that s is at least linear. For details, see Exercise 5.14.

Before proceeding any further, let us justify the focus on the on-line model in the rest of this section. Indeed, the off-line model fits better the motivations to $\mathcal{NP}$ (as presented in Section 2.1.2), but the on-line model seems more adequate for the study of non-deterministic in the context of space complexity. One reason is that an off-line non-deterministic input can be used to code computations (see Exercise 5.14), and in a sense allows to "cheat" with respect to the "actual" space complexity of the computation. This is reflected in the fact that the off-line model can emulate the on-line model while using space that is logarithmic in the space used by the on-line model. A related phenomenon is that $\text{NSPACE}_{\text{off-line}}(s)$ is only known to be contained in $\text{DTIME}(2^{2^s})$, whereas $\text{NSPACE}_{\text{on-line}}(s) \subseteq \text{DTIME}(2^s)$. This fact motivates the study of $\mathcal{NL} = \text{NSPACE}_{\text{on-line}}(\texttt{log})$, as a study of a (natural) sub-class of $\mathcal{P}$. Indeed, the various results regarding $\mathcal{NL}$ justify its study in retrospect.

In light of the foregoing, we adopt the standard conventions and let $\text{NSPACE}(s) = \text{NSPACE}_{\text{on-line}}(s)$. Our main focus will be the study of $\mathcal{NL} = \text{NSPACE}(\texttt{log})$.

5.3.2 NL and directed connectivity

This section is devoted to the study of $\mathcal{NL}$, which we view as the non-deterministic analogue of $\mathcal{L}$. Specifically, $\mathcal{NL} = \cup_c \text{NSPACE}(\ell_c)$, where $\ell_c(n) = c \log_2 n$ (and the definitional issues pertaining $\text{NSPACE} = \text{NSPACE}_{\text{on-line}}$ are discussed in Section 5.3.1).

We first note that the proof of Theorem 5.3 can be easily extended to the (on-line) non-deterministic context. The reason being that moving from the deterministic model to the current model does not affect the number of instantaneous configurations (as defined in the proof of Theorem 5.3), whereas this number bounds the time complexity. Thus, $\mathcal{NL} \subseteq \mathcal{P}$.

The following problem, called directed connectivity (st-CONN), captures the essence of non-deterministic log-space computations (and, in particular, is complete for $\mathcal{NL}$ under log-space reductions). The input to st-CONN consists of a directed graph $G = (V, E)$ and a pair of vertices (s, t), and the task is to determine whether there exists a directed path from s to t (in G).[13] Indeed, the study of $\mathcal{NL}$ is often conducted via st-CONN. For example, note that $\mathcal{NL} \subseteq \mathcal{P}$ follows easily from the fact that st-CONN is in $\mathcal{P}$ (and the fact that $\mathcal{NL}$ is log-space reducible to st-CONN).

5.3.2.1 Completeness and beyond

Clearly, st-CONN is in $\mathcal{NL}$ (see Exercise 5.15). The $\mathcal{NL}$-completeness of st-CONN under log-space reductions follows by noting that the computation of any non-deterministic space-bounded machine yields a directed graph in which vertices correspond to possible configurations and edges represent the "successive" relation of the computation. In particular, for log-space computations the graph has polynomial size, but in general the relevant graph is strongly explicit (in a natural sense; see Exercise 5.16).

Theorem 5.11 *Every problem in $\mathcal{NL}$ is log-space reducible to* st-CONN (via a many-to-one reduction).

Proof Sketch: Fixing a non-deterministic (on-line) machine M and an input x, we consider the following directed graph $G_x = (V_x, E_x)$. The vertices of V_x are possible instantaneous configurations of $M(x)$, where each configuration consists of the contents of the work-tape (and the machine's finite state), the machine's location on it, and the machine's location on the input. The directed edges represent possible single moves in such a computation. We stress that such a move depends on the machine M as well as on the (single) bit of x that resides in the location specified by the first configuration (i.e., the configuration corresponding to the start-point of the potential edge).[14] Note that, given x, the graph G_x can be constructed in log-space (by scanning all pairs of vertices and outputting only the pairs that are valid edges (which, in turn, can be tested in constant-space)).

By definition, the graph G_x represents the possible computations of M on input x. In particular, there exists an accepting computation of M on input x if and only if there exists a directed path, in G_x, starting at the vertex s that corresponds to the initial configuration and ending at the vertex t that corresponds to a canonical accepting configuration. Thus, $x \in S$ if and only if (G_x, s, t) is a yes-instance of st-CONN. $\square$

[13] See Appendix G.1 for basic graph theoretic terminology. We note that, here (and in the sequel), s stands for *start* and t stands for *terminate*.

[14] Thus, the actual input x only affects the set of edges of G_x (whereas the set of vertices is only affected by $|x|$). A related construction is obtained by incorporating in the configuration also the (single) bit of x that resides in the machine's location on the input. In the latter case, x itself also affects V_x.

Reflection. We believe that the proof of Theorem 5.11 (see also Exercise 5.16) justifies saying that `st-CONN` captures the essence of non-deterministic space-bounded computations. Note that this (intuitive and imformal) statement goes beyond saying that `st-CONN` is $\mathcal{NL}$-complete under log-space reductions.

We note the discrepancy between the status of undirected connectivity (see Theorem 5.6 and Exercise 5.11) and directed connectivity (see Theorem 5.11). In this context it is worthwhile to note that determining the existence of relatively short paths (rather than arbitrary paths) in undirected (or directed) graphs is also $\mathcal{NL}$-complete under log-space reductions; see Exercise 5.18.

5.3.2.2 Relating NSPACE to DSPACE

Recall that in the context of time complexity, converting non-deterministic computation to deterministic computation is known only when allowing an exponential blow-up in the complexity. In contrast, space complexity allows such a conversion at the cost of a polynomial blow-up in the complexity.

Theorem 5.12 (Non-deterministic versus deterministic space): *For any space-constructible $s : \mathbb{N} \to \mathbb{N}$ that is at least logarithmic, it holds that* $\textsc{Nspace}(s) \subseteq \textsc{Dspace}(O(s^2))$.

In particular, non-deterministic polynomial-space is contained in deterministic polynomial-space (and non-deterministic poly-logarithmic space is contained in deterministic poly-logarithmic space).

Proof Sketch: We focus on the special case of $\mathcal{NL}$ and the argument extends easily to the general case. Alternatively, the general statement can be derived from the special case by using a suitable upwards-translation lemma (see, e.g., [116, Sec. 12.5]). The special case boils down to presenting a log-square space algorithm for deciding directed connectivity.

The basic idea is that checking whether or not there is a path of length at most ℓ from u to v in G, reduces (in log-space) to checking whether there is an intermediate vertex w such that there is a path of length at most $\lceil \ell/2 \rceil$ from u to w and a path of length at most $\lfloor \ell/2 \rfloor$ from w to v. Let $\phi_G(u,v,\ell) \stackrel{\rm def}{=} 1$ if there is a path of length at most ℓ from u to v in G and $\phi_G(u,v,\ell) \stackrel{\rm def}{=} 0$ otherwise. Thus, $\phi_G(u,v,\ell)$ can be decided recursively by scanning all vertices w in G, and checking for each w whether both $\phi_G(u,w,\lceil \ell/2 \rceil) = 1$ and $\phi_G(w,v,\lfloor \ell/2 \rfloor) = 1$ hold. We stress that all these computations re-use the same space, and we need only store one additional bit representing the results of all prior computations.

Turning back to the origianl problem, note that, given a directed graph $G = (V,E)$ and a pair of vertices (s,t), we should merely compute $\phi_G(s,t,|V|)$. This is done by invoking a recursive procedure that computes $\phi_G(u,v,\ell)$ by scanning all vertices in G, and computing for each vertex w the values of $\phi_G(u,w,\lceil \ell/2 \rceil)$ and $\phi_G(w,v,\lfloor \ell/2 \rfloor)$. We return the value 1 if and only if for some w it holds that $\phi_G(u,w,\lceil \ell/2 \rceil) = \phi_G(w,v,\lfloor \ell/2 \rfloor) = 1$ (see Figure 5.2). Needless to say, $\phi_G(u,v,1)$ and $\phi_G(u,v,0)$ can be decided easily in logarithmic space.

Recursive computation of $\phi_G(u,v,2\ell)$, for $\ell \geq 1$.	
For $w = 1, ..., \|V\|$ do begin	(*storing the vertex name*)
Compute $\sigma \leftarrow \phi_G(u,w,\ell)$	(*by a recursive call*)
Compute $\sigma \leftarrow \sigma \wedge \phi_G(w,v,\ell)$	(*by a second recursive call*)
If $\sigma = 1$ then `return 1`.	(*success: an intermediate vertex was found*)
End	(*of scan*).
`return 0`.	(*reached only if the scan was completed without success*).

Figure 5.2: The recursive procedure in $\mathcal{NL} \subseteq \text{DSPACE}(O(\log^2))$.

The amount of space taken by each level of the recursion is $\log_2 |V|$ (for storing the current value of w), and the number of levels is $\log_2 |V|$. We stress that when computing $\phi_G(u,v,\ell)$, we make many recursive calls, but all these calls re-use the same work space. That is, when we compute $\phi_G(u,w,\lceil \ell/2 \rceil)$ we re-use the space that was used for computing $\phi_G(u,w',\lfloor \ell/2 \rfloor)$ for the previous w', and we re-use the same space when we compute $\phi_G(w,v,\lfloor \ell/2 \rfloor)$. The theorem follows. □

Digest. The proof of Theorem 5.12 relies on two main observations. The first observation is that an existential claim can be verifying by scanning all possible values in the relevant domain, which in terms of space complexity has a cost that is logarithmic in the size of the domain. The second observation is that a disjunction (resp., conjunction) of two Boolean conditions can be verifier using space $s + O(1)$, where s is the space complexity of verifying a single condition. This follows by applying naive composition (i.e., Lemma 5.1). The proof of Theorem 5.12 is facilitated by the fact that we may consider a concrete and simple computational problem such as `st-CONN`. Nevertheless, the same ideas can be applied directly to $\mathcal{NL}$ (or ant NSPACE class).

The simple formulation of `st-CONN` facilitates placing $\mathcal{NL}$ in complexity classes such as $\mathcal{NC}^2$. All that is needed is observing that `st-CONN` can be solved by raising the adequate matrix (i.e., the adjacency matrix of the graph augmented with 1-entries on the diagonal) to the adequate power (i.e., its dimension). Squaring a matrix can be done by a uniform family of logarithmic depth bounded fan-in circuits (i.e., in NC1), and by repeated squaring the n^{th} power of an n-by-n matrix can be computed by a uniform family of bounded fan-in circuits of polynomial size and depth $O(\log^2 n)$; thus, $\mathcal{NL} \subseteq \mathcal{NC}^2$.

5.3.2.3 Complementation or NL=coNL

Recall that (natural) non-deterministic time-complexity classes are not known to be closed under complementation. Furthermore, it is widely believed that $\mathcal{NP} \neq \text{co}\mathcal{NP}$. In contrast, (natural) non-deterministic space-complexity classes are closed under complementation, as captured by the result $\mathcal{NL} = \text{co}\mathcal{NL}$, where co$\mathcal{NL} \stackrel{\rm def}{=} \{\{0,1\}^* \setminus S : S \in \mathcal{NL}\}$. Before proving this result, let us take a closer look at the problem it resolves (which we rephrase as $\mathcal{NL}$ versus $\mathcal{NL} \cap {\rm co}\mathcal{NL}$).

Recall that a set S is in $\mathcal{NL}$ if there exists a non-deterministic log-space machine M that accepts S, and that the acceptance condition of non-deterministic machines is asymmetric in nature. That is, $x \in S$ implies the *existence* of an accepting computation of M on input x, whereas $x \notin S$ implies that *all* computations of M on input x are non-accepting. Thus, the existence of a accepting computation of M on input x is an absolute indication for $x \in S$, but the existence of a rejecting computation of M on input x is not an absolute indication for $x \notin S$. We note that, for $S \in \mathcal{NL} \cap {\rm co}\mathcal{NL}$, there exist absolute indications both for $x \in S$ and for $x \notin S$ (or, equivalently for $x \in \overline{S} \stackrel{\rm def}{=} \{0,1\}^* \setminus S$), where each of the two types of indication is provided by a different machine (i.e., the one accepting S or the one accepting $\overline{S}$). Combining both machines, we obtain a single non-deterministic machine that, for every input, sometimes outputs the correct answer and always outputs either the correct answer or a special ("don't know") symbol. This yields the following definition, which refers to Boolean functions as a special case.

Definition 5.13 (non-deterministic computation of functions): *We say that a* non-deterministic machine M computes the function $f : \{0,1\}^* \to \{0,1\}^*$ *if for every $x \in \{0,1\}^*$ the following two conditions hold.*

1. *Every computation of M on input x yields an output in $\{f(x), \perp\}$, where $\perp \notin \{0,1\}^*$ is a special symbol* (indicating "don't know").
2. *There exists a computation of M on input x that yields the output $f(x)$.*

Note that $S \in \mathcal{NL} \cap {\rm co}\mathcal{NL}$ if and only if there exists a non-deterministic log-space machine that computes χ_S, where $\chi_S(x) = 1$ if $x \in S$ and $\chi_S(x) = 0$ otherwise (see Exercise 5.20). It follows that $\mathcal{NL} = {\rm co}\mathcal{NL}$ if and only if for every $S \in \mathcal{NL}$ there exists a non-deterministic log-space machine that computes χ_S.

Theorem 5.14 ($\mathcal{NL} = {\rm co}\mathcal{NL}$): *For every $S \in \mathcal{NL}$ there exists a non-deterministic log-space machine that computes χ_S.*

As in the case of Theorem 5.12, the result extends to any space-constructible $s : \mathbb{N} \to \mathbb{N}$ that is at least logarithmic; that is, for such s and every $S \in$ NSPACE(s), it holds that $\{0,1\}^* \setminus S \in$ NSPACE$(O(s))$. This extension can be proved either by generalizing the following proof or by using an adequate upwards-translation lemma.

Proof Sketch: It suffices to present a non-deterministic (on-line) log-space machine that computes the characteristic function of `st-CONN`, denoted χ (i.e., $\chi(G,s,t) = 1$ if there is a directed path from s to t in G and $\chi(G,s,t) = 0$ otherwise).

We first show that the computation of χ is log-space reducible (by two queries)[15] to determining the number of vertices that are reachable from a given vertex in

[15]We stress the fact that two queries are used in the reduction, because this avoids the difficulties (discussed in §5.1.2.3) regarding emulative composition for general space-bounded reduction. Alternatively, we may use a version of the naive composition, while relying on the fact that the oracle queries are "highly related" to the input and that the answers have logarithmic length. For details, see Exercises 5.23 and 5.24.

a given graph. On input (G,s,t), the reduction computes the number of vertices that are reachable from s in the graph G and compares it to the number of vertices reachable from s in the graph obtained by deleting t from G. (An alternative reduction that uses a single query is presented in Exercise 5.22.) Note that if computing f is log-space reducible by a constant number of queries to computing some function g and there exists a non-deterministic log-space machine that computes g, then there exists a non-deterministic log-space machine that computes f (see Exercise 5.23). Thus, we focus on providing a non-deterministic log-space machine that compute the number of vertices that are reachable from a given vertex in a given graph.

Fixing an n-vertex graph $G=(V,E)$ and a vertex v, we consider the set of vertices that are reachable from v by a path of length at most i. We denote this set by R_i, and observe that $R_0=\{v\}$ and that for every $i=1,2,...$, it holds that

$$R_i = R_{i-1} \cup \{u : \exists w \in R_{i-1} \text{ s.t. } (w,u) \in E\} \tag{5.1}$$

Our aim is to compute $|R_n|$. This will be done in n iterations such that at the i^{th} iteration we compute $|R_i|$. When computing $|R_i|$ we rely on the fact that $|R_{i-1}|$ is known to us, which means that we shall store $|R_{i-1}|$ in memory. We stress that we discard $|R_{i-1}|$ from memory as soon as we complete the computation of $|R_i|$, which we store instead. Thus, at each iteration i, our record of past iterations only contains $|R_{i-1}|$.

Computing $|R_i|$. Given $|R_{i-1}|$, we non-deterministically compute $|R_i|$ by making a guess (for $|R_i|$), denoted g, and verifying its correctness as follows:

1. We verify that $|R_i| \geq g$ in a straightforward manner. That is, scanning V in some canonical order, we verify for g vertices that they are each in R_i. That is, during the scan, we select non-deterministically g vertices, and for each selected vertex w we verify that w is reachable from v by a path of length at most i, where this verification is performed by just guessing and verifying an adequate path (see Exercise 5.15).

   We use $\log_2 n$ bits to store the currently scanned vertex (i.e., w), and another $O(\log n)$ bits for implementing the verification of the existence of a path of length at most i from v to w.

2. The verification of the condition $|R_i| \leq g$ (equivalently, $|V \setminus R_i| \geq n-g$) is the interesting part of the procedure. Indeed, as we saw, demonstrating membership in R_i is easy, but here we wish to demonstrate non-membership in R_i. We so so by relying on the fact that we know $|R_{i-1}|$, which allows for a non-deterministic enumeration of R_{i-1} itself, which in turn allows for proofs of non-membership in R_i (via the use of Eq. (5.1)). Details follows (and an even more structured description is provided in Figure 5.3).

   Scanning V (again), we verify for $n-g$ (guessed) vertices that they are *not* in R_i (i.e., are *not* reachable from v by paths of length at most i). By Eq. (5.1), verifying that $u \notin R_i$ amounts to proving that for every $w \in R_{i-1}$, it holds that $u \neq w$ and $(w,u) \notin E$. As hinted, the knowledge of $|R_{i-1}|$ allows for the enumeration of R_{i-1}, and thus we merely check the aforementioned condition on each vertex in R_{i-1}. Thus, verifying that $u \notin R_i$ is done as follows.

(a) We scan V guessing $|R_{i-1}|$ vertices that are in R_{i-1}, and verify each such guess in the straightforward manner (i.e., as in Step 1).[16]

(b) For each $w \in R_{i-1}$ that was guessed and verified in Step 2a, we verify that both $u \neq w$ and $(w, u) \notin E$.

By Eq. (5.1), if u passes the foregoing verification then indeed $u \notin R_i$.

Note that we use $\log_2 n$ bits to store the vertex u, another $\log_2 n$ bits to count the number of vertices that are verified to be in R_{i-1}, another $\log_2 n$ bits to store such a vertex w, and another $O(\log n)$ bits for verifying that $w \in R_{i-1}$ (as in Step 1).

If any of the foregoing verifications fails, then the procedure halts outputting the "don't know" symbol $\perp$. Otherwise, it outputs g.

It can be verified that, when given the correct value of $|R_{i-1}|$, the foregoing non-deterministic log-space procedure computes the value of $|R_i|$. That is, if all verifications are satisfied then it must hold that $g = |R_i|$, and if $g = |R_i|$ then there are adequate non-deterministic choices that satisfy all verifications.

Recall that R_n is computed iteratively, starting with $|R_0| = 1$ and computing $|R_i|$ based on $|R_{i-1}|$. Each iteration $i = 1, ..., n$ is non-deterministic, and is either completed with the correct value of $|R_i|$ (at which point $|R_{i-1}|$ is discarded) or halts in failure (in which case we halt the entire process and output $\perp$). This yields a non-deterministic log-space machine for computing $|R_n|$, and the theorem follows. □

Digest. Step 2 is the heart of the proof (of Theorem 5.14). In this step a non-deterministic procedure is used to verify non-membership in an NL-type set. Indeed, verifying membership in NL or NP-type sets is the archetypical task of non-deterministic procedures (i.e., they are defined so to fit these tasks). The verification of non-membership is enabled, in the current context, by an iterative (non-deterministic) process that consumes an admissible amount of resources (i.e., a logarithmic amount of space).

5.3.3 Discussion

The current section may be viewed as a study of the "power of non-determinism in computation" (which is a somewhat contradictory term). Recall that we view non-deterministic processes as fictitious abstractions aimed at capturing fundamental phenomena such as verification of proofs (cf., Section 2.1.4). Since these fictitious abstractions are fundamental in the context of time-complexity, we may hope to

[16] Note that implicit in Step 2a is a non-deterministic procedure that computes the mapping $(G, v, i, |R_{i-1}|) \rightarrow R_{i-1}$, where R_{i-1} denotes the set of vertices that are reachable in G by a path of length at most i from v.

Given $|R_{i-1}|$ and a guess g, the claim $g \geq |R_i|$ is verified as follows.

Set $c \leftarrow 0$. (*initializing the main counter*)
For $u = 1, ..., n$ do begin (*the main scan*)
  Guess whether or not $u \in R_i$.
  For a negative guess (i.e., $u \notin R_i$), do begin
  (*Verify that* $u \notin R_i$ *via Eq.* (5.1).)
    Set $c' \leftarrow 0$. (*initializing a secondary counter*)
    For $w = 1, ..., n$ do begin (*the secondary scan*)
      Guess whether or not $w \in R_{i-1}$.
      For a positive guess (i.e., $w \in R_{i-1}$), do begin
        Verify that $w \in R_{i-1}$ (as in Step 1).
        Verify that $u \neq w$ and $(w, u) \notin E$.
        If some verification failed
        then halt with output $\perp$ otherwise increment c'.
      End (*of handling a positive guess for* $w \in R_{i-1}$).
    End (*of secondary scan*). (c' *vertices in* R_{i-1} *were checked*)
    If $c' < |R_{i-1}|$ then halt with output $\perp$.
    Otherwise ($c' = |R_{i-1}|$), increment c. (u *verified to be outside of* R_i)
  End (*of handling a negative guess for* $u \notin R_i$).
End (*of main scan*). (c *vertices were shown outside of* R_i)
If $c < n - g$ then halt with output $\perp$.
Otherwise $n - |R_i| \geq c \geq n - g$ is verified.

Figure 5.3: The main step in proving $\mathcal{NL} = \text{co}\mathcal{NL}$.

gain some understanding by a comparative study (conducted in the context of space-complexity). Furthermore, we may discover that non-deterministic space bounded machines give rise to interesting computational phenomena.

The aforementioned hopes seems to come true in the current section. For example, the fact that $\mathcal{NL} = \text{co}\mathcal{NL}$, while the common conjecture is that $\mathcal{NP} \neq \text{co}\mathcal{NP}$, indicates that the latter conjecture is less generic than sometimes stated. It is not that an existential quantifier cannot be "feasibly replaced" by a universal quantifier, but rather the feasibility of such a replacement depends very much of the type of notion of feasibility. Turning to the other type of benefits, we learned that `st-CONN` can be reduced in log-space to `st-unCONN` (i.e., the set of graphs in which there is no directed path between the two designated vertices; see Exercise 5.21).

Still, one may ask what does the class $\mathcal{NL}$ actually represent (beyond `st-CONN`, which is more than merely a complete problem for this class; see §5.3.2.1). Turning back to Section 5.3.1, we recall that the class $\text{NSPACE}_{\text{off-line}}$ captures the straightforward notion of space-bounded verification. In this model (called the off-line model), the alleged proof is written on a special device (similarly to the claim being proved by it), which is being read freely. In contrast, underlying the alternative class $\text{NSPACE}_{\text{on-line}}$ is a notion of proofs that are verified by reading them sequentially (rather than scanning them back and forth). In this case, if the verification procedure needs to relate to the currently read part of the proof in the future, then it must store the relevant part (and is charged for this storage). Thus, the on-line model underlying $\text{NSPACE}_{\text{on-line}}$ refers to the standard process of reading proofs in a sequential manner and making side notes for future verification, rather than scanning them back and forth all the time. In retrospect (see discussion in Section 5.3.1), our feeling is that the off-line model allows for an unfair accounting of space as well as unintendedly long proofs.

5.4 PSPACE and Games

As stated up-front, we will rarely treat computational problems that require less than logarithmic space. On the other hand, we will rarely treat computational problems that require more than polynomial space. The class of decision problems that are solvable in polynomial-space is denoted $\mathcal{PSPACE} \stackrel{\text{def}}{=} \cup_c \text{DSPACE}(p_c)$, where $p_c(n) = n^c$.

To get a sense of the power of $\mathcal{PSPACE}$, we observe that $\mathcal{PH} \subseteq \mathcal{PSPACE}$; for example, a polynomial-space algorithm can easily verify the quantified condition underlying Definition 3.8. In fact, such an algorithm can handle an unbounded number of alternating quantifiers (see Theorem 5.15). On the other hand, by Theorem 5.3, $\mathcal{PSPACE} \subseteq \mathcal{EXP}$, where $\mathcal{EXP} = \cup_c \text{DTIME}(2^{p_c})$ for $p_c(n) = n^c$.

The class $\mathcal{PSPACE}$ can be interpreted as capturing the complexity of determining the winner in certain *efficient two-party game*; specifically, the very games considered in Section 3.2.1 (modulo Footnote 2 there). Recall that we refer to two-party games that satisfy the following three conditions:

1. The parties alternate in taking moves that effect the game's (global) position, where each move has a description length that is bounded by a polynomial in the length of the *initial* position.

2. The current position is updated based on the previous position and the current party's move. This updating can be performed in time that is polynomial in the length of the *initial* position. (Equivalently, we may require a polynomial-time updating procedure and postulate that the length of the current position be bounded by a polynomial in the length of the *initial* position.)

3. The winner in each position can be determined in polynomial-time.

A set $S \in \mathcal{PSPACE}$ can be viewed as the set of initial positions (in a suitable game) for which the first party has a winning strategy *consisting of a polynomial number of moves*. Specifically, $x \in S$ if starting at the initial position x, there exists move y_1 for the first party, such that for every response move y_2 of the second party, there exists move y_3 for the first party, etc, such that after $\text{poly}(|x|)$ many moves the parties reach a position in which the first party wins, where the final position as well as which party wins in it can be computed in polynomial-time (from the initial position x and the sequence of moves $y_1, y_2, \ldots$). The fact that every set in $\mathcal{PSPACE}$ corresponds to such a game follows from Theorem 5.15, which refers to the satisfiability of quantified Boolean formulae (QBF).[17]

Theorem 5.15 QBF *is complete for* $\mathcal{PSPACE}$ *under polynomial-time many-to-one reductions.*

Proof: As note before, QBF is solvable by a polynomial-space algorithm that just evaluates the quantified formula. Specifically, consider a recursive procedure that eliminates a Boolean quantifier by evaluating the value of the two residual formulae, and note that the space used in the first (recursive) evaluation can be re-used in the second evaluation. (Alternatively, consider a DFS-type procedure as in Section 5.1.3.) Note that the space used is linear in the depth of the recursion, which in turn is linear in the length of the input formula.

We now turn to show that any set $S \in \mathcal{PSPACE}$ is many-to-one reducible to QBF. The proof is similar to the proof of Theorem 5.12, except that here we work with an implicit graph (rather than with an explicitly given graph). Specifically, we refer to the directed graph of configuration (of the algorithm A deciding membership in S) as defined in Exercise 5.16. Actually, here we use a different notion of a configuration that *includes also the input*. That is, in the rest of this proof, *a configuration consists of the contents of all storage devices of the algorithm* (including the input device) as well as the location of the algorithm on each device.

Recall that for a graph G, we defined $\phi_G(u,v,\ell) = 1$ if there is a path of length at most ℓ from u to v in G (and $\phi_G(u,v,\ell) = 0$ otherwise). We need to determine $\phi_G(s,t,2^m)$ for s that encodes the initial configuration of $A(x)$ and t that encodes the canonical accepting configuration, where G depends on the algorithm A and $m = \mathrm{poly}(|x|)$ is such that $A(x)$ uses at most m space and runs for at most 2^m steps. By the specific definition of a configuration (which contains all relevant information including the input x), the value of $\phi_G(u,v,1)$ can be determined easily based solely on the fixed algorithm A (i.e., either $u = v$ or v is a configuration following u). Recall that $\phi_G(u,v,2\ell) = 1$ if and only if there exists a configuration w such that both $\phi_G(u,w,\ell) = 1$ and $\phi_G(w,v,\ell) = 1$ hold. Thus, we obtain the recursion

$$\phi_G(u,v,2\ell) = \exists w \in \{0,1\}^m \phi_G(u,w,\ell) \wedge \phi_G(w,v,\ell), \tag{5.2}$$

where the bottom of the recursion (i.e., $\phi_G(u,v,1)$) is a simple propositional formula (see foregoing discussion). The problem with Eq. (5.2) is that the expression for $\phi_G(\cdot,\cdot,2\ell)$ involves two occurrences of $\phi_G(\cdot,\cdot,\ell)$, which doubles the length of the recursively constructed formula (yielding an exponential blow-up).

Our aim is to express $\phi_G(\cdot,\cdot,2\ell)$ *while using* $\phi_G(\cdot,\cdot,\ell)$ *only once*. The extra restriction, which prevents an exponential blow-up, corresponds to the *re-using of space* in the (two evaluations of $\phi_G(\cdot,\cdot,\ell)$ that take place in the) computation of $\phi_G(u,v,2\ell)$. The main idea is replacing the condition $\phi_G(u,w,\ell) = \phi_G(w,v,\ell) = 1$ by the condition $\forall (u'v') \in \{(u,w),(w,v)\}\ \phi_G(u',v',\ell)$. Next, we reformulate the

[17] See Appendix G.2.

"non-standard quantifier" (which ranges over a specific pair of strings) by using additional quantifiers as well as some simple boolean conditions. That is, $\forall(u'v') \in \{(u,w),(w,v)\}$ is replaced by $\forall\sigma \in \{0,1\}\exists u',v' \in \{0,1\}^m$ and the auxiliary condition

$$[(\sigma=0) \Rightarrow (u'=u \wedge v'=w)] \wedge [(\sigma=1) \Rightarrow (u'=w \wedge v'=v)]. \tag{5.3}$$

Thus, $\phi_G(u,v,2\ell)$ holds if and only if there exist w such that for every σ there exists (u',v') such that both Eq. (5.3) and $\phi_G(u',v',\ell)$ hold. Note that the length of this expression for $\phi_G(\cdot,\cdot,2\ell)$ equals the length of $\phi_G(\cdot,\cdot,\ell)$ plus an additive overhead term of $O(m)$. Thus, using a recursive construction, the length of the formula grows only linearly in the number of recursion steps.

The reduction itself maps an instance x (of S) to the quantified Boolean formula $\Phi(s_x,t,2^m)$, where s_x denotes the initial configuration of $A(x)$, (t and $m = \text{poly}(|x|)$ are as above), and Φ is recursively defined as follows

$$\Phi(u,v,2\ell) \stackrel{\text{def}}{=} \begin{array}{l} \exists w \in \{0,1\}^m \,\forall\sigma \in \{0,1\}\exists u',v' \in \{0,1\}^m \\ \qquad [(\sigma=0) \Rightarrow (u'=u \wedge v'=w)] \\ \qquad \wedge\, [(\sigma=1) \Rightarrow (u'=w \wedge v'=v)] \\ \qquad \wedge\, \Phi(u',v',\ell) \end{array} \tag{5.4}$$

with $\Phi(u,v,1) = 1$ if and only if either $u = v$ or there is an edge from u to v. Note that $\Phi(u,v,1)$ is a propositional formula with Boolean variables representing the bits of u and v such that $\Phi(u,v,1)$ is satisfies if and only if either $u = v$ or v is a configuration that follows the configuration u in a computation of A. On the other hand, note that $\Phi(s_x,t,2^m)$ is a quantified formula in which the quantified variables are not shown in the notation.

We stress that the mapping of x to $\Phi(s_x,t,2^m)$ can be computed in polynomial-time. Firstly, note that the propositional formula $\Phi(u,v,1)$, having Boolean variables representing the bits of u and v, expresses extremely simple conditions and can certainly be constructed in polynomial-time (i.e., polynomial in the number of Boolean variables, which in turn equals $2m$). Next note that, given $\Phi(u,v,\ell)$, which (for $\ell > 1$) contains quantified variables that are not shown in the notation, we can construct $\Phi(u,v,2\ell)$ by merely replacing variables names and adding quantifiers and Boolean conditions as in the recursive definition of Eq. (5.4). This is certainly doable in polynomial-time. Lastly, note that the construction of $\Phi(s_x,t,2^m)$ depends mainly on the length of x, where x itself only affects s_x (and does so in a trivial manner). Recalling that $m = \text{poly}(|x|)$, it follows that everything is computable in time polynomial in $|x|$. Thus, given x, the formula $\Phi(s_x,t,2^m)$ can be constructed in polynomial-time.

Finally, note that $x \in S$ if and only if the formula $\Phi(s_x,t,2^m)$ is satisfiable. The theorem follows. ■

Other $\mathcal{PSPACE}$-complete problems. Several generalizations of natural games give rise to $\mathcal{PSPACE}$-complete problems (see [194, Sec. 8.3]). This further justifies the title of the current section.

Chapter Notes

The material presented in the current chapter is based on a mix of "classical" results (proven in the 1970's if not earlier) and "modern" results (proven in the late 1980's and even later). What we wish to emphasize is the time gap between the formulation of some questions and their resolution. Details follow.

We first mention the "classical" results. These include the $\mathcal{NL}$-completeness of st-CONN, the emulation of non-deterministic space-bounded machines by deterministic space-bounded machines (i.e., Theorem 5.12 due to Savitch [184]), the $\mathcal{PSPACE}$-completeness of QBF, and the connections between circuit depth and space complexity (see Section 5.1.3 and Exercise 5.7 due to Borodin [45]).

Before turning to the "modern" results, we mention that some people tend to be discouraged by the impression that "decades of research have failed to answer any of the famous open problems of complexity theory." In addition to the fact that substantial progress towards the understanding of many fundamental issues has been achieved, people tend to forget that some famous open problems were actually resolved. Two such examples were presented in this chapter.

The question of whether $\mathcal{NL} = \text{co}\mathcal{NL}$ was a famous open problem for almost two decades. Furthermore, this question is related to an even older open problem dating to the early days of research in the area of formal languages (i.e., to the 1950's).[18] This open problem was resolved in 1988 by Immerman [118] and Szelepcsenyi [205], who (independently) proved Theorem 5.14 (i.e., $\mathcal{NL} = \text{co}\mathcal{NL}$).

For more than two decades, undirected connectivity was one of the most appealing examples of the computational power of randomness. Recall that the classical (deterministic) linear-time algorithms (e.g., BFS and DFS) require an extensive use of (extra) memory (i.e., space linear in the size of the graph). On the other hand, it was known (since 1979, see §6.1.4.2) that, with high probability, a random walk of polynomial length visits all vertices (in the corresponding connected component). Thus, the randomized algorithm requires a minimal amount of auxiliary memory (i.e., logarithmic in the size of the graph). In the early 1990's, this algorithm (as well as the entire class $\mathcal{BPL}$ (see Definition 6.9)) was derandomized in polynomial-time and poly-logarithmic space (see Theorem 8.23), but despite more than a decade of research attempts, a significant gap remained between the space complexity of randomized and deterministic polynomial-time algorithms for this natural and ubiquitous problem. This gap was closed by Reingold [177], who established Theorem 5.6 in 2004. Our presentation (in Section 5.2) follows his ideas, but the specific formulation in §5.2.4.2 is new.

Exercises

Exercise 5.1 (rewriting on the write-only output-tape) Let A be an arbitrary algorithm of space complexity s. Show that there exists a functionally

[18] Specifically, the class of sets recognized by linear-space non-deterministic machines equals the class of context-sensitive languages (see, e.g., [116, Sec. 9.3]), and thus Theorem 5.14 resolves the question of whether the latter class is closed under complementation.

equivalent algorithm A' that never rewrites on (the same location of) its output-device and has space complexity s' such that $s'(n) = s(n) + O(\log \ell(n))$, where $\ell(n) = \max_{x\in\{0,1\}^n} |A(x)|$.

Guideline: Algorithm A' proceeds in iterations, where in the i^{th} iteration it outputs the i^{th} bit of $A(x)$ by emulating the computation of A on input x. The i^{th} emulation of A avoids printing $A(x)$, but rather keeps a records of the i^{th} location of $A(x)$'s output-tape (and terminates by outputing the final value of this bit). Indeed, this emulation requires maintaining the current value of i as well as the current location of emulated machine (i.e., A) on its output-tape.

Exercise 5.2 (on the power of double-logarithmic space) For any $k \in \mathbb{N}$, let w_k denote the concatenation of all k-bit long strings (in lexicographic order) separated by $*$'s (i.e., $w_k = 0^{k-2}00 * 0^{k-2}01 * 0^{k-2}10 * 0^{k-2}11 * \cdots * 1^k$). Show that the set $S \stackrel{\text{def}}{=} \{w_k : k \in \mathbb{N}\} \subset \{0,1,*\}$ is not regular and yet is decidable in double-logarithmic space.

Guideline: The non-regularity of S can be shown using standard techniques. Towards developing an algorithm, note that $|w_k| > 2^k$, and thus $O(\log k) = O(\log\log |w_k|)$. Membership of x in S is determined by iteratively checking whether $x = w_i$, for $i = 1, 2, \ldots$. Note that the i^{th} iteration can be implemented in space $O(\log i)$, and that on input $x \notin S$ we halt and reject after at most $\log |x|$ iterations.

Exercise 5.3 (on the weakness of less than double-logarithmic space) Prove that for $\ell(n) = \log\log n$, it holds that $\text{DSPACE}(o(\ell)) = \text{DSPACE}(O(1))$.

Guideline: Let s denote the machine's (binary) space complexity (see Footnote 2). Assuming that s is unbounded, consider for each m the shortest string x such that on input x the machine uses space at least m. Consider, for each location on the input, the sequence of the residual configurations of the machine (i.e., the contents of its temporary storage)[19] at the times in which the machine crosses (or rather passes through) this input location. For starters, note that the length of this "crossing sequence" is upper-bounded by the number of possible configurations, which is at most $t \stackrel{\text{def}}{=} 2^{s(|x|)} \cdot s(|x|)$. Thus, the number of such crossing sequences is upper-bounded by $(t+1)! < t^t$. Now, if $t^t < |x|/2$ then there exist three input locations that have the same crossing sequence, and two of them hold the same bit value. Contracting the string at these two locations, we get a shorter input on which the machine behaves in exactly the same manner, contradicting the hypothesis that x is the shortest input on which the machine uses space at least m. We conclude that $t^t \geq |x|/2$ must hold, and $s(|x|) = \Omega(\log t) = \Omega(\log\log |x|)$ follows.

Exercise 5.4 (some log-space algorithms) Present log-space algorithms for the following computational problems.

1. Addition and multiplication of a pair of integers.

[19]Note that, unlike in the proof of Theorem 5.3, the machine's location on the input is not part of the notion of a configuration used here. On the other hand, although not stated explicitly, the configuration also encodes the machine's location on the storage tape.

Guideline: Relying on Lemma 5.2, first transform the input to a more convenient format, then perform the operation, and finally transform the result to the adequate format. For example, when adding $x = \sum_{i=0}^{n-1} x_i 2^i$ and $y = \sum_{i=0}^{n-1} y_i 2^i$ a convenient format is $((x_0, y_0), ..., (x_{n-1}, y_{n-1}))$.

2. Transforming the adjacency matrix representation of a graph to its incidence list representation, and vice versa.

3. Deciding whether the input graph is acyclic (i.e., has no simple cycles).

   Guideline: Consider a scanning of the graph that proceeds as follows. Upon entering a vertex v via the i^{th} edge incident at it, we exit this vertex using its $i+1^{\text{st}}$ if v has degree at least $i+1$ and exit via the first edge otherwise. Note that when started at any vertex of any tree, this scanning performs a DFS. On the other hand, for every cyclic graph there exists a vertex v and an edge e incident to v such that if this scanning is started by traversing the edge e from v then it returns to v via an edge different from e.

4. Deciding whether the input graph is a tree.

   Guideline: Use the fact that a graph $G = (V, E)$ is a tree if and only if it is acyclic and $|E| = |V| - 1$.

Exercise 5.5 (another composition result) In continuation to the discussion in §5.1.2.3, prove that *if* Π *can be computed in space* s_1 *when given an* (ℓ, ℓ')*-restricted oracle access to* Π' *and* Π' *is solvable is space* s_2, *then* Π *is solvable in space* s *such that* $s(n) = 2s_1(n) + s_2(\ell(n)) + 2\ell'(n) + \log_2 \ell(n)$.

Guideline: Combine the ideas underlying the proofs of Lemmas 5.1 and 5.2. Specifically, view the oracle-aided computation of Π as consisting of iterations such that in the i^{th} iteration the i^{th} query is determined based on the initial input, the $i-1^{\text{st}}$ oracle answer and the contents of the work tape at the time the $i-1^{\text{st}}$ answer was given. Composing each iteration with the computation of Π' using Lemma 5.2, we conclude that the i^{th} answer can be computed (without storing the i^{th} query) in space $s_1(n) + s_2(\ell(n)) + \log_2 \ell(n)$. Thus, we can emulate the entire computation using space $s(n)$, where the extra space $s_1(n) + 2\ell'(n)$ is used for storing the work-tape of the oracle machine and the $i-1^{\text{st}}$ and i^{th} oracle answers.

Exercise 5.6 Referring to the discussion in §5.1.2.3, prove that any problem having space complexity s can be solved by a constant-space $(2s, 2s)$-restricted reduction to a problem that is solvable in constant-space.

Guideline: The reduction is to the "next configuration function" associated with the said algorithm (of space complexity s). To facilitate the computation of this function, represent each configuration in a redundant manner (e.g., as a sequence over a 4-ary rather than a binary alphabet). The reduction consists of iteratively copying strings from the (input or) oracle-answer tape to the oracle-query (or output) tape.

Exercise 5.7 (log-space uniform $\mathcal{NC}^1$ is in $\mathcal{L}$) Suppose that a problem Π is solvable by a family of log-space uniform bounded fan-in circuits of depth d such that $d(n) \geq \log n$. Prove that Π is solvable by an algorithm having space complexity $O(d)$.

Guideline: Combine the algorithm outlined in Section 5.1.3 with the definition of log-space uniformity (using Lemma 5.2).

Exercise 5.8 (transitivity of log-space reductions) Prove that log-space Karp-reductions are transitive. Define log-space Levin-reductions and prove that they are transitive.

Guideline: Use Lemma 5.2, noting that such reductions are merely log-space computable functions.

Exercise 5.9 (UCONN in constant degree graphs of logarithmic diameter) Present a log-space algorithm for deciding the following promise problem, which is parameterized by constants c and d. The input graph satisfies the promise if each vertex has degree at most d and every pair of vertices that reside in the same connected component are connected by a path of length at most $c \log_2 n$, where n denotes the number of vertices in the input graph. The task is to decide whether the input graph is connected.

Guideline: For every pair of vertices in the graph, we check whether these vertices are connected in the graph. (Alternatively, we may just check whether each vertex is connected to the first vertex.) Relying on the promise, it suffices to inspect all paths of length at most $\ell \stackrel{\text{def}}{=} c \log_2 n$, and these paths can be enumerated using $\ell \cdot \lceil \log_2 d \rceil$ bits of storage.

Exercise 5.10 (warm-up towards §5.2.4.2) In continuation to §5.2.4.1, present a log-space transformation of G_i to G_{i+1}.

Guideline: Given the graph G_i as input, we may construct G_{i+1} by first constructing $G' = G_i^c$ and then constructing $G' \textcircled{z} G$. To construct G', we scan all vertices of G_i (holding the current vertex in temporary storage), and for each such vertex construct its neighborhood in G' (by using $O(c)$ space for enumerating all possible neighbors). Similarly, we can construct the vertex neighborhoods in $G' \textcircled{z} G$ (by storing the current vertex name and using a constant amount of space for indicating incident edges in G).

Exercise 5.11 (st-UCONN) In continuation to Section 5.2.4, prove that the following computational problem is in $\mathcal{L}$: Given an undirected graph $G = (V, E)$ and two designated vertices, s and t, determine whether there is a path from s to t in G.

Guideline: Note that the transformation described in Section 5.2.4 can be easily extended such that it map vertices in G_0 to vertices in $G_{O(\log |V|)}$ while preserving the connectivity relation (i.e., u and v are connected in G_0 if and only if their images under the map are connected in $G_{O(\log |V|)}$).

Exercise 5.12 Prove that the problem of determining whether or not the input graph is bipartite (2-colorable) is computationally equivalent under log-space reductions to `st-UCONN` (as defined in Exercise 5.11).

Guideline: Both reductions use the mapping of a graph $G=(V,E)$ to a bipartite graph $G'=(V',E')$ such that $V' = \{v^{(1)}, v^{(2)} : v \in V\}$ and $E' = \{(u^{(1)}, v^{(2)}), (u^{(2)}, v^{(1)}) : \{u,v\} \in E\}$. When reducing to `st-UCONN` note that a vertex v resides on an odd cycle in G if and only if $v^{(1)}$ and $v^{(2)}$ are connected in G'. When reducing from `st-UCONN` note that s and t are connected in G by a path of even (resp., odd) length if and only if the graph G' ceases to be bipartite when augmented with the edge $\{s^{(1)}, t^{(1)}\}$ (resp., $\{s^{(1)}, t^{(2)}\}$).

Exercise 5.13 (finding paths in undirected graphs) In continuation to Exercise 5.11, present a log-space algorithm that given an undirected graph $G = (V,E)$ and two designated vertices, s and t, finds a path from s to t in G (in case such a path exists).

Guideline: In continuation to Exercise 5.11, we may finds and store a logarithmic path in $G_{O(\log|V|)}$ that connects a representative of s and a representative of t. Focusing on the task of finding a path in G_0 that corresponds to an edge in $G_{O(\log|V|)}$, we note that such a path can be found by using the reduction underlying the combination of Claim 5.9 and Lemma 5.10. (An alternative description appears in [177].)

Exercise 5.14 (relating the two models of NSPACE) Referring to the definitions in Section 5.3.1, prove that for every function s such that $\log s$ is space contructible and at least logarithmic, it holds that $\text{NSPACE}_{\text{on-line}}(s) = \text{NSPACE}_{\text{off-line}}(\Theta(\log s))$.

Guideline (for $\text{NSPACE}_{\text{on-line}}(s) \subseteq \text{NSPACE}_{\text{off-line}}(O(\log s))$): Use the non-deterministic input of the off-line machine for encoding an accepting computation of the on-line machine; that is, this input should contain a sequence of consecutive configurations leading from the initial configuration to an accepting configuration, where each configuration contains the contents of the work-tape as well as the machine's location on the work-tape and on the input tape. The emulating off-line machine (which verifies the correctness of the sequence of configurations recorded on its non-deterministic input tape) needs only store its *location within the current pair of consecutive configurations* that it examines, which requires space logarithmic in the length of a single configuration (which in turn equals $s(n) + \log_2 s(n) + \log_2 n$). (Note that this verification relies on a two-directional access to the non-deterministic input.)

Guideline (for $\text{NSPACE}_{\text{off-line}}(s') \subseteq \text{NSPACE}_{\text{on-line}}(\exp(s'))$): Here we refer to the notion of a crossing-sequence. Specifically, for each location on the off-line non-deterministic input, consider the sequence of the residual configurations of the machine, where such a residual configuration consists of the bit residing in this non-deterministic tape location, the contents of the machine's temporary storage and the machine's locations on the input and storage tapes (but not its location on the non-deterministic tape). Show that the length of such a crossing-sequence is exponential in the space complexity of the off-line machine, and that the time complexity of the off-line machine is at most double-exponential in its space complexity (see Exercise 5.3). The on-line machine merely generates a sequence of crossing-sequences ("on the fly") and checks that each consecutive pair of crossing-sequences is consistent. This requires holding two crossing-sequences in storage, which require space linear in the length of such sequences (which, in turn, is exponential in the space complexity of the off-line machine).

Exercise 5.15 (st-CONN and variants of it are in NL) Prove that the following computational problem is in $\mathcal{NL}$. The instances have the form (G, v, w, ℓ), where $G=(V, E)$ is a directed graph, $v, w \in V$, and ℓ is an integer, and the question is whether G contains a path of length at most ℓ from v to w.

Guideline: Consider a non-deterministic (on-line) machine that generates and verifiers an adequate path on the fly. That is, starting at $v_0 = v$, the machine proceeds in iterations, such that in the i^{th} iteration it non-deterministically generates v_i, verifiers that $(v_{i-1}, v_i) \in E$, and checks whether $i \leq \ell$ and $v_i = w$. Note that this machine need only store the last two vertices on the path (i.e., v_{i-1} and v_i) as well as the number of edges traversed so far (i.e., i). Using a careful implementation, it suffices to store only one of these vertices (as well as i).

Exercise 5.16 (NSPACE and directed connectivity) Our aim is to establish a relation between general non-deterministic space-bounded computation and directed connectivity in "strongly constructible" graphs that have size exponential in the space bound. Let s be space constructible and at least logarithmic. For every $S \in \text{NSPACE}(s)$, present a linear-time oracle machine (somewhat as in §5.2.4.2) that given oracle access to x provides oracle access to a directed graph G_x of size $\exp(s(|x|))$ such that $x \in S$ if and only if there is a directed path between the first and last vertices of G_x. That is, on input a pair (u, v) and oracle access to x, the machine decides whether or not (u, v) is a directed edge in G_x.

Guideline: Follow the proof of Theorem 5.11.

Exercise 5.17 (an alternative presentation of the proof of Theorem 5.12) We refer to directed graphs in which each vertex has a self-loop.

1. Viewing the adjacency matrices of directed graphs as oracles (cf. Exercise 5.16), present a linear space oracle machine that determines whether a given pair of vertices is connected by a directed path of length two in the input graph. Note that this machine computes the adjacency relation of the square of the graph represented in the oracle.
2. Using naive composition (as in Lemma 5.1), present a quadratic space oracle machine that determines whether a given pair of vertices is connected by a directed path in the input graph.

Note that the machine in Item 2 implies that `st-CONN` can be decided in log-square space. In particular, justify the self-loop assumption made up-front.

Exercise 5.18 (finding shortest paths in undirected graphs) Prove that the following computational problem is $\mathcal{NL}$-complete under (many-to-one) log-space reductions: Given an undirected graph $G = (V, E)$, two designated vertices, s and t, and an integer K, determine whether there is a path of length at most K from s to t in G.

Guideline (for $\mathcal{NL}$-hardness): Reduce from `st-CONN`. Specifically, given a directed graph $G = (V, E)$ and vertices s, t, consider a ("layered") graph $G' = (V', E')$ such that

$V' = \cup_{i=0}^{|V|-1}\{\langle i,v\rangle : v\in V\}$ and $E' = \cup_{i=0}^{|V|-2}\{\{\langle i,u\rangle, \langle i+1,v\rangle\} : (u,v)\in E \vee u=v\}$. Note that there exists a directed path from s to t in G if and only if there exists a path of length $|V|-1$ between $\langle s,0\rangle$ and $\langle t,|V|-1\rangle$ in G'.

Exercise 5.19 (deciding strong connectivity) A directed graph is called strongly connected if there exists a directed path between every ordered pair of vertices in the graph (or, equivalently, a directed cycle passing through every two vertices). Prove that the problem of deciding whether a directed graph is strongly connected is $\mathcal{NL}$-complete under (many-to-one) log-space reductions.

Guideline (for $\mathcal{NL}$-hardness): Reduce from `st-CONN`, noting that (G,s,t) is a yes-instance of `st-CONN`, where $G=(V,E)$, if and only if the graph $G' = (V, E\cup\{(v,s) : v\in V\}\cup\{(t,v) : v\in V\})$ is strongly connected.

Exercise 5.20 (an operational interpretation of $\mathcal{NL}\cap \text{co}\mathcal{NL}$, $\mathcal{NP}\cap\text{co}\mathcal{NP}$, etc) Referring to Definition 5.13, prove that $S\in\mathcal{NL}\cap\text{co}\mathcal{NL}$ if and only if there exists a non-deterministic log-space machine that computes χ_S, where $\chi_S(x)=1$ if $x\in S$ and $\chi_S(x)=0$ otherwise. State and prove an analogous result for $\mathcal{NP}\cap\text{co}\mathcal{NP}$.

Guideline: A non-deterministic machine computing any function f yields, for each value v, a machine of similar complexity that accept $\{x : f(x)=v\}$. (Extra hint: Invoke the machine M that computes f and accept if and only if M outputs v.) On the other hand, for any function f of finite range, combining machines that accept the various $S_v \stackrel{\text{def}}{=} \{x : f(x)=v\}$, we obtain a machine of similar complexity that computes f. (Extra hint: On input x, the combined machine invokes each of the aforementioned machines on input x and outputs the value v if and only if the machine accepting S_v has accepted. In the case that none of the machines accepts, the combined machine outputs $\perp$.)

Exercise 5.21 (a graph algorithmic interpretation of $\mathcal{NL}=\text{co}\mathcal{NL}$) Show that there exists a log-space computable function f such that for every (G,s,t) it holds that (G,s,t) is a yes-instance of `st-CONN` if and only if $(G',s',t')=f(G,s,t)$ is a no-instance of `st-CONN`.

Exercise 5.22 As an alternative to the two-query reduction presented in the proof of Theorem 5.14, show that computing the characteristic function of `st-CONN` is log-space reducible via a single query to the problem of determining the number of vertices that are reachable from a given vertex in a given graph.
(Hint: On input (G,s,t), where $G=([N],E)$, consider the number of vertices reachable from s in the graph $G'=([2N], E\cup\{(t,N+i) : i=1,...,N\})$.)

Exercise 5.23 (reductions and non-deterministic computations) Suppose that computing f is log-space reducible by a constant number of queries to computing some function g. Referring to the non-deterministic computations as in Definition 5.13, prove that if there exists a non-deterministic log-space machine that computes g then there exists a non-deterministic log-space machine that computes f.

Guideline: Use the emulative composition (as in Lemma 5.2). If any of the non-deterministic computations of g returns the value $\perp$ then return $\perp$ as the value of f.

Otherwise, use the non-$\perp$ values provided by the non-deterministic computations of g to compute the value of f.

Exercise 5.24 (reductions and non-deterministic computations, revisited) Suppose that computing f is log-space reducible (by any number of queries) to computing some function g such that for every x it holds that $|g(x)| = O(\log|x|)$. Referring to the non-deterministic computations as in Definition 5.13, prove that if there exists a non-deterministic log-space machine that computes g then there exists a non-deterministic log-space machine that computes f. As a warm-up consider the special case in which every query to g is computable in log-space based on the input to f.

Guideline: As in Exercise 5.23, except that here we use different composition techniques. Specifically, in the warm-up we use the naive composition (as in Lemma 5.1), whereas in the general case we use the semi-naive composition of Exercise 5.5.

Exercise 5.25 Referring to Definition 5.13, prove that there exists a non-deterministic log-space machine that computes the distance between two given vertices in a given undirected graph.

Guideline: Relate this computational problem to the decision problem considered in Exercise 5.18, and use $\mathcal{NL} = \text{co}\mathcal{NL}$.

Chapter 6

Randomness and Counting

I owe this almost atrocious variety to an institution which other republics do not know or which operates in them in an imperfect and secret manner: the lottery.

Jorge Luis Borges, The Lottery In Babylon

So far, our approach to computing devices was somewhat conservative: we thought of them as executing a deterministic rule. A more liberal and quite realistic approach, which is pursued in this chapter, considers computing devices that use a probabilistic rule. This relaxation has an immediate impact on the notion of efficient computation, which is consequently associated with *probabilistic* polynomial-time computations rather than with deterministic (polynomial-time) ones. We stress that the association of efficient computation with probabilistic polynomial-time computation makes sense provided that the failure probability of the latter is negligible (which means that it may be safely ignored).

The quantitative nature of the failure probability of probabilistic algorithm provides one connection between probabilistic algorithms and counting problems. The latter are indeed a new type of computational problems, and our focus is on counting efficiently recognizable objects (e.g., NP-witnesses for a given instance of set in $\mathcal{NP}$). Randomized procedures turn out to play an important role in the study of such counting problems.

Summary: Focusing on probabilistic polynomial-time algorithms, we consider various types of failure of such algorithms yielding complexity classes such as $\mathcal{BPP}$, $\mathcal{RP}$, and $\mathcal{ZPP}$. The results presented include $\mathcal{BPP} \subset \mathcal{P}/\text{poly}$ and $\mathcal{BPP} \subseteq \Sigma_2$.

We then turn to counting problems (i.e., counting the number of solutions for an instance of a search problem in $\mathcal{PC}$, or, equivalently, counting the number of NP-witnesses for an instance of a decision problem in $\mathcal{NP}$). We distinguish between exact counting and approximate counting (in the sense of relative approximation). In particular, while any

problem in $\mathcal{PH}$ is reducible to the exact counting class $\#\mathcal{P}$, approximate counting (for $\#\mathcal{P}$) is reducible in probabilistic polynomial-time to $\mathcal{NP}$.

Additional related topics include the $\#\mathcal{P}$-completeness of various counting problems (e.g., counting the number of satisfying assignments to a given CNF formula and counting the number of perfect matchings in a given graph), the complexity of searching for unique solutions, and the relation between approximate counting and generating (almost) uniformly distributed solutions.

Prerequisites: We assume basic familiarity with elementary probability theory (see Section D.1). In Section 6.2 we will rely extensively on the formulation presented in Section 2.1 (i.e., the "NP search problem" class $\mathcal{PC}$ as well as the sets $R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\}$, and $S_R \stackrel{\text{def}}{=} \{X : R(x) = \emptyset\}$ defined for every $R \in \mathcal{PC}$).

6.1 Probabilistic Polynomial-Time

Considering algorithms that utilize random choices, we extend our notion of *efficient algorithms* from *deterministic* polynomial-time algorithms to *probabilistic* polynomial-time algorithms.

Rigorous models of probabilistic (or randomized) algorithms are defined by natural extensions of the basic machine model. We will exemplify this approach by describing the model of probabilistic Turing machines, but we stress that (again) the specific choice of the model is immaterial (as long as it is "reasonable"). A probabilistic Turing machine is defined exactly as a non-deterministic machine (see the first item of Definition 2.7), *but the definition of its computation is fundamentally different*. Specifically, whereas Definition 2.7 refers to the question of whether or not there exists a computation of the machine that (started on a specific input) reaches a certain configuration, in case of probabilistic Turing machines we refer to *the probability that this event occurs, when at each step a choice is selected uniformly among the relevant possible choices available at this step*. That is, if the transition function of the machine maps the current state-symbol pair to several possible triples, then the probabilistic machine randomly selects one of these triples (with equal probability) and the next configuration is determined accordingly. These random choices may be viewed as the internal coin tosses of the machine. (Indeed, as in the case of non-deterministic machines, we may assume without loss of generality that the transition function of the machine maps each state-symbol pair to exactly two possible triples; see Exercise 2.4.)

We stress the fundamental difference between the fictitious model of a non-deterministic machine and the realistic model of a probabilistic machine. In the case of a non-deterministic machine we consider the *existence* of an adequate sequence of choices (leading to a desired outcome), and ignore the question of how these choices are actually made. In fact, the selection of such a sequence of choices is merely a mental experiment. In contrast, in the case of a probabilistic machine, at each step a choice is made uniformly (among a set of predetermined possibilities), and we consider the *probability* of reaching a desired outcome.

In view of the foregoing, we consider the output distribution of such a probabilistic machine on fixed inputs; that is, for a probabilistic machine M and string $x \in \{0,1\}^*$, we denote by $M(x)$ the output distribution of M when invoked on input x, where the probability is taken uniformly over the machine's internal coin tosses. Needless to say, we will consider the probability that $M(x)$ is a "correct" answer; that is, in the case of a search problem (resp., decision problem) we will be interested in the probability that $M(x)$ is a valid solution for the instance x (resp., represents the correct decision regarding x).

The foregoing description views the internal coin tosses of the machine as taking place on-the-fly; that is, these coin tosses are performed *on-line* by the machine itself. An alternative model is one in which the sequence of coin tosses is provided by an external device, on a special "random input" tape. In such a case, we view these coin tosses as performed *off-line*. Specifically, we denote by $M'(x,r)$ the (uniquely defined) output of the residual deterministic machine M', when given the (primary) input x and random input r. Indeed, M' is a deterministic machine that takes two inputs (the first representing the actual input and the second representing the "random input"), but we consider the random variable $M(x) \stackrel{\text{def}}{=} M'(x, U_{\ell(|x|)})$, where $\ell(|x|)$ denotes the number of coin tosses "expected" by $M'(x,\cdot)$.

These two perspectives on probabilistic algorithms are clearly related: The computation of an on-line machine M is captured by the residual machine M', which may be viewed as receiving a sequence of coin tosses (obtained off-line), where the number of these coin tosses is determined by the time-complexity of M. (Indeed, there is no harm in supplying more coin tosses than are actually used by M.) For sake of clarity and future reference, we state the following definition.

Definition 6.1 (on-line and off-line formulations of probabilistic polynomial-time):

- *We say that M is a* on-line probabilistic polynomial-time machine *if there exists a polynomial p such that when invoked on any input $x \in \{0,1\}^*$, machine M always halts within at most $p(|x|)$ steps* (regardless of the outcome of its internal coin tosses). *In such a case $M(x)$ is a random variable.*

- *We say that M' is a* off-line probabilistic polynomial-time machine *if there exists a polynomial p such that, for every $x \in \{0,1\}^*$ and $r \in \{0,1\}^{p(|x|)}$, when invoked on the* primary input *x and the* random-input *sequence r, machine M' halts within at most $p(|x|)$ steps. In such a case, we will consider the random variable $M'(x, U_{p(|x|)})$.*

Clearly, the on-line and off-line formulations are equivalent (i.e., given a on-line probabilistic polynomial-time machine we can derive a functionally equivalent off-line (probabilistic polynomial-time) machine, and vice versa). Thus, in the sequel, we will freely use whichever is more convenient.

Failure probability. A major aspect of randomized algorithms (probabilistic machines) is that they may fail (see Exercise 6.1). That is, with some specified ("failure") probability, these algorithms may fail to produce the desired output. We discuss two aspects of this failure: its *type* and its *magnitude*.

1. The type of failure is a qualitative notion. One aspect of this type is whether, in case of failure, the algorithm produces a wrong answer or merely an indication that it failed to find a correct answer. Another aspect is whether failure may occur on all instances or merely on certain types of instances. Let us clarify these aspects by considering three natural types of failure, giving rise to three different types of algorithms.

   (a) The most liberal notion of failure is the one of two-sided error. This term originates from the setting of decision problems, where it means that (in case of failure) the algorithm may err in both directions (i.e., it may rule that a yes-instance is a no-instance, and vice versa). In the case of search problems two-sided error means that, when failing, the algorithm may output a wrong answer on any input. Furthermore, the algorithm may falsely rule that the input has no solution and it may also output a wrong solution (both in case the input has a solution and in case it has no solution).

   (b) An intermediate notion of failure is the one of one-sided error. Again, the term originates from the setting of decision problems, where it means that the algorithm may err only in one direction (i.e., either on yes-instances or on no-instances). Indeed, there are two natural cases depending on whether the algorithm errs on yes-instances (but not on no-instances), or the other way around. Analogous cases occur also in the setting of search problems. In one case the algorithm never outputs a wrong solution but may falsely rule that the input has no solution. In the other case the indication that an input has no solution is never wrong, but the algorithm may output a wrong solution.

   (c) The most conservative notion of failure is the one of zero-sided error. In this case, the algorithm's failure amounts to indicating its failure to find an answer (by outputting a special `don't know` symbol). We stress that in this case the algorithm *never provides a wrong answer*.

   Indeed, the forgoing discussion ignores the probability of failure, which is the subject of the next item.

2. The magnitude of failure is a quantitative notion. It refer to the probability that the algorithm fails, where the type of failure is fixed (e.g., as in the forgoing discussion).

   When actually using a randomized algorithm we typically wish its failure probability to be negligible, which intuitively means that the failure event is so rare that it can be ignored in practice. Formally, we say that a quantity is negligible if, as a function of the relevant parameter (e.g., the input length), this quantity vanishes faster than the reciprocal of any positive polynomial.

> For ease of presentation, we sometimes consider alternative upper-bounds on the probability of failure. These bounds are selected in a way that allows (and in fact facilitates) "error reduction" (i.e., converting a probabilistic polynomial-time algorithm that satisfies such an upper-bound into one in which the failure probability is negligible). For example, in case of two-sided error we need to be able to distinguish the correct answer from wrong answers by sampling, and in the other types of failure "hitting" a correct answer suffices.

In the following three subsections, we will discuss complexity classes corresponding to the aforementioned three types of failure. For sake of simplicity, the failure probability itself will be set to a constant that allows error reduction.

Randomized reductions. Before turning to the more detailed discussion, we note that randomized reductions play an important role in complexity theory. Such reductions can be defined analogously to the standard Cook-Reductions (resp., Karp-reduction), and again a discussion of the type and magnitude of the failure probability is in place. For clarity, we spell-out the two-sided error versions.

- In analogy to Definition 2.9, we say that a problem Π is probabilistic polynomial-time reducible to a problem Π' if there exists a probabilistic polynomial-time oracle machine M such that, for every function f that solves Π' and for every x, with probability at least $1-\mu(|x|)$, the output $M^f(x)$ is a correct solution to the instance x, where μ is a negligible function.
- In analogy to Definition 2.10, we say that a decision problem S is reducible to a decision problem S' via a randomized Karp-reduction if there exists a probabilistic polynomial-time algorithm A such that, for every x, it holds that $\Pr[\chi_{S'}(A(x)) = \chi_S(x)] \geq 1-\mu(|x|)$, where χ_S (resp., $\chi_{S'}$) is the characteristic function of S (resp., S') and μ is a negligible function.

These reductions preserve efficient solvability and are transitive: see Exercise 6.2.

6.1.1 Two-sided error: The complexity class BPP

In this section we consider the most liberal notion of probabilistic polynomial-time algorithms that is still meaningful. We allow the algorithm to err on each input, but require the error probability to be *negligible*. The latter requirement guarantees the usefulness of such algorithms, because in reality we may ignore the negligible error probability.

Before focusing on the decision problem setting, let us say a few words on the search problem setting (see Definition 1.1). Following the previous paragraph, we say that a probabilistic (polynomial-time) algorithm A solves the search problem of the relation R if for every $x \in S_R$ (i.e., $R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\} \neq \emptyset$) it holds that $\Pr[A(x) \in R(x)] > 1-\mu(|x|)$ and for every $x \notin S_R$ it holds that $\Pr[A(x) = \perp] > 1-\mu(|x|)$, where μ is a negligible function. Note that we did not require that, when invoked on input x that has a solution (i.e., $R(x) \neq \emptyset$), the algorithm always outputs the same solution. That is, a stronger requirement is that for every such x there exists $y \in R(x)$ such that $\Pr[A(x)=y] > 1 - \mu(|x|)$. The latter version as well as quantitative relaxations of it allow for error-reduction (see Exercise 6.3), discussed next.

Turning to decision problems, we consider probabilistic polynomial-time algorithms that err with negligible probability. That is, we say that a probabilistic (polynomial-time) algorithm A decides membership in S if for every x it holds that $\Pr[A(x) = \chi_S(x)] > 1 - \mu(|x|)$, where χ_S is the characteristic function of S (i.e., $\chi_S(x) = 1$ if $x \in S$ and $\chi_S(x) = 0$ otherwise) and μ is a negligible function. The class of decision problems that are solvable by probabilistic polynomial-time algorithms is denoted $\mathcal{BPP}$, standing for Bounded-error Probabilistic Polynomial-time. Actually, the standard definition refers to machines that err with probability at most $1/3$.

Definition 6.2 (the class $\mathcal{BPP}$): *A decision problem S is in $\mathcal{BPP}$ if there exists a probabilistic polynomial-time algorithm A such that for every $x \in S$ it holds that* $\Pr[A(x) = 1] \geq 2/3$ *and for every $x \notin S$ it holds that* $\Pr[A(x) = 0] \geq 2/3$.

The choice of the constant $2/3$ is immaterial, and any other constant greater than $1/2$ will do (and yields the very same class). Similarly, the complementary constant $1/3$ can be replaced by various negligible functions (while preserving the class). Both facts are special cases of the robustness of the class, which is established using the process of error reduction.

Error reduction (or confidence amplification). For $\varepsilon : \mathbb{N} \to (0, 0.5)$, let $\mathcal{BPP}_\varepsilon$ denote the class of decision problems that can be solved in probabilistic polynomial-time with error probability upper-bounded by ε; that is, $S \in \mathcal{BPP}_\varepsilon$ if there exists a probabilistic polynomial-time algorithm A such that for every x it holds that $\Pr[A(x) \neq \chi_S(x)] \leq \varepsilon(|x|)$. By definition, $\mathcal{BPP} = \mathcal{BPP}_{1/3}$. However, a wide range of other classes also equal $\mathcal{BPP}$. In particular, we mention two extreme cases:

1. For every positive polynomial p and $\varepsilon(n) = (1/2) - (1/p(n))$, the class $\mathcal{BPP}_\varepsilon$ equals $\mathcal{BPP}$. That is, any error that is ("noticeably") bounded away from $1/2$ (i.e., error $(1/2) - (1/\text{poly}(n))$) can be reduced to an error of $1/3$.

2. For every positive polynomial p and $\varepsilon(n) = 2^{-p(n)}$, the class $\mathcal{BPP}_\varepsilon$ equals $\mathcal{BPP}$. That is, an error of $1/3$ can be further reduced to an exponentially vanishing error.

Both facts are proved by invoking the weaker algorithm (i.e., the one having a larger error probability bound) for an adequate number of times, and ruling by majority. We stress that invoking a randomized machine several times means that the random choices made in the various invocations are independent of one another. The success probability of such a process is analyzed by applying an adequate Law of Large Numbers (see Exercise 6.4).

6.1.1.1 On the power of randomization

A natural question arises: *Did we gain anything in extending the definition of efficient computation to include also probabilistic polynomial-time ones?*

This phrasing seems too generic. We certainly gained the ability to toss coins (and generate various distributions). More concretely, randomized algorithms are essential in many settings (see, e.g., Chapter 9, Section 10.1.2, and Appendix C) and seem essential in others (see, e.g., Sections 6.2.2-6.2.4). What we mean to ask here is *whether allowing randomization increases the power of polynomial-time algorithms also in the restricted context of solving decision and search problems?*

The question is whether $\mathcal{BPP}$ extends beyond $\mathcal{P}$ (where clearly $\mathcal{P} \subseteq \mathcal{BPP}$). It is commonly conjectured that the answer is negative. Specifically, under some reasonable assumptions, it holds that $\mathcal{BPP} = \mathcal{P}$ (see Part 1 of Theorem 8.19). We note, however, that a polynomial slow-down occurs in the proof of the latter result; that is, randomized algorithms that run in time $t(\cdot)$ are emulated by deterministic algorithms that run in time $\text{poly}(t(\cdot))$. Furthermore, for some concrete problems (most notably primality testing (cf. §6.1.1.2)), the known probabilistic polynomial-time algorithm is significantly faster (and conceptually simpler) than the known deterministic polynomial-time algorithm. Thus, we believe that even in the context of decision problems, the notion of probabilistic polynomial-time algorithms is advantageous. We note that the fundamental nature of $\mathcal{BPP}$ will hold even in the (rather unlikely) case that it turns out that it offers no computational advantage (i.e., even if every problem that can be decided in probabilistic polynomial-time can be decided by a deterministic algorithm of essentially the same complexity).[1]

BPP is in the Polynomial-Time Hierarchy: While it may be that $\mathcal{BPP} = \mathcal{P}$, it is not known whether or not $\mathcal{BPP}$ is contained in $\mathcal{NP}$. The source of trouble is the two-sided error probability of $\mathcal{BPP}$, which is incompatible with the absolute rejection of no-instances required in the definition of $\mathcal{NP}$ (see Exercise 6.11). In view of this ignorance, it is interesting to note that $\mathcal{BPP}$ resides in the second level of the Polynomial-Time Hierarchy (i.e., $\mathcal{BPP} \subseteq \Sigma_2$). This is a corollary of Theorem 6.7.

Trivial derandomization. A straightforward way of eliminating randomness from an algorithm is trying all possible outcomes of its internal coin tosses, collecting the relevant statistics and deciding accordingly. This yields $\mathcal{BPP} \subseteq \mathcal{PSPACE} \subseteq \mathcal{EXP}$, which is considered the trivial derandomization of $\mathcal{BPP}$. In Section 8.4 we will consider various non-trivial derandomizations of $\mathcal{BPP}$, which are known under various intractability assumptions. The interested reader, who may be puzzled by the connection between derandomization and computational difficulty, is referred to Chapter 8.

[1] Such a result would address a fundamental question regarding the power of randomness. By analogy, Theorem 9.4 establishing that $\mathcal{IP} = \mathcal{PSPACE}$ does not diminish the importance of any of these classes.

Non-uniform derandomization. In many settings (and specifically in the context of solving search and decision problems), the power of randomization is superseded by the power of non-uniform advice. Intuitively, the non-uniform advice may specify a sequence of coin tosses that is good for all (primary) inputs of a specific length. In the context of solving search and decision problems, such an advice must be good for *each* of these inputs[2], and thus its existence is guaranteed only if the error probability is low enough (so as to support a union bound). The latter condition can be guaranteed by error-reduction, and thus we get the following result.

Theorem 6.3 *$\mathcal{BPP}$ is* (strictly) *contained in $\mathcal{P}$/poly.*

Proof: Recall that $\mathcal{P}$/poly contains undecidable problems (Theorem 3.7), which are certainly not in $\mathcal{BPP}$. Thus, we focus on showing that $\mathcal{BPP} \subseteq \mathcal{P}$/poly. By the discussion regarding error-reduction, for every $S \in \mathcal{BPP}$ there exists a (deterministic) polynomial-time algorithm A and a polynomial p such that for every x it holds that $\Pr[A(x, U_{p(|x|)}) \neq \chi_S(x)] < 2^{-|x|}$. Using a union bound, it follows that $\Pr_{r\in\{0,1\}^{p(n)}}[\exists x \in \{0,1\}^n \text{ s.t. } A(x,r) \neq \chi_S(x)] < 1$. Thus, for every $n \in \mathbb{N}$, there exists a string $r_n \in \{0,1\}^{p(n)}$ such that for every $x \in \{0,1\}^n$ it holds that $A(x, r_n) = \chi_S(x)$. Using such a sequence of r_n's as advice, we obtain the desired non-uniform machine (establishing $S \in \mathcal{P}$/poly). ■

Digest. The proof of Theorem 6.3 combines error-reduction with a simple application of the Probabilistic Method (cf. [10]), where the latter refers to proving the existence of an object by analyzing the probability that a random object is adequate. In this case, we sought an non-uniform advice, and proved it existence by analyzing the probability that a random advice is good. The latter event was analyzed by identifying the space of advice with the set of possible sequences of internal coin tosses of a randomized algorithm.

6.1.1.2 A probabilistic polynomial-time primality test

Teaching note: Although primality has been recently shown to be in $\mathcal{P}$, we believe that the following example provides a nice illustration to the power of randomized algorithms.

We present a simple probabilistic polynomial-time algorithm for deciding whether or not a given number is a prime. The only Number Theoretic facts that we use are:

Fact 1: For every prime $p > 2$, each quadratic residue mod p has exactly two square roots mod p (and they sum-up to p).[3]

[2] In other contexts (see, e.g., Chapters 7 and 8), it suffices to have an advice that is good on the average, where the average is taken over all relevant (primary) inputs.

[3] That is, for every $r \in \{1, ..., p-1\}$, the equation $x^2 \equiv s^2 \pmod{p}$ has two solutions modulo p (i.e., s and $p - s$).

Fact 2: For every (odd and non-integer-power) composite number N, each quadratic residue mod N has at least four square roots mod N.

Our algorithm uses as a black-box an algorithm, denoted `sqrt`, that given a prime p and a quadratic residue mod p, denoted s, returns the smallest among the two modular square roots of s. There is no guarantee as to what the output is in the case that the input is not of the aforementioned form (and in particular in the case that p is not a prime). Thus, we actually present a probabilistic polynomial-time reduction of testing primality to extracting square roots modulo a prime (which is a search problem with a promise; see Section 2.4.1).

Construction 6.4 (the reduction): *On input a natural number* $N > 2$ *do*

1. *If* N *is either even or an integer-power*[4] *then* `reject`.
2. *Uniformly select* $r \in \{1, ..., N-1\}$, *and set* $s \leftarrow r^2 \bmod N$.
3. *Let* $r' \leftarrow \texttt{sqrt}(s, N)$. *If* $r' \equiv \pm r \pmod N$ *then* `accept` *else* `reject`.

Indeed, in the case that N is composite, the reduction invokes `sqrt` on an illegitimate input (i.e., it makes a query that violates the promise of the problem at the target of the reduction). In such a case, there is not guarantee as to what `sqrt` answers, but actually a bluntly wrong answer only plays in our favor. In general, we will show that if N is composite, then the reduction rejects with probability at least $1/2$, regardless of how `sqrt` answers. We mention that there exists a probabilistic polynomial-time algorithm for implementing `sqrt` (see Exercise 6.14).

Proposition 6.5 *Construction 6.4 constitutes a probabilistic polynomial-time reduction of testing primality to extracting square roots module a prime. Furthermore, if the input is a prime then the reduction always accepts, and otherwise it rejects with probability at least* $1/2$.

We stress that Proposition 6.5 refers to the reduction itself; that is, `sqrt` is viewed as a ("perfect") oracle that, for every prime P and quadratic residue $s \pmod P$, returns $r < s/2$ such that $r^2 \equiv s \pmod P$. Combining Proposition 6.5 with a probabilistic polynomial-time algorithm that computes `sqrt` with negligible error probability, we obtain that testing primality is in $\mathcal{BPP}$.

Proof: By Fact 1, on input a prime number N, Construction 6.4 always accepts (because in this case, for every $r \in \{1, ..., N-1\}$, it holds that $\texttt{sqrt}(r^2 \bmod N, N) = \pm r$). On the other hand, suppose that N is an odd composite that is not an integer-power. Then, by Fact 2, each quadratic residue s has at least four square roots, and each of these square roots is equally likely to be chosen at Step 2 (in other words, s yields no information regarding which of its modular square roots was selected in Step 2). Thus, for every such s, the probability that either $\texttt{sqrt}(s, N)$

[4]This can be checked by scanning all possible powers $e \in \{2, ..., \log_2 N\}$, and (approximately) solving the equation $x^e = N$ for each value of e (i.e., finding the smallest integer i such that $i^e \geq N$). Such a solution can be found by binary search.

or $N - \texttt{sqrt}(s, N)$ equal the root chosen in Step 2 is at most $2/4$. It follows that, on input a composite number, the reduction rejects with probability at least $1/2$. ■

Reflection. Construction 6.4 illustrates an interesting aspect of randomized algorithms (or rather reductions); that is, the ability to hide information from a subroutine. Specifically, Construction 6.4 generates a problem instance (N, s) without disclosing any additional information. Furthermore, a correct solution to this instance is likely to help the reduction; that is, a correct answer to the instance (N, s) provides probabilistic evidence regarding whether N is a prime, where the probability space refers to the missing information (regarding how s was generated).

Comment. Testing primality is actually in $\mathcal{P}$, however, the deterministic algorithm demonstrating this fact is more complex (and its analysis is even more complicated).

6.1.2 One-sided error: The complexity classes RP and coRP

In this section we consider notions of probabilistic polynomial-time algorithms having one-sided error. The notion of one-sided error refers to a natural partition of the set of instances; that is, yes-instances versus no-instances in the case of decision problems, and instances having solution versus instances having no solution in the case of search problems. We focus on decision problems, and comment that an analogous treatment can be provided for search problems (see the second paragraph of Section 6.1.1).

Definition 6.6 (the class $\mathcal{RP}$)[5]: *A decision problem S is in $\mathcal{RP}$ if there exists a probabilistic polynomial-time algorithm A such that for every $x \in S$ it holds that* $\Pr[A(x)=1] \geq 1/2$ *and for every $x \notin S$ it holds that* $\Pr[A(x)=0] = 1$.

The choice of the constant $1/2$ is immaterial, and any other constant greater than zero will do (and yields the very same class). Similarly, this constant can be replaced by $1-\mu(|x|)$ for various negligible functions μ (while preserving the class). Both facts are special cases of the robustness of the class (see Exercise 6.5).

Observe that $\mathcal{RP} \subseteq \mathcal{NP}$ (see Exercise 6.11) and that $\mathcal{RP} \subseteq \mathcal{BPP}$ (by the aforementioned error-reduction). Defining $\text{co}\mathcal{RP} = \{\{0,1\}^* \setminus S : S \in \mathcal{RP}\}$, note that co$\mathcal{RP}$ corresponds to the opposite direction of one-sided error probability. That is, *a decision problem S is in* co$\mathcal{RP}$ *if there exists a probabilistic polynomial-time algorithm A such that for every $x \in S$ it holds that* $\Pr[A(x)=1] = 1$ *and for every $x \notin S$ it holds that* $\Pr[A(x)=0] \geq 1/2$.

[5]The initials RP stands for Random Polynomial-time, which fails to convey the restricted type of error allowed in this class. The only nice feature of this notation is that it is reminiscent of NP, thus reflecting the fact that $\mathcal{RP}$ is a randomized polynomial-time class contained in $\mathcal{RP}$.

Relating BPP to RP

A natural question regarding probabilistic polynomial-time algorithms refers to the relation between two-sided and one-sided error probability. For example, *is $\mathcal{BPP}$ contained in $\mathcal{RP}$*? Loosely speaking, we show that $\mathcal{BPP}$ is reducible to co$\mathcal{RP}$ by *one-sided error* randomized Karp-reductions. Note that $\mathcal{BPP}$ is trivially reducible to co$\mathcal{RP}$ by *two-sided error* randomized Karp-reductions whereas a deterministic reduction of $\mathcal{BPP}$ to co$\mathcal{RP}$ would imply $\mathcal{BPP} = \text{co}\mathcal{RP} = \mathcal{RP}$ (see Exercise 6.8). The actual statement refers to the promise problem versions of both classes, which are briefly defined next.

First, we refer the reader to the general discussion of promise problems in Section 2.4.1. Analogously to Definition 2.30, we say that the promise problem $\Pi = (S_{\text{yes}}, S_{\text{no}})$ is in (promise problem extension of) $\mathcal{BPP}$ *if there exists a probabilistic polynomial-time algorithm A such that for every $x \in S_{\text{yes}}$ it holds that* $\mathsf{Pr}[A(x)=1] \geq 2/3$ *and for every $x \in S_{\text{no}}$ it holds that* $\mathsf{Pr}[A(x)=0] \geq 2/3$. Similarly, Π is in co$\mathcal{RP}$ if for every $x \in S_{\text{yes}}$ it holds that $\mathsf{Pr}[A(x)=1] = 1$ and for every $x \in S_{\text{no}}$ it holds that $\mathsf{Pr}[A(x)=0] \geq 1/2$. Probabilistic reductions among promise problems are defined by adapting the conventions of Section 2.4.1; specifically, queries that violate the promise at the target of the reduction may be answered arbitrarily.

Theorem 6.7 *Any problem in $\mathcal{BPP}$ is reducible by a one-sided error randomized Karp-reduction to* co$\mathcal{RP}$, *where* co$\mathcal{RP}$ *(and possibly also $\mathcal{BPP}$) denotes the corresponding class of promise problems. Specifically, the reduction always maps a* no*-instance to a* no*-instance.*

It follows that $\mathcal{BPP}$ is reducible by a one-sided error randomized Cook-reduction to $\mathcal{RP}$. Thus, using the conventions of Section 3.2.2 and referring to classes of promise problems, we may write $\mathcal{BPP} \subseteq \mathcal{RP}^{\mathcal{RP}}$. In fact, since $\mathcal{RP}^{\mathcal{RP}} \subseteq \mathcal{BPP}^{\mathcal{BPP}} = \mathcal{BPP}$, we have $\mathcal{BPP} = \mathcal{RP}^{\mathcal{RP}}$. Theorem 6.7 may be paraphrased as saying that the combination of the one-sided error probability of the reduction and the one-sided error probability of co$\mathcal{RP}$ can account for the two-sided error probability of $\mathcal{BPP}$. We warn that this statement is not a triviality like $1 + 1 = 2$, and in particular we do not know whether it holds for classes of standard decision problems (rather than for the classes of promise problems considered in Theorem 6.7).

Proof: Recall that we can easily reduce the error probability of BPP-algorithms, and derive probabilistic polynomial-time algorithms of exponentially vanishing error probability. But this does not eliminate the error (even on "one side") altogether. In general, there seems to be no hope to eliminate the error, unless we (either do something earth-shaking or) change the setting as done when allowing a one-sided error randomized reduction to a problem in co$\mathcal{RP}$. The latter setting can be viewed as a two-move randomized game (i.e., a random move by the reduction followed by a random move by the decision procedure of co$\mathcal{RP}$), and it enables applying different quantifiers to the two moves (i.e., allowing error in one direction in the first quantifier and error in the other direction in the second quantifier). In the next paragraph, which is inessential to the actual proof, we illustrate the potential power of this setting.

Teaching note: The following illustration represents an alternative way of proving Theorem 6.7. This way seems conceptual simpler but it requires a starting point (or rather an assumption) that is much harder to establish, where both comparisons are with respect to the actual proof of Theorem 6.7 (which follows the illustration).

An illustration. Suppose that for some set $S \in \mathcal{BPP}$ there exists a polynomial p' and an off-line BPP-algorithm A' such that for every x it holds that $\mathsf{Pr}_{r\in\{0,1\}^{2p'(|x|)}}[A'(x,r)\neq\chi_S(x)] < 2^{-(p'(|x|)+1)}$; that is, the algorithm uses $2p'(|x|)$ bits of randomness and has error probability smaller than $2^{-p'(|x|)}/2$. Note that such an algorithm cannot be obtained by standard error-reduction (see Exercise 6.9). Anyhow (by applying a union bound), such a small error probability allows for meaningful outcomes even if only half of the string r is random. Thus, for every $x \in S$, it holds that $\mathsf{Pr}_{r'\in\{0,1\}^{p'(|x|)}}[(\forall r'' \in \{0,1\}^{p'(|x|)})\, A'(x,r'r'')=1] > 1/2$, whereas for every $x \notin S$ and every $r' \in \{0,1\}^{p'(|x|)}$ it holds that $\mathsf{Pr}_{r''\in\{0,1\}^{p'(|x|)}}[A'(x,r'r'')=1] < 1/2$. That is, the error on yes-instances is "pushed" to the selection of r', whereas the error on no-instances is pushed to the selection of r''. This yields a one-sided error randomized Karp-reduction that maps x to (x,r'), where r' is uniformly selected in $\{0,1\}^{p'(|x|)}$, such that deciding S is reduced to the coRP problem (regarding pairs (x,r')) that is decided by the (on-line) randomized algorithm A'' defined by $A''(x,r') \stackrel{\rm def}{=} A'(x,r'U_{p'(|x|)})$. For details, see Exercise 6.10. The actual proof, which avoids the aforementioned hypothesis, follows.

The actual starting point. Consider any BPP-problem with a characteristic function χ (which, in case of a promise problem, is a partial function, defined only over the promise). By standard error-reduction, there exists a probabilistic polynomial-time algorithm A such that for every x on which χ is defined it holds that $\mathsf{Pr}[A(x)\neq\chi(x)] < \mu(|x|)$, where μ is a negligible function. Looking at the corresponding off-line algorithm A' and denoting by p the polynomial that bounds the running time of A, we have

$$\mathsf{Pr}_{r\in\{0,1\}^{p(|x|)}}[A'(x,r)\neq\chi(x)] \;<\; \mu(|x|) \;<\; \frac{1}{2p(|x|)} \tag{6.1}$$

for all sufficiently long x's on which χ is defined. We show a randomized one-sided error Karp-reduction of χ to a promise problem in co$\mathcal{RP}$.

The main idea. As in the illustrating paragraph, the basic idea is "pushing" the error probability on yes-instances (of χ) to the reduction, while pushing the error probability on no-instances to the coRP-problem. Focusing on the case that $\chi(x) = 1$, this is achieved by augmenting the input x with a random sequence of "modifiers" that act on the random-input of algorithm A such that for a good choice of modifiers it holds that for every $r \in \{0,1\}^{p(|x|)}$ there exists a modifier in this sequence that when applied to r yields r' that satisfies $A(x,r') = 1$. Indeed, not all sequences of modifiers are good, but a random sequence will be good with high probability and bad sequences will be accounted for in the error probability of the reduction. On the other hand, using only modifiers that are permutations guarantees that the error probability on no-instances only increase by a factor that equals the number of modifiers we use, and this error probability will be accounted for by the error probability of the coRP-problem. Details follow.

The aforementioned modifiers are implemented by shifts (of the set of all strings by fixed offsets). Thus, we augment the input x with a random sequence of shifts, denoted $s_1, ..., s_m \in \{0,1\}^{p(|x|)}$, such that for a good choice of $(s_1, ..., s_m)$ it holds that for every $r \in \{0,1\}^{p(|x|)}$ there exists an $i \in [m]$ such that $A'(x, r \oplus s_i) = 1$. We will show that, for any yes-instance x and a suitable choice of m, with very high probability, a random sequence of shifts is good. Thus, for $A''(\langle x, s_1, ..., s_m \rangle, r) \stackrel{\text{def}}{=} \vee_{i=1}^{m} A'(x, r \oplus s_i)$, it holds that, with very high probability over the choice of $s_1, ..., s_m$, a yes-instance x is mapped to an augmented input $\langle x, s_1, ..., s_m \rangle$ that is accepted by A'' with probability 1. On the other hand, the acceptance probability of augmented no-instances (for any choice of shifts) only increases by a factor of m. In further detailing the foregoing idea, we start by explicitly stating the simple randomized mapping (to be used as a randomized Karp-reduction), and next define the target promise problem.

The randomized mapping. On input $x \in \{0,1\}^n$, we set $m = p(|x|)$, uniformly select $s_1, ..., s_m \in \{0,1\}^m$, and output the pair $(x, \overline{s})$, where $\overline{s} = (s_1, ..., s_m)$. Note that this mapping, denoted M, is easily computable by a probabilistic polynomial-time algorithm.

The promise problem. We define the following promise problem, denoted $\Pi = (\Pi_{\text{yes}}, \Pi_{\text{no}})$, having instances of the form $(x, \overline{s})$ such that $|\overline{s}| = p(|x|)^2$.

- The yes-instances are pairs $(x, \overline{s})$, where $\overline{s} = (s_1, ..., s_m)$ and $m = p(|x|)$, such that for every $r \in \{0,1\}^m$ there exists an i satisfying $A'(x, r \oplus s_i) = 1$.
- The no-instances are pairs $(x, \overline{s})$, where again $\overline{s} = (s_1, ..., s_m)$ and $m = p(|x|)$, such that for at least half of the possible $r \in \{0,1\}^m$, for every i it holds that $A'(x, r \oplus s_i) = 0$.

To see that Π is indeed a co$\mathcal{RP}$ promise problem, we consider the following randomized algorithm. On input $(x, (s_1, ..., s_m))$, where $m = p(|x|) = |s_1| = \cdots = |s_m|$, the algorithm uniformly selects $r \in \{0,1\}^m$, and accepts if and only if $A'(x, r \oplus s_i) = 1$ for some $i \in \{1, ..., m\}$. Indeed, yes-instances of Π are accepted with probability 1, whereas no-instances of Π are rejected with probability at least 1/2.

Analyzing the reduction: We claim that the randomized mapping M reduces χ to Π with one-sided error. Specifically, we will prove two claims.

Claim 1: If x is a yes-instance (i.e., $\chi(x) = 1$) then $\Pr[M(x) \in \Pi_{\text{yes}}] > 1/2$.

Claim 2: If x is a no-instance (i.e., $\chi(x) = 0$) then $\Pr[M(x) \in \Pi_{\text{no}}] = 1$.

We start with Claim 2, which is easier to establish. Recall that $M(x) = (x, (s_1, ..., s_m))$, where $s_1, ..., s_m$ are uniformly and independently distributed in $\{0,1\}^m$. We note that (by Eq. (6.1) and $\chi(x) = 0$), for every possible choice of $s_1, ..., s_m \in \{0,1\}^m$ and every $i \in \{1, ..., m\}$, the fraction of r's that satisfy $A'(x, r \oplus s_i) = 1$ is at most $\frac{1}{2m}$. Thus, for every possible choice of $s_1, ..., s_m \in \{0,1\}^m$, the fraction of r's for which there exists an i such that $A'(x, r \oplus s_i) = 1$ holds is at most 1/2. Hence, the reduction M always maps the no-instance x (i.e., $\chi(x) = 0$) to a no-instance of Π (i.e., an element of $\Pi_{\rm no}$).

Turning to Claim 1 (which refers to $\chi(x) = 1$), we will show shortly that in this case, with very high probability, the reduction M maps x to a yes-instance of Π. We upper-bound the probability that the reduction fails (in case $\chi(x) = 1$) as follows:

$$\begin{aligned}
\Pr[M(x) \notin \Pi_{\rm yes}] &= \Pr_{s_1,\ldots,s_m}[\exists r \in \{0,1\}^m \text{ s.t. } (\forall i)\ A'(x, r \oplus s_i) = 0] \\
&\leq \sum_{r \in \{0,1\}^m} \Pr_{s_1,\ldots,s_m}[(\forall i)\ A'(x, r \oplus s_i) = 0] \\
&= \sum_{r \in \{0,1\}^m} \prod_{i=1}^{m} \Pr_{s_i}[A'(x, r \oplus s_i) = 0] \\
&< 2^m \cdot \left(\frac{1}{2m}\right)^m
\end{aligned}$$

where the last inequality is due to Eq. (6.1). It follows that if $\chi(x) = 1$ then $\Pr[M(x) \in \Pi_{\rm yes}] \gg 1/2$. Thus, the randomized mapping M reduces χ to Π, with one-sided error on yes-instances. Recalling that $\Pi \in {\rm co}\mathcal{RP}$, the theorem follows. ■

Corollaries. The traditional presentation uses the foregoing reduction for showing that $\mathcal{BPP}$ is in the Polynomial-Time Hierarchy (where both classes refer to standard decision problems). Specifically, to prove that $\mathcal{BPP} \subseteq \Sigma_2$ (see Definition 3.8), define the polynomial-time computable predicate $\varphi(x, \overline{s}, r) \stackrel{\rm def}{=} \bigvee_{i=1}^{m} (A'(x, s_i \oplus r) = 1)$, and observe that

$$\chi(x) = 1 \quad\Rightarrow\quad \exists \overline{s}\, \forall r\ \ \varphi(x, \overline{s}, r) \tag{6.2}$$

$$\chi(x) = 0 \quad\Rightarrow\quad \forall \overline{s}\, \exists r\ \ \neg\varphi(x, \overline{s}, r) \tag{6.3}$$

(where Eq. (6.3) is equivalent to $\neg\exists\overline{s}\,\forall r\ \ \varphi(x, \overline{s}, r)$). Note that Claim 1 (in the proof of Theorem 6.7) establishes that *most* sequences $\overline{s}$ satisfy $\forall r\, \varphi(x, \overline{s}, r)$, whereas Eq. (6.2) only requires the existence of *at least one* such $\overline{s}$. Similarly, Claim 2 establishes that for every $\overline{s}$ *most* choices of r violate $\varphi(x, \overline{s}, r)$, whereas Eq. (6.3) only requires that for every $\overline{s}$ there exists *at least one* such r. We comment that the same proof idea yields a variety of similar statements (e.g., $\mathcal{BPP} \subseteq \mathcal{MA}$, where $\mathcal{MA}$ is a randomized version of $\mathcal{NP}$ defined in Section 9.1).[6]

[6] Specifically, the class $\mathcal{MA}$ is defined by allowing the verification algorithm V in Definition 2.5 to be probabilistic and err on no-instances; that is, for every $x \in S$ there exists $y \in \{0,1\}^{{\rm poly}(|x|)}$ such that $\Pr[V(x,y)=1] = 1$, whereas for every $x \notin S$ and every y it holds that $\Pr[V(x,y)=0] \geq 1/2$. We note that $\mathcal{MA}$ can be viewed as a hybrid of the two aforementioned pairs of conditions; specifically, each problem in $\mathcal{MA}$ satisfy the conjunction of Eq. (6.2) and Claim 2. (Note that it makes no sense to require zero error on yes-instances, whereas the two-sided error version is equivalent to $\mathcal{MA}$; see Exercise 6.12.)

6.1.3 Zero-sided error: The complexity class ZPP

We now consider probabilistic polynomial-time algorithms that never err, but may fail to provide an answer. Focusing on decision problems, the corresponding class is denoted $\mathcal{ZPP}$ (standing for Zero-error Probabilistic Polynomial-time). The standard definition of $\mathcal{ZPP}$ is in terms of machines that output $\perp$ (indicating failure) with probability at most $1/2$. That is, $S \in \mathcal{ZPP}$ *if there exists a probabilistic polynomial-time algorithm A such that for every $x \in \{0,1\}^*$ it holds that* $\mathsf{Pr}[A(x) \in \{\chi_S(x), \perp\}] = 1$ *and* $\mathsf{Pr}[A(x) = \chi_S(x)] \geq 1/2$, *where* $\chi_S(x) = 1$ *if* $x \in S$ *and* $\chi_S(x) = 0$ *otherwise*. Again, the choice of the constant (i.e., $1/2$) is immaterial, and "error-reduction" can be performed showing that algorithms that yield a meaningful answer with noticeable probability can be amplified to algorithms that fail with negligible probability (see Exercise 6.6).

Theorem 6.8 $\mathcal{ZPP} = \mathcal{RP} \cap \mathrm{co}\mathcal{RP}$.

Proof Sketch: The fact that $\mathcal{ZPP} \subseteq \mathcal{RP}$ (as well as $\mathcal{ZPP} \subseteq \mathrm{co}\mathcal{RP}$) follows by a trivial transformation of the ZPP-algorithm; that is, replacing the failure indicator $\perp$ by a "no" verdict (resp., "yes" verdict). Note that the choice of what to say in case the ZPP-algorithm fails is determined by the type of error that we are allowed.

In order to prove that $\mathcal{RP} \cap \mathrm{co}\mathcal{RP} \subseteq \mathcal{ZPP}$ we combine the two algorithm guaranteed for a set in $\mathcal{RP} \cap \mathrm{co}\mathcal{RP}$. The point is that we can trust the RP-algorithm (resp., coNP-algorithm) in the case that it says "yes" (resp., "no"), but not in the case that it says "no" (resp., "yes"). Thus, we invoke both algorithms, and output a definite answer only if we obtain an answer that we can trust (which happen with high probability). Otherwise, we output $\perp$. $\square$

Expected polynomial-time. In some sources $\mathcal{ZPP}$ is defined in terms of randomized algorithms that run in expected polynomial-time and always output the correct answer. This definition is equivalent to the one we used (see Exercise 6.7).

6.1.4 Randomized Log-Space

In this section we discuss probabilistic polynomial-time algorithms that are further restricted such that they are allowed to use only a logarithmic amount of space.

6.1.4.1 Definitional issues

When defining space-bounded randomized algorithms, we face a problem analogous to the one discussed in the context of non-deterministic space-bounded computation (see Section 5.3). Specifically, the on-line and the off-line versions (formulated in Definition 6.1) are no longer equivalent, unless we restrict the off-line machine to access its random-input tape in a uni-directional manner. The issue is that, in the context of space-bounded computation (and unlike in the case that we only care about time-bounds), the results of the internal coin tosses (in the on-line model) cannot be recorded for free. Bearing in mind that, *in the current context*, we wish to model real algorithms (rather than present a fictitious model that captures a fundamental phenomena as in Section 5.3), it is clear that *using the on-line version is the natural choice.*

An additional issue that arises is the need to explicitly bound the running-time of space-bounded randomized algorithms. Recall that, without loss of generality, the number of steps taken by a space-bounded non-deterministic machine is at most exponential in its space complexity, because the shortest path between two configurations in the (directed) graph of possible configurations is upper-bounded by its size (which in turn is exponential in the space-bound). This reasoning fails in the case of randomized algorithms, because the shortest path between two configurations does not bound the expected number of random steps required for going from the first configuration to the second one. In fact, as we shall shortly see, failing to upper-bound the running time of log-space randomized algorithms seems to allow them too much power; that is, such (unrestricted) log-space randomized algorithms can emulate non-deterministic log-space computations (in exponential time). The emulation consists of repeatedly invoking the NL-machine, while using random choices in the role of the non-deterministic moves. If the input is a yes-instance then, in each attempt, with probability at least 2^{-t}, we "hit" an accepting t-step (non-deterministic) computation, where t is polynomial in the input length. Thus, the randomized machine accepts such a yes-instance after an expected number of 2^t trials. To allow for the rejection of no-instances (rather than looping infinitely in vain), we wish to implement a counter that counts till 2^t (or so) and reject the input if this number of trials have failed. We need to implement such a counter within space $O(\log t)$ rather than t (which is easy). In fact, it suffices to have a "randomized counter" that, with high probability, counts to approximately 2^t. The implementation of such a counter is left to Exercise 6.15, and using it we may obtain a randomized algorithm that halts with high probability (on every input), always rejects a no-instance, and accepts each yes-instance with probability at least 1/2.

In light of the foregoing discussion, when defining randomized log-space algorithms we explicitly require that the algorithms halt in polynomial-time. Under this convention, the class $\mathcal{RL}$ relates to $\mathcal{NL}$ analogously to the relation of $\mathcal{RP}$ to $\mathcal{NP}$. That is, the probabilistic acceptance condition in $\mathcal{RL}$ is as in the case of $\mathcal{RP}$. The class $\mathcal{BPL}$ is defined similarly, when using a probabilistic acceptance condition as in the case of $\mathcal{BPP}$.

Definition 6.9 (the classes $\mathcal{RL}$ and $\mathcal{BPL}$): *We say that a randomized log-space algorithm is* admissible *if it always halts in a polynomial number of steps.*

- *A decision problem S is in $\mathcal{RL}$ if there exists an admissible* (on-line) *randomized log-space algorithm A such that for every $x \in S$ it holds that* $\Pr[A(x) = 1] \geq 1/2$ *and for every $x \notin S$ it holds that* $\Pr[A(x) = 0] = 1$.

- *A decision problem S is in $\mathcal{BPL}$ if there exists an admissible* (on-line) *randomized log-space algorithm A such that for every $x \in S$ it holds that* $\Pr[A(x) = 1] \geq 2/3$ *and for every $x \notin S$ it holds that* $\Pr[A(x) = 0] \geq 2/3$.

Clearly, $\mathcal{RL} \subseteq \mathcal{NL} \subseteq \mathcal{P}$ and $\mathcal{BPL} \subseteq \mathcal{P}$. Note that the classes $\mathcal{RL}$ and $\mathcal{BPL}$ remain unchanged even if we allow the algorithms to run for *expected* polynomial-time and have non-halting computations. Such algorithms can be easily transformed into admissible algorithms by truncating long computations, while using a (standard) counter (which can be implemented in logarithmic-space). Also note that error-reduction is applicable in the current setting (while essentially preserving both the time and space bounds).

6.1.4.2 The accidental tourist sees it all

An appealing example of a randomized log-space algorithm is presented next. It refers to the problem of deciding undirected connectivity, and demonstrated that this problem is in $\mathcal{RL}$. (Recall that in Section 5.2.4 we proved that this problem is actually in $\mathcal{L}$, but the algorithm and its analysis were more complicated.) Recall that Directed Connectivity is complete for $\mathcal{NL}$ (under log-space reductions). For sake of simplicity, we consider the following version of undirected connectivity, which is equivalent under log-space reductions to the version in which one needs to determine whether or not the input (undirected) graph is connected. In the current version, *the input consists of a triple (G, s, t), where G is an undirected graph, s, t are two vertices in G, and the task is to determine whether or not s and t are connected in G.*

Construction 6.10 *On input (G, s, t), the randomized algorithm starts a* $\text{poly}(|G|)$*-long random walk at vertex s, and accepts the triplet if and only if the walk passed through the vertex t. By a random walk we mean that at each step the algorithm selects uniformly one of the neighbors of the current vertex and moves to it.*

Observe that the algorithm can be implemented in logarithmic space (because we only need to store the current vertex as well as the number of steps taken so far). Obviously, if s and t are not connected in G then the algorithm always rejects (G, s, t). Proposition 6.11 implies that undirected connectivity is indeed in $\mathcal{RL}$.

Proposition 6.11 *If s and t are connected in $G = (V, E)$ then a random walk of length $O(|V| \cdot |E|)$ starting at s passes through t with probability at least* $1/2$.

In other words, a random walk starting at s visits all vertices of the connected component of s (i.e., it sees all that there is to see).

Proof Sketch: We will actually show that if G is connected then, with probability at least $1/2$, a random walk starting at s visits all the vertices of G. For any pair of vertices (u, v), let $X_{u,v}$ be a random variable representing the number of steps taken in a random walk starting at u until v is *first encountered*. The reader may verify that for every edge $\{u, v\} \in E$ it holds that $\mathrm{E}[X_{u,v}] \leq 2|E|$; see Exercise 6.16. Next, we let $\text{cover}(G)$ denote the expected number of steps in a random walk starting at s and ending when the last of the vertices of V is encountered. Our goal is to upper-bound $\text{cover}(G)$. Towards this end, we consider an arbitrary directed cyclic-tour

C that visits all vertices in G, and note that

$$\mathrm{cover}(G) \leq \sum_{(u,v)\in C} \mathrm{E}[X_{u,v}] \leq |C| \cdot 2|E|.$$

In particular, selecting C as a traversal of some spanning tree of G, we conclude that $\mathrm{cover}(G) < 4 \cdot |V| \cdot |E|$. Thus, with probability at least 1/2, a random walk of length $8 \cdot |V| \cdot |E|$ starting at s visits all vertices of G. □

6.2 Counting

We now turn to a new type of computational problems, which vastly generalize decision problems of the NP-type. We refer to counting problems, and more specifically to counting objects that can be efficiently recognized. The two formulations of NP provide a suitable definition of such objects and yield corresponding counting problems:

1. Counting the number of solutions for a given instance of a search problem (of a relation) $R \subseteq \{0,1\}^* \times \{0,1\}^*$ having efficiently checkable solutions (i.e., $R \in \mathcal{PC}$; see Definition 2.3). That is, on input x, we are required to output $|\{y : (x,y) \in R\}|$.
2. Counting the number of NP-witnesses (with respect to a specific verification procedure V) for a given instance of an NP-set S (i.e., $S \in \mathcal{NP}$ and V is the corresponding verification procedure; see Definition 2.5). That is, on input x, we are required to output $|\{y : V(x,y)=1\}|$.

We shall consider these counting problems as well as relaxations of them (which refer to approximating the said quantities), and see connections between these relaxed counting problems and randomized algorithms.

6.2.1 Exact Counting

In continuation to the foregoing discussion, we define the class of problems concerned with counting efficiently recognized objects. (Recall that $\mathcal{PC}$ denotes the class of search problems having polynomially long solutions that are efficiently checkable; see Definition 2.3.)

Definition 6.12 (counting efficiently recognized objects – $\#\mathcal{P}$): *The class $\#\mathcal{P}$ consists of all functions that count solutions to a search problem in $\mathcal{PC}$. That is, $f : \{0,1\}^* \to \mathbb{N}$ is in $\#\mathcal{P}$ if there exists $R \in \mathcal{PC}$ such that, for every x, it holds that $f(x) = |R(x)|$, where $R(x) = \{y : (x,y) \in R\}$. In this case we say that f is the* counting problem associated with *R, and denote the latter by $\#R$* (i.e., $\#R = f$).

Every decision problem in $\mathcal{NP}$ is Cook-reducible to $\#\mathcal{P}$, because every such problem can be cast as deciding membership in $S_R = \{x : |R(x)| > 0\}$ for some $R \in \mathcal{PC}$ (see Section 2.1.2). It also holds that $\mathcal{BPP}$ is Cook-reducible to $\#\mathcal{P}$ (see Exercise 6.17). The class $\#\mathcal{P}$ is sometimes defined in terms of decision problems, as is implicit in the following proposition.

Proposition 6.13 (a decisional version of $\#\mathcal{P}$): *For any $f \in \#\mathcal{P}$, deciding membership in $S_f \stackrel{\text{def}}{=} \{(x, N) : f(x) \geq N\}$ is computationally equivalent to computing f.*

Actually, the claim holds for any function $f : \{0,1\}^* \to \mathbb{N}$ for which there exists a polynomial p such that for every $x \in \{0,1\}^*$ it holds that $f(x) \leq 2^{p(|x|)}$.

Proof: Since the relation R vouching for $f \in \#\mathcal{P}$ (i.e., $f(x) = |R(x)|$) is polynomially bounded, there exists a polynomial p such that for every x it holds that $f(x) \leq 2^{p(|x|)}$. Deciding membership in S_f is easily reduced to computing f (i.e., we accept the input (x, N) if and only if $f(x) \geq N$). Computing f is reducible to deciding S_f by using a binary search (see Exercise 2.9). This relies on the fact that, on input x and oracle access to S_f, we can determine whether or not $f(x) \geq N$ by making the query (x, N). Note that we know a priori that $f(x) \in [0, 2^{p(|x|)}]$. ■

The counting class $\#\mathcal{P}$ is also related to the problem of enumerating all possible solutions to a given instance (see Exercise 6.18).

6.2.1.1 On the power of $\#\mathcal{P}$

As indicated, $\mathcal{NP} \cup \mathcal{BPP}$ is (easily) reducible to $\#\mathcal{P}$. Furthermore, as stated in Theorem 6.14, the entire Polynomial-Time Hierarchy (as defined in Section 3.2) is Cook-reducible to $\#\mathcal{P}$ (i.e., $\mathcal{PH} \subseteq \mathcal{P}^{\#\mathcal{P}}$). On the other hand, any problem in $\#\mathcal{P}$ is solvable in polynomial space, and so $\mathcal{P}^{\#\mathcal{P}} \subseteq \mathcal{PSPACE}$.

Theorem 6.14 *Every set in $\mathcal{PH}$ is Cook-reducible to $\#\mathcal{P}$.*

We do not present a proof of Theorem 6.14 here, because the known proofs are rather technical. Furthermore, one main idea underlying these proofs appears in a more clear form in the proof of Theorem 6.27. Nevertheless, in Section F.1 we present a proof of a related result, which implies that $\mathcal{PH}$ is reducible to $\#\mathcal{P}$ via *randomized* Karp-reductions.

6.2.1.2 Completeness in $\#\mathcal{P}$

The definition of $\#\mathcal{P}$-completeness is analogous to the definition of $\mathcal{NP}$-completeness. That is, *a counting problem f is* $\#\mathcal{P}$-complete *if $f \in \#\mathcal{P}$ and every problem in $\#\mathcal{P}$ is Cook-reducible to f.*

We claim that the counting problems associated with the NP-complete problems presented in Section 2.3.3 are all $\#\mathcal{P}$-complete. We warn that this fact is not due to the mere NP-completeness of these problems, but rather to an additional property of the reductions establishing their NP-completeness. Specifically, the Karp-reductions that were used (or variants of them) have the extra property of preserving the number of NP-witnesses (as captured by the following definition).

Definition 6.15 (parsimonious reductions): *Let $R, R' \in \mathcal{PC}$ and let g be a Karp-reduction of $S_R = \{x : R(x) \neq \emptyset\}$ to $S_{R'} = \{x : R'(x) \neq \emptyset\}$, where $R(x) = \{y : (x, y) \in R\}$ and $R'(x) = \{y : (x, y) \in R'\}$. We say that g is* parsimonious (with respect to R and R') *if for every x it holds that $|R(x)| = |R'(g(x))|$. In such a case we say that g is a* parsimonious reduction of R to R'.

We stress that the condition of being parsimonious refers to the two underlying relations R and R' (and not merely to the sets S_R and $S_{R'}$). The requirement that g is a Karp-reduction is partially redundant, because if g is polynomial-time computable and for every x it holds that $|R(x)| = |R'(g(x))|$, then g constitutes a Karp-reduction of S_R to $S_{R'}$. Specifically, $|R(x)| = |R'(g(x))|$ implies that $|R(x)| > 0$ (i.e., $x \in S_R$) if and only if $|R'(g(x))| > 0$ (i.e., $g(x) \in S_{R'}$). The reader may easily verify that the Karp-reduction underlying the proof of Theorem 2.18 as well as many of the reductions used in Section 2.3.3 are parsimonious (see Exercise 2.17).

Theorem 6.16 *Let $R \in \mathcal{PC}$ and suppose that every search problem in $\mathcal{PC}$ is parsimoniously reducible to R. Then the counting problem associated with R is $\#\mathcal{P}$-complete.*

Proof: Clearly, the counting problem associated with R, denoted $\#R$, is in $\#\mathcal{P}$. To show that every $f' \in \#\mathcal{P}$ is reducible to f, we consider the relation $R' \in \mathcal{PC}$ that is counted by f'; that is, $\#R' = f'$. Then, by the hypothesis, there exists a parsimonious reduction g of R' to R. This reduction also reduces $\#R'$ to $\#R$; specifically, $\#R'(x) = \#R(g(x))$ for every x. ■

Corollaries. As an immediate corollary of Theorem 6.16, we get that counting the number of satisfying assignments to a given CNF formula is $\#\mathcal{P}$-complete. Similar statement hold for all the other NP-complete problems mentioned in Section 2.3.3 and in fact for all NP-complete problems listed in [81]. These corollaries follow from the fact that all known reductions among natural NP-complete problems are either parsimonious or can be easily modified to be so.

We conclude that many counting problems associated with NP-complete search problems are $\#\mathcal{P}$-complete. It turns out that also counting problems associated with efficiently solvable search problems may be $\#\mathcal{P}$-complete.

Theorem 6.17 *There exist $\#\mathcal{P}$-complete counting problems that are associated with efficiently solvable search problems. That is, there exists $R \in \mathcal{PF}$* (see Definition 2.2) *such that $\#R$ is $\#\mathcal{P}$-complete.*

Proof: Consider the relation $R_{\mathtt{dnf}}$ consisting of pairs (ϕ, τ) such that ϕ is a DNF formula and τ is an assignment satisfying it. Note that the search problem of $R_{\mathtt{dnf}}$ is easy to solve (e.g., by picking an arbitrary truth assignment that satisfies the first term in the input formula). To see that $\#R_{\mathtt{dnf}}$ is $\#\mathcal{P}$-complete consider the following reduction from $\#R_{\mathtt{SAT}}$ (which is $\#\mathcal{P}$-complete by Theorem 6.16). Given a CNF formula ϕ, transform $\neg\phi$ into a DNF formula ϕ' by applying de-Morgan's Law, and return $2^n - \#R_{\mathtt{dnf}}(\phi')$, where n denotes the number of variables in ϕ (resp., ϕ'). ■

Reflections. We note that Theorem 6.17 is not established by a parsimonious reduction. This fact should not come as a surprise because a parsimonious reduction of $\#R'$ to $\#R$ implies that $S_{R'} = \{x : \exists y \text{ s.t. } (x,y) \in R'\}$ is reducible to

$S_R = \{x : \exists y \text{ s.t. } (x,y) \in R\}$, where in our case $S_{R'}$ is NP-Complete while $S_R \in \mathcal{P}$ (since $R \in \mathcal{PF}$). Nevertheless, the proof of Theorem 6.17 is related to the hardness of some underlying decision problem (i.e., the problem of deciding whether a given DNF formula is a tautology (i.e., whether $\#R_{\texttt{dnf}}(\phi') = 2^n$)). But does there exist a $\#\mathcal{P}$-complete problem that is "not based on some underlying NP-complete decision problem"? Amazingly enough, the answer is positive.

Theorem 6.18 *Counting the number of perfect matchings in a bipartite graph is $\#\mathcal{P}$-complete.*[7]

Equivalently (see Exercise 6.19), the problem of computing the permanent of matrices with 0/1-entries is $\#\mathcal{P}$-complete. Recall that the permanent of an n-by-n matrix $M = (m_{i,j})$, denoted $\texttt{perm}(M)$, equals the sum over all permutations π of $[n]$ of the products $\prod_{i=1}^{n} m_{i,\pi(i)}$. Theorem 6.18 is proven by composing the following two (many-to-one) reductions (asserted in Propositions 6.19 and 6.20, respectively) and using the fact that $\#R_{\texttt{3SAT}}$ is $\#\mathcal{P}$-complete (see Theorem 6.16 and Exercise 2.17). Needless to say, the resulting reduction is not parsimonious.

Proposition 6.19 *The counting problem of* 3SAT (i.e., $\#R_{\texttt{3SAT}}$) *is reducible to computing the permanent of integer matrices. Furthermore, there exists an* even *integer $c > 0$ and a finite set of integers I such that, on input a 3CNF formula ϕ, the reduction produces an integer matrix with entries in I and a permanent value that equals $c^m \cdot \#R_{\texttt{3SAT}}(\phi)$, where m denotes the number of clauses in ϕ.*

The original proof of Proposition 6.19 uses $c = 2^{10}$ and $I = \{-1, 0, 1, 2, 3\}$. It follows that, for every integer $n > 1$ that is relatively prime to c, computing the permanent modulo n is NP-hard (see Exercise 6.20, which also uses Theorem 6.27). Thus, using the case of $c = 2^{10}$, this means that computing the permanent modulo n is NP-hard for any odd $n > 1$. In contrast, computing the permanent modulo 2 (which is equivalent to computing the determinant modulo 2) is easy (i.e., can be done in polynomial-time and even in $\mathcal{NC}$). Thus, assuming $\mathcal{NP} \not\subseteq \mathcal{BPP}$, Proposition 6.19 cannot hold for an odd c (because by Exercise 6.20 it would follow that computing the permanent modulo 2 is NP-Hard). We also note that, assuming $\mathcal{P} \neq \mathcal{NP}$, Proposition 6.19 cannot possibly hold for a set I containing only non-negative integers (see Exercise 6.21).

Proposition 6.20 *Computing the permanent of integer matrices is reducible to computing the permanent of 0/1-matrices. Furthermore, the reduction transforms an integer matrix A into a 0/1-matrix A'' such that the permanent of A can be easily computed from A and the permanent of A''.*

[7] See Section G.1 for basic terminology regarding graphs.

Teaching note: We do not recommend presenting the proofs of Propositions 6.19 and 6.20 in class. The high-level structure of the proof of Proposition 6.19 has the flavor of some sophisticated reductions among NP-problems, but the crucial point is the existence of adequate gadgets. We do not know of a high-level argument establishing the existence of such gadgets nor of any intuition as to why such gadgets exist.[8] Instead, the existence of such gadgets is proved by a design that is both highly non-trivial and *ad hoc* in nature. Thus, the proof of Proposition 6.19 boils down to a complicated design problem that is solved in a way that has little pedagogical value. In contrast, the proof of Proposition 6.20 uses two simple ideas that can be useful in other settings. With suitable hints, this proof can be used as a good exercise.

Proof of Proposition 6.19: We will use the correspondence between the permanent of a matrix A and the sum of the weights of the cycle covers of the weighted directed graph represented by the matrix A. A cycle cover of a graph is a collection of simple[9] *vertex-disjoint* directed cycles that covers all the graph's vertices, and its weight is the product of the weights of the corresponding edges. The SWCC of a weighted directed graph is the sum of the weights of all its cycle covers.

Given a 3CNF formula ϕ, we construct a directed weighted graph G_ϕ such that the SWCC of G_ϕ equals equals $c^m \cdot \#R_{\mathtt{3SAT}}(\phi)$, where c is a universal constant and m denotes the number of clauses in ϕ. We may assume, without loss of generality, that each clause of ϕ has exactly three variables (which are not necessarily distinct).

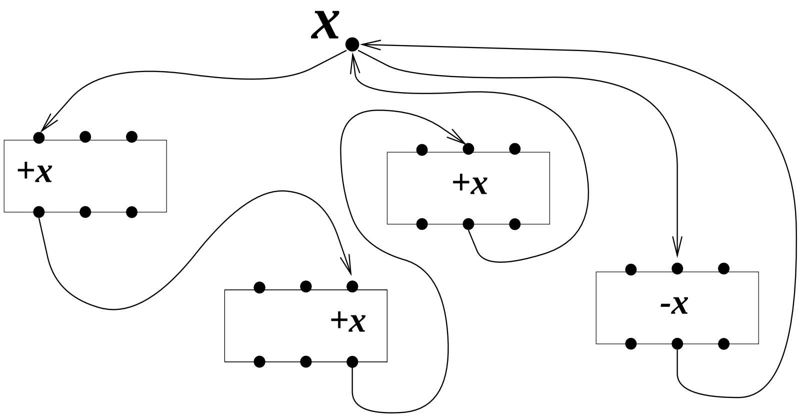


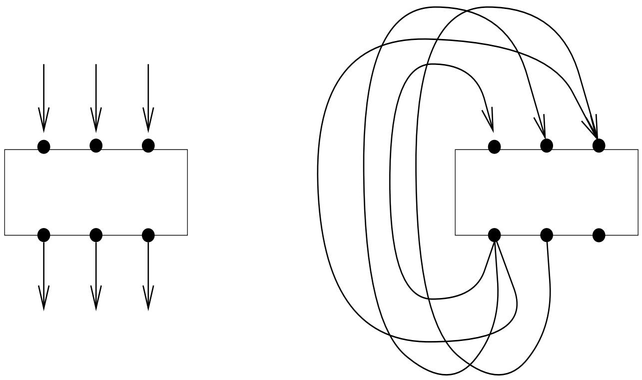
Figure 6.1: Tracks connecting gadgets for the reduction to cycle cover.

We start with a high-level description (of the construction) that refers to (clause) gadgets, each containing some internal vertices and internal (weighted) edges, which are *unspecified at this point*. In addition, each gadget has three pairs of designated vertices, one pair per each literal appearing in the clause, where one vertex in

[8]Indeed, the conjecture that such gadgets exist can only be attributed to ingenuity.

[9]Here a simple cycle is a strongly connected directed graph in which each vertex has a single incoming (resp., outgoing) edge. In particular, self-loops are allowed.

the pair is designated as an entry vertex and the other as an exit vertex. The graph G_ϕ consists of m such gadgets, one per each clause (of ϕ), and n auxiliary vertices, one per each variable (of ϕ), as well as some *additional directed edges*, each having weight 1. Specifically, for each variable, we introduce two tracks, one per each of the possible literals of this variable. The track associated with a literal consists of directed edges (each having weight 1) that form a simple "cycle" passing through the corresponding vertex as well as through the designated vertices that correspond to the occurrences of this literal in the various clauses. Specifically, for each such occurrence, the track enters the corresponding clause gadget at the entry-vertex corresponding to this literal and exits at the corresponding exit-vertex. (If a literal does not appear in ϕ then the corresponding track is a self-loop on the corresponding variable.) See Figure 6.1 showing the two tracks of a variable x that occurs positively in three clauses and negatively in one clause. The entry-vertices (resp., exit-vertices) are drawn on the top (resp., bottom) part of each gadget.



On the left is a gadget with the track edges adjacent to it (as in the real construction). On the right is a gadget and four out of the nine external edges (two of which are nice) used in the analysis.

Figure 6.2: External edges for the analysis of the clause gadget

For the purpose of stating the desired properties of the clause gadget, we augment the gadget by nine external edges (of weight 1), one per each pair of (not necessarily matching) entry and exit vertices such that the edge goes from the exit-vertex to the entry-vertex (see Figure 6.2). (We stress that this is an auxiliary construction that differs from and yet is related to the use of gadgets in the foregoing construction of G_ϕ.) The three edges that link the designated pairs of vertices that correspond to the three literals are called nice. We say that a collection of edges C (e.g., a collection of cycles) uses the external edges S if the intersection of C with the set of the (nine) external edges equals S. We postulate the following three properties of the clause gadget.

1. The sum of the weights of all cycle covers (of the gadget) that do not use any external edge (i.e., use the empty set of external edges) equals zero.

2. Let $V(S)$ denote the set of vertices incident to S, and say that S is nice if it is non-empty and the vertices in $V(S)$ can be perfectly matched using nice edges.[10] Then, there exists a constant c (indeed the one postulated in the proposition's claim) such that, for any nice set S, the sum of the weights of all cycle covers that use the external edges S equals c.

3. For any non-nice set S of external edges, the sum of the weights of all cycle covers that use the external edges S equals zero.

Note that the foregoing three cases exhaust all the possible ones, and that the set of external edges used by a cycle cover must be a matching (i.e., these edges are vertex disjoint). Using the foregoing conditions, it can be shown that each satisfying assignment of ϕ contributes exactly c^m to the SWCC of G_ϕ (see Exercise 6.22). It follows that the SWCC of G_ϕ equals $c^m \cdot \#R_{\mathtt{3SAT}}(\phi)$.

Having established the validity of the abstract reduction, we turn to the implementation of the clause gadget. The first implementation is a *Deus ex Machina*, with a corresponding adjacency matrix depicted in Figure 6.3. Its validity (for the value $c = 12$) can be verified by computing the permanent of the corresponding sub-matrices (see analogous analysis in Exercise 6.24).

A more structured implementation of the clause gadget is depicted in Figure 6.4, which refers to a (hexagon) box to be implemented later. The box contains several vertices and weighted edges, but only two of these vertices, called terminals, are connected to the outside (and are shown in Figure 6.4). The clause gadget consists of five copies of this box, where three copies are designated for the three literals of the clause (and are marked LB1, LB2, and LB3), as well as additional vertices and edges shown in Figure 6.4. In particular, the clause gadget contains the six aforementioned designated vertices (i.e., a pair of entry and exit vertices per each literal), two additional vertices (shown at the two extremes of the figure), and some edges (all having weight 1). Each designated vertex has a self-loop, and is incident to a single additional edge that is outgoing (resp., incoming) in case the vertex is an entry-vertex (resp., exit-vertex) of the gadget. The two terminals of each box that is associated with some literal are connected to the corresponding pair of designated vertices (e.g., the outgoing edge of entry1 is incident at the right terminal of the box LB1). Note that the five boxes reside on a directed path (going from left to right), and the only edges going in the opposite direction are those drawn below this path.

In continuation to the foregoing, we wish to state the desired properties of the box. Again, we do so by considering the augmentation of the box by external edges (of weight 1) incident at the specified vertices. In this case (see Figure 6.5), we have a pair of anti-parallel edges connecting the two terminals of the box as well as two self-loops (one on each terminal). We postulate the following three properties of the box.

[10] Clearly, any non-empty set of nice edges is a nice set. Thus, a singleton set is nice if and only if the corresponding edge is nice. On the other hand, any set S of three external edges is nice, because $V(S)$ has a perfect matching using all three nice edges. Thus, the notion of nice sets is "non-trivial" only for sets of two edges. Such a set S is nice if and only if $V(S)$ consists of two pairs of corresponding designated vertices.

The gadget uses eight vertices, where the first six are the designated (entry and exit) vertices. The entry-vertex (resp., exit-vertex) associated with the i^{th} literal is numbered i (resp., $i+3$). The corresponding adjacency matrix follows.

$$\begin{pmatrix} 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & -1 & 0 & 1 & 1 \\ 0 & 0 & -1 & -1 & 2 & 0 & 1 & 1 \\ 0 & 0 & 0 & -1 & -1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 2 & -1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Note that the edge $3 \to 6$ can be contracted, but the resulting 7-vertex graph will not be consistent with our (inessentially stringent) definition of a gadget by which the six designated vertices should be distinct.

Figure 6.3: A Deus ex Machina clause gadget for the reduction to cycle cover.

1. The sum of the weights of all cycle covers (of the box) that do not use any external edge equals zero.

2. There exists a constant b (in our case $b = 4$) such that, for each of the two anti-parallel edges, the sum of the weights of all cycle covers that use this edge equals b.

3. For any (non-empty) set S of the self-loops, the sum of the weights of all cycle covers (of the box) that use S equals zero.

Note that the foregoing three cases exhaust all the possible ones. It can be shown that the conditions regarding the box imply that the construction presented in Figure 6.4 satisfies the conditions that were postulated for the clause gadget (see Exercise 6.23). Specifically, we have $c = b^5$. As for box itself, a smaller *Deus ex Machina* is provided by the following 4-by-4 adjacency matrix

$$\begin{pmatrix} 0 & 1 & -1 & -1 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 0 & 1 & 3 & 0 \end{pmatrix} \tag{6.4}$$

where the two terminals correspond to the first and the fourth vertices. Its validity (for the value $b = 4$) can be verified by computing the permanent of the corresponding sub-matrices (see Exercise 6.24). ■

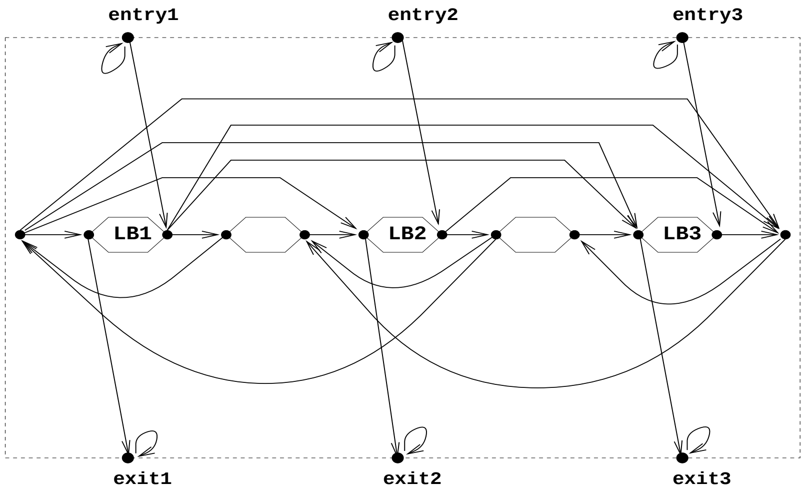
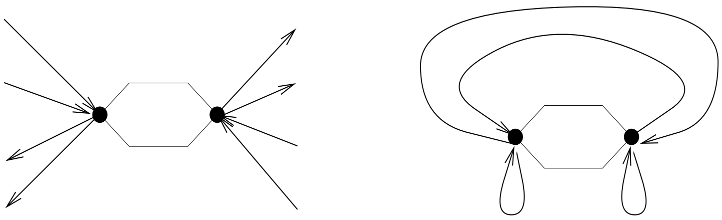


Figure 6.4: A structured clause gadget for the reduction to cycle cover.



On the left is a box with potential edges adjacent to it (as in the gadget construction). On the right is a box and the four external edges used in the analysis.

Figure 6.5: External edges for the analysis of the box

Proof of Proposition 6.20: The proof proceeds in two steps. In the first step we show that computing the permanent of *integer matrices* is reducible to computing the permanent of *non-negative matrices*. This reduction proceeds as follows. For an n-by-n integer matrix $A = (a_{i,j})_{i,j\in[n]}$, let $\|A\|_\infty = \max_{i,j}(|a_{i,j}|)$ and $Q_A = 2(n!)\cdot\|A\|_\infty^n + 1$. We note that, given A, the value Q_A can be computed in polynomial-time, and in particular $\log_2 Q_A < n^2 \log\|A\|_\infty$. Given the matrix A, the reduction constructs the non-negative matrix $A' = (a_{i,j} \bmod Q_A)_{i,j\in[n]}$ (i.e., the entries of A' are in $\{0, 1, ..., Q_A - 1\}$), queries the oracle for the permanent of A', and outputs $v \stackrel{\rm def}{=} \texttt{perm}(A') \bmod Q_A$ if $v < Q_A/2$ and $-(Q_A - v)$ otherwise. The key observation is that

$$\texttt{perm}(A) \equiv \texttt{perm}(A') \pmod{Q_A}, \text{ while } |\texttt{perm}(A)| \le (n!)\cdot\|A\|_\infty^n < Q_A/2.$$

Thus, $\texttt{perm}(A') \bmod Q_A$ (which is in $\{0, 1, ..., Q_A - 1\}$) determines $\texttt{perm}(A)$. We note that $\texttt{perm}(A')$ is likely to be much larger than $Q_A > |\texttt{perm}(A)|$; it is merely that $\texttt{perm}(A')$ and $\texttt{perm}(A)$ are equivalent modulo Q_A.

In the second step we show that computing the permanent of non-negative matrices is reducible to computing the permanent of 0/1-matrices. In this reduction, we view the computation of the permanent as the computation of the sum of the weights of the cycle covers (SWCC) of the corresponding weighted directed graph (see proof of Proposition 6.19). Thus, we reduce the computation of the SWCC of directed graphs *with non-negative weights* to the computation of the SWCC of *unweighted directed graphs with no parallel edges* (which correspond to 0/1-matrices). The reduction is via local replacements that preserve the value of the SWCC. These local replacements combine the following two local replacements (which preserve the SWCC):

1. Replacing an edge of weight $w = \prod_{i=1}^{t} w_i$ by a path of length t (i.e., $t-1$ internal nodes) with the corresponding weights $w_1, ..., w_t$, and self-loops (with weight 1) on all internal nodes.

   Note that a cycle-cover that uses the original edge corresponds to a cycle-cover that uses the entire path, whereas a cycle-cover that does not use the original edge corresponds to a cycle-cover that uses all the self-loops.

2. Replacing an edge of weight $w = \sum_{i=1}^{t} w_i$ by t parallel 2-edge paths such that the first edge on the i^{th} path has weight w_i, the second edge has weight 1, and the intermediate node has a self-loop (with weight 1). (Paths of length two are used because parallel edges are not allowed.)

   Note that a cycle-cover that uses the original edge corresponds to a collection of cycle-covers that use one out of the t paths (and the self-loops of all other intermediate nodes), whereas a cycle-cover that does not use the original edge corresponds to a cycle-cover that uses all the self-loops.

In particular, writing the positive integer w, having binary expansion $\sigma_{|w|-1} \cdots \sigma_0$, as $\sum_{i:\sigma_i=1} (1+1)^i$, we may apply the additive replacement (for the sum over $\{i : \sigma_i = 1\}$), next the product replacement (for each 2^i), and finally the additive replacement (for $1+1$). Applying this process to the matrix A' obtained in the first step, we *efficiently* obtain a matrix A'' with 0/1-entries such that $\texttt{perm}(A') = \texttt{perm}(A'')$. (In particular, the dimension of A'' is polynomial in the length of the binary representation of A', which in turn is polynomial in the length of the binary representation of A.) Combining the two reductions (steps), the proposition follows. ■

6.2.2 Approximate Counting

Let us consider the counting problem associated with an arbitrary $R \in \mathcal{PC}$. Without loss of generality, we assume that all solutions to n-bit instances have the same length $\ell(n)$, where indeed ℓ is a polynomial. We first note that, while it may be hard to compute $\#R$, given x it is easy to approximate $\#R(x)$ up to $0.01 \cdot 2^{\ell(|x|)}$.

Indeed, such an approximation is very rough, but it is not trivial. More generally, we have the following algorithm that produces an estimate of $\#R(x)$ that deviates from the correct value by an additive term that is related to the absolute bound on the number of solutions (i.e., $2^{\ell(|x|)}$).

Proposition 6.21 (approximation with additive deviation): *Let $R \in \mathcal{PC}$ and ℓ be a polynomial such that $R \subseteq \cup_{n\in\mathbb{N}}\{0,1\}^n \times \{0,1\}^{\ell(n)}$. Then, for every polynomial p, there exists a probabilistic polynomial-time algorithm A such that for every $x \in \{0,1\}^*$ and $\delta \in (0,1)$ it holds that*

$$\Pr[|A(x,\delta) - \#R(x)| > (1/p(|x|)) \cdot 2^{\ell(|x|)}] < \delta. \tag{6.5}$$

(As usual, δ is presented to A in binary, and hence the running time of $A(x,\delta)$ is upper-bounded by $\text{poly}(|x| \cdot \log(1/\delta))$.

Proof Sketch: On input x and δ, algorithm A sets $t = \Theta(p(|x|)^2 \cdot \log(1/\delta))$, selects uniformly $y_1, ..., y_t$ and outputs $|\{i : (x, y_i) \in R\}|/t$. □

Discussion. Proposition 6.21 is meaningful in case $\#R(x) > (1/p(|x|)) \cdot 2^{\ell(|x|)}$ holds for some x's. But otherwise, a trivial approximation (i.e., outputting the constant value zero) meets the bound of Eq. (6.5). In general, an approximation of $\#R(x)$ up-to a constant factor (or some other reasonable factor) is more meaningful.[11] In §6.2.2.1, we consider a non-trivial case where such a relative approximation can be obtained in probabilistic polynomial-time. For reasons explained in §6.2.2.1, we do not expect this to happen for every counting problem in $\#\mathcal{P}$, but in §6.2.2.2 we show that relative approximation for any problem in $\#\mathcal{P}$ can be obtained by a randomized Cook-reduction to $\mathcal{NP}$. But before turning to these results, let us state the underlying definition (and actually strengthen it by requiring approximation to within a factor of $1 \pm \varepsilon$).

Definition 6.22 (approximation with relative deviation): *Let $f : \{0,1\}^* \to \mathbb{N}$ and $\varepsilon, \delta : \mathbb{N} \to [0,1]$. A randomized process Π is called an* (ε,δ)-approximator of f *if for every x it holds that*

$$\Pr[|\Pi(x) - f(x)| > \varepsilon(|x|) \cdot f(x)] \;<\; \delta(|x|). \tag{6.6}$$

We say that f is efficiently $(1-\varepsilon)$-approximable (or just $(1-\varepsilon)$-approximable) *if there exists a probabilistic polynomial-time algorithm A that constitute an $(\varepsilon, 1/3)$-approximator of f.*

The error probability of the latter algorithm A (which has error probability $1/3$) can be reduced to δ by $O(\log(1/\delta))$ repetitions (see Exercise 6.25). Typically, the running time of A will be polynomial in $1/\varepsilon$, and ε is called the deviation parameter.

[11] We refrain from formally defining an F-factor approximation in this chapter, although we shall refer to this notion in several informal discussions. There are several ways of defining the aforementioned term (and they are all equivalent when applied to our informal discussions). For example, an F-factor approximation of $\#R$ may mean that, with high probability, the output $A(x)$ satisfies $\#R(x)/F(|x|) \leq A(x) \leq F(|x|) \cdot \#R(x)$. Alternatively, we may require that $\#R(x) \leq A(x) \leq F(|x|) \cdot \#R(x)$ (or, alternatively, that $\#R(x)/F(|x|) \leq A(x) \leq \#R(x)$.

6.2.2.1 Relative approximation for $\#R_{\tt dnf}$

Consider the relation $R_{\tt dnf}$ consisting of pairs (ϕ,τ) such that ϕ is a DNF formula and τ is an assignment satisfying it. Recall that the search problem of $R_{\tt dnf}$ is easy to solve and that the proof of Theorem 6.17 establishes that $\#R_{\tt dnf}$ is $\#\mathcal{P}$-complete (via a non-parsimonious reduction). Still there exists a probabilistic polynomial-time algorithm that provides a constant factor approximation of $\#R_{\tt dnf}$. We warn that the fact that $\#R_{\tt dnf}$ is $\#\mathcal{P}$-complete *via a non-parsimonious reduction* means that the constant factor approximation for $\#R_{\tt dnf}$ does not seem to imply a similar approximation for all problems in $\#\mathcal{P}$. In fact, we should not expect each problem in $\#\mathcal{P}$ to have a (probabilistic) polynomial-time constant-factor approximation algorithm because this would imply $\mathcal{NP}\subseteq\mathcal{BPP}$ (since a constant factor approximation allows for distinguishing the case in which the instance has no solution from the case in which the instance has a solution).

The following algorithm is actually a deterministic reduction of the task of $(\varepsilon,1/3)$-approximating $\#R_{\tt dnf}$ to the (additive deviation) approximation provided in Proposition 6.21. Consider a DNF formula $\phi=\bigvee_{i=1}^m C_i$, where each $C_i:\{0,1\}^n\to\{0,1\}$ is a conjunction. Actually, we will deal with the more general problem in which we are (implicitly) given m subsets $S_1,...,S_m\subseteq\{0,1\}^n$ and wish to approximate $|\bigcup_i S_i|$. In our case, each S_i is the set of assignments satisfying the conjunction C_i. In general, we make two computational assumptions regarding these sets (letting efficient mean implementable in time polynomial in $n\cdot m$):

1. Given $i\in[m]$, one can efficiently determine $|S_i|$.
2. Given $i\in[m]$ and $J\subseteq[m]$, one can efficiently approximate $\mathsf{Pr}_{s\in S_i}\left[s\in\bigcup_{j\in J}S_j\right]$ up to an additive deviation of $1/\mathrm{poly}(n+m)$.

These assumptions are satisfied in our setting (where $S_i=C_i^{-1}(1)$, see Exercise 6.26). The key observation towards approximating $|\bigcup_{i=1}^m S_i|$ is that

$$\left|\bigcup_{i=1}^m S_i\right| = \sum_{i=1}^m\left|S_i\setminus\bigcup_{j<i}S_j\right| = \sum_{i=1}^m |S_i|\cdot\mathsf{Pr}_{s\in S_i}\left[s\notin\bigcup_{j<i}S_j\right] \tag{6.7}$$

and that the probabilities in Eq. (6.7) can be approximated by the second assumption. This leads to the following algorithm, where ε denotes the desired deviation parameter (i.e., we wish to obtain $(1\pm\varepsilon)\cdot|\bigcup_{i=1}^m S_i|$).

Construction 6.23 *Let $\varepsilon'=\varepsilon/m$. For $i=1$ to m do:*

1. *Using the first assumption, compute $|S_i|$.*
2. *Using the second assumption, obtain $\widetilde{p}_i=(1\pm\varepsilon')\cdot p_i$, where $p_i\stackrel{\rm def}{=}\mathsf{Pr}_{s\in S_i}[s\notin\bigcup_{j<i}S_j]$. Set $a_i\stackrel{\rm def}{=}\widetilde{p}_i\cdot|S_i|$.*

Output the sum of the a_i's.

Let $N_i = p_i \cdot |S_i|$. We are interested in the quality of the approximation to $\sum_i N_i = |\bigcup_i S_i|$ provided by $\sum_i a_i$. Using $a_i = (p_i \pm \varepsilon') \cdot |S_i| = N_i \pm \varepsilon' \cdot |S_i|$ (for all i's), we have $\sum_i a_i = \sum_i N_i \pm \varepsilon' \cdot \sum_i |S_i|$. Using $\sum_i |S_i| \leq m \cdot |\bigcup_i S_i| = m \cdot \sum_i N_i$ (and $\varepsilon = m\varepsilon'$), we get $\sum_i a_i = (1 \pm \varepsilon) \cdot \sum_i N_i$. Thus, we obtain the following result (see Exercise 6.26).

Proposition 6.24 *For every positive polynomial p, the counting problem of $R_{\mathtt{dnf}}$ is efficiently $(1-(1/p))$-approximable.*

Using the reduction presented in the proof of Theorem 6.17, we conclude that the number of *un*satisfying assignments to a given CNF formula is efficiently $(1-(1/p))$-approximable. We warn, however, that the number of satisfying assignments to such a formula is *not* efficiently approximable. This concurs with the general phenomenon by which *relative approximation may be possible for one quantity, but not for the complementary quantity.* Needless to say, such a phenomenon does not occur in the context of additive-deviation approximation.

6.2.2.2 Relative approximation for $\#\mathcal{P}$

Recall that we cannot expect to efficiently approximate every $\#\mathcal{P}$ problem. Specifically, efficiently approximating $\#R$ yields an efficient algorithm for deciding membership in $S_R = \{x : R(x) \neq \emptyset\}$. Thus, at best we can hope that approximating $\#R$ is not harder than deciding S_R (i.e., that approximating $\#R$ is reducible in polynomial-time to S_R). This is indeed the case for every NP-complete problem (i.e., if S_R is NP-complete). More generally, we show that approximating any problem in $\#\mathcal{P}$ is reducible in probabilistic polynomial-time to $\mathcal{NP}$.

Theorem 6.25 *For every $R \in \mathcal{PC}$ and positive polynomial p, there exists a probabilistic polynomial-time oracle machine that when given oracle access to $\mathcal{NP}$ constitutes a $(1/p, \mu)$-approximator of $\#R$, where μ is a negligible function* (e.g., $\mu(n) = 2^{-n}$).

Recall that it suffices to provide a $(1/p, \delta)$-approximator of $\#R$, for any constant $\delta < 0.5$, because error reduction is applicable in this context (see Exercise 6.25). Also, it suffices to provide a $(1/2, \delta)$-approximator for every problem in $\#\mathcal{P}$ (see Exercise 6.27).

Proof: Given x, we show how to approximate $|R(x)|$ to within a constant factor. The desired approximation can be obtained as in Exercise 6.27. We may also assume that $R(x) \neq \emptyset$, by starting with the query "is x in S_R" and halting (with output 0) if the answer is negative. Without loss of generality, we assume that $R(x) \subseteq \{0,1\}^\ell$, where $\ell = \mathrm{poly}(|x|)$. Our task is to find some $i \in \{1, ..., \ell\}$ such that $2^{i-4} \leq |R(x)| \leq 2^{i+4}$. We proceed in iterations. For $i = 1, ..., \ell+1$, we find out whether or not $|R(x)| < 2^i$. If the answer is positive then we halt with output 2^i, and otherwise we proceed to the next iteration. (Indeed, if we were able to obtain correct answers to these queries then the output 2^i would satisfy $2^{i-1} \leq |R(x)| < 2^i$.)

Needless to say, the key issue is how to check whether $|R(x)| < 2^i$. The main idea is to use a "random sieve" on the set $R(x)$ such that each element passes the sieve with probability 2^{-i}. Thus, we expect $|R(x)|/2^i$ elements of $R(x)$ to pass the sieve. Assuming that the number of elements in $R(x)$ that pass the random sieve is indeed $\lfloor |R(x)|/2^i \rfloor$, it holds that $|R(x)| \geq 2^i$ if and only if some element of $R(x)$ passes the sieve. Assuming that the sieve can be implemented efficiently, the question of whether or not some element in $R(x)$ passed the sieve is of an "NP-type" (and thus can be referred to our NP-oracle). Combining both assumptions, we may implement the foregoing process by proceeding to the next iteration as long as some element of $R(x)$ passes the sieve. Furthermore, this implementation will provide a reasonably good approximation even if the number of elements in $R(x)$ that pass the random sieve is only approximately equal to $|R(x)|/2^i$. In fact, the level of approximation that this implementation provides is closely related to the level of approximation that is provided by the random sieve. Details follow.

Implementing a random sieve. The random sieve is implemented by using a family of hashing functions (see Section D.2). Specifically, in the i^{th} iteration we use a family H_ℓ^i such that each $h \in H_\ell^i$ has a poly(ℓ)-bit long description and maps ℓ-bit long strings to i-bit long strings. Furthermore, the family is accompanied with an efficient evaluation algorithm (i.e., mapping adequate pairs (h,x) to $h(x)$) and satisfies (for every $S \subseteq \{0,1\}^\ell$)

$$\mathsf{Pr}_{h \in H_\ell^i}[|\{y \in S : h(y) = 0^i\}| \notin (1 \pm \varepsilon) \cdot 2^{-i}|S|] \; < \; \frac{2^i}{\varepsilon^2 |S|} \tag{6.8}$$

(see Lemma D.4). The random sieve will let y pass if and only if $h(y) = 0^i$. Indeed, this random sieve is not as perfect as we assumed in the foregoing discussion, but Eq. (6.8) says that in some sense this sieve is good enough.

Implementing the queries. Recall that for some x, i and $h \in H_\ell^i$, we need to determine whether $\{y \in R(x) : h(y) = 0^i\} = \emptyset$. This type of question can be cast as membership in the set

$$S_{R,H} \stackrel{\text{def}}{=} \{(x,i,h) : \exists y \text{ s.t. } (x,y) \in R \wedge h(y) = 0^i\}. \tag{6.9}$$

Using the hypotheses that $R \in \mathcal{PC}$ and that the family of hashing functions has an efficient evaluation algorithm, it follows that $S_{R,H}$ is in $\mathcal{NP}$.

The actual procedure. On input $x \in S_R$ and oracle access to $S_{R,H}$, we proceed in iterations, starting with $i = 1$ and halting at $i = \ell$ (if not before), where ℓ denotes the length of the potential solutions for x. In the i^{th} iteration (where $i < \ell$), we uniformly select $h \in H_\ell^i$ and query the oracle on whether or not $(x,i,h) \in S_{R,H}$. If the answer is negative then we halt with output 2^i, and otherwise we proceed to the next iteration (using $i \leftarrow i + 1$). Needless to say, if we reach the last iteration (i.e., $i = \ell$) then we just halt with output 2^ℓ.

Indeed, we have ignored the case that $x \notin S_R$, which can be easily handled by a minor modification of the foregoing procedure. Specifically, on input x, we first query S_R on x and halt with output 0 if the answer is negative. Otherwise we proceed as in the foregoing procedure.

The analysis. We upper-bound separately the probability that the procedure outputs a value that is too small and the probability that it outputs a value that is too big. In light of the foregoing discussion, we may assume that $|R(x)| > 0$, and let $i_x = \lfloor \log_2 |R(x)| \rfloor \geq 0$.

1. The probability that the procedure *halts in a specific iteration* $i < i_x$ equals $\mathsf{Pr}_{h\in H_\ell^i}[|\{y \in R(x) : h(y) = 0^i\}| = 0]$, which in turn is upper-bounded by $2^i/|R(x)|$ (using Eq. (6.8) with $\varepsilon = 1$). Thus, the probability that the procedure halts *before* iteration $i_x - 3$ is upper-bounded by $\sum_{i=0}^{i_x-4} 2^i/|R(x)|$, which in turn is less than $1/8$ (because $i_x \leq \log_2 |R(x)|$). Thus, with probability at least $7/8$, the output is at least $2^{i_x-3} > |R(x)|/16$ (because $i_x > (\log_2 |R(x)|) - 1$).

2. The probability that the procedure *does not halt in iteration* $i > i_x$ equals $\mathsf{Pr}_{h\in H_\ell^i}[|\{y \in R(x) : h(y) = 0^i\}| \geq 1]$, which in turn is upper-bounded by $\alpha/(\alpha-1)^2$, where $\alpha = 2^i/|R(x)| > 1$ (using Eq. (6.8) with $\varepsilon = \alpha - 1$).[12] Thus, the probability that the procedure does not halt by iteration $i_x + 4$ is upper-bounded by $8/49 < 1/6$ (because $i_x > (\log_2 |R(x)|) - 1$). Thus, with probability at least $5/6$, the output is at most $2^{i_x+4} \leq 16 \cdot |R(x)|$ (because $i_x \leq \log_2 |R(x)|$).

Thus, with probability at least $(7/8) - (1/6) > 2/3$, the foregoing procedure outputs a value v such that $v/16 \leq |R(x)| < 16v$. Reducing the deviation by using the ideas presented in Exercise 6.27 (and reducing the error probability as in Exercise 6.25), the theorem follows. ■

Perspective. The key observation underlying the proof Theorem 6.25 is that while we cannot test (even with the help of an NP-oracle) whether the number of solutions is greater than a given number, we can test (with the help of an NP-oracle) whether the number of solutions that "survive a random sieve" is greater than zero. If fact, we can also test whether the number of solutions that "survive a random sieve" is greater than a small number, where small means polynomial in the length of the input (see Exercise 6.29). In general, our complexity is linear in the size of the threshold, and not in the length of its binary description. Indeed, in many settings it is more advantageous to use a threshold that is polynomial in some efficiency parameter (rather than using the threshold zero); examples appear in §6.2.4.2 and in [102].

6.2.3 Searching for unique solutions

A natural computational problem (regarding search problems), which arises when discussing the number of solutions, is the problem of distinguishing instances having

[12] A better bound can be obtained by using the hypothesis that, for every y, when h is uniformly selected in H_ℓ^i, the value of $h(y)$ is uniformly distributed in $\{0,1\}^i$. In this case, $\mathsf{Pr}_{h\in H_\ell^i}[|\{y \in R(x) : h(y) = 0^i\}| \geq 1]$ is upper-bounded by $\mathsf{E}_{h\in H_\ell^i}[|\{y \in R(x) : h(y) = 0^i\}|] = |R(x)|/2^i$.

a single solution from instances having no solution (or finding the unique solution whenever such exists). We mention that instances having a single solution facilitate numerous arguments (see, for example, Exercise 6.20 and §10.2.2.1). Formally, searching for and deciding the existence of unique solutions are defined within the framework of promise problems (see Section 2.4.1).

Definition 6.26 (search and decision problems for unique solution instances): *The* set of instances having unique solutions *with respect to the binary relation R is defined as* $\mathtt{US}_R \stackrel{\text{def}}{=} \{x : |R(x)| = 1\}$*, where* $R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\}$*. As usual, we denote* $S_R = \{x : |R(x)| \geq 1\}$*, and* $\overline{S}_R \stackrel{\text{def}}{=} \{0,1\}^* \setminus S_R = \{x : |R(x)| = 0\}$*.*

- *The problem of* finding unique solutions for R *is defined as the* search problem R with promise $\mathtt{US}_R \cup \overline{S}_R$ (see Definition 2.28).

  In continuation to Definition 2.29, the candid searching for unique solutions for R *is defined as the* search problem R with promise $\mathtt{US}_R$.

- *The problem of* deciding unique solution for R *is defined as the* promise problem $(\mathtt{US}_R, \overline{S}_R)$ (see Definition 2.30).

Interestingly, in many natural cases, the promise does not make any of these problems any easier than the original problem. That is, for all known NP-complete problems, the original problem is reducible in probabilistic polynomial-time to the corresponding unique instances problem.

Theorem 6.27 *Let $R \in \mathcal{PC}$ and suppose that every search problem in $\mathcal{PC}$ is parsimoniously reducible to R. Then solving the search problem of R* (resp., deciding membership in S_R) *is reducible in probabilistic polynomial-time to finding unique solutions for R* (resp., to the promise problem $(\mathtt{US}_R, \overline{S}_R)$). *Furthermore, there exists a probabilistic polynomial-time computable mapping M such that for every $x \in \overline{S}_R$ it holds that $M(x) \in \overline{S}_R$, whereas for every $x \in S_R$ it holds that* $\Pr[M(x) \in \mathtt{US}_R] \geq 1/\text{poly}(|x|)$.

We note that the condition regarding parsimonious reductions is crucial (see Exercise 6.30).

Proof: As in the proof of Theorem 6.25, the idea is to apply a "random sieve" on $R(x)$, this time with the hope that a single element survives. Specifically, if we let each element passes the sieve with probability approximately $1/|R(x)|$ then with constant probability a single element survives. Sieving will be performed by a random function selected in an adequate hashing family (see Section D.2). A couple of questions arise:

1. *How do we get an approximation to $|R(x)|$?* Note that we need such an approximation in order to determine the adequate hashing family. Indeed, we may just invoke Theorem 6.25, but this will not yield a many-to-one reduction. Instead, we just select $m \in \{0, ..., \text{poly}(|x|)\}$ uniformly and note that (if $|R(x)| > 0$ then) $\Pr[m = \lceil \log_2 |R(x)| \rceil] = 1/\text{poly}(|x|)$.

   Thus, we randomly map x to (x, m, h), where h is uniformly selected in an adequate hashing family.

2. *How does the question of whether a single element of $R(x)$ pass the random sieve translate to an instance of the unique-instance problem for R?* Recall that in the proof of Theorem 6.25 the non-emptiness of the set of element of $R(x)$ that pass the sieve defined by h was determined by checking membership (of (x,m,h)) in $S_{R,H} \in \mathcal{NP}$ (defined in Eq. (6.9)). Furthermore, the number of NP-witnesses for $(x,m,h) \in S_{R,H}$ equals the number of elements of $R(x)$ that pass the sieve. Using the parsimonious reduction of $S_{R,H}$ to S_R (which is guaranteed by the theorem's hypothesis), we obtained the desired instance.

Note that in case $R(x) = \emptyset$ the aforementioned mapping always generates a no-instance (of $S_{R,H}$ and thus of S_R). Details follow.

Implementation (i.e., the mapping M). As in the proof of Theorem 6.25, we assume, without loss of generality, that $R(x) \subseteq \{0,1\}^\ell$, where $\ell = \text{poly}(|x|)$. We start by uniformly selecting $m \in \{0,1,...,\ell\}$ and $h \in H_\ell^m$, where H_ℓ^m is a family of efficiently computable and pairwise-independent hashing functions (see Definition D.1) mapping ℓ-bit long strings to m-bit long strings.[13] Thus, we obtain an instance (x,m,h) of $S_{R,H} \in \mathcal{NP}$ such that the set of valid solutions for (x,m,h) equals $\{y \in R(x) : h(y) = 0^m\}$. Using the parsimonious reduction g of $S_{R,H}$ to S_R, we map (x,m,h) to $g(x,m,h)$, and it holds that $|\{y \in R(x) : h(y) = 0^m\}|$ equals $|R(g(x,m,h))|$. To summarize, on input x the randomized mapping M outputs the instance $M(x) \stackrel{\text{def}}{=} g(x,m,h)$, where $m \in \{0,1,...,\ell\}$ and $h \in H_\ell^m$ are uniformly selected.

The analysis. Note that for any $x \in \overline{S}_R$ it holds that $\Pr[M(x) \in \overline{S}_R] = 1$. Assuming that $x \in S_R$, with probability exactly $1/(\ell+1)$ it holds that $m = m_x$, where $m_x \stackrel{\text{def}}{=} \lceil \log_2 |R(x)| \rceil$. In this case, for a uniformly selected $h \in H_\ell^m$, we lower-bound the probability that $\{y \in R(x) : h(y) = 0^m\}$ is a singleton. Using the Inclusion-Exclusion Principle, we have

$$\begin{aligned} & \Pr_{h \in H_\ell^{m_x}}[|\{y \in R(x) : h(y) = 0^{m_x}\}| = 1] \qquad (6.10) \\ \geq & \sum_{y \in R(x)} \Pr_{h \in H_\ell^{m_x}}[h(y) = 0^{m_x}] \\ & - \sum_{y_1 < y_2 \in R(x)} \Pr_{h \in H_\ell^{m_x}}[h(y_1) = h(y_2) = 0^{m_x}] \\ = & |R(x)| \cdot 2^{-m_x} - \binom{|R(x)|}{2} \cdot 2^{-2m_x} \end{aligned}$$

where the equality is due to the pairwise independence property. Using $2^{m_x - 1} < |R(x)| \leq 2^{m_x}$, it follows that Eq. (6.10) is lower-bounded by 1/4. Thus, $\Pr[M(x) \in \text{US}_R] \geq 1/4(\ell+1)$, and the theorem follows. ■

Comment. Theorem 6.27 is sometimes stated as referring to the unique solution problem of SAT. In this case and when using a specific family of pairwise indepen-

[13]For sake of uniformity, we allow also the case of $m = 0$, which is rather artificial. In this case all hashing functions in H_ℓ^0 map $\{0,1\}^\ell$ to the empty string, which is viewed as 0^0.

dent hashing functions, the use of the parsimonious reduction can be avoided. For details see Exercise 6.31.

6.2.4 Uniform generation of solutions

We now turn to a new type of computational problems, which may be viewed as a straining of search problems. We refer to the task of generating a uniformly distributed solution for a given instance, rather than merely finding an adequate solution. Needless to say, by definition, algorithms solving this ("uniform generation") task must be randomized. Focusing on relations in $\mathcal{PC}$ we consider two versions of the problem, which differ by the level of approximation provided for the desired (uniform) distribution.[14]

Definition 6.28 (uniform generation): *Let $R \in \mathcal{PC}$ and $S_R = \{x : |R(x)| \geq 1\}$, and let Π be a probabilistic process.*

1. *We say that Π* solves the uniform generation problem of *R if, on input $x \in S_R$, the process Π outputs either an element of $R(x)$ or a special symbol, denoted $\perp$, such that $\Pr[\Pi(x) \in R(x)] \geq 1/2$ and for every $y \in R(x)$ it holds that $\Pr[\Pi(x)=y \mid \Pi(x)\in R(x)] = 1/|R(x)|$.*
2. *For $\varepsilon : \mathbb{N} \to [0,1]$, we say that Π* solves the $(1-\varepsilon)$-approximate uniform generation problem of *R if, on input $x \in S_R$, the distribution $\Pi(x)$ is $\varepsilon(|x|)$-close*[15] *to the uniform distribution on $R(x)$.*

In both cases, without loss of generality, we may require that if $x \notin S_R$ then $\Pr[\Pi(x) = \perp] = 1$. More generally, we may require that Π never outputs a string not in $R(x)$.

Note that the error probability of uniform generation (as in Item 1) can be made exponentially vanishing (in $|x|$) by employing error-reduction. In contrast, we are not aware of any general way of reducing the deviation of an approximate uniform generation procedure (as in Item 2).[16]

In §6.2.4.1 we show that, for many search problems, approximate uniform generation is computationally equivalent to approximate counting. In §6.2.4.2 we present a direct approach for solving the uniform generation problem of any search problem in $\mathcal{PC}$ by using an oracle to $\mathcal{NP}$.

6.2.4.1 Relation to approximate counting

We show that for every $R \in \mathcal{PC}$ that is NP-complete under parsimonious reductions, the approximate counting problem associated with R is computationally

[14] Note that a probabilistic algorithm running in strict polynomial-time is not able to output a perfectly uniform distribution on sets of certain sizes. Specifically, referring to the standard model that allows only for uniformly selected binary values, such algorithms cannot output a perfectly uniform distribution on sets having cardinality that is not a power of two.

[15] See Section D.1.1.

[16] We note that in some cases, the deviation of an approximate uniform generation procedure can be reduced. See discussion following Theorem 6.29.

equivalent to approximate uniform generation with respect to R. Recalling that both approximate problems are parameterized by the level of precision, we obtain the following quantitative form of the aforementioned equivalence.

Theorem 6.29 *Let $R \in \mathcal{PC}$ and let ℓ be a polynomial such that for every $(x,y) \in R$ it holds that $|y| \leq \ell(|x|)$. Suppose that every search problem in $\mathcal{PC}$ is parsimoniously reducible to R.*

1. From approximate counting to approximate uniform generation: *Let $\varepsilon(n) = 1/5\ell(n)$ and let $\mu : \mathbb{N} \to (0,1)$ be a function satisfying $\mu(n) \geq \exp(-\text{poly}(n))$. Then, $(1-\mu)$-approximate uniform generation for R is reducible in probabilistic polynomial-time to $(1-\varepsilon)$-approximating $\#R$.*

2. From approximate uniform generation to approximate counting: *For every noticeable $\varepsilon : \mathbb{N} \to (0,1)$* (i.e., $\varepsilon(n) \geq 1/\text{poly}(n)$ for every n)*, the problem of $(1-\varepsilon)$-approximating $\#R$ is reducible in probabilistic polynomial-time to $(1-\varepsilon')$-approximate uniform generation problem of R, where $\varepsilon'(n) = \varepsilon(n)/5\ell(n)$.*

Note that the quality of the approximate uniform generation asserted in Part 1 (i.e., μ) is independent of the quality of the approximate counting procedure (i.e., ε) to which the former is reduced, provided that the approximate counter performs better than some threshold. On the other hand, the quality of the approximate counting asserted in Part 2 (i.e., ε) does depend on the quality of the approximate uniform generation (i.e., ε'). Recall, however, that the quality of approximate counting procedures for problems that are NP-complete under parsimonious reductions can be improved (see Exercise 6.28). Thus, for such problems, the quality of approximate uniform generation procedures can be improved by applying both parts of Theorem 6.29.

Proof: Throughout the proof, we assume for simplicity (and in fact without loss of generality) that $R(x) \neq \emptyset$ and $R(x) \subseteq \{0,1\}^{\ell(|x|)}$.

Towards Part 1, let use first reduce the uniform generation problem of R to $\#R$ (rather than to approximating $\#R$). On input $x \in S_R$, we generate a uniformly distributed $y \in R(x)$ by randomly generating its bits one after the other. We proceed in iterations, entering the i^{th} iteration with an $(i-1)$-bit long string y' such that $R'(x;y') \stackrel{\text{def}}{=} \{y'' : (x, y'y'') \in R\}$ is not empty. With probability $|R'(x;y'1)|/|R'(x;y')|$ we set the i^{th} bit to equal 1, and otherwise we set it to equal 0. We obtain both $|R'(x;y'1)|$ and $|R'(x;y')|$ by using a parsimonious reduction g of $R' = \{((x;y'),y'') : (x,y'y'') \in R\} \in \mathcal{PC}$ to R. That is, we obtain $|R'(x;y')|$ by querying for the value of $|R(g(x;y'))|$. Ignoring integrality issues, all this works perfectly (i.e., we generate an $\ell(n)$-bit string uniformly distributed in $R(x)$) as long as we have oracle access to $\#R$. But we only have oracle access to an approximation of $\#R$, and thus a careful modification is in place.

Let us denote the approximation oracle by A. Firstly, by adequate error reduction, we may assume that, for every x, it holds that $\Pr[A(x) \in (1 \pm \varepsilon(n)) \cdot \#R(x)] > 1 - \mu'(|x|)$, where $\mu'(n) = \mu(n)/\ell(n)$. In the rest of the analysis we ignore the probability that the estimate deviates from the aforementioned interval, and note that this rare event is the only source of the possible deviation of the output distribution from the uniform distribution on $R(x)$.[17] Let us assume for a moment that A is *deterministic* and that for every x and y' it holds that

$$A(g(x,y'0)) + A(g(x,y'1)) \;\le\; A(g(x;y')). \tag{6.11}$$

We also assume that the approximation is correct at the "trivial level" (where one may just check whether or not (x,y) is in R); that is, for every $y \in \{0,1\}^{\ell(|x|)}$, it holds that

$$A(g(x;y)) = 1 \text{ if } (x,y) \in R \text{ and } A(g(x;y)) = 0 \text{ otherwise.} \tag{6.12}$$

We modify the $i^{\rm th}$ iteration of the foregoing procedure such that, when entering with the $(i-1)$-bit long prefix y', we set the $i^{\rm th}$ bit to 1 (resp., to 0) with probability $A(g(x;y'1))/A(g(x;y'))$ (resp., with probability $A(g(x;y'0))/A(g(x;y'))$) and halt (with output $\perp$) with the residual probability. If we completed the last (i.e., $\ell(|x|)^{\rm th}$) iteration, then we output the $\ell(|x|)$-bit long string that was generated. Thus, as long as Eq. (6.11) holds (but regardless of other aspects of the quality of the approximation), every $y = \sigma_1 \cdots \sigma_{\ell(|x|)} \in R(x)$, is output with probability

$$\frac{A(g(x;\sigma_1))}{A(g(x;\lambda))} \cdot \frac{A(g(x;\sigma_1\sigma_2))}{A(g(x;\sigma_1))} \cdots \frac{A(g(x;\sigma_1\sigma_2\cdots\sigma_{\ell(|x|)}))}{A(g(x;\sigma_1\sigma_2\cdots\sigma_{\ell(|x|)-1}))} \tag{6.13}$$

which, by Eq. (6.12), equals $1/A(g(x;\lambda))$. Thus, the procedure outputs each element of $R(x)$ with equal probability, and never outputs a non-$\perp$ value that is outside $R(x)$. It follows that the quality of approximation only effects the probability that the procedure outputs a non-$\perp$ value (which equals $|R(x)|/A(g(x;\lambda))$).

We now turn to enforcing Eq. (6.11) and Eq. (6.12). We may enforce Eq. (6.12) by performing the straightforward check (of whether or not $(x,y) \in R$) rather than invoking $A(g(x,y))$.[18] As for Eq. (6.11), we enforce it artificially by using $A'(x,y') \stackrel{\rm def}{=} (1+\varepsilon(|x|))^{3(\ell(|x|)-|y'|)} \cdot A(g(x;y'))$ instead of $A(g(x;y'))$. Recalling that $A(g(x;y')) = (1 \pm \varepsilon(|xy'|)) \cdot |R'(x;y')|$, we have

$$\begin{aligned} A'(x,y') &> (1+\varepsilon(|x|))^{3(\ell(|x|)-|y'|)} \cdot (1-\varepsilon(|x|)) \cdot |R'(x;y')| \\ A'(x,y'\sigma) &< (1+\varepsilon(|x|))^{3(\ell(|x|)-|y'|-1)} \cdot (1+\varepsilon(|x|)) \cdot |R'(x;y'\sigma)| \end{aligned}$$

and the claim follows using $(1-\varepsilon(|x|)) \cdot (1+\varepsilon(|x|))^3 > (1-\varepsilon(|x|))$. Note that the foregoing modification only decreases the probability of outputting a non-$\perp$ value by a factor of $(1+\varepsilon(|x|))^{3\ell(|x|)} < 2$, where the inequality is due to the setting of ε (i.e., $\varepsilon(n) = 1/5\ell(n)$). Finally, we refer to our assumption that A is deterministic. This assumption was only used in order to identify the value of $A(g(x,y'))$ obtained and used in the $(|y'|-1)^{\rm st}$ iteration with the value of $A(g(x,y'))$ obtained and used

[17] The possible deviation is due to the fact that this rare event may occur with different probability in the different invocations of algorithm A.

[18] Alternatively, we note that since A is a $(1-\varepsilon)$-approximator for $\varepsilon < 1$ it must hold that $\#R'(z) = 0$ implies $A(z) = 0$. Also, since $\varepsilon < 1/3$, if $\#R'(z) = 1$ then $A(z) \in (2/3, 4/3)$, which may be rounded to 1.

in the $|y'|^{\rm th}$ iteration, but the same effect can be obtained by just using the former value (in the $|y'|^{\rm th}$ iteration) rather than re-invoking A in order to obtain it. Part 1 follows.

Towards Part 2, let use first reduce the task of approximating $\#R$ to the task of (exact) uniform generation for R. On input $x \in S_R$, the reduction uses the tree of possible prefixes of elements of $R(x)$ in a somewhat different manner. Again, we proceed in iterations, entering the $i^{\rm th}$ iteration with an $(i-1)$-bit long string y' such that $R'(x;y') \stackrel{\rm def}{=} \{y'' : (x, y'y'') \in R\}$ is not empty. At the $i^{\rm th}$ iteration we estimate the bigger among the two fractions $|R'(x;y'0)|/|R'(x;y')|$ and $|R'(x;y'1)|/|R'(x;y')|$, by uniformly sampling the uniform distribution over $R'(x;y')$. That is, taking ${\rm poly}(|x|/\varepsilon'(|x|))$ uniformly distributed samples in $R'(x;y')$, we obtain with overwhelmingly high probability an approximation of these fractions up to an additive deviation of at most $\varepsilon'(|x|)/3$. This means that we obtain a relative approximation up-to a factor of $1 \pm \varepsilon'(|x|)$ for the fraction (or fractions) that is (resp., are) bigger than $1/3$. Indeed, we may not be able to obtain such a good relative approximation of the other fraction (in case it is very small), but this does not matter. It also does not matter that we cannot tell which is the bigger fraction among the two; it only matter that we use an approximation that indicates a quantity that is, say, bigger than $1/3$. We proceed to the next iteration by augmenting y' using the bit that corresponds to such a quantity. Specifically, suppose that we obtained the approximations $a_0(y') \approx |R'(x;y'0)|/|R'(x;y')|$ and $a_1(y') \approx |R'(x;y'1)|/|R'(x;y')|$. Then we extend y' by the bit 1 if $a_1(y') > a_0(y')$ and extend y' by the bit 0 otherwise. Finally, when we reach $y = \sigma_1 \cdots \sigma_{\ell(|x|)}$ such that $(x, y) \in R$, we output

$$a_{\sigma_1}(\lambda)^{-1} \cdot a_{\sigma_2}(\sigma_1)^{-1} \cdots a_{\sigma_{\ell(|x|)}}(\sigma_1\sigma_2 \cdots \sigma_{\ell(|x|)-1})^{-1}. \tag{6.14}$$

As in Part 1, actions regarding R' (in this case uniform generation in R') are conducted via the parsimonious reduction g to R. That is, whenever we need to sample uniformly in the set $R'(x;y')$, we sample the set $R(g(x;y'))$ instead. Finally, note that the deviation from uniform distribution (i.e., the fact that we can only approximately sample R) merely introduces such a deviation in each of our approximations to the relevant fractions (i.e., to a fraction bigger than $1/3$). Specifically, on input x, using an oracle that provides a $(1-\varepsilon')$-approximate uniform generation for R, with overwhelmingly high probability, the output (as defined in Eq. (6.14)) is in

$$\prod_{i=1}^{\ell(|x|)} \left((1 \pm 2\varepsilon'(|x|)) \cdot \frac{|R'(x;\sigma_1 \cdots \sigma_{i-1})|}{|R'(x;\sigma_1 \cdots \sigma_i)|} \right) \tag{6.15}$$

where the error probability is due to the unlikely case that in one of the iterations our approximations deviates from the correct value by more than an additive deviation term of $\varepsilon'(n)/3$. Noting that Eq. (6.15) equals $(1 \pm 2\varepsilon'(|x|))^{\ell(|x|)} \cdot |R(x)|$ and using $(1 \pm 2\varepsilon'(|x|))^{\ell(|x|)} \subset (1 \pm \varepsilon(|x|))$, Part 2 follows, and so does the theorem. ■

6.2.4.2 Direct uniform generation

We conclude the current chapter by presenting a direct procedure for solving the uniform generation problem of any $R \in \mathcal{PC}$. This procedure uses an oracle to $\mathcal{NP}$, which is unavoidable because solving the uniform generation problem implies solving the corresponding search problem. One advantage of this process, over the reduction presented in §6.2.4.1, is that it solves the uniform generation problem rather than the *approximate* uniform generation problem.

We are going to use hashing again, but this time we use a family of hashing functions having a stronger "uniformity property" (see Section D.2.3). Specifically, we will use a family of ℓ-wise independent hashing functions mapping ℓ-bit strings to m-bit strings, where ℓ bounds the length of solutions in R, and rely on the fact that such a family satisfies Lemma D.6. Intuitively, such functions partition $\{0,1\}^\ell$ into 2^m cells and Lemma D.6 asserts that these partitions "uniformly shatter" all sufficiently large sets. That is, for every set $S \subseteq \{0,1\}^\ell$ of size $\Omega(\ell \cdot 2^m)$ the partition induced by almost every function is such that each cell contains approximately $|S|/2^m$ elements of S. In particular, if $|S| = \Theta(\ell \cdot 2^m)$ then each cell contains $\Theta(\ell)$ elements of S.

In the following construction, we assume that on input x we also obtain a good approximation to the size of $R(x)$. This assumption can be enforced by using an approximate counting procedure as a preprocessing stage. Alternatively, the ideas presented in the following construction yield such an approximate counting procedure.

Construction 6.30 (uniform generation): *On input x and $m'_x \in \{m_x, m_x+1\}$, where $m_x \stackrel{\text{def}}{=} \lfloor \log_2 |R(x)| \rfloor$ and $R(x) \subseteq \{0,1\}^\ell$, the oracle machine proceeds as follows.*

1. Selecting a partition that "uniformly shatters" $R(x)$. *The machine sets $m = \max(0, m'_x - 6 - \log_2 \ell)$ and selects uniformly $h \in H_\ell^m$. Such a function defines a partition of $\{0,1\}^\ell$ into 2^m cells*[19]*, and the hope is that each cell contains approximately the same elements of $R(x)$. Next, the machine checks that this is indeed the case or rather than no cell contains more that 1000ℓ elements of $R(x)$. This is done by checking whether or not $(x, h, 1^{1000\ell})$ is in the set $S^{(1)}_{R,H}$ defined as follows*

$$\begin{aligned} S^{(1)}_{R,H} &\stackrel{\text{def}}{=} \{(x', h', 1^t) : \exists v \text{ s.t. } |\{y : (x',y) \in R \wedge h'(y) = v\}| \geq t\} \quad (6.16) \\ &= \{(x', h', 1^t) : \exists v, y_1, ..., y_t \text{ s.t. } \psi^{(1)}(x', h', v, y_1, ..., y_t)\}, \end{aligned}$$

where $\psi^{(1)}(x', h', v, y_1, ..., y_t)$ holds if and only if $y_1 < y_2 \cdots < y_t$ and for every $j \in [t]$ it holds that $(x', y_j) \in R \wedge h'(y_j) = v$. Note that $S^{(1)}_{R,H} \in \mathcal{NP}$.

If the answer is positive (i.e., there exists a cell that contains more that 1000ℓ elements of $R(x)$) *then the machine halts with output $\perp$. Otherwise,*

[19] For sake of uniformity, we allow also the case of $m = 0$, which is rather artificial. In this case all hashing functions in H_ℓ^0 map $\{0,1\}^\ell$ to the empty string, which is viewed as 0^0, and thus define a trivial partition of $\{0,1\}^\ell$ (i.e., into a single cell).

the machine continues with this choice of h. *In this case, for every* $v \in \{0,1\}^m$, *it holds that no cell contains more that* 1000ℓ *elements of* $R(x)$ (i.e., $|\{y : (x,y) \in R \wedge h(y) = v\}| < 1000\ell$). *We stress that this is an absolute guarantee that follows from* $(x, h, 1^{1000\ell}) \notin S^{(1)}_{R,H}$.

2. Selecting a cell and determining the number of elements of $R(x)$ that are contained in it. *The machine selects uniformly* $v \in \{0,1\}^m$ *and determines* $s_v \stackrel{\text{def}}{=} |\{y : (x,y) \in R \wedge h(y) = v\}|$ *by making queries to the following NP-set*

$$S^{(2)}_{R,H} \stackrel{\text{def}}{=} \{(x', h', v', 1^t) : \exists y_1, ..., y_t \text{ s.t. } \psi^{(1)}(x', h', v', y_1, ..., y_t)\}. \quad (6.17)$$

Specifically, for $i = 1, ..., 1000\ell$, *it checks whether* $(x, h, v, 1^i)$ *is in* $S^{(2)}_{R,H}$, *and sets* s_v *to be the largest value of* i *for which the answer is positive.*

3. Obtaining all the elements of $R(x)$ that are contained in the selected cell, and outputting one of them at random. *Using* s_v, *the procedure reconstructs the set* $S_v \stackrel{\text{def}}{=} \{y : (x,y) \in R \wedge h(y) = v\}$, *by making queries to the following NP-set*

$$S^{(3)}_{R,H} \stackrel{\text{def}}{=} \{(x', h', v', 1^t, j) : \exists y_1, ..., y_t \text{ s.t. } \psi^{(3)}(x', h', v', y_1, ..., y_t, j)\}, \quad (6.18)$$

where $\psi^{(3)}(x', h', v', y_1, ..., y_t, j)$ *holds if and only if* $\psi^{(1)}(x', h', v', y_1, ..., y_t)$ *holds and the* j^{th} *bit of* $y_1 \cdots y_t$ *equals* 1. *Specifically, for* $j_1 = 1, ..., s_v$ *and* $j_2 = 1, ..., \ell$, *we make the query* $(x, h, v, 1^{s_v}, (j_1 - 1) \cdot \ell + j_2)$ *in order to determine the* j_2^{th} *bit of* y_{j_1}. *Finally, having recovered* S_v, *the procedure outputs each* $y \in S_v$ *with probability* $1/1000\ell$, *and outputs* $\perp$ *otherwise.*

By Lemma D.6 (and $m \leq m_x + 1 - 6 - \log_2 \ell$), with overwhelmingly high probability, each set $\{y : (x,y) \in R \wedge h(y) = v\}$ has cardinality $(1 \pm 0.5)|R(x)|/2^m$. Using $m'_x > (\log_2 |R(x)|) - 1$ (resp., $m'_x \leq (\log_2 |R(x)|) + 1$), it follows that $|R(x)|/2^m < 1000\ell$ (resp., $|R(x)|/2^m > 10\ell$). Thus, Step 1 can be easily adapted to yield an approximate counting procedure for $\#R$ (see Exercise 6.32). However, our aim is to establish the following fact.

Proposition 6.31 *Construction 6.30 solves the uniform generation problem of* R.

Proof: By Lemma D.6 (and the setting of m), with overwhelmingly high probability, a uniformly selected $h \in H^m_\ell$ partitions $R(x)$ into 2^m cells, each containing at most 1000ℓ elements. The key observation, stated in Step 1, is that if the procedure does not halt in Step 1 then it is indeed the case that h induces such a partition. The fact that these cells may contain a different number of elements is immaterial, because each element is output with the same probability (i.e., $1/1000\ell$). What matters is that the average number of elements in the cells is sufficiently large, because this average number determines the probability that the procedure outputs an element of $R(x)$ (rather than $\perp$). Specifically, the latter probability equals the aforementioned average number (which equals $|R(x)|/2^m$) divided by 1000ℓ. Using $m \leq \max(0, \log_2(2|R(x)|) - 6 - \log_2 \ell)$, we have $|R(x)|/2^m \geq \max(1, 32\ell)$, which means that the procedure outputs some element of $R(x)$ with probability at least $1/1000\ell$. ■

Technical comments. We can easily improve the performance of Construction 6.30 by dealing separately with the case $m = 0$. In such a case, Step 3 can be simplified and improved by uniformly selecting and outputting an element of S_λ (which equals $R(x)$). Recall that the probability that a uniform generation procedure outputs $\perp$ can be deceased by repeated invocations.

Chapter Notes

One key aspect of randomized procedures is their success probability, which is obviously a quantitative notion. This aspect provides a clear connection between probabilistic polynomial-time algorithms considered in Section 6.1 and the counting problems considered in Section 6.2 (see also Exercise 6.17). More appealing connections between randomized procedures and counting problems (e.g., the application of randomization in approximate counting) are presented in Section 6.2. These connections justify the presentation of these two topics in the same chapter.

Randomized algorithms

Making people take an unconventional step requires compelling reasons, and indeed the study of randomized algorithms was motivated by a few compelling examples. Ironically, the appeal of the two most famous examples (discussed next) has diminished due to subsequent finding, but the fundamental questions that emerged remain fascinating regardless of the status of these and other appealing examples (see §6.1.1.1).

The first example: primality testing. For more than two decades, primality testing was the archetypical example of the usefulness of randomization in the context of efficient algorithms. The celebrated algorithms of Solovay and Strassen [197] and of Rabin [171], proposed in the late 1970's, established that deciding primality is in co$\mathcal{RP}$ (i.e., these tests always recognize correctly prime numbers, but they may err on composite inputs). (The approach of Construction 6.4, which only establishes that deciding primality is in $\mathcal{BPP}$, is commonly attributed to M. Blum.) In the late 1980's, Adleman and Huang [2] proved that deciding primality is in $\mathcal{RP}$ (and thus in $\mathcal{ZPP}$). In the early 2000's, Agrawal, Kayal, and Saxena [3] showed that deciding primality is actually in $\mathcal{P}$. One should note, however, that strong evidence to the fact that deciding primality is in $\mathcal{P}$ was actually available from the start: we refer to Miller's deterministic algorithm [154], which relies on the Extended Riemann Hypothesis.

The second example: undirected connectivity. Another celebrated example to the power of randomization, specifically in the context of log-space computations, was provided by testing undirected connectivity. The random-walk algorithm presented in Construction 6.10 is due to Aleliunas, Karp, Lipton, Lovász, and Rackoff [5]. Recall that a deterministic log-space algorithm was found twenty-five years later (see Section 5.2.4 or [177]).

Other randomized algorithms. Although randomized algorithms are more abundant in the context of approximation problems (let alone in other computational settings (cf. §6.1.1.1)), quite a few such algorithms are known also in the context of search and decision problems. We mention the algorithms for finding perfect matchings and minimum cuts in graphs (see, e.g., [86, Apdx. B.1] or [156, Sec. 12.4&10.2]), and note the prominent role of randomization in computational number theory (see, e.g., [21] or [156, Chap. 14]). For a general textbook on randomized algorithms, we refer the interested reader to [156].

On the general study of $\mathcal{BPP}$. Turning to the general study of $\mathcal{BPP}$, we note that our presentation of Theorem 6.7 follows the proof idea of Lautemann [140]. A different proof technique, which yields a weaker result but found more applications (see, e.g., Theorem 6.25 and [107]), was presented (independently) by Sipser [193].

On the role of promise problems. In addition to their use in the formulation of Theorem 6.7, promise problems allow for establishing time hierarchy theorems (as in §4.2.1.1) for randomized computation (see Exercise 6.13). We mention that such results are not known for the corresponding classes of standard decision problems. The technical difficulty is that we do not know how to enumerate probabilistic machines that utilize a non-trivial probabilistic decision rule.

On the feasibility of randomized computation. Different perspectives on this question are offered by Chapter 8 and Section D.4. Specifically, as advocated in Chapter 8, generating uniformly distributed bit sequences is not really necessary for implementing randomized algorithms; it suffices to generate sequences that look as if they are uniformly distributed. A less radical approach is presented in Section D.4, which deals with the task of extracting almost uniformly distributed bit sequences from sources of weak randomness. Needless to say, these two approaches are complimentary and can be combined.

Counting problems

The counting class $\#\mathcal{P}$ was introduced by Valiant [213], who proved that computing the permanent of 0/1-matrices is $\#\mathcal{P}$-complete (i.e., Theorem 6.18). Interestingly, like in the case of Cook's introduction of NP-completeness [55], Valiant's motivation was determining the complexity of a specific problem (i.e., the permanent).

Our presentation of Theorem 6.18 is based both on Valiant's paper [213] and on subsequent studies (most notably [29]). Specifically, the high-level structure of the reduction presented in Proposition 6.19 as well as the "structured" design of the clause gadget is taken from [213], whereas the Deus Ex Machina gadget presented in Figure 6.3 is based on [29]. The proof of Proposition 6.20 is also based on [29] (with some variants). Turning back to the design of clause gadgets we regret not being able to cite and/or use a systematic study of this design problem.

As noted in the main text, we decided not to present a proof of Toda's Theorem [206], which asserts that every set in $\mathcal{PH}$ is Cook-reducible to $\#\mathcal{P}$ (i.e., Theorem 6.14). A proof of a related result appears in Section F.1 (implying that $\mathcal{PH}$ is reducible to $\#\mathcal{P}$ via probabilistic polynomial-time reductions). Alternative proofs can be found in [126, 198, 206].

Approximate counting and related problems. The approximation procedure for $\#\mathcal{P}$ is due to Stockmeyer [200], following an idea of Sipser [193]. Our exposition, however, follows further developments in the area. The randomized reduction of $\mathcal{NP}$ to problems of unique solutions was discovered by Valiant and Vazirani [215]. Again, our exposition is a bit different.

The connection between approximate counting and uniform generation (presented in §6.2.4.1) was discovered by Jerrum, Valiant, and Vazirani [124], and is applicable also beyond the setting of Theorem 6.29 (e.g., in the "Markov Chain approach" (see [156, Sec. 11.3.1])). The direct solution to uniform generation (presented in §6.2.4.2) is taken from [26].

In continuation to §6.2.2.1, which is based on [129], we refer the interested reader to [123], which presents a probabilistic polynomial-time algorithm for approximating the permanent of non-negative matrices. This fascinating algorithm is based on the fact that knowing (approximately) certain parameters of a non-negative matrix M allows to approximate the same parameters for a matrix M', provided that M and M' are sufficiently similar. Specifically, M and M' may differ only on a single entry, and the ratio of the corresponding values must be sufficiently close to one. Needless to say, the actual observation (is not generic but rather) refers to specific parameters of the matrix, which include its permanent. Thus, given a matrix M for which we need to approximate the permanent, we consider a sequence of matrices $M_0, ..., M_t \approx M$ such that M_0 is the all 1's matrix (for which it is easy to evaluate the said parameters), and each M_{i+1} is obtained from M_i by reducing some adequate entry by a factor sufficiently close to one. This process of (polynomially many) gradual changes, allows to transform the dummy matrix M_0 into a matrix M_t that is very close to M (and hence has a permanent that is very close to the permanent of M). Thus, approximately obtaining the parameters of M_t allows to approximate the permanent of M.

Exercises

Exercise 6.1 Show that if a search (resp., decision) problem can be solved by a probabilistic polynomial-time algorithm having zero error probability, then the problem can be solve by a deterministic polynomial-time algorithm.
(Hint: replace the internal coin tosses by a fixed outcome.)

Exercise 6.2 (randomized reductions) In continuation to the definitions presented at the beginning of Section 6.1, prove the following:

1. If a problem Π is probabilistic polynomial-time reducible to a problem that is solvable in probabilistic polynomial-time then Π is solvable in probabilistic

polynomial-time, where by solving we mean solving correctly except with negligible probability.

Warning: Recall that in the case that Π' is a search problem, we required that on input x the solver provides a correct solution with probability at least $1-\mu(|x|)$, but we did not require that it always returns the same solution.

(Hint: without loss of generality, the reduction does not make the same query twice.)

2. Prove that probabilistic polynomial-time reductions are transitive.

3. Prove that randomized Karp-reductions are transitive and that they yield a special case of probabilistic polynomial-time reductions.

Define one-sided error and zero-sided error randomized (Karp and Cook) reductions, and consider the foregoing items when applied to them. Note that the implications for the case of one-sided error are somewhat subtle.

Exercise 6.3 (on the definition of probabilistically solving a search problem) In continuation to the discussion at the beginning of Section 6.1.1, suppose that for some probabilistic polynomial-time algorithm A and a positive polynomial p the following holds: for every $x \in S_R \stackrel{\text{def}}{=} \{z : R(z) \neq \emptyset\}$ there exists $y \in R(x)$ such that $\Pr[A(x)=y] > 0.5 + (1/p(|x|))$, whereas for every $x \notin S_R$ it holds that $\Pr[A(x)=\perp] > 0.5 + (1/p(|x|))$.

1. Show that there exists a probabilistic polynomial-time algorithm that solves the search problem of R with negligible error probability.

   (Hint: See Exercise 6.4 for a related procedure.)

2. Reflect on the need to require that one (correct) solution occurs with probability greater than $0.5+(1/p(|x|))$. Specifically, what can we do if it is only guaranteed that for every $x \in S_R$ it holds that $\Pr[A(x) \in R(x)] > 0.5 + (1/p(|x|))$ (and for every $x \notin S_R$ it holds that $\Pr[A(x)=\perp] > 0.5 + (1/p(|x|))$)?

Exercise 6.4 (error-reduction for $\mathcal{BPP}$) For $\varepsilon : \mathbb{N} \to [0,1]$, let $\mathcal{BPP}_\varepsilon$ denote the class of decision problems that can be solved in probabilistic polynomial-time with error probability upper-bounded by ε. Prove the following two claims:

1. For every positive polynomial p and $\varepsilon(n) = (1/2) - (1/p(n))$, the class $\mathcal{BPP}_\varepsilon$ equals $\mathcal{BPP}$.

2. For every positive polynomial p and $\varepsilon(n) = 2^{-p(n)}$, the class $\mathcal{BPP}$ equals $\mathcal{BPP}_\varepsilon$.

Formulate a corresponding version for the setting of search problem. Specifically, for every input that has a solution, consider the probability that a specific solution is output.

Guideline: Given an algorithm A for the syntactically weaker class, consider an algorithm A' that on input x invokes A on x for $t(|x|)$ times, and rules by majority. For Part 1 set $t(n) = O(p(n)^2)$ and apply Chebyshev's Inequality. For Part 2 set $t(n) = O(p(n))$ and apply the Chernoff Bound.

Exercise 6.5 (error-reduction for $\mathcal{RP}$) For $\rho : \mathbb{N} \to [0,1]$, we define the class of decision problem $\mathcal{RP}_\rho$ such that it contains S if there exists a probabilistic polynomial-time algorithm A such that for every $x \in S$ it holds that $\Pr[A(x) = 1] \geq \rho(|x|)$ and for every $x \notin S$ it holds that $\Pr[A(x) = 0] = 1$. Prove the following two claims:

1. For every positive polynomial p, the class $\mathcal{RP}_{1/p}$ equals $\mathcal{RP}$.
2. For every positive polynomial p, the class $\mathcal{RP}$ equals $\mathcal{RP}_\rho$, where $\rho(n) = 1 - 2^{-p(n)}$.

(Hint: The one-sided error allows using an "or-rule" (rather than a "majority-rule") for the decision.)

Exercise 6.6 (error-reduction for $\mathcal{ZPP}$) For $\rho : \mathbb{N} \to [0,1]$, we define the class of decision problem $\mathcal{ZPP}_\rho$ such that it contains S if there exists a probabilistic polynomial-time algorithm A such that for every x it holds that $\Pr[A(x) = \chi_S(x)] \geq \rho(|x|)$ and $\Pr[A(x) \in \{\chi_S(x), \perp\}] = 1$, where $\chi_S(x) = 1$ if $x \in S$ and $\chi_S(x) = 0$ otherwise. Prove the following two claims:

1. For every positive polynomial p, the class $\mathcal{ZPP}_{1/p}$ equals $\mathcal{ZPP}$.
2. For every positive polynomial p, the class $\mathcal{ZPP}$ equals $\mathcal{ZPP}_\rho$, where $\rho(n) = 1 - 2^{-p(n)}$.

Exercise 6.7 (an alternative definition of $\mathcal{ZPP}$) We say that the decision problem S is solvable in expected probabilistic polynomial-time if there exists a randomized algorithm A and a polynomial p such that for every $x \in \{0,1\}^*$ it holds that $\Pr[A(x) = \chi_S(x)] = 1$ and the expected number of steps taken by $A(x)$ is at most $p(|x|)$. Prove that $S \in \mathcal{ZPP}$ if and only if S is solvable in expected probabilistic polynomial-time.

Guideline: Repeatedly invoking a ZPP algorithm until it yields an output other than $\perp$, results in an expected probabilistic polynomial-time solver. On the other hand, truncating runs of an expected probabilistic polynomial-time algorithm once they exceed twice the expected number of steps (and outputting $\perp$ on such runs), we obtain a ZPP algorithm.

Exercise 6.8 Let $\mathcal{BPP}$ and co$\mathcal{RP}$ be classes of promise problems (as in Theorem 6.7).

1. Prove that every problem in $\mathcal{BPP}$ is reducible to the set $\{1\} \in \mathcal{P}$ by a *two-sided error* randomized Karp-reduction.

   (Hint: Such a reduction may effectively decide membership in any set in $\mathcal{BPP}$.)

2. Prove that if a set S is Karp-reducible to $\mathcal{RP}$ (resp., co$\mathcal{RP}$) via a deterministic reduction then $S \in \mathcal{RP}$ (resp., $S \in$ co$\mathcal{RP}$).

Exercise 6.9 (randomness-efficient error-reductions) Note that standard error-reduction (see Exercise 6.4) yields error probability δ at the cost of increasing the randomness complexity by a *factor* of $O(\log(1/\delta))$. Using the randomness-efficient error-reductions outlined in §D.4.1.3, show that error probability δ can be obtained at the cost of increasing the randomness complexity by a constant factor and an *additive term* of $1.5\log_2(1/\delta)$. Note that this allows satisfying the hypothesis made in the illustrative paragraph of the proof of Theorem 6.7.

Exercise 6.10 In continuation to the illustrative paragraph of the proof of Theorem 6.7, consider the promise problem $\Pi' = (\Pi'_{\rm yes}, \Pi'_{\rm no})$ such that $\Pi'_{\rm yes} = \{(x, r') : |r'| = p(|x|)/2 \wedge (\forall r'' \in \{0,1\}^{|r'|})\, A'(x, r'r'') = 1\}$ and $\Pi'_{\rm no} = \{(x, r') : x \notin S\}$. Recall that for every x it holds that $\mathsf{Pr}_{r\in\{0,1\}^{p(|x|)}}[A'(x,r) \neq \chi_S(x)] < 2^{-((p(|x|)/2)+1)}$.

1. Show that mapping x to (x, r'), where r' is uniformly distributed in $\{0,1\}^{p(|x|)/2}$, constitutes a one-sided error randomized Karp-reduction of S to Π'.
2. Show that Π' is in the promise problem class co$\mathcal{RP}$.

Exercise 6.11 Prove that for every $S \in \mathcal{NP}$ there exists a probabilistic polynomial-time algorithm A such that for every $x \in S$ it holds that $\mathsf{Pr}[A(x) = 1] > 0$ and for every $x \notin S$ it holds that $\mathsf{Pr}[A(x) = 0] = 1$. That is, A has error probability at most $1 - \exp(-{\rm poly}(|x|))$ on yes-instances but never errs on no-instances. Thus, $\mathcal{NP}$ may be fictitiously viewed as having a huge one-sided error probability.

Exercise 6.12 (randomized versions of $\mathcal{NP}$) In continuation to Footnote 6, consider the following two variants of $\mathcal{MA}$ (which we consider the main randomized version of $\mathcal{NP}$).

1. $S \in \mathcal{MA}^{(1)}$ if there exists a probabilistic polynomial-time algorithm V such that for every $x \in S$ there exists $y \in \{0,1\}^{{\rm poly}(|x|)}$ such that $\mathsf{Pr}[V(x,y)=1] \geq 1/2$, whereas for every $x \notin S$ and every y it holds that $\mathsf{Pr}[V(x,y)=0] = 1$.
2. $S \in \mathcal{MA}^{(2)}$ if there exists a probabilistic polynomial-time algorithm V such that for every $x \in S$ there exists $y \in \{0,1\}^{{\rm poly}(|x|)}$ such that $\mathsf{Pr}[V(x,y)=1] \geq 2/3$, whereas for every $x \notin S$ and every y it holds that $\mathsf{Pr}[V(x,y)=0] \geq 2/3$.

Prove that $\mathcal{MA}^{(1)} = \mathcal{NP}$ whereas $\mathcal{MA}^{(2)} = \mathcal{MA}$.

Guideline: For the first part, note that a sequence of internal coin tosses that makes V accept (x, y) can be incorporated into y itself (yielding a standard NP-witness). For the second part, apply the ideas underlying the proof of Theorem 6.7.

Exercise 6.13 (time hierarchy theorems for promise problem versions of BPTIME**)** Fixing a model of computation, let BPTIME(t) denote the class of promise problems that are solvable by a randomized algorithm of time complexity t that has a two-sided error probability at most $1/3$. (The common definition refers only to decision problems.) Formulate and prove results analogous to Theorem 4.3 and Corollary 4.4.

Guideline: Analogously to the proof of Theorem 4.3, we construct a Boolean function f by associating with each admissible machine M an input x_M, and making sure that $\Pr[f(x_M) \neq M'(x)] \geq 2/3$, where $M'(x)$ denotes the emulation of $M(x)$ suspended after $t_1(|x|)$ steps. The key point is that f is a partial function (corresponding to a promise problem) that is defined only for machines that have two-sided error at most 1/3 (on every input). This allows for a randomized computation of f with two-sided error probability at most 1/3 (on each input on which f is defined).

Exercise 6.14 (extracting square roots modulo a prime) Using the following guidelines, present a probabilistic polynomial-time algorithm that, on input a prime P and a quadratic residue $s \pmod P$, returns r such that $r^2 \equiv s \pmod P$.

1. Prove that if $P \equiv 3 \pmod 4$ then $s^{(P+1)/4} \bmod P$ is a square root of the quadratic residue $s \pmod P$.

2. Note that the procedure suggested in Item 1 relies on the ability to find an odd power e such that $s^e \equiv 1 \pmod P$. (In Item 1, we used $e = (P-1)/2$, which is odd since $P \equiv 3 \pmod 4$.) Once such a power is found, we may output $s^{(e+1)/2} \bmod P$.

   Show that it suffices to find an odd power e together with a residue r and an even power e' such that $s^e r^{e'} \equiv 1 \pmod P$.

3. Given a prime $P \equiv 1 \pmod 4$, a quadratic residue s, and a quadratic non-residue r (equiv., $r^{(P-1)/2} \equiv -1 \pmod P$), show that e and e' as in Item 2 can be efficiently found.[20]

4. Prove that, for a prime P, with probability 1/2 a uniformly chosen $r \in \{1, ..., P\}$ satisfies $r^{(P-1)/2} \equiv -1 \pmod P$.

Note that randomization is used only in the last item, which in turn is used only for $P \equiv 1 \pmod 4$.

Exercise 6.15 (small-space randomized step-counter) A step-counter is an algorithm that runs for a number of steps that is specified in its input. Actually, such an algorithm may run for a somewhat larger number of steps but halt after issuing a number of "signals" as specified in its input, where these signals are defined as entering (and leaving) a designated state (of the algorithm). A step-counter may be run in parallel to another procedure in order to suspend the execution after a desired number of steps (of the other procedure) has elapsed. We note that there exists a simple deterministic machine that, on input n, halts after issuing n signals while using $O(1) + \log_2 n$ space (and $\widetilde{O}(n)$ time). The goal of this exercise is presenting a (randomized) step-counter that allows for many more signals while

[20]Write $(P-1)/2 = (2j+1) \cdot 2^{i_0}$, and note that $s^{(2j+1)\cdot 2^{i_0}} \equiv 1 \pmod P$. Assuming we have that if for some $i' > i > 0$ and j' it holds that $s^{(2j+1)\cdot 2^i} r^{(2j'+1)\cdot 2^{i'}} \equiv 1 \pmod P$, show how to find $i'' > i-1$ and j'' such that $s^{(2j+1)\cdot 2^{i-1}} r^{(2j''+1)\cdot 2^{i''}} \equiv 1 \pmod P$. (Extra hint: $s^{(2j+1)\cdot 2^{i-1}} r^{(2j'+1)\cdot 2^{i'-1}} \equiv \pm 1 \pmod P$ and $r^{(2j+1)\cdot 2^{i_0}} \equiv -1 \pmod P$.) Note that for $i = 1$ we obtain what we need.

using the same amount of space. Specifically, present a (randomized) algorithm that, on input n, uses $O(1)+\log_2 n$ space (and $\widetilde{O}(2^n)$ time) and halts after issuing an expected number of 2^n signals. Furthermore, prove that, with probability at least $1-2^{-k+1}$, this step-counter halts after issuing a number of signals that is between 2^{n-k} and 2^{n+k}.

Guideline: Repeat the following experiment till reaching success. Each trial consists of uniformly selecting n bits (i.e., tossing n unbiased coins), and is deemed successful if all bits turn out to equal the value 1 (i.e., all outcomes equal HEAD). Note that such a trial can be implemented by using space $O(1)+\log_2 n$ (mainly for implementing a standard counter for determining the number of bits). Thus, each trial is successful with probability 2^{-n}, and the expected number of trials is 2^n.

Exercise 6.16 (analysis of random walks on arbitrary undirected graphs) In order to complete the proof of Proposition 6.11, prove that if $\{u,v\}$ is an edge of the graph $G=(V,E)$ then $\mathrm{E}[X_{u,v}] \leq 2|E|$. Recall that, for a fixed graph, $X_{u,v}$ is a random variable representing the number of steps taken in a random walk that starts at the vertex u until the vertex v is first encountered.

Guideline: Let $Z_{u,v}(n)$ be a random variable counting the number of *minimal* paths from u to v that appear along a random walk of length n, where the walk starts at the stationary vertex distribution (which is well-defined assuming the graph is not bipartite, which in turn may be enforced by adding a self-loop). On one hand, $\mathsf{E}[X_{u,v}+X_{v,u}] = \lim_{n\to\infty}(n/\mathsf{E}[Z_{u,v}(n)])$, due to the memoryless property of the walk. On the other hand, letting $\chi_{v,u}(i) \stackrel{\rm def}{=} 1$ if the edge $\{u,v\}$ was traversed from v to u in the $i^{\rm th}$ step of such a random walk and $\chi_{v,u}(i) \stackrel{\rm def}{=} 0$ otherwise, we have $\sum_{i=1}^{n}\chi_{v,u}(i) \leq Z_{u,v}(n)+1$ and $\mathrm{E}[\chi_{v,u}(i)] = 1/2|E|$ (because, in each step, each directed edge appears on the walk with equal probability). It follows that $\mathrm{E}[X_{u,v}] < 2|E|$.

Exercise 6.17 (the class $\mathcal{PP} \supseteq \mathcal{BPP}$ and its relation to $\#\mathcal{P}$) In contrast to $\mathcal{BPP}$, which refers to useful probabilistic polynomial-time algorithms, the class $\mathcal{PP}$ does not capture such algorithms but is rather closely related to $\#\mathcal{P}$. A decision problem S is in $\mathcal{PP}$ if there exists a probabilistic polynomial-time algorithm A such that, for every x, it holds that $x \in S$ if and only if $\mathsf{Pr}[A(x)=1] > 1/2$. Note that $\mathcal{BPP} \subseteq \mathcal{PP}$. Prove that $\mathcal{PP}$ is Cook-reducible to $\#\mathcal{P}$ and vise versa.

Guideline: For $S \in \mathcal{PP}$ (by virtue of the algorithm A), consider the relation R such that $(x,r) \in R$ if and only if A accepts the input x when using the random-input $r \in \{0,1\}^{p(|x|)}$, where p is a suitable polynomial. Thus, $x \in S$ if and only if $|R(x)| > 2^{p(|x|)-1}$, which in turn can de determined by querying the counting function of R. To reduce $f \in \#\mathcal{P}$ to $\mathcal{PP}$, consider the relation $R \in \mathcal{PC}$ that is counted by f (i.e., $f(x)=|R(x)|$) and the decision problem S_f as defined in Proposition 6.13. Let p be the polynomial specifying the length of solutions for R (i.e., $(x,y) \in R$ implies $|y|=p(|x|)$), and consider the algorithm A' that on input (x,N) proceeds as follows: With probability $1/2$, it uniformly selects $y \in \{0,1\}^{p(|x|)}$ and accepts if and only if $(x,y) \in R$, and otherwise (i.e., in the other case) it accepts with probability $\frac{2^{p(|x|)}-N+0.5}{2^{p(|x|)}}$. Prove that $(x,N) \in S_f$ if and only if $\mathsf{Pr}[A'(x)=1] > 1/2$.

Exercise 6.18 (enumeration problems) For any binary relation R, define the enumeration problem of R as a function $f_R : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^* \cup \{\perp\}$ such that $f_R(x,i)$ equals the i^{th} element in $|R(x)|$ if $|R(x)| \geq i$ and $f_R(x,i) = \perp$ otherwise. The above definition refers to the standard lexicographic order on strings, but any other efficient order of strings will do.[21]

1. Prove that, for any polynomially bounded R, computing $\#R$ is reducible to computing f_R.

2. Prove that, for any $R \in \mathcal{PC}$, computing f_R is reducible to some problem in $\#\mathcal{P}$.

   Guideline: Consider the binary relation $R' = \{(\langle x,b\rangle, y) : (x,y) \in R \wedge y \leq b\}$, and show that f_R is reducible to $\#R'$. (Extra hint: Note that $f_R(x,i) = y$ if and only if $|R'(\langle x,y\rangle)| = i$ and for every $y' < y$ it holds that $|R'(\langle x,y'\rangle)| < i$.)

Exercise 6.19 (computing the permanent of integer matrices) Prove that computing the permanent of matrices with 0/1-entries is computationally equivalent to computing the number of perfect matchings in bipartite graphs.
(Hint: Given a bipartite graph $G = ((X,Y),E)$, consider the matrix M representing the edges between X and Y (i.e., the (i,j)-entry in M is 1 if the i^{th} vertex of X is connected to the j^{th} entry of Y), and note that only perfect matchings in G contribute to the permanent of M.)

Exercise 6.20 (computing the permanent modulo 3) Combining Proposition 6.19 and Theorem 6.27, prove that for every integer $n > 1$ that is relatively prime to c, computing the permanent modulo n is NP-hard under randomized reductions.[22] Since Proposition 6.19 holds for $c = 2^{10}$, hardness holds for every odd integer $n > 1$.

Guideline: Applying the reduction of Proposition 6.19 to the promise problem of deciding whether a 3CNF formula has a unique satisfiable assignment or is unsatisfiable. Use the fact that n does not divide any power of c.

Exercise 6.21 (negative values in Proposition 6.19) Assuming $\mathcal{P} \neq \mathcal{NP}$, prove that Proposition 6.19 cannot hold for a set I containing only non-negative integers. Note that the claim holds even if the set I is not finite (and even if I is the set of all non-negative integers).

Guideline: A reduction as in Proposition 6.19 provides a Karp-reduction of `3SAT` to deciding whether the permanent of a matrix with entries in I is non-zero. Note that the permanent of a *non-negative* matrix is non-zero if and only if the corresponding bipartite graph has a perfect matching.

---

[21] An order of strings is a 1-1 and onto mapping μ from the natural numbers to the set of all strings. Such order is called efficient if both μ and its inverse are efficiently computable. The standard lexicographic order satisfies $\mu(i) = y$ if the (compact) binary expansion of i equals $1y$; that is $\mu(1) = \lambda$, $\mu(2) = 0$, $\mu(3) = 1$, $\mu(4) = 00$, etc.

[22] Actually, a sufficient condition is that n does not divide any power of c. Thus (referring to $c = 2^{10}$), hardness holds for every integer $n > 1$ that is not a power of 2. On the other hand, for any fixed $n = 2^e$, the permanent modulo n can be computed in polynomial-time [213, Thm. 3].

Exercise 6.22 (high-level analysis of the permanent reduction) Establish the correctness of the high-level reduction presented in the proof of Proposition 6.19. That is, show that if the clause gadget satisfies the three conditions postulated in the said proof, then each satisfying assignment of ϕ contributes exactly c^m to the SWCC of G_ϕ and unsatisfying assignments have no contribution.

Guideline: Cluster the cycle covers of G_ϕ according to the set of track edges that they use (i.e., the edges of the cycle cover that belong to the various tracks). (Note the correspondence between these edges and the external edges used in the definition of the gadget's properties.) Using the postulated conditions (regarding the clause gadget) prove that, for each such set T of track edges, if the sum of the weights of all cycle covers that use the track edges T is non-zero then the following hold:

1. The intersection of T with the set of track edges incident at each specific clause gadget is non-empty. Furthermore, if this set contains an incoming edge (resp., outgoing edge) of some entry-vertex (resp., exit-vertex) then it also contains an outgoing edge (resp., incoming edge) of the corresponding exit-vertex (resp., entry-vertex).
2. If T contains an edge that belongs to some track then it contains all edges of this track. It follows that, for each variable x, the set T contains the edges of a single track associated with x.
3. The tracks "picked" by T correspond to a single truth assignment to the variables of ϕ, and this assignment satisfies ϕ (because, for each clause, T contains an external edge that corresponds to a literal that satisfies this clause).

It follows that each satisfying assignment of ϕ contributes exactly c^m to the SWCC of G_ϕ.

Exercise 6.23 (analysis of the implementation of the clause gadget) Establish the correctness of the implementation of the clause gadget presented in the proof of Proposition 6.19. That is, show that if the box satisfy the three conditions postulated in the said proof, then the clause gadget of Figure 6.4 satisfies the conditions postulated for it.

Guideline: Cluster the cycle covers of a gadget according to the set of non-box edges that they use, where non-box edges are the edges shown in Figure 6.4. Using the postulated conditions (regarding the box) prove that, for each set S of non-box edges, if the sum of the weights of all cycle covers that use the non-box edges S is non-zero then the following hold:

1. The intersection of S with the set of edges incident at each box must contain two (non-selfloop) edges, one incident at each of the box's terminals. Needless to say, one edge is incoming and the other outgoing. Referring to the six edges that connects one of the six designated vertices (of the gadget) with the corresponding box terminals as connectives, note that if S contains a connective incident at the terminal of some box then it must also contain the connective incident at the other terminal. In such a case, we say that this box is picked by S,
2. Each of the three (literal-designated) boxes that is not picked by S is "traversed" from left to right (i.e., the cycle cover contains an incoming edge of the left terminal and an outgoing edge of the right terminal). Thus, the set S must contain a connective, because otherwise no directed cycle may cover the leftmost vertex shown in Figure 6.4. That is, S must pick some box.

3. The set S is fully determined by the non-empty set of boxes that it picks.

The postulated properties of the clause gadget follow, with $c = b^5$.

Exercise 6.24 (analysis of the design of a box for the clause gadget) Prove that the 4-by-4 matrix presented in Eq. (6.4) satisfies the properties postulated for the "box" used in the second part of the proof of Proposition 6.19. In particular:

1. Show a correspondence between the conditions required of the box and conditions regarding the value of the permanent of certain sub-matrices of the adjacency matrix of the graph.

   (Hint: For example, show that the first condition correspond to requiring that the value of the permanent of the entire matrix equals zero. The second condition refers to sub-matrices obtained by omitting either the first row and fourth column or the fourth row and first column.)

2. Verify that the matrix in Eq. (6.4) satisfies the aforementioned conditions (regarding the value of the permanent of certain sub-matrices).

Prove that no 3-by-3 matrix (and thus also no 2-by-2 matrix) can satisfy the aforementioned conditions.

Exercise 6.25 (error reduction for approximate counting) Show that the error probability δ in Definition 6.22 can be reduced from $1/3$ (or even $(1/2) + (1/\mathrm{poly}(|x|))$ to $\exp(-\mathrm{poly}(|x|))$.

Guideline: Invoke the weaker procedure for an adequate number of times and take the *median* value among the values obtained in these invocations.

Exercise 6.26 (relative approximation for DNF satisfaction) Referring to the text of §6.2.2.1, prove the following claims.

1. Both assumptions regarding the general setting hold in case $S_i = C_i^{-1}(1)$, where $C_i^{-1}(1)$ denotes the set of truth assignments that satisfy the conjunction C_i.

   Guideline: In establishing the second assumption note that it reduces to the conjunction of the following two assumptions:

   (a) Given i, one can efficiently generate a uniformly distributed element of S_i. Actually, generating a distribution that is almost uniform over S_i suffices.

   (b) Given i and x, one can efficiently determine whether $x \in S_i$.

2. Prove Proposition 6.24, relating to details such as the error probability in an implementation of Construction 6.23.

3. Note that Construction 6.23 does not require exact computation of $|S_i|$. Analyze the output distribution in the case that we can only approximate $|S_i|$ up-to a factor of $1 \pm \varepsilon'$.

Exercise 6.27 (reducing the relative deviation in approximate counting) Prove that, for any $R \in \mathcal{PC}$ and every polynomial p and constant $\delta < 0.5$, there exists $R' \in \mathcal{PC}$ such that $(1/p, \delta)$-approximation for $\#R$ is reducible to $(1/2, \delta)$-approximation for $\#R'$.

Guideline: For $t(n) = \Omega(p(n))$, let $R' = \{(x, (y_1, ..., y_{t(|x|)})) : (\forall i)\,(x, y_i) \in R\}$. Note that $|R(x)| = |R'(x)|^{1/t(|x|)}$, and thus if $a = (1 \pm (1/2)) \cdot |R'(x)|$ then $a^{1/t(|x|)} = (1 \pm (1/2))^{1/t(|x|)} \cdot |R(x)|$. Furthermore, prove that $(1/p, \delta)$-approximation for $\#R$ is reducible to approximating $\#R''$ to within a factor of $F(n) = \exp(p(n))$ with error probability δ, for some $\mathbb{R}'' \in \mathcal{PC}$.
(Hint: Same as the main part. Note that the length of the solution for $R''(x)$ is larger than $p(|x|)$ and so there is nothing wrong in approximating $\#R''(|x|)$ to within $F(|x|)$.)

Exercise 6.28 (deviation reduction in approximate counting, cont.) In continuation to Exercise 6.27, prove that if R is NP-complete via parsimonious reductions then, for every positive polynomial p and constant $\delta < 0.5$, the problem of $(1/p, \delta)$-approximation for $\#R$ is reducible to $(1/2, \delta)$-approximation for $\#R$.
(Hint: Compose the reduction (to the problem of $(1/2, \delta)$-approximation for $\#R'$) provided in Exercise 6.27 with the parsimonious reduction of $\#R'$ to $\#R$.)
Prove that, for every function F' such that $F'(n) = \exp(n^{o(1)})$, we can also reduce the aforementioned problems to the problem of approximating $\#R$ to within a factor of F' with error probability δ.

Guideline: Using R'' as in Exercise 6.27, we encounter a technical difficulty. The issue is that the composition of the ("amplifying") reduction of $\#R$ to $\#R''$ with the parsimonious reduction of $\#R''$ to $\#R$ may increase the length of the instance. Indeed, the length of the new instance is polynomial in the length of the original instance, but this polynomial may depend on R'', which in turn depends on F'. Thus, we cannot use $F'(n) = \exp(n^{1/O(1)})$ but $F'(n) = \exp(n^{o(1)})$ is fine.

Exercise 6.29 Referring to the procedure in the proof Theorem 6.25, show how to use an NP-oracle in order to determine whether the number of solutions that "pass a random sieve" is greater than t. You are allowed queries of length polynomial in the length of x, h and in the size of t.
(Hint: Consider the set $S'_{R,H} \stackrel{\text{def}}{=} \{(x, i, h, 1^t) : \exists y_1, ..., y_t \text{ s.t. } \psi'(x, h, y_1, ..., y_t)\}$, where $\psi'(x, h, y_1, ..., y_t)$ holds if and only if the y_j are different and for every j it holds that $(x, y_j) \in R \wedge h(y_j) = 0^i$.)

Exercise 6.30 (parsimonious reductions and Theorem 6.27) Demonstrate the importance of parsimonious reductions in Theorem 6.27 by proving the following:

1. There exists a search problem $R \in \mathcal{PC}$ such that every problem in $\mathcal{PC}$ is reducible to R (by a non-parsimonious reduction) and still the the promise problem $(\mathtt{US}_R, \overline{S}_R)$ is decidable in polynomial-time.

   Guideline: Consider the following artificial witness relation R for $\mathtt{SAT}$ in which $(\phi, \sigma\tau) \in R$ if $\sigma \in \{0, 1\}$ and τ satisfies ϕ. Note that the standard witness relation of $\mathtt{SAT}$ is reducible to R, but this reduction is not parsimonious. Also note that $\mathtt{US}_R = \emptyset$ and thus $(\mathtt{US}_R, \overline{S}_R)$ is trivial.

2. There exists a search problem $R \in \mathcal{PC}$ such that $\#R$ is $\#\mathcal{P}$-complete and still the the promise problem $(\mathtt{US}_R, \overline{S}_R)$ is decidable in polynomial-time.

   Guideline: One easy proof is to use the relation suggested in the guideline to Part 1. A totally different proof relies on Theorem 6.18 and on the fact that it is easy to decide $(\mathtt{US}_R, \overline{S}_R)$ when R is the corresponding perfect matching relation (by computing the determinant).

Exercise 6.31 Prove that SAT is randomly reducible to deciding unique solution for SAT, *without using the fact that* SAT *is NP-complete via parsimonious reductions.*

Guideline: Follow the proof of Theorem 6.27, while using the family of pairwise independent hashing functions provided in Construction D.3 (or in Eq. (8.17)). Note that, in this case, the condition $(\tau \in R_{\mathtt{SAT}}(\phi)) \wedge (h(\tau) = 0^i)$ can be directly encoded as a CNF formula. That is, consider the formula ϕ_h such that $\phi_h(z) \stackrel{\text{def}}{=} \phi(z) \wedge (h(z) = 0^i)$, and note that $h(z) = 0^i$ can be written as the conjunction of i clauses, where each clause is a CNF that is logically equivalent to the parity of some of the bits of z (where the identity of these bits is determined by h).

Exercise 6.32 (an alternative procedure for approximate counting) Adapt Step 1 of Construction 6.30 so to obtain an approximate counting procedure for $\#R$.

Guideline: For $m = 0, 1, ...\ell$, the procedure invokes Step 1 of Construction 6.30 until a negative answer is obtained, and outputs 2^m for the current value of m. For $|R(x)| > 1000\ell$, this yields a constant factor approximation of $|R(x)|$. In fact, we can obtain a better estimate by making additional queries at iteration m (i.e., queries of the form $(x, h, 1^i)$ for $i = 10\ell, ..., 1000\ell$). The case $|R(x)| \leq 1000\ell$ can be treated by using Step 2 of Construction 6.30, in which case we obtain an exact count.

Chapter 7

The Bright Side of Hardness

So saying she donned her beautiful, glittering golden–Ambrosial sandals, which carry her flying like the wind over the vast land and sea; she grasped the redoubtable bronze-shod spear, so stout and sturdy and strong, wherewith she quells the ranks of heroes who have displeased her, the [bright-eyed] daughter of her mighty father.

Homer, Odyssey, 1:96–101

The existence of natural computational problems that are (or seem to be) infeasible to solve is usually perceived as bad news, because it means that we cannot do things we wish to do. But these bad news have a positive side, because hard problem can be "put to work" to our benefit, most notably in cryptography.

One key issue that arises whenever one tries to utilize hard problem is bridging the gap between "occasional" hardness (e.g., worst-case hardness or mild average-case hardness) and "typical" hardness (i.e., inapproximability). Much of the current chapter is devoted to this issue, which is known by the term *hardness amplification*.

Summary: We consider two conjectures that are related to $\mathcal{P} \neq \mathcal{NP}$. The first conjecture is that there are problems that are solvable in exponential-time but are not solvable by (non-uniform) families of small (say polynomial-size) circuits. We show that this worst-case conjecture can be transformed into an average-case hardness result of the type that can be used towards derandomized $\mathcal{BPP}$ in a non-trivial way (see Section 8.4).

The second conjecture is that there are problems in NP (i.e., search problems in $\mathcal{PC}$) for which it is easy to generate (solved) instances that are hard to solve for other people. This conjecture is captured in the formulation of *one-way functions*, which are functions that are easy to evaluate but hard to invert (in an average-case sense). We show that

functions that are hard to invert in a relatively mild average-case sense yield functions that are hard to invert almost everywhere, and that the latter yield predicates that are very hard to approximate (called *hard-core predicates*). The latter are useful for the construction of general-purpose pseudorandom generators (see Section 8.3) as well as for a host of cryptographic applications (see Appendix C).

The order of presentation of the two aforementioned conjectures and their consequences is actually reversed: We start (in Section 7.1) with the study of one-way function, and only later (in Section 7.2) turn to the study of problems in $\mathcal{E}$ that are hard for small circuits.

Teaching note: We list several reasons for preferring the aforementioned order of presentation. First, we mention the conceptual appeal of one-way functions and the fact that they have very practical applications. Second, hardness amplification in the context of one-way functions is technically simpler in comparison to the amplification of hardness in the context of $\mathcal{E}$. (In fact, Section 7.2 seems the most technical text in this book.) Third, some of the techniques that are shared by both treatments seem easier to understand first in the context of one-way functions. Last, the chosen order facilitates the possibility of teaching hardness amplification only in one incarnation, where the context of one-way functions is recommended as the incarnation of choice (for the aforementioned reasons).
If you wish to teach hardness amplification and pseudorandomness in the two aforementioned incarnations, then we suggest following the order of the current text. That is, first teach hardness amplification in its two incarnations, and only next teach pseudorandomness in the corresponding incarnations.

Prerequisites: We assume a basic familiarity with elementary probability theory (see Section D.1) and randomized algorithms (see Section 6.1). In particular, standard conventions regarding random variables (presented in Section D.1.1) will be extensively used.

7.1 One-Way Functions

Loosely speaking, one-way functions are functions that are easy to evaluate but hard (on the average) to invert. Thus, in assuming that one-way functions exist, we are postulating the existence of *efficient processes* (i.e., the computation of the function in the forward direction) *that are hard to reverse.* Analogous phenomena in daily life are known to us in abundance (e.g., the lighting of a match). Thus, the assumption that one-way functions exists is a complexity theoretic analogue of daily experience.

One-way functions can also be thought of as efficient ways for generating "puzzles" that are infeasible to solve; that is, the puzzle is a random image of the function and a solution is a corresponding preimage. Furthermore, the person generating the puzzle knows a solution to it and can efficiently verify the validity of (possibly other) solutions to the puzzle. In fact, as explained in Section 7.1.1, every mechanism for generating such puzzles can be converted to a one-way function.

The reader may note that when presented in terms of generating hard puzzles, one-way functions have a clear cryptographic flavor. Indeed, one-way functions are central to cryptography, but we shall not explore this aspect here (and rather refer the reader to Appendix C). Similarly, one-way functions are closely related to (general-purpose) pseudorandom generators, but this connection will be explored in Section 8.3. Instead, in the current section, we will focus on one-way functions *per se*.

Teaching note: While we recommend including a basic treatment of pseudorandomness within a course on complexity theory, we do not recommend doing so with respect to cryptography. The reason is that cryptography is far more complex than pseudorandomness (e.g., compare the definition of secure encryption to the the definition of pseudorandom generators). The extra complexity is due to conceptual richness, which is something good, except that some of these conceptual issues are central to cryptography but not to complexity theory. Thus, teaching cryptography in the context of a course on complexity theory is likely to either overload the course with material that is not central to complexity theory or cause a superficial and misleading treatment of cryptography. We are not sure as to which of these two possibilities is worse. Still, for the benefit of the interested reader, we have included an overview of the foundations of cryptography as an appendix to the main (see Appendix C).

7.1.1 The concept of one-way functions

Let us assume that $\mathcal{P} \neq \mathcal{NP}$ or even that $\mathcal{NP}$ is not contained in $\mathcal{BPP}$. Can we use this assumption to our benefit? Not really, because the assumption refers to the worst-case complexity of problems, and it may be that hard instances are hard to find. But then, it seems that if we cannot generate hard instances then we cannot benefit from their existence.

In Section 7.2 we shall see that worst-case hardness (of $\mathcal{NP}$ or even $\mathcal{E}$) can be transformed into average-case hardness of $\mathcal{E}$. Such a transformation is not known for $\mathcal{NP}$ itself, and in some applications (e.g., in cryptography) we wish the hard on the average problem to be in $\mathcal{NP}$. In this case, we need to assume that, for some problem in $\mathcal{NP}$, hard instances not only exist but are easy to generate. That is, $\mathcal{NP}$ is "hard on the average" with respect to a distribution that is efficiently sampleable. This assumption will be further discussed in Section 10.2.

However, for the aforementioned applications (e.g., in cryptography) this assumption does not seem to suffice either and we know how to utilize such "hard on the average" problems only when we can efficiently generate hard instances coupled with adequate solutions.[1] That is, we assume that, for some search problem in $\mathcal{PC}$ (resp., decision problem in $\mathcal{NP}$), we can efficiently generate instance-solution pairs (resp., yes-instances coupled with corresponding NP-witnesses) such that the

[1] We wish to stress the difference between the two gaps discussed here. Our feeling is that worst-case hardness (*per se*) is far more difficult to utilize than average-case hardness that does not correspond to an efficient generation of "solved" instances.

instance is hard to solve (of course, for a person that does not get the solution (resp., witness)).

Let us formulate the latter notion. Referring to Definition 2.3, we consider a relation R in $\mathcal{PC}$ (i.e., R is polynomially bounded and membership in R can be determined in polynomial-time), and assume that there exists a probabilistic polynomial-time algorithm G that satisfies the following two conditions:

1. On input 1^n, algorithm G always generates a pair in R such that the first element has length n. That is, $\mathsf{Pr}[G(1^n) \in R \cap (\{0,1\}^n \times \{0,1\}^*)] = 1$.

2. It is infeasible to find solutions to instances that are generated by G; that is, when only given the first element of $G(1^n)$, it is infeasible to find an adequate solution. Formally, denoting the first element of $G(1^n)$ by $G_1(1^n)$, for every probabilistic polynomial-time (solver) algorithm S, it holds that $\mathsf{Pr}[(G_1(1^n), S(G_1(1^n)) \in R] = \mu(n)$, where μ vanishes faster than any polynomial fraction (i.e., for every positive polynomial p and all sufficiently large n it is the case that $\mu(n) < 1/p(n)$).

We call G a generator of solved intractable instances for R. We will show that such a generator exists if and only if one-way functions exists, where one-way functions are functions that are easy to evaluate but hard (on the average) to invert. That is, a function $f:\{0,1\}^* \to \{0,1\}^*$ is called one-way if there is an efficient algorithm that on input x outputs $f(x)$, whereas any feasible algorithm that tries to find a preimage of $f(x)$ under f may succeed only with negligible probability (where the probability is taken uniformly over the choices of x and the algorithm's coin tosses). Associating feasible computations with probabilistic polynomial-time algorithms and negligible functions with functions that vanish faster than any polynomial fraction, we obtain the following definition.

Definition 7.1 (one-way functions): *A function $f:\{0,1\}^* \to \{0,1\}^*$ is called* one-way *if the following two conditions hold:*

1. Easy to evaluate: *There exist a polynomial-time algorithm A such that $A(x) = f(x)$ for every $x \in \{0,1\}^*$.*

2. Hard to invert: *For every probabilistic polynomial-time algorithm A', every polynomial p, and all sufficiently large n,*

$$\mathsf{Pr}_{x\in\{0,1\}^n}[A'(f(x), 1^n) \in f^{-1}(f(x))] \;<\; \frac{1}{p(n)} \tag{7.1}$$

   where the probability is taken uniformly over all the possible choices of $x \in \{0,1\}^n$ and all the possible outcomes of the internal coin tosses of algorithm A'.[2]

[2] An alternative formulation of Eq. (7.1) relies on the conventions in Section D.1.1. Specifically, letting U_n denote a random variable uniformly distributed in $\{0,1\}^n$, we may write Eq. (7.1) as $\mathsf{Pr}[A'(f(U_n), 1^n) \in f^{-1}(f(U_n))] \;<\; 1/p(n)$, recalling that both occurrences of U_n refer to the same sample.

Algorithm A' is given the auxiliary input 1^n so as to allow it to run in time polynomial in the length of x, which is important in case f drastically shrinks its input (e.g., $|f(x)| = O(\log |x|)$). Typically (and, in fact, without loss of generality, see Exercise 7.1), f is length preserving, in which case the auxiliary input 1^n is redundant. Note that A' is not required to output a specific preimage of $f(x)$; any preimage (i.e., element in the set $f^{-1}(f(x))$) will do. (Indeed, in case f is 1-1, the string x is the only preimage of $f(x)$ under f; but in general there may be other preimages.) It is required that algorithm A' fails (to find a preimage) with overwhelming probability, when the probability is also taken over the input distribution. That is, f is "typically" hard to invert, not merely hard to invert in some ("rare") cases.

Proposition 7.2 *The following two conditions are equivalent:*

1. *There exists a generator of solved intractable instances for some $R \in \mathcal{NP}$.*
2. *There exist one-way functions.*

Proof Sketch: Suppose that G is such a generator of solved intractable instances for some $R \in \mathcal{NP}$, and suppose that on input 1^n it tosses $\ell(n)$ coins. For simplicity, we assume that $\ell(n) = n$, and consider the function $g(r) = G_1(1^{|r|}, r)$, where $G(1^n, r)$ denotes the output of G on input 1^n when using coins r (and G_1 is as in the foregoing discussion). Then g must be one-way, because an algorithm that inverts g on input $x = g(r)$ obtains r' such that $G_1(1^n, r') = x$ and $G(1^n, r')$ must be in R (which means that the second element of $G(1^n, r')$ is a solution to x). In case $\ell(n) \neq n$ (and assuming without loss of generality that $\ell(n) \geq n$), we define $g(r) = G_1(1^n, s)$ where n is the largest integer such that $\ell(n) \leq |r|$ and s is the $\ell(n)$-bit long prefix of s.

Suppose, on the other hand, that f is a one-way function. Then $R \stackrel{\text{def}}{=} \{(f(x), x) : x \in \{0,1\}^*\}$ is in $\mathcal{PC}$, and $G(1^n) = (f(r), r)$ for a uniformly selected $r \in \{0,1\}^n$ is a generator of solved intractable instances for R, because any solver of R is effectively inverting f on $f(U_n)$. $\square$

Comments. Several candidates one-way functions and variation on the basic definition appear in Section C.2.1. Here, for the sake of future discussions, we define a stronger version of one-way functions, which refers to the infeasibility of inverting the function by non-uniform circuits of polynomial-size. Here we use the form discussed in Footnote 2.

Definition 7.3 (one-way functions, non-uniformly hard): *A one-way function $f : \{0,1\}^* \to \{0,1\}^*$ is said to be* non-uniformly hard to invert *if for every family of polynomial-size circuits $\{C_n\}$, every polynomial p, and all sufficiently large n,*

$$\Pr[C_n(f(U_n), 1^n) \in f^{-1}(f(U_n))] \; < \; \frac{1}{p(n)}$$

We note that if a function is infeasible to invert by polynomial-size circuits then it is hard to invert by probabilistic polynomial-time algorithms; that is, non-uniformity (more than) compensates for lack of randomness. See Exercise 7.2.

7.1.2 Amplification of Weak One-Way Functions

In the forgoing discussion we have interpreted "hardness on the average" in a very strong sense. Specifically, we required that any feasible algorithm fails to solve the problem (e.g., invert the one-way function) *almost always* (i.e., *except with negligible probability*). This interpretation is indeed the one that is suitable for various applications. Still, a weaker interpretation of hardness on the average, which is also appealing, only requires that any feasible algorithm fails to solve the problem *often enough* (i.e., *with noticeable probability*). The main thrust of the current section is showing that the mild form of hardness on the average can be transformed into the strong form discussed in Section 7.1.1. Let us first define the mild form of hardness on the average, using the framework of one-way functions. Specifically, we define weak one-way functions.

Definition 7.4 (weak one-way functions): *A function $f:\{0,1\}^*\to\{0,1\}^*$ is called* weakly one-way *if the following two conditions hold:*

1. Easy to evaluate: *As in Definition 7.1.*
2. Weakly hard to invert: *There exists a positive polynomial p such that for every probabilistic polynomial-time algorithm A' and all sufficiently large n,*

$$\mathsf{Pr}_{x\in\{0,1\}^n}[A'(f(x),1^n)\notin f^{-1}(f(x))] \;>\; \frac{1}{p(n)} \tag{7.2}$$

where the probability is taken uniformly over all the possible choices of $x\in\{0,1\}^n$ and all the possible outcomes of the internal coin tosses of algorithm A'. In such a case, we say that f is $1/p$-one-way.

Here we require that algorithm A' fails (to find an f-preimage for a random f-image) with noticeable probability, rather than with overwhelmingly high probability (as in Definition 7.1). For clarity, we will occasionally refer to one-way functions as in Definition 7.1 by the term strong one-way functions.

We note that, assuming that one-way functions exist at all, there exists weak one-way functions that are not strongly one-way (see Exercise 7.3). Still, any weak one-way function can be transformed into a strong one-way function. This is indeed the main result of the current section.

Theorem 7.5 (amplification of one-way functions): *The existence of weak one-way functions implies the existence of strong one-way functions.*

Proof Sketch: The construction itself is straightforward. We just parse the argument to the new function into sufficiently many blocks, and apply the weak one-way function on the individual blocks. That is, suppose that f is $1/p$-one-way, for some polynomial p, and consider the following function

$$\begin{aligned} F(x_1,...,x_t) &= (f(x_1),...,f(x_t)) \\ &\quad \text{where } t \stackrel{\text{def}}{=} n\cdot p(n) \text{ and } x_1,...,x_t\in\{0,1\}^n. \end{aligned} \tag{7.3}$$

(Indeed F should be extended to strings of length outside $\{n^2 \cdot p(n) : n \in \mathbb{N}\}$ and this extension must be hard to invert on all preimage lengths.)[3]

We warn that the hardness of inverting the resulting function F is not established by mere "combinatorics" (i.e., considering the relative volume of S^t in $(\{0,1\}^n)^t$, for $S \subset \{0,1\}^n$, where S represents the set of "easy to invert" f-images). Specifically, one may *not* assume that the potential inverting algorithm works independently on each block. Indeed this assumption seems reasonable, but we should not make assumptions regarding arbitrary algorithms (as appearing in the definition of one-way functions) unless we can actually prove that nothing is lost by such assumptions.

The hardness of inverting the resulting function is proved via a so called "reducibility argument" (which is used to prove all conditional results in the area). By a reducibility argument we actually mean a reduction, but one that is analyzed with respect to average case complexity. Specifically, we show that any algorithm that inverts the resulting function F with non-negligible success probability can be used to construct an algorithm that inverts the original function f with success probability that violates the hypothesis (regarding f). In other words, we reduce the task of "strongly inverting" f (i.e., violating its weak one-wayness) to the task of "weakly inverting" F (i.e., violating its strong one-wayness). In particular, on input $y = f(x)$, the reduction invokes the F-inverter (polynomially) many times, each time feeding it with a sequence of random f-images that contains y at a random location. (Indeed such a sequence corresponds to a random image of F.) Details follow.

Suppose towards the contradiction that F is not strongly one-way; that is, there exists a probabilistic polynomial-time algorithm B' and a polynomial $q(\cdot)$ so that for infinitely many m's

$$\mathsf{Pr}[B'(F(U_m)) \in F^{-1}(F(U_m))] > \frac{1}{q(m)} \tag{7.4}$$

Focusing on such a generic m and assuming (see Footnote 3) that $m = n^2 p(n)$, we present the following probabilistic polynomial-time algorithm, A', for inverting f. On input y and 1^n (where supposedly $y = f(x)$ for some $x \in \{0,1\}^n$), algorithm A' proceeds by applying the following probabilistic procedure, denoted I, on input y for $t'(n)$ times, where $t'(\cdot)$ is a polynomial that depends on the polynomials p and q (specifically, we set $t'(n) \stackrel{\text{def}}{=} 2n^2 \cdot p(n) \cdot q(n^2 p(n))$).

Procedure I (on input y and 1^n):

For $i = 1$ to $t(n) \stackrel{\text{def}}{=} n \cdot p(n)$ do begin

(1) Select uniformly and independently a sequence of strings $x_1, ..., x_{t(n)} \in \{0,1\}^n$.

(2) Compute $(z_1, ..., z_{t(n)}) \leftarrow B'(f(x_1), ..., f(x_{i-1}), y, f(x_{i+1}), ..., f(x_{t(n)}))$
(Note that y is placed in the i^{th} position instead of $f(x_i)$.)

(3) If $f(z_i) = y$ then halt and output z_i.

[3] One simple extension is to define $F(x)$ to equal $F(x_1, ..., x_{n \cdot p(n)})$, where n is the largest integer satisfying $n^2 p(n) \leq |x|$ and x_i is the i^{th} consecutive n-bit long string in x (i.e., $x = x_1 \cdot x_{n \cdot p(n)} x'$, where $x_1, ..., x_{n \cdot p(n)} \in \{0,1\}^n$).

(This is considered a *success*).
end

Using Eq. (7.4), we now present a lower bound on the success probability of algorithm A', deriving a contradiction to the theorem's hypothesis. To this end we define a set, denoted S_n, that contains all n-bit strings on which the procedure I succeeds with probability greater than $n/t'(n)$. (The probability is taken only over the coin tosses of procedure I). Namely,

$$S_n \stackrel{\text{def}}{=} \left\{ x \in \{0,1\}^n : \Pr[I(f(x)) \in f^{-1}(f(x))] > \frac{n}{t'(n)} \right\}$$

In the next two claims we shall show that S_n contains all but at most a $1/2p(n)$ fraction of the strings of length n, and that for each string $x \in S_n$ algorithm A' inverts f on $f(x)$ with probability exponentially close to 1. It will follow that A' inverts f on $f(U_n)$ with probability greater than $1-(1/p(n))$, in contradiction to the theorem's hypothesis.

Claim 7.5.1: For every $x \in S_n$

$$\Pr\left[A'(f(x)) \in f^{-1}(f(x))\right] > 1 - 2^{-n}$$

This claim follows directly from the definitions of S_n and A'.

Claim 7.5.2:

$$|S_n| > \left(1 - \frac{1}{2p(n)}\right) \cdot 2^n$$

The rest of the proof is devoted to establishing this claim, and indeed combining Claims 7.5.1 and 7.5.2, the theorem follows.

The key observation is that, for every $i \in [t(n)]$ and every $x_i \in \{0,1\}^n \setminus S_n$, it holds that

$$\begin{aligned}&\Pr\left[B'(F(U_{n^2p(n)})) \in F^{-1}(F(U_{n^2p(n)})) \,\middle|\, U_n^{(i)} = x_i\right] \\ &\quad \leq \Pr\left[I(f(x_i)) \in f^{-1}(f(x_i))\right] \leq \frac{n}{t'(n)}\end{aligned}$$

where $U_n^{(1)}, ..., U_n^{(n \cdot p(n))}$ denote the n-bit long blocks in the random variable $U_{n^2p(n)}$. On the other hand, by Eq. (7.4) we have

$$\begin{aligned}&\sum_{i=1}^{t(n)} \Pr\left[B'(F(U_{n^2p(n)})) \in F^{-1}(F(U_{n^2p(n)})) \wedge U_n^{(i)} \in \{0,1\}^n \setminus S_n\right] \\ &\quad > \frac{1}{q(n^2p(n))} - \Pr\left[(\forall i)\, U_n^{(i)} \in S_n\right]\end{aligned}$$

Thus, we have $\Pr[U_n \in S_n]^{t(n)} > \frac{1}{q(n^2p(n))} - t(n) \cdot \frac{n}{t'(n)}$. Using $t'(n) = 2n^2 \cdot p(n) \cdot q(n^2p(n))$ and $t(n) = n \cdot p(n)$, we get $\Pr[U_n \in S_n] > (1/2q(n^2p(n)))^{1/(n \cdot p(n))}$, which implies $\Pr[U_n \in S_n] > 1 - (1/2p(n))$ for sufficiently large n. Claim 7.5.2 follows, and so does the theorem. $\square$

Digest. Let us recall the structure of the proof of Theorem 7.5. Given a weak one-way function f, we first constructed a polynomial-time computable function F with the intention of later proving that F is strongly one-way. To prove that F is strongly one-way, we used a *reducibility argument*. The argument transforms efficient algorithms that supposedly contradict the strong one-wayness of F into efficient algorithms that contradict the hypothesis that f is weakly one-way. Hence F must be strongly one-way. We stress that our algorithmic transformation, which is in fact a randomized Cook reduction, makes no implicit or explicit assumptions about the structure of the prospective algorithms for inverting F. Such assumptions, as the "natural" assumption that the inverter of F works independently on each block, cannot be justified (at least not at our current state of understanding of the nature of efficient computations).

We use the term a *reducibility argument*, rather than just saying a reduction so as to emphasize that we do *not* refer here to standard (worst-case complexity) reductions. Let us clarify the distinction: In both cases we refer to *reducing* the task of solving one problem to the task of solving another problem; that is, we use a procedure solving the second task in order to construct a procedure that solves the first task. However, in standard reductions one assumes that the second task has a perfect procedure solving it on all instances (i.e., on the worst-case), and constructs such a procedure for the first task. Thus, the reduction may invoke the given procedure (for the second task) on very "non-typical" instances. This cannot be allowed in our reducibility arguments. Here, we are given a procedure that solves the second task *with certain probability with respect to a certain distribution*. Thus, in employing a reducibility argument, we cannot invoke this procedure on any instance. Instead, we must consider the probability distribution, on instances of the second task, induced by our reduction. In our case (as in many cases) the latter distribution equals the distribution to which the hypothesis (regarding solvability of the second task) refers, but other cases may be handled too (e.g., these distributions may be "sufficiently close" for the specific purpose). In any case, a careful analysis of the distribution induced by the reducibility argument is due. (Indeed, the same issue arises in the context of reductions among "distributional problems" considered in Section 10.2.)

An information theoretic analogue. Theorem 7.5 has a natural information theoretic (or "probabilistic") analogue that asserts that repeating an experiment that has a noticeable failure probability, sufficiently many times yields some failure with very high probability. The reader is probably convinced at this stage that the proof of Theorem 7.5 is much more complex than the proof of the information theoretic analogue. In the information theoretic context the repeated events are independent by definition, whereas in the computational context no such independence (which corresponds to the naive argument discussed at the beginning of the proof of Theorem 7.5) can be guaranteed. Another indication to the difference between the two settings follows. In the information theoretic setting the probability that none of the failure events occurs decreases exponentially in the number of repetitions. In contrast, in the computational setting we can only reach an unspecified negligible bound on the inverting probabilities of polynomial-time algorithms. Furthermore, it may be the case that F constructed in the proof of Theorem 7.5 can be efficiently inverted on $F(U_{n^2 p(n)})$ with success probability that is sub-exponentially decreasing (e.g., with probability $2^{-(\log_2 n)^3}$), whereas the analogous information theoretic bound is exponentially decreasing (i.e., e^{-n}).

7.1.3 Hard-Core Predicates

One-way functions *per se* suffice for one central application: the construction of secure signature schemes (see Section C.6). For other applications, one relies not merely on the infeasibility of fully recovering the preimage of a one-way function, but rather on the infeasibility of meaningfully guessing bits in the preimage. The latter notion is captured by the definition of a hard-core predicate.

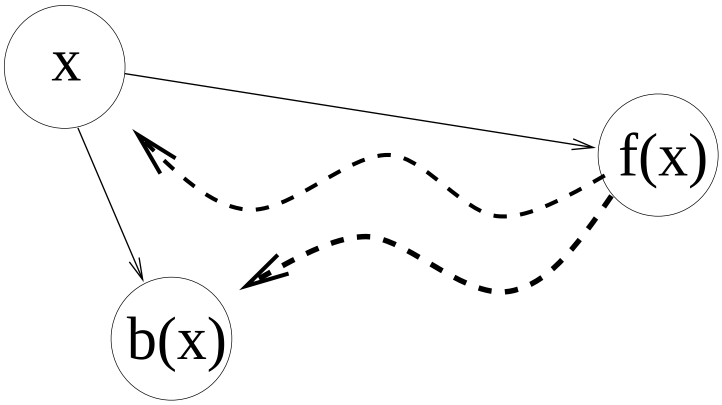
Recall that saying that a function f is one-way means that given a typical y (in the range of f) it is infeasible to find a preimage of y under f. This does not mean that it is infeasible to find out partial information about the preimage(s) of y under f. Specifically, it may be easy to retrieve half of the bits of the preimage (e.g., given a one-way function f consider the function f' defined by $f'(x,r) \stackrel{\rm def}{=} (f(x),r)$, for every $|x|=|r|$). We note that hiding partial information (about the function's preimage) plays an important role in more advanced constructs (e.g., pseudorandom generators and secure encryption). With this motivation in mind, we will show that essentially any one-way function hides specific partial information about its preimage, where this partial information is easy to compute from the preimage itself. This partial information can be considered as a "hard core" of the difficulty of inverting f. Loosely speaking, a *polynomial-time computable* (Boolean) predicate b, is called a hard-core of a function f if no feasible algorithm, given $f(x)$, can guess $b(x)$ with success probability that is non-negligibly better than one half.

Definition 7.6 (hard-core predicates): *A polynomial-time computable predicate $b:\{0,1\}^*\to\{0,1\}$ is called a* hard-core *of a function f if for every probabilistic polynomial-time algorithm A', every positive polynomial $p(\cdot)$, and all sufficiently large n's*

$$\mathsf{Pr}\left[A'(f(x))=b(x)\right] < \frac{1}{2}+\frac{1}{p(n)}$$

where the probability is taken uniformly over all the possible choices of $x\in\{0,1\}^n$ and all the possible outcomes of the internal coin tosses of algorithm A'.

Note that for every $b:\{0,1\}^*\to\{0,1\}$ and $f:\{0,1\}^*\to\{0,1\}^*$, there exist obvious algorithms that guess $b(x)$ from $f(x)$ with success probability at least one half (e.g., the algorithm that, obliviously of its input, outputs a uniformly chosen bit). Also, if b is a hard-core predicate (of any function) then it follows that b is almost unbiased (i.e., for a uniformly chosen x, the difference $|\mathsf{Pr}[b(x)=0]-\mathsf{Pr}[b(x)=1]|$ must be a negligible function in n). Finally, if b is a hard-core of a 1-1 function f that is polynomial-time computable then f must be a one-way function. In general, the interesting case is when being a hard-core is a computational phenumenon rather an information theoretic one (which is due to "information loss" of f).



The solid arrows depict easily computable transformation while the dashed arrows depict infeasible transformations.

Figure 7.1: The hard-core of a one-way function – an illustration.

Theorem 7.7 (a generic hard-core predicate): *For any one-way function f, the inner-product mod 2 of x and r, denoted $b(x,r)$, is a hard-core of $f'(x,r) = (f(x),r)$.*

In other words, given $f(x)$ and a random subset $S \subseteq [|x|]$, it is infeasible to guess $\oplus_{i\in S} x_i$ significantly better than with probbaility $1/2$, where $x = x_1 \cdots x_n$ is uniformly distributed in $\{0,1\}^n$.

Proof Sketch: The proof is by a so-called "reducibility argument" (see Section 7.1.2). Specifically, we reduce the task of inverting f to the task of predicting the hard-core of f', while making sure that the reduction (when applied to input distributed as in the inverting task) generates a distribution as in the definition of the predicting task. Thus, a contradiction to the claim that b is a hard-core of f' yields a contradiction to the hypothesis that f is hard to invert. We stress that this argument is far more complex than analyzing the corresponding "probabilistic" situation (i.e., the distribution of the inner-product mod 2 of X and r, conditioned on a uniformly selected $r \in \{0,1\}^n$, where X is a random variable with super-logarithmic min-entropy, which represents the "effective" knowledge of x, when given $f(x)$).[4]

Our starting point is a probabilistic polynomial-time algorithm B that satisfies, for some polynomial p and infinitely many n's, $\mathsf{Pr}[B(f(X_n),U_n) = b(X_n,U_n)] > (1/2) + (1/p(n))$, where X_n and U_n are uniformly and independently distributed

[4]The min-entropy of X is defined as $\min_v\{\log_2(1/\mathsf{Pr}[X = v])\}$; that is, if X has min-entropy m then $\max_v\{\mathsf{Pr}[X = v]\} = 2^{-m}$. The Leftover Hashing Lemma (see Section D.2) implies that, in this case, $\mathsf{Pr}[b(X,U_n) = 1|U_n] = \frac{1}{2} \pm 2^{-\Omega(m)}$, where U_n denotes the uniform distribution over $\{0,1\}^n$, and $b(u,v)$ denotes the inner-product mod 2 of u and v.

over $\{0,1\}^n$. Using a simple averaging argument, we focus on a $\varepsilon \stackrel{\text{def}}{=} 1/2p(n)$ fraction of the x's for which $\Pr[B(f(x),U_n)=b(x,U_n)] > (1/2)+\varepsilon$ holds. We will show how to use B in order to invert f, on input $f(x)$, provided that x is in the good set (which has density ε).

As a warm-up, suppose for a moment that, for the aforementioned x's, algorithm B succeeds with probability $p > \frac{3}{4}+1/\text{poly}(|x|)$ rather than at least $\frac{1}{2}+1/\text{poly}(|x|)$. In this case, retrieving x from $f(x)$ is quite easy: To retrieve the i^{th} bit of x, denoted x_i, we randomly select $r \in \{0,1\}^{|x|}$, and obtain $B(f(x),r)$ and $B(f(x),r{\oplus}e^i)$, where $e^i = 0^{i-1}10^{|x|-i}$ and $v{\oplus}u$ denotes the addition mod 2 of the binary vectors v and u. A key observation underlying the foregoing scheme as well as the rest of the proof is that $b(x,r{\oplus}s) = b(x,r)\oplus b(x,s)$, which can be readily verified by writing $b(x,y) = \sum_{i=1}^n x_i y_i \bmod 2$ and noting that addition modulo 2 of bits corresponds to their XOR. Indeed, note that if both $B(f(x),r)=b(x,r)$ and $B(f(x),r{\oplus}e^i)=b(x,r{\oplus}e^i)$ hold, then $B(f(x),r)\oplus B(f(x),r\oplus e^i)$ equals $b(x,r)\oplus b(x,r\oplus e^i) = b(x,e^i) = x_i$. The probability that both $B(f(x),r){=}b(x,r)$ and $B(f(x),r{\oplus}e^i){=}b(x,r{\oplus}e^i)$ hold, for a random r, is at least $1-2\cdot(1-p) > \frac{1}{2}+\frac{1}{\text{poly}(|x|)}$. Hence, repeating the above procedure sufficiently many times (using independent random choices of such r's) and ruling by majority, we retrieve x_i with very high probability. Similarly, we can retrieve all the bits of x, and hence invert f on $f(x)$. However, the entire analysis was conducted under (the unjustifiable) assumption that $p > \frac{3}{4}+\frac{1}{\text{poly}(|x|)}$, whereas we only know that $p > \frac{1}{2}+\varepsilon$ for $\varepsilon = 1/\text{poly}(|x|)$.

The problem with the foregoing procedure is that it doubles the original error probability of algorithm B on inputs of the form $(f(x),\cdot)$. Under the unrealistic assumption (made above), that B's average error on such inputs is non-negligibly smaller than $\frac{1}{4}$, the "error-doubling" phenomenon raises no problems. However, in general (and even in the special case where B's error is exactly $\frac{1}{4}$) the above procedure is unlikely to invert f. Note that the *average* error probability of B (for a fixed $f(x)$, when the average is taken over a random r) can not be decreased by repeating B several times (e.g., for every x, it may be that B always answer correctly on three quarters of the pairs $(f(x),r)$, and always err on the remaining quarter). What is required is an *alternative way of using* the algorithm B, a way that does not double the original error probability of B.

The key idea is generating the r's in a way that allows to apply algorithm B only once per each r (and i), instead of twice. Specifically, we will use algorithm B to obtain a "guess" for $b(x,r{\oplus}e^i)$, and obtain $b(x,r)$ in a different way (which does not require using B). The good news is that the error probability is no longer doubled, since we only use B to get a "guess" of $b(x,r{\oplus}e^i)$. The bad news is that we still need to know $b(x,r)$, and it is not clear how we can know $b(x,r)$ without applying B. The answer is that we can guess $b(x,r)$ by ourselves. This is fine if we only need to guess $b(x,r)$ for one r (or logarithmically in $|x|$ many r's), but the problem is that we need to know (and hence guess) the value of $b(x,r)$ for polynomially many r's. The obvious way of guessing these $b(x,r)$'s yields an exponentially small success probability. Instead, we generate these polynomially many r's such that, on one hand they are "sufficiently random" whereas, on the other hand, we can guess all the $b(x,r)$'s with noticeable success probability.[5] Specifically, generating the r's in a specific *pairwise independent* manner will satisfy both (seemingly contradictory) requirements. We stress that in case we are successful (in our guesses for all the $b(x,r)$'s), we can retrieve x with high probability. Hence, we retrieve x with noticeable probability.

A word about the way in which the pairwise independent r's are generated (and the corresponding $b(x,r)$'s are guessed) is indeed in place. To generate $m = \text{poly}(|x|)$ many r's, we uniformly (and independently) select $\ell \stackrel{\text{def}}{=} \log_2(m+1)$ strings in $\{0,1\}^{|x|}$. Let us denote these strings by $s^1,...,s^\ell$. We then guess $b(x,s^1)$ through $b(x,s^\ell)$. Let us denote these guesses, which are uniformly (and independently) chosen in $\{0,1\}$, by σ^1 through σ^ℓ. Hence, the probability that all our guesses for the $b(x,s^i)$'s are correct is $2^{-\ell} = \frac{1}{\text{poly}(|x|)}$. The different r's correspond to the different *non-empty* subsets of $\{1,2,...,\ell\}$. Specifically, for every such subset J, we let $r^J \stackrel{\text{def}}{=} \oplus_{j\in J} s^j$. The reader can easily verify that the r^J's are pairwise independent and each is uniformly distributed in $\{0,1\}^{|x|}$; see Exercise 7.5. The key observation is that $b(x,r^J) = b(x,\oplus_{j\in J}s^j) = \oplus_{j\in J} b(x,s^j)$. Hence, our guess for $b(x,r^J)$ is $\oplus_{j\in J}\sigma^j$, and with noticeable probability all our guesses are correct. Wrapping-up everything, we obtain the following procedure, where $\varepsilon = 1/\text{poly}(n)$ represents a lower-bound on the advantage of B in guessing $b(x,\cdot)$ for an ε fraction of the x's.

Inverting procedure (on input $y = f(x)$ and parameters n and ε):
Set $\ell = \log_2(n/\varepsilon^2) + O(1)$.
(1) Select uniformly and independently $s^1,...,s^\ell \in \{0,1\}^n$.
Select uniformly and independently $\sigma^1,...,\sigma^\ell \in \{0,1\}$.
(2) For every non-empty $J \subseteq [\ell]$, compute $r^J = \oplus_{j\in J}s^j$ and $\rho^J = \oplus_{j\in J}\sigma^j$.
(3) For $i = 1,...,n$ determine the bit z_i according to the majority vote of the $(2^\ell - 1)$-long sequence of bits $(\rho^J \oplus B(f(x), r^J \oplus e^i))_{\emptyset\neq J\subseteq[\ell]}$.
(4) Output $z_1 \cdots z_n$.

Note that the "voting scheme" employed in Step 3 uses pairwise independent samples (i.e., the r^J's), but works essentially as well as it would have worked with independent samples (i.e., the independent r's).[6] That is, for every i and J, it holds that $\Pr_{s^1,...,s^\ell}[B(f(x), r^J\oplus e^i) = b(x, r^J\oplus e^i)] > (1/2)+\varepsilon$, where $r^J = \oplus_{j\in J}s^j$, and for any fixed the events corresponding to different J's are pairwise independent. It follows that *if for every $j \in [\ell]$ it holds that $\sigma^j = b(x,s^j)$*, then for every i

[5]Alternatively, we can try all polynomially many possible guesses. In such a case, we shall output a list of candidates that, with high probability, contains x.

[6]Our focus here is on the accuracy of the approximation obtained by the sample, and not so much on the error probability. We wish to approximate $\Pr[b(x,r) \oplus B(f(x), r\oplus e^i) = 1]$ up to an additive term of ε, because such an approximation allows to correctly determine $b(x,e^i)$. A pairwise independent sample of $O(t/\varepsilon^2)$ points allows for an approximation of a value in $[0,1]$ up to an additive term of ε with error probability $1/t$, whereas a totally random sample of the same size yields error probability $\exp(-t)$. Since we can afford to set $t = \text{poly}(n)$ and work with error $1/2n$, the difference in the error probability between the two approximation schemes is not important here. For a wider perspective see Sections D.1.2 and D.3.

and J we have

$$\begin{aligned} & \mathsf{Pr}_{s^1,\ldots,s^\ell}[\rho^J \oplus B(f(x), r^J \oplus e^i) = b(x, e^i)] \\ & \quad = \mathsf{Pr}_{s^1,\ldots,s^\ell}[B(f(x), r^J \oplus e^i) = b(x, r^J \oplus e^i)] \; > \; \frac{1}{2} + \varepsilon \end{aligned} \tag{7.5}$$

where the equality is due to $\rho^J = \oplus_{j\in J}\sigma^j = b(x, r^J) = b(x, r^J \oplus e^i) \oplus b(x, e^i)$. Note that Eq. (7.5) refers to the correctness of a single vote for $b(x, e^i)$. Using $m = O(n/\varepsilon^2)$ and noting that these (Boolean) votes are pairwise independent, we infer that the probability that the majority of these votes is wrong is upper-bounded by $1/2n$. Using a union bound on all i's, we infer that with probability at least $1/2$, all majority votes are correct and thus x is retreived correctly. Recall that the foregoing is conditioned on $\sigma^j = b(x, s^j)$ for every $j \in [\ell]$, which in turn holds with probability $2^{-\ell} = (m+1)^{-1} = \Omega(\varepsilon^2/n) = 1/\text{poly}(n)$, Thus, x is retreived correctly with probability $1/\text{poly}(n)$, and the theorem follows. □

Digest. Looking at the proof of Theorem 7.7, we note that it actually refers to a black-box $B_x(\cdot)$ that approximates $b(x, \cdot)$; specifically, in the case of Theorem 7.7 we used $B_x(r) \stackrel{\text{def}}{=} B(f(x), r)$. In particular, the proof does not use the fact that we can verify the correctness of the preimage recovered by the described process. Indeed, using the alternative procedure outlined in Footnote 5, the proof extends to establish *the existence of a* $\text{poly}(n/\varepsilon)$*-time oracle machine that, for every* $x \in \{0,1\}^n$*, given oracle access to any* $B_x : \{0,1\}^n \to \{0,1\}$ *satisfying*

$$\mathsf{Pr}_{r\in\{0,1\}^n}[B_x(r) = b(x, r)] \geq \frac{1}{2} + \varepsilon \tag{7.6}$$

outputs, with probability at least $1/2$*, a list of* n*-bit strings that includes* x. Noting that x is merely a string for which Eq. (7.6) holds, and that the procedure may get n and ε as inputs, we derive

Theorem 7.8 (Theorem 7.7, revisited): *There exists a probabilistic oracle machine that, given parameters* n, ε *and oracle access to any function* $B : \{0,1\}^n \to \{0,1\}$*, for every* $x \in \{0,1\}^n$*, given oracle access to any* B_x *halts after* $\text{poly}(n/\varepsilon)$ *steps and with probability at least* $1/2$ *outputs a list of all strings* $x \in \{0,1\}^n$ *that satisfy*

$$\mathsf{Pr}_{r\in\{0,1\}^n}[B(r) = b(x, r)] \geq \frac{1}{2} + \varepsilon,$$

where $b(x, r)$ *denotes the inner-product mod 2 of* x *and* r.

This machine can be modified such that, with high probability, its output list does not include any string x such that $\mathsf{Pr}_{r\in\{0,1\}^n}[B(r) = b(x, r)] < \frac{1}{2} + \frac{\varepsilon}{2}$. Theorem 7.8 can be viewed as a list decoding[7] procedure for the Hadamard Code, where the

[7] In contrast to standard decoding in which one recovers the unique information that is encoded in the codeword that is closest to the given string, in list decoding one recovers all strings having encoding that is at a specified distance from the given string. We mention that list decoding is applicable and valuable in the case that the specified distance does not allow for unique decoding and/or that the specified distance is greater than half the distance of the code. See further discussion in Section E.1.

Hadamard encoding of a string $x \in \{0,1\}^n$ is the 2^n-bit long string containing $b(x,r)$ for every $r \in \{0,1\}^n$.

Applications. Hard-core predicates play a central role in the construction of general-purpose pseudorandom generators (see Section 8.3), commitment schemes and zero-knowledge proofs (see Sections 9.2.2 and C.4.3), and encryption schemes (see Section C.5).

7.2 Hard Predicates in E

We start again with the assumption $\mathcal{P} \neq \mathcal{NP}$. In fact, we consider the seemingly stronger assumption by which $\mathcal{NP}$ cannot be solved by (non-uniform) families of polynomial-size circuits; that is, $\mathcal{NP}$ is not contained in $\mathcal{P}/\text{poly}$ (even not infinitely often). Our goal is to transform this worst-case assumption into an average-case condition, which is useful for our applications. Since the transformation will not yield a problem in $\mathcal{NP}$ but rather one in $\mathcal{E}$, we might as well take the weaker assumption (see Exercise 7.8). That is, our starting point is actually that *there exists an exponential-time solvable decision problem such that any family of polynomial-size circuit fails to solve it correctly on all but finitely many input lengths.*

Recall that our goal is to obtain a predicate (i.e., a decision problem) that is computable in exponential-time but is inapproximable by small circuits, where small may mean polynomial-size. For sake of later developments, we formulate a general notion of inapproximability.

Definition 7.9 (inapproximability, a general formulation): *We say that $f : \{0,1\}^* \to \{0,1\}$ is (S,ρ)-inapproximable if for every family of S-size circuits $\{C_n\}_{n\in\mathbb{N}}$ and all sufficiently large n it holds that*

$$\Pr[C(U_n) \neq f(U_n)] \geq \frac{\rho(n)}{2} \tag{7.7}$$

We say that f is T-inapproximable if it is $(T, 1-(1/T))$-inapproximable.

We chose the specific form of Eq. (7.7) such that the "level of inapproximability" represented by the parameter ρ will range in $(0,1)$ and increase with the value of ρ. Specifically, (almost-everywhere) worst case hardness for circuits of size S is represented by (S,ρ)-inapproximability with $\rho(n) = 2^{-n+1}$ (i.e., in this case $\Pr[C(U_n) \neq f(U_n)] \geq 2^{-n}$ for every circuit C_n of size $S(n)$), whereas no predicate can be (S,ρ)-inapproximability for $\rho(n) = 1 - O(2^{-n})$ even with $S(n) = O(n)$ (i.e., in this case $\Pr[C(U_n) = f(U_n)] \geq 0.5 + O(2^{-n})$ for some linear-size circuit; see Exercise 7.9). Indeed, Eq. (7.7) can be interpreted as an upper-bound on the *correlation* of each adequate circuit with f (i.e., $\mathsf{E}[\chi(C(U_n), f(U_n))] \leq 1 - \rho(n)$, where $\chi(\sigma,\tau) = 1$ if $\sigma = \tau$ and $\chi(\sigma,\tau) = -1$ otherwise). Thus, T-inapproximability means that no family of size T circuits can correlate f better than $1/T$.

Comments. Recall that $\mathcal{E}$ denote the class of exponential-time solvable decision problems (equivalently, exponential-time computable Boolean predicates); that is, $\mathcal{E} = \cup_\varepsilon \text{DTIME}(t_\varepsilon)$, where $t_\varepsilon(n) \stackrel{\text{def}}{=} 2^{\varepsilon n}$. We highlight the aforementioned term *almost everywhere*: Our starting point is not merely that $\mathcal{E}$ is not contained in $\mathcal{P}$/poly (or in other circuit size classes to be discussed), but rather that this is the case almost everywhere. Note that by saying that f has circuit complexity exceeding S, we merely mean that *there are infinitely many n's* such that no circuit of size $S(n)$ can computes f correctly on all inputs of length n. In contrast, by saying that f has circuit complexity exceeding S almost everywhere, we mean that *for all but finite many n's* no circuit of size $S(n)$ can computes f correctly on all inputs of length n.

7.2.1 Amplification wrt polynomial-size circuits

As hinted above, our goal is to prove the following result.

Theorem 7.10 *Suppose that for every polynomial p there exists a problem in $\mathcal{E}$ having circuit complexity that is almost-everywhere greater than p. Then there exist polynomial-inapproximable Boolean functions in $\mathcal{E}$; that is, for every polynomial p there exists a p-inapproximable Boolean function in $\mathcal{E}$.*

Theorem 7.10 is used towards deriving a meaningful derandomization of $\mathcal{BPP}$ under the aforementioned assumption (see Part 2 of Theorem 8.19). We present two proofs of Theorem 7.10. The first proof proceeds in two steps:

1. Starting from the worst-case hypothesis, we first establish some mild level of average-case hardness (i.e., a mild level of inapproximability). Specifically, we show that for every polynomial p there exists a problem in $\mathcal{E}$ that is (p, ε)-inapproximable for $\varepsilon(n) = 1/n^2$.

2. For any polynomial p, we prove that *if for every polynomial q the function f is $(q, 1/p)$-inapproximable, then the function $F(x_1, ..., x_{t(n)}) = \oplus_{i=1}^{t(n)} f(x_i)$, where $x_1, ..., x_{t(n)} \in \{0,1\}^n$ and $t(n) = n \cdot p(n)$, is T-inapproximable for any polynomial T.* This claim is known as Yao's XOR Lemma, and its proof is far more complex than the proof of its information theoretic analogue.

The second proof consists of showing that the construction employed in the first step, when composed with Theorem 7.8, actually yields the desired end result. This proof will uncover a connection between hardness amplification and coding theory (which is implicit at the end of Section D.4.2). Our presentation will thus proceed in three corresponding steps (presented in §7.2.1.1-7.2.1.3, and schematically depicted in Figure 7.2).

7.2.1.1 From worst-case hardness to mild average-case hardness

The construction is based on the self-correction paradigm (see also §9.3.2.1), to be reviewed first. The paradigm refers to functions g that can be evaluated at

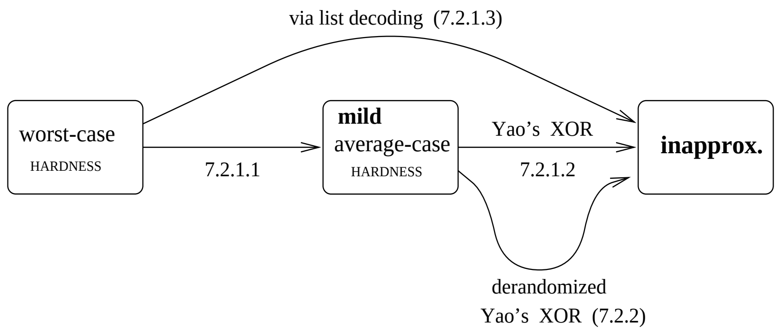


Figure 7.2: Proofs of hardness amplification: organization

any desired point by using the value of g at a few random points, where each of these points is uniformly distributed in the function's domain (but indeed the points are not independently distributed). The key observation is that if $g(x)$ can be reconstructed based on the value of g at t such random points, then such a reconstruction can tolerate a $1/3t$ fraction of errors (regarding the values of g). Thus, if we can correctly obtain the value of g on all but at most a $1/3t$ fraction of its domain, then we can probabilistically recover the correct value of g at any point with very high probability. It follows that if no probabilistic polynomial-time algorithm can correctly compute g *in the worst-case sense*, then every probabilistic polynomial-time algorithm must fail to correctly compute g *on at least a* $1/3t$ *fraction of its domain.*

The archetypical example of a self-correctable function is any m-variate polynomial of individual degree d over a finite field F such that $|F| > dm + 1$. The value of such a polynomial at any desired point x can be recovered based on the values of $dm + 1$ points (other than x) that reside on a random line that passes through x. Note that each of these points is uniformly distributed in F^m, which is the function's domain.

Recall that we are given an arbitrary function $f \in \mathcal{E}$ that is hard to compute in the worst-case. Needless to say, this function is not necessarily self-correctable (based on relatively few points), but it can be extended into such a function. Specifically, we extend $f : [N] \to \{0,1\}$ (viewed as $f : [N^{1/m}]^m \to \{0,1\}$) to an m-variate polynomial of individual degree d over a finite field F such that $|F| > dm+1$ and $(d+1)^m = N$. Intuitively, the extended function is at least as hard on the worst-case as f, and by self-correction the extended function must be mildly hard in the average-case. Details follow.

Construction 7.11 (multi-variate extension)[8]: *For any function* $f_n : \{0,1\}^n \to \{0,1\}$, *finite field* F, $H \subset F$ *and integer* m *such that* $|H|^m = 2^n$ *and* $|F| \geq m|H|$,

[8]The algebraic fact underlying this construction is that for any function $f : H^m \to F$ there exists a unique m-variate polynomial $\hat{f} : F^m \to F$ of individual degree $|H| - 1$ such that for every $x \in H^m$ it holds that $\hat{f}(x) = f(x)$. This polynomial is called a multi-variate polynomial extension of f, and it can be found in poly$(|H|^m \log |F|)$-time.

we consider the function $\hat{f}_n : F^m \to F$ *defined as the* m*-variate polynomial of individual degree* $|H|-1$ *that extends* $f_n : H^m \to \{0,1\}$*. That is, we identify* $\{0,1\}^n$ *with* H^m*, and define* $\hat{f}_n$ *as the unique* m*-variate polynomial of individual degree* $|H|-1$ *that satisfies* $\hat{f}_n(x) = f_n(x)$ *for every* $x \in H^m$*, where we view* $\{0,1\}$ *as a subset of* F*.*

Note that $\hat{f}_n$ can be evaluated at any desired point, by evaluating f_n on its entire domain, and determining the unique m-variate polynomial of individual degree $|H|-1$ that agrees with f_n on H^m. Thus, for $f : \{0,1\}^* \to \{0,1\}$ in $\mathcal{E}$, the corresponding $\hat{f}$ (defined by separately extending the restriction of f to each input length) is also in $\mathcal{E}$. For the sake of preserving various complexity measures, we wish to have $|F^m| = \text{poly}(2^n)$, which leads to setting $m = O(n/\log n)$ (yielding $|F| = \text{poly}(n)$), as in §9.3.2.2. In particular, in this case $\hat{f}_n$ is defined over strings of length $O(n)$. The mild average-case hardness of $\hat{f}$ follows by the forgoing discussion. In fact, we state and prove a more general result.

Theorem 7.12 *Suppose that there exists a Boolean function* f *in* $\mathcal{E}$ *having circuit complexity that is almost-everywhere greater than* S*. Then, there exists an exponential-time computable function* $\hat{f} : \{0,1\}^* \to \{0,1\}^*$ *such that* $|\hat{f}(x)| \le |x|$ *and for every family of circuit* $\{C'_{n'}\}_{n' \in \mathbb{N}}$ *of size* $S'(n') = S(n'/O(1))/\text{poly}(n')$ *it holds that* $\Pr[C'_{n'}(U_{n'}) \neq \hat{f}(U_{n'})] > (1/n')^2$*.*

Theorem 7.12 completes the first step of the proof of Theorem 7.10, except that we desire a Boolean function rather than one that does not stretch its input. The extra step (of obtaining a Boolean function that is $(\text{poly}(n), n^{-3})$-inapproximable) may be taken by considering the bits in the output of the function (see Exercise 7.10).[9] That is, if $\hat{f}$ is hard to compute on an $(1/n')^2$ fraction of the n'-bit long inputs then the Boolean predicate that returns an indicated bit of $\hat{f}(x)$ must be mildly inapproximable.

Proof: Given f as in the hypothesis and for every $n \in \mathbb{N}$, we consider the restriction of f to $\{0,1\}^n$, denoted f_n, and apply Construction 7.11 to it, while using $m = n/\log n$, $|H| = n$ and $n^2 < |F| = \text{poly}(n)$. Recall that the resulting function $\hat{f}_n$ maps strings of length $n' = \log_2 |F^m| = O(n)$ to strings of length $\log_2 |F| = O(\log n)$. Following the foregoing discussion, we note that by making $m|H| = o(n^2)$ oracle calls to any circuit $C'_{n'}$ that satisfies $\Pr[C'_{n'}(U_{n'}) = \hat{f}_n(U_{n'})] > 1-(1/n')^2 > 1-(1/3m|H|)$, we can probabilistically recover the value of ($\hat{f}_n$ and thus) f_n on each input, with probability at least $2/3$. Using amplification and derandomization as in the proof of Theorem 6.3, we obtain a circuit of size $n^3 \cdot |C'_{n'}|$ that computes f_n. By the hypothesis $n^3 \cdot |C'_{n'}| > S(n)$, and the theorem follows. ■

[9] A quantitatively stronger bound can be obtained by noting that the proof of Theorem 7.12 actually establishes an error lower-bound of $\Omega((\log n')/(n')^2)$ and that $|\hat{f}(x)| = O(\log |x|)$.

Digest. The proof of Theorem 7.12 is actually a worst-case to average-case reduction. That is, the proof consists of a self-correction procedure that allows for the evaluation of f at any desired point, using oracle calls to circuits that, for every n', compute $\hat{f}$ correctly on a $1-(1/n')^2$ fraction of the n'-bit long inputs. We note that if $f \in \mathcal{E}$ then $\hat{f} \in \mathcal{E}$, but we do not know how to preserve the complexity of f in case it is in $\mathcal{NP}$. (Various indications to the difficulty of a worst-case to average-case reduction for $\mathcal{NP}$ are known; see, e.g., [40].)

7.2.1.2 Yao's XOR Lemma

Having obtained a mildly inapproximable predicate, we wish to obtain a strongly inapproximable one. The information theoretic context provides an appealing suggestion: Suppose that X is a Boolean random variable (representing the mild inapproximability of the aforementioned predicate) that equals 1 with probability ε. Then XORing the outcome of n/ε independent samples of X yields a bit that equals 1 with probability $0.5 \pm \exp(-\Omega(n))$. It is tempting to think that the same should happen in the computational setting. That is, if f is hard to approximate correctly with probability exceeding $1-\varepsilon$ then XORing the output of f on n/ε non-overlapping parts of the input should yield a predicate that is hard to approximate correctly with probability that is non-negligibly higher than $1/2$. The latter assertion turns out to be correct, but (as in Section 7.1.2) the proof of the computational phenomenon is considerably more complex than the analysis of the information theoretic analogue.

Theorem 7.13 (Yao's XOR Lemma): *Let p be a polynomial and suppose that the Boolean function f is $(T, 1/p)$-inapproximable, for every polynomial T. Then the function $F(x_1, ..., x_{t(n)}) = \oplus_{i=1}^{t(n)} f(x_i)$, where $x_1, ..., x_{t(n)} \in \{0,1\}^n$ and $t(n) = n \cdot p(n)$, is T'-inapproximable for every polynomial T'.*

Combining Theorems 7.12 and 7.13 (and Exercise 7.10), we complete the (first) proof of Theorem 7.10. Several different proofs of Theorem 7.13 are known. We choose using a proof that benefits most from the material already presented (in Section 7.1). This proof proceeds in two steps: First we prove that the corresponding "direct product" function $P(x_1, ..., x_{t(n)}) = (f(x_1), ..., f(x_{t(n)}))$ is difficult to compute in a strong average-case sense, and next we establish the desired result by an application of Theorem 7.8. In fact, the first step is the main one, and we believe that it is of independent interest (and thus generalize it from Boolean functions to arbitrary ones).

Theorem 7.14 (The Direct Product Lemma): *Let p be a polynomial and $f : \{0,1\}^* \to \{0,1\}^*$. Suppose that for every family of polynomial-size circuits, $\{C_n\}_{n\in\mathbb{N}}$, and all sufficiently large $n \in \mathbb{N}$, it holds that $\Pr[C_n(U_n) \neq f(U_n)] > 1/p(n)$. Let $P(x_1, ..., x_{t(n)}) = (f(x_1), ..., f(x_{t(n)}))$, where $x_1, ..., x_{t(n)} \in \{0,1\}^n$ and $t(n) = n \cdot p(n)$. Then, for every family of polynomial-size circuits, $\{C'_m\}_{m\in\mathbb{N}}$, it holds that $\Pr[C'_m(U_m) = P(U_m)] < \mu(m)$, where μ is a negligible function.*

Theorem 7.13 follows from Theorem 7.14 by considering the function $P'(x_1,...,x_{t(n)},r) = b(f(x_1)\cdots f(x_{t(n)}),r)$, where f is a Boolean function, $r \in \{0,1\}^{t(n)}$, and $b(y,r)$ is the inner-product modulo 2 of the $t(n)$-bit long strings y and r. Applying Theorem 7.8, we infer that P' is T'-inapproximable for every polynomial T'. Lastly, we reduce the approximation of P' to the approximation of F (see Exercise 7.11), and Theorem 7.13 follows.

Proof of Theorem 7.14. As in the proof of Theorem 7.5, we show how to converts circuits that violate the theorem's conclusion into circuits that violate the theorem's hypothesis. We note, however, that things were much simpler in the context of Theorem 7.5: There we could (efficiently) check whether or not a value contained in the output of the circuit that solves the direct-product problem constitutes a correct answer for the corresponding instance of the basic problem. Lacking such an ability in the current context, we shall have to use such values more carefully. Loosely speaking, we will take a weighted majority vote among various answers, where the weights reflect our confidence in the correctness of the various answers.

We derive Theorem 7.14 by applying the following lemma that provides quantitative bounds on the feasibility of computing the direct product of two functions. In this lemma, $\{Y_m\}_{m\in\mathbb{N}}$ and $\{Z_m\}_{m\in\mathbb{N}}$ are independent probability ensembles such that $Y_m, Z_m \in \{0,1\}^m$, and $X_n = (Y_{\ell(n)}, Z_{n-\ell(n)})$ for some function $\ell : \mathbb{N} \to \mathbb{N}$. The lemma refers to the success probability of computing the direct product function $F : \{0,1\}^* \to \{0,1\}^*$ defined by $F(yz) = (F_1(y), F_2(z))$, where $|y| = \ell(|yz|)$, when given bounds on the success probability of computing F_1 and F_2 (separately). Needless to say, these probability bounds refer to circuits of certain sizes. We stress that the statement of the lemma is not symmetric with respect to the two functions, guaranteeing a stronger (and in fact lossless) preservation of circuit sizes for one of the functions (which is arbitrarily chosen to be F_1).

Lemma 7.15 (Direct Product, a quantitative two argument version): *For $\{Y_m\}$, $\{Z_m\}$, F_1, F_2, ℓ, $\{X_n\}$ and F as in the foregoing, let $\rho_1(\cdot)$ be an upper-bound on the success probability of $s_1(\cdot)$-size circuits in computing F_1 over $\{Y_m\}$. That is, for every such circuit family $\{C_m\}$*

$$\Pr[C_m(Y_m) = F_1(Y_m)] \leq \rho_1(m)$$

Likewise, suppose that $\rho_2(\cdot)$ is an upper-bound on the probability that $s_2(\cdot)$-size circuits compute F_2 over $\{Z_m\}$. Then, for every function $\varepsilon : \mathbb{N} \to \mathbb{R}$, the function ρ defined as

$$\rho(n) \stackrel{\text{def}}{=} \rho_1(\ell(n)) \cdot \rho_2(n - \ell(n)) + \varepsilon(n)$$

is an upper-bound on the probability that families of $s(\cdot)$-size circuits correctly compute F over $\{X_n\}$, where

$$s(n) \stackrel{\text{def}}{=} \min\left\{ s_1(\ell(n)) \;,\; \frac{s_2(n-\ell(n))}{\text{poly}(n/\varepsilon(n))} \right\}$$

Theorem 7.14 is derived from Lemma 7.15 by using careful induction, which capitalizes on the asymmetry of Lemma 7.15. Specifically, we write $P(x_1, x_2, ..., x_{t(n)})$ as $P^{(t(n))}(x_1, x_2, ..., x_{t(n)})$, where $P^{(i)}(x_1, ..., x_i) = (f(x_1), ..., f(x_i))$ and $P^{(i)}(x_1, ..., x_i) \equiv (P^{(i-1)}(x_1, ..., x_{i-1}), f(x_i))$. For every polynomial s and a noticeable function ε (i.e., $\varepsilon(n) > 1/p(n)$ for some positive polynomial p), we prove by induction on i that circuits of size $s(n)$ cannot compute $P^{(i)}(U_{i \cdot n})$ with success probability greater than $(1 - (1/p(n))^i + i \cdot \varepsilon(n)$. (The induction basis is guaranteed by the theorem's hypothesis.) The induction step is proved using Lemma 7.15 with $F_1 = P^{(i-1)}$ and $F_2 = f$ (along with $\rho_1((i-1)n) = (1-(1/p(n))^{i-1} + (i-1) \cdot \varepsilon(n)$, $s_1((i-1)n) = s(n)$, $\rho_2(n) = 1 - (1/p(n))$ and $s_2(n) = \text{poly}(n/\varepsilon(n)) \cdot s(n))$. In particular, we use again the theorem's hypothesis regarding f, and note that $((1 - (1/p(n))^{i-1} + (i-1) \cdot \varepsilon(n)) \cdot (1 - (1/p(n)) + \varepsilon(n)$ is upper-bounded by $(1-(1/p(n))^i + i \cdot \varepsilon(n)$. Thus, no $s(n)$-size circuit can compute $P^{(t(n))}(U_{t(n) \cdot n})$ with success probability greater than $(1-(1/p(n))^{t(n)} + t(n) \cdot \varepsilon(n) = \exp(-n) + t(n) \cdot \varepsilon(n)$.

Proof of Lemma 7.15: Proceeding (as usual) by the contrapositive, we consider a family of $s(\cdot)$-size circuits $\{C_n\}_{n\in\mathbb{N}}$ that violates the lemma's conclusion; that is, $\mathsf{Pr}[C_n(X_n) = F(X_n)] > \rho(n)$. We will show how to use such circuits in order to obtain either circuits that violate the lemma's hypothesis regarding F_1 or circuits that violate the lemma's hypothesis regarding F_2. Towards this end, it is instructive to write the success probability of C_n in a conditional form, while denoting the i^{th} output of $C_n(x)$ by $C_n(x)_i$ (i.e., $C_n(x) = (C_n(x)_1, C_n(x)_2)$):

$$\begin{aligned}
&\mathsf{Pr}[C_n(Y_{\ell(n)}, Z_{n-\ell(n)}) = F(Y_{\ell(n)}, Z_{n-\ell(n)})] \\
&\quad = \mathsf{Pr}[C_n(Y_{\ell(n)}, Z_{n-\ell(n)})_1 = F_1(Y_{\ell(n)})] \\
&\qquad \cdot \mathsf{Pr}[C_n(Y_{\ell(n)}, Z_{n-\ell(n)})_2 = F_2(Z_{n-\ell(n)}) \mid C_n(Y_{\ell(n)}, Z_{n-\ell(n)})_1 = F_1(Y_{\ell(n)})].
\end{aligned}$$

The basic idea is that if the first factor is greater than $\rho_1(\ell(n))$ then we derive a circuit contradicting the lemma's hypothesis regarding F_1, whereas if the second factor is significantly greater than $\rho_2(n-\ell(n))$ then we derive a circuit contradicting the lemma's hypothesis regarding F_2. The basic idea for the latter case is that a sufficiently large sample of $(Y_{\ell(n)}, F_1(Y_{\ell(n)}))$, which may be hard-wired into the circuit, allows using the conditional probability space (in such a circuit) towards an attempt to approximate F_2. This may work provided the condition holds with noticeable probability. The last caveat motivates a separate treatment of z's with noticeable $\mathsf{Pr}[C_n(Y_{\ell(n)}, z)_1 = F_1(Y_{\ell(n)})]$ and of the rest.

Let us first simplify the notations by fixing a generic n and using the abbreviations $C = C_n$, $\varepsilon = \varepsilon(n)$, $\ell = \ell(n)$, $Y = Y_\ell$, and $Z = Y_{n-\ell}$. We call z good if $\mathsf{Pr}[C(Y, z)_1 = F_1(Y)] \geq \varepsilon/2$ and let G be the set of good z's. Then, we upper-bound the success probability of C by $\mathsf{Pr}[C(Y, Z) = F(Y, Z) \land Z \in G] + \varepsilon/2$, where the bound follows by observing that for any $z \notin G$:

$$\mathsf{Pr}[C(Y, z) = F(Y, z)] \leq \mathsf{Pr}[C(Y, z)_1 = F_1(Y)] < \varepsilon/2\,.$$

Thus, using $\mathsf{Pr}[C(Y, z) = F(Y, z)] > \rho(n) = \rho_1(\ell) \cdot \rho_2(n - \ell) + \varepsilon$, we have

$$\mathsf{Pr}[C(Y, Z) = F(Y, Z) \land Z \in G] > \rho_1(\ell) \cdot \rho_2(n - \ell) + \frac{\varepsilon}{2}. \tag{7.8}$$

We proceed according to the forgoing outline, first showing that if $\mathsf{Pr}[C(Y,Z)_1 = F_1(Y)] > \rho_1(\ell)$ then we derive circuits violating the hypothesis concerning F_2. Actually, we prove something stronger (which we will actually need for the other case).

Claim 7.15.1: For every z, it holds that $\mathsf{Pr}[C(Y,z)_1 = F_1(Y)] \leq \rho_1(\ell)$.

Proof: Otherwise, using any $z \in \{0,1\}^{n-\ell}$ that satisfies $\mathsf{Pr}[C(Y,z)_1 = F_1(Y)] > \rho_1(\ell)$, we obtain a circuit $C'(y) \stackrel{\text{def}}{=} C(y,z)_1$ that contradicts the lemma's hypothesis concerning F_1. □

Using Claim 7.15.1, we show how to obtain a circuit that violates the lemma's hypothesis concerning F_2, and doing so we complete the proof of the lemma.

Claim 7.15.2: There exists a circuit C'' of size $s_2(n-\ell)$ such that

$$\begin{aligned}\mathsf{Pr}[C''(Z) = F_2(Z)] &\geq \frac{\mathsf{Pr}[C(Y,Z) = F(Y,Z) \wedge Z \in G]}{\rho_1(\ell)} - \frac{\varepsilon}{2} \\ &> \rho_2(n-\ell)\end{aligned}$$

Proof: The second inequality is due to Eq. (7.8), and thus we focus on establishing the first one. We construct the circuit C'' as suggested in the foregoing outline. Specifically, we take a $\text{poly}(n/\varepsilon)$-large sample, denoted S, from the distribution $(Y, F_1(Y))$ and let $C''(z) \stackrel{\text{def}}{=} C(y,z)_2$, where (y,v) is a uniformly selected among the elements of S for which $C(y,z)_1 = v$ holds. Details follow.

Let S be a sequence of $m \stackrel{\text{def}}{=} \text{poly}(n/\varepsilon)$ pairs, generated by taking m independent samples from the distribution $(Y, F_1(Y))$. We stress that we do not assume here that such a sample can be produced by an efficient (uniform) algorithm (but, jumping ahead, we remark that such a sequence can be fixed non-uniformly). For each $z \in G \subseteq \{0,1\}^{n-\ell}$, we denote by S_z the set of pairs $(y,v) \in S$ for which $C(y,z)_1 = v$. Note that S_z is a random sample for the residual probability space defined by $(Y, F_1(Y))$ conditioned on $C(Y,z)_1 = F_1(Y)$. Also, with overwhelmingly high probability, $|S_z| = \Omega(n/\varepsilon^2)$, because $z \in G$ implies $\mathsf{Pr}[C(Y,z)_1 = F_1(Y)] \geq \varepsilon/2$ and $m = \Omega(n^2/\varepsilon^3)$. Thus, for each $z \in G$, with overwhelming probability taken over the choices of S, the sample S_z provides a good approximation to the conditional probability space. In particular, with probability greater than $1 - 2^{-n}$, it holds that

$$\frac{|\{(y,v) \in S_z : C(y,z)_2 = F_2(z)\}|}{|S_z|} \geq \mathsf{Pr}[C(Y,z)_2 = F_2(z) \,|\, C(Y,z)_1 = F_1(Y)] - \frac{\varepsilon}{2} \tag{7.9}$$

Thus, with positive probability, Eq. (7.9) holds for all $z \in G \subseteq \{0,1\}^{n-\ell}$. The circuit C'' computing F_2 is now defined as follows. A set $S = \{(y_i, v_i) : i = 1, ..., m\}$ satisfying Eq. (7.9) for all good z's is "hard-wired" into the circuit C''. (In particular, S_z is not empty for any good z.) On input z, the circuit C'' first determines the set S_z, by running C for m times and checking, for each $i = 1, ..., m$, whether or not $C(y_i, z) = v_i$. In case S_z is empty, the circuit returns an arbitrary value. Otherwise, the circuit selects uniformly a pair $(y,v) \in S_z$ and outputs

$C(y,z)_2$. (The latter random choice can be eliminated by a standard averaging argument.) Using the definition of C'', Eq. (7.9), and Claim 7.15.1, we have:

$$\begin{aligned}
\Pr[C''(Z)=F_2(Z)] &\geq \sum_{z\in G} \Pr[Z=z]\cdot \Pr[C''(z)=F_2(z)] \\
&= \sum_{z\in G} \Pr[Z=z]\cdot \frac{|\{(y,v)\in S_z : C(y,z)_2=F_2(z)\}|}{|S_z|} \\
&\geq \sum_{z\in G} \Pr[Z=z]\cdot \left(\Pr[C(Y,z)_2=F_2(z)\,|\,C(Y,z)_1=F_1(Y)] - \frac{\varepsilon}{2}\right) \\
&= \sum_{z\in G} \Pr[Z=z]\cdot \left(\frac{\Pr[C(Y,z)_2=F_2(z)\wedge C(Y,z)_1=F_1(Y)]}{\Pr[C(Y,z)_1=F_1(Y)]} - \frac{\varepsilon}{2}\right) \\
&\geq \left(\sum_{z\in G} \Pr[Z=z]\cdot \frac{\Pr[C(Y,z)=F(Y,z)]}{\rho_1(\ell)}\right) - \frac{\varepsilon}{2}
\end{aligned}$$

where the last inequality is due to Claim 7.15.1. The claim follows. □

This completes the proof of the lemma. ■

Comments. Firstly, we wish to call attention to the care with which an inductive argument needs to be carried out in the computational setting, especially when a non-constant number of inductive steps is concerned. Indeed, our inductive proof of Theorem 7.14 involves invoking a quantitative lemma that allows to keep track of the relevant quantities (e.g., success probability and circuit size) throughout the induction process. Secondly, we mention that Lemma 7.15 (as well as Theorem 7.14) has a uniform complexity version that assumes that one can efficiently sample the distribution $(Y_{\ell(n)}, F_1(Y_{\ell(n)}))$ (resp., $(U_n, f(U_n))$). For details see [98]. Indeed, a good lesson from the proof of Lemma 7.15 is that non-uniform circuits can "effectively sample" any distribution. Lastly, we mention that Theorem 7.5 (the amplification of one-way functions) and Theorem 7.13 (Yao's XOR Lemma) also have (tight) quantitative versions (see, e.g., [87, Sec. 2.3.2] and [98, Sec. 3], respectively).

7.2.1.3 List decoding and hardness amplification

In this subsection we provide an alternative proof of Theorem 7.10, showing that (for a suitable choice of parameters) combining Construction 7.11 with the inner-product construction (of Theorem 7.8) yields the desired result. Specifically, we claim the following.

Proposition 7.16 *Suppose that there exists a Boolean function f in $\mathcal{E}$ having circuit complexity that is almost-everywhere greater than S, and let $\varepsilon:\mathbb{N}\to[0,1]$ satisfying $\varepsilon(n) > n/2^n$. Let f_n be the restriction of f to $\{0,1\}^n$, and let $\hat{f}_n$ be the function obtained from f_n when applying Construction 7.11 with $|H| = n/\varepsilon(n)$ and $|F| = |H|^3$. Then, the function $\hat{f}:\{0,1\}^*\to\{0,1\}^*$, defined by $\hat{f}(x) = \hat{f}_{|x|/3}(x)$,* *is computable in exponential-time and for every family of circuit* $\{C'_{n'}\}_{n'\in\mathbb{N}}$ *of size* $S'(n') = \text{poly}(\varepsilon(n'/3)/n') \cdot S(n'/3)$ *it holds that* $\Pr[C'_{n'}(U_{n'}) = \hat{f}(U_{n'})] < \varepsilon'(n') \stackrel{\text{def}}{=} \varepsilon(n'/3)$.

In particular, for some $\gamma > 0$, Proposition 7.16 yields an exponential-time computable function $\hat{f}$ such that $|\hat{f}(x)| \leq |x|$ and for every family of circuit $\{C'_{n'}\}_{n'\in\mathbb{N}}$ of size $S'(n') = S(n'/3)^\gamma/\text{poly}(n')$ it holds that $\Pr[C'_{n'}(U_{n'}) = \hat{f}(U_{n'})] < 1/S'(n')$. Combining this with Theorem 7.8, we infer that $P(x,r) = b(\hat{f}(x), r)$, where $|r| = |\hat{f}(x)| \leq |x|$, is S''-inapproximable for $S''(n'') = S(n''/2)^{\Omega(1)}/\text{poly}(n'')$. In particular, for every polynomial p, we obtain a p-inapproximable predicate in $\mathcal{E}$ by applying the foregoing with $S(n) = \text{poly}(n, p(n))$. Thus, Theorem 7.10 follows.

Proposition 7.16 is proven by observing that the transformation of f_n to $\hat{f}_n$ constitutes a "good" code (see §E.1.1.4) and that any such code provides a worst-case to (strongly) average-case reduction. We start by defining the class of codes that suffices for the latter reduction, while noting that the code underlying the mapping $f_n \mapsto \hat{f}_n$ is actually stronger than needed.

Definition 7.17 (efficient codes supporting implicit decoding): *For fixed functions* $q, \ell : \mathbb{N} \to \mathbb{N}$ *and* $\alpha : \mathbb{N} \to [0,1]$*, the mapping* $\Gamma : \{0,1\}^* \to \{0,1\}^*$ *is* efficient and supports implicit decoding with parameters q, ℓ, α *if it satisfies the following two conditions:*

1. Encoding: *The mapping* Γ *is polynomial-time computable.*

   It is instructive to view Γ *as mapping* N*-bit long strings to sequences of length* $\ell(N)$ *over* $[q(N)]$*, and to view* $\Gamma(x) \in [q(|x|)]^{\ell(|x|)}$ *as a mapping from* $[\ell(|x|)]$ *to* $[q(|x|)]$.

2. Decoding: *There exists a polynomial* p *such that the following holds. For every* $w : [\ell(N)] \to [q(N)]$ *and* $x \in \{0,1\}^N$ *such that* $\Gamma(x)$ *is* $(1-\alpha(N))$*-close to* w*, there exists an oracle-aided*[10] *circuit* C *of size* $p((\log N)/\alpha(N))$ *such that, for every* $i \in [N]$*, it holds that* $C^w(i)$ *equals the* i^{th} *bit of* x.

The encoding condition implies that ℓ is polynomially bounded. The decoding condition refers to any Γ-codeword that agrees with trhe oracle $w : [\ell(N)] \to [q(N)]$ on an $\alpha(N)$ fraction of the $\ell(N)$ coordinates, *where* $\alpha(N)$ *may be very small.* We highlight the non-triviality of the decoding condition: There are N bits of information in x, while the size of the circuit C is only $p((\log N)/\alpha(N))$ and yet C should be able to recover any desired entry of x by making only $p((\log N)/\alpha(N))$ queries to w, which may be a highly corrupted version of $\Gamma(x)$. On the other hand, the decoding condition does not refer to the complexity of obtaining the aforementioned oracle-aided circuits.

We mention that the transformation of f_n to $\hat{f}_n$ underlying Proposition 7.16 is efficient and supports implicit decoding with parameters q, ℓ, α such that $\ell(2^n) =$

[10]Oracle-aided are defined analogously to oracle Turing machines. Alternatively, we may consider here oracle machines that take advice such that both the advice length and the machine's running time are upper-bounded by $p((\log N)/\alpha(N))$. We comment that we potentially consider also non-binary oracles, which return elements in $[q(N)]$.

$\ell(|\langle f_n\rangle|) = |\langle f_n\rangle|^3 = 2^{3n}$, $\alpha(2^n) = \varepsilon(n)$, and $q(2^n) = (n/\alpha(2^n))^3$. Furthermore, there are at most $O(1/\alpha(2^n)^2)$ codewords (i.e., $\hat{f}_n$'s) that are $(1-\alpha(2^n))$-close to any fixed $w : [\ell(2^n)] \to [q(2^n)]$, and the corresponding oracle-aided circuits can be constructed in probabilistic $p(n/\alpha(2^n))$-time.[11] These results are termed "list decoding" (with implicit representations). We stress that all these facts are highly non-trivial, but beyond the scope of the current text (and the interested reader is referred to [204]). Our focus is on showing that such efficient codes that supports implicit decoding suffice for worst-case to (strongly) average-case reductions.

Theorem 7.18 *Suppose that there exists a Boolean function f in $\mathcal{E}$ having circuit complexity that is almost-everywhere greater than S, and let $\varepsilon : \mathbb{N} \to [0,1]$. Consider $\ell : \mathbb{N} \to \mathbb{N}$ such that $n \mapsto \log_2 \ell(2^n)$ is a 1-1 map of the integers, and let $m(n) = \log_2 \ell(2^n)$. Suppose that the mapping $\Gamma : \{0,1\}^* \to \{0,1\}^*$ is efficient and supports implicit decoding with parameters q, ℓ, α such that $\alpha(N) = \varepsilon(\lfloor \log_2 N \rfloor)$. Define $g_n : [\ell(2^n)] \to [q(2^n)]$ such that $g_n(i) = \Gamma(\langle f_n\rangle)(i)$, where $\langle f_n\rangle$ denotes the 2^n-bit long description of the truth-table of f_n. Then, the function $g : \{0,1\}^* \to \{0,1\}^*$, defined by $g(z) = g_{m^{-1}(|z|)}(z)$, is computable in exponential-time and for every family of circuit $\{C'_{n'}\}_{n'\in\mathbb{N}}$ of size $S'(n') = \mathrm{poly}(\varepsilon(m^{-1}(n'))/n') \cdot S(m^{-1}(n'))$ it holds that $\Pr[C'_{n'}(U_{n'}) = g(U_{n'})] < \varepsilon'(n') \stackrel{\text{def}}{=} \varepsilon(m^{-1}(n'))$.*

Proof Sketch: First note that we can generate the truth-table of f_n in exponential-time, and by the encoding condition of Γ it follows that g_n can be evaluated in exponential-time. Regarding g's average-case hardness, consider a circuit $C' = C'_{n'}$ violating the conclusion of the theorem, let $n = m^{-1}(n')$, and recall that $\varepsilon'(n') = \varepsilon(n) = \alpha(2^n)$. Then, C' is $(1-\alpha(2^n))$-close to $g_n = \Gamma(\langle f_n\rangle)$, and the decoding condition of Γ asserts that we can recover each bit of $\langle f_n\rangle$ (i.e., evaluate f_n) by a circuit of size $p(n/\alpha(2^n)) \cdot S'(n') < S(n)$, in contradiction to the hypothesis. □

Comment. For simplicity, we formulated Definition 7.17 in a crude manner that suffices for the foregoing application. A more careful formulation of the decoding condition refers to codewords that are $(1-\alpha(N))$-close to the oracle $w : [\ell(N)] \to [q(N)]$ rather than $(1 - ((1/q(N)) + \alpha(N)))$-close to it.[12] Needless to say, the difference is insignificant in the case that $\alpha(N) \gg 1/q(N)$ (as in Proposition 7.16, where we used $q(N) = ((\log_2 N)/\alpha(N))^3$), but it is significant in case we care about binary codes (i.e., $q(N) = 2$, or codes over other small alphabets). We mention

[11] Needless to say, the construction may yield also oracle-aided circuits that compute the decoding of codewords that are almost $(1-\alpha(2^n))$-close to w. That is, there exists a probabilistic $p(n/\alpha(2^n))$-time algorithm that outputs a list of circuits that, with high probability, contains an oracle-aided circuit for the decoding of each codeword that is $(1-\alpha(2^n))$-close to w. Furthermore, with high probability, the list contains only circuits that decode codewords that are $(1-\alpha(2^n)/2)$-close to w.

[12] Note that this is the "right" formulation, because a random $\ell(N)$-sequence over $[q(N)]$ is expected to be $(1-(1/q(N)))$-close to any fixed codeword, and with overwhelmingly high probability it will be $(1-((1-o(1))/q(N)))$-close to almost all the codewords, provided $\ell(N) \gg q(n)^2$. But in case $N \gg \log q(N)$, we cannot hope to recover almost all N-bit long strings based on $\mathrm{poly}(q(N)\log N)$ bits of advice.

that Theorem 7.18 can be adapted to this context, and directly yields strongly inapproximable predicates. For details, see Exercise 7.12.

7.2.2 Amplification wrt exponential-size circuits

For the purpose of stronger derandomization of $\mathcal{BPP}$, we start with a stronger assumption regarding the worst-case circuit complexity of $\mathcal{E}$ and turn it to a stronger inapproximability result.

Theorem 7.19 *Suppose that there exists a decision problem $L \in \mathcal{E}$ having almost-everywhere exponential circuit complexity; that is, there exists a constant $b > 0$ such that, for all but finitely many n's, any circuit that correctly decides L on $\{0,1\}^n$ has size at least 2^{bk}. Then, for some constant $c > 0$ and $T(n) \stackrel{\text{def}}{=} 2^{c \cdot n}$, there exists a T-inapproximable Boolean function in $\mathcal{E}$.*

Theorem 7.19 can be used for deriving a full derandomization of $\mathcal{BPP}$ (i.e., $\mathcal{BPP} = \mathcal{P}$) under the aforementioned assumption (see Part 1 of Theorem 8.19).

Theorem 7.19 follows as a special case of Proposition 7.16 (and the modification discussed right after it). An alternative proof, which uses different ideas that are of independent interest, will be briefly reviewed next. The starting point of this proof is a mildly inapproximable predicate, as provided by Theorem 7.12. However, here we cannot afford to apply Yao's XOR Lemma (i.e., Theorem 7.13), because the latter relates the circuit complexity of a *strongly* inapproximable predicate defined over poly(n)-bit long strings to the circuit complexity of a *mildly* inapproximable predicate defined over n-bit long strings. That is, if $f : \{0,1\}^n \to \{0,1\}$ is mildly inapproximable by S_f-size circuits then $F : \{0,1\}^{\text{poly}(n)} \to \{0,1\}$ is strongly inapproximable by S_F-size circuits, for some $S_F(\text{poly}(n))$ that is polynomially related to $S_f(n)$. In particular, $S_F(\text{poly}(n)) < S_f(n)$ seems inherent in this reasoning. For the case of polynomial lower-bounds, this is good enough (i.e., if S_f can be an arbitrarily large polynomial then so can S_F), but for $S_F(n) = \exp(\Omega(n))$ we cannot obtain $S_F(m) = \exp(\Omega(m))$ (but rather only $S_F(m) = \exp(m^{\Omega(1)})$).

The source of trouble is that amplification of inapproximability was achieved by taking a polynomial number of independent instances. Indeed, we cannot hope to amplify hardness without applying f on many instances, but these instances need not be independent. Thus, the idea is to define $F(r) = \oplus_{i=1}^{\text{poly}(n)} f(x_i)$, where $x_1, ..., x_{\text{poly}(n)} \in \{0,1\}^n$ are generated from r and still $|r| = O(n)$. That is, we seek a "derandomized" version of Yao's XOR Lemma. In other words, we seek a "pseudorandom generator" of a type appropriate for expanding r to dependent x_i's such that the XOR of the $f(x_i)$'s is as inapproximable as it would have been for independent x_i's.[13]

[13]Indeed, this falls within the general paradigm discussed in Section 8.2. Furthermore, this suggestion provides another perspective on the connection between randomness and computational difficulty, which is the focus of much discussion in Chapter 8 (see, e.g., §8.3.7.2).

Teaching note: In continuation to Footnote 13, we note that there is a strong connection between the rest of this section and Chapter 8. On top of the aforementioned conceptual aspects, we will refer to pairwise independence generators (see Section 8.6.1), random walks on expanders (see Section 8.6.3), and even to the Nisan-Wigderson Construction (Construction 8.17).

The pivot of the proof is the notion of a hard region. Loosely speaking, S is a hard region of a Boolean function f if f is *strongly inapproximable on a random input in* S; that is, for every (relatively) small circuit C_n, it holds that $\Pr[C_n(U_n) = f(U_n)|U_n \in S] \approx 1/2$. By definition, $\{0,1\}^*$ is a hard region of any *strongly* inapproximable predicate. One important (and non-trivial) observation is that any *mildly* inapproximable predicate has a hard region of density related to its inapproximability parameter. Loosely speaking, hardness amplification will proceed via methods for generating related instances that hit the hard region with sufficiently high probability. But, first let us study the notion of a hard region.

7.2.2.1 Hard regions

We actually generalize the notion of hard regions to arbitrary distributions. The important special case of uniform distributions is obtained by taking X_n to be U_n (i.e., the uniform distribution over $\{0,1\}^n$). In general, we only assume that $X_n \in \{0,1\}^n$.

Definition 7.20 (hard region relative to arbitrary distribution): *Let* $f:\{0,1\}^* \to \{0,1\}$ *be a Boolean predicate,* $\{X_n\}$ *be a probability ensemble,* $s:\mathbb{N}\to\mathbb{N}$ *and* $\varepsilon:\mathbb{N}\to[0,1]$.

- *We say that a set* S *is a* hard region of f relative to $\{X_n\}$ *with respect to* $s(\cdot)$*-size circuits and advantage* $\varepsilon(\cdot)$ *if for every* n *and every circuit* C_n *of size at most* $s(n)$*, it holds that*

$$\Pr[C_n(X_n)=f(X_n)|X_n\in S] \le \frac{1}{2} + \varepsilon(n).$$

- *We say that* f *has a* hard region of density $\rho(\cdot)$ *relative to* $\{X_n\}$ *with respect to* $s(\cdot)$*-size circuits and advantage* $\varepsilon(\cdot)$ *if there exists a set* S *that is a hard region of* f *relative to* $\{X_n\}$ *with respect to the above such that* $\Pr[X_n\in S_n] \ge \rho(n)$.

Note that a Boolean function f is $(s, 1-2\varepsilon)$-inapproximable if and only if $\{0,1\}^*$ is a hard region of f relative to $\{U_n\}$ with respect to $s(\cdot)$-size circuits and advantage $\varepsilon(\cdot)$. Thus, *strongly* inapproximable predicates (e.g., S-inapproximable predicates for super-polynomial S) have a hard region of density 1 (with respect to a negligible advantage). But this trivial observation does not provide hard regions (with small advantage) for *mildly* inapproximable predicates. Providing such hard regions is the contents of the following theorem.

Theorem 7.21 (hard regions for mildly inapproximable predicates): *Let* $f:\{0,1\}^* \to \{0,1\}$ *be a Boolean predicate,* $\{X_n\}$ *be a probability ensemble,* $s:\mathbb{N}\to\mathbb{N}$*, and*

$\rho:\mathbb{N}\to[0,1]$ such that $\rho(n)>1/\mathrm{poly}(n)$. Suppose that, for every circuit C_n of size at most $s(n)$, it holds that $\mathsf{Pr}[C_n(X_n)=f(X_n)]\le 1-\rho(n)$. Then, for every $\varepsilon:\mathbb{N}\to[0,1]$, the function f has a hard region of density $\rho'(\cdot)$ relative to $\{X_n\}$ with respect to $s'(\cdot)$-size circuits and advantage $\varepsilon(\cdot)$, where $\rho'(n)\stackrel{\rm def}{=}(1-o(1))\cdot\rho(n)$ and $s'(n)\stackrel{\rm def}{=}s(n)/\mathrm{poly}(n/\varepsilon(n))$.

In particular, if f is $(s,2\rho)$-inapproximable then f has a hard region of density $\rho'(\cdot)$ relative to the uniform distribution (with respect to $s'(\cdot)$-size circuits and advantage $\varepsilon(\cdot)$).

Proof Sketch:[14] We start by proving a weaker statement; namely, that $\{X_n\}$ "dominates" an ensemble $\{Y_n\}$ such that f is strongly inapproximable on $\{Y_n\}$. For $\rho:\mathbb{N}\to[0,1]$, we say that $\{X_n\}$ ρ-dominates $\{Y_n\}$ if for every x it holds that $\mathsf{Pr}[X_n=x]\ge\rho(n)\cdot\mathsf{Pr}[Y_n=x]$. Fixing the function ρ (to the one provided by the theorem's hypothesis), in this case we also say that $\{Y_n\}$ is dominated by $\{X_n\}$. We say that $\{Y_n\}$ is critically dominated by $\{X_n\}$ if for every x either $\mathsf{Pr}[Y_n=x]=(1/\rho(n))\cdot\mathsf{Pr}[X_n=x]$ or $\mathsf{Pr}[Y_n=x]=0$.

The notion of domination and critical domination play a central role in the proof, which consists of two parts. In the first part (Claim 7.21.1), we prove the existence of a ensemble $\{Y_n\}$ that is dominated by $\{X_n\}$ such that f is strongly inapproximable on $\{Y_n\}$. In the second part (Claim 7.21.2), we prove that the existence of such a dominated ensemble implies the existence of an ensemble $\{Z_n\}$ that essentially is *critically* dominated by $\{X_n\}$ such that f is strongly inapproximable on $\{Z_n\}$. Finally, we note that such a critically dominated ensemble defines a hard region of f relative to $\{X_n\}$, and the theorem follows.

Claim 7.21.1: Under the hypothesis of the theorem it holds that there exists a probability ensemble $\{Y_n\}$ that is ρ-dominated by $\{X_n\}$ such that, for every $s'(n)$-size circuit C_n, it holds that

$$\mathsf{Pr}[C_n(Y_n)=f(Y_n)]\le\frac{1}{2}+\frac{\varepsilon(n)}{2}. \tag{7.10}$$

Proof: We employ von Neumann's Min-Max Principle (cf. [217]) to a "game" between all critically dominated (by X_n) probability distributions and all possible $s'(n)$-size circuits.[15] We start by assuming, towards the contradiction, that for every distribution Y_n that is dominated by X_n there exists a $s'(n)$-size circuits C_n such that $\mathsf{Pr}[C_n(Y_n)=f(Y_n)]>0.5+\varepsilon'(n)$, where $\varepsilon'(n)=\varepsilon(n)/2$. One key observation is that any distribution that is dominated by X_n can be written as a convex combination of critically dominated (by X_n) distributions (cf., a special case in §D.4.1.1). Considering an enumeration $Y_n^{(1)},...,Y_n^{(t)}$ of the critically dominated (by X_n) distributions, we conclude that for every distribution π on $[t]$ there exists a $s'(n)$-size circuits C_n such that

$$\sum_{i=1}^{t}\pi(i)\cdot\mathsf{Pr}[C_n(Y_n^{(i)})=f(Y_n^{(i)})]>0.5+\varepsilon'(n). \tag{7.11}$$

[14] See details in [98, Apdx. A].

[15] We warn that this application of the min-max principle is somewhat non-straightforward.

Consider a finite game between two players, where the first player selects a critically dominated (by X_n) distribution, and the second player selects a $s'(n)$-size circuit and obtains a payoff as determined by the corresponding success probability; that is, if the first player selects the i^{th} critically dominated distribution and the second player selects the circuit C then the payoff equals $\Pr[C(Y_n^{(i)}) = f(Y_n^{(i)})]$. Now, Eq. (7.11) means that for any randomized strategy for the first player there exists a deterministic strategy for the second player yielding average payoff greater than $0.5 + \varepsilon'(n)$. The min-max principle asserts that in such a case there exists a randomized strategy for the second player that yields average payoff greater than $0.5 + \varepsilon'(n)$ no matter what strategy is employed by the first player. This means that there exists a distribution, D_n, on $s'(n)$-size circuits such that for every i it holds that

$$\Pr[D_n(Y_n^{(i)}) = f(Y_n^{(i)})] > 0.5 + \varepsilon'(n), \tag{7.12}$$

where the probability refers both to the choice of the circuit D_n and to the random variable Y_n. Let $B_n = \{x : \Pr[D_n(x) = f(x)] \leq 0.5 + \varepsilon'(n)\}$. Then, $\Pr[X_n \in B_n] < \rho(n)$, because otherwise we reach a contradiction to Eq. (7.12) by defining Y_n such that $\Pr[Y_n = x] = \Pr[X_n = x]/\Pr[X_n \in B_n]$ if $x \in B_n$ and $\Pr[Y_n = x] = 0$ otherwise, and noting that Y_n is dominated by X_n and $\Pr[D_n(Y_n) = f(Y_n)] > 0.5 + \varepsilon'(n)$.[16] By employing standard amplification to D_n, we obtain a distribution D'_n over $\text{poly}(n/\varepsilon'(n)) \cdot s'(n)$-size circuits such that for every $x \in \{0,1\}^n \setminus B_n$ it holds that $\Pr[D'_n(x) = f(x)] > 1 - 2^{-n}$. It follows that there exists a $s(n)$-sized circuit C_n such that $C_n(x) = f(x)$ for every $x \in \{0,1\}^n \setminus B_n$, and it follows that $\Pr[C_n(X_n) = f(X_n)] \geq \Pr[X_n \in \{0,1\}^n \setminus B_n] > 1 - \rho(n)$, in contradiction to the theorem's hypothesis. The claim follows. □

Claim 7.21.2: If there exists a probability ensemble $\{Y_n\}$ that is ρ-dominated by $\{X_n\}$ and satisfies Eq. (7.10), then there exists a probability ensemble $\{Z_n\}$ that is ρ'-critically dominated by $\{X_n\}$ and satisfies Eq. (7.10) with $\varepsilon(n)/2$ replaced by $\varepsilon(n)$.

In other words, Claim 7.21.2 asserts that the function f has a hard region of density $\rho'(\cdot)$ relative to $\{X_n\}$ with respect to $s'(\cdot)$-size circuits and advantage $2\varepsilon(\cdot)$, thus establishing the theorem. The proof of Claim 7.21.2 is by the Probabilistic Method (cf. [10]). Specifically, we select a set S_n at random by including each n-bit long string x with probability

$$p(x) \stackrel{\text{def}}{=} \frac{\rho(n) \cdot \Pr[Y_n = x]}{\Pr[X_n = x]} \leq 1 \tag{7.13}$$

independently of the choice of all other strings. It can be shown that, with high probability over the choice of S_n, it holds that $\Pr[X_n \in S_n] \approx \rho(n)$ and that $\Pr[C_n(X_n) = f(X_n) | X_n \in S_n] < 0.5 + \varepsilon(n)$ for every circuit C_n of size $s'(n)$. The latter assertion is proved by a union bound on all relevant circuits, showing that for each such circuit C_n, with probability $1 - \exp(-s'(n)^2)$ over the choice of S_n,

[16] We use again the fact that any dominated distribution is a convex combination of critically dominated distributions.

it holds that $|\mathsf{Pr}[C_n(X_n)=f(X_n)|X_n\in S_n] - \mathsf{Pr}[C_n(Y_n)=f(Y_n)]| < \varepsilon(n)/2$. For details see [98, Apdx. A]. □

7.2.2.2 Hardness amplification via hard regions

Before showing how to use the notion of a hard region in order to prove a derandomized version of Yao's XOR Lemma, we show how to use it in order to prove the original version of Yao's XOR Lemma (i.e., Theorem 7.13).

An alternative proof of Yao's XOR Lemma. Let f, p, and T be as in Theorem 7.13. Then, by Theorem 7.21, for $\rho'(n) = 1/3p(n)$ and $s'(n) = T(n)^{\Omega(1)}/\text{poly}(n)$, the function f has a hard region S of density ρ' (relative to $\{U_n\}$) with respect to $s'(\cdot)$-size circuits and advantage $1/s'(\cdot)$. Thus, for $t(n) = n\cdot p(n)$ and F as in Theorem 7.13, with probability at least $1-(1-\rho'(n))^{t(n)} = 1-\exp(-\Omega(n))$, one of the $t(n)$ random n-bit blocks of F resides in S (i.e., the hard region of f). Intuitively, this suffices for establishing the strong inapproximability of F. Indeed, suppose towards the contradiction that a small circuit C_n can approximate F with advantage $\varepsilon(n)+\exp(-\Omega(n))$, where $\varepsilon(n) > 1/s'(n)$. Then, the $\varepsilon(n)$ term must be due to $t(n)\cdot n$-bit long inputs that contain a block in S. Using an averaging argument, we can first fix the index of this block and then the contents of the other blocks, and infer the following: for some $i\in[t(n)]$ and $x_1,...,x_{t(n)}\in\{0,1\}^n$ it holds that

$$\mathsf{Pr}[C_n(x',U_n,x'') = F(x',U_n,x'')\,|\,U_n\in S] \;\geq\; \frac{1}{2}+\varepsilon(n)$$

where $x' = (x_1,...,x_{i-1})\in\{0,1\}^{(i-1)\cdot n}$ and $x'' = (x_{i+1},...,x_{t(n)})\in\{0,1\}^{(t(n)-i)\cdot n}$. Hard-wiring $i\in[t(n)]$, $x' = (x_1,...,x_{i-1})$ and $x'' = (x_{i+1},...,x_{t(n)})$ as well as $\sigma \stackrel{\text{def}}{=} \oplus_{j\neq i} f(x_j)$ in C_n, we obtain a contradiction to the (established) fact that S is a hard region of f (by using the circuit $C'_n(z) = C_n(x',z,x'')\oplus\sigma$), and the theorem follows. Actually, we derive a generalization of Theorem 7.13 asserting that *for any function T such that f is $(T,1/p)$-inapproximable it holds that F is T'-inapproximable for $T'(t(n)\cdot n) = s'(n) = T(n)^{\Omega(1)}/\text{poly}(n)$.*

Derandomized versions of Yao's XOR Lemma. We first show how to use the notion of a hard region in order to amplify very mild inapproximability to a constant level of inapproximability. This amplification utilizes a pairwise independence generator (see Section 8.6.1), G, that stretches $2n$-bit long seeds to sequences of n strings, each of length n.

Lemma 7.22 (derandomized XOR Lemma up to constant inapproximability): *Suppose that $f:\{0,1\}^*\to\{0,1\}$ is (T,ρ)-inapproximable, for $\rho(n) > 1/\text{poly}(n)$. Let b denote the inner-product mod 2 predicate, and G be the aforementioned pairwise independence generator. Then $F_1(s,r) = b(f(x_1)\cdots f(x_n),r)$, where $|r| = n = |s|/2$ and $(x_1,...,x_n) = G(s)$, is (T',ρ')-inapproximable for $T'(n') = T(n'/3)/\text{poly}(n')$ and $\rho'(n') = \Omega(\min(n'\cdot\rho(n'/3),1))$.*

Needless to say, if $f \in \mathcal{E}$ then $F_1 \in \mathcal{E}$.

Proof Sketch: Again, by Theorem 7.21, for $\alpha(n) = \rho(n)/3$ and $s'(n) = T(n)/\text{poly}(n)$, the function f has a hard region S of density α (relative to $\{U_n\}$) with respect to $s'(\cdot)$-size circuits and advantage 0.01. Consider the function $P_1(s) = (f(x_1), ..., f(x_n))$, where $|s| = 2n$ and $(x_1, ..., x_n) = G(s)$. By Exercise 7.13, with probability at least $\beta(n) \stackrel{\text{def}}{=} \min(n \cdot \alpha(n), 1)/2$, at least one of the n strings output by $G(U_{2n})$ resides in S. Intuitively, we expect every $s'(n)$-sized circuit to fail in computing $P_1(U_{2n})$ with probability at least $0.49\beta(n)$, because with probability $\beta(n)$ the sequence $G(U_{2n})$ contains an element in the hard region of f. Things are somewhat more involved (than in the non-derandomized case) because it is not clear what is the conditional distribution on the hard region.

For simplicity[17] (and without loss of generality), we assume that $\alpha(n) < 1/2n$, and note that in this case with probability at least $\beta(n) \stackrel{\text{def}}{=} 0.75 \cdot n \cdot \alpha(n)$, at least one of the n strings output by $G(U_{2n})$ resides in S. We claim that every $(s'(n) - \text{poly}(n))$-sized circuit fails to compute P_1 correctly with probability at least $\gamma(n) = 0.2\beta(n)$. As usual, the claim is proved by a reducibility argument. Let $G(s)_i$ denote the i^{th} string in the sequence $G(s)$ (i.e., $G(s) = (G(s)_1, ..., G(s)_n)$), and note that given i and x we can efficiently sample $G_i^{-1}(x) \stackrel{\text{def}}{=} \{s \in \{0,1\}^{2n} : G(s)_i = x\}$. Given a circuit C_n that computes $P_1(U_{2n})$ correctly with probability $1 - \gamma(n)$, we consider the circuit C'_n that, on input x, uniformly selects $i \in [n]$ and $s \in G_i^{-1}(x)$, and outputs the i^{th} bit in $C_n(s)$. Then,

$$\begin{aligned}
\Pr[C'_n(U_n) = f(U_n) | U_n \in S] &\geq \sum_{i=1}^{n} \frac{1}{n} \cdot \Pr[C_n(U_{2n}) = P_1(U_{2n}) | G(U_{2n})_i \in S] \\
&\geq \frac{1}{n} \cdot \frac{\Pr[C_n(U_{2n}) = P_1(U_{2n}) \wedge \exists i\, G_i(U_{2n})_i \in S]}{\max_i\{\Pr[G(U_{2n})_i \in S]\}} \\
&\geq \frac{1}{n} \cdot \frac{(1 - \gamma(n)) - (1 - \beta(n))}{\alpha(n)} \;=\; \frac{0.8\beta(n)}{n \cdot \alpha(n)} \;=\; 0.6
\end{aligned}$$

contradicting the fact that S is a hard region of f with respect to $s'(\cdot)$-size circuits and advantage 0.01. Employing the simple (warm-up) case discussed at the beginning of the proof of Theorem 7.7, it follows that, for $s''(n') = s(n'/3)/\text{poly}(n')$, every $s''(|s| + |r|)$-sized circuits fails to compute $(s, r) \mapsto b(P_1(s), r)$ with probability at least $\delta(|s| + |r|) \stackrel{\text{def}}{=} 0.24 \cdot \gamma(|r|)$. Thus, F_1 is $(s'', 2\delta)$-inapproximable, and the lemma follows. □

The next lemma offers an amplification of constant inapproximability to strong inapproximability. Indeed, combining Theorem 7.12 with Lemmas 7.22 and 7.23, yields Theorem 7.19 (as a special case).

[17] The choice of some of the constants in the following argument is rather arbitrary. In general, assuming that $\alpha(n) < c/n$, for some constant $c \in (0, 1)$, we can set $\beta(n) = (1 - (c/2)) \cdot n \cdot \alpha(n)$. This allows setting $\gamma(n) = c'\beta(n)$ for c' satisfying $(1 - c')(1 - (c/2)) \geq (0.5 + \varepsilon)$, where ε is the advantage in the definition of the hard region. (We have used $\varepsilon = 0.01$, $c = 0.5$ and $c' = 0.2$.) Finally, in moving from P_1 to F_1 we lose a factor that can be made arbitrarily close to 4.

Lemma 7.23 (derandomized XOR Lemma starting with constant inapproximability): *Suppose that $f : \{0,1\}^* \to \{0,1\}$ is (T,ρ)-inapproximable, for some constant ρ, and let b denote the inner-product mod 2 predicate. Then there exists a exponential-time computable function G such that $F_2(s,r) = b(f(x_1)\cdots f(x_n),r)$, where $(x_1,...,x_n) = G(s)$ and $n = \Omega(|s|) = |r| = |x_1| = \cdots = |x_n|$, is T'-inapproximable for $T'(n') = T(n'/O(1))^{\Omega(1)}/\mathrm{poly}(n')$.*

Again, if $f \in \mathcal{E}$ then $F_2 \in \mathcal{E}$.

Proof Outline:[18] As in the proof of Lemma 7.22, we start with a hard region of density $\rho/3$ (and advantage $1/T'$) for f and focus on the analysis of the function $P_2(s) = (f(x_1),...,f(x_n))$, where $|s| = O(n)$ and $(x_1,...,x_n) = G(s)$. The "generator" G is defined such that $G(s's'') = G_1(s') \oplus G_2(s'')$, where $|s'| = |s''|$, $|G_1(s')| = |G_2(s'')|$, and the following conditions hold:

1. G_1 is the Expander Random Walk Generator discussed in Section 8.6.3. It can be shown that $G_1(U_{O(n)})$ outputs a sequence of n strings such that for any set S of density ρ, with probability $1 - \exp(-\Omega(\rho n))$, at least $\Omega(\rho n)$ of the strings hit S. Note that this property is inherited by G, provided $|G_1(s')| = |G_2(s'')|$ for any $|s'| = |s''|$. It follows that, with probability $1 - \exp(-\Omega(\rho n))$, a constant fraction of the x_i's in the definition of P_2 hit the hard region of f.

   It is tempting to say that small circuits cannot compute P_2 better than with probability $\exp(-\Omega(\rho n))$, but this is clear only in case the the x_i's that hit the hard region are drawn independently from it, which is hardly the case here. Indeed, G_2 is used to handle this problem.

2. G_2 is the "set projection" system underlying Construction 8.17; specifically, $G_2(s) = (s_{S_1},...,s_{S_n})$, where each S_i is an n-subset of $[|s|]$ and the S_i's have pairwise intersections of size at most $n/O(1)$.[19] An analysis as in the proof of Theorem 8.18 can be employed for showing that the dependency among the x_i's does not help for computing a particular $f(x_i)$ when given x_i as well as all the other $f(x_j)$'s. (Note that the relevant property of G_2 is inherited by G.)

The actual analysis of the construction (via a guessing game presented in [121, Sec. 3]), links the success probability of computing P_2 to the advantage of guessing f on its hard region. The interested reader is referred to [121]. □

Digest. Both Lemmas 7.22 and 7.23 are proved by first establishing corresponding "direct product" versions (i.e., derandomized versions of Theorem 7.14). We call the reader's attention to the seemingly crucial role of this step (especially in the proof of Lemma 7.23): We cannot treat the values $f(x_1),...f(x_n)$ as independent (at least not for the generator G as postulated in the proof), and so we seek

[18] For details, see [121].

[19] Recall that s_S denotes the projection of s on coordinates $S \subseteq [|s|]$; that is, for $s = \sigma_1 \cdots \sigma_k$ and $S = \{i_j : j = 1,...,n\}$, we have $s_S = \sigma_{i_1} \cdots \sigma_{i_n}$.

to avoid analyzing the probability of correctly computing the XOR of *all these values*. In contrast, we have established that it is very hard to correctly compute all n values, and thus *XORing a random subset of these values* yields a strongly inapproximable predicate. Note that the argument used in Exercise 7.11 fails here, becuase the x_i's are not independent.

Chapter Notes

The notion of a one-way function was suggested by Diffie and Hellman [62]. The notion of weak one-way functions as well as the amplification of one-way functions (Theorem 7.5) were suggested by Yao [221]. A proof of Theorem 7.5 has first appeared in [83].

The concept of hard-core predicates was suggested by Blum and Micali [37]. They also proved that a particular predicate constitutes a hard-core for the "DLP function" (i.e., exponentiation in a finite field), provided that the latter function is one-way. The generic hard-core predicate (Theorem 7.7) was suggested by Levin, and proven as such by Goldreich and Levin [95]. The proof presented here was suggested by Rackoff. We comment that the original proof has its own merits (cf., e.g., [101]).

The construction of canonical derandomizers and, specifically, the Nisan-Wigderson framework (Construction 8.17) has been the driving force behind the study of inapproximable predicates in $\mathcal{E}$. Theorem 7.10 is due to [19], whereas Theorem 7.19 is due to [121]. Both results rely heavily of variants of Yao's XOR Lemma, to be reviewed next.

Like several other fundamental insights attributed to Yao's paper [221], Yao's XOR Lemma (Theorem 7.13) is not even stated in [221] but is rather due to Yao's oral presentations of his paper. The first published proof of Yao's XOR Lemma was given by Levin (see [98, Sec. 3]). Levin's proof is the only one known giving a tight quantitative analysis (on the decrease in the level of approximability), and the interested reader is referred to it (via the non-laconic presentation of [98, Sec. 3]). The proof presented in §7.2.1.2 is due to Goldreich, Nisan and Wigderson [98, Sec. 5].

The notion of a hard region and its applications to proving the original version of Yao's XOR Lemma as well as the first derandomization of it (Lemma 7.22) are due to Impagliazzo [119]. The second derandomization (Lemma 7.23) as well as Theorem 7.19 are due to Impagliazzo and Wigderson [121].

The connection between list decoding and hardness amplification (Section 7.2.1.3), yielding an alternative proof of Theorem 7.19, is due to Sudan, Trevisan, and Vadhan [204].

Hardness amplification for $\mathcal{NP}$ has been the subject of recent attention: An amplification of mild inapproximablity to strong inapproximablity is provided in [115], an indication to the impossibility of a worst-case to average-case reductions (at least non-adaptive ones) is provided in [40].

Exercises

Exercise 7.1 Prove that if one way-functions exist then there exists one-way functions that are length preserving (i.e., $|f(x)| = |x|$ for every $x \in \{0,1\}^n$).
(Hint: Clearly, for some polynomial p, it holds that $|f(x)| \leq p(|x|)$ for all x. Assume, without loss of generality that $n \mapsto p(n)$ is 1-1, and let $p^{-1}(m) = n$ if $p(n) \leq m < p(n+1)$. Define $f'(z) = f(x)0^{|z|-|f(x)|}$, where x is the $p^{-1}(|z|)$-bit long prefix of z.)

Exercise 7.2 Prove that if a function f is hard to invert in the sense of Definition 7.3 then it is hard to invert in the sense of Definition 7.1.
(Hint: consider a sequence of internal coin tosses that maximizes the probability in Eq. (7.1).)

Exercise 7.3 Assuming the existence of one-way functions, prove that there exists a weak one-way function that is not strongly one-way.

Exercise 7.4 (a universal one-way function) Using the notion of a universal machine, present a polynomial-time computable that is hard to invert (in the sense of Definition 7.1) if and only if there exist one-way functions.

Guideline: Consider the function F that parses its input into a pair (M, x) and emulates $|x|^3$ steps of M on input x. Note that if there exists a one-way function that can be evaluated in cubic time then F is a weak one-way function. Using padding, prove that there exists a one-way function that can be evaluated in cubic time if and only if there exist one-way functions.

Exercise 7.5 For $\ell > 1$, prove that the following $2^\ell - 1$ samples are pairwise independent and uniformly distributed in $\{0,1\}^n$. The samples are generated by uniformly and independently selecting ℓ strings in $\{0,1\}^n$. Denoting these strings by $s^1, ..., s^\ell$, we generate $2^\ell - 1$ samples corresponding to the different *non-empty* subsets of $\{1, 2, ..., \ell\}$ such that for subset J we let $r^J \stackrel{\text{def}}{=} \oplus_{j \in J} s^j$.
(Hint: for $J \neq J'$, it holds that $r^J \oplus r^{J'} = \oplus_{j \in K} s^j$, where K denotes the symmetric difference of J and J'. See related material in Section 8.6.1.)

Exercise 7.6 Prove Theorem 7.8. In particular, provide a detailed presentation of the alternative procedure outlined in Footnote 5.

Exercise 7.7 A polynomial-time computable predicate $b:\{0,1\}^* \to \{0,1\}$ is called a universal hard-core predicate if for every one-way function f, the predicate b is a hard-core of f. Note that the predicate presented in Theorem 7.7 is "almost universal" (i.e., for every one-way function f, that predicate is a hard-core of $f'(x,r) = (f(x), r)$, where $|x| = |r|$). Prove that there exist no universal hard-core predicate.
(Hint: Let b be a candidate universal hard-core predicate, and let f be an arbitrary one-way function. Then consider the function $f'(x) = (f(x), b(x))$.)

Exercise 7.8 Prove that if $\mathcal{NP}$ is not contained in $\mathcal{P}/\text{poly}$ then neither is $\mathcal{E}$. Furthermore, for every $S : \mathbb{N} \to \mathbb{N}$, if some problem in $\mathcal{NP}$ does not have circuits of size S then some problem in $\mathcal{E}$ does not have circuits of size S', where $S'(n) = S(n^\varepsilon)$ for some constant $\varepsilon > 0$.
(Hint: SAT is in $\mathcal{E}$.)

Exercise 7.9 For every function $f:\{0,1\}^n \to \{0,1\}$, present a linear-size circuit C_n such that $\Pr[C(U_n) = f(U_n)] \geq 0.5 + O(2^{-n})$. Furthermore, for every $t \leq 2^{n-1}$, present a circuit C_n of size $O(t \cdot n)$ such that $\Pr[C(U_n) = f(U_n)] \geq 0.5 + t \cdot 2^{-n}$. Warning: you may not assume that $\Pr[f(U_n) = 1] = 0.5$.

Exercise 7.10 Let $\hat{f}$ be as in the conclusion of Theorem 7.12. Prove that there exists a Boolean function g in $\mathcal{E}$ that is (p, ε)-inapproximable for every polynomial p and for $\varepsilon(n) = 1/n^3$.
(Hint: consider the function g defined such that $g(x,i)$ equals the i^{th} bit of $\hat{f}(x)$.)

Exercise 7.11 Let f be a Boolean function, and $b(y,r)$ denote the inner-product modulo 2 of the equal-length strings y and r. Suppose that $F'(x_1, ..., x_{t(n)}, r) \stackrel{\text{def}}{=} b(f(x_1) \cdots f(x_{t(n)}), r)$, where $x_1, ..., x_{t(n)} \in \{0,1\}^n$ and $r \in \{0,1\}^{t(n)}$, is T-inapproximable for every polynomial T. Assuming that $n \mapsto t(n) \cdot n$ is 1-1, prove that $F(x) \stackrel{\text{def}}{=} F'(x, 1^{t'(|x|)})$, where $t'(t(n) \cdot n) = t(n)$, is T-inapproximable for every polynomial T.

Guideline: Reduce the approximation of F' to the approximation of F. An important observation is that for any $x = (x_1, ..., x_{t(n)})$, $x' = (x'_1, ..., x'_{t(n)})$, and $r = r_1 \cdots r_{t(n)}$ such that $x'_i = x_i$ if $r_i = 1$, it holds that $F'(x, r) = F(x') \oplus \oplus_{i:r_i=0} f(x'_i)$. Note that the equality holds regardless of the choice of the string $x'_i \in \{0,1\}^n$ for which $r_i = 0$. Also note that the suggested reduction requires knowledge of $\sigma = \oplus_{i:r_i=0} f(x'_i)$, but in our context the reduction may be performed by a small non-uniform circuit, which may incorporate the values of $f(z)$'s for a small number of z's. Indeed, for uniformly chosen $z_1, ..., z_{t(n)} \in \{0,1\}^n$, we use these z_i's as well as the $f(z_i)$'s as advice to the reduction. On input $x_1, ..., x_{t(n)}, r_1 \cdots r_{t(n)}$, the reduction sets $x'_i = x_i$ if $r_i = 1$ and $x'_i = z_i$ otherwise, makes the query $x' = (x'_1, ..., x'_{t(n)})$ to F, and returns $F(x') \oplus_{i:r_i=0} f(z_i)$.

Exercise 7.12 Consider a modification of Definition 7.17, in which the decoding condition reads as follows (where p is a fixed polynomial): *For every $w : [\ell(N)] \to [q(N)]$ and $x \in \{0,1\}^N$ such that $\Gamma(x)$ is $(1 - ((1/q(N)) + \alpha(N)))$-close to w, there exists an oracle-aided circuit C of size $p((\log N)/\alpha(N))$ such that $C^w(i)$ yields the i^{th} bit of x for every $i \in [N]$.*

1. Formulate and prove a version of Theorem 7.18 that refers to the modified definition (rather than to the original one).

   (Hint: the modified version should refer to computing $g(U_{m(n)})$ with success probability greater than $(1/q(n)) + \varepsilon(n)$.)

2. Prove that, when applied to binary codes (i.e., $q \equiv 2$), the version in Item 1 yields S''-inapproximable predicates, for $S''(n') = S(m^{-1}(n'))^{\Omega(1)}/\text{poly}(n')$.

3. Prove that the Hadamard Code allows implicit decoding under the modified definition (but not according to the original one).[20]

   (Hint: this is the actual contents of Theorem 7.8.)

[20] Needless to say, the Hadamard Code is not efficient (for the trivial reason that its codewords have exponential length).

Note that encoding the symbols of a non-binary code Γ that allows implicit decoding with the Hadamard code yields a binary code that allows implicit decoding. Note that efficient encoding is preserved only if $q(N) = \text{poly}(N)$.

Exercise 7.13 Let G be as in Lemma 7.22, $S \subset \{0,1\}^n$ and $\alpha \stackrel{\text{def}}{=} |S|/2^n$. Prove that, with probability at least $\min(n \cdot \alpha, 1)/2$, at least one of the n strings output by $G(U_{2n})$ resides in S.

Guideline: Using the pairwise independence property and employing the Inclusion-Exclusion formula, we lower-bound the aforementioned probability by $n \cdot p - \binom{n}{2} \cdot p^2$. If $p \leq 1/n$ then the claim follows, otherwise we employ the same reasoning to the first $1/p$ elements in the output of $G(U_{2n})$.

Exercise 7.14 (one-way functions versus inapproximable predicates) Prove that the existence of a non-uniformly hard one-way function (as in Definition 7.3) implies the existence of an exponential-time computable predicate that is T-inapproximable (as per Definition 7.9), for every polynomial T.

Guideline: Suppose first that the one-way function f is length-preserving and 1-1. Consider the corresponding function g and hard-core predicate b guaranteed by Theorem 7.7, and show that the Boolean function h such that $h(z) = b(g^{-1}(z))$ is polynomially inapproximable. For the general case a different approach seems needed. Specifically, given a (length preserving) one-way function f, consider the Boolean function h defined as $h(z, i, \sigma) = 1$ if the i^{th} bit of the lexicographically first element in $f^{-1}(z) = \{x : f(x) = z\}$ equals σ. Note that h is computable in exponential-time, but is not (worst-case) computable in polynomial-time. Applying Theorem 7.10, we are done.

Chapter 8

Pseudorandom Generators

Indistinguishable things are identical.[1]

G.W. Leibniz (1646–1714)

A fresh view at the *question of randomness* has been taken in the theory of computing: It has been postulated that a distribution is pseudorandom if it cannot be told apart from the uniform distribution by any efficient procedure. The paradigm, originally associating efficient procedures with polynomial-time algorithms, has been applied also with respect to a variety of limited classes of such distinguishing procedures.

At the extreme, this approach says that the question of whether the world is deterministic or allows for some free choice (which may be viewed as sources of randomness) is irrelevant. *What matters is how the world looks to us and to various computationally bounded devices.* That is, if some phenumenon looks random then we may just treat it as if it were random. Likewise, if we can generate sequences that cannot be told apart from the uniform distribution by any efficient procedure, then we can use these sequences in any efficient randomized application instead of the ideal random bits that are postulated in the design of this application.

Summary: A generic formulation of pseudorandom generators consists of specifying three fundamental aspects – the *stretch measure* of the generators; the class of distinguishers that the generators are supposed to fool (i.e., the algorithms with respect to which the *computational indistinguishability* requirement should hold); and the resources that the generators are allowed to use (i.e., their own *computational complexity*).

The archetypical case of pseudorandom generators refers to efficient generators that fool any feasible procedure; that is, the potential distinguisher is any probabilistic polynomial-time algorithm, which may

[1] This is the *Principle of Identity of Indiscernibles.* Leibniz admits that counterexamples to this principle are conceivable but will not occur in real life because God is much too benevolent. We thus believe that he would have agreed to the theme of this chapter, which asserts that *indistinguishable things should be considered as identical.*

be more complex than the generator itself (which, in turn, has time-complexity bounded by a fixed polynomial). These generators are called general-purpose, because their output can be safely used in an efficient application. Such (general-purpose) pseudorandom generators exist if and only if one-way functions exist.

For purposes of derandomization one may use pseudorandom generators that are somewhat more complex than the potential distinguisher (which represents the algorithm to be derandomized). Following this approach, suitable pseudorandom generators, which can be constructed assuming the existence of problems in $\mathcal{E}$ that have no sub-exponential size circuits, yield a full derandomization of $\mathcal{BPP}$ (i.e., $\mathcal{BPP} = \mathcal{P}$).

It is also beneficial to consider pseudorandom generators that fool space-bounded distinguishers and generators that exhibit some limited random behavior (e.g., outputting a pair-wise independent or a small-bias sequence).

8.1 Introduction

The second half of this century has witnessed the development of three theories of randomness, a notion which has been puzzling thinkers for ages. The first theory (cf., [60]), initiated by Shannon [189], is rooted in probability theory and is focused at distributions that are not perfectly random. Shannon's Information Theory characterizes perfect randomness as the extreme case in which the *information contents* is maximized (i.e., there is no redundancy at all). Thus, perfect randomness is associated with a unique distribution – the uniform one. In particular, by definition, one cannot (deterministically) generate such perfect random strings from shorter random seeds.

The second theory (cf., [143, 146]), due to Solomonov [196], Kolmogorov [137] and Chaitin [48], is rooted in computability theory and specifically in the notion of a universal language (equiv., universal machine or computing device; see §1.2.3.3). It measures the complexity of objects in terms of the shortest program (for a fixed universal machine) that generates the object. Like Shannon's theory, Kolmogorov Complexity is quantitative and perfect random objects appear as an extreme case. However, in this approach one may say that a single object, rather than a distribution over objects, is perfectly random. Still, Kolmogorov's approach is inherently intractable (i.e., Kolmogorov Complexity is uncomputable), and – by definition – one cannot (deterministically) generate strings of high Kolmogorov Complexity from short random seeds.

The third theory is rooted in complexity theory and is the focus of this chapter. This approach is explicitly aimed at providing a notion of randomness that nevertheless allows for an efficient (and deterministic) generation of random strings from shorter random seeds. The heart of this approach is the suggestion to view objects as equal if they cannot be told apart by any efficient procedure. Consequently, a distribution that cannot be efficiently distinguished from the uniform distribution will be considered as being random (or rather called pseudorandom). Thus, randomness is not an "inherent" property of objects (or distributions) but is rather relative to an observer (and its computational abilities). To demonstrate this approach, let us consider the following mental experiment.

> Alice and Bob play "head or tail" in one of the following four ways. In each of them Alice flips an unbiased coin and Bob is asked to guess its outcome *before* the coin hits the floor. The alternative ways differ by the knowledge Bob has before making his guess.
>
> In the first alternative, Bob has to announce his guess before Alice flips the coin. Clearly, in this case Bob wins with probability 1/2.
>
> In the second alternative, Bob has to announce his guess while the coin is spinning in the air. Although the outcome is *determined in principle* by the motion of the coin, Bob does not have accurate information on the motion and thus we believe that also in this case Bob wins with probability 1/2.
>
> The third alternative is similar to the second, except that Bob has at his disposal sophisticated equipment capable of providing accurate *information* on the coin's motion as well as on the environment effecting the outcome. However, Bob cannot process this information in time to improve his guess.
>
> In the fourth alternative, Bob's recording equipment is directly connected to a *powerful computer* programmed to solve the motion equations and output a prediction. It is conceivable that in such a case Bob can improve substantially his guess of the outcome of the coin.

We conclude that the randomness of an event is relative to the information and computing resources at our disposal. Thus, a natural concept of pseudorandomness arises – a distribution is *pseudorandom* if no efficient procedure can distinguish it from the uniform distribution, where efficient procedures are associated with (probabilistic) polynomial-time algorithms. This notion of pseudorandomness is indeed the most fundamental one, and much of this chapter is focused on it. Weaker notions of pseudorandomness arise as well – they refer to indistinguishability by weaker procedures such as space-bounded algorithms, constant-depth circuits, etc. Stretching this approach even further one may consider algorithms that are designed on purpose so not to distinguish even weaker forms of "pseudorandom" sequences from random ones (such algorithms arise naturally when trying to convert some natural randomized algorithm into deterministic ones; see Section 8.6).

The foregoing discussion has focused at one aspect of the pseudorandomness question – the resources or type of the observer (or potential distinguisher). Another important aspect is whether such pseudorandom sequences can be generated from much shorter ones, and at what cost (or complexity). A natural approach is that the generation process has to be at least as efficient as the distinguisher (equiv., that the distinguisher is allowed at least as much resources as the generator). Coupled with the aforementioned strong notion of pseudorandomness, this yields the archetypical notion of pseudorandom generators – these operating in polynomial-time and producing sequences that are indistinguishable from uniform ones by *any* polynomial-time observer. Such (general-purpose) pseudorandom generators allow to reduced the randomness complexity of *any efficient application*, and are thus of great relevance to randomized algorithms and cryptography (see Section 8.3).

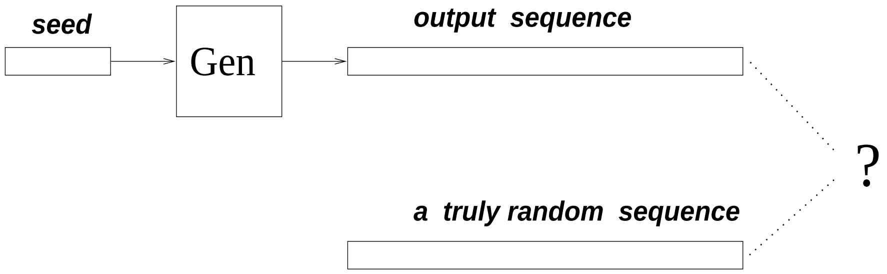


Figure 8.1: Pseudorandom generators – an illustration.

We stress that there are important reasons for considering also an alternative that seems less natural; that is, allowing the pseudorandom generator to use more resources (e.g., time or space) than the observer it tries to fool. This alternative is natural in the context of derandomization (i.e., converting randomized algorithms to deterministic ones), where the crucial step is replacing the random input of an algorithm by a pseudorandom input, which in turn can be generated based on a much shorter random seed. In particular, when derandomizing a probabilistic polynomial-time algorithm, the observer (to be fooled by the genewrator) is a fixed algorithm. In this case employing a more complex generator merely means that the complexity of the derived deterministic algorithm is dominated by the complexity of the generator (rather than by the complexity of the original randomized algorithm). Needless to say, allowing the generator to use more resources than the observer it tries to fool makes the task of designing pseudorandom generators easier, and enables derandomization results that are not known when using general-purpose pseudorandom generators. The usefulness of this approach is demonstrated in Sections 8.4 through 8.6.

We note that the goal of all types of pseudorandom generators is to allow the generation of "sufficiently random" sequences based on much shorter random seeds. Thus, pseudorandom generators offer significant saving in the randomness complexity of various applications. This saving is valuable because many applications are severly limited in their ability to generate or obtain truly random bits. Furthermore, typically generating truly random bits is significantly more expensive than standard computation steps. Thus, randomness is a computational resource that should be considered on top of time complexity (analogously to the consideration of space complexity).

Organization. In Section 8.2 we present the general paradigm underlying the various notions of pseudorandom generators. The archetypical case of general-purpose pseudorandom generators is presented in Section 8.3. We then turn to the alternative notions of pseudorandom generators: Generators that suffice for the derandomization of complexity classes such as $\mathcal{BPP}$ are discussed in Section 8.4; Pseudorandom generators in the domain of space-bounded computations are discussed in Section 8.5; and special-purpose generators are discussed in Section 8.6. (For an alternative presentation, which focuses on general-purpose pseudorandom generators and provides more details on it, the reader is referred to [87, Chap. 3].)

Teaching note: If you can afford teaching only one of the alternative notions of pseudorandom generators, then we suggest teaching the notion of general-purpose pseudorandom generators (presented in Section 8.3). Our reasons being that this notion is relevant to computer science at large and that the technical material is relatively simpler. The chapter is organized to facilitate this option.

Prerequisites: We assume a basic familiarity with elementary probability theory (see Section D.1) and randomized algorithms (see Section 6.1). In particular, standard conventions regarding random variables (presented in Section D.1.1) will be extensively used.

8.2 The General Paradigm

Teaching note: We advocate a unified view of various notions of pseudorandom generators. That is, we view these notions as incarnations of a general abstract paradigm, to be presented in this section. A teacher that wishes to focus on one of the special cases may still use this section as a general motivation towards the specific definitions used later.

A generic formulation of pseudorandom generators consists of specifying three fundamental aspects – the *stretch measure* of the generators; the class of distinguishers that the generators are supposed to fool (i.e., the algorithms with respect to which the *computational indistinguishability* requirement should hold); and the resources that the generators are allowed to use (i.e., their own *computational complexity*).

Stretch function: A necessary requirement from any notion of a pseudorandom generator is that it is a deterministic algorithm that stretches short strings, called seeds, into longer output sequences. Specifically, it stretches k-bit long seeds into $\ell(k)$-bit long outputs, where $\ell(k) > k$. The function $\ell : \mathbb{N} \rightarrow \mathbb{N}$ is called the stretch measure (or stretch function). In some settings the specific stretch measure is immaterial (e.g., see Section 8.3.4).

Computational Indistinguishability: A necessary requirement from any notion of a pseudorandom generator is that it "fools" some non-trivial algorithms. That is, any algorithm taken from some class of interest cannot distinguish the output produced by the generator (when the generator is fed with a uniformly chosen seed) from a uniformly chosen sequence. Typically, we consider a class $\mathcal{D}$ of distinguishers and a class $\mathcal{F}$ of (threshold) functions, and require that the generator G satisfies the following: For any $D \in \mathcal{D}$, any $f \in \mathcal{F}$, and for all sufficiently large k's

$$|\,\mathsf{Pr}[D(G(U_k)) = 1] \;-\; \mathsf{Pr}[D(U_{\ell(k)}) = 1]\,| \;<\; f(k) \qquad (8.1)$$

where U_n denotes the uniform distribution over $\{0,1\}^n$ and the probability is taken over U_k (resp., $U_{\ell(k)}$) as well as over the coin tosses of algorithm D in case it is probabilistic.[2] The reader may think of such a distinguisher, D, as trying to tell whether the "tested string" is a random output of the generator (i.e., distributed as $G(U_k)$) or is a truly random string (i.e., distributed as $U_{\ell(k)}$). The condition in Eq. (8.1) requires that D cannot make a meaningful decision; that is, ignoring a negligible difference (represented by $f(k)$), D's verdict is the same in both cases. The archetypical choice is that $\mathcal{D}$ is the set of all probabilistic polynomial-time algorithms, and $\mathcal{F}$ is the set of all functions that are the reciprocal of some positive polynomial.

Complexity of Generation: The archetypical choice is that the generator has to work in polynomial-time (in length of its input – the seed). Other choices will be discussed as well. We note that placing no computational requirements on the generator (or, alternatively, putting very mild requirements such as a double-exponential running-time upper bound), yields "generators" that can fool any subexponential-size circuit family (see Exercise 8.1).

Notational conventions. We will consistently use k to denote the length of the seed of a pseudorandom generator, and $\ell(k)$ to denote the length of the corresponding output. In some cases, this makes our presentation a little more cumbersome (as a natural presentation may specify some other parameters and let the seed-length be a function of these). However, our choice has the advantage of focusing attention on the fundamental parameter of pseudorandom generation – the length of the random seed. We note that whenever a pseudorandom generator is used to "derandomize" an algorithm, n will denote the length of the input to this algorithm, and k will be selected as a function of n.

Some instantiations of the general paradigm. Two important instantiations of the notion of pseudorandom generators relate to probabilistic polynomial-time observers.

[2] The class of threshold functions $\mathcal{F}$ should be viewed as determining the class of noticeable probabilities (as a function of k). Thus, we require certain functions (i.e., the absolute difference between the above probabilities), to be smaller than any noticeable function *on all but finitely many integers*. We call the former functions negligible. Note that a function may be neither noticeable nor negligible (e.g., it may be smaller than any noticeable function on infinitely many values and yet larger than some noticeable function on infinitely many other values).

1. General-purpose pseudorandom generators correspond to the case that the generator itself runs in polynomial time and needs to withstand *any probabilistic polynomial-time distinguisher*, including distinguishers that run for more time than the generator. Thus, the same generator may be used safely in any efficient application.

2. In contrast, pseudorandom generators intended for derandomization may run more time than the distinguisher, which is viewed as a fixed circuit having size that is upper-bounded by a fixed polynomial.

In addition, the general paradigm may be instantiated by focusing on the space complexity of the potential distinguishers (and the generator), rather than on their time complexity. Furthermore, one may also consider distinguishers that merely reflect probabilistic properties such as pair-wise independence, small-bias, and hitting frequency.

8.3 General-Purpose Pseudorandom Generators

Randomness is playing an increasingly important role in computation: It is frequently used in the design of sequential, parallel and distributed algorithms, and it is of course central to cryptography. Whereas it is convenient to design such algorithms making free use of randomness, it is also desirable to minimize the usage of randomness in real implementations. Thus, general-purpose pseudorandom generators (as defined next) are a key ingredient in an "algorithmic tool-box" – they provide an automatic compiler of programs written with free usage of randomness into programs that make an economical use of randomness.

8.3.1 The basic definition

Loosely speaking, general-purpose pseudorandom generators are efficient (i.e., polynomial-time) deterministic programs that expand short randomly selected seeds into longer pseudorandom bit sequences, where the latter are defined as computationally indistinguishable from truly random sequences by *any* efficient (i.e., polynomial-time) algorithm. Thus, the distinguisher is more complex than the generator: The generator is a fixed algorithm working within *some fixed* polynomial-time, whereas a potential distinguisher is *any* algorithm that runs in polynomial-time. Thus, for example, the distinguisher *may* always run in time cubic in the running-time of the generator. Furthermore, to facilitate the development of this theory, we allow the distinguisher to be probabilistic (whereas the generator remains deterministic as above). We require that such distinguishers cannot tell the output of the generator from a truly random string of similar length, or rather that the difference that such distinguishers may detect (or sense) is negligible. Here a negligible function is one that vanishes faster than the reciprocal of any positive polynomial.

Definition 8.1 (general-purpose pseudorandom generator): *A deterministic polynomial-time algorithm G is called a* pseudorandom generator *if there exists a* stretch function, *$\ell : \mathbb{N} \to \mathbb{N}$* (satisfying $\ell(k) > k$ for all k)*, such that for any probabilistic* *polynomial-time algorithm D, for any positive polynomial p, and for all sufficiently large k's*

$$|\,\mathsf{Pr}[D(G(U_k))=1] \;-\; \mathsf{Pr}[D(U_{\ell(k)})=1]\,| \;<\; \frac{1}{p(k)} \tag{8.2}$$

where U_n denotes the uniform distribution over $\{0,1\}^n$ and the probability is taken over U_k (resp., $U_{\ell(k)}$) as well as over the internal coin tosses of D.

Thus, Definition 8.1 is derived from the generic framework (presented in Section 8.2) by taking the class of distinguishers to be the set of all probabilistic polynomial-time algorithms, and taking the class of (noticeable) threshold functions to be the set of all functions that are the reciprocals of some positive polynomial.[3] The latter choice is naturally coupled with the association of efficient computation with polynomial-time algorithms: An event that occurs with noticeable probability occurs almost always when the experiment is repeated a "feasible" (i.e., polynomial) number of times.

We note that Definition 8.1 does not make any requirement regarding the stretch function $\ell:\mathbb{N}\to\mathbb{N}$, except for the generic requirement that $\ell(k)>k$ for all k. Needless to say, the larger ℓ is the more useful is the pseudorandom generator. In Section 8.3.4 we show how to use any pseudorandom generator (even one with minimal stretch $\ell(k)=k+1$) in order to obtain a pseudorandom generator of any desired polynomial stretch function. But before going so, we rigorously discuss the "reduction in randomness" offered by pseudorandom generators, and the notion of computational indistinguishability underlying Definition 8.1.

8.3.2 The archetypical application

We note that "pseudo-random number generators" appeared with the first computers. However, typical implementations use generators that are not pseudorandom according to Definition 8.1. Instead, at best, these generators are shown to pass *some* ad-hoc statistical test (cf., [136]). We warn that the fact that a "pseudorandom number generator" passes some statistical tests, does not mean that it will pass a new test and that it will be good for a future (untested) application. Furthermore, the approach of subjecting the generator to some ad-hoc tests fails to provide general results of the form "for *all* practical purposes using the output of the generator is as good as using truly unbiased coin tosses." In contrast, the approach encompassed in Definition 8.1 aims at such generality, and in fact is tailored to obtain it: The notion of computational indistinguishability, which underlines Definition 8.1, covers all possible efficient applications guaranteeing that for all of them pseudorandom sequences are as good as truly random ones. Indeed, any efficient randomized algorithm maintains its performance when its internal coin tosses are substituted by a sequence generated by a pseudorandom generator. That is:

[3] Definition 8.1 requires that the distinguishing gap of certain algorithms must be smaller than the reciprocal of any positive polynomial for all but finitely many k's. Such functions are called *negligible*; see Footnote 2. The notion of negligible probability is robust in the sense that an event which occurs with negligible probability occurs with negligible probability also when the experiment is repeated a "feasible" (i.e., polynomial) number of times.

Construction 8.2 (typical application of pseudorandom generators): *Let G be a pseudorandom generator with stretch function $\ell:\mathbb{N}\to\mathbb{N}$. Let A be a probabilistic algorithm, and $\rho(n)$ denote a* (polynomial) *upper bound on its randomness complexity. Denote by $A(x,r)$ the output of A on input x and coin tosses sequence $r \in \{0,1\}^{\rho(|x|)}$. Consider the following randomized algorithm, denoted A_G:*

> *On input x, set $k = k(|x|)$ to be the smallest integer such that $\ell(k) \geq \rho(|x|)$, uniformly select $s \in \{0,1\}^k$, and output $A(x,r)$, where r is the $\rho(|x|)$-bit long prefix of $G(s)$.*

That is, $A_G(x,s) = A(x,G'(s))$, for $|s| = k(|x|) = \mathrm{argmin}_i\{\ell(i) \geq \rho(|x|)\}$, where $G'(s)$ is the $\rho(|x|)$-bit long prefix of $G(s)$.

Thus, using A_G instead of A, the randomness complexity is reduced from ρ to $\ell^{-1} \circ \rho$, while (as we show next) it is infeasible to find inputs (i.e., x's) on which the *noticeable behavior* of A_G is different from the one of A. For example, if $\ell(k) = k^2$, then the randomness complexity is reduced from ρ to $\sqrt{\rho}$. We stress that the pseudorandom generator G is *universal*; that is, it can be applied to reduce the randomness complexity of *any* probabilistic polynomial-time algorithm A.

Proposition 8.3 *Let A, ρ and G be as in Construction 8.2, and suppose that $\rho : \mathbb{N} \to \mathbb{N}$ is 1-1. Then, for every pair of probabilistic polynomial-time algorithms,* a finder F *and* a distinguisher D, *every positive polynomial p and all sufficiently long n's*

$$\sum_{x\in\{0,1\}^n} \mathsf{Pr}[F(1^n) = x] \cdot |\, \Delta_{A,D}(x)\,| \;<\; \frac{1}{p(n)} \tag{8.3}$$

where $\Delta_{A,D}(x) \stackrel{\text{def}}{=} \mathsf{Pr}[D(x, A(x, U_{\rho(|x|)})) = 1] - \mathsf{Pr}[D(x, A_G(x, U_{k(|x|)})) = 1]$, and the probabilities are taken over the U_m's as well as over the coin tosses of F and D.

Algorithm F represents a potential attempt to find an input x on which the output of A_G is distinguishable from the output of A. This "attempt" may be benign as in the case that a user employs algorithm A_G on inputs that are generated by some probabilistic polynomial-time application. However, the attempt may also be adversarial as in the case that a user employs algorithm A_G on inputs that are provided by a potentially malicious party. The potential distinguisher, denoted D, represents the potential use of the output of algorithm A_G, and captures the requirement that this output be as good as a corresponding output produced by A. Thus, D is given x as well as the corresponding output produced either by $A_G(x) \stackrel{\text{def}}{=} A(x, U_{k(n)})$ or by $A(x) = A(x, U_{\rho(n)})$, and it is required that D cannot tell the difference. In the case that A is a probabilistic polynomial-time *decision procedure*, this means that it is infeasible to find an x on which A_G decides incorrectly (i.e., differently than A). In the case that A is a *search procedure for some NP-relation*, it is infeasible to find an x on which A_G outputs a wrong solution. For details, see Exercise 8.2.

Proof: The proposition is proven by showing that any triplet (A, F, D) violating the claim can be converted into an algorithm D' that distinguishes the output of G from the uniform distribution, in contradiction to the hypothesis. The key observation is that $\Delta_{A,D}(x)$ equals $\mathsf{Pr}[D(x, A(x, U_{\rho(n)})) = 1] - \mathsf{Pr}[D(x, A(x, G'(U_{k(n)}))) = 1]$, where $G'(s)$ is the $\rho(n)$-bit long prefix of $G(s)$. Details follow.

On input r (taken from either $U_{\ell(k(n))}$ or $G(U_{k(n)})$), algorithm D' first obtains $x \leftarrow F(1^n)$, where n can be obtained easily from $|r|$ (because ρ is 1-1 and $1^n \mapsto \rho(n)$ is computable via A). Next, D' obtains $y = A(x, r')$, where r' is the $\rho(|x|)$-bit long prefix of r. Finally D' outputs $D(x, y)$. Note that D' is implementable in probabilistic polynomial-time, and that

$$
\begin{aligned}
D'(U_{\rho(n)}) &= D(X_n, A(X_n, U_{\rho(n)}))\,, \text{ where } X_n \stackrel{\text{def}}{=} F(1^n) \\
D'(G'(U_{k(n)})) &= D(X_n, A(X_n, G'(U_{k(n)})))\,, \text{ where } X_n \stackrel{\text{def}}{=} F(1^n)
\end{aligned}
$$

It follows that $|\,\mathsf{Pr}[D'(U_{\ell(k(n))}) = 1] - \mathsf{Pr}[D'(G(U_{k(n)})) = 1]\,|$ equals $\mathsf{E}[\Delta_{A,D}(F(1^n))]$, which implies a weaker version of the proposition (referring to $\mathsf{E}[\Delta_{A,D}(F(1^n))]$ rather than to $\mathsf{E}[|\Delta_{A,D}(F(1^n))|]$).

In order to prove that $\mathsf{E}[|\Delta_{A,D}(F(1^n))|]$ (rather than to $\mathsf{E}[\Delta_{A,D}(F(1^n))]$) is negligible, we need to modify D' a little. We start by assuming, towards the contradiction, that $\mathsf{E}[|\Delta_{A,D}(F(1^n))|] > \varepsilon(n)$ for some non-negligible function ε. On input r (taken from either $U_{\ell(k(n))}$ or $G(U_{k(n)})$), the modified algorithm D' first obtains $x \leftarrow F(1^n)$, as before. Next, using a sample of size $\text{poly}(n/\varepsilon(n))$, it approximates $p_U(x) \stackrel{\text{def}}{=} \mathsf{Pr}[D(x, A(x, U_{\rho(n)}) = 1]$ and $p_G(x) \stackrel{\text{def}}{=} \mathsf{Pr}[D(x, A(x, G'(U_{k(n)})) = 1]$ such that each probability is approximated to within a deviation of $\varepsilon(n)/8$ with negligible error probability (say, $\exp(-n)$). (Note that, so far, the actions of D' only depend on the length of its input r, which determines n.) If these approximations indicate that $p_U(x) \geq p_G(x)$ (equiv., that $\Delta_{A,D} \geq 0$) then D' outputs $D(x, A(x, r'))$ else it outputs $1 - D(x, A(x, r'))$, where r' is the $\rho(|x|)$-bit long prefix of r and we assume without loss of generality that the output of D is in $\{0,1\}$. It follows that

$$
\begin{aligned}
&\mathsf{Pr}[D'(U_{\rho(n)}) = 1 | F(1^n) = x] \;-\; \mathsf{Pr}[D'(G'(U_{k(n)})) = 1 | F(1^n) = x] \\
&\geq\;\; |p_U(x) - p_G(x)| - \frac{\varepsilon(n)}{2} - \exp(-n)
\end{aligned}
$$

where the $\varepsilon(n)/2$ term is due to the maximal typical deviation (i.e., $\varepsilon(n)/4$) of our approximation of $p_U(x) - p_G(x)$, and the $\exp(-n)$ term is due to the rare case that our approximation errs by more than $\varepsilon(n)/4$. Thus, $\mathsf{Pr}[D'(U_{\ell(k(n))}) = 1] - \mathsf{Pr}[D'(G(U_{k(n)})) = 1]$ is lower-bounded by $\mathsf{E}[|\Delta_{A,D}(F(1^n))|] - (\varepsilon(n)/2) - \exp(-n) > \varepsilon(n)/3$, and the proposition follows. ■

Conclusion. Analogous arguments are applied whenever one wishes to prove that an efficient randomized process (be it an algorithm as above or a multi-party computation) preserves its behavior when one replaces true randomness by pseudorandomness as defined above. Thus, given a pseudorandom generator with a large stretch function, *one can considerably reduce the randomness complexity in any efficient application.*

8.3.3 Computational Indistinguishability

In this section we spell-out (and study) the definition of computational indistinguishability that underlies Definition 8.1. The general definition of computational indistinguishability refers to *arbitrary* probability ensembles, where a probability ensemble is an infinite sequence of random variables $\{Z_n\}_{n\in\mathbb{N}}$ such that each Z_n ranges over strings of length bounded by a polynomial in n. We say that $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$ are computationally indistinguishable if for every feasible algorithm A the difference $d_A(n) \stackrel{\rm def}{=} |\mathsf{Pr}[A(X_n)=1] - \mathsf{Pr}[A(Y_n)=1]|$ is a negligible function in n. That is:

Definition 8.4 (computational indistinguishability): *We say that the probability ensembles $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$ are* computationally indistinguishable *if for every probabilistic polynomial-time algorithm D, every positive polynomial p, and all sufficiently large n,*

$$|\mathsf{Pr}[D(X_n)=1] - \mathsf{Pr}[D(Y_n)=1]| \;<\; \frac{1}{p(n)} \tag{8.4}$$

where the probabilities are taken over the relevant distribution (i.e., either X_n or Y_n) *and over the internal coin tosses of algorithm D. The l.h.s. of Eq.* (8.4)*, when viewed as a function of n, is often called the* distinguishing gap of D *where $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$ is understood from the context.*

That is, we can think of D as somebody who wishes to distinguish two distributions (based on a sample given to it), and think of 1 as D's verdict that the sample was drawn according to the first distribution. Saying that the two distributions are computationally indistinguishable means that if D is a feasible procedure then its verdict is not really meaningful (because the verdict is almost as often 1 when the input is drawn from the first distribution as when the input is drawn from the second distribution). We comment that the absolute value in Eq. (8.4) can be omitted without affecting the definition (see Exercise 8.3), and we will often do so without warning.

In Definition 8.1, we required that the probability ensembles $\{G(U_k)\}_{k\in\mathbb{N}}$ and $\{U_{\ell(k)}\}_{k\in\mathbb{N}}$ be computationally indistinguishable. Indeed, an important special case of Definition 8.4 is when one ensemble is uniform, and in such a case we call the other ensemble pseudorandom.

Non-triviality of Computational Indistinguishability. Clearly, any two probability ensembles that are statistically close[4] are computationally indistinguishable. Needless to say, this is a trivial case of computational indistinguishability, which is due to information theoretic reasons. In contrast, as noted in Section 8.2, there exist probability ensembles that are statistically far apart and yet are computationally

[4]Two probability ensembles, $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$, are said to be statistically close if for every positive polynomial p and sufficient large n the variation distance between X_n and Y_n (i.e., $\frac{1}{2}\sum_z |\mathsf{Pr}[X_n=z] - \mathsf{Pr}[Y_n=z]|$) is bounded above by $1/p(n)$.

indistinguishable (see Exercise 8.1). However, at least one of the probability ensembles in Exercise 8.1 is *not* polynomial-time constructible. One non-trivial case of computational indistinguishability in which both ensembles are polynomial-time constructible is provided by the definition of pseudorandom generators (see Exercise 8.4). As we shall see (in Theorem 8.11), the existence of one-way functions implies the existence of pseudorandom generators, which in turn implies the existence of *polynomial-time constructible* probability ensembles that are statistically far apart and yet are computationally indistinguishable. We mention that this sufficient condition is also necessary (see Exercise 8.5).

Indistinguishability by Multiple Samples

The definition of computational indistinguishability (i.e., Definition 8.4) refers to distinguishers that obtain a single sample from one of the two probability ensembles (i.e., $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$). A more general definition refers to distinguishers that obtain several independent samples from such an ensemble.

Definition 8.5 (indistinguishability by multiple samples): *Let $s:\mathbb{N}\rightarrow\mathbb{N}$ be polynomially-bounded. Two probability ensembles, $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$, are* computationally indistinguishable by $s(\cdot)$ samples *if for every probabilistic polynomial-time algorithm, D, every positive polynomial $p(\cdot)$, and all sufficiently large n's*

$$\left|\Pr\left[D(X_n^{(1)},...,X_n^{(s(n))})=1\right]-\Pr\left[D(Y_n^{(1)},...,Y_n^{(s(n))})=1\right]\right| < \frac{1}{p(n)}$$

where $X_n^{(1)}$ through $X_n^{(s(n))}$ and $Y_n^{(1)}$ through $Y_n^{(s(n))}$ are independent random variables, with each $X_n^{(i)}$ identical to X_n and each $Y_n^{(i)}$ identical to Y_n.

It turns out that in the most interesting cases, computational indistinguishability by a single sample implies computational indistinguishability by any polynomial number of samples. One such case is the case of polynomial-time constructible ensembles. We say that the ensemble $\{Z_n\}_{n\in\mathbb{N}}$ is polynomial-time constructible if there exists a polynomial-time algorithm S so that $S(1^n)$ and Z_n are identically distributed.

Proposition 8.6 *Suppose that $X\stackrel{\text{def}}{=}\{X_n\}_{n\in\mathbb{N}}$ and $Y\stackrel{\text{def}}{=}\{Y_n\}_{n\in\mathbb{N}}$ are both polynomial-time constructible, and s be a polynomial. Then, X and Y are computationally indistinguishable by a single sample if and only if they are computationally indistinguishable by $s(\cdot)$ samples.*

Clearly, for every polynomial s, computational indistinguishability by $s(\cdot)$ samples implies computational indistinguishability by a single sample. We now prove that, for efficiently constructible distributions, indistinguishability by a single sample implies indistinguishability by multiple samples. The proof provides a simple demonstration of a central proof technique, known as the *hybrid technique*.

Proof Sketch:[5] To prove that a sequence of independently drawn samples of one distribution is indistinguishable from a sequence of independently drawn samples

[5] For more details see [87, Sec. 3.2.3].

from the other distribution, we consider *hybrid* sequences such that the $i^{\rm th}$ hybrid consists of i samples taken from the first distribution and the rest taken from the second distribution. The "homogeneous" sequences (which we wish to prove to be computational indistinguishable) are the extreme hybrids (i.e., the first and last hybrids considered above). The key observation is that distinguishing the extreme hybrids (towards the contradiction hypothesis) means distinguishing neighboring hybrids, which in turn yields a procedure for distinguishing single samples of the two original distributions (contradicting the hypothesis that these two distributions are indistinguishable by a single sample). Details follow.

Suppose that D distinguishes $s(n)$ samples of one distribution from $s(n)$ samples of the other, with a distinguishing gap of $\delta(n)$. Denoting the $i^{\rm th}$ hybrid by H_n^i (i.e., $H_n^i = (X_n^{(1)},...,X_n^{(i)},Y_n^{(i+1)},...,Y_n^{(s(n))})$), this means that D distinguishes the extreme hybrids (i.e., H_n^0 and $H_n^{s(n)}$) with gap $\delta(n)$. Then D distinguishes a random pair of neighboring hybrids (i.e., D distinguishes the $i^{\rm th}$ hybrid from the $i+1^{\rm st}$ hybrid, for a randomly selected i) with gap at least $\delta(n)/s(n)$. The reason being that

$$\begin{aligned}
&\mathsf{E}_{i\in\{0,...,s(n)-1\}}\left[\mathsf{Pr}[D(H_n^i)=1]-\mathsf{Pr}[D(H_n^{i+1})=1]\right]\\
&= \frac{1}{s(n)}\cdot\sum_{i=0}^{s(n)-1}\left(\mathsf{Pr}[D(H_n^i)=1]-\mathsf{Pr}[D(H_n^{i+1})=1]\right) \qquad (8.5)\\
&= \frac{1}{s(n)}\cdot\left(\mathsf{Pr}[D(H_n^0)=1]-\mathsf{Pr}[D(H_n^{s(n)})=1]\right) = \frac{\delta(n)}{s(n)}
\end{aligned}$$

Using D, we obtain a distinguisher D' of single samples: Given a single sample, D' selects $i\in\{0,...,s(n)-1\}$ at random, generates i samples from the first distribution and $s(n)-i-1$ samples from the second distribution, and invokes D with the $s(n)$-samples sequence obtained when placing the input sample in location $i+1$. Thus, the construction of D' relies on the hypothesis that both proabbility ensembles are polynomial-time constructible. In analyzing D', observe that when the single sample (i.e., the input to D') is taken from the first (resp., second) distribution, algorithm D' invokes D on the $i+1^{\rm st}$ hybrid (resp., $i^{\rm th}$ hybrid). Thus, the distinguishing gap of D' is captured by Eq. (8.5), and the claim follows. □

The hybrid technique – a digest: The hybrid technique constitutes a special type of a "reducibility argument" in which the computational indistinguishability of *complex* ensembles is proven using the computational indistinguishability of *basic* ensembles. The actual reduction is in the other direction: efficiently distinguishing the basic ensembles is reduced to efficiently distinguishing the complex ensembles, and *hybrid* distributions are used in the reduction in an essential way. The following three properties of the construction of the hybrids play an important role in the argument:

1. *The extreme hybrids collide with the complex ensembles*: this property is essential because what we want to prove (i.e., the indistinguishability of the

complex ensembles) relates to the complex ensembles.

2. *Neighboring hybrids are easily related to the basic ensembles*: this property is essential because what we know (i.e., the indistinguishability of the basic ensembles) relates to the basic ensembles. We need to be able to translate our knowledge (i.e., computational indistinguishability) of the basic ensembles to knowledge (i.e., computational indistinguishability) of any pair of neighboring hybrids. Typically, it is required to efficiently transform strings in the range of a basic distribution into strings in the range of a hybrid, so that the transformation maps the first basic distribution to one hybrid and the second basic distribution to the neighboring hybrid. (In the proof of Proposition 8.6, the hypothesis that both X and Y are polynomial-time constructible is instrumental for such an efficient transformation.)

3. *The number of hybrids is small* (i.e., polynomial): this property is essential in order to deduce the computational indistinguishability of extreme hybrids from the computational indistinguishability of each pair of neighboring hybrids. Typically, the provable "distinguishability gap" is inversely proportional to the number of hybrids. Indeed, see Eq. (8.5).

We remark that in the course of an hybrid argument, a distinguishing algorithm referring to the complex ensembles is being analyzed and even invoked on arbitrary hybrids. The reader may be annoyed of the fact that the algorithm "was not designed to work on such hybrids" (but rather only on the extreme hybrids). However, *an algorithm is an algorithm*: once it exists we can invoke it on inputs of our choice, and analyze its performance on arbitrary input distributions.

8.3.4 Amplifying the stretch function

Recall that the definition of pseudorandom generators (i.e., Definition 8.1) makes a minimal requirement regarding their stretch; that is, it is only required that the length of the output of such generators is longer than their input. Needless to say, we seek pseudorandom generators with a significant stretch. It turns out (see Construction 8.7) that pseudorandom generators of any stretch function and in particular of stretch $\ell_1(k) \stackrel{\text{def}}{=} k+1$, are easily converted into pseudorandom generators of any desired (polynomially bounded) stretch function, ℓ. (On the other hand, since pseudorandom generators are required (in Definition 8.1) to run in polynomial time, their stretch must be polynomially bounded.) Thus, when talking about the existence of pseudorandom generators, as in Definition 8.1, we may ignore the stretch function.

Construction 8.7 *Let G_1 be a pseudorandom generator with stretch function $\ell_1(k) = k+1$, and ℓ be any polynomially bounded stretch function that is polynomial-time computable. Let*

$$G(s) \stackrel{\text{def}}{=} \sigma_1\sigma_2\cdots\sigma_{\ell(|s|)} \tag{8.6}$$

where $x_0 = s$ and $x_i\sigma_i = G_1(x_{i-1})$, for $i = 1, ..., \ell(|s|)$. (That is, σ_i is the last bit of $G_1(x_{i-1})$ and x_i is the $|s|$-bit long prefix of $G_1(x_{i-1})$.)

Needless to say, G is polynomial-time computable and has stretch ℓ.

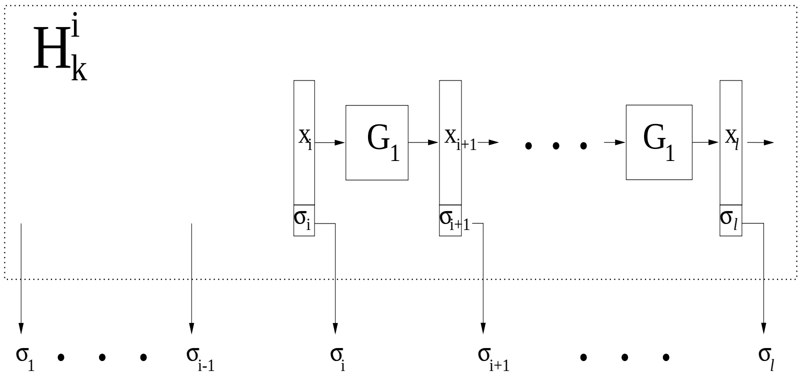


Figure 8.2: Analysis of stretch amplification – the $i^{\rm th}$ hybrid.

Proposition 8.8 *Let G_1 and G be as in Construction 8.7. Then G constitutes a pseudorandom generator.*

Proof Sketch:[6] The proposition is proven using the *hybrid technique*, presented and discussed in Section 8.3.3. Here (for $i = 0, ..., \ell(k)$) we consider the hybrid distributions H_k^i, defined by

$$H_k^i \stackrel{\rm def}{=} U_i^{(1)} \cdot P_{\ell(k)-i}(U_k^{(2)}),$$

where $U_i^{(1)}$ and $U_k^{(2)}$ are independent uniform distributions (over $\{0,1\}^i$ and $\{0,1\}^k$, respectively), and $P_j(x)$ denotes the j-bit long prefix of $G(x)$. (See Figure 8.2.) The extreme hybrids (i.e., H_k^0 and H_k^k) correspond to $G(U_k)$ and $U_{\ell(k)}$, whereas distinguishability of neighboring hybrids can be worked into distinguishability of $G_1(U_k)$ and U_{k+1}. Details follow.

Suppose that algorithm D distinguishes H_k^i from H_k^{i+1} (with some gap $\delta(k)$). Denoting the first $|x|-1$ bits (resp., last bit) of x by $F(x)$ (resp., $L(x)$), we may write $P_j(s) \equiv (L(G_1(s)), P_{j-1}(F(G_1(s))))$ and

$$\begin{aligned}
H_k^i &= U_i^{(1)} \cdot P_{\ell(k)-i}(U_k^{(2)}) \\
&\equiv (U_i^{(1)}, L(G_1(U_k^{(2)})), P_{(\ell(k)-i)-1}(F(G_1(U_k^{(2)})))) \\
H_k^{i+1} &= U_{i+1}^{(1')} \cdot P_{\ell(k)-i-1}(U_k^{(2)}) \\
&\equiv (U_i^{(1)}, L(U_{k+1}^{(2')}), P_{(\ell(k)-i)-1}(F(U_{k+1}^{(2')}))).
\end{aligned}$$

Then, incorporating the generation of $U_i^{(1)}$ and the evaluation of $P_{\ell(k)-i-1}$ into the distinguisher D, we distinguish $(F(G_1(U_k^{(2)})), L(G_1(U_k^{(2)}))) \equiv G_1(U_k)$ from

[6] For more details see [87, Sec. 3.3.3].

$(F(U_{k+1}^{(2')}), L(U_{k+1}^{(2')})) \equiv U_{k+1}$, in contradiction to the pseudorandomness of G_1. Specifically, on input $x \in \{0,1\}^{k+1}$, we uniformly select $r \in \{0,1\}^i$ and output $D(r \cdot L(x) \cdot P_{\ell(k)-i-1}(F(x)))$. Thus, the probability we output 1 on input $G_1(U_k)$ (resp., U_{k+1}) equals $\mathsf{Pr}[D(H_k^i) = 1]$ (resp., $\mathsf{Pr}[D(H_k^{i+1}) = 1]$). A final detail refers to the question which i to use. As usual (when the hybrid technique is used), a random i (in $\{0, ..., k-1\}$) will do. □

8.3.5 Constructions

The constructions surveyed in this section "transform" computational difficulty, in the form of one-way functions, into generators of pseudorandomness. Recall that a *polynomial-time computable function* is called one-way if any efficient algorithm can invert it only with negligible success probability (see Definition 7.1 and Section 7.1 for further discussion). We will actually use hard-core predicates of such functions, and refer the reader to their treatment in Section 7.1.3. Loosely speaking, a *polynomial-time computable* predicate b is called a hard-core of a function f if any efficient algorithm, given $f(x)$, can guess $b(x)$ only with success probability that is negligible better than half. Recall that, for any one-way function f, the inner-product mod 2 of x and r is a hard-core of $f'(x,r) = (f(x), r)$. Finally, we get to the construction of pseudorandom generators.

Proposition 8.9 (A simple construction of pseudorandom generators): *Let b be a hard-core predicate of a polynomial-time computable 1-1 and length-preserving function f. Then, $G(s) \stackrel{\text{def}}{=} f(s) \cdot b(s)$ is a pseudorandom generator.*

Proof Sketch:[7] The $|s|$-bit long prefix of $G(s)$ is uniformly distributed, because f is 1-1 and onto $\{0,1\}^{|s|}$. Hence, the proof boils down to showing that distinguishing $f(s)b(s)$ from $f(s)\cdot\sigma$, where σ is a random bit, yields contradiction to the hypothesis that b is a hard-core of f (i.e., that $b(s)$ is *unpredictable* from $f(s)$). Intuitively, such a distinguisher also distinguishes $f(s)b(s)$ from $f(s) \cdot \overline{b(s)}$, where $\overline{\sigma} = 1 - \sigma$, but distinguishing $f(s) \cdot b(s)$ from $f(s) \cdot \overline{b(s)}$ yields an algorithm for predicting $b(s)$ based on $f(s)$. Details follow.

We start with any potential distinguisher D, and let

$$\delta(k) \stackrel{\text{def}}{=} \mathsf{Pr}[D(G(U_k)) = 1] - \mathsf{Pr}[D(U_{k+1}) = 1].$$

We may assume, without loss of generality, that $\delta(k)$ is non-negative (for infinitely many k's). Using $G(U_k) = f(U_k) \cdot b(U_k)$ and $U_{k+1} \equiv f(U_k) \cdot Z$, where $Z = b(U_k)$ with probability 1/2 and $Z = \overline{b(U_k)}$ otherwise, we have

$$\mathsf{Pr}[D(f(U_k)b(U_k)) = 1] - \mathsf{Pr}[D(f(U_k)\overline{b(U_k)}) = 1] \;=\; 2\delta(k).$$

Consider an algorithm A that, on input y, uniformly selects $\sigma \in \{0,1\}$, invokes $D(y\sigma)$, and outputs σ if $D(y\sigma) = 1$ and $\overline{\sigma}$ otherwise. Then

$$\mathsf{Pr}[A(f(U_k)) = b(U_k)] \;=\; \mathsf{Pr}[D(f(U_k) \cdot \sigma) = 1 \wedge \sigma = b(U_k)]$$

[7] For more details see [87, Sec. 3.3.4].

$$
\begin{aligned}
& + \Pr[D(f(U_k)\cdot\sigma)=0 \wedge \sigma=\overline{b(U_k)}] \\
= \; & \frac{1}{2}\cdot\Big(\Pr[D(f(U_k)\cdot b(U_k))=1] \\
& \quad + 1 \; - \; \Pr[D(f(U_k)\cdot\overline{b(U_k)})=1]\Big)
\end{aligned}
$$

which equals $(1+2\delta(k))/2$. The proposition follows. □

Combining Theorem 7.7, Proposition 8.9 and Construction 8.7, we obtain the following corollary.

Theorem 8.10 (A sufficient condition for the exsitemce of pseudorandom generators): *If there exists 1-1 and length-preserving one-way function then, for every polynomially bounded stretch function ℓ, there exists a pseudorandom generator of stretch ℓ.*

Digest. The key point in the proof of Proposition 8.9 is showing that the (rather obvious) unpredictability of the output of G implies its pseudorandomness. The fact that (next bit) unpredictability and pseudorandomness are equivalent, in general, is proven explicitly in the alternative proof of Theorem 8.10 provided next.

An alternative presentation. Let us take a closer look at the pseudorandom generators obtained by combining Construction 8.7 and Proposition 8.9. For a stretch function $\ell:\mathbb{N}\rightarrow\mathbb{N}$, a 1-1 one-way function f with a hard-core b, we obtain

$$G(s) \stackrel{\text{def}}{=} \sigma_1\sigma_2\cdots\sigma_{\ell(|s|)}\,, \tag{8.7}$$

where $x_0 = s$ and $x_i\sigma_i = f(x_{i-1})b(x_{i-1})$ for $i = 1,...,\ell(|s|)$. Denoting by $f^i(x)$ the value of f iterated i times on x (i.e., $f^i(x) = f^{i-1}(f(x))$ and $f^0(x) = x$), we rewrite Eq. (8.7) as follows

$$G(s) \stackrel{\text{def}}{=} b(s)\cdot b(f(s))\cdots b(f^{\ell(|s|)-1}(s))\,. \tag{8.8}$$

The pseudorandomness of G is established in two steps, using the notion of (next bit) unpredictability. An ensemble $\{Z_k\}_{k\in\mathbb{N}}$ is called unpredictable if any probabilistic polynomial-time machine obtaining a (random)[8] prefix of Z_k fails to predict the next bit of Z_k with probability non-negligibly higher than $1/2$. Specifically, we need to establish the following two results.

1. A **general result** asserting that *an ensemble is pseudorandom if and only if it is unpredictable.* Recall that an ensemble is pseudorandom if it is computationally indistinguishable from a uniform distribution (over bit strings of adequate length).

[8] For simplicity, we define unpredictability as referring to prefices of a random length (distributed uniformly in $\{0,...,|Z_k|-1\}$).

Clearly, pseudorandomness implies polynomial-time unpredictability, but here we actually need the other direction, which is less obvious. Still, using a hybrid argument, one can show that (next-bit) unpredictability implies indistinguishability from the uniform ensemble. For details see Exercise 8.9.

2. A **specific result** asserting that the ensemble $\{G(U_k)\}_{k\in\mathbb{N}}$ is unpredictable *from right to left*. Equivalently, $G'(U_n)$ is polynomial-time unpredictable (from left to right (as usual)), where $G'(s) = b(f^{\ell(|s|)-1}(s)) \cdots b(f(s)) \cdot b(s)$ is the reverse of $G(s)$.

   Using the fact that f induces a permutation over $\{0,1\}^n$, observe that the $(j+1)$-bit long prefix of $G'(U_k)$ is distributed identically to $b(f^j(U_k)) \cdots b(f(U_k)) \cdot b(U_k)$. Thus, an algorithm that predicts the $j+1^{\text{st}}$ bit of $G'(U_n)$ based on the j-bit long prefix of $G'(U_n)$ yields an algorithm that guesses $b(U_n)$ based on $f(U_n)$. For details see Exercise 8.11.

Needless to say, G is a pseudorandom generator if and only if G' is a pseudorandom generator (see Exercise 8.10). We mention that Eq. (8.8) is often referred to as the Blum-Micali Construction.[9]

A general condition for the existence of pseudorandom generators. Recall that given any one-way 1-1 length-preserving function, we can easily construct a pseudorandom generator. Actually, the 1-1 (and length-preserving) requirement may be dropped, but the currently known construction – for the general case – is quite complex.

Theorem 8.11 (On the existence of pseudorandom generators): *Pseudorandom generators exist if and only if one-way functions exist.*

To show that the existence of pseudorandom generators imply the existence of one-way functions, consider a pseudorandom generator G with stretch function $\ell(k) = 2k$. For $x, y \in \{0,1\}^k$, define $f(x,y) \stackrel{\text{def}}{=} G(x)$, and so f is polynomial-time computable (and length-preserving). It must be that f is one-way, or else one can distinguish $G(U_k)$ from U_{2k} by trying to invert and checking the result: Inverting f on its range distribution refers to the distribution $G(U_k)$, whereas the probability that U_{2k} has inverse under f is negligible.

The interesting direction of the proof of Theorem 8.11 is the construction of pseudorandom generators based on any one-way function. In general (when f may not be 1-1) the ensemble $f(U_k)$ may not be pseudorandom, and so Construction 8.9 (i.e., $G(s) = f(s)b(s)$, where b is a hard-core of f) cannot be used *directly*. One idea underlying the construction is to hash $f(U_k)$ to an almost uniform string of length related to its entropy, using Universal Hash Functions. (This is done after guaranteeing, that the logarithm of the probability mass of a value of $f(U_k)$ is

[9] Given the popularity of the term, we deviate from our convention of not specifying credits in the main text. Indeed, this construction originates in [37].

typically close to the entropy of $f(U_k)$.)[10] But "hashing $f(U_k)$ down to length comparable to the entropy" means shrinking the length of the output to, say, $k' < k$. This foils the entire point of stretching the k-bit seed. Thus, a second idea underlying the construction is to compensate for the $k - k'$ loss by extracting these many bits from the seed U_k itself. This is done by hashing U_k, and the point is that the $(k - k')$-bit long hash value does not make the inverting task any easier. Implementing these ideas turns out to be more difficult than it seems, and indeed an alternative construction would be most appreciated.

8.3.6 Non-uniformly strong pseudorandom generators

Recall that we said that truly random sequences can be replaced by pseudorandom ones without affecting any efficient computation. The specific formulation of this assertion, presented in Proposition 8.3, refers to randomized algorithms that take a "primary input" and use a secondary "random input" in their computation. Proposition 8.3 asserts that it is infeasible to find a primary input for which the replacement of a truly random secondary input by a pseudorandom one affects the final output of the algorithm in a noticeable way. This, however, does not mean that such primary inputs do not exist (but rather that they are hard to find). Consequently, Proposition 8.3 falls short of yielding a (worst-case)[11] "derandomization" of a complexity class such as $\mathcal{BPP}$. To obtain such results, we need a stronger notion of pseudorandom generators, presented next. Specifically, we need pseudorandom generators that can fool all polynomial-size circuits (cf. §1.2.4.1), and not merely all probabilistic polynomial-time algorithms.[12]

Definition 8.12 (strong pseudorandom generator – fooling circuits): *A deterministic polynomial-time algorithm G is called a* non-uniformly strong pseudorandom generator *if there exists a* stretch function, *$\ell : \mathbb{N} \to \mathbb{N}$, such that for any family $\{C_k\}_{k\in\mathbb{N}}$ of polynomial-size circuits, for any positive polynomial p, and for all sufficiently large k's*

$$|\,\mathsf{Pr}[C_k(G(U_k)) = 1] \;-\; \mathsf{Pr}[C_k(U_{\ell(k)}) = 1]\,| \;<\; \frac{1}{p(k)}$$

[10] Specifically, given an arbitrary one-way function f', one first constructs f by taking a "direct product" of sufficiently many copies of f'. For example, for $x_1, ..., x_{k^{2/3}} \in \{0,1\}^{k^{1/3}}$, we let $f(x_1, ..., x_{k^{2/3}}) \stackrel{\text{def}}{=} f'(x_1), ..., f'(x_{k^{2/3}})$.

[11] Indeed, Proposition 8.3 yields an *average-case derandomization of* $\mathcal{BPP}$. In particular, for every polynomial-time constructible ensemble $\{X_n\}_{n\in\mathbb{N}}$, every $L \in \mathcal{BPP}$, and every $\varepsilon > 0$, there exists a randomized algorithm A' of randomness complexity $r_\varepsilon(n) = n^\varepsilon$ such that the probability that $A'(X_n) \neq L(X_n)$ is negligible. A corresponding deterministic ($\exp(r_\varepsilon)$-time) algorithm A'' can be obtained, as in the proof of Theorem 8.13, and again the probability that $A''(X_n) \neq L(X_n)$ is negligible, where here the probability is taken only over the distribution of the primary input (represented by X_n). In contrast, worst-case derandomization, as captured by the assertion $\mathcal{BPP} \subseteq \textsc{Dtime}(2^{r_\varepsilon})$, requires that the probability that $A''(X_n) \neq L(X_n)$ is zero.

[12] Needless to say, strong pseudorandom generators in the sense of Definition 8.12 satisfy the basic definition of a pseudorandom generator (i.e., Definition 8.1); see Exercise 8.12.

An alternative formulation is obtained by referring to polynomial-time machines that take advice (Section 3.1.2). Using such pseudorandom generators, we can "derandomize" $\mathcal{BPP}$.

Theorem 8.13 (Derandomization of $\mathcal{BPP}$): *If there exists non-uniformly strong pseudorandom generators then $\mathcal{BPP}$ is contained in $\cap_{\varepsilon>0}\mathrm{DTIME}(t_\varepsilon)$, where $t_\varepsilon(n) \stackrel{\rm def}{=} 2^{n^\varepsilon}$.*

Proof Sketch: Given any $L \in \mathcal{BPP}$ and any $\varepsilon > 0$, we let A denote the decision procedure for L and G denote a non-uniformly strong pseudorandom generator stretching n^ε-bit long seeds into poly(n)-long sequences (to be used by A as secondary input when processing a primary input of length n). We thus obtain an algorithm $A' = A_G$ (as in Construction 8.2). We note that A and A' may differ in their decision on at most finitely many inputs, because otherwise we can use these inputs (together with A) to derive a (non-uniform) family of polynomial-size circuits that distinguishes $G(U_{n^\varepsilon})$ and $U_{\mathrm{poly}(n)}$, contradicting the the hypothesis regarding G. (Specifically, in terms of the proof of Proposition 8.3, the finder F consists of a non-uniform family of polynomial-size circuits that print the "problematic" primary inputs that are hard-wired in them, and the corresponding distinguisher D' is thus also non-uniform.) Incorporating the finitely many "bad" inputs into A', we derive a probabilistic polynomial-time algorithm that decides L while using randomness complexity n^ε.

Finally, emulating A' on each of the 2^{n^ε} possible random choices (i.e., seeds to G), we obtain a deterministic algorithm A'' as required. That is, let $A'(x,r)$ denote the output of algorithm A' on input x when using coins $r \in \{0,1\}^{n^\varepsilon}$. Then $A''(x)$ invokes $A'(x,r)$ on every $r \in \{0,1\}^{n^\varepsilon}$, and rules by majority. □

We comment that stronger results regarding derandomization of $\mathcal{BPP}$ are presented in Section 8.4.

On constructing non-uniformly strong pseudorandom generators. Non-uniformly strong pseudorandom generators (as in Definition 8.12) can be constructed using any one-way function that is hard to invert by any non-uniform family of polynomial-size circuits (as in Definition 7.3), rather than by probabilistic polynomial-time machines. In fact, the construction in this case is simpler than the one employed in the uniform case (i.e., the construction underlying the proof of Theorem 8.11).

8.3.7 Other variants and a conceptual discussion

We first mention two stronger variants on the definition of pseudorandom generators, and conclude this section by highlighting various conceptual issues.

8.3.7.1 Stronger notions

The following two notion represent strengthening of the standard definition of pseudorandom generators (as presented in Definition 8.1). Non-uniform versions of these variants (strengthening Definition 8.12) are also of interest.

Fooling stronger distinguishers. One strengthening of Definition 8.1 amounts to explicitly quantifying the resources (and success gaps) of distinguishers. We chose to bound these quantities as a function of the length of the seed (i.e., k), rather than as a function of the length of the string that is being examined (i.e., $\ell(k)$). For a class of time bounds $\mathcal{T}$ (e.g., $\mathcal{T} = \{t(k) \stackrel{\text{def}}{=} 2^{c\sqrt{k}}\}_{c\in\mathbb{N}}$) and a class of noticeable functions (e.g., $\mathcal{F} = \{f(k) \stackrel{\text{def}}{=} 1/t(k) : t \in \mathcal{T}\}$), we say that a pseudorandom generator, G, is $(\mathcal{T},\mathcal{F})$-strong if for any probabilistic algorithm D having running-time bounded by a function in $\mathcal{T}$ (applied to k)[13], for any function f in $\mathcal{F}$, and for all sufficiently large k's

$$|\,\mathsf{Pr}[D(G(U_k)) = 1] \;-\; \mathsf{Pr}[D(U_{\ell(k)}) = 1]\,| \;<\; f(k)$$

An analogous strengthening may be applied to the definition of one-way functions. Doing so reveals the weakness of the construction that underlies the proof of Theorem 8.11: It only implies that for some $\varepsilon > 0$ ($\varepsilon = 1/5$ will do), for any $\mathcal{T}$ and $\mathcal{F}$, the existence of "$(\mathcal{T},\mathcal{F})$-strong one-way functions" implies the existence of $(\mathcal{T}',\mathcal{F}')$-strong pseudorandom generators, where $\mathcal{T}' = \{t'(k) \stackrel{\text{def}}{=} t(k^\varepsilon)/\text{poly}(k) : t \in \mathcal{T}\}$ and $\mathcal{F}' = \{f'(k) \stackrel{\text{def}}{=} \text{poly}(k) \cdot f(k^\varepsilon) : f \in \mathcal{F}\}$. What we *would like* to have is an analogous result with $\mathcal{T}' = \{t'(k) \stackrel{\text{def}}{=} t(k)/\text{poly}(k) : t \in \mathcal{T}\}$ and $\mathcal{F}' = \{f'(k) \stackrel{\text{def}}{=} \text{poly}(k) \cdot f(k) : f \in \mathcal{F}\}$.

Pseudorandom Functions. Pseudorandom generators allow to efficiently generate long pseudorandom sequences from short random seeds. Pseudorandom functions (defined in Appendix C.3.3) are even more powerful: They allow *efficient direct access* to a huge pseudorandom sequence, which is not even feasible to scan bit-by-bit. Put in other words, pseudorandom functions can replace truly random functions in any efficient application (e.g., most notably in cryptography). We mention that pseudorandom functions can be constructed from any pseudorandom generator (see Appendix C.3.3), and found many applications in cryptography (see Appendix C.3.3, C.5.2, and C.6.2). Pseudorandom functions have been used to derive negative results in computational learning theory [214] and in complexity theory (cf., Natural Proofs [176]).

8.3.7.2 Conceptual Discussion

Whoever does not value preoccupation with thoughts, can skip this chapter.

Robert Musil, The Man without Qualities, Chap. 28

We highlight several conceptual aspects of the foregoing computational approach to randomness. Some of these aspects are common to other instantiation of the general paradigm (esp., the one presented in Section 8.4).

[13] That is, when examining a sequence of length $\ell(k)$ algorithm D makes at most $t(k)$ steps, where $t \in \mathcal{T}$.

Behavioristic versus Ontological. The behavioristic nature of the computational approach to randomness is best demonstrated by confronting this approach with the Kolmogorov-Chaitin approach to randomness. Loosely speaking, a string is *Kolmogorov-random* if its length equals the length of the shortest program producing it. This shortest program may be considered the "true explanation" to the phenomenon described by the string. A Kolmogorov-random string is thus a string that does not have a substantially simpler (i.e., shorter) explanation than itself. Considering the simplest explanation of a phenomenon may be viewed as an ontological approach. In contrast, considering the effect of phenomena on certain objects, as underlying the definition of pseudorandomness, is a behavioristic approach. Furthermore, there exist probability distributions that are not uniform (and are not even statistically close to a uniform distribution) and nevertheless are indistinguishable from a uniform distribution (by any efficient method). Thus, distributions that are ontologically very different, are considered equivalent by the behavioristic point of view taken in the definition of computational indistinguishability.

A relativistic view of randomness. We have defined pseudorandomness in terms of its observer. Specifically, we have considered the class of efficient (i.e., polynomial-time) observers and defined as pseudorandom objects that look random to any observer in that class. In subsequent sections, we shall consider restricted classes of such observers (e.g., space-bounded polynomial-time observers and even very restricted observers that merely apply specific tests such as linear tests or hitting tests). Each such class of observers gives rise to a different notion of pseudorandomness. Furthermore, the general paradigm (of pseudorandomness) explicitly aims at distributions that are not uniform and yet are considered as such from the point of view of certain observers. Thus, our entire approach to pseudorandomness is relativistic and subjective (i.e., depending on the abilities of the observer).

Randomness and Computational Difficulty. Pseudorandomness and computational difficulty play dual roles: The general paradigm of pseudorandomness relies on the fact that putting computational restrictions on the observer gives rise to distributions that are not uniform and still cannot be distinguished from uniform. Thus, the pivot of the entire approach is the computational difficulty of distinguishing pseudorandom distributions from truly random ones. Furthermore, many of the constructions of pseudorandom generators rely on either conjectures or facts regarding computational difficulty (i.e., that certain computations that are hard for certain classes). For example, one-way functions were used to construct general-purpose pseudorandom generators (i.e., those working in polynomial-time and fooling all polynomial-time observers). Analogously, as we shall see in Section 8.4.3, the fact that parity function is hard for polynomial-size constant-depth circuits can be used to generate (highly non-uniform) sequences that fool such circuits.

Randomness and Predictability. The connection between pseudorandomness and unpredictability (by efficient procedures) plays an important role in the analysis of several constructions (cf. Sections 8.3.5 and 8.4.2). We wish to highlight the intuitive appeal of this connection.

8.4 Derandomization of time-complexity classes

Let us take a second look at the proof of Theorem 8.13: A pseudorandom generator was used to shrink the randomness complexity of a BPP-algorithm, and derandomization was achieved by scanning all possible seeds to the generator. A key observation regarding this process is that there is no point in insisting that the pseudorandom generator runs in time polynomial in its seed length. Instead, it suffices to require that the generator runs in time exponential in its seed length, because we are incurring such an overhead anyhow due to the scanning of all possible seeds. Furthermore, in this context, the running-time of the generator may be larger than the running time of the algorithm, which means that the generator need only fool distinguishers that take less steps than the generator. These considerations motivate the following definition.

8.4.1 Definition

Recall that in order to "derandomize" a probabilistic polynomial-time algorithm A, we first obtain a functionally equivalent algorithm A_G (as in Construction 8.2) that has (significantly) smaller randomness complexity. Algorithm A_G has to maintain A's input-output behavior on all (but finitely many) inputs. Thus, the set of the relevant distinguishers (considered in the proof of Theorem 8.13) is the set of all possible circuits obtained from A by hard-wiring each of the possible inputs. Such a circuit, denoted C_x, emulates the execution of algorithm A on input x, when using the circuit's input as the algorithm's internal coin tosses (i.e., $A(x,r) = C_x(r)$). Furthermore, the size of C_x is polynomial in the running-time of A on input x, and the length of the input to C_x is linear in the running-time of A (on input x).[14] For simplicity, let's say that the size of C_x is quadratic in the running-time of A on input x. Thus, the pseudorandom generator in use (i.e., G) needs to fool each of these possible circuits. Recalling that we may allow the generator to run in exponential time (in the length of its own input)[15], we arrive at the following definition.

[14] Indeed, if algorithm A is represented as a Turing machine then C_x has size that is at most quadratic (and in fact even almost-linear [167]) in the running-time of A on input x, which in turn means that C_x has size that is at most quadratic (or almost linear) in the length of its own input. We note that most sources use the fictitious convention by which the circuit size equals the length of its input, which can be justified by considering a suitably padded input.

[15] Actually, in Definition 8.14 we allow the generator to run in time $\text{poly}(2^k\ell(k))$, rather than $\text{poly}(2^k)$. This is done in order not to rule out trivially generators of super-exponential stretch (i.e., $\ell(k) = 2^{\omega(k)}$). However (see Exercise 8.13), the condition in Eq. (8.9) does not allow for super-exponential stretch, and so in retrospect the two formulations are equivalent (because $\text{poly}(2^k\ell(k)) = \text{poly}(2^k)$ for $\ell(k) = 2^{O(k)}$).

Definition 8.14 (pseudorandom generator for derandomizing $\mathrm{BP}\textsc{time}(\cdot)$)[16]: *Let $\ell:\mathbb{N}\to\mathbb{N}$ be a 1-1 function. A* canonical derandomizer of stretch ℓ *is a deterministic algorithm G of time complexity upper-bounded by* $\mathrm{poly}(2^k\cdot\ell(k))$ *such that for every circuit D_k of size $\ell(k)^2$ it holds that*

$$|\Pr[D_k(G(U_k))=1] - \Pr[D_k(U_{\ell(k)})=1]| < \frac{1}{6} \tag{8.9}$$

The circuits D_k are potential distinguishers, which are given inputs of length $\ell(k)$. When seeking to derandomize an algorithm A of time-complexity t, the aforementioned $\ell(k)$-bit long inputs represent possible random-inputs of A when invoked on a generic (primary) input of length $n=t^{-1}(\ell(k))$. That is, letting $D_k(r)=A(x,r)$ for some choice of $x\in\{0,1\}^n$, where $|r|=t(n)=\ell(k)$, and Eq. (8.9) implies that $A_G(x)$ maintains the majority vote of $A(x)$. The straightforward deterministic emulation of A_G takes time $2^k\cdot(\mathrm{poly}(2^k\cdot\ell(k))+t(n))$, which is upper-bounded by $\mathrm{poly}(2^k\cdot\ell(k))=\mathrm{poly}(2^{\ell^{-1}(t(n))}\cdot t(n))$. The following proposition is easy to establish.

Proposition 8.15 *If there exists a canonical derandomizer of stretch ℓ then, for every time-constructible $t:\mathbb{N}\to\mathbb{N}$, it holds that $\mathrm{BP}\textsc{time}(t)\subseteq\mathrm{D}\textsc{time}(T)$, where $T(n)=\mathrm{poly}(2^{\ell^{-1}(t(n))}\cdot t(n))$.*

Proof Sketch: Just follow the proof of Theorem 8.13, noting that the adequate value of k (i.e., $k=\ell^{-1}(t(n))$) can be determined easily (e.g., by invoking $G(1^i)$ for $i=1,...,k$, using the fact that $\ell:\mathbb{N}\to\mathbb{N}$ is 1-1). Note that the complexity of the deterministic procedure is dominated by the 2^k invocations of $A_G(x,s)=A(x,G(s))$, where $s\in\{0,1\}^{\ell^{-1}(t(|x|))}$, and each of these invocations takes time $\mathrm{poly}(2^k\cdot\ell(k))+t(n)=\mathrm{poly}(2^{\ell^{-1}(t(n))}\cdot t(n))$. □

The goal. In light of Proposition 8.15, we seek canonical derandomizers with stretch that is as big as possible. The stretch cannot be super-exponential (in fact $\ell(k)=O(2^k)$), because a circuit of size $O(2^k\cdot\ell(k))$ may violate Eq. (8.9) (see Exercise 8.13) whereas for $\ell(k)=\omega(2^k)$ it holds that $O(2^k\cdot\ell(k))<\ell(k)^2$. Thus, our goal is to construct canonical derandomizer with stretch $\ell(k)=2^{\Omega(k)}$. Such canonical derandomizers will allow for a "full derandomization of $\mathcal{BPP}$":

Theorem 8.16 *If there exists a canonical derandomizer of stretch $\ell(k)=2^{\Omega(k)}$ for $\mathrm{BP}\textsc{time}(\cdot)$, then $\mathcal{BPP}=\mathcal{P}$.*

Proof: Using Proposition 8.15, we get $\mathrm{BP}\textsc{time}(t)\subseteq\mathrm{D}\textsc{time}(T)$, where $T(n)=\mathrm{poly}(2^{\ell^{-1}(t(n))}\cdot t(n))=\mathrm{poly}(t(n))$. ■

[16]Fixing a model of computation, we denote by $\mathrm{BP}\textsc{time}(t)$ the class of decision problems that are solvable by a randomized algorithm of time complexity t that has two-sided error 1/3.

Reflections. We stress that a canonical derandomizer G was defined in a way that allows it to have time complexity t_G that is larger than the size of the circuits that it fools (i.e., $t_G(k) > \ell(k)^2$ is allowed). Furthermore, $t_G(k) > 2^k$ was also allowed. Thus, if indeed $t_G(k) = 2^{\Omega(k)}$ (as is the case in Section 8.4.2) then *$G(U_k)$ can be distinguished from $U_{\ell(k)}$ in time* $2^k \cdot t_G(k) = \text{poly}(t_G(k))$ (greater than $\ell(k)^2$), by trying all possible seeds. In contrast, for a general-purpose pseudorandom generator G (as discussed in Section 8.3) it holds that $t_G(k) = \text{poly}(k)$, while *for every polynomial p it holds that $G(U_k)$ is indistinguishable from $U_{\ell(k)}$ in time $p(t_G(k))$*.

8.4.2 Construction

The fact that canonical derandomizers are allowed to be more complex than the corresponding distinguisher makes *some* of the techniques of Section 8.3 inapplicable in the current context. On the other hand, the techniques developed below are inapplicable to Section 8.3. Amazingly enough, the pseudorandomness (or rather the next-bit unpredictability) of the following generators hold even when the "observer" is given the seed (capitalizing on the fact that the observer's time-complexity does not allow running the generator).

As in Section 8.3.5, the construction surveyed below transforms computational difficulty into pseudorandomness, except that here both computational difficulty and pseudorandomness are of a somewhat different form than in Section 8.3.5. Specifically, here we use Boolean predicates that are computable in exponential-time but are T-inapproximated for some exponential function T (see Definition 7.9 in Section 7.2). That is, for constants $c, \varepsilon > 0$ and all but finitely many m, the (residual) predicate $f : \{0,1\}^m \to \{0,1\}$ is computable in time 2^{cm} but for any circuit C of size $2^{\varepsilon m}$ it holds that $\Pr[C(U_m) = f(U_m)] < \frac{1}{2} + 2^{-\varepsilon m}$. (Needless to say, $\varepsilon < c$.) Recall that such predicates exist under the assumption that $\mathcal{E}$ has (almost-everywhere) exponential circuit complexity (see Theorem 7.19 for an exact formulation). With these preliminaries, we turn to the construction of canonical derandomizers with exponential stretch.

Construction 8.17 (The Nisan-Wigderson Construction):[17] *Let $f : \{0,1\}^m \to \{0,1\}$ and $S_1, ..., S_\ell$ be a sequence of m-subsets of $\{1, ..., k\}$. Then, for $s \in \{0,1\}^k$, we let*

$$G(s) \stackrel{\text{def}}{=} f(s_{S_1}) \cdots f(s_{S_\ell}) \tag{8.10}$$

where s_S denotes the projection of s on the bit locations in $S \subseteq \{1, ..., |s|\}$; that is, for $s = \sigma_1 \cdots \sigma_k$ and $S = \{i_1, ..., i_m\}$, we have $s_S = \sigma_{i_1} \cdots \sigma_{i_m}$.

Letting k vary and $\ell, m : \mathbb{N} \to \mathbb{N}$ be functions of k, we wish G to be a canonical derandomizer and $\ell(k) = 2^{\Omega(k)}$. Obvious necessary conditions for this to happen include the requirement that the sets be distinct and hence $m(k) = \Omega(k)$; consequently, f must be computable in exponential-time. Furthermore, the sequence of

[17] Given the popularity of the term, we deviate from our convention of not specifying credits in the main text. This construction originates in [160, 163].

sets $S_1, ..., S_{\ell(k)}$ must be constructible in poly(2^k) time. Intuitively, it is desirable to use a set system with small pairwise intersections (because this restricts the overlap among the various inputs to which f is applied), and a function f that is strongly inapproximable (i.e., T-inapproximble for some exponential function T). Interestingly, these conditions are essentially sufficient.

Theorem 8.18 (analysis of Construction 8.17): *Let $\alpha, \beta, \gamma, \varepsilon > 0$ be constants satisfying $\varepsilon > (2\alpha/\beta) + \gamma$, and $\ell, m, T : \mathbb{N} \to \mathbb{N}$ satisfy $\ell(k) = 2^{\alpha k}$, $m(k) = \beta k$, and $T(n) = 2^{\varepsilon n}$. Suppose that the following two conditions hold:*

1. *There exists an exponential-time computable function $f : \{0,1\}^* \to \{0,1\}$ that is T-inapproximable.* (See Definition 7.9.)
2. *There exists an exponential-time computable function $S : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that $|S(k,i)| = m(k)$ for every k and $i = 1, ..., \ell(k)$, and $|S(k,i) \cap S(k,j)| \leq \gamma \cdot m(k)$ for every k and $i \neq j$.*

Then using G as defined in Construction 8.17, with $S_i = S(k,i)$, yields a canonical derandomizer with stretch ℓ.

For any $\gamma > 0$, a function S as in Condition 2 does exist (see Exercise 8.14), with some $m(k) = \Omega(k)$ and $\ell(k) = 2^{\Omega(k)}$. Combining such S with Theorems 7.19 and 8.18, we obtain a canonical derandomizer with exponential stretch based on the assumption that $\mathcal{E}$ has (almost-everywhere) exponential circuit complexity.[18] Combining this with Theorem 8.16, we get the first item of the following theorem.

Theorem 8.19 (Derandomization of BPP, revisited):

1. *Suppose that there exists a set $S \in \mathcal{E}$ having almost-everywhere exponential circuit complexity* (i.e., there exists a constant $\varepsilon > 0$ such that, for all but finitely many m's, any circuit that correctly decides S on $\{0,1\}^m$ has size at least $2^{\varepsilon m}$). *Then, $\mathcal{BPP} = \mathcal{P}$.*
2. *Suppose that for every polynomial p there exists a set $S \in \mathcal{E}$ having circuit complexity that is almost-everywhere greater than p. Then $\mathcal{BPP}$ is contained in $\cap_{\varepsilon>0}$*DTIME*(t_ε), where $t_\varepsilon(n) \stackrel{\text{def}}{=} 2^{n^\varepsilon}$.*

Part 2 is proved (in Exercise 8.18) by using a generalization of Theorem 8.18, which in turn is provided in Exercise 8.17. We note that Part 2 of Theorem 8.19 superseeds Theorem 8.13 (see Exercise 7.14). The two parts of Theorem 8.19 exhibit two extreme cases: Part 1 (often referred to as the "high end") assumes an extremely strong circuit lower-bound and yields "full derandomization" (i.e., $\mathcal{BPP} = \mathcal{P}$), whereas Part 2 (often referred to as the "low end") assumes an extremely weak

[18] Specifically, starting with a function having circuit complexity at least $\exp(\varepsilon_0 m)$, we apply Theorem 7.19 obtaining a T-inapproximble predicate for $T(m) = 2^{\varepsilon m}$, where the constant $\varepsilon \in (0, \varepsilon_0)$ depends on the constant ε_0. Next, we set $\gamma = \varepsilon/2$ and invoke Exercise 8.14, which determines $\alpha, \beta > 0$ such that $\ell(k) = 2^{\alpha k}$ and $m(k) = \beta k$. In fact, $\beta = \gamma/2$ and $\alpha = \gamma\beta/10$, hence $(2\alpha/\beta) + \gamma < \varepsilon$.

circuit lower-bound and yields weak but meaningfull derandomization. Intermediate results (relying on intermediate lower-bound assumptions) can be obtained analogous to Exercise 8.18, but tight trade-offs are obtained differently (cf., [210]).

Proof of Theorem 8.18: Using the time complexity bounds on f and S, it follows that G can be computed in exponential time. Out focus is on showing that $\{G(U_k)\}$ cannot be distinguished from $\{U_{\ell(k)}\}$ by circuits of size $\ell(k)^2$; that is, that G satisfies Eq. (8.9). In fact, we will prove that this holds for $G'(s) = s \cdot G(s)$; that is, G fools such circuits even if they are given the seed as auxiliary input. (Indeed, these circuits are smaller than the running time of G, and so they cannot just evaluate G on the given seed.)

We start with the intuition underlying the proof. As a warm-up suppose that the sets (i.e., $S(k,i)$'s) used in the construction are disjoint. In such a case (which is indeed impossible becuase $k < \ell(k) \cdot m(k)$), the pseudorandomness of $G(U_k)$ would follow easily from the inapproximablity of f, because in this case G consists of applying f to non-overlaping parts of the seed (see Exercise 8.15). In the actual construction being analyzed here, the sets (i.e., $S(k,i)$'s) are not disjoint but have relatively small pairwise intersection, which means that G applies f on parts of the seed that have relatively small overlap. Intuitively, such small overlaps guarantee that the values of f on the corresponding inputs are "computationally independent" (i.e., having the value of f at one input does not help to approximate the value of f at another input). This intuition will be backed by showing that the former values can be computed at a relatively small computational cost. With this intuition in mind, we now turn to the actual proof.

The proof that G' fools circuits of size $\ell(k)^2$ utilizes the relation between pseudorandomness and unpredictability. Specifically, as detailed in Exercise 8.16, any circuit that distinguishes $G'(U_k)$ from $U_{\ell(k)+k}$ with gap $1/6$, yields a next-bit predictor of similar size that succeeds in predicting the next bit with probability at least $\frac{1}{2} + \frac{1}{6\ell'(k)} > \frac{1}{2} + \frac{1}{7\ell(k)}$, where the $\ell'(k) = \ell(k) + k < (1 + o(1))\ell(k)$ factor is introduced by the hybrid technique (cf. Eq. (8.5)). Furthermore, given the non-uniform setting of the current proof, we may fix a bit location $i+1$ for prediction, rather than analyzing the prediction at a random bit location. Indeed, $i \geq k$ must hold, because the first k bits of $G'(U_k)$ are uniformly distributed. In the rest of the proof, we transform such a predictor into a circuit that approximates f better than allowed by the hypothesis (regarding the inapproximability of f).

Assuming that a small circuit C' can predict the $i+1^{\text{st}}$ bit of $G'(U_k)$, when given the previous i bits, we construct a small circuit C for approximating $f(U_{m(k)})$ on input $U_{m(k)}$. The point is that the $i+1^{\text{st}}$ bit of $G'(s)$ equals $f(s_{S(k,j+1)})$, where $j = i - k \geq 0$, and so C' approximates $f(s_{S(k,j+1)})$ based on $s, f(s_{S(k,1)}), ..., f(s_{S(k,j)})$, where $s \in \{0,1\}^k$ is uniformly distributed. This is the type of thing that we are after, except that the circuit we seek may only get $s_{S(k,j+1)}$ as input.

The first observation is that C' maintains its advantage when we fix the best choice for the bits of s that are not at bit locations $S_{j+1} = S(k, j+1)$ (i.e., the bits $s_{[k]\setminus S_{j+1}}$). That is, by an averaging argument, it holds that

$$\begin{aligned}
&\max_{s'\in\{0,1\}^{k-m(k)}} \{\mathsf{Pr}_{s\in\{0,1\}^k}[C'(s,f(s_{S_1}),...,f(s_{S_j}))=f(s_{S_{j+1}})\,|\,s_{[k]\setminus S_{j+1}}=s']\} \\
&\geq\ p' \stackrel{\rm def}{=} \mathsf{Pr}_{s\in\{0,1\}^k}[C'(s,f(s_{S_1}),...,f(s_{S_j}))=f(s_{S_{j+1}})].
\end{aligned}$$

Recall that by the hypothesis $p'>\frac{1}{2}+\frac{1}{7\ell(k)}$. Hard-wiring the fixed string s' into C', and letting $\pi(x)$ denote the (unique) string s satisfying $s_{S_{j+1}}=x$ and $s_{[k]\setminus S_{j+1}}=s'$, we obtain a circuit C'' that satisfies

$$\mathsf{Pr}_{x\in\{0,1\}^m}[C''(x,f(\pi(x)_{S_1}),...,f(\pi(x)_{S_j}))=f(x)]\ \geq\ p'.$$

The circuit C'' is almost what we seek. The only problem is that C'' gets as input not only x, but also $f(\pi(x)_{S_1}),...,f(\pi(x)_{S_j})$, whereas we seek an appproximator of $f(x)$ that only gets x.

The key observation is that each of the "missing" values $f(\pi(x)_{S_1}),...,f(\pi(x)_{S_j})$ depend only on a relatively small number of the bits of x. This fact is due to the hypothesis that $|S_t\cap S_{j+1}|\leq\gamma\cdot m(k)$ for $t=1,...,j$, which means that $\pi(x)_{S_t}$ is an $m(k)$-bit long string in which $m_t\stackrel{\rm def}{=}|S_t\cap S_{j+1}|$ bits are projected from x and the rest are projected from the *fixed* string s'. Thus, given x, the value $f(\pi(x)_{S_t})$ can be computed by a (trivial) circuit of size $\widetilde{O}(2^{m_t})$; that is, by a circuit implementing a look-up table on m_t bits. Using all these circuits (together with C''), we will obtain the desired approximator of f. Details follow.

We obtain the desired circuit C, which depends on the index j and the string s' that are fixed as in the foregoing analysis. On input $x\in\{0,1\}^m$, the circuit computes the values $f(\pi(x)_{S_1}),...,f(\pi(x)_{S_j})$, invokes C'' on input x and these values, and outputs the answer as a guess for $f(x)$. That is,

$$C(x)=C''(x,f(\pi(x)_{S_1}),...,f(\pi(x)_{S_j}))=C'(\pi(x),f(\pi(x)_{S_1}),...,f(\pi(x)_{S_j})).$$

By the foregoing analysis, $\mathsf{Pr}_x[C(x)=f(x)]\geq p'>\frac{1}{2}+\frac{1}{T(m)}$, where the second inequality is due to $T(m(k))=2^{\varepsilon m(k)}=2^{\varepsilon\beta k}\gg 2^{2\alpha k}\gg 7\ell(k)$. The size of C is upper-bounded by $\ell(k)^2+\ell(k)\cdot\widetilde{O}(2^{\gamma\cdot m(k)})\ll\widetilde{O}(\ell(k)^2\cdot 2^{\gamma\cdot m(k)})\ll T(m(k))$, where the second inequality is due to $T(m(k))=2^{\varepsilon m(k)}\gg\widetilde{O}(2^{2\alpha\cdot k+\gamma\cdot m(k)})$ and $\ell(k)=2^{\alpha k}$. Thus, we derived a contradiction to the hypothesis that f is T-inapproximable. ■

8.4.3 Variants and a conceptual discussion

The Nisan–Wigderson Construction (Construction 8.17) is actually a general framework, which can be instantiated in various ways. We start this section by briefly reviewing some of these instantiations and end it with a conceptual discussion regarding derandomization.

Derandomization of constant-depth circuits. Using (Construction 8.17 with) a different setting of parameters and the parity function in the role of the inapproximable predicate (i.e., inapproximable by "small" constant-depth circuits), one can obtain pseudorandom generators that fool "small" constant-depth circuits (see [160]). The analysis of the modified construction proceeds very much like the proof of Theorem 8.18. One important observation is that incorporating the (straightforward) circuits that compute $f(\pi(x)_{S_i})$ into the distinguishing circuit only increases its depth by two levels. The resulting pseudorandom generator, which use a seed of polylogarithmic length (equiv., $\ell(k) = \exp(k^{1/O(1)})$), can be used for derandomizing $\mathcal{RAC}^0$ (i.e., random $\mathcal{AC}^0$), analogously to Theorem 8.16. In other words, we can *deterministically* approximate, in quasi-polynomial-time and up-to an additive error, the fraction of inputs that satisfy a given (constant-depth) circuit. Specifically, for any constant d, given a depth-d circuit C, one can approximate the fraction of the inputs that satisfy C (i.e., cause C to evaluate to 1) to within any *additive constant error*[19] in time $\exp(\text{poly}(\log|C|))$, where the polynomial depends on d. Providing a deterministic polynomial-time aproximation, even in the case $d = 2$ (i.e., CNF/DNF formulae) is an open problem.

Derandomization of probabilistic proof systems. A different (and more surprising) instantiation of Construction 8.17 utilizes predicates that are hard for small circuits having oracle access to $\mathcal{NP}$. The result is a pseudorandom generator robust against two-move public-coin interactive proofs (which are as powerful as constant-round interactive proofs). The key observation is that the above proof provides a black-box procedure for approximating the underlying predicate when given oracle access to a distinguisher (and this procedure in valid also in case the distinguisher is a non-deterministic machine). Thus, under suitably strong (and yet plausible) assumptions, constant-round interactive proofs collapse to $\mathcal{NP}$. We note that a stronger result, which deviates from the foregoing framework, has been subsequently obtained (cf. [155]).

An even more radical instantiation of Construction 8.17 was used to obtain explicit constructions of randomness extractors (see Section D.4). In addition to the foregoing observation, one also utilizes the fact that the generator itself uses the predicate as a black-box. Further discussion appears in Section D.4.

A conceptual discussion regarding derandomization

Part 1 of Theorem 8.19 is often summarized by saying that (under some reasonable assumptions) *randomness is useless*. We believe that this interpretation is wrong even within the restricted context of traditional complexity classes, and is bluntly wrong if taken outside of the latter context. Let us elaborate.

[19]We mention that in the special case of approximating the number of satisfying assignment of a DNF formula, *relative error* approximations can be obtained by employing a deterministic reduction to the case of additive constant error (see §6.2.2.1). Thus, using a pseudorandom generator that fools DNF formulae, we can deterministically obtain a relative (rather than additive) error approximation to the number of satisfying assignment in a given DNF formula.

Taking a closer look at the proof of (the underlying) Theorem 8.16, we note that a randomized algorithm A of time complexity t is emulated by a deterministic algorithm A' of time complexity $t' = \text{poly}(t)$. Further noting that $A' = A_G$ invokes A and the canonical derandomizer G for a number of times that must exceed t, we infer that $t' > t^2$ must hold. Thus, derandomization via (Part 1 of) Theorem 8.19 is not really for free.

More importantly, we note that derandomization is not possible in various distributed settings, when both parties may protect their conflicting interests by employing randomization. Notable examples include most cryptographic primitives (e.g., encryption) as well as most types of probabilistic proof systems (e.g., PCP). For further discussion see Chapter 9 and Appendix C. Additional settings where randomness makes a difference (either between impossibility and possibility or between prohibited and affordable cost) include distributed computing (see [15]), communication complexity (see [138]), parallel architectures (see [141]), sampling (see Section D.3), and property testing (see Section 10.1.2).

8.5 Space Pseudorandom Generators

In the previous two sections we have considered generators the output of which is indistinguishable by any efficient procedures. The latter were modeled by time-bounded computations; specifically, polynomial-time computations. A finer characterization of time-bounded computations is obtained by considering their space-complexity (i.e., restricting the space-complexity of time-bounded computations). In contrast to the definitions of pseudorandom generators that were considered in Sections 8.3 and 8.4, the existence of pseudorandom generators that fool space-bounded distinguishers can be established without relying on computational assumptions.

8.5.1 Definitional issues

Unfortunately, natural notions of space-bounded computations are quite subtle, especially when non-determinism or randomization are concerned (see Sections 5.3 and 6.1.4). Two major issues are *time bounds* and *access to the random tape*.

1. Time bound: The question is whether or not one restricts the space-bounded machines to run in time-complexity that is at most exponential in the space-complexity.[20] Recall that such an upper-bound follows automatically in the deterministic case, and can be assumed without loss of generality in the non-deterministic case *but not in the randomized case*.

   Indeed, as in Section 6.1.4, we do postulate the aforementioned time-bound.

2. Access to the random tape: The question is whether whether the space-bounded machine has one-way or two-way access to the randomness tape.

[20] Alternatively, one can ask whether these machines must always halt or only halt with probability approaching 1. It can be shown that the only way to ensure "absolute halting" is to have time-complexity that is at most exponential in the space-complexity.

(Allowing two-way access means that the randomness is recorded for free; that is, without being accounted for in the space-bound; see discussions in Sections 5.3 and 6.1.4.) Recall that one-way access to the randomness tape corresponds to the natural model of on-line randomized machine (which determines its moves based on its internal coin tosses).

Again, as in Section 6.1.4, we consider one-way access.[21]

In accordance with the resulting definition of randomized space-bounded computation, we consider space-bounded distinguishers that have a one-way access to the input sequence that they examine. Since all known constructions remain valid also when these distinguishers are non-uniform (and since non-uniform distinguishers arise anyhow in derandomization), we use this stronger notion here.[22]

In the context of non-uniform algorithms that have one-way access to their input, we may assume, without loss of generality, that the running-time of such algorithms equals the length of their input, denoted $\ell = \ell(k)$. Thus, we define a non-uniform machine of space $s:\mathbb{N}\to\mathbb{N}$ as a family, $\{D_k\}_{k\in\mathbb{N}}$, of directed layered graphs such that D_k has at most $2^{s(k)}$ vertices at each layer, and labeled directed edges from each layer to the next layer.[23] Each vertex has two (possibly parallel) outgoing directed edges, one labeled 0 and the other labeled 1, and there is a single vertex in the first layer of D_k. The result of the computation of such a machine, on an input of adequate length (i.e., length ℓ where D_k has $\ell+1$ layers), is defined as the vertex (in last layer) reached when following the sequence of edges that are labelled by the corresponding bits of the input. That is, on input $x = x_1 \cdots x_\ell$, for $i = 1, ..., \ell$, we move from the vertex reached in the i^{th} layer by using the outgoing edge labelled x_i (thus reaching a vertex in the $i+1^{\text{st}}$ layer). Using a fixed partition of the vertices of the last layer, this defines a natural notion of decision (by D_k); that is, we write $D_k(x) = 1$ if on input x machine D_k reached a vertex that belongs to the first part of the aforementioned partition.

Definition 8.20 (Indistinguishability by space-bounded machines):

- *For a non-uniform machine, $\{D_k\}_{k\in\mathbb{N}}$, and two probability ensembles, $\{X_k\}_{k\in\mathbb{N}}$ and $\{Y_k\}_{k\in\mathbb{N}}$, the function $d:\mathbb{N}\to[0,1]$ defined as*

$$d(k) \stackrel{\text{def}}{=} |\Pr[D_k(X_k) = 1] - \Pr[D_k(Y_k) = 1]|$$

is called the distinguishability-gap *of $\{D_k\}$ between the two ensembles.*

---

[21] We note that the fact that we restrict our attention to one-way access is instrumental in obtaining space-robust generators without making intractability assumptions. Analogous generators for two-way space-bounded computations would imply hardness results of a breakthrough nature in the area.

[22] We note that these non-uniform space-bounded distinguishers correspond to branching programs of width that is exponential in the space-bound. Furthermore, these branching programs read their input in a fixed predetermined order (which is determined by the designer of the generator).

[23] Note that the space bound of the machine is stated in terms of a parameter k, rather than in terms of the length of its input. In the sequel this parameter will be set to the length of a seed to a pseudorandom generator. We warn that our presentation here is indeed non-standard for this area. To compensate for this, we will also state the consequences in the standard format.

- *A probability ensemble,* $\{X_k\}_{k\in\mathbb{N}}$*, is called* (s,ε)-pseudorandom *if for any (non-uniform)* $s(\cdot)$*-space-bounded machine, the distinguishability-gap of the machine between* $\{X_k\}_{k\in\mathbb{N}}$ *and a uniform ensemble (of length* $|X_k|$*) is at most* $\varepsilon(\cdot)$*.*
- *A deterministic algorithm* G *of stretch function* ℓ *is called a* (s,ε)-pseudorandom generator *if the ensemble* $\{G(U_k)\}_{k\in\mathbb{N}}$ *is* (s,ε)*-pseudorandom.*

8.5.2 Two constructions

In contrast to the case of pseudorandom generators that fool time-bounded distinguishers, pseudorandom generators that fool space-bounded distinguishers can be established without relying on any computational assumption. The following two constructions exhibit two extreme cases of a general trade-off between the length of the seed and the stretch function of the generator.[24] We start with an attempt to maximize the stretch.

Theorem 8.21 (exponential stretch with quadratic length seed): *For every space constructible function* $s:\mathbb{N}\to\mathbb{N}$*, there exists a* $(s,2^{-s})$*-pseudorandom generator of stretch function* $\ell(k)=2^{k/O(s(k))}$*. Furthernore, the generator works in space linear in the length of the seed, and in time linear in the stretch function.*

In other words, we have a generator that takes a random seed of length $k=O(t\cdot m)$ and produce sequences of length 2^t that look random to any m-space-bounded machine. In particular, using a random seed of length $k=O(m^2)$, one can produce sequences of length 2^m that look random to any m-space bounded machine. Thus, *one may replace random sequences used by any space-bounded computation, by sequences that are efficiently generated from random seeds of length quadratic in the space bound*. The common instantiation is for log-space machines. In §8.5.2.2, we apply Theorem 8.21 (and its underlying ideas) for the derandomization of space complexity classes such as $\mathcal{BPL}$ (i.e., the log-space analogue of $\mathcal{BPP}$).

We now turn to the case where one wishes to minimize the seed length. We warn that Theorem 8.22 only guarantees a subexponential distinguishing gap (rather than the exponential distinguishing gap guaranteed in Theorem 8.21). This warning is voiced because failing to recall this limitation has led to errors in the past.

Theorem 8.22 (polynomial stretch with linear length seed): *For any polynomial* p *and for* $s(k)=k/O(1)$*, there exists a* $(s,2^{-\sqrt{s}})$*-pseudorandom generator of stretch function* p*. Furthermore, the generator works in linear-space and polynomial-time* (both stated in terms of the length of the seed).

In other words, we have a generator that takes a random seed of length $k=O(m)$ and produce sequences of length $\mathrm{poly}(m)$ that look random to any m-space-bounded machine. Thus, one may *convert any randomized computation utilizing*

[24] These two results have been "interpolated" in [11]: There exists a parameterized family of space pseudorandom generators that includes both results as extreme special cases.

polynomial-time and linear-space into a functionally equivalent randomized computation of similar time and space complexities that uses only a linear number of coin tosses.

8.5.2.1 Overviews of the proofs of Theorems 8.21 and 8.22

In both cases, we describe the construction by starting with an adequate distinguisher and showing how the input distribution it examines can be modified (from the uniform one into a pseudorandom one) without the distinguisher noticing the difference.

Overview of the proof of Theorem 8.21.[25] Theorem 8.21 is proven by using the "mixing property" of pairwise independent hash functions (see Section D.2). A family of functions H_n which map $\{0,1\}^n$ to itself is called *mixing* if for every pair of subsets $A, B \subseteq \{0,1\}^n$ for all but very few (i.e., $\exp(-\Omega(n))$ fraction) of the functions $h \in H_n$,

$$\mathsf{Pr}[U_n \in A \wedge h(U_n) \in B] \approx \frac{|A|}{2^n} \cdot \frac{|B|}{2^n} \tag{8.11}$$

where the approximation is up to an additive term of $\exp(-\Omega(n))$. (See the generalization of Lemma D.4.)

Given a $s(k)$-space distinguisher D_k as in Definition 8.20, we set $n \stackrel{\text{def}}{=} \Theta(s(k))$ and $\ell' \stackrel{\text{def}}{=} \ell(k)/n < 2^{s(k)}$, and consider an auxiliary "distinguisher" D'_k that is a directed layered graph with ℓ' layers and $2^{s(k)}$ vertices in each layer. In D'_k, each vertex has directed edges going to each vertex of the next layer and these edges are *labeled with* (possibly empty) *subsets of* $\{0,1\}^n$ such that these subsets form a partition of $\{0,1\}^n$. The graph D'_k simulates D_k in the obvious manner; that is, the computation of D'_k on an input of length $\ell(k) = \ell' \cdot n$ is defined by breaking the input into consecutive blocks of length n and following the path of edges that are labeled by the subsets containing the corresponding block. Now, for each pair of neighboring vertices, u and v (in layers i and $i+1$, respectively), consider the label, $L_{u,v} \subseteq \{0,1\}^n$, of the edge going from u to v. Similarly, for a vertex w at layer $i+2$, we consider the label $L'_{v,w}$ of the edge from v to w. By Eq. (8.11), for all but very few few of $h \in H_n$,

$$\mathsf{Pr}[U_n \in L_{u,v} \wedge h(U_n) \in L'_{v,w}] \approx \mathsf{Pr}[U_n \in L_{u,v}] \cdot \mathsf{Pr}[U_n \in L'_{v,w}]$$

where "very few" and $\approx$ are as in Eq. (8.11). Thus, replacing the coins in the second block (i.e., used in transitions from layer $i+1$ to layer $i+2$) with the value of h applied to the outcomes of the coins used in the first block (i.e., in transitions from layer i to $i+1$), approximately maintains the probability that D'_k moves from u to w via v. The same (with "few" being $2^{3s(k)} \cdot \ell'$ times larger here)[26] holds for every triple of vertices in any three layers as in the foregoing discussion. The point is that we can use the same h in all these approximations. Thus, at the cost

[25] A detailed proof appears in [161].

[26] Note that "very few" means an $\exp(-\Omega(n))$ fraction and that $n = \Omega(s(k))$ and $\ell' < \exp(s(k))$.

of extra $|h|$ random bits, we can reduce the number of true random coins used in transitions on D'_k by a factor of 2, without significantly affecting the final decision of D'_k. In other words, at the cost of extra $|h|$ random bits, we can effectively contract the distinguisher to half its length. That is, fixing a good h (i.e., one that provides a good approximation to all $2^{3s(k)} \cdot \ell'$ relevant pairs of sets), we can replace the 2-edge paths in D'_k by edges in a new distinguisher D''_k such that r is in the set that labels the edge u–w in D''_k if and only if, for some v, the string r is in the label of the edge u–v in D'_k and $h(r)$ is in the label of the edge v–w (also in D'_k).

Repeating the process for a logarithmic (in D'_k's length) number of times we obtain a distinguisher that only examines n bits, at which point we stop. In total, we have used $\log_2(\ell(k)/O(s(k))) < \log_2 \ell(k)$ random hash functions, which means that we can generate a sequence that fools the original D_k using a seed of length $n + \log_2 \ell(k) \cdot \log_2 |H_n|$ (see Exercise 8.19). Using $n = \Theta(s(k))$ and an adequate family H_n (e.g., Construction D.3) yields the claimed seed length of $O(s(k) \cdot \log_2 \ell(k)) = k$. □

Overview of the proof of Theorem 8.22.[27] Theorem 8.22 is proven by using a suitable randomness extractor (as defined in §D.4.1.1), which is indeed a much more powerful tool than hashing functions. The basic idea is that when D_k is at some distant layer, say at layer t, it typically "knows" little about the random choices that led it there. That is, D_k has only $s(k)$ bits of memory, which leaves out $t - s(k)$ bits of "uncertainty" (or randomness) regarding the previous moves. Thus, much of the randomness that led D_k to its current state may be "re-used" (or "recycled"). To re-use these bits we need to extract *almost* uniform distribution on strings of sufficient length out of the aforementioned distribution over $\{0,1\}^t$ that has entropy[28] at least $t - s(k)$. Furthermore, such an extraction requires some – yet relatively few – truly random bits. In particular, using $k' = \Omega(\log t)$ bits towards this end, the extracted bits are $\exp(-\Omega(k'))$ away from uniform.

One important point is how to use the foregoing argument repeatedly. Towards this end, we break the k-bit long seed into two parts, denoted $r' \in \{0,1\}^{k/2}$ and $(r_1, ..., r_{3\sqrt{k}})$, where $|r_i| = \sqrt{k}/6$, and set $n = k/3$. Intuitively, r' will be used for determining the first n steps, and it will be re-used (or recycled) together with r_i for determining the steps $i \cdot n + 1$ through $(i+1) \cdot n$. Looking at layer $i \cdot n$, we consider the information regarding r' that is known to D_k (at layer $i \cdot n$). Typically, the conditional distribution of r', given that we reached a specific vertex at layer $i \cdot n$ has (min-)entropy greater than $0.99 \cdot (t - s(k))$. Using r_i (as a seed of an extractor applied to r'), we can extract $0.9 \cdot ((k/2) - s(k) - o(k)) > k/3 = n$ bits that are almost-random with respect to D_k, and use these bits for determining the next n steps. Hence, using k random bits we are produce a sequence[29] of length

[27] A detailed proof appears in [164].

[28] Actually, a stronger technical condition needs and can be imposed on the latter distribution. Specifically, with overwhlemingly high probability, at layer t machine D_k is at a vertex that can be reached in more than $2^{0.99 \cdot (t - s(k))}$ different ways. In this case, the distribution representing a random walk that reaches this vertex has min-entropy greater than $0.99 \cdot (t - s(k))$. The reader is referred to §D.4.1.1 for definitions of min-entropy and extractors.

[29] Specifically, using an extractor of the form $\mathrm{Ext} : \{0,1\}^{\sqrt{k}/6} \times \{0,1\}^{k/2} \to \{0,1\}^{k/3}$, we map

$(1+3\sqrt{k})\cdot n > k^{3/2}$ that fools machines of space bound, say, $s(k)=k/10$. That is, we obtained a $(s,2^{-\Omega(\sqrt{s})})$-pseudorandom generator of stretch function $\ell(k)=k^{3/2}$.

To obtain an arbitrary polynomial stretch rather than a specific polynomial stretch (i.e., $\ell(k)=k^{3/2}$) we repeatedly apply an adequate composition, to be outlined next. Suppose that G_1 is a (s_1,ε_1)-pseudorandom generator of stretch function ℓ_1 that works in linear space, and similarly for G_2 with respect to (s_1,ε_1) and ℓ_2. Then, we consider the following construction of a generator G:

1. On input $s\in\{0,1\}^k$, obtain $G_1(s)$, and parse it into consecuetive blocks, each of length $m=s_1(k)/O(1)$, denoted $r_1,...,r_t$, where $t=\ell_1(k)/m$.

2. Output the $t\cdot\ell_2(m)$-bit long sequence $G_2(r_1)\cdots G_2(r_t)$.

Note that $|G(s)|=\ell_1(k)\cdot\ell_2(m)/m$, which for $s_1(k)=\Theta(k)$ yields $|G(s)|=\ell_1(k)\cdot\ell_2(\Omega(k))/O(k)$. We claim that G is a (s,ε)-pseudorandom generator, for $s(k)=\min(s_1(k)/2,s_2(\Omega(s_1(k)))$ and $\varepsilon(k)=\varepsilon_1(k)+\ell_1(k)\cdot\varepsilon_2(\Omega(s_1(k))$. The proof uses a hybrid argument, which focuses on the intermediate distribution $G_2(U_m^{(1)})\cdots G_2(U_m^{(t)})$. The key claim is that the intermediate distribution is $(s_1/2,\varepsilon_1)$-indistinguishable from $G(U_k)$, and it is proven by converting a potential distinguisher into a distinguisher of $U_m^{(1)}\cdots U_m^{(t)}$ and $G_1(U_k)$ by invoking G_2 on the corresponding m-bit long blocks (of the $\ell(k)$-bit long input). For this reason, it crucial that G_2 can be evaluate on m-bit long strings using space at most $s_1(k)/2$, which is guaranteed by our setting of $m=s_1(k)/O(1)$ and the hypothesis that G_2 works in linear space. □

8.5.2.2 Derandomization of space-complexity classes

As a direct application of Theorem 8.21, we obtain that $\mathcal{BPL}\subseteq$ DSPACE($\log^2$), where $\mathcal{BPL}$ denotes the log-space analogue of $\mathcal{BPP}$ (see Definition 6.9). (Recall that $\mathcal{NL}\subseteq$ DSPACE($\log^2$), but it is not known whether or not $\mathcal{BPL}\subseteq\mathcal{NL}$.)[30] A stronger derandomization result can be obtained by a finer analysis of the proof of Theorem 8.21.

Theorem 8.23 *$\mathcal{BPL}\subseteq\mathcal{SC}$, where $\mathcal{SC}$ denotes the class of decision problems that can be solved by a deterministic machine that runs in polynomial-time and polylogarithmic-space.*

Thus, $\mathcal{BPL}\supseteq\mathcal{RL}$ is placed in a class not known to contain $\mathcal{NL}$. Another such result was subsequently obtained in [183]: Randomized log-space can be simulated in deterministic space $o(\log^2)$; specifically, in space $\log^{3/2}$. We mention that the archetypical problem of $\mathcal{RL}$ has been recently proved to be in $\mathcal{L}$ (see Section 5.2).

Overview of the proof of Theorem 8.23.[31] Looking at the proof of Theorem 8.21, we note that the question of whether or not a specific hash function

the seed $(r',r_1,...,r_{3\sqrt{k}})$ to the output sequence $(r',\mathrm{Ext}(r_1,r'),...,\mathrm{Ext}(r_{3\sqrt{k}},r'))$.

[30] Indeed, the log-space analogue of $\mathcal{RP}$, denoted $\mathcal{RL}$, is contained in $\mathcal{NL}\subseteq$ DSPACE($\log^2$), and thus the fact that Theorem 8.21 implies $\mathcal{RL}\subseteq$ DSPACE($\log^2$) is of no interest.

[31] A detailed proof appears in [162].

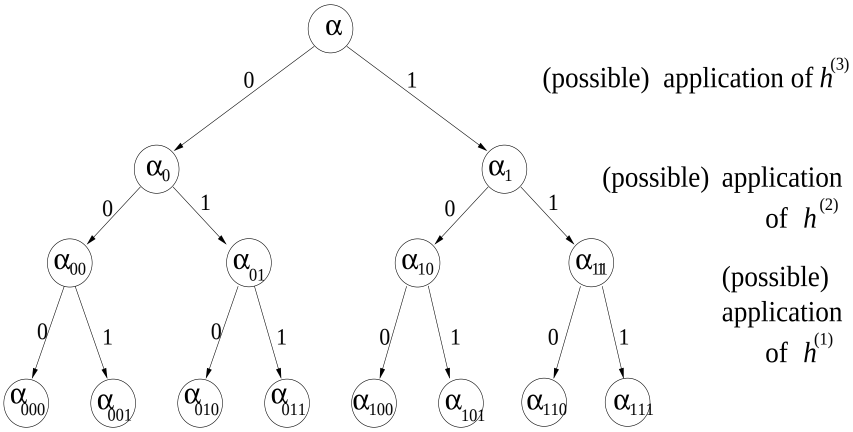
$h \in H_n$ is good for a specific D'_k can be determined in space that is linear in $n = |h|/2$ and logarithmic in the size of D'_k. Indeed, the time complexity of this decision procedure is exponential in its space complexity. It follows that we can find a good $h \in H_n$, for a given D'_k, within these complexities (by scanning through all possible $h \in H_n$). Once a good h is found, we can also construct the corresponding graph D''_k (in which edges represent 2-edge paths in D'_k), again within the same complexity. Actually, it will be more instructive to note that we can determine a step (i.e., an edge-traversal) in D''_k by making two steps (edge-traversals) in D'_k.

The key claim is that the entire process of finding a sequence of $t \stackrel{\text{def}}{=} \log_2 \ell'(k)$ good hash functions can be performed in space $t \cdot O(n + \log|D_k|) < O(n + \log|D_k|)^2$ and time $\mathrm{poly}(2^n \cdot |D_k|)$; that is, the time complexity is sub-exponential in the space complexity (i.e., the time complexity is significantly smaller than than the generic bound of $\exp(O(n + \log|D_k|)^2))$. Starting with $D_k^{(1)} = D'_k$, we find a good (for $D_k^{(1)}$) hashing function $h^{(1)} \in H_n$, which defines $D_k^{(2)} = D''_k$. Having found (and stored) $h^{(1)}, ..., h^{(i)} \in H_n$, which determine $D_k^{(i+1)}$, we find a good hashing function $h^{(i+1)} \in H_n$ for $D_k^{(i+1)}$ by emulating pairs of edge-traversals on $D_k^{(i+1)}$. Indeed, a key point is that we do *not* construct the sequence of graphs $D_k^{(2)}, ..., D_k^{(i+1)}$, but rather emulate an edge-traversal in $D_k^{(i+1)}$ by making 2^i edge-traversals in D'_k, using $h^{(1)}, ..., h^{(i)}$: The (edge-traversal) move $\alpha \in \{0,1\}^n$ starting at vertex v of $D_k^{(i+1)}$ translates to a sequence of 2^i moves starting at vertex v of D'_k, where the moves are determined by

$$\alpha, h^{(1)}(\alpha), h^{(2)}(\alpha), h^{(1)}(h^{(2)}(\alpha)), ..., h^{(1)}(h^{(2)}(\cdots h^{(i)}(\alpha)\cdots)).$$

(See Figure 8.3.) Thus, for $n = \Theta(\log|D'_k|)$, given D'_k and a pair (u, v) of source and sink in D'_k (which reside in the first and last layer, respectively), we can (deterministically) approximate the probability that a random walk starting at u reaches v in $O(\log|D'_k|)^2$-space and $\mathrm{poly}(|D'_k|)$-time. The approximation can be made accurate up to a factor of $1 \pm (1/\mathrm{poly}(|D'_k|))$. We conclude the proof by recalling the connection between such an approximation and the derandomization of $\mathcal{BPL}$.

The computation of a log-space probabilistic machine M, on input x, can be represented by a directed layer graph $G_{M,x}$ of size $\mathrm{poly}(|x|)$. Specifically, the probability that M accepts x equals the probability that a random walk starting at the single vertex of the first layer of $G_{M,x}$ reaches some vertex in the last layer that represents an accepting configuration. Setting $k = \Theta(\log|x|)$ and $n = \Theta(k)$, the graph $G_{M,x}$ coincides with the graph D_k referred to at the beginning of the proof of Theorem 8.21, and D'_k is obtained from D_k by an "n-layer contraction" (see ibid.). Combining this with the foregoing analysis, we conclude that the probability that M accepts x can be deterministically approximated in $O(\log|x|)^2$-space and $\mathrm{poly}(|x|)$-time. The theorem follows. $\square$



The output of the generator (on seed α) consists of the concatanation of the strings denoted $\alpha_{0^i},...,\alpha_{1^i}$, appearing in the leaves of the tree. For every $x \in \{0,1\}^*$ it holds that $\alpha_{x0} = \alpha_x$ and $\alpha_{x1} = h^{(i-|x|)}(\alpha_x)$. In particular, for $i=3$, we have $\alpha_{011} = h^{(1)}(\alpha_{01})$, which equals $h^{(1)}(\alpha_{01}) = h^{(1)}(h^{(2)}(\alpha))$, where $\alpha = \alpha_\lambda$.

Figure 8.3: Derandomization of $\mathcal{BPL}$ – the generator for $i=3$.

8.6 Special Purpose Generators

In this section we consider even weaker types of pseudorandom generators, producing sequences that can fool only very restricted types of distinguishers. Still, such generators have many applications in complexity theory and in the design of algorithms. (These applications will only be mentioned briefly.)

Our choice is to start with the simplest of these generators: the pairwise-independent generator, and its generalization to t-wise independence for any $t \geq 2$. Such generators perfectly fool any distinguisher that only observe t locations in the output sequence. This leads naturally to almost pairwise (or t-wise) independence generators, which also fool (albeit non-perfectly) such distinguishers. The latter generators are implied by a stronger class of generators, which is of independent interest: the small-bias generators. Small-bias generators fool any linear test (i.e., any distinguisher that merely considers the XOR of some fixed locations in the input sequence). We then turn to the Expander Random Walk Generator: this generator produces a sequence of strings that hit any dense subset of strings with probability that is close to the hitting probability of a truly random sequence. Related notions such as samplers, dispersers, and extractors are treated in Appendix D.

Comment regarding our parameterization: To maintain consistency with prior sections, we continue to present the generators in terms of the seed length, denoted k. Since this is not the common presentation for most results presented in the sequel, we provide (in footnotes) the common presentation in which the seed length is determined as a function of other parameters.

8.6.1 Pairwise-Independence Generators

Pairwise (resp., t-wise) independence generators fool tests that inspect only two (resp., t) elements in the output sequence of the generator. Such loacl tests are indeed very restricted, yet they arise naturally in many settings. For example, such a test corresponds to a probabilistic analysis (of a procedure) that only relies on the pairwise independence of certain choices made by the procedure. We also mention that, in some natural range of parameters, pairwise independent sampling is as good as sampling by totally independent sample points; see Sections D.1.2 and D.3.

A t-wise independence generator of block-size $b : \mathbb{N} \rightarrow \mathbb{N}$ (and stretch function ℓ) is an efficient deterministic algorithm (e.g., one that works in time polynomial in the output length) that expands a k-bit long random seed into a sequence of $\ell(k)/b(k)$ strings, each of length $b(k)$, such that any t blocks are uniformly and independently distributed in $\{0,1\}^{t \cdot b(k)}$. In case $t = 2$, we call the generator pairwise independent. We note that this condition holds even if the inspected t blocks are selected adaptively (see Exercise 8.20)

8.6.1.1 Constructions

In the first construction, we refer to $\mathrm{GF}(2^{b(k)})$, the finite field of $2^{b(k)}$ elements, and associate its elements with $\{0,1\}^{b(k)}$.

Proposition 8.24 (t-wise independence generator):[32] *Let t be a fixed integer and $b, \ell, \ell' : \mathbb{N} \rightarrow \mathbb{N}$ such that $b(k) = k/t$, $\ell'(k) = \ell(k)/b(k) > t$ and $\ell'(k) \leq 2^{b(k)}$. Let $\alpha_1, ..., \alpha_{\ell'(k)}$ be distinct elements of this field. For $s_0, s_1, ..., s_{t-1} \in \{0,1\}^{b(k)}$, let*

$$G(s_0, s_1, ..., s_{t-1}) \stackrel{\text{def}}{=} \left(\sum_{j=0}^{t-1} s_j \alpha_1^j \, , \, \sum_{j=0}^{t-1} s_j \alpha_2^j \, , ..., \, \sum_{j=0}^{t-1} s_j \alpha_{\ell'(k)}^j \right) \tag{8.12}$$

where the arithmetic is that of $\mathrm{GF}(2^{b(k)})$. Then, G is a t-wise independence generator of block-size b and stretch ℓ.

That is, given a seed that consists of t elements of $\mathrm{GF}(2^{b(k)})$, the generator outputs a sequence of $\ell'(k)$ such elements. To make the above generator totally explicit, we need an explicit representation of $\mathrm{GF}(2^{b(k)})$, which requires an irreducible polynomial of degree $b(k)$ over $\mathrm{GF}(2)$. For specific values of $b(k)$, a good representation does exist: Specifically, for $d \stackrel{\text{def}}{=} b(k) = 2 \cdot 3^e$ (with e being an integer), the polynomial $x^d + x^{d/2} + 1$ is irreducible over $\mathrm{GF}(2)$. The proof of Proposition 8.24 is left as an exercise (see Exercise 8.21).

[32] The common parameterization of t-wise independence generator is as follows. Given parameters b and $\ell' \leq 2^b$, and a uniformly distributed seed of length $t \cdot b$, one efficiently and deterministically generates a random sequence of ℓ' strings, each of length b, that are t-wise independent.

An alternative construction for the case of $t = 2$ is obtained by using (random) affine transformations (as possible seeds). In fact, better performance (i.e., shorter seed length) is obtained by using affine transformations defined by Toeplitz matrices. A Toeplitz matrix is a matrix with all diagonals being homogeneous; that is, $T = (t_{i,j})$ is a Toeplitz matrix if $t_{i,j} = t_{i+1,j+1}$, for all i,j. Note that a Toeplitz matrix is determined by its first row and first column (i.e., the values of $t_{1,j}$'s and $t_{i,1}$'s).

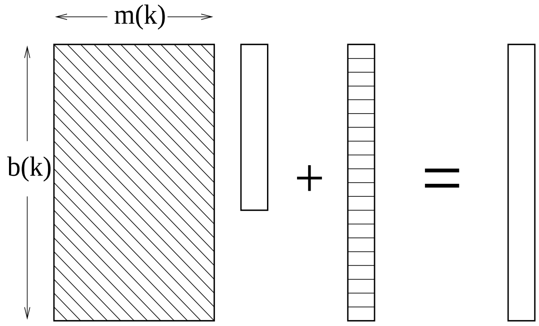


Figure 8.4: An affine transformation defined by a Toeplitz matrix.

Proposition 8.25 (Alternative pairwise independence generator, see Figure 8.4):[33] *Let $b, \ell, \ell', m : \mathbb{N} \to \mathbb{N}$ such that $\ell'(k) = \ell(k)/b(k)$ and $m(k) = \lceil \log_2 \ell'(k) \rceil = k - 2b(k) + 1$. Associate $\{0,1\}^n$ with the n-dimensional vector space over* GF(2), *and let $v_1, ..., v_{\ell'(k)}$ be distinct vectors in the $m(k)$-dimensional vector space over* GF(2). *For $s \in \{0,1\}^{b(k)+m(k)-1}$ and $r \in \{0,1\}^{b(k)}$, let*

$$G(s,r) \stackrel{\text{def}}{=} (T_s v_1 + r \, , \, T_s v_2 + r \, , ..., \, T_s v_{\ell'(k)} + r) \tag{8.13}$$

where T_s is an $b(k)$-by-$m(k)$ Toeplitz matrix specified by the string s. Then G is a pairwise independence generator of block-size b and stretch ℓ.

That is, given a seed that represents an affine transformation defined by an $b(k)$-by-$m(k)$ Toeplitz matrix, the generator outputs a sequence of $\ell'(k) \leq 2^{m(k)}$ strings, each of length $b(k)$. Note that $m(k) = k - 2b(k) + 1$, and that the stretching property requires $\ell'(k) > k/b(k)$. The proof of Proposition 8.25 is left as an exercise (see Exercise 8.22).

A stronger notion of efficient generator. We note that the aforementioned constructions satisfy a stronger notion of efficient generation, which is useful in several applications. Specifically, there exists a polynomial-time algorithm that given a seed, $s \in \{0,1\}^k$, and a block location $i \in [\ell'(k)]$ (in bianry), outputs the i^{th} block of the corresponding output (i.e., the i^{th} block of $G(s)$).

[33] The common parameterization of this pairwise independence generator is as follows. Given parameters b and ℓ', and a uniformly chosen seed of length $2b + \lceil \log_2 \ell' \rceil - 1$, one efficiently and deterministically generates a random sequence of ℓ' strings, each of length b, that are pairwise independent.

8.6.1.2 Applications

Pairwise independence generators do suffice for a variety of applications (cf., [220, 149]). In particular, we mention the application to sampling discussed in Section D.3, and the celebrated derandomization of the Maximal Independent Set algorithm. The latter uses the fact that the analysis of the "target" randomized algorithm only relies on the hypothesis that some objects are selected in pairwise independent manner. Thus, such weak generators do suffice to fool distinguishers that are derived from some natural and interesting algorithms.

Referring to Eq. (8.12), we remark that for constant $t \geq 2$, the cost of derandomization (i.e., going over all 2^k possible seeds) is exponential in the block-size (because $b(k) = \Omega(k)$), which in turn also bounds the number of blocks (because $\ell'(k) \leq 2^{b(k)}$). Note that if a larger number of blocks is needed, we can artificially increase the block-length in order to allow for it (i.e., allow $\ell'(k) = 2^{b(k)} = \exp(k/t)$), and in this case the cost of derandomization will be polynomial in the number of blocks. Thus, *whenever the analysis of a randomized algorithm can be based on a constant amount of independence* between (feasibly-many) random choices, each made inside a feasible domain, *a feasible derandomization is possible.*[34] On the other hand, the relationship $\ell(k) = \exp(k/t)$ is the best possible; that is, one cannot produce from a seed of length k an $\exp(k/O(1))$-long sequence of non-constantly independent random bits. In other words, t-wise independent generators of (any block-length and) stretch ℓ require a seed of length $\Omega(t \cdot \log \ell)$. In the next subsection we will see that meaningful approximations may be obtained with much shorter seeds.

8.6.2 Small-Bias Generators

Trying to go beyond constant-independence in derandomizations (while using seeds of length that is logarithmic in the length of the pseudorandom sequence) was the original motivation (and remain an important application) of the notion of small-bias generators. Still, small-bias generators are interesting for their own sake, and in particular they fool "global tests" that look at the entire output sequence and not merely at a fixed number of positions in it (as the limited independence generators). Specifically, small-bias generators generate a sequence of bits that fools any linear test (i.e., a test that computes a fixed linear combination of the bits).

For $\varepsilon : \mathbb{N} \to [0,1]$, an ε-bias generator with stretch function ℓ is an efficient deterministic algorithm (e.g., working in $\text{poly}(\ell(k))$ time) that expands a k-bit long random seed into a sequence of $\ell(k)$ bits such that for any fixed non-empty set $S \subseteq \{1, ..., \ell(k)\}$ the bias of the output sequence over S is at most $\varepsilon(k)$. The bias of a sequence of n (possibly dependent) Boolean random variables $\zeta_1, ..., \zeta_n \in \{0,1\}$ over a set $S \subseteq \{1, .., n\}$ is defined as

$$2 \cdot \left| \Pr[\oplus_{i \in S} \zeta_i = 1] - \frac{1}{2} \right| = |\Pr[\oplus_{i \in S} \zeta_i = 1] - \Pr[\oplus_{i \in S} \zeta_i = 0]|. \quad (8.14)$$

[34] We stress that it is important to have the cost of derandomization be polynomial in the length of the produced pseudorandom sequence, because the latter is typically polynomially-related to the length of the input to the algorithm we wish to derandomize.

The factor of 2 was introduced so to make these biases correspond to the Fourier coefficients of the distribution (viewed as a function from $\{0,1\}^n$ to the reals). To see the correspondence replace $\{0,1\}$ by $\{\pm 1\}$, and substitute XOR by multiplication. The bias with respect to set S is thus written as

$$\left|\mathsf{Pr}\left[\prod_{i\in S}\zeta_i=+1\right]-\mathsf{Pr}\left[\prod_{i\in S}\zeta_i=-1\right]\right| \;=\; \left|\mathsf{E}\left[\prod_{i\in S}\zeta_i\right]\right|,$$

which is merely the (absolute value of the) Fourier coefficient corresponding to S.

8.6.2.1 Constructions

Efficient small-bias generators with exponential stretch and exponentially vanishing bias are know.

Theorem 8.26 (small-bias generators):[35] *For some universal constant $c>0$, let $\ell:\mathbb{N}\to\mathbb{N}$ and $\varepsilon:\mathbb{N}\to[0,1]$ such that $\ell(k)\le\varepsilon(k)\cdot\exp(k/c)$. Then, there exists an ε-bias generator with stretch function ℓ operating in time polynomial in the length of its output.*

Three simple constructions of small-bias generators that satisfy Theorem 8.26 are known (see [9]). One of these constructions is based on Linear Feedback Shift Registers. Loosely speaking, the first half of the seed, denoted $f_0f_1\cdots f_{(k/2)-1}$, is interpreted as a (non-degenerate) feedback rule[36], the other half, denoted $s_0s_1\cdots s_{(k/2)-1}$, is interpreted as "the start sequence", and the output sequence, denoted $r_0r_1\cdots r_{\ell(k)-1}$, is obtained by setting $r_i=s_i$ for $i<k/2$ and $r_i=\sum_{j=0}^{(k/2)-1}f_j\cdot r_{i-(k/2)+j}$ for $i\ge k/2$. (See Figure 8.5 and Exercise 8.26.)

As in Section 8.6.1.1, we note that the aforementioned constructions satisfy a stronger notion of efficient generation, which is useful in several applications. Specifically, there exists a polynomial-time algorithm that given a seed and a bit location $i\in[\ell(k)]$ (in bianry), outputs the $i^{\rm th}$ bit of the corresponding output.

8.6.2.2 Applications

An archetypical application of small-bias generators is for generating random checks for fast string equality. The key obserevation is that checking whether or not $x=y$ is probabilistically reducible to checking whether the inner product modulo 2 of x and r equals the inner product modulo 2 of y and r, where r is generated by a small-bias generator. One advantage of this reduction is that only few bits (i.e., the seed of the generator and the result of the inner product) needs to be "communicated

[35]Here the common parameterization differs from ours merely in the point of view: Rather than saying that the functions ℓ and ε should satisfy $\ell(k)\le\varepsilon(k)\cdot\exp(k/c)$, one says that given the desired parameters ℓ and ε the seed length k is set to $O(\log(\ell/\varepsilon))$. We also comment that using [9] the constant in the O-notation is merely 2 (i.e., $k\approx 2\log_2(\ell/\varepsilon)$), whereas using [158] $k\approx\log_2\ell+4\log_2(1/\varepsilon)$.

[36]That is, $f_0=1$ and $f(z)\stackrel{\rm def}{=}z^{k/2}+\sum_{j=0}^{(k/2)-1}f_j\cdot z^j$ is an irreducible polynomial over GF(2).

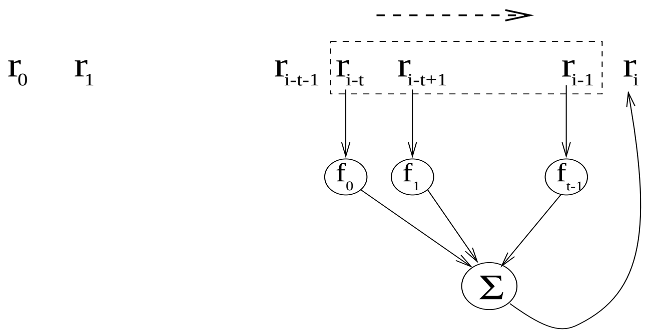


Figure 8.5: The LFSR small-bias generator (for $t = k/2$).

between x and y" (see Exercise 8.24). A related advantage (i.e., low randomness complexity) underlies the application of small-bias generators in §9.3.2.2.

Small-bias generators have been used in a variety of areas (e.g., inapproximation, structural complexity, and applied cryptography; see references in [86, Sec 3.6.2]). In addition, they seem an important tool in the design of various types of "pseudorandom" objects; see next.

Approximate independence generators. As hinted at the beginning of this section, small-bias is related to approximate limited independence.[37] Actually, even a restricted type of ε-bias (in which only subsets of size $t(k)$ are required to have bias upper-bounded by ε) implies that any $t(k)$ bits in the said sequence are $2^{t(k)/2} \cdot \varepsilon(k)$-close to $U_{t(k)}$, where here we refer to the variation distance (i.e., Norm-1 distance) between the two distributions. (The max-norm of the difference is bounded by $\varepsilon(k)$.)[38] Combining Theorem 8.26 and the foregoing upper-bound, and relying on the linearity of the construction in Proposition 8.24, we obtain generators with $\exp(k)$ stretch that are approximately $t(k)$-independent, for some non-constant $t(k)$; see Exercise 8.29. Specifically, for $k = O(t(k) + \log(1/\varepsilon(k)) + \log\log \ell(k))$ (equiv., for $\ell(k) = 2^{2^{k/O(1)}}$, $t(k) = k/O(1)$, and $\varepsilon(k) = 2^{-k/O(1)}$), one may obtain generators with stretch function ℓ, producing bit sequences in which any $t(k)$ positions are at most $\varepsilon(k)$-away from uniform (in variation distance). In the corresponding result for the max-norm distance, it suffices to have $k = O(\log(t(k)/\varepsilon(k) + \log\log \ell(k))$. Thus, whenever the analysis of a randomized algorithm can be based on a logarithmic amount of (almost) independence between feasibly-many binary random choices, a feasible derandomization is possible (by using an adequate generator of logarithmic seed length).

Extensions to non-binary choices were considered in various works (see references in [86, Sec 3.6.2]). Some of these works also consider the related problem of constructing small "discrepancy sets" for geometric and combinatorial rectangles.

[37] We warn that, unlike in the case of perfect independence, here we refer only to the distribution on fixed bit locations. See Exercise 8.23 for further discussion.

[38] Both bounds are derived from the Norm2 bound on the difference vector (i.e., the difference between the two probability vectors). For details, see Exercise 8.25.

t-universal set generators. Using the aforementioned upper-bound on the max-norm, for $\varepsilon < 2^{-t}$, any ε-bias generator yields a t-universal set generator. The latter generator outputs sequences such that in every subsequence of length t all possible 2^t patterns occur (i.e., each for at least one possible seed). Such generators have many applications.

8.6.3 Random Walks on Expanders

In this section we review generators that produce a sequence of values by taking a random walk on a large graph (called an expander) having small degree but good "mixing" properties. Thus, producing a sequence of length ℓ over 2^b values, requires a random seed of length $b + (\ell - 1) \cdot \log_2 d$, where d is the degree of the said graph (of 2^b vertices). This should be compared against the randomness needed for generating a sequence of ℓ independent samples from $\{0,1\}^b$ (or taking a random walk on a clique of size 2^b). It will turn out that the pseudorandom sequence (generated by the said random walk on an expander) *behaves analogously to a truly random sequence with respect to hitting any fixed subset of* $\{0,1\}^b$. Let us start by defining this property, or rather define the hitting problem.

Definition 8.27 (the hitting problem): *A distribution on sequences over* $\{0,1\}^b$ *is* (ε, δ)-hitting *if for any* (target) *set* $T \subseteq \{0,1\}^b$ *of cardinality at least* $\varepsilon \cdot 2^b$*, with probability at least* $1 - \delta$*, at least one of the elements of a sequence drawn from this distribution hits* T.

Clearly, a truly random sequence of length ℓ over $\{0,1\}^b$ is (ε, δ)-hitting for $\delta = (1-\varepsilon)^\ell$. The aforementioned "expander random walk generator" (to be described next) achieves similar behavior. Specifically, for arbitrary small $c > 0$ (which depends on the degree and the mixing property of the expander), the generator's output is (ε, δ)-hitting for $\delta = (1 - (1-c) \cdot \varepsilon)^\ell$. To describe this generator, we need to discuss expanders.

Expanders. By expander graphs (or expanders) of degree d and eigenvalue bound $\lambda < d$, we actually mean an infinite family of d-regular graphs, $\{G_N\}_{N \in S}$ ($S \subseteq \mathbb{N}$), such that G_N is a d-regular graph over N vertices and the absolute value of all eigenvalues, save the biggest one, of the adjacency matrix of G_N is upper-bounded by λ. We will refer to such a family as to a (d, λ)-expander (for S). This technical definition is related to the aforementioned notion of "mixing" (which refers to the rate at which a random walk starting at a fixed vertex reaches uniform distribution over the graph's vertices). For further detail, see Section E.2.

We are interested in explicit constructions of such graphs, by which we mean that there exists a polynomial-time algorithm that on input N (in binary), a vertex $v \in G_N$ and an index $i \in \{1, ..., d\}$, returns the i^{th} neighbor of v. (We also require that the set S for which G_N's exist is sufficiently "tractable" – say that given any $n \in \mathbb{N}$ one may efficiently find an $s \in S$ such that $n \le s < 2n$.) Several explicit constructions of expanders are known (see Section E.2.2). Below, we rely on the fact that for every $\varepsilon > 0$, there exist d and $\lambda < \varepsilon \cdot d$ such that there exists an explicit construction of a (d, λ)-expander over $\{2^b : b \in \mathbb{N}\}$.[39] The relevant (to us) fact about expanders is stated next.

Theorem 8.28 (Expander Random Walk Theorem): *Let $G = (V, E)$ be an expander graph of degree d and eigenvalue bound λ. Let W be a subset of V and $\rho \stackrel{\text{def}}{=} |W|/|V|$, and consider walks on G that start from a uniformly chosen vertex and take $\ell - 1$ additional random steps, where in each such step one uniformly selects one out of the d edges incident at the current vertex and traverses it. Then the probability that such a random walk stays in W is at most*

$$\rho \cdot \left(\rho + (1-\rho) \cdot \frac{\lambda}{d}\right)^{\ell-1} \tag{8.15}$$

Thus, a random walk on an expander is "pseudorandom" with respect to the hitting property (i.e., when we consider hitting the set $V \setminus W$); that is, a set of density $\varepsilon = 1 - \rho$ is hit with probability $1 - \delta$, where $\delta = (1-\varepsilon) \cdot (1 - \varepsilon + (\lambda/d) \cdot \varepsilon)^{\ell-1} < (1 - (1 - (\lambda/d)) \cdot \varepsilon)^{\ell}$. A proof of an upper-bound that is weaker than Eq. (8.15) is outlined in Exercise 8.30. Using Theorem 8.28 and an explicit $(2^t, \overline{\lambda} \cdot 2^t)$-expander, we get

Proposition 8.29 (The Expander Random Walk Generator):[40]

- *For every constant $\overline{\lambda} > 0$, consider an explicit construction of $(2^t, \overline{\lambda} \cdot 2^t)$-expanders for $\{2^n : n \in \mathbb{N}\}$, where $t \in \mathbb{N}$ is a sufficiently latge constant. For $v \in [2^n] \equiv \{0,1\}^n$ and $i \in [2^t] \equiv \{0,1\}^t$, denote by $\Gamma_i(v)$ the vertex of the corresponding 2^n-vertex graph that is reached from vertex v when following its i^{th} edge.*
- *For $b, \ell' : \mathbb{N} \to \mathbb{N}$ such that $k = b(k) + (\ell'(k) - 1) \cdot t < \ell'(k) \cdot b(k)$, and for $v_0 \in \{0,1\}^{b(k)}$ and $i_1, ..., i_{\ell'(k)-1} \in [2^t]$, let*

$$G(v_0, i_1, ..., i_{\ell'(k)-1}) \stackrel{\text{def}}{=} (v_0, v_1, ..., v_{\ell'(k)-1}), \tag{8.16}$$

  where $v_j = \Gamma_{i_j}(v_{j-1})$.

Then G has stretch $\ell(k) = \ell'(k) \cdot b(k)$, and $G(U_k)$ is (ε, δ)-hitting for any $\varepsilon > 0$ and $\delta = (1 - (1 - \overline{\lambda}) \cdot \varepsilon)^{\ell'(k)}$.

The stretch of G is optimized at $b(k) \approx k/2$ (and $\ell'(k) = k/2t$), but optimizing the stretch is not necessaily the goal in all applications. Expander random-walk generators have been used in a variety of areas (e.g., PCP and inapproximability (see [27, Sec. 11.1]), cryptography (see [87, Sec. 2.6]), and the design of various types of "pseudorandom" objects (see, in particular, Section D.3)).

[39] This can be obtained with $d = \text{poly}(1/\varepsilon)$. In fact $d = O(1/\varepsilon^2)$, which is optimal, can be obtained too, albeit with graphs of sizes that are only approximately close to powers of two.

[40] The common parameterization starts with parameters b and ℓ'. Given a uniformly chosen seed of length $b + O(\ell' - 1)$, one can efficiently and deterministically generate a random sequence of ℓ' strings, each of length b, which is (ε, δ)-hitting for any $\varepsilon > 0$ and $\delta = (1 - \Omega(\varepsilon))^{\ell'}$.

TYPE	distinguisher's resources	generator's resources	stretch (i.e., $\ell(k)$)	comments
gen.-purpose	$p(k)$-time, $\forall$ poly. p	poly(k)-time	poly(k)	Assumes OW[41]
derand. BPP	$2^{k/O(1)}$-time	$2^{O(k)}$-time	$2^{k/O(1)}$	Assumes EvEC[41]
space-bounded	$s(k)$-space	$O(k)$-space	$2^{k/O(s(k))}$	runs in time
robustness	$k/O(1)$-space	$O(k)$-space	poly(k)	poly$(k) \cdot \ell(k)$
t-wise indepen.	"t-wise"	poly$(k) \cdot \ell(k)$-time	$2^{k/O(t)}$	(e.g., pairwise)
small bias	"ε-bias"	poly$(k) \cdot \ell(k)$-time	$2^{k/O(1)} \cdot \varepsilon(k)$	
expander	"hitting"	poly$(k) \cdot \ell(k)$-time	$\ell'(k) \cdot b(k)$	
rand. walk	$(0.5, 2^{-\ell'(k)/O(1)})$-hitting for $\{0,1\}^{b(k)}$, with $\ell'(k) = ((k - b(k))/O(1)) + 1$.			

Figure 8.6: Pseudorandom generators at a glance

Chapter Notes

Figure 8.6 depicts some of the notions of pseudorandom generators discussed in this chapter. We highlight a key distinction between the case of general-purpose pseudorandom generators (treated in Section 8.3) and the other cases (cf. Sections 8.4 and 8.5): in the former case the distinguisher is more complex than the generator, whereas in the latter cases the generator is more complex than the distinguisher. Specifically, in the general-purpose case the generator runs in (some *fixed*) polynomial-time and needs to withstand *any* probabilistic polynomial-time distinguisher. In fact, some of the proofs presented in Section 8.3 utilize the fact that the distinguisher can invoke the generator on seeds of its choice. In contrast, the Nisan-Wigderson Generator, analyzed in Theorem 8.18 (of Section 8.4), runs more time than the distinguishers that it tries to foil, and the proof relies on this fact in an essential manner. Similarly, the space complexity of the space-resilient generators presented in Section 8.5 is higher than the space-bound on the distinguishers that they foil.

The general paradigm of pseudorandom generators. Our presentation, which views vastly different notions of pseudorandom generators as incarnations of a general paradigm, has emerged mostly in retrospect. We note that, while the historical technical development of the various notions was mostly unrelated, the case of general-purpose pseudorandom generators served as a source of inspiration to most of the other cases. In particular, the concept of computational indistinguishability, the connection between hardness and pseudorandomness, and the equivalence between pseudorandomness and unpredictability, appeared first in the context of general-purpose pseudorandom generators (and inspired the development of "generators for derandomization" and "generators for space bounded machines"). Indeed, the development of the special-purpose generators (see Section 8.6) was unrelated to all of these.

[41] By the OW we denote the assumption that one-way functions exists. By EvEC we denote the assumption that the class $\mathcal{E}$ has (almost-everywhere) exponential circuit complexity.

General-purpose pseudorandom generators. The concept of *computational indistinguishability*, which underlies the entire computational approach to randomness, was suggested by Goldwasser and Micali [104] in the context of defining secure encryption schemes. Indeed, computational indistinguishability plays a key role in cryptography (see Appendix C). The general formulation of computational indistinguishability is due to Yao [221]. Yao also observed (using the hybrid technique of [104]) that defining pseudorandom generators as producing sequences that are computationally indistinguishable from the corresponding uniform distribution is equivalent to defining such generators as producing unpredictable sequences. The latter definition originates in the earlier work of Blum and Micali [37].

Blum and Micali [37] pioneered the rigorous study of pseudorandom generators and, in particular, their construction based on some simple intractability assumption (in their case, the intractability of Discrete Logarithm problem over prime fields). Their work also introduces basic paradigms that were used in all subsequent improvements (cf., e.g., [221, 113]). We refer to the trasformation of computational difficulty into pseudorandomness, the use of hard-core predicates (defined in [37]), and the iteration paradigm (cf. Eq. (8.8)).

Theorem 8.11 (by which pseudorandom generators exist if and only if one-way functions exist) is due to Håstad, Impagliazzo, Levin and Luby [113], building upon the hard-core predicate of [95] (see Theorem 7.7). Unfortunately, the current proof of Theorem 8.11 is very complicated and unfit for presentation in a book of the current nature. Presenting a simpler and tighter (cf. §8.3.7.1) proof is indeed an important research project.

Pseudorandom functions (further discussed in Appendix C.3.3) were defined and first constructed by Goldreich, Goldwasser and Micali [91]. We advocate the study of a general theory of pseudorandom objects initiated in [92].

Derandomization of time-complexity classes. As observed by Yao [221], a non-uniformly strong notion of pseudorandom generators yields improved derandomization of time-complexity classes. A key observation of Nisan [160, 163] is that whenever a pseudorandom generator is used this way, it suffices to require that the generator runs in time exponential in its seed length, and so the generator may have running-time greater than the distinguisher (representing the algorithm to be derandomized). This observation underlines the construction of Nisan and Wigderson [160, 163], and is the basis for further improvements culminating in [121]. Part 1 of Theorem 8.19 (i.e., the so-called "high end" derandomization of $\mathcal{BPP}$) is due to Impagliazzo and Wigderson [121], whereas Part 2 (the "low end") is from [163].

The Nisan–Wigderson Generator [163] was subsequently used in several ways transcending its original presentation. We mention its application towards fooling non-deterministic machines (and thus derandomizing constant-round interactive proof systems) and to the construction of randomness extractors [208].

Space Pseudorandom Generators. As stated in the first paper on the subject of space-resilient pseudorandom generators [4][42], this research direction was inspired by the derandomization result obtained via use of general-purpose pseudorandom generators. The latter result (necessarily) depends on intractability assumptions, and so the objective was to find classes of algorithms for which derandomization is possible without relying on intractability assumptions. (This objective was achieved before for the case of constant-depth circuits.) Fundamentally different constructions of space pseudorandom generators were given in several works, but are superseeded by the two incomparable results mentioned in Section 8.5.2: Theorem 8.21 (a.k.a Nisan's Generator [161]) and Theorem 8.22 (a.k.a the Nisan–Zuckerman Generator [164]). These two results have been "interpolated" in [11]. Theorem 8.23 ($\mathcal{BPL} \subseteq \mathcal{SC}$) was proved by Nisan [162].

Special Purpose Generators. The various generators presented in Section 8.6 were not inspired by any of the other types of pseudorandom generator (nor even by the generic notion of pseudorandomness). Pairwise-independence generator were explicitly suggested in [51] (and are implicit in [47]). The generalization to t-wise independence (for $t \geq 2$) is due to [6]. Small-bias generators were first defined and constructed by Naor and Naor [158], and three simple constructions were subsequently given in [9]. The Expander Random Walk Generator was suggested by Ajtai, Komlos, and Szemerédi [4], who discovered that random walks on expander graphs provide a good approximation to repeated independent attempts for hitting any arbitrary fixed subset of sufficient density (within the vertex set). The analysis of the hitting property of such walks was subsequently improved, culminating in the bound cited in Theorem 8.28, which is taken from [125, Cor. 6.1].

(The foregoing historical notes do not mention several technical contributions that played an important role in the development of the area. For further details, the reader is referred to [86, Chap. 3]. In fact, the current chapter is a revision of [86, Chap. 3], providing more details for the main topics, and omitting [86, Sec. 3.6.4 and 3.6.5] (a revision of which appears in Appendix D).)

Exercises

Exercise 8.1 Prove that placing no computational requirements on the generator yields "generators" that can fool any family of subexponential-size circuits. That is, prove that there exist functions $G : \{0,1\}^* \to \{0,1\}^*$ such that $\{G(U_k)\}_{k\in\mathbb{N}}$ is (strongly) pseudorandom, while $|G(s)| = 2|s|$ for every $s \in \{0,1\}^*$. Furthermore, show that G can be computed in double-exponential time.

Guideline: Use the Probabilistic Method (cf. [10]). First, for any fixed circuit $C : \{0,1\}^n \to \{0,1\}$, upper-bound the probabity that for a random set $S \subset \{0,1\}^n$ of size $2^{n/2}$ the absolute value of $\Pr[C(U_n) = 1] - (|\{x \in S : C(x) = 1\}|/|S|)$ is larger than $2^{-n/100}$. Next, using a union bound, prove the existence of a set $S \subset \{0,1\}^n$ of size

[42]This paper is more frequently cited for the Expander Random Walk technique which it has introduced.

$2^{n/2}$ such that no circuit of size $2^{n/100}$ can distinguish a uniformly distributed element of S from a uniformly distributed element of $\{0,1\}^n$, where distinguishing means with a probability gap of at least $2^{-n/100}$.

Exercise 8.2 Let A be a probabilistic polynomial-time algorithm solving the search associated with the NP-relation R, and let A_G be as in Construction 8.2. Prove that it is infeasible to find an x on which A_G outputs a wrong solution; that is, assuming for simplicity that A has error probability 1/3, prove that on input 1^n it is infeasible to find an $x \in \{0,1\}^n \cap L_R$ such that $\Pr[(x, A_G(x)) \notin R] > 0.4$, where $L_R \stackrel{\rm def}{=} \{x : \exists y\, (x,y) \in R\}$.
(Hint: For x that violates the claim, it holds that $|\Pr[(x, A(x)) \notin R] - \Pr[(x, A_G(x)) \notin R]| > 0.06$.)

Exercise 8.3 Prove that omitting the absolute value in Eq. (8.4) keeps Definition 8.4 intact.
(Hint: consider $D'(z) \stackrel{\rm def}{=} 1 - D(z)$.)

Exercise 8.4 Show that the existence of pseudorandom generators implies the existence of polynomial-time constructible probability ensembles that are statistically far apart and yet are computationally indistinguishable.
(Hint: lower-bound the statistical distance between $G(U_k)$ and $U_{\ell(k)}$, where G is a pseudorandom generator with stretch ℓ.)

Exercise 8.5 Prove that the sufficient condition in Exercise 8.4 is in fact necessary.[43] Recall that $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$ are said to be statistically far apart if, for some positive polynomial p and all sufficiently large n, the variation distance between X_n and Y_n is greater than $1/p(n)$. Using the following three steps, prove that the existence of *polynomial-time constructible* probability ensembles that are statistically far apart and yet are computationally indistinguishable implies the existence of pseudorandom generators.

1. Show that, without loss of generality, we may assume that the variation distance between X_n and Y_n is greater than $1 - \exp(-n)$.

   (Hint: Consider $\overline{X}_n = (X_n^{(1)}, ..., X_n^{(t(n))})$ and $\overline{Y}_n = (Y_n^{(1)}, ..., Y_n^{(t(n))})$, where the $X_n^{(i)}$'s (resp., $Y_n^{(i)}$'s) are independent copies of X_n (resp., Y_n), and $t(n) = O(n \cdot p(n)^2)$.)

2. Using $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$ as in Step 1, prove the existence of a *false entropy generator*, where a false entropy generator is a deterministic polynomial-time algorithm G such that $G(U_k)$ has entropy $e(k)$ but $\{G(U_k)\}_{k\in\mathbb{N}}$ is computationally indistinguishable from a polynomial-time constructible ensemble that has entropy greater than $e(\cdot) + (1/2)$.

   (Hint: Let S_0 and S_1 be sampling algorithms such that $X_n \equiv S_0(U_{\mathrm{poly}(n)})$ and $Y_n \equiv S_1(U_{\mathrm{poly}(n)})$. Consider the generator $G(\sigma, r) = (\sigma, S_\sigma(r))$, and the distribution Z_n that equals (U_1, X_n) with probability 1/2 and (U_1, Y_n) otherwise. Note that in $G(U_1, U_{\mathrm{poly}(n)})$ the first bit is almost determined by the rest, whereas in Z_n the first bit is statistically independent of the rest.)

[43] This exercise follows [84], which in turn builds on [113].

3. Using a false entropy generator, obtain one in which the excess entropy is $\sqrt{k}$, and using the latter construct a pseudorandom generator.

   (Hint: Use the ideas presented at the end of Section 8.3.5 (i.e., the discussion of the interesting direction of the proof of Theorem 8.11).)

Exercise 8.6 Prove that *if* $\{X_n\}_{n\in\mathbb{N}}$ *and* $\{Y_n\}_{n\in\mathbb{N}}$ *are computationally indistinguishable and* A *is a probabilistic polynomial-time algorithm then* $\{A(X_n)\}_{n\in\mathbb{N}}$ *and* $\{A(Y_n)\}_{n\in\mathbb{N}}$ *are computationally indistinguishable.*
(Hint: If D distinguishes the latter ensembles then D' such that $D'(z) \stackrel{\text{def}}{=} D(A(z))$ distinguishes the former.)

Exercise 8.7 In continuation to Exercise 8.6, show that the conclusion may not hold in case A is not computationally bounded. That is, show that there exists computationally indistinguishable ensembles, $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$, and an exponential-time algorithm A such that $\{A(X_n)\}_{n\in\mathbb{N}}$ and $\{A(Y_n)\}_{n\in\mathbb{N}}$ are *not* computationally indistinguishable.
(Hint: For any pair of ensembles $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$, consider the Boolean function f such that $f(z) = 1$ if and only if $\Pr[X_n = z] > \Pr[Y_n = z]$. Show that $|\Pr[f(X_n) = 1] - \Pr[f(Y_n) = 1]|$ equals the statistical difference between X_n and Y_n. Consider an adequate (approximate) implementation of f (e.g., approximate $\Pr[X_n = z]$ and $\Pr[Y_n = z]$ up to $\pm 2^{-2|z|}$), and use Exercise 8.1.)

Exercise 8.8 For G_1 and ℓ as in Construction 8.7, consider $G(s) \stackrel{\text{def}}{=} G_1^{\ell(|s|)}(s)$, where $G_1^i(x)$ denotes G_1 iterated i times on x (i.e., $G_1^i(x) = G_1^{i-1}(G_1(x))$ and $G_1^0(x) = x$). Prove that G is a pseudorandom generator of stretch ℓ. Reflect on the advantages of Construction 8.7 over the current construction.
(Hint: Use a hybrid argument, with the i^{th} hybrid being $G_1^i(U_{\ell(k)-i})$. Note that $G_1^{i+1}(U_{\ell(k)-(i+1)}) = G_1^i(G_1(U_{\ell(k)-i-1}))$ and $G_1^i(U_{\ell(k)-i}) = G_1^i(U_{|G_1(U_{\ell(k)-i-1})|})$, and use Exercise 8.6.)

Exercise 8.9 (pseudorandom versus unpredictability) Prove that a probability ensemble $\{Z_k\}_{k\in\mathbb{N}}$ is pseudorandom if and only if it is unpredictable. For simplicity, we say that $\{Z_k\}_{k\in\mathbb{N}}$ is (next-bit) unpredictable if for every probabilistic polynomial-time algorithm A it holds that $\Pr_i[A(F_i(Z_k)) = B_{i+1}(Z_k)]$ is negligible, where $i \in \{0, ..., |Z_k| - 1\}$ is uniformly distributed, and $F_i(z)$ (resp., $B_{i+1}(z)$) denotes the i-bit prefix (resp., $i + 1^{\text{st}}$ bit) of z.

Guideline: Show that pseudorandomness implies polynomial-time unpredictability; that is, polynomial-time predictability violates pseudorandomness (because the uniform ensemble is unpredictable regardless of computing power). Use a hybrid argument to prove that unpredictability implies pseudorandomness. Specifically, the i^{th} hybrid consists of the i-bit long prefix of Z_k followed by $|Z_k| - i$ uniformly distributed bits. Thus, distinguishing the extreme hybrids (which correspond to Z_k and $U_{|Z_k|}$) implies distinguishing some neighboring hybrids, which in turn implies next-bit predictability. For the last step, use an argument as in the proof of Proposition 8.9.

Exercise 8.10 Prove that a probability ensemble is unpredictable (from left to right) if and only if it is unpredictable from right to left (or in any other canonical order).

(Hint: use Exercise 8.9, and note that an ensemble is pseudorandom if and only if its reverse is pseudorandom.)

Exercise 8.11 Let f be 1-1 and length preserving, and b be a hard-core predicate of f. For any polynomial ℓ, prove that $\{G'(U_k)\}$ is unpredictable (in the sense of Exercise 8.9), where $G'(s) \stackrel{\text{def}}{=} b(f^{\ell(|s|)-1}(s)) \cdots b(f(s)) \cdot b(s)$.

Guideline: Suppose towards the contradiction that, for a uniformly distributed $j \in \{0, ..., \ell(k)-1\}$, given the j-bit long prefix of $G'(U_k)$ an algorithm A' can predict the $j+1^{\text{st}}$ bit of $G'(U_k)$. That is, given $b(f^{\ell(k)-1}(s)) \cdots b(f^{\ell(k)-j}(s))$, algorithm A' predicts $b(f^{\ell(k)-(j+1)}(s))$, where s is uniformly distributed in $\{0,1\}^k$. Consider an algorithm A that given $y = f(x)$ approximates $b(x)$ by invoking A' on input $b(f^{j-1}(y)) \cdots b(y)$, where j is uniformly selected in $\{0, ..., \ell(k)-1\}$. Analyze the success probability of A using the fact that f induces a permutation over $\{0,1\}^n$, and thus $b(f^j(U_k)) \cdots b(f(U_k)) \cdot b(U_k)$ is distributed identically to $b(f^{\ell(k)-1}(U_k)) \cdots b(f^{\ell(k)-j}(U_k)) \cdot b(f^{\ell(k)-(j+1)}(U_k))$.

Exercise 8.12 Prove that if G is a strong pseudorandom generator in the sense of Definition 8.12 then it a pseudorandom generator in the sense of Definition 8.1.

(Hint: consider a sequence of internal coin tosses that maximizes the probability in Eq. (8.2).)

Exercise 8.13 Show that there exists a circuit of size $O(2^k \cdot \ell(k))$ that violates Eq. (8.9), provided $\ell(k) > k$.

(Hint: The circuit may incorporate all values in the range of G and deciding by comparing its input to these values.)

Exercise 8.14 (constructing a set system for Theorem 8.18) For every $\gamma > 0$, show a construction of a set system S as in Condition 2 of Theorem 8.18, with $m(k) = \Omega(k)$ and $\ell(k) = 2^{\Omega(k)}$.

Guideline: We assume, without loss of generality, that $\gamma < 1$, and set $m(k) = (\gamma/2) \cdot k$ and $\ell(k) = 2^{\gamma m(k)/10}$. We construct the set system $S_1, ..., S_{\ell(k)}$ in iterations, selecting S_i as the first $m(k)$-subset of $[k]$ that has sufficiently small intersections with each of the previous sets $S_1, ..., S_{i-1}$. The existence of such a set S_i can be proved using the Probabilistic Method (cf. [10]). Specifically, for a fixed $m(k)$-subset S', the probability that a random $m(k)$-subset has intersection greater than $\gamma m(k)$ with S' is upper-bounded[44] by $2^{-\gamma m(k)/10}$, because the expected intersection size is $(\gamma/2) \cdot m(k)$. Thus, with positive probability a random $m(k)$-subset has intersection at most $\gamma m(k)$ with each of the previous $i - 1 < \ell(k) = 2^{\gamma m(k)/10}$ subsets. Note that we construct S_i in time $\binom{k}{m(k)} \cdot (i-1) \cdot m(k) < 2^k \cdot \ell(k) \cdot k$, and thus S is computable in time $k2^k \cdot \ell(k)^2 < 2^{2k}$.

[44]Applying the standard Chernoff Bound yields an upper-bound of $\exp(-(\gamma/2)^2 \cdot m(k))$, which suffices for $\ell(k) = 2^{\gamma^2 m(k)/10}$. However, using a Multiplicative Chernoff Bound yields an upper-bound of $\exp(-(\gamma/2) \cdot m(k)/3)$.

Exercise 8.15 Suppose that the sets S_i's in Construction 8.17 are disjoint and that $f : \{0,1\}^m \to \{0,1\}$ is T-inapproximable. Prove that for every circuit C of size $T - O(1)$ it holds that $|\mathsf{Pr}[C(G(U_k)) = 1] - \mathsf{Pr}[C(U_\ell) = 1]| < \ell/T$.

Guideline: Prove the contrapositive using Exercise 8.16. Note that the values of the $i+1^{\text{st}}$ bit of $G(U_k)$ is statistically independent of the values of the first i bits of $G(U_k)$, and thus predicting it yields an approximator for f. Indeed, such an approximator can be obtained by fixing the the first i bits of $G(U_k)$ via an averaging argument.

Exercise 8.16 In continuation to Exercise 8.9, show that if there exists a circuit of size s that distinguishes Z_n from U_ℓ with gap δ, then there exists an $i < \ell = |Z_n|$ and a circuit of size $s + O(1)$ that given an i-bit long prefix of Z_n guesses the $i+1^{\text{st}}$ bit with success probability at least $\frac{1}{2} + \frac{\delta}{\ell}$.
(Hint: defining hybrids as in Exercise 8.9, note that, for some i, the given circuit distinguishes the i^{th} hybrid from the $i+1^{\text{st}}$ hybrid with gap at least δ/ℓ.)

Exercise 8.17 (Theorem 8.18, generalized) Let $\ell, m, m', T : \mathbb{N} \to \mathbb{N}$ satisfy $\ell(k)^2 + \widetilde{O}(\ell(k)2^{m'(k)}) < T(m(k))$. Suppose that the following two conditions hold:

1. There exists an exponential-time computable function $f : \{0,1\}^* \to \{0,1\}$ that is T-inapproximable.
2. There exists an exponential-time computable function $S : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that $|S(k,i)| = m(k)$ for every k and $i = 1, ..., \ell(k)$, and $|S(k,i) \cap S(k,j)| \leq m'(k)$ for every k and $i \neq j$.

Prove that using G as defined in Construction 8.17, with $S_i = S(k,i)$, yields a canonical derandomizer with stretch ℓ.
(Hint: following the proof of Theorem 8.18, just note that the circuit constructed for approximating $f(U_{m(k)})$ has size $\ell(k)^2 + \ell(k) \cdot \widetilde{O}(2^{m'(k)})$ and success probability at least $(1/2) + (1/7\ell(k))$.)

Exercise 8.18 (Part 2 of Theorem 8.19) Prove that if for every polynomial T there exists a T-inapproximable predicate in $\mathcal{E}$ then $\mathcal{BPP} \subseteq \cap_{\varepsilon > 0} \text{DTIME}(t_\varepsilon)$, where $t_\varepsilon(n) \stackrel{\text{def}}{=} 2^{n^\varepsilon}$.
(Hint: For any p-time algorithm, apply Exercise 8.17 using $\ell(k) = p(k^{1/\varepsilon})$, $m(k) = \sqrt{k}$ and $m'(k) = O(\log k)$. Revisit Exercise 8.14 in order to obtain a set system as required in Exercise 8.17 (for these parameters), and use Theorem 7.10.)

Exercise 8.19 Provide an explicit description of the generator outlined in the proof of Theorem 8.21.
(Hint: for $r \in \{0,1\}^n$ and $h^{(1)}, ..., h^{(t)} \in H_n$, the generaor outputs a 2^t-long sequence of n-bit strings such that the i^{th} block equals $h'(r)$, where h' is a composition of some of the $h^{(j)}$'s.)

Exercise 8.20 (adaptive t-wise independence tests) Prove that the output of a t-wise independence generator is indistinguishable to any test than examines t of the blocks, even if the examined blocks are selected adaptively (i.e., the location of the i^{th} block is determined based on the contents of the previously inspected blocks).

Guideline: First show that, without loss of generality, it suffices to consider deterministic (adaptive) tester. Next, show that the probability that such a tester sees any fixed sequence of t values at locations selected adaptively in the generator's output is $2^{-t \cdot b(k)}$, where $b(k)$ is the block length.

Exercise 8.21 (t-wise independence generator) Prove that G as defined in Proposition 8.24 produces a t-wise independent sequence over $\mathrm{GF}(2^{b(k)})$.

Guideline: For every t fixed indices $i_1, ..., i_t \in [\ell'(k)]$, consider the distribution of $G(U_k)_{i_1,...,i_t}$ (i.e., the projection of $G(U_k)$ on locations $i_1, ..., i_t$). Show that for every sequence of t possible values $v_1, ..., v_t \in \mathrm{GF}(2^{b(k)})$, there exists a unique seed $s \in \{0,1\}^k$ such that $G(s)_{i_1,...,i_t} = (v_1, ..., v_t)$.

Exercise 8.22 (pairwise independence generators) As a warm-up, consider a construction analogous to the one in Proposition 8.25, where the seed specifies an affine $b(k)$-by-$m(k)$ transformation. That is, for $s \in \{0,1\}^{b(k) \cdot m(k)}$ and $r \in \{0,1\}^{b(k)}$, where $k = b(k) \cdot m(k) + b(k)$, let

$$G(s, r) \stackrel{\text{def}}{=} (A_s v_1 + r \, , \, A_s v_2 + r \, , ..., \, A_s v_{\ell'(k)} + r) \tag{8.17}$$

where A_s is an $b(k)$-by-$m(k)$ matrix specified by the string s. Show that G as in Eq. (8.17) is a pairwise independence generator of block-size b and stretch ℓ. (Note that a related construction appears in the proof of Theorem 7.7; see also Exercise 7.5.) Next, show that G as in Eq. (8.13) is a pairwise independence generator of block-size b and stretch ℓ.

Guideline: The following description applies to both constructions. First note that for every fixed $i \in [\ell'(k)]$, the i^{th} element in the sequence $G(U_k)$ is uniformly distributed in $\{0,1\}^{b(k)}$. Actually, show that for every fixed $s \in \{0,1\}^{k-b(k)}$, it holds that $G(s, U_{b(k)})_i$ is uniformly distributed in $\{0,1\}^{b(k)}$. Next note that it suffices to show that, for every $j \neq i$, conditioned on the value of the i^{th} element in $G(U_k)$, the j^{th} element is uniformly distributed in $\{0,1\}^{b(k)}$. The key technical detail is to show that for any non-zero vector $v \in \{0,1\}^{m(k)}$ it holds that $A_{U_{k-b(k)}} v$ (resp., $T_{U_{k-b(k)}} v$) is uniformly distributed in $\{0,1\}^{b(k)}$. This is easy in case of a random $b(k)$-by-$m(k)$ matrix, and can be proven also for a random Toeplitz matrix.

Exercise 8.23 (adaptive t-wise independence tests, revisited) In contrast to Exercise 8.20, we note that almost uniform distribution on any fixed t bit locations does not imply that an adaptive test that inspects t locations cannot detect "non-uniformity" (i.e., a "non random behavior" of the inspected sequence). Specifically, present a distribution over 2^{t-1}-bit long strings in which each $t-1$ fixed bit positions are $t \cdot 2^{-(t-1)}$-close to uniform, but some test that adaptively inspects t positions can distinguish this distribution from the uniform one with constant gap. (Hint: Modify the uniform distribution over $((t-1) + 2^{t-1})$-bit long strings such that the first $t-1$ locations indicate a bit position (among the rest) that is set to zero.)

Exercise 8.24 Suppose that G is an ε-bias generator with stretch ℓ. Show that equality between the $\ell(k)$-bit strings x and y can be probabilistically checked by comparing the inner product modulo 2 of x and $G(s)$ to the inner product modulo 2 of y and $G(s)$, where $s \in \{0,1\}^k$ is selected uniformly.

(Hint: reduce the problem to the special case in which $y = 0^{\ell(k)}$.)

Exercise 8.25 (bias versus statistical difference from uniform) Let X be a random variable assuming values in $\{0,1\}^t$. Prove that if X has bias at most ε over any non-empty set then the statistical difference between X and U_t is at most $2^{t/2} \cdot \varepsilon$, and that for every $x \in \{0,1\}^t$ it holds that $\Pr[X = x] = 2^{-t} \pm \varepsilon$.

Guideline: Consider the probability function $p : \{0,1\}^t \to [0,1]$ defined by $p(x) \stackrel{\text{def}}{=} \Pr[X = x]$, and let $\delta(x) \stackrel{\text{def}}{=} p(x) - 2^{-t}$ denote the deviation of p from the uniform probability function. Viewing the set of real functions over $\{0,1\}^t$ as a 2^t-dimensional vector space, we consider two orthonormal bases for this space. The first basis is of the (Kroniker) functions $\{k_\alpha\}_{\alpha \in \{0,1\}^t}$ such that $k_\alpha(x) = 1$ if $x = \alpha$ and $k_\alpha(x) = 0$ otherwise. The second basis is of the (normalize Fourier) functions $\{f_S\}_{S \subseteq [t]}$ defined by $f_S(x) \stackrel{\text{def}}{=} 2^{-t/2} \prod_{i \in S} (-1)^{x_i}$ (where $f_\emptyset \equiv 2^{-t/2}$).[45] Note that the bias of X over any $S \neq \emptyset$ equals $|\sum_x p(x) \cdot 2^{t/2} f_S(x)|$, which in turn equals $2^{t/2} |\sum_x \delta(x) f_S(x)|$. Thus, for every S (including the empty set), we have $|\sum_x \delta(x) f_S(x)| \leq 2^{-t/2}\varepsilon$, which means that the representation of δ in the normalize Fourier basis is by coefficients that have each an absolute value of at most $2^{-t/2}\varepsilon$. Thus, the Norm-2 of this vector of coefficients is bounded by $\sqrt{2^t \cdot (2^{-t/2}\varepsilon)^2} = \varepsilon$, and all claims follow by noting that they refer to norms of δ according to the Kroniker basis. In particular, Norm-2 is preserved under orthonormal bases, the max-norm is upper-bounded by Norm-2, and Norm-1 is upper-bounded by $\sqrt{2^t}$ times the value of the Norm-2.

Exercise 8.26 (The LFSR small-bias generator (following [9])) Using the following guidelines (and letting $t = k/2$), analyze the construction outlined following Theorem 8.26 (and depicted in Figure 8.5):

1. Prove that $r_i = \sum_{j=0}^{t-1} c_j^{(i)} \cdot s_j$, where $c_j^{(i)}$ is the coefficient of z^j in the (degree $t-1$) polynomial obtained by reducing z^i modulo the polynomial $f(z)$ (i.e., $z^i \equiv \sum_{j=0}^{t-1} c_j^{(i)} z^j \pmod{f(z)}$).

   (Hint: Recall that $z^t \equiv \sum_{j=0}^{t-1} f_j z^j \pmod{f(z)}$, and thus $z^i \equiv \sum_{j=0}^{t-1} f_j z^{i-t+j} \pmod{f(z)}$. Note the correspondance to $r_i = \sum_{j=0}^{t-1} f_j \cdot r_{i-t+j}$.)

2. For any non-empty $S \subseteq \{0, ..., \ell(k)-1\}$, evaluate the bias of the sequence $r_0, ..., r_{\ell(k)-1}$ over S, where f is a random irreducible polynomial of degree t and $s = (s_0, ..., s_{t-1}) \in \{0,1\}^t$ is uniformly distributed. Specifically:

   (a) For a fixed f and random $s \in \{0,1\}^t$, prove that $\sum_{i \in S} r_i$ has non-zero bias if and only if $f(z)$ divides $\sum_{i \in S} z^i$.

   (Hint: Note that $\sum_{i \in S} r_i = \sum_{j=0}^{t-1} \sum_{i \in S} c_j^{(i)} s_j$, and use Item 1.)

[45] Verify that both bases are indeed orthogonal (i.e., $\sum_x k_\alpha(x) k_\beta(x) = 0$ for every $\alpha \neq \beta$ and $\sum_x f_S(x) f_T(x) = 0$ for every $S \neq T$) and normal (i.e., $\sum_x k_\alpha(x)^2 = 1$ and $\sum_x f_S(x)^2 = 1$).

(b) Prove that the probability that a random irreducible polynomial of degree t divides $\sum_{i\in S} z^i$ is $\Theta(\ell(k)/2^t)$.

(Hint: A polynomial of degree n can be divided by at most n/d different irreducible polynomials of degree d. On the other hand, the number of irreducible polynomials of degree d over GF(2) is $\Theta(2^d/d)$.)

Conclude that for random f and s, the sequence $r_0, ..., r_{\ell(k)-1}$ has bias $O(\ell(k)/2^t)$.

Note that an implementation of the LFSR generator requires a mapping of random $k/2$-bit long seeds to almost random irreducible polynomials of degree $k/2$. Such a mapping can be constructed in $\exp(k)$ time, which is $\mathrm{poly}(\ell(k))$ if $\ell(k) = \exp(\Omega(k))$. A more efficient mapping that uses a $O(k)$-bit long seek is described in [9, Sec. 8].

Exercise 8.27 (limitations on small-bias generators) Let G be an ε-bias generator with stretch ℓ, and view G as a mapping from $\mathrm{GF}(2)^k$ to $\mathrm{GF}(2)^{\ell(k)}$. As such, each bit in the output of G can be viewed as a polynomial in the k input variables (each ranging in GF(2)). Prove that if $\varepsilon(k) < 1$ and each of these polynomials has degree at most d then $\ell(k) \leq \sum_{i=1}^{d} \binom{k}{i}$.

Guideline: First note that, without loss of generality, all polynomials have a free term equal to zero. Then, consider the vector space spanned by all d-monomials over k variables (i.e., monomial having at most d variables). Since $\varepsilon(k) < 1$, the polynomials representing the output bits of G must correspond to a sequence of independent vectors in this space. Derive the following corollaries:

1. If $\varepsilon(k) < 1$ then $\ell(k) < 2^k$.
2. If $\varepsilon(k) < 1$ and $\ell(k) > k$ then G cannot be a linear transformation.

Note that $G(s) = (s, b(s))$, where $b(s_1, ..., s_k) = \sum_{i=1}^{k/2} s_i s_{(k/2)+i} \bmod 2$, is an ε-biased generator with $\varepsilon(k) = \exp(-\Omega(k))$.

(Hint: Focusing on bias over sets that include the last output bit, prove that without loss of generality it suffices to analyze the bias of $b(U_k)$.)

Exercise 8.28 (a sanity check for pseudorandomness) The following fact is suggested as a sanity check for candidate pseudorandom generators with respect to space-bounded machines. The fact (to be proven as an exercise) is that, for every $\varepsilon(\cdot)$ and $s(\cdot)$ such that $s(k) \geq 1$ for every k, if G is (s, ε)-pseudorandom (as per Definition 8.20), then G is an ε-bias generator.

Exercise 8.29 (approximate t-wise independent generators (following [158])) Combining Theorem 8.26 and relying on the linearity of the t-wise independent generator of Eq. (8.12), construct a generator producing ℓ-bit long sequences in which any t positions are at most ε-away from uniform (in variation distance), while using a seed of length $O(t + \log(1/\varepsilon) + \log\log \ell)$. (For max-norm a seed of length $O(\log(t/\varepsilon) + \log\log \ell)$ suffices.)

Guideline: First note that, for any t, ℓ' and b, the transformation of Eq. (8.12) can be implemented by a fixed linear (over GF(2)) transformation of a $t{\cdot}b$-bit seed into an ℓ-bit bit sequence, where $\ell = \ell' \cdot b$. It follows that there exists a fixed GF(2)-linear transformation T of a random seed of length $t \cdot b$, where $b = \log_2 \ell$, into a t-wise independent bit sequence of the length ℓ (i.e., $T\,U_{t \cdot b}$ is t-wise independent over $\{0,1\}^\ell$). Thus, every t rows of T are linearly independent. The key observation is that when we replace the aforementioned random seed by an ε'-biased sequence, every $i \leq t$ positions in the output sequence have bias at most ε' (because they define a non-zero linear test on the bits of the ε'-biased sequence). Note that the length of the new seed (used to produce ε'-biased sequence of length $t \cdot b$) is $O(\log tb/\varepsilon')$. Applying Exercise 8.25, we conclude that any t positions are at most $2^{t/2} \cdot \varepsilon'$-away from uniform (in variation distance). Recall that this was obtained using a seed of length $O(\log(t/\varepsilon') + \log\log \ell)$, and the claim follows by using $\varepsilon' = 2^{-t/2} \cdot \varepsilon$.

Exercise 8.30 (A version of the Expander Random Walk Theorem) Using notations as in Theorem 8.28, prove that the probability that a random walk of length ℓ stays in W is at most $(\rho + (\lambda/d)^2)^{\ell/2}$. In fact, prove a more general claim that refers to the probability that a random walk of length ℓ intersects $W_0 \times W_1 \times \cdots \times W_{\ell-1}$. The claimed upper-bound is

$$\sqrt{\rho_0} \cdot \prod_{i=1}^{\ell-1} \sqrt{\rho_i + (\lambda/d)^2}, \tag{8.18}$$

where $\rho_i \stackrel{\text{def}}{=} |W_i|/|V|$.

Guideline: View the random walk as the evolution of a corresponding probability vector under suitable transformations. The transformations correspond to taking a random step in the graph and to passing through a "sieve" that keeps only the entries that correspond to the current set W_i. The key observation is that the first trasformation shrinks the component that is orthogonal to the uniform distribution (which is the first eigenvalue of the adjacency matrix of the expander), whereas the second trasformation shrinks the component that is in the direction of the uniform distribution. For further details, see §E.2.1.3.

Exercise 8.31 Using notations as in Theorem 8.28, prove that the probability that a random walk of length ℓ visits W more than $\alpha\ell$ times is smaller than $\binom{\ell}{\alpha\ell} \cdot (\rho + (\lambda/d)^2)^{\alpha\ell/2}$. For example, for $\alpha = 1/2$ and $\lambda/d < \sqrt{\rho}$, we get an upper-bound of $(32\rho)^{\ell/4}$. We comment that much better bounds can be obtained (cf. [82]).
(Hint: Use a union bound on all possible sequences of $m = \alpha\ell$ visits, and upper-bound the probability of visiting W in steps $j_1, \ldots, j_m$ by applying Eq. (8.18) with $W_i = W$ if $i \in \{j_1, \ldots, j_m\}$ and $W = V$ otherwise.)

Chapter 9

Probabilistic Proof Systems

A proof is whatever convinces me.

Shimon Even (1935–2004)

Various types of *probabilistic* proof systems have played a central role in the development of computer science in the last couple of decades. In this chapter, we concentrate on three such proof systems: *interactive proofs*, *zero-knowledge proofs*, and *probabilistic checkable proofs*. These proof systems share a common (untraditional) feature – they carry a probability of error; yet, this probability is explicitly bounded and can be reduced by successive application of the proof system. The gain in allowing this untraditional relaxation is substantial, as demonstrated by the three results mentioned in the summary.

Summary: The association of efficient procedures with *deterministic* polynomial-time procedures is the basis for viewing NP-proof systems as the canonical formulation of proof systems (with efficient verification procedures). Allowing *probabilistic* verification procedures and, moreover, ruling by statistical evidence gives rise to various types of probabilistic proof systems. These probabilistic proof systems carry a probability of error (which is explicitly bounded), but they offer various advantages over the traditional (deterministic and errorless) proof systems.

Randomized and interactive verification procedures, giving rise to *interactive proof systems*, seem much more powerful than their deterministic counterparts. In particular, such interactive proof systems exist for any set in $\mathcal{PSPACE} \supseteq \text{co}\mathcal{NP}$ (e.g., for the set of unsatisfied propositional formulae), whereas it is widely believed that some sets in $\text{co}\mathcal{NP}$ do *not* have NP-proof systems (i.e., $\mathcal{NP} \neq \text{co}\mathcal{NP}$). We stress that a "proof" in this context is not a fixed and static object, but rather a randomized (and dynamic) process in which the verifier interacts with the prover.

Intuitively, one may think of this interaction as consisting of questions asked by the verifier, to which the prover has to reply convincingly.

Such randomized and interactive verification procedures allow for the introduction of *zero-knowledge proofs*, which are of great conceptual and practical interest (especially in cryptography). Loosely speaking, zero-knowledge proofs are interactive proofs that yield nothing (to the verifier) beyond the fact that the assertion is indeed valid. For example, a zero-knowledge proof that a certain propositional formula is satisfiable does not reveal a satisfying assignment to the formula nor any partial information regarding such an assignment (e.g., whether the first variable can assume the value `true`). Thus, zero-knowledge proofs exhibit an extreme contrast between being convinced of the validity of a statement and learning anything in addition (while receiving such a convincing proof). Under reasonable complexity assumptions (i.e., assuming the existence of one-way functions), every set in $\mathcal{NP}$ has a zero-knowledge proof system.

NP-proofs can be efficiently transformed into a (redundant) form that offers a trade-off between the number of (randomly) examined locations in the resulting proof and the confidence in its validity. It particular, it is known that any set in $\mathcal{NP}$ has an NP-proof system that supports probabilistic verification such that the error probability decreases exponentially with the number of bits read from the alleged proof. These redundant NP-proofs are called *probabilistically checkable proofs* (or PCPs). In addition to their conceptually fascinating nature, PCPs have played a key role in the study of the complexity of approximation problems.

Introduction and Preliminaries

The glory attached to the creativity involved in finding proofs, makes us forget that it is the less glorified procedure of verification that gives proofs their value. Conceptually speaking, proofs are secondary to the verification procedure; whereas technically speaking, proof systems are defined in terms of their verification procedures.

The notion of a verification procedure presumes the notion of computation and furthermore the notion of efficient computation. This implicit stipulation is made explicit in the definition of $\mathcal{NP}$ (cf. Definition 2.5), in which efficient computation is associated with (deterministic) polynomial-time algorithms.[1] Thus, NP provides the ultimate formulation of proof systems (with efficient verification procedures)

[1]Recall that the formulation of NP-proof systems explicitly restricts the length of proofs to be polynomial in the length of the assertion. Thus, verification is performed in a number of steps that is polynomial in the length of the assertion. We comment that deterministic proof systems that allow for longer proofs (but require that verification is efficient in terms of the length of the alleded proof) can be modeled as NP-proof systems by adequate padding (of the assertion).

as long as one associates efficient procedures with *deterministic* polynomial-time algorithms. However, we can gain a lot if we are willing to take a somewhat non-traditional step and allow *probabilistic* verification procedures. In particular:

- Interactive proof systems, which employ randomized and interactive verification procedures, seem much more powerful than their deterministic counterparts.
- Such interactive proof systems allow for the introduction of zero-knowledge proofs, which are of great theoretical and practical interest.
- NP-proofs can be efficiently transformed into a (redundant) form that offers a trade-off between the number of randomly examined locations in the alleged proof and the confidence in its validity.

In all these cases, explicit bounds are imposed on the computational complexity of the verification procedure, which in turn is personified by the notion of a verifier. Furthermore, in all these proof systems, the verifier is allowed to toss coins and rule by statistical evidence. Thus, all these proof systems carry a probability of error; yet, this probability is explicitly bounded and, furthermore, can be reduced by successive application of the proof system.

One important convention. When presenting a proof system, we state all complexity bounds in terms of the length of the assertion to be proven (which is viewed as an input to the verifier). Namely, when we say "polynomial-time" we mean time that is polynomial in the length of this assertion. Actually, as will become evident, this is *the* natural choice in all the cases that we consider. Note that this convention is consistent with the definition of NP-proof systems (cf. Definition 2.5), because $\text{poly}(|(x,y)|) = \text{poly}(|x|)$ for $|y| = \text{poly}(|x|)$.

Notational Conventions. Denote by `poly` the set of all integer functions bounded by a polynomial and by `log` the set of all integer functions bounded by a logarithmic function (i.e., $f \in \texttt{log}$ iff $f(n) = O(\log n)$). All complexity measures mentioned in the subsequent exposition are assumed to be constructible in polynomial-time.

Organization. In Section 9.1 we present the basic definitions and results regarding interactive proof systems. The definition of an interactive proof systems is the starting point for a discussion of zero-knowledge proofs, which is provided in Section 9.2. Section 9.3, which presents the basic definitions and results regarding probabilistically checkable proofs (PCP), can be read independently of the other sections.

9.1 Interactive Proof Systems

In light of the growing acceptability of randomized and distributed computations, it is only natural to associate the notion of efficient computation with probabilistic and interactive polynomial-time computations. This leads naturally to the notion of an interactive proof system in which the verification procedure is interactive and randomized, rather than being non-interactive and deterministic. Thus, a "proof" in this context is not a fixed and static object, but rather a randomized (dynamic) process in which the verifier interacts with the prover. Intuitively, one may think of this interaction as consisting of questions asked by the verifier, to which the prover has to reply convincingly. The foregoing discussion, as well as the definition provided in Section 9.1.1, makes explicit reference to a prover, whereas a prover is only implicit in the traditional definitions of proof systems (e.g., NP-proof systems). Before turning to the actual definition, we highlight and further discuss some of the foregoing issues.

A static object versus an interactive process. Traditionally in mathematics, a "proof" is a *fixed* sequence consisting of statements that are either self-evident or are derived from previous statements via self-evident rules. Actually, both conceptually and technically, it is more accurate to substitute the phrase "self-evident" by the phrase "commonly agreed" (because, at the last account, self-evidence is a matter of common agreement). In fact, in the formal study of proofs (i.e., logic), the commonly agreed statements are called *axioms*, whereas the commonly agreed rules are referred to as *derivation rules*. We highlight *a key property of mathematics proofs: proofs are viewed as fixed* (*static*) *objects*. In contrast, in other areas of human activity, the notion of a "proof" has a much wider interpretation. In particular, a proof is not a fixed object but rather a process by which the validity of an assertion is established. For example, the cross-examination of a witness in court is considered a proof in law, and failure to answer a rival's claim is considered a proof in daily discussions. The latter convention (by which failure to answer a claim is considered a proof) is common in philosophy, politics, and often even in Science. Furthermore, some technical "proofs by contradiction" take a similar attitude by emulating an imaginary debate with a potential generic skeptic.

We note that, in mathematics, proofs are often considered more fundamental than their consequence (i.e., the theorem). In contrast, in many daily situations, proofs are considered secondary (in importance) to their consequence.

The aforementioned daily attitudes will be adequate in the current chapter, where proofs are the object of study. We will be interested in modeling proofs that can be verified by automated procedures, which are designed to check the validity of potential proofs and are oblivious of additional features such as beauty, insight, etc. Thus, we view proofs merely as a vehicle for the verification of the validity of the claimed assertion. In fact, this attitude gets to an extreme in the case of zero-knowledge proofs, where we actually require that the proofs themselve be useless beyond being convincing of the validity of the claimed assertion. We note that the proof systems that we study refer to mundane theorems (e.g., asserting that a *specific* propositional formula is not satisfiable or that a party sent a message as instructed by a predetermined protocol). We stress that the (meta) theorems that we shall state regarding these proof systems will be proven in the traditional mathematical sense.

Prover and Verifier. The notion of a prover is implicit in all discussions of proofs, be it in mathematics or in other situations: the prover is the (sometimes hidden or transcendental) entity providing the proof. In contrast, the notion of a verifier tends to be more explicit in such discussions, which typically emphasize the *verification process*, or in other words the role of the verifier. Both in mathematics and in daily situations, proofs are defined in terms of the verification procedure. The verification procedure is considered to be relatively simple, and the burden is placed on the party/person supplying the proof (i.e., the prover). The asymmetry between the complexity of the verification task and the complexity of the theorem-proving task is captured by the definition of NP-proof systems (i.e., verification is required to be efficient whereas $\mathcal{P} \neq \mathcal{NP}$ implies that in some cases finding adequate proofs is infeasible).

We highlight the "distrustful attitude" towards the prover, which underlies any proof system. If the verifier trusts the prover then no proof is needed. Hence, whenever discussing a proof system one considers a setting in which the verifier is not trusting the prover, and furthermore is skeptic of anything that the prover says.

Completeness and Soundness. Two fundamental properties of a proof system (i.e., of a verification procedure) are its *soundness* (or *validity*) and *completeness*. The soundness property asserts that the verification procedure cannot be "tricked" into accepting false statements. In other words, *soundness* captures the verifier's ability to protect itself from being convinced of false statements (no matter what the prover does in order to fool it). On the other hand, *completeness* captures the ability of some prover to convince the verifier of true statements (belonging to some predetermined set of true statements). Note that both properties are essential to the very notion of a proof system.

We note that not every set of true statements has a "reasonable" proof system in which each of these statements can be proven (while no false statement can be "proven"). This fundamental fact is given a precise meaning in results such as *Gödel's Incompleteness Theorem* and Turing's theorem regarding the *undecidability of the Halting Problem*. This section is devoted to the study of a liberal notion of "reasonable proof systems".

9.1.1 Definition

Loosely speaking, an interactive proof is a game between a computationally bounded verifier and a computationally unbounded prover whose goal is to convince the verifier of the validity of some assertion. Specifically, the verifier employs a probabilistic polynomial-time strategy. It is required that if the assertion holds then the verifier always accepts (i.e., when interacting with an appropriate prover strategy). On the other hand, if the assertion is false then the verifier must reject with probability at least $\frac{1}{2}$, no matter what strategy is being employed by the prover. (The error probability can be reduced by running such a proof system several times.)

Formally, a strategy for a party describes the *party's next move* (i.e., its next message or its final decision) *as a function of the common input* (i.e., the aforementioned assertion), *its internal coin tosses, and all messages it has received so far.* That is, we assume that each party records the outcomes of its past coin tosses as well as all the messages it has received, and determines its moves based on these. Thus, an interaction between two parties, employing strategies A and B respectively, is determined by the common input, denoted x, and the randomness of both parties, denoted r_A and r_B. Assuming that A takes the first move (and B takes the last one), the corresponding interaction transcript (on common input x and randomness r_A and r_B) is $\alpha_1, \beta_1, ..., \alpha_t, \beta_t$, where $\alpha_i = A(x, r_A, \beta_1, ..., \beta_{i-1})$ and $\beta_i = B(x, r_B, \alpha_1, ..., \alpha_i)$. The corresponding final decision of A is defined as $A(x, r_A, \beta_1, ..., \beta_t)$.

We say that a party employs a probabilistic polynomial-time strategy if its next move can be computed in a number of steps that is *polynomial in the length of the common input.* In particular, this means that, on input common input x, the strategy may only consider a polynomial in $|x|$ many messages, which are each of $\text{poly}(|x|)$ length.[2] Intuitively, if the other party exceeds an a priori (polynomial in $|x|$) bound on the total length of the messages that it is allowed to send, then the execution is suspended. Referring to the foregoing strategies, we say that A is a probabilistic polynomial-time strategy if, for every i and $r_A, \beta_1, ..., \beta_i$, the value of $A(x, r_A, \beta_1, ..., \beta_i)$ can be computed in time polynomial in $|x|$. Again, in proper use, it must hold that $|r_A|, t$ and the $|\beta_i|$'s are all polynomial in $|x|$.

Definition 9.1 (Interactive Proof systems – IP):[3] *An* interactive proof system for a set S *is a two-party game, between a* verifier *executing a probabilistic polynomial-time strategy* (denoted V) *and a* prover *that executes a computationally unbounded strategy* (denoted P)*, satisfying the following two conditions:*

- Completeness: *For every $x \in S$, the verifier V always accepts after interacting with the prover P on common input x.*
- Soundness: *For every $x \notin S$ and every strategy P^*, the verifier V rejects with probability at least $\frac{1}{2}$ after interacting with P^* on common input x.*

We denote by $\mathcal{IP}$ the class of sets having interactive proof systems.

The error probability (in the soundness condition) can be reduced by successive applications of the proof system. (This is easy to see in the case of sequential repetitions, but holds also for parallel repetitions; see Exercise 9.1.) In particular, repeating the proving process for k times, reduces the probability that the verifier is fooled (i.e., accepts a false assertion) to 2^{-k}, and we can afford doing so for any $k = \text{poly}(|x|)$. Still, we have relaxed the requirements from the verification procedure by allowing it to interact with the prover, toss coins and risk some (bounded) error probability. (Variants on the basic definition are discussed in Section 9.1.3.)

[2] Needless to say, the number of internal coin tosses fed to a polynomial-time strategy must also be bounded by a polynomial in the length of x.

[3] We follow the convention of specifying strategies for both the verifier and the prover. An alternative presentation only specifies the verifier's strategy, while rephrasing the completeness condition as follows: *There exists a prover strategy P so that, for every $x \in S$, the verifier V always accepts after interacting with P on common input x.*

The role of randomness. Randomness is essential to the power of interactive proofs; that is, if randomness is not allowed, then interactive proof systems collapse to NP-proof systems. The reason being that, in case the verifier is deterministic, the prover can predict the verifier's part of the interaction. Thus, the prover can just supply its own answers to the verifier's (predictable) questions, and the verifier can just check that these answers are convincing. Actually we establish that soundness error (and not merely randomized verification) is essential to the power of interactive proof systems (i.e., their ability to reach beyond NP-proofs).

Proposition 9.2 *Suppose that S has an interactive proof system (P, V) with* no soundness error*; that is, for every $x \notin S$ and every potential strategy P^*, the verifier V rejects with probability* one *after interacting with P^* on common input x. Then $S \in \mathcal{NP}$.*

Proof: We may assume, without loss of generality, that V is deterministic (by just fixing arbitrarily the contents of its random-tape and noting that both (perfect) completeness and perfect (i.e., errorless) soundness still hold). Since V is deterministic, the prover may predict each message sent by V, and thus a sequence of optimal prover's messages (i.e., a sequence of messages leading V to accept x) can be (pre)determined (without interacting with V) *based solely on the common input x*. (In contrast, in the case that V is randomized, its random coins may effect the messages that it sends, which in turn may effect the optimal prover's responses.) Thus, $x \in S$ if and only if there exists a sequence of (prover's) messages that make (the deterministic) V accept x. It follows that $S \in \mathcal{NP}$. ■

Indeed, the punch-line of the foregoing proof is that the prover gains nothing from interacting with an easily predictable verifier (i.e., a verifier that determines its messages in deterministic polynomial-time based on the common input and the prover's prior messages). The prover can just produce the entire interaction by itself (and send it to the verifier for verification). *The moral is is that there is no point to interact with a party whose moves are easily predictable.* This moral represents the prover's point of view (regarding deterministic verifiers). Certainly, from the verifier's point of view it is beneficial to interact with the prover, because the latter is computationally stronger (and thus its moves may not be easily predictable by the verifier even in case they are predictable in an information theoretic sense).

9.1.2 The Power of Interactive Proofs

We have seen that randomness is essential to the power of interactive proof systems in the sense that without randomness interactive proofs are not more powerful than NP-proofs. Indeed, the power of interactive proof arises from the combination of randomization and interaction. We first demonstrate this point by a simple proof system for a specific coNP-set that is not known to have an NP-proof system, and next prove the celebrated result $\mathcal{IP} = \mathcal{PSPACE}$, which suggests that interactive proofs are much stronger than NP-proofs.

9.1.2.1 A simple example

> *One day on the Olympus, bright-eyed Athena claimed that Nectar poured out of the new silver-coated jars tastes less good than Nectar poured out of the older gold-decorated jars. Mighty Zeus, who was forced to introduce the new jars by the practically oriented Hera, was annoyed at the claim. He ordered that Athena be served one hundred glasses of Nectar, each poured at random either from an old jar or from a new one, and that she tell the source of the drink in each glass. To everybody's surprise, wise Athena correctly identified the source of each serving, to which the Father of the Gods responded "my child, you are either right or extremely lucky." Since all gods knew that being lucky was not one of the attributes of Pallas-Athena, they all concluded that the impeccable goddess was right in her claim.*

The foregoing story illustrates the main idea underlying the interactive proof for Graph Non-Isomorphism, presented in Construction 9.3. Informally, this interactive proof system is designed for proving dissimilarity of two given objects (in the foregoing story these are the two brands of Nectar, whereas in Construction 9.3 these are two non-isomorphic graphs). We note that, typically, proving similarity between objects is easy, because one can present a mapping (of one object to the other) that demonstrates this similarity. In contrast, proving dissimilarity seems harder, because in general there seems to be no succinct proof of dissimilarity. More generally, it is typically easy to prove the existence of an easily verifiable structure in the given object by merely presenting this structure, but proving the non-existence of such a structure seems hard. Formally, membership in an NP-set is proved by presenting an NP-witness, but it is not clear how to prove the non-existence of such witness. Indeed, recall that the common belief is that co$\mathcal{NP} \neq \mathcal{NP}$.

Recall that two graphs, $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$, are called isomorphic if there exists a 1-1 and onto mapping, ϕ, from the vertex set V_1 to the vertex set V_2 such that $\{u,v\} \in E_1$ if and only if $\{\phi(v),\phi(u)\} \in E_2$. The ("edge preserving") mapping ϕ, if existing, is called an *isomorphism* between the graphs. The following protocol allows proving that two graphs are not isomorphic. (We note that it is not known whether or not such a statement can be proven via an NP-proof system.)

Construction 9.3 (Interactive proof for Graph Non-Isomorphism):

- Common Input: *A pair of graphs, $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$. Suppose, without loss of generality, that $V_1 = \{1,2,...,|V_1|\}$, and similarly for V_2.*
- Verifier's first step (V1): *The verifier selects at random one of the two input graphs, and sends to the prover a random isomorphic copy of this graph. Namely, the verifier selects uniformly $\sigma \in \{1,2\}$, and a random permutation π from the set of permutations over the vertex set V_σ. The verifier constructs a graph with vertex set V_σ and edge set*

  $$E \stackrel{\rm def}{=} \{\{\pi(u),\pi(v)\} : \{u,v\} \in E_\sigma\}$$

  and sends (V_σ, E) to the prover.

- Motivating Remark: *If the input graphs are non-isomorphic, as the prover claims, then the prover should be able to distinguish (not necessarily by an efficient algorithm) isomorphic copies of one graph from isomorphic copies of the other graph. However, if the input graphs are isomorphic, then a random isomorphic copy of one graph is distributed identically to a random isomorphic copy of the other graph.*

- Prover's step: *Upon receiving a graph, $G' = (V', E')$, from the verifier, the prover finds a $\tau \in \{1,2\}$ such that the graph G' is isomorphic to the input graph G_τ. (If both $\tau = 1,2$ satisfy the condition then τ is selected arbitrarily. In case no $\tau \in \{1,2\}$ satisfies the condition, τ is set to 0). The prover sends τ to the verifier.*

- Verifier's second step (V2): *If the message, τ, received from the prover equals σ (chosen in Step V1) then the verifier outputs 1 (i.e., accepts the common input). Otherwise the verifier outputs 0 (i.e., rejects the common input).*

The verifier's strategy in Construction 9.3 is easily implemented in probabilistic polynomial-time. We do not known of a probabilistic polynomial-time implementation of the prover's strategy, but this is not required. The motivating remark justifies the claim that Construction 9.3 constitutes an interactive proof system for the set of pairs of non-isomorphic graphs.[4] Recall that the latter is a co$\mathcal{NP}$-set (which is not known to be in $\mathcal{NP}$).

9.1.2.2 The full power of interactive proofs

The interactive proof system of Construction 9.3 refers to a specific coNP-set that is not known to be in $\mathcal{NP}$. It turns out that interactive proof systems are powerful enough to prove membership in *any* coNP-set (e.g., prove that a graph is not 3-colorable). Thus, assuming that $\mathcal{NP} \neq$ co$\mathcal{NP}$, this establishes that interactive proof systems are more powerful than NP-proof systems. Furthermore, the class of sets having interactive proof systems coincides with the class of sets that can be decided using a polynomial amount of work-space.

Theorem 9.4 (The IP Theorem): $\mathcal{IP} = \mathcal{PSPACE}$.

Recall that it is widely believed that $\mathcal{NP}$ is a *proper* subset of $\mathcal{PSPACE}$. Thus, under this conjecture, interactive proofs are more powerful than NP-proofs.

Sketch of the Proof of Theorem 9.4

Theorem 9.4, was established using algebraic methods (see details below). In particular, the following approach – unprecedented in complexity theory – was employed:

[4] In case G_1 is not isomorphic to G_2, no graph can be isomorphic to both G_1 and G_2. On the other hand, if G_1 and G_2 are isomorphic then for every G' the number of isomorphisms between G_1 and G' equals the number of isomorphisms between G_2 and G'. It follows that in this case G' yields no information about the σ chosen by V, and so no prover may convince V with probability exceeding 1/2.

In order to demonstrate that a particular set is in a particular class, an arithmetic generalization of the Boolean problem is presented, and (elementary) algebraic methods are applied for showing that the arithmetic problem is solvable within the class. Following is a sketch of the proof. We first show that co$\mathcal{NP} \subseteq \mathcal{IP}$, by presenting an interactive proof system for the co$\mathcal{NP}$-complete set of non-satisfiable CNF formulae. Next we extend this proof system to obtain one for the $\mathcal{PSPACE}$-complete set of non-satisfiable Quantified Boolean Formulae. Finally, we observe that $\mathcal{IP} \subseteq \mathcal{PSPACE}$.

Teaching note: We present a sketch of the proof that focuses on the main ideas, and omit various implementation details (which can be found in [150, 191]). Furthermore, we devote most of the presentation to establishing that co$\mathcal{NP} \subseteq \mathcal{IP}$, and recommend doing the same in class.

Arithmetization of Boolean (CNF) formulae: Given a Boolean (CNF) formula, we replace the Boolean variables by integer variables, OR-clauses by sums, and the top level conjunction by a product. Then we sum over all 0-1 assignments to these variables. For example, the Boolean formula

$$(x_3 \vee \neg x_5 \vee x_{17}) \wedge (x_5 \vee x_9) \wedge (\neg x_3 \vee \neg x_4)$$

is replaces by the arithmetic expression

$$(x_3 + (1 - x_5) + x_{17}) \cdot (x_5 + x_9) \cdot ((1 - x_3) + (1 - x_4))$$

and the Boolean formula is non-satisfiable if and only if the sum of the arithmetic expression, taken over all choices of $x_1, x_2, ..., x_{17} \in \{0, 1\}$, equals 0. Observe that the arithmetic expression is a low degree polynomial over the integers (i.e., its degree is upper-bounded by the number of clauses). Also observe that, for any Boolean formula, the value of the corresponding arithmetic expression (for any choice of $x_1, ..., x_n \in \{0, 1\}$) resides within the interval $[0, v^m]$, where v is the maximum number of variables in a clause, and m is the number of clauses. Summing over all 2^n possible 0-1 assignments, where $n < vm$ is the number of variables, the result resides in $[0, 2^n v^m]$.

Moving to a Finite Field: Whenever we check equality between two integers in $[0, M]$, it suffices to check equality mod q, where $q > M$. The benefit is that the arithmetic is now in a finite field (mod q) and so certain things are "nicer" (e.g., uniformly selecting a value). Thus, proving that a CNF formula is not satisfiable reduces to proving equality of the following form

$$\sum_{x_1=0,1} \cdots \sum_{x_n=0,1} \phi(x_1, ..., x_n) \equiv 0 \pmod{q}, \tag{9.1}$$

where ϕ is a low degree multi-variate polynomial.

The actual protocol: stripping summations in iterations. We strip off summations in iterations. In each iteration the prover is supposed to supply the univariate polynomial representing the expression in one (currently stripped) variable. (By the foregoing observation, this is a low degree polynomial and so has a short description.) The verifier checks that the polynomial (say, p) is of low degree, and that it corresponds to the current value (say, v) being claimed (i.e., $p(0)+p(1) \equiv v$). Next, the verifier randomly instantiates the variable (i.e., selects uniformly $r \in \mathrm{GF}(q)$)[5], yielding a new value to be claimed for the resulting expression (i.e., the verifier computes $v \leftarrow p(r)$, and expects a proof that the residual expression equals v). The verifier sends the uniformly chosen instantiation (i.e., r) to the prover, and the proceed to the next iteration (which refers to the residual expression and to the value v). At the end of the last iteration, the verifier has a closed form expression (i.e., an expression without formal summations), which can be easily checked against the claimed value.

A single iteration (revisited): The i^{th} iteration is aimed at proving a claim of the form

$$\sum_{x_i=0,1} \cdots \sum_{x_n=0,1} \phi(r_1,...,r_{i-1},x_i,x_{i+1},...,x_n) \equiv v_{i-1} \pmod{q}, \tag{9.2}$$

where $v_0 = 0$, and $r_1,...,r_{i-1}$ and v_{i-1} are as determined in previous iterations. The i^{th} iteration consists of two steps (messages): a prover step followed by a verifier step. The prover is supposed to provide the verifier with the univariate polynomial p_i that satisfies

$$p_i(z) \stackrel{\text{def}}{=} \sum_{x_{i+1}=0,1} \cdots \sum_{x_n=0,1} \phi(r_1,...,r_{i-1},z,x_{i+1},...,x_n) \bmod q\,. \tag{9.3}$$

Denote by p_i' the actual polynomial sent by the prover (i.e., the honest prover sets $p_i' = p_i$). Then, the verifier first checks if $p_i'(0)+p_i'(1) \equiv v_{i-1} \pmod{q}$, and next uniformly selects $r_i \in \mathrm{GF}(q)$ and sends it to the prover. Needless to say, the verifier will reject if the first check is violated. The claim to be proven in the next iteration is

$$\sum_{x_{i+1}=0,1} \cdots \sum_{x_n=0,1} \phi(r_1,...,r_{i-1},r_i,x_{i+1},...,x_n) \equiv v_i \pmod{q}, \tag{9.4}$$

where $v_i \stackrel{\text{def}}{=} p_i'(r_i) \bmod q$.

Completeness of the protocol: When the initial claim (i.e., Eq. (9.1)) holds, the prover can supply the correct polynomials (as determined in Eq. (9.3)), and this will lead the verifier to always accept.

[5]Here and elsewhere, we denote by $\mathrm{GF}(q)$ the finite field having q elements.

Soundness of the protocol: It suffices to upper-bound the probability that, for a particular iteration, the entry claim (i.e., Eq. (9.2)) is false while the ending claim (i.e., Eq. (9.4)) is valid. Both claims refer to the current summation expression being equal to the current value, where 'current' means either at the beginning of the iteration or at its end. Let $p(\cdot)$ be the actual polynomial representing the expression when stripping the current variable, and let $p'(\cdot)$ be any potential answer by the prover. We may assume that $p'(0) + p'(1) \equiv v \pmod q$ and that p' is of low-degree (as otherwise the verifier will reject). Using our hypothesis (that the entry claim of Eq. (9.2) is false), we know that $p(0) + p(1) \not\equiv v \pmod q$. Thus, p' and p are different low-degree polynomials, and so they may agree on very few points (if at all). In case the verifier instantiation (i.e., its choice of random r) does not happen to be one of these few points, the ending claim (i.e., Eq. (9.4)) is false too. Details are left as an exercise (see Exercise 9.2).

This establishes that the set of non-satisfiable CNF formulae has an interactive proof system. Actually, the same proof system can be used to prove that a given formula has a given number of satisfying assignment; i.e., prove membership in the ("counting") set

$$\{(\phi, k) : |\{\tau : \phi(\tau) = 1\}| = k\}.$$

Using adequate reductions, it follows that every problem in $\#\mathcal{P}$ has an interactive proof system (i.e., for every $R \in \mathcal{PC}$, the set $\{(x, k) : |\{y : (x, y) \in R\}| = k\}$ is in $\mathcal{IP}$). Proving that $\mathcal{PSPACE} \subseteq \mathcal{IP}$ requires a little more work.

Interactive Proofs for PSPACE (basic idea). We present an interactive proof for the set of satisfied Quantified Boolean Formulae (QBF), which is complete for $\mathcal{PSPACE}$ (see Theorem 5.15). Recall that the number of quantifiers in such formulae is unbounded (e.g., it may be polynomially related to the length of the input), that there are both existential and universal quantifiers, and furthermore these quantifiers may alternate. In the arithmetization of these formulae, we replace existential quantifiers by summations and universal quantifiers by products. Two difficulties arise when considering the application of the forgoing protocol to the resulting arithmetic expression. Firstly, the value of the expression (which may involve a big number of nested formal products) is only upper-bounded by a double exponential function (in the length of the input). Secondly, when stripping a summation (or a product), the expression may be a polynomial of high degree (due to nested formal products that may appear in the remaining expression). For example, both phenomena occur in the following expression

$$\sum_{x=0,1} \prod_{y_1=0,1} \cdots \prod_{y_n=0,1} (x + y_n),$$

which equals $\sum_{x=0,1} x^{2^{n-1}} \cdot (1+x)^{2^{n-1}}$. The first difficulty is easy to resolve by using the Chinese Reminder Theorem (i.e., if two integers in $[0, M]$ are different then they must be different modulo most of the primes in the interval $[3, \mathrm{poly}(\log M)]$). Thus, we let the verifier selects a random prime q of length that is linear in the length of the original formula, and the two parties consider the arithmetic expression reduced modulo this q. The second difficulty is resolved by noting that $\mathcal{PSPACE}$ is actually reducible to a special form of QBF in which no variable appears both to the left and to the right of more than one universal quantifier (see the proof of Theorem 5.15 or alternatively Exercise 9.4). It follows that when arithmetizing and stripping summations (or products) from the resulting arithmetic expression, the corresponding univariate polynomial is of low degree (i.e., at most twice the length of the original formula, where the factor of two is due to the single universal quantifier that has this variable quantified on its left and appearing on its right).

IP is contained in PSPACE: We show that, for every interactive proof system, there exists an *optimal prover strategy* that can be implemented in polynomial-space, where an optimal prover strategy is one that maximizes the probability that the prescribed verifier accepts the common input. It follows that $\mathcal{IP} \subseteq \mathcal{PSPACE}$, because (for every $S \in \mathcal{IP}$) we can emulate the interaction of the prescribed verifier with an optimal prover strategy in polynomial space.

Proposition 9.5 *Let V be a probabilistic polynomial-time interactive machine. Then, there exists a polynomial-space computable prover strategy f that, for every x maximizes the probability that V accepts x. That is, for every P^* and every x it holds that the probability that V accepts x after interacting with P^* is upper-bounded by the probability that V accepts x after interacting with f.*

Proof Sketch: For every common input x and any possible partial transcript γ of the interaction so far, the strategy f determines an optimal next message for the prover by considering all possible coin tosses of the verifier that are consistent with (x, γ). Specifically, f is determined recursively such that $f(x, \gamma) = m$ if m maximizes the number of verifier coins that are consistent with (x, γ) and lead the verifier to accept when subsequent prover moves are determined by f (which is where recursion is used). That is, coins r support the setting $f(x, \gamma) = m$, where $\gamma = (\alpha_1, \beta_1, ..., \alpha_t, \beta_t)$, if the following two conditions hold:

1. r is consistent with (x, γ), which means that $\beta_i = V(x, r, \alpha_1, ..., \alpha_i)$ for $i = 1, ..., t$.
2. r leads V to accept (when subsequent prover moves are determined by f), which means that $V(x, r, \alpha_1, ..., \alpha_t, m, \alpha_{t+2}, ..., \alpha_T) = 1$, where $\alpha_{i+1} = f(x, \gamma, m, \beta_{t+1}, ..., \alpha_i \beta_i)$ and $\beta_i = V(x, r, \alpha_1, ..., \alpha_t, m, \alpha_{t+2}, ..., \alpha_i)$ for $i = t+1, ..., T-1$.

That is, $f(x, \gamma) = m$ if m maximizes the value of $\mathsf{E}[f(x, \gamma, m, V(x, R_\gamma, m))]$, where R_γ is selected uniformly among the r's that are consistent with (x, γ). Thus, $f(x, \gamma)$ can be computed in polynomial-space when given oracle access to $f(x, \gamma, \cdot, \cdot)$, and the proposition follows by standard composition of space-bounded computations.
□

9.1.3 Variants and finer structure: an overview

In this subsection we consider several variants on the basic definition of interactive proofs as well as finer complexity measures. This is an advanced subsection, which only provides an overview of the various notions and results (as well as pointers to proofs of the latter).

9.1.3.1 Arthur-Merlin games a.k.a public-coin proof systems

The verifier's messages in a general interactive proof system are determined arbitrarily (but efficiently) based on its view of the interaction so far (which includes its internal coin tosses, which without loss of generality can take place at the onset of the interaction). In public-coin proof systems (a.k.a Arthur-Merlin proof systems), the verifier's messages are the outcome of any coin it tosses at the current round. Assuming, without loss of generality, that the verifier tosses the same number ℓ of coins in each round, we may assume that the verifier's message in each round consists of the outcome of ℓ coin tosses (because any other string that the verifier may compute is determined by the coins tossed so far). Note that the proof systems presented in the proof of Theorem 9.4 are of the public-coin type, whereas this is not the case for the Graph Non-Isomorphism proof system (of Construction 9.3). Thus, although not all natural proof systems are of the public-coin type, every set having an interactive proof system also has a public-coin interactive proof system. This means that, *in the context of interactive proof systems, asking random questions is as powerful as asking clever questions.*

Indeed, public-coin proof systems are a syntactically restricted type of interactive proof systems. This restriction may make the design of such systems more complex, but potentially facilitates their analysis (and especially the analysis of a generic system). Another advantage of public-coin proof systems is that the verifier's actions (except for its final decision) are oblivious of the prover's messages. This property is used in the proof of Theorem 9.12.

9.1.3.2 Interactive proof systems with two-sided error

In Definition 9.1 error probability is allowed in the soundness condition but not in the completeness condition. In such a case, we say that the proof system has perfect completeness (or one-sided error probability). A more general definition allows an error probability (upper-bounded by, say, 1/3) in both the completeness and soundness conditions. Note that sets having such generalized (two-sided error) interactive proofs are also in $\mathcal{PSPACE}$, and thus allowing two-sided error does not increase the power of interactive proofs. See further discussion at the end of §9.1.3.3.

9.1.3.3 A hierarchy of interactive proof systems

Definition 9.1 only refers to the total computation time of the verifier, and thus allows an arbitrary (polynomial) number of messages to be exchanged. A finer definition refers to the number of messages being exchanged (also called the number of rounds).[6]

[6]An even finer structure emerges when considering also the total length of the messages sent by the prover (see [102]).

Definition 9.6 (The round-complexity of interactive proof):

- *For an integer function m, the complexity class $\mathcal{IP}(m)$ consists of sets having an interactive proof system in which, on common input x, at most $m(|x|)$ messages are exchanged between the parties.*[7]

- *For a set of integer functions, M, we let $\mathcal{IP}(M) \stackrel{\text{def}}{=} \bigcup_{m \in M} \mathcal{IP}(m)$. Thus, $\mathcal{IP} = \mathcal{IP}(\texttt{poly})$.*

For example, interactive proof systems in which the verifier sends a single message that is answered by a single message of the prover corresponds to $\mathcal{IP}(2)$. Clearly, $\mathcal{NP} \subseteq \mathcal{IP}(1)$, yet the inclusion may be strict because in $\mathcal{IP}(1)$ the verifier may toss coins after receiving the prover's single message. (Also note that $\mathcal{IP}(0) = \text{co}\mathcal{RP}$.) Concerning the finer structure of the IP-hierarchy, the following is known:

- A linear speed-up (see Section F.2 (or [20] and [107])): For every integer function, f, such that $f(n) \geq 2$ for all n, the class $\mathcal{IP}(O(f(\cdot)))$ collapses to the class $\mathcal{IP}(f(\cdot))$. In particular, $\mathcal{IP}(O(1))$ collapses to $\mathcal{IP}(2)$.

- The class $\mathcal{IP}(2)$ contains sets not known to be in $\mathcal{NP}$; e.g., Graph Non-Isomorphism (see Construction 9.3). However, under plausible intractability assumptions, $\mathcal{IP}(2) = \mathcal{NP}$ (see [155]).

- If $\text{co}\mathcal{NP} \subseteq \mathcal{IP}(2)$ then the Polynomial-Time Hierarchy collapses (see [42]).

It is conjectured that $\text{co}\mathcal{NP}$ is *not* contained in $\mathcal{IP}(2)$, and consequently that interactive proofs with an unbounded number of message exchanges are more powerful than interactive proofs in which only a bounded (i.e., constant) number of messages are exchanged.[8] The class $\mathcal{IP}(1)$ (also denoted $\mathcal{MA}$) seems to be *the* "real" randomized (and yet non-interactive) version of $\mathcal{NP}$: Here the prover supplies a candidate (polynomial-size) "proof", and the verifier assesses its validity probabilistically (rather than deterministically).

The IP-hierarchy (i.e., $\mathcal{IP}(\cdot)$) equals an analogous hierarchy, denoted $\mathcal{AM}(\cdot)$, that refers to public-coin (a.k.a Arthur-Merlin) interactive proofs. That is, for every integer function f, it holds that $\mathcal{AM}(f) = \mathcal{IP}(f)$. For $f \geq 2$, it is also the case that $\mathcal{AM}(f) = \mathcal{AM}(O(f))$; actually, the aforementioned linear speed-up for $\mathcal{IP}(\cdot)$ is established by combining the following two results:

1. Emulating $\mathcal{IP}(\cdot)$ by $\mathcal{AM}(\cdot)$ (see §F.2.1 or [107]): $\mathcal{IP}(f) \subseteq \mathcal{AM}(f+3)$.

2. Linear speed-up for $\mathcal{AM}(\cdot)$ (see §F.2.2 or [20]): $\mathcal{AM}(2f) \subseteq \mathcal{AM}(f)$.

In particular, $\mathcal{IP}(O(1)) = \mathcal{AM}(2)$, even if $\mathcal{AM}(2)$ is restricted such that the verifier tosses no coins after receiving the prover's message. (Note that $\mathcal{IP}(1) = \mathcal{AM}(1)$ and $\mathcal{IP}(0) = \mathcal{AM}(0)$ are trivial.) We comment that it is common to denote $\mathcal{AM}(2)$

[7] We count the total number of messages exchanged regardless of the direction of communication.

[8] Note that the linear speed-up cannot be applied for an unbounded number of times, because each application may increase (e.g., square) the time-complexity of verification.

by $\mathcal{AM}$, which is indeed inconsistent with the convention of using $\mathcal{IP}$ to denote $\mathcal{IP}(\texttt{poly})$.

The fact that $\mathcal{IP}(O(f)) = \mathcal{IP}(f)$ is proved by establishing an analogous result for $\mathcal{AM}(\cdot)$ demonstrates the advantage of the public-coin setting for the study of interactive proofs. A similar phenomenon occurs when establishing that the IP-hierarchy equals an analogous two-sided error hierarchy (see Exercise 9.5).

9.1.3.4 Something completely different

We stress that although we have relaxed the requirements from the verification procedure, by allowing it to interact with the prover, toss coins, and risk some (bounded) error probability, we did not restrict the validity of its assertions by assumptions concerning the potential prover. This should be contrasted with other notions of proof systems, such as computationally-sound ones (see §9.1.4.2), in which the validity of the verifier's assertions depends on assumptions concerning the prover(s).

9.1.4 On computationally bounded provers: an overview

Recall that our definition of interactive proofs (i.e., Definition 9.1) makes no reference to the computational abilities of the potential prover. This fact has two conflicting consequences:

1. The completeness condition does not provide any upper bound on the complexity of the corresponding proving strategy (which convinces the verifier to accept valid assertions).

2. The soundness condition guarantees that, regardless of the computational effort spend by a cheating prover, the verifier cannot be fooled to accept invalid assertions (with probability exceeding the soundness error).

Note that providing an upper-bound on the complexity of the (presecribed) prover strategy P of a specific interactive proof system (P, V) only strengthens the claim that (P, V) is a proof system for the corresponding set (of valid assertions). We stress that the presecribed prover strategy is referred to only in the completeness condition (and is irrelevant to the soundness condition). On the other hand, relaxing the definition of interactive proofs such that soundness holds only for a specific class of cheating prover strategies (rather than for all cheating prover strategies) weakens the corresponding claim. In this advanced section we consider both possibilities.

Teaching note: Indeed, this is an advanced subsection, which is best left for independent reading. It merely provides an overview of the various notions, and the reader is directed to the chapter's notes for further detail (i.e., pointers to the relevant literature).

9.1.4.1 How powerful should the prover be?

Assume that a set S is in $\mathcal{IP}$. This means that there is a verifier V that can be convinced to accept any input in S but cannot be fooled to accept any input not in S (except with small probability). One may ask how powerful should a prover be such that it can convince the verifier V to accept any input in S. Note that Proposition 9.5 asserts that an optimal prover strategy can be implemented in polynomial-space (and that we cannot expect better for a generic set in $\mathcal{PSPACE} = \mathcal{IP}$), but we will seek better upper-bounds on the complexity of the prover that convinces a specific verifier (which in turn corresponds to a specific set S). More interestingly, considering all possible verifiers that give rise to interactive proof systems for S, we ask what is the minimum power required from a prover that satisfies the completeness requirement with respect to one of these verifiers?

We stress that, unlike the case of computationally-sound proof systems (see §9.1.4.2), we do not restrict the power of the prover in the soundness condition, but rather consider the minimum complexity of provers meeting the completeness condition. Specifically, we are interested in *relatively efficient* provers that meet the completeness condition. The term "relatively efficient prover" has been given three different interpretations.

1. A prover is considered *relatively efficient* if, when given an auxiliary input (in addition to the common input in S), it works in (probabilistic) polynomial-time. Specifically, in case $S \in \mathcal{NP}$, the auxiliary input maybe an NP-proof that the common input is in the set. Still, even in this case the interactive proof need not consist of the prover sending the auxiliary input to the verifier; for example, an alternative procedure may allow the prover to be zero-knowledge (see Construction 9.10).

   This interpretation is adequate and in fact crucial for applications in which such an auxiliary input is available to the otherwise polynomial-time parties. Typically, such auxiliary input is available in cryptographic applications in which parties wish to prove in (zero-knowledge) that they have correctly conducted some computation. In these cases the NP-proof is just the transcript of the computation by which the claimed result has been generated, and thus the auxiliary input is available to the proving party.

2. A prover is considered *relatively efficient* if it can be implemented by a probabilistic polynomial-time oracle machine with oracle access to the set S itself. (Note that the prover in Construction 9.3 has this property.)

   This interpretation generalizes the notion of self-reducibility of NP-sets. (Recall that by self-reducibility of an NP-set we mean that the search problem of finding an NP-witness is polynomial-time reducible to deciding membership in the set (cf. Definition 2.13).)

3. A prover is considered *relatively efficient* if it can be implemented by a probabilistic machine that runs in time that is polynomial in the deterministic complexity of the set. This interpretation relates the difficulty of convincing a "lazy verifier" to the complexity of finding the truth alone.

> Hence, in contrast to the first interpretation, which is adequate in settings where assertions are generated along with their NP-proofs, the current interpretation is adequate in settings in which the prover is given only the assertion and has to find a proof to it by itself (before trying to convince a lazy verifier of its validity).

9.1.4.2 Computational-soundness

Relaxing the soundness condition such that it only refers to relatively-efficient ways of trying to fool the verifier (rather than to all possible ways) yields a fundamentally different notion of a proof system. Assertions proven in such a system are not necessarily correct; they are correct only if the potential cheating prover does not exceed the presumed complexity limits. As in §9.1.4.1, the notion of "relative efficiency" can be given different interpretations, the most popular one being that the cheating prover strategy can be described by a (non-uniform) family of polynomial-size circuits. The latter interpretation coincides with the first interpretation used in §9.1.4.1 (i.e., a probabilistic polynomial-time strategy that is given an auxiliary input (of polynomial length)). Specifically, the soundness condition is replaced by the following computational soundness condition that asserts that it is infeasible to fool the verifier into accepting false statements:

> For every prover strategy that is implementable by a family of polynomial-size circuits $\{C_n\}$, and every sufficiently long $x \in \{0,1\}^* \setminus S$, the probability that V accepts x when interacting with $C_{|x|}$ is less than $1/2$.

As in case of standard soundness, the computational-soundness error can be reduced by repetitions. We warn, however, that unlike in the case of standard soundness (where both sequential and parallel repetitions will do), the computational-soundness error cannot always be reduced by parallel repetitions.

It is common and natural to consider proof systems in which the prover strategies considered both in the completeness and soundness conditions satisfy the same notion of relative efficiency. Protocols that satisfy these conditions with respect to the aforementioned interpretation are called arguments. We mention that argument systems may be more efficient (e.g., in terms of their communication complexity) than interactive proof systems.

9.2 Zero-Knowledge Proof Systems

Zero-Knowledge proofs are fascinating and extremely useful constructs. Their fascinating nature is due to their seemingly contradictory definition: zero-knowledge proofs are both convincing and yet yield nothing beyond the validity of the assertion being proven. Their applicability in the domain of cryptography is vast; they are typically used to force malicious parties to behave according to a predetermined protocol. In addition to their direct applicability in Cryptography, zero-knowledge proofs serve as a good bench-mark for the study of various problems regarding cryptographic protocols. In this section we focus on the conceptual contents of zero-knowledge, and relegate their cryptographic applications to Appendix C.

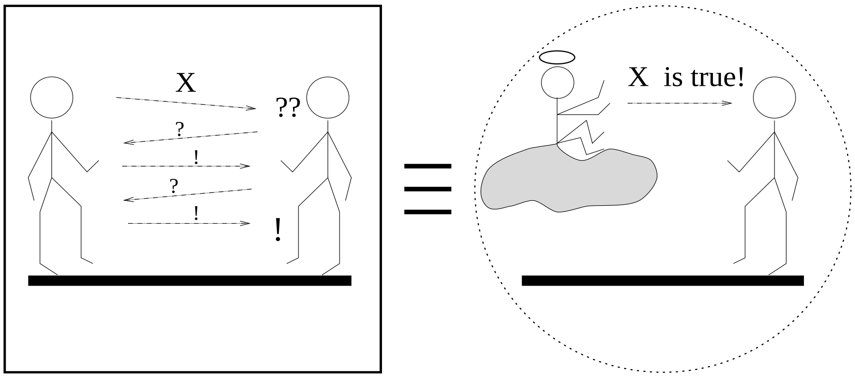


Figure 9.1: Zero-knowledge proofs – an illustration.

Turning back to the conceptual angle, we highlight the fact that standard proofs are believed to yield knowledge and not merely establish the validity of the assertion being proven. Indeed, it is commonly believed that (good) proofs provide a deeper understanding of the theorem being proved. At the technical level, an NP-proof of membership in some set $S \in \mathcal{NP} \setminus \mathcal{P}$ yields something (i.e., the NP-proof itself) that is typically hard to compute (even when assuming that the input is in S). For example, a 3-coloring of a graph is an NP-proof that the graph is 3-colorable, but it yields information (i.e., the coloring) that is infeasible to compute (when given an arbitrary 3-colorable graph). In contrast to such NP-proofs, which seem to yield a lot of knowledge, zero-knowledge proofs yield no knowledge at all; that is, the latter exhibit an extreme contrast between being convincing (of the validity of a statement) and teaching anything on top of the validity of the statement.

Teaching note: We believe that the treatment of zero-knowledge proofs provided in this section suffices for the purpose of a course in complexity theory. For an extensive treatment of (computational) zero-knowledge proofs, the interested reader is referred to [87, Chap. 4].

9.2.1 Definitional Issues

Loosely speaking, zero-knowledge proofs are proofs that yield nothing beyond the validity of the assertion; that is, a verifier obtaining such a proof only gains conviction in the validity of the assertion. This is formulated by saying that anything that can be feasibly obtained from a zero-knowledge proof is also feasibly computable from the (valid) assertion itself. The latter formulation follows the simulation paradigm, which is discussed next.

9.2.1.1 A wider perspective: the simulation paradigm

In defining zero-knowledge proofs, we view the verifier as a potential adversary that tries to gain knowledge from the (prescribed) prover.[9] We wish to state that no (feasible) adversary strategy for the verifier can gain anything from the prover (beyond conviction in the validity of the assertion). Let us consider the desired formulation from a wide perspective.

A key question regarding the modeling of security concerns is how to express the intuitive requirement that an adversary "gains nothing substantial" by deviating from the prescribed behavior of an honest user. Our approach is that the adversary *gains nothing* if whatever it can obtain by unrestricted adversarial behavior can be obtained within essentially the same computational effort by a benign behavior. The definition of the "benign behavior" captures what we want to achieve in terms of security, and is specific to the security concern to be addressed. For example, in the previous paragraph, we said that a proof is zero-knowledge if it yields nothing beyond the validity of the assertion (i.e., the benign behavior is any computation that is based (only) on the assertion itself, while assuming that the latter is valid). Thus, in a zero-knowledge proof no feasible adversarial strategy for the verifier can obtain more than a "benign verifier" (which believes the assertion) can obtain from the assertion itself. We comment that the simulation paradigm is pivotal to many definitions in cryptography (e.g., it underlies the definition of security of encryption schemes and cryptographic protocols); for further details see Appendix C.

9.2.1.2 The basic definitions

Zero-knowledge is a property of some prover strategies. More generally, zero-knowledge is a property of some interactive machines. Fixing an interactive machine (e.g., a prescribed prover), we consider what can be gained (i.e., computed) by an *arbitrary feasible adversary* (e.g., a verifier) *that interacts with the aforementioned fixed machine* on a common input taken from a predetermined set (in our case the set of valid assertions). This gain is compared against what can be computed by an *arbitrary feasible algorithm* (called a simulator) that is only given the input itself. The fixed machine is zero-knowledge if the "computational power" of these two (fundamentally different settings) is essentially equivalent. Details follow.

The formulation of the zero-knowledge condition considers two ensembles of probability distributions, each ensemble associates a probability distribution to each valid assertion. Specifically, in the case of interactive proofs, the first ensemble represents the output distribution of the verifier after interacting with the specified prover strategy P, where the verifier is employing an arbitrary efficient strategy (not necessarily the specified one). The second ensemble represents the output distribution of some probabilistic polynomial-time algorithm (which does not interact with anyone). The basic paradigm of zero-knowledge asserts that for every ensemble of the first type there exist a "similar" ensemble of the second type.

[9] Recall that when defining a proof system (e.g., an interactive proof system), we view the prover as a potential adversary that tries to fool the (prescribed) verifier (into accepting invalid assertions).

The specific variants differ by the interpretation given to the notion of *similarity*. The most strict interpretation, leading to perfect zero-knowledge, is that similarity means equality.

Definition 9.7 (perfect zero-knowledge, over-simplified):[10] *A prover strategy, P, is said to be* perfect zero-knowledge over a set S *if for every probabilistic polynomial-time verifier strategy, V^*, there exists a probabilistic polynomial-time algorithm, M^*, such that*

$$(P, V^*)(x) \equiv M^*(x)\,, \qquad \textit{for every } x \in S$$

where $(P, V^*)(x)$ is a random variable representing the output of verifier V^* after interacting with the prover P on common input x, and $M^*(x)$ is a random variable representing the output of machine M^* on input x.

We comment that any set in co$\mathcal{RP}$ has a perfect zero-knowledge proof system in which the prover keeps silence and the verifier decides by itself. The same holds for $\mathcal{BPP}$ provided that we relax the definition of interactive proof system to allow two-sided error. Needless to say, our focus is on non-trivial proof systems; that is, proof systems for sets outside of $\mathcal{BPP}$.

A somewhat more relaxed interpretation (of the notion of similarity), leading to almost-perfect zero-knowledge (a.k.a statistical zero-knowledge), is that similarity means statistical closeness (i.e., negligible difference between the ensembles). The most liberal interpretation, leading to the standard usage of the term zero-knowledge (and sometimes referred to as computational zero-knowledge), is that similarity means computational indistinguishability (i.e., failure of any efficient procedure to tell the two ensembles apart). Combining the foregoing discussion with the relevant definition of computational indistinguishability (i.e., Definition C.5), we obtain the following definition.

Definition 9.8 (zero-knowledge, somewhat simplified): *A prover strategy, P, is said to be* zero-knowledge over a set S *if for every probabilistic polynomial-time verifier strategy, V^*, there exists a probabilistic polynomial-time* simulator, M^*, *such that for every probabilistic polynomial-time* distinguisher, D, *it holds that*

$$d(n) \stackrel{\text{def}}{=} \max_{x \in S \cap \{0,1\}^n} \{|\Pr[D(x, (P, V^*)(x)) = 1] - \Pr[D(x, M^*(x)) = 1]|\}$$

is a negligible function.[11] *We denote by $\mathcal{ZK}$ the class of sets having zero-knowledge interactive proof systems.*

Definition 9.8 is a simplified version of the actual definition. Specifically, in order to guarantee that zero-knowledge is preserved under sequential composition it is necessary to slightly augment the definition (by providing V^* and M^* with the same arbitrary auxiliary input of poly($|x|$) length). For details see Section C.4.2. Other definitional issues and related notions are briefly discussed in Section C.4.4.

[10] In the actual definition one either allows M^* to run for *expected* polynomial-time or allows M^* to have no output with probability at most 1/2. The latter alternative implies the former, but the converse is not known to hold.

[11] That is, d vanishes faster that the reciprocal of any positive polynomial (i.e., for every positive polynomial p and for sufficiently large n, it holds that $d(n) < 1/p(n)$).

On the role of randomness and interaction. It can be shown that only sets in $\mathcal{BPP}$ have zero-knowledge proofs in which the verifier is deterministic. The same holds for deterministic provers, provided we consider "auxiliary-input" zero-knowledge (as in Definition C.9). It can also be shown that only sets in $\mathcal{BPP}$ have zero-knowledge proofs in which a single message is sent. Thus, both randomness and interaction are essential to the non-triviality of zero-knowledge proof systems. (For details, see [87, Sec. 4.5.1].)

Advanced Comment: Knowledge Complexity. Zero-knowledge is the lowest level of a knowledge-complexity hierarchy that quantifies the "knowledge revealed in an interaction." Specifically, the knowledge complexity of an interactive proof system may be defined as the minimum number of oracle-queries required in order to efficiently simulate an interaction with the prover. (See [86, Sec. 2.3.1] for references.)

9.2.2 The Power of Zero-Knowledge

When faced with a definition as complex (and seemingly self-contradictory) as the definition of zero-knowledge, one should indeed wonder whether the definition can be met (in a non-trivial manner).[12] It turns out that the existence of non-trivial zero-knowledge proofs is related to the existence of intractable problems in $\mathcal{NP}$. In particular, we will show that if one-way functions exist then every NP-set has a zero-knowledge proof system. (For the converse, see [87, Sec. 4.5.2] or [212].) We first demonstrate the scope of zero-knowledge by a presenting a simple (perfect) zero-knowledge proof system for a specific NP-set that is not known to be in $\mathcal{BPP}$. In this case we make no intractability assumptions, but the result is significant only if $\mathcal{NP}$ is not contained in $\mathcal{BPP}$.

9.2.2.1 A simple example

> *A story not found in the Odyssey refers to the not so famous Labyrinth of the Island of Aeaea. The Sorceress Circe, daughter of Helius, challenged godlike Odysseus to traverse the Labyrinth from its North Gate to its South Gate. Canny Odysseus doubted whether such a path existed at all and asked beautiful Circe for a proof, to which she replied that if she showed him a path this would trivialize for him the challenge of traversing the Labyrinth. "Not necessarily," clever Odysseus replied, "you can use your magic to transport me to a random place in the labyrinth, and then guide me by a random walk to a gate of my choice. If we repeat this enough times then I'll be convinced that there is a labyrinth-path between the two gates, while you will not reveal to me such a path." "Indeed," wise Circe thought to herself, "showing this mortal a random path from a random location in the labyrinth to*

[12] Note that any set in $\mathcal{BPP}$ has a trivial zero-knowledge (two-sided) proof system in which the verifier just determines membership by itself.

> *the gate he chooses will not teach him more than his taking a random walk from that gate."*

The foregoing story illustrates the main idea underlying the zero-knowledge proof for Graph Isomorphism presented next. Recall that the set of pairs of isomorphic graphs is not known to be in $\mathcal{BPP}$, and thus the straightforward NP-proof system (in which the prover just suplies the isomorphism) may not be zero-knowledge. Furthermore, assuming that Graph Isomorphism is not in $\mathcal{BPP}$, this set has no zero-knowledge NP-proofs of membership, but as we shall see it has zero-knowledge interactive proofs.

Construction 9.9 (zero-knowledge proof for Graph Isomorphism):

- Common Input: *A pair of graphs,* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$. *Let* ϕ *be an isomorphism between the input graphs; namely,* ϕ *is a 1-1 and onto mapping of the vertex set* V_1 *to the vertex set* V_2 *such that* $\{u, v\} \in E_1$ *if and only if* $\{\phi(v), \phi(u)\} \in E_2$.

- Prover's first Step (P1): *The prover selects a random isomorphic copy of* G_2, *and sends it to the verifier. Namely, the prover selects at random, with uniform probability distribution, a permutation* π *from the set of permutations over the vertex set* V_2, *and constructs a graph with vertex set* V_2 *and edge set*

$$E \stackrel{\text{def}}{=} \{\{\pi(u), \pi(v)\} : \{u, v\} \in E_2\}.$$

  The prover sends (V_2, E) *to the verifier.*

- Motivating Remark: *If the input graphs are isomorphic, as the prover claims, then the graph sent in Step P1 is isomorphic to both input graphs. However, if the input graphs are* not *isomorphic then no graph can be isomorphic to both of them.*

- Verifier's first Step (V1): *Upon receiving a graph,* $G' = (V', E')$, *from the prover, the verifier asks the prover to show an isomorphism between* G' *and one of the input graphs, chosen at random by the verifier. Namely, the verifier uniformly selects* $\sigma \in \{1, 2\}$, *and sends it to the prover (who is supposed to answer with an isomorphism between* G_σ *and* G'*).*

- Prover's second Step (P2): *If the message,* σ, *received from the verifier equals* 2 *then the prover sends* π *to the verifier. Otherwise (i.e.,* $\sigma \neq 2$*), the prover sends* $\pi \circ \phi$ *(i.e., the composition of* π *on* ϕ, *defined as* $\pi \circ \phi(v) \stackrel{\text{def}}{=} \pi(\phi(v))$*) to the verifier.*

  (Indeed, the prover treats any $\sigma \neq 2$ *as* $\sigma = 1$. *In the analysis we shall assume, without loss of generality, that* $\sigma \in \{1, 2\}$ *always holds.)*

- Verifier's second Step (V2): *If the message, denoted* ψ, *received from the prover is an isomorphism between* G_σ *and* G' *then the verifier outputs 1, otherwise it outputs 0.*

The verifier strategy in Construction 9.9 is easily implemented in probabilistic polynomial-time. In case the prover is given an isomorphism between the input graphs as auxiliary input, also the prover's program can be implemented in probabilistic polynomial-time. The motivating remark justifies the claim that Construction 9.9 constitutes an interactive proof system for the set of pairs of isomorphic graphs. As for the zero-knowledge property, consider first the special case in which the verifier actually follows the prescribed strategy (and selects σ at random, and in particular obliviously of the graph G' it receives). The view of this verifier can be easily simulated by selecting σ and ψ at random, constructing G' as a random isomorphic copy of G_σ (via the isomorphism ψ), and outputting the triplet (G', σ, ψ). Indeed (even in this case), the simulator behaves differently from the prescribed prover (which selects G' as a random isomorphic copy of G_2, via the isomorphism π), but its output distribution is identical to the verifier's view in the real interaction. However, the forgoing description assumes that the verifier follows the prescribed strategy, while in general the verifier may select σ depending on the graph G'. Thus, a slightly more complicated simulation (described next) is required.

A general clarification may be in place. Recall that we wish to simulate the interaction of an arbitrary verifier strategy with the prescribed prover, and so the simulator must depend on the verifier strategy. Indeed, we shall describe the simulator while referring to such a generic verifier strategy. Formally this means that the simulator's program incorporates the aforementioned verifier strategy. (Actually, the following simulator uses the generic verifier strategy as a subroutine.)

Turning back to the specific protocol of Construction 9.9, the basic idea is that simulator tries to guess σ and can complete a simulation if its guess turns out to be correct. Specifically, the simulator selects $\tau \in \{1, 2\}$ uniformly (hoping that the verifier selects $\sigma = \tau$), and constructs G' by randomly permuting G_τ (and thus being able to present an isomorphism between G_τ and G'). Recall that the simulator is analyzed only on yes-instances (i.e., the input graphs G_1 and G_2 are isomorphic). The point is that if G_1 and G_2 are isomorphic, then the graph G' does not yield any information regarding the simulator's guess (i.e., τ), and thus this guess is correct with probability $1/2$. (Indeed, this is analogous to the analysis of the soundness of Construction 9.3.) If the guess is correct then the simulator can produce an output that has the correct distribution, otherwise the entire process is repeated.

Useful conventions. We wish to highlight three conventions that were either used (implicitly) in the foregoing analysis or can be used to simplify the description of (this and/or) other zero-knowledge simulators.

1. Without loss of generality, we may assume that the cheating verifier strategy is implemented by a *deterministic* polynomial-size circuit (or, equivalently, by a polynomial-time algorithm with an auxiliary input).[13]

[13] This observation is not crucial, but it does simplify the analysis (by eliminating the need to specify a sequence of coin tosses in each invocation of the verifier's strategy).

This is justified by fixing any outcome of the verifier's coins, and observing that our (uniform) simulation of the various (residual) deterministic strategies yields a simulation of the original probabilistic strategy.

2. Without loss of generality, it suffices to consider cheating verifiers that (only) output their view of the interaction (i.e., the common input, their internal coin tosses, and messages that they received). In other words, it suffices to simulate the view of that cheating verifiers have of the real interaction.

   This is justified by noting that the final output of any verifier can be obtained from its view of the interaction, where the complexity of the transformation is upper-bounded by the complexity of the verifier's strategy.

3. Without loss of generality, it suffices to construct a "weak simulator" that produces output with some noticeable[14] probability such that whenever an output is produced it is distributed "correctly" (i.e., similarly to the distribution occuring in real interactions with the prescribed prover).

   This is justified by repeatedly invoking such a weak simulator (polynomially) many times and using the first output produced by any of these invocations. Note that by using an adequate number of invocations, we fail to produce an output with negligible probability. (Alternatively, we always produce an output after an expected polynomial number of invocations.) Furthermore, note that a simulator that fails to produce output with negligible probability can be converted to a simulator that always produces an output, while incurring a negligible statistic deviation in the output distribution.

9.2.2.2 The full power of zero-knowledge proofs

The forgoing zero-knowledge proof system refers to one NP-set that is not known to be in $\mathcal{BPP}$. It turns out that, under reasonable assumptions, zero-knowledge can be used to prove membership in *any* NP-set. Intuitively, it suffices to establish this fact for a single NP-complete set, and thus we focus on presenting a zero-knowledge proof system for the set of 3-colorable graphs. Note that it is easy to prove that a given graph G is 3-colorable by just presenting a 3-coloring of G (and the same holds for membership in any set in $\mathcal{NP}$), but this NP-proof is not a zero-knowledge proof (unless $\mathcal{P} = \mathcal{NP}$). In fact, assuming $\mathcal{P} \neq \mathcal{NP}$, graph 3-colorability has no zero-knowledge NP-proofs, but as we shall see it has zero-knowledg interactive proofs. This interactive proof will be described while referring to "boxes" in which information can be hidden and later revealed. Such boxes can be implemented using one-way functions (see below).

Construction 9.10 (Zero-knowledge proof of 3-colorability, abstract description): *The description refers to abstract non-transparent boxes that can be perfectly locked and unlocked such that these boxes perfectly hide their contents while being locked.*

[14] Recall that a probability is called noticeable if it is greater than the reciprocal of some positive polynomial (in the relevant parameter).

- Common Input: *A simple graph* $G=(V,E)$.

- Prover's first step: *Let* ψ *be a 3-coloring of* G. *The prover selects a random permutation,* π, *over* $\{1,2,3\}$, *and sets* $\phi(v) \stackrel{\text{def}}{=} \pi(\psi(v))$, *for each* $v \in V$. *Hence, the prover forms a random relabeling of the 3-coloring* ψ. *The prover sends the verifier a sequence of* $|V|$ *locked and non-transparent boxes such that the* v^{th} *box contains the value* $\phi(v)$.

- Verifier's first step: *The verifier uniformly selects an edge* $\{u,v\} \in E$, *and sends it to the prover.*

- Motivating Remark: *The boxes are supposed to contain a 3-coloring of the graph, and the verifier asks to inspect the colors of vertices* u *and* v. *Indeed, for the zero-knowledge condition, it is crucial that the prover only responds to pairs that correspond to edges of the graph.*

- Prover's second step: *Upon receiving an edge* $\{u,v\} \in E$, *the prover sends to the verifier the keys to boxes* u *and* v.

  For simplicity of the analysis, if the verifier sends $\{u,v\} \notin E$ *then the prover behaves as if it has received a fixed* (or random) *edge in* E, *rather than suspending the interaction, which would have been the natural thing to do.*

- Verifier's second step: *The verifier unlocks and opens boxes* u *and* v, *and accepts if and only if they contain two different elements in* $\{1,2,3\}$.

The verifier strategy in Construction 9.10 is easily implemented in probabilistic polynomial-time. The same holds with respect to the prover's strategy, provided that it is given a 3-coloring of G as auxiliary input. Clearly, if the input graph is 3-colorable then the verfier accepts with probability 1 when interacting with the prescribed prover. On the other hand, if the input graph is not 3-colorable, then any contents put in the boxes must be invalid on at least one edge, and consequently the verifier will reject with probability at least $\frac{1}{|E|}$. Hence, the foregoing protocol exhibits a non-negligible gap in the accepting probabilities between the case of 3-colorable graphs and the case of non-3-colorable graphs. To increase the gap, the game may be repeated sufficiently many times (of course, using independent coin tosses in each repetition).

In the abstract setting of Construction 9.10, the zero-knowledge property follows easily, because one can simulate the real interaction by placing a random pair of different colors in the boxes indicated by the verifier. This indeed demonstrates that the verifier learns nothing from the interaction, because it expects to see a random pair of different colors (and indeed this is what it sees). Note that the aforementioned expectation relies on the fact that the boxes correspond to vertices that are connected by an edge.

This simple demonstration of the zero-knowledge property is not possible in the digital implementation (discussed next), because the boxes are not totally unaffected by their contents (but are rather effected, yet in an indistinguishable manner). Instead, we simulate the interaction as follows. We first guess (at random) which pair of boxes (rorresponding to an edge) the verifier would ask to open, and place a random pair of distinct colors in these boxes (and garbage in the rest).[15] We hand all boxes to the verifier. In case the verifier asks for the chosen pair (i.e., the one we guessed), we can complete the simulation. Otherwise, we try again (with a new random guess). Thus, it suffices to use boxes that hide their contents almost perfectly (rather than being perfectly opaque). Such boxes can be implemented digitally.

Teaching note: Indeed, we recommend presenting and analyzing in class only the foregoing abstract protocol. It suffices to briefly comment about the digital implementation, rather than presenting a formal proof of Theorem 9.11 (which can be found in [96] (or [87, Sec. 4.4])).

Digital implementation. We implement the abstract boxes (referred to in Construction 9.10) by using (adequately defined) commitment schemes. Loosely speaking, such a scheme is a two-phase game between a sender and a receiver such that after the first phase the sender is "committed" to a value and yet, at this stage, it is infeasible for the receiver to find out the committed value (i.e., the commitment is "hiding"). The committed value will be revealed to the receiver in the second phase and it is guaranteed that the sender cannot reveal a value other than the one committed (i.e., the commitment is "binding"). Such commitment schemes can be implemented assuming the existence of one-way functions (as in Definition 7.3).

Zero-knowledge proofs for other NP-sets. Using the fact that 3-colorability is NP-complete, one gets zero-knowledge proofs for any NP-set.[16] Furthermore, NP-witnesses can be efficiently transformed into polynomial-size circuits that implement the corresponding (presecribed zero-knowledge) prover strategies.

Theorem 9.11 (The ZK Theorem): *Assuming the existence of one-way functions, any NP-proof can be efficiently transformed into a* (computational) *zero-knowledge interactive proof. In particular, $\mathcal{NP} \subseteq \mathcal{ZK}$.*

The hypothesis (regarding the existence of one-way functions) in Theorem 9.11 seems unavoidable, because the existence of zero-knowledge proofs for "hard on the average" problems implies the existence of one-way functions (and, likewise, the existence of zero-knowledge proofs for sets outside $\mathcal{BPP}$ implies the existence of "auxiliary-input one-way functions"). Theorem 9.11 has a dramatic effect on the design of cryptographic protocols (see Appendix C). In a different vein we mention that, under the same assumption, any interactive proof can be transformed into a zero-knowledge one. (This transformation, however, is not efficient.)

[15] An alternative (and more efficient) simulation consists of putting random independent colors in the various boxes, hoping that the verifier asks for an edge that is properly colored. The latter event occurs with probability (approximately) 2/3, provided that the boxes hide their contents (almost) perfectly.

[16] Actually, we should either rely on the fact that the standard Karp-reductions are invertible in polynomial time or on the fact that the 3-colorability protocol is actually zero-knowledge with respect to auxiliary inputs (as in Definition C.9).

Theorem 9.12 (The ultimate ZK Theorem): *Assuming the existence of one-way functions, $\mathcal{IP} = \mathcal{ZK}$.*

Loosely speaking, Theorem 9.12 can be proved by recalling that $\mathcal{IP} = \mathcal{AM}(\texttt{poly})$ and modifying any public-coin protocol as follows: the modified prover sends commitments to its messages rather than the messages themselves, and once the original interaction is completed it proves (in zero-knowledge) that the corresponding transcript would have been accepted by the original verifier. Indeed, the latter assertion is of the "NP type", and thus the zero-knowledge proof system guaranteed in Theorem 9.11 can be invoked to prove it.

Reflection. The proof of Theorem 9.11 uses the fact that 3-colorability is NP-complete in order to obtain a zero-knowledge proofs for any set in $\mathcal{NP}$ by using such a protocol for 3-colorability (i.e., Construction 9.10). Thus, an NP-completeness result is used here in a "positive" way; that is, in order to construct something rather than in order to derive a hardness result. This was probably the first positive application of NP-completeness. Subsequent positive uses of completeness results have appeared in the context of interactive proofs [150, 191], probabilistically checkable proofs [18, 70, 14, 13], and "hardness versus randomness trade-offs" [19].

Perfect and Statistical Zero-Knowledge. The foregoing results may be contrasted with the results regarding the complexity of statistical zero-knowledge proof systems: Statistical zero-knowledge proof systems exist only for sets in $\mathcal{IP}(2) \cap \mathrm{co}\mathcal{IP}(2)$, and thus are unlikely to exist for all NP-sets. On the other hand, the class Statistical Zero-Knowledge is known to contain some hard problems, and turns out to have interesting complexity theoretic properties (e.g., being closed under complementation, and having very natural complete problems). The interested reader is referred to [211].

9.2.3 Proofs of Knowledge – a parenthetical subsection

Teaching note: Technically speaking, this topic belongs to Section 9.1, but the more interesting examples of proof of knowledge are ones that are zero-knowledge.

Loosely speaking, "proofs of knowledge" are interactive proofs in which the prover asserts "knowledge" of some object (e.g., a 3-coloring of a graph), and not merely its existence (e.g., the existence of a 3-coloring of the graph, which in turn is equivalent to the assertion that the graph is 3-colorable).

What do we mean by saying that a *machine* knows something? Any standard dictionary suggests several meanings for the verb `to know`, but these are typically phrased with reference to the notion of *awareness*, a notion which is certainly inapplicable in the context of machines. Instead, we should look for a *behavioristic* interpretation of the verb `to know`. Indeed, it is reasonable to link knowledge with the ability to do something (e.g., the ability to write down whatever one knows). Hence, we will say that a machine knows a string α if it *can* output the string α. But this seems as total non-sense too: a machine has a well defined output –

either the output equals α or it does not. So what can be meant by saying that *a machine can do something?* Loosely speaking, it may mean that the machine can be *easily modified* so that it does whatever is claimed. More precisely, it may mean that there exists an *efficient* machine that, using the original machine as a black-box (or given its code as an input), outputs whatever is claimed.

So much for defining the *knowledge of machines*. Yet, whatever a machine knows or does not know is its own business. What can be of interest and reference *to the outside* is whatever can be deduced about the knowledge of a machine by interacting with it. Hence, we are interested in proofs of knowledge (rather than in mere knowledge).

For sake of simplicity let us consider a concrete question: *how can a machine prove that it knows a 3-coloring of a graph?* An obvious way is just sending the 3-coloring to the verifier. Yet, we claim that applying the protocol in Construction 9.10 (i.e., the zero-knowledge proof system for 3-Colorability) is an alternative way of proving knowledge of a 3-coloring of the graph.

The definition of a *verifier of knowledge of 3-coloring* refers to any possible prover strategy and links the ability to "extract" a 3-coloring (of a given graph) from such a prover to the probability that this prover convinces the verifier. That is, the definition postulates the existence of an efficient universal way of "extracting" a 3-coloring of a given graph by using any prover strategy that convinces this verifier to accept this graph with probability 1 (or, more generally, with some noticeable probability). On the other hand, we should no expect this extrator to obtain much from prover strategies that fail to convince the verifier (or, more generally, convince it with negligible probability). A robust definition should allow a smooth transition between these two extremes (and in particular between provers that convince the verifier with noticeable probability and those that convince it with negligible probability). Such a definition should also support the intuition by which the following strategy of Alice is zero-knowledge: *Alice sends Bob a 3-coloring of a given graph provided that Bob has successfully proved to her that he knows such a coloring*. We stress that the zero-knowledge property of Alice's strategy should hold regardless of the proof-of-knowledge system used for proving Bob's knowledge of a 3-coloring.

Loosely speaking, we say that an interactive machine, V, constitutes a verifier for knowledge of 3-coloring if, for any prover strategy P, the complexity of extracting a 3-coloring of G when using machine P as a "black box"[17] is inversely proportional to the probability that V is convinced by P (to accept the graph G). Namely, the extraction of the 3-coloring is done by an oracle machine, called an extractor, that is given access to a function specifying the behavior P (i.e., the messages it sends in response to particular messages it may receive). We require that the (*expected*) *running time of the extractor, on input G and access to an oracle specifying P's messages, be inversely related* (by a factor polynomial in $|G|$) *to the probability that P convinces V to accept G*. In particular, if P always convinces V to accept G, then the extractor runs in expected polynomial-time. The same holds in case P convinces V to accept with noticeable probability. On the other hand, if P never

[17] Indeed, one may consider also non-black-box extractors.

convinces V to accept, then nothing is required of the extractor. We stress that the latter special cases do not suffice for a satisfactory definition; see discussion in [87, Sec. 4.7.1].

Proofs of knowledge, and in particular zero-knowledge proofs of knowledge, have many applications to the design of cryptographic schemes and cryptographic protocols. These are enabled by the following general result.

Theorem 9.13 (Theorem 9.11, revisited): *Assuming the existence of one-way functions, any NP-relation has a zero-knowledge proof of knowledge* (of corresponding NP-witnesses). *Furthermore, the prescribed prover strategy can be implemented in probabilistic polynomial-time, provided it is given such an NP-witness.*

9.3 Probabilistically Checkable Proof Systems

Teaching note: Probabilistically checkable proof (PCP) systems may be viewed as a restricted type of interactive proof systems in which the prover is memoryless and responds to each verifier message as if it were the first such message. This perspective creates a tighter link with previous sections, but is somewhat contrived. Indeed, such a memoryless prover may be viewed as a static object that the verifier may query at locations of its choice. But then it is more appealing to present the model using the (more traditional) terminology of oracle machines rather than using (and degenerating) the terminology of interactive machines.

Probabilistically checkable proof systems can be viewed as standard (deterministic) proof systems that are augmented with a probabilistic procedure capable of evaluating the validity of the assertion by examining few locations in the alleged proof. In fact, we focus on the latter probabilistic procedure, which is given direct access to the individual bits of the alleged proof (and need not scan it bit-by-bit). Thus, the alleged proof is a string, as in the case of a traditional proof system, but we are interested in probabilistic verification procedures that access only few locations in the proof, and yet are able to make a meaningful probabilistic verdict regarding the validity of the alleged proof. Specifically, the verification procedure should accept any valid proof (with probability 1), but rejects with probability at least 1/2 any alleged proof for a false assertion.

The main complexity measure associated with probabilistically checkable proof systems is indeed their query complexity. Another complexity measure of natural concern is the length of the proofs being employed, which in turn is related to the randomness complexity of the system. The randomness complexity of PCPs plays a key role in numerous applications (e.g., in composing PCP systems as well as when applying PCP systems to derive inapproximability results), and thus we specify this parameter rather than the proof length.

Teaching note: Indeed, PCP systems are most famous for their role in deriving numerous inapproximation results (see Section 9.3.3), but our view is that the latter is merely one extremely important application of the fundamental notion of a PCP system. Our presentation is organized accordingly.

9.3.1 Definition

Loosely speaking, a probabilistically checkable proof system consists of a probabilistic polynomial-time verifier having access to an oracle that represents an alleged proof (in redundant form). Typically, the verifier accesses only few of the oracle bits, and these bit positions are determined by the outcome of the verifier's coin tosses. As in the case of interactive proof systems, it is required that if the assertion holds then the verifier always accepts (i.e., when given access to an adequate oracle); whereas, if the assertion is false then the verifier must reject with probability at least $\frac{1}{2}$, no matter which oracle is used. The basic definition of the PCP setting is given in Item (1) below. Yet, the complexity measures introduced in Item (2) are of key importance for the subsequent discussions.

Definition 9.14 (Probabilistic Checkable Proofs – PCP):

1. *A* probabilistic checkable proof system (PCP) for a set *S is a probabilistic polynomial-time oracle machine, called* verifier *and denoted V, that satisfies the following two conditions:*

   - Completeness: *For every $x \in S$ there exists an oracle π_x such that, on input x and access to oracle π_x, machine V always accepts x.*
   - Soundness: *For every $x \notin S$ and every oracle π, on input x and access to oracle π, machine V rejects x with probability at least $\frac{1}{2}$.*

2. *We say that a probabilistic checkable proof system has* query complexity *$q : \mathbb{N} \to \mathbb{N}$ if, on any input of length n, the verifier makes at most $q(n)$ oracle queries.*[18] *Similarly, the* randomness complexity *$r : \mathbb{N} \to \mathbb{N}$ upper-bounds the number of coin tosses performed by the verifier on a generic n-bit long input.*

   For integer functions r and q, we denote by $\mathcal{PCP}(r,q)$ the class of sets having probabilistic checkable proof systems of randomness complexity r and query complexity q. For sets of integer functions, R and Q,

$$\mathcal{PCP}(R,Q) \stackrel{\rm def}{=} \bigcup_{r\in R\,,\,q\in Q} \mathcal{PCP}(r,q)\,.$$

We note that the oracle π_x referred to in the completeness condition a PCP system constitutes a proof in the standard mathematical sense (with respect to a verification procedure that examines all possible outcomes of V's internal coin tosses). Furthermore, the oracles in PCP systems of logarithmic randomness complexity constitute NP-proofs. However, these oracles have the extra remarkable property of enabling a lazy verifier to toss coins, take its chances and "assess" the validity of the proof without reading all of it (but rather by reading a tiny portion of it). Potentially, this allows the verifier to utilize very long proofs (i.e., of super-polynomial length) or alternatively examine very few bits of an NP-proof.

[18] As usual in complexity theory, the oracle answers are always binary (i.e., either 0 or 1).

We note that the error probability (in the soundness condition) of PCP systems can be reduced by successive applications of the proof system. In particular, repeating the process for k times, reduces the probability that the verifier is fooled by a false assertion to 2^{-k}, whereas all complexities increase by at most a factor of k. Thus, PCP systems provide a trade-off between the number of locations examined in the proof and the confidence in the validity of the assertion.

Adaptive versus non-adaptive verifiers. Definition 9.14 allows the verifier to be adaptive; that is, the verifier may determine its queries based on the answers it has received to previous queries (in addition to their dependence on the input and the verifier's internal coin tosses). In contrast, non-adaptive verifiers determine all their queries based solely on their input and internal coin tosses. We comment that most constructions of PCP systems use non-adaptive verifiers, and in fact in many sources PCP systems are defined as non-adaptive.

Randomness versus proof length. Note that the "effective" length of proofs for any PCP system is related to its query and randomness complexities, where the effective length means the number of locations in a generic proof-oracle that may be examined on a fixed input and any possible sequence of internal coin tosses. Specifically, if the PCP system has query complexity q and randomness complexity r then its effective proof length is upper-bounded by 2^{q+r}, whereas a bound of $2^r \cdot q$ holds for non-adaptive systems (see Exercise 9.7). On the other hand, in some sense, the randomness complexity of a PCP can be upper-bounded by the logarithm of the length of the proofs employed (provided we allow non-uniform verifiers; see Exercise 9.9).

On the role of randomness. The PCP Theorem (i.e., $\mathcal{NP} = \mathcal{PCP}(\texttt{log}, O(1))$) exhibits a trade-off between the number of bits examined in the alleged proof and the confidence in the validity of the assertion. We note that such a trade-off is impossible if one requires the verifier to be deterministic. This is due to the fact that every set in $\mathcal{PCP}(r,q)$ has an NP-proof system that employs proofs of length $2^r q$ (see Exercise 9.8). Thus, $\mathcal{PCP}(r,q) \subseteq \textsc{Dtime}(2^{2^r q} \cdot \mathrm{poly})$, and, in particular, $\mathcal{PCP}(0,\texttt{log}) = \mathcal{P}$. Furthermore, since it is unlikely that all NP-sets have NP-proof systems that employs proofs of (say) linear length, it follows that $\mathcal{PCP}(r,q)$ is unlikely to contain $\mathcal{NP}$ for $2^{r(n)}q(n) \leq n$ (or for any other fixed polynomial that bounds $2^r q$). Actually, $\mathcal{P} \neq \mathcal{NP}$ implies that $\mathcal{NP}$ is not contained in $\mathcal{PCP}(o(\log), o(\log))$ (see Exercise 9.11).

9.3.2 The Power of Probabilistically Checkable Proofs

The celebrated PCP Theorem asserts that $\mathcal{NP} = \mathcal{PCP}(\texttt{log}, O(1))$, and this result is indeed the focus of the current section. But before getting to it we make several simple observation regarding the PCP Hierarchy.

We first note that $\mathcal{PCP}(\texttt{poly}, 0)$ equals co$\mathcal{RP}$, whereas $\mathcal{PCP}(0, \texttt{poly})$ equals $\mathcal{NP}$. It is easy to prove an upper bound on the non-deterministic time complexity of sets in the PCP hierarchy (see Exercise 9.8):

Proposition 9.15 (upper-bounds on the power of PCPs): *For every polynomially bounded integer function r, it holds that $\mathcal{PCP}(r, \texttt{poly}) \subseteq \mathrm{N}\textsc{time}(2^r \cdot \texttt{poly})$. In particular, $\mathcal{PCP}(\texttt{log}, \texttt{poly}) \subseteq \mathcal{NP}$.*

The focus on PCP systems of logarithmic randomness complexity reflects an interest in PCP systems that utilize proof oracles of polynomial length (see discussion in Section 9.3.1). We stress that such PCP systems (i.e., $\mathcal{PCP}(\texttt{log}, q)$) are NP-proof systems with a (potentially amazing) extra property: the validity of the assertion can be "probabilistically evaluated" by examining a (small) portion (i.e., $q(n)$ bits) of the proof. Thus, for any fixed polynomially bounded function q, a result of the form

$$\mathcal{NP} \subseteq \mathcal{PCP}(\texttt{log}, q) \tag{9.5}$$

is interesting (because it applies also to NP-sets having witnesses of length exceeding q), and the smaller q – the better. The PCP Theorem asserts the amazing fact by which q can be made a constant.

Theorem 9.16 (The PCP Theorem): *$\mathcal{NP} \subseteq \mathcal{PCP}(\texttt{log}, O(1))$.*

Thus, probabilistically checkable proofs in which the verifier tosses only logarithmically many coins and makes only a constant number of queries exist for every set in $\mathcal{NP}$. Furthermore, the proof of Theorem 9.16 is constructive in the sense that it allows to efficiently transform any NP-witness (for an instance of a set in $\mathcal{NP}$) into an oracle that makes the PCP verifier accept (with probability 1). Thus, NP-proofs can be transformed into NP-proofs that offer a trade-off between the portion of the proof being read and the confidence it offers. Specifically, for every $\varepsilon > 0$, if the verifier is willing to tolerate an error probability of ε then it for it examine $O(\log(1/\varepsilon))$ bits of the (transformed) NP-proof. Indeed (as discussed in Section 9.3.1), these bit locations need to be selected at random.

A new characterization of NP: Combining Theorem 9.16 with Proposition 9.15 we obtain the following characterization of $\mathcal{NP}$.

Corollary 9.17 (The PCP characterization of NP): *$\mathcal{NP} = \mathcal{PCP}(\texttt{log}, O(1))$.*

The proof of the PCP Theorem: Theorem 9.16 is a culmination of a sequence of remarkable works, each establishing meaningful and increasingly stronger versions of Eq. (9.5). A presentation of the full proof of Theorem 9.16 is beyond the scope of the current work (and is, in our opinion, unsuitable for a basic course in complexity theory). Instead, we present an overview of the original proof (see §9.3.2.2) as well as of an alternative proof (see §9.3.2.3) that was found more than a decade later. We will start, however, by presenting a weaker result that is used in both proofs of Theorem 9.16 and is also of independent interest. This weaker result (see §9.3.2.1) asserts that any NP-set has a PCP system with constant query complexity (albeit with polynomial randomness complexity); that is, $\mathcal{NP} \subseteq \mathcal{PCP}(\texttt{poly}, O(1))$.

Teaching note: In our opinion, presenting in class any part of the proof of the PCP Theorem should be given low priority. In particular, presenting the connections between PCP and the complexity of approximation should be given a higher priority. As for relative priorities among the following three subsections, we recommend giving §9.3.2.1 the highest priority, because it offers a direct demonstration of the power of PCPs. As for the two alternative proofs of the PCP Theorem itself, our recommendation depends on the intended goal. On one hand, for the purpose of merely giving a taste of the ideas involved in the proof, we prefer an overview of the original proof (provided in §9.3.2.2). On the other hand, for the purpose of actually providing a full proof, we definitely prefer the new proof (which is only outlined in §9.3.2.3).

9.3.2.1 Proving that $\mathcal{NP} \subseteq \mathcal{PCP}(\texttt{poly}, O(1))$

The fact that any NP-set has a PCP system with constant query complexity (regardless of its randomness complexity) already testifies to the power of PCP systems. It asserts that *probabilistic verification of proofs is possible by inspecting very few locations in a* (potentially huge) *proof.* Indeed, the PCP systems presented next utilize exponentially long proofs, but they do so while inspecting these proofs at a constant number of (randomly selected) locations.

We start with a brief overview of the construction. It suffices to construct a PCP for proving the satisfiability of a given system of quadratic equations over GF(2), because this problem is NP-complete.[19] The oracle (of this PCP) is supposed to provide the values of all quadratic expressions evaluated at some assignment to the (say n) variables of the system of quadratic equations (given as input). This assignment is supposed to satisfy the latter system. We distinguish two tables in the oracle: The first table corresponding to the (2^n) linear expressions and the second table to the (2^{n^2}) quadratic expressions. Each table is tested for self-consistency (via a "linearity test"), and the two tables are tested to be consistent with each other (via a "matrix-equality" test, which utilizes "self-correction"). Each of these tests utilizes a constant number of Boolean queries, and randomness that is logarithmic in the size of the corresponding table (and is thus $O(n^2)$). Finally, we test (again via self-correction) the value assigned by these tables to an expression obtained by a random linear combination of the system of quadratic equations that is given as input. Details follow.

The starting point. We construct a PCP system for the set of satisfiable quadratic equations over GF(2). The input is a sequence of such equations over the variables $x_1, ..., x_n$, and the proof oracle consist of two parts (or tables), which are supposed to provide information reagrding some satisfying assignment $\alpha = \alpha_1 \cdots \alpha_n$ (also viewed as an n-ary vector over GF(2)). The first part, T_1, is supposed to provide a Hadamard encoding of the said satisfying assignment; that is, entry $\beta \in \mathrm{GF}(2)^n$ in this table is supposed to provide the inner product mod 2 of the n-ary vectors α and β (i.e., $T_1(\beta)$ is supposed to equal $\sum_{i=1}^n \beta_i \alpha_i$). The second part, T_2, is supposed to provide all linear combinations of the values of the $\alpha_i \alpha_j$'s;

[19] Here and elsewhere, we denote by GF(2) the 2-element field.

that is, for $\gamma \in \mathrm{GF}(2)^{n^2}$, the value of $T_2(\gamma)$ is supposed to equal $\sum_{i,j} \gamma_{i,j}\alpha_i\alpha_j$. (Indeed T_1 is contained in T_2, because $\sigma^2 = \sigma$ for any $\sigma \in \mathrm{GF}(2)$.) The PCP verifier will uses the two tables to check that the input (i.e., a sequence of quadratic equations) is satisfied by the assignment that is encoded in the two tables. Needless to say, these tables may not be a valid encoding of any n-ary vector (let alone one that satisfies the input), and so the verifier also needs to check that the encoding is (close to being) valid. We will focus on this task first.

Testing the Hadamard Code. Note that T_1 is supposed to encode a linear function; that is, there must be some $\alpha = \alpha_1 \cdots \alpha_n \in \mathrm{GF}(2)^n$ such that $T_1(\beta) = \sum_{i=1}^n \alpha_i\beta_i$ for every $\beta = \beta_1 \cdots \beta_n \in \mathrm{GF}(2)^n$. This can be tested by selecting uniformly $\beta', \beta'' \in \mathrm{GF}(2)^n$ and checking whether $T_1(\beta') + T_1(\beta'') = T_1(\beta' + \beta'')$, where $\beta' + \beta''$ denotes addition of vectors over GF(2). The analysis of this natural tester turns out to be quite complex. Nevertheless, it is indeed the case that any table that is 0.01-far from being linear is rejected with probability at least 0.02 (see Exercise 9.12), where T is ε-far from being linear if T disagrees with any linear function f on more than an ε fraction of the domain (i.e., $\mathsf{Pr}_r[T(r) = f(r)] > \varepsilon$).

By repeating the linearity test for a constant number of times, we may reject each table that is 0.01-far from being a codeword of the Hadamard Code with probability at least 0.99. Thus, using a constant number of queries, the verifier rejects any T_1 that is 0.01-far from being a Hadamard encoding of any $\alpha \in \mathrm{GF}(2)^n$, and likewise rejects any T_2 that is 0.01-far from being a Hadamard encoding of any $\alpha' \in \mathrm{GF}(2)^{n^2}$. We may thus assume that T_1 (resp., T_2) is 0.01-close to the Hadamard encoding of some α (resp., α'). (This does not mean, however, that α' equals the outer produce of α with itself.)

In the rest of the analysis, we fix $\alpha \in \mathrm{GF}(2)^n$ and $\alpha' \in \mathrm{GF}(2)^{n^2}$, and denote the Hadamard encoding of α (resp., α') by $f_\alpha : \mathrm{GF}(2)^n \to \mathrm{GF}(2)$ (resp., $f_{\alpha'} : \mathrm{GF}(2)^{n^2} \to \mathrm{GF}(2)$). Recall that T_1 (resp., T_2) is 0.01-close to f_α (resp., $f_{\alpha'}$).

Self-correction of the Hadamard Code. Suppose that T is ε-close to a linear function f (i.e., $\mathsf{Pr}_r[T(r) = f(r)] \le \varepsilon$). Then, we can recover the value of f at any desired point x, by making two (random) queries to T. Specifically, for a uniformly selected $r \in \mathrm{GF}(2)^n$, we use the value $T(x+r) - T(r)$. Note that the probability that we recover the correct value is at least $1 - 2\varepsilon$, because $\mathsf{Pr}_r[T(x+r) - T(r) = f(x+r) - f(r)] \ge 1 - 2\varepsilon$ and $f(x+r) - f(r) = f(x)$ by linearity of f. (Needless to say, for $\varepsilon < 1/4$, the function T cannot be ε-close to two different linear functions.)[20] Thus, assuming that T_1 is 0.01-close to f_α (resp., T_2 is 0.01-close to $f_{\alpha'}$) we may correctly recover (i.e., with error probability 0.02) the value of f_α (resp., $f_{\alpha'}$) at any desired point by making 2 queries to T_1 (resp., T_2).

Checking consistency of f_α and $f_{\alpha'}$. Suppose that we are given access to $f_\alpha : \mathrm{GF}(2)^n \to \mathrm{GF}(2)$ and $f_{\alpha'} : \mathrm{GF}(2)^{n^2} \to \mathrm{GF}(2)$, where $f_\alpha(\beta) = \sum_i \alpha_i\beta_i$ and

[20]Indeed, this fact follows from the self-correction argument, but a simpler proof merely refers to the fact that the Hadamard code has relative distance 1/2.

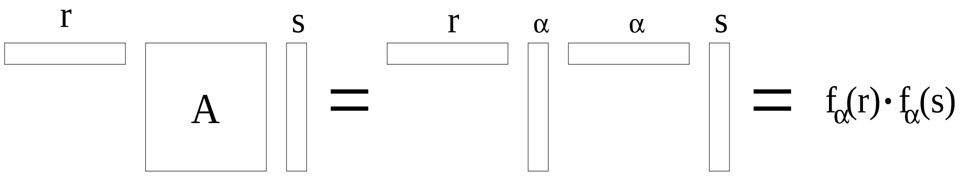


Figure 9.2: Detail for testing consistency of linear and quadratic forms.

$f_{\alpha'}(\beta') = \sum_{i,j} \alpha'_{i,j}\beta'_{i,j}$, and that we wish to verify that $\alpha'_{i,j} = \alpha_i\alpha_j$ for every $i,j \in \{1,...,n\}$. In other words, we are given a (somewhat weird) encoding of two matrices, $A = (\alpha_i\alpha_j)_{i,j}$ and $A' = (\alpha'_{i,j})_{i,j}$, and we wish to check whether or not these matrices are identical. It can be shown (see Exercise 9.14) that if $A \neq A'$ then $\mathsf{Pr}_{r,s}[r^\top As \neq r^\top A's] \geq 1/4$, where r and s are random n-ary vectors. Note that, in our case (see Figure 9.2), $r^\top As = \sum_j(\sum_i r_i\alpha_i\alpha_j)s_j = f_\alpha(r)f_\alpha(s)$ and $r^\top A's = \sum_j(\sum_i r_i\alpha'_{i,j})s_j = f_{\alpha'}(rs^\top)$, where $rs^\top$ is the outer-product of s and r, and so (for $A \neq A'$) we have $\mathsf{Pr}_{r,s}[f_\alpha(r)f_\alpha(s) \neq f_{\alpha'}(rs^\top)] \geq 1/4$. Using self-correction (to obtain the desired value of $f_{\alpha'}$ on $rs^\top$, which is not uniformly distributed in $\mathrm{GF}(2)^{n^2}$), we test the consistency of f_α and $f_{\alpha'}$; that is, we select uniformly $r,s \in \mathrm{GF}(2)^n$ and $R \in \mathrm{GF}(2)^{n^2}$ and check whether or not $T_1(r)T_1(s) = T_2(rs^\top + R) - T_2(R)$.

Checking that α satisfies the quadratic system. Suppose that we are given access to f_α and $f_{\alpha'}$ as above (i.e., in particular, $\alpha' = \alpha\alpha^\top$ is the outer product of α with itself). A key observation is that if α does not satisfy a system of quadratic equations then, with probability 1/2, it does not satisfy a random linear combination of these equations. Thus, in order to check whether α satisfies the quadratic system, we create a single quadratic equation (by taking such a random linear combination) and compare the value of the resulting quadratic expression to the corresponding value, by recovering the value of $f_{\alpha'}$ at a single point (which corresponds to the quadratic equation). That is, to test whether α satisfies the quadratic equation $Q(x) = \sigma$, we test whether $f_{\alpha'}(Q) = \sigma$. The actual checking is implemented by the verifier using self-correction (of the table T_2).

To summarize, the verifier performs a constant number of queries and uses randomness that is quadratic in the number of variables. If the quadratic system is satisfiable (by some α), then the verifier accepts the corresponding tables T_1 and T_2 (i.e., $T_1 = f_\alpha$ and $T_2 = f_{\alpha\alpha^\top}$) with probability 1. On the other hand, if the quadratic system is unsatisfiable, then any pair of tables (T_1, T_2) will be rejected with constant probability (by one of the foregoing tests). It follows that $\mathcal{NP} \subseteq \mathcal{PCP}(r, O(1))$, where $r(n) = O(n^2)$.

9.3.2.2 Overview of the first proof of the PCP Theorem

The proof of the PCP Theorem (Theorem 9.16) is one of the most complicated proofs in the Theory of Computation. The original proof consists of three main conceptual steps, which we roughly sketch first and further discuss later.

1. Constructing a (non-adaptive) PCP system for $\mathcal{NP}$ having *logarithmic randomness and polylogarithmic query complexity*. Furthermore, this proof system has additional properties that enable proof composition as in Step (3) below.

2. Constructing a PCP system for $\mathcal{NP}$ having *polynomial randomness and constant query complexity* (indeed as in §9.3.2.1). This proof system too has additional properties enabling proof composition as in Step (3).

3. The proof composition paradigm:[21] In general, this paradigm allows to compose two proof systems such that the "inner" one is used to probabilistically verify the acceptance criteria of the "outer" verifier. The aim is to conduct the latter verification using much fewer queries than the query complexity of the "outer" proof system. In particular, the inner verifier cannot afford to read its input, which makes composition more subtle than the term suggests.

   Loosely speaking, the *outer* verifier should be robust in the sense that its soundness condition guarantee that with high probability the oracle answers are "far" from satisfying the residual decision predicate (rather than merely not satisfy it). (Furthermore, the latter predicate, which is well-defined by the non-adaptive nature of the outer verifier, must have a circuit of size bounded by a polynomial in the number of queries.) The *inner* verifier is given oracle access to its input and is charged for each query made to it, but is only required to reject with high probability inputs that are far from being valid (and, as usual, accept inputs that are valid). That is, the inner verifier is actually a verifier of proximity.

   Composing two such PCPs yields a new PCP for $\mathcal{NP}$, where the new proof oracle consists of the proof oracle of the "outer" system and a sequence of proof oracles for the "inner" system (one "inner" proof per each possible random-tape of the "outer" verifier). Thus, composing an outer verifier of randomness complexity r' and query complexity q' with an inner verifier of randomness complexity r'' and query complexity q'' yields a PCP of randomness complexity $r(n) = r'(n) + r''(q'(n))$ and query complexity $q(n) = q''(q'(n))$, because $q'(n)$ represents the length of the input (oracle) that is accessed by the inner verifier. Recall that the outer verifier is non-adaptive, and thus if the inner verifier is non-adaptive (resp., robust) then so is the verifier resulting from the composition, which is important in case we wish to compose the latter verifier with another inner verifier.

In particular, the proof system of Step (1) is composed with itself [using $r'(n) = r''(n) = O(\log n)$ and $q'(n) = q''(n) = \text{poly}(\log n)$] yielding a PCP system (for $\mathcal{NP}$) of randomness complexity $r(n) = r'(n)+r''(q'(n)) = O(\log n)$ and query complexity $q(n) = q''(q'(n)) = \text{poly}(\log\log n)$. Composing the latter system (used as an "outer" system) with the the PCP system of Step (2), yields a PCP system (for

[21] Our presentation of the composition paradigm follows [33], rather than the original presentation of [14, 13].

$\mathcal{NP}$) of randomness complexity $r(n)+\text{poly}(q(n)) = O(\log n)$ and query complexity $O(1)$, thus establishing the PCP Theorem.

A more detailed overview – the plan. The foregoing description of Step 3 uses two (non-trivial) PCP systems and refers to additional properties such as robustness and verification of proximity. A PCP system of polynomial randomness complexity and constant query complexity (as postulated in Step 2) is outlined in §9.3.2.1. We thus start by discussing the notions of verifying proximity and being robust, while demonstrating their applicability to the said PCP. Finally, we outline the other PCP system (i.e., the one postulated in Step 1).

PCPs of Proximity. Recall that a standard PCP verifier gets an explicit input and is given oracle access to an alleged proof (for membership of the input in a predetermined set). In contrast, a PCP of proximity is given oracle access to two oracles, one representing an input and the other being an alleged proof. Typically, the query complexity of the corresponding verifier is lower than the length of the input oracle, and hence the verifier cannot afford to read the entire input and cannot be expected to make absolute statements about it. Indeed, instead of deciding whether or not the input is in a predetermined set, the verifier needs only distinguish the case that the input is in the set from the case that the input is *far* from the set (where far means being at *relative* Hamming distance at least 0.01 (or any other constant)).

For example, consider a variant of the system of §9.3.2.1 in which the quadratic system is fixed[22] and the verifier needs to determine whether an input oracle satisfies the said system or is far from any assignment that satisfies it. The proof oracle is as in §9.3.2.1, and a PCP of proximity may proceed as in §9.3.2.1 and in addition perform a proximity test to verify that the input oracle is close to the assignment encoded in the proof oracle. Specifically, the verifier may read a uniformly selected bit of the input oracle and compare the value to the self-correction obtained from the proof oracle (i.e., for a uniformly selected $i \in \{1, ..., n\}$, we compare the i^{th} bit of the input oracle to the self-correction of the value $T_1(0^{i-1}10^{n-i})$, obtained from the proof oracle).

Robust PCPs. Composing an "outer" PCP verifier with an "inner" PCP verifier of proximity makes sense provided that the *outer* verifier rejects in a "robust" manner. That is, the soundness condition of a robust verifier requires that (with probability at least 1/2) the oracle answers are *far* from any sequence that is acceptable by the residual predicate (rather than merely that the answers are rejected by this predicate). Indeed, if the outer verifier is (non-adaptive and) robust, then it suffices that the inner verifier distinguish (with the help of an adequate proof) answers that are valid from answers that are far from being valid.

For example, if robustness is defined as referring to *relative constant distance* (which is indeed the case), then the PCP of §9.3.2.1 (as well as any PCP of con-

[22] Indeed, in our applications the quadratic system will be "known" to the ("inner") verifier, because it is determined by the ("outer") verifier.

stant query complexity) is trivially robust. However, we will not care about the robustness of this PCP, because we only use this PCP as an inner verifier in proof composition. In contrast, we will care about the robustness of PCPs that are used as outer verifiers (e.g., the PCP presented next).

Teaching note: Unfortunately, the construction of a PCP of logarithmic randomness and polylogarithmic query complexity for NP is quite complicated and involves many technical details. Furthermore, obtaining a robust version of this PCP is beyond the scope of the current text. Thus, we view the following description as merely providing a flavor of the underlying ideas.

PCP of logarithmic randomness and polylogarithmic query complexity for NP. We start by showing that $NP \subseteq \mathcal{PCP}(f,f)$, for $f(n) = \text{poly}(\log n)$. The proof system is based on a different arithmetization of CNF formulae (i.e., different than the one used in §9.1.2.2 for constructing an interactive proof system for co$\mathcal{NP}$). In this arithmetization, the names of the variables (resp., clauses) of the input formula ϕ are represented by binary strings of logarithmic (in $|\phi|$) length, and a *generic* variable (resp., clause) of ϕ is represented by a logarithmic number of *new variables* (which are assigned values in a finite field $\mathrm{F} \supset \{0,1\}$). The (structure of the) input 3CNF formula $\phi(x_1,...,x_n)$ is represented by a Boolean function $C_\phi : \{0,1\}^{O(\log n)} \to \{0,1\}$ such that $C_\phi(\alpha,\beta_1,\beta_2,\beta_3) = 1$ if and only if, for $i = 1,2,3$, the i^{th} literal in the α^{th} clause has index $\beta_i = (\gamma_i,\sigma_i)$ (which is viewed as a variable name augmented by its sign). We consider a multi-linear extension of C_ϕ over F, denoted Φ; that is, Φ is the (unique) multi-linear polynomial that agrees with C_ϕ on $\{0,1\}^{O(\log n)} \subset \mathrm{F}^{O(\log n)}$. Thus, on input ϕ, the verifier first constructs C_ϕ and Φ. Part of the proof oracle of this verifier is viewed as function $A : \mathrm{F}^{\log n} \to \mathrm{F}$, which is supposed to be a multi-linear extension of a truth assignment that satisfies ϕ (i.e., for every $\gamma \in \{0,1\}^{\log n} \equiv [n]$, the value $A(\gamma)$ is supposed to be the value of the γ^{th} variable in such an assignment). Thus, we wish to check whether, for every $\alpha \in \{0,1\}^{\log|\phi|}$, it holds that

$$\sum_{\beta_1\beta_2\beta_3 \in \{0,1\}^{3\log 2n}} \Phi(\alpha,\beta_1,\beta_2,\beta_3) \cdot \prod_{i=1}^{3} (1 - A'(\beta_i)) = 0 \qquad (9.6)$$

where $A'(\beta)$ is the value assigned by A to the β^{th} literal (i.e., if $\beta = (\gamma,\sigma)$, where $\gamma \in \{0,1\}^{\log n}$ is a variable name and $\sigma \in \{0,1\}$ is the literal's type, then $A'(\beta) = \sigma \cdot A(\gamma) + (1-\sigma)\cdot(1-A(\gamma))$). Thus, Eq. (9.6) holds if and only if the α^{th} clause is satisfied by the assignment induced by A (since $A'(\beta) = 1$ must hold for at least one of the three literals β that appear in this clause). As in §9.3.2.1, we cannot afford to verify all instances of Eq. (9.6), and unlike in §9.3.2.1 we cannot afford to take a random linear combination of them either (as this requires too much randomness). Fortuntaely, taking a "pseudorandom" linear combination of these equations is good enough. Specifically, using an adequate small-bias probability space[23] $S \subset$

[23]See Section 8.6.2.

$\mathrm{F}^{|\phi|}$ (of size $\mathrm{poly}(|\phi|\cdot|F|)$), the verifier may select uniformly $(s_1,...,s_{|\phi|}) \in S$ and verify

$$\sum_{\alpha\beta_1\beta_2\beta_3\in\{0,1\}^\ell} s_\alpha \cdot \Phi(\alpha,\beta_1,\beta_2,\beta_3) \cdot \prod_{i=1}^{3}(1-A'(\beta_i)) = 0 \tag{9.7}$$

where $\ell \stackrel{\rm def}{=} \log|\phi| + 3\log 2n$. The small-bias property guarantees that if A fails to satisfy any of the equations of type Eq. (9.6) then, with high probability (over the choice of $(s_1,...,s_{|\phi|}) \in S$), it is the case that A fails to satisfy Eq. (9.7). (Since $|S| = \mathrm{poly}(|\phi|\cdot|F|)$ rather that $|S| = 2^{|\phi|}$, we can select a sample in S using $O(\log|\phi|)$ coin tosses.)

Now, assuming that A is a low-degree polynomial, we can probabilistically verify Eq. (9.7) by applying a summation test (as in the interactive proof for $\mathrm{co}\mathcal{NP}$). Indeed, the verifier obtains the relevant univariant polynomials by making adequate queries (which specify the entire sequence of choices made so far in the summation test). Note that after stripping the ℓ summations, the verifier end-ups up with an expression that contains three unknown values of A', which it may obtain by making corresponding queries to A. This summation test involves tossing $\ell\cdot\log|\mathrm{F}|$ coins and making $(\ell+3)\cdot O(\log|\mathrm{F}|)$ Boolean queries (which correspond to ℓ queries that are each answered by a univariate polynomial of constant degree (over F), and three queries to A (each answered by an element of F)). Needless to say, we must also check that A is indeed a multi-variate polynomial of low degree (or rather that it is close to such a polynomial). A low-degree test of complexities similar to those of the summation text does exist. Using a finite field F of $\mathrm{poly}(\log(n))$ elements, this yields $\mathcal{NP} \subseteq \mathcal{PCP}(f,f)$ for $f(n) \stackrel{\rm def}{=} O(\log(n)\cdot\log\log(n))$.

To obtain the desired PCP system of logarithmic randomness complexity, we represent the names of the original variables and clauses by $\frac{O(\log n)}{\log\log n}$-long sequences over $\{1,...,\log n\}$, rather than by logarithmically-long binary sequences. This requires using low degree polynomial extensions (i.e., polynomial of degree $(\log n)-1$), rather than multi-linear extensions. We can still use a finite field of $\mathrm{poly}(\log(n))$ elements, and so we need only $\frac{O(\log n)}{\log\log n}\cdot O(\log\log n)$ random bits for the summation and low-degree tests. However, the number of queries (needed for obtaining the answers in these tests) grows, because now the polynomials involved have individual degree $(\log n)-1$ rather than constant individual degree. This merely means that the query complexity increases by a factor of $O(\log n/\log\log n)$. Thus, we obtain $\mathcal{NP} \subseteq \mathcal{PCP}(\mathtt{log},q)$ for $q(n) \stackrel{\rm def}{=} O(\log^2 n)$.

Recall that, in order to use the latter PCP system in composition, we need to guarantee that it (or a version of it) is robust as well as to present a version that is a PCP of proximity. The latter version is relatively easy to obtain (using ideas as applied to the PCP of §9.3.2.1), whereas obtaining robustness is too complex to be described here. We comment that one way of obtaining a robust PCP system is by a generic application of a (randomness-efficient) "parallelization" of PCP systems (cf. [13]), which in turn depends heavily on highly efficient low-degree tests. A alternative approach (cf. [33]) capitalizes of the specific structure of the summation test (as well as on the actual robustness of the low-degree test).

Digest. Assuming that $\mathcal{P} \neq \mathcal{NP}$, the PCP Theorem asserts a PCP system that obtains simultaneously the minimal possible randomness and query complexity (up to a multiplicative factor). The forgoing construction obtains this remarkable result by combining two different PCPs: the first PCP obtains logarithmic randomness but uses polylogarithmically many queries, whereas the second PCP uses a constant number of queries but has polynomial randomness complexity. We stress that each of the two PCP systems is highly non-trivial and very interesting by itself. We highlight the fact that these PCPs can be composed using a very simple composition method that refers to auxiliary properties such as robustness and proximity testing. (Composition of PCP systems that lack these extra properties is possible, but is far more cumbersome and complex.)

9.3.2.3 Overview of the second proof of the PCP Theorem

The original proof of the PCP Theorem focuses on the construction of two PCP systems that are highly non-trivial and interesting by themselves, and combines them in a natural manner. Loosely speaking, this combination (via proof composition) *preserves* the good features of each of the two systems; that is, it yields a PCP system that inherits the (logarithmic) randomness complexity of one system and the (constant) query complexity of the other. In contrast, the following alternative proof is focused at the "amplification" of PCP systems, via a gradual process of logarithmically many steps. We start with a trivial "PCP" system that rejects false assertions with probability inversely proportional to their length, and double the rejection probability in each step. Furthermore, in each step, the constant query complexity is preserved and the length of the PCP oracle is increased only by a constant factor. Thus, the process gradually transforms an extremely weak PCP system into a remarkably strong PCP system as postulated in the PCP Theorem.

In order to describe the aforementioned process we need to redefine PCP systems so to allow arbitrary soundness error. In fact, for technical reasons it is more convenient to describe the process as an interated reduction of a "constraint satisfaction" problem to itself. Specifically, we refer to systems of 2-variable constraints, which are readily represented by (labelled) graphs such that the vertices correspond to (non-Boolean) variables and the edges are associated with constraints.

Definition 9.18 (CSP with 2-variable constraints): *For a fixed finite set Σ, an instance of* CSP *consists of a graph $G = (V, E)$* (which may have parallel edges and self-loops) *and a sequence of 2-variable constraints $\Phi = (\phi_e)_{e \in E}$ associated with the edges, where each constraint has the form $\phi_e : \Sigma^2 \to \{0, 1\}$. The value of an assignment $\alpha : V \to \Sigma$ is the number of constraints satisfied by α; that is, the value of α is $|\{(u, v) \in E : \phi_{(u,v)}(\alpha(u), \alpha(v)) = 1\}|$. We denote by $\mathtt{vlt}(G, \Phi)$ the fraction of unsatisfied constraints under the best possible assignment; that is,*

$$\mathtt{vlt}(G, \Phi) = \min_{\alpha : V \to \Sigma} \{|\{(u, v) \in E : \phi_{(u,v)}(\alpha(u), \alpha(v)) = 0\}| / |E|\} \tag{9.8}$$

For various functions $t : \mathbb{N} \to [0, 1]$, we will consider the promise problem $\mathtt{gapCSP}_t^{\Sigma}$, having instances as above, such that the yes-*instances are fully satisfiable instances* (i.e., $\texttt{vlt} = 0$) *and the* no-*instances are pairs* (G, Φ) *satisfying* $\texttt{vlt}(G, \Phi) \geq t(|G|)$, *where* $|G|$ *denotes the number of edges in* G.

Note that 3SAT is reducible to $\texttt{gapCSP}_t^{\{1,\dots,7\}}$ for $t(m) = 1/m$; see Exercise 9.15. Our goal is to reduce 3SAT (or rather $\texttt{gapCSP}_t^{\{1,\dots,7\}}$) to $\texttt{gapCSP}_c^{\Sigma}$, for some fixed finite Σ and constant $c > 0$. The PCP Theorem will follow by showing a simple PCP system for $\texttt{gapCSP}_c^{\Sigma}$; see Exercise 9.16. The desired reduction is obtained by iteratively applying the following reduction logarithmically many times.

Lemma 9.19 (amplifying reduction of gapCSP to itself): *For some finite* Σ *and constant* $c > 0$, *there exists a polynomial-time reduction of* $\texttt{gapCSP}^{\Sigma}$ *to itself such that the following conditions hold with respect to the mapping of any instance* (G, Φ) *to the instance* (G', Φ').

1. *If* $\texttt{vlt}(G, \Phi) = 0$ *then* $\texttt{vlt}(G', \Phi') = 0$.
2. $\texttt{vlt}(G', \Phi') \geq \min(2 \cdot \texttt{vlt}(G, \Phi), c)$.
3. $|G'| = O(|G|)$.

Proof Outline:[24] The reduction consists of three steps. We first apply a pre-processing step that makes the underlying graph suitable for further analysis. The value of vlt may decrease during this step by a constant factor. The heart of the reduction is the second step in which we may increase vlt by any desired constant factor. The latter step also increases the alphabet Σ, and thus a post-processing step is employed to regain the original alphabet (by using any inner PCP systems; e.g., the one presented in §9.3.2.1). Details follow.

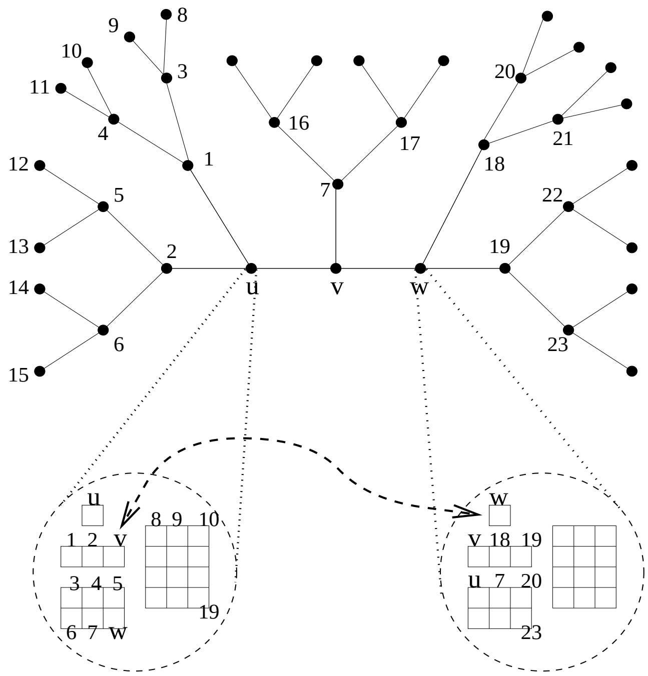
We first note that the aforementioned Σ and c, as well as the auxiliary parameters d and t, are fixed constants that will be determined so to satisfy various conditions that arise in the course of our argument.

We start with the pre-processing step. Our aim in this step is to reduce the input (G, Φ) of $\texttt{gapCSP}^{\Sigma}$ to an instance (G_1, Φ_1) such that G_1 is a d-regular expander graph.[25] Furthermore, each vertex in G_1 will have at least $d/2$ self-loops, $|G_1| = O(|G|)$, and $\texttt{vlt}(G_1, \Phi_1) = \Theta(\texttt{vlt}(G, \Phi))$. This step is quite simple: see Exercise 9.17.

The main step is aimed at increasing the fraction of violated constraints by a sufficiently large constant factor. This is done by reducing the instance (G_1, Φ_1) of $\texttt{gapCSP}^{\Sigma}$ to an instance (G_2, Φ_2) of $\texttt{gapCSP}^{\Sigma'}$ such that $\Sigma' = \Sigma^{d^t}$. Specifically, the vertex set of G_2 is identical to the vertex set of G_1, and each t-edge long path in G_1 is replaced by a corresponding edge in G_2, which is thus a d^t-regular graph. The constraints in Φ_2 are the natural ones, viewing each element of Σ' as a Σ-labelling of the ("distance $\leq t$") neigborhood of a vertex (see Figure 9.3), and checking that

[24] For details, see [63].

[25] A d-regular graph is a graph in which each vertex is incident to exactly d edges. Loosely speaking, an expander graph has the property that each moderately balanced cut (i.e., partition of its vertex set) has relatively many edges crossing it. An equivalent definition, also used in the actual analysis, is that the second eigenvalue of the corresponding adjacency matrix has asolute value that is bounded away from d.



The alphabet Σ' as a labelling of the distance $t = 3$ neighborhoods, when repetitions are omitted. In this case $d = 6$ but the self-loops are not shown. The two-sided arrow indicates one of the edges in G_1 that will contribute to the edge constraint between u and w in (G_2, Φ_2).

Figure 9.3: The amplifying reduction in the second proof of the PCP Theorem.

two such labellings are consistent as well as satisfy Φ_1. That is, suppose that there is a path of length at most t in G_1 going from vertex u to vertex w and passing through vertex v. Then, there is an edge in G_1 between vertices u and w, and the constraint associated with it with mandates that the entries correspondings to vertex v in the Σ'-labelling of vertices u and w are identical. In addition, if the G_1-edge (v, v') is on a path of length at most t starting at u then the corresponding edge in G_2 is associated a constraint that enforces the constraint that is associated to (v, v') in Φ_1.

Clearly, if $\texttt{vlt}(G_1, \Phi_1) = 0$ then $\texttt{vlt}(G_2, \Phi_2) = 0$. The interesting fact is that the fraction of violated constraints increases by a factor of $\Omega(\sqrt{t})$; that is, $\texttt{vlt}(G_2, \Phi_2) \geq \min(\Omega(\sqrt{t} \cdot \texttt{vlt}(G_1, \Phi_1)), c)$. Here we merely provide a rough intuition and refer the interested reader to [63]. The intuition is that any Σ'-labelling to the vertices of G_2 must either be consistent with a Σ-labelling of G_1 or violate the "equality constraints" of many edges in G_2. Focusing on the first case and relying on the hypothesis that G_1 is an expander, it follows that the set of violated edge-constaints (of Φ_1) with respect to the aforementioned Σ-labelling causes many more edge-constaints of Φ_2 to be violated (by virtue of the latter enforcing many edge-constaints of Φ_1). The point is that a set F of edges of G_1 is likely to appear on a $\min(\Omega(t) \cdot |F|/|G_1|, \Omega(1))$ fraction of the edges of G_2 (i.e., t-paths of G_1). (Note that the claim is obvious if G_1 were a complete graph, but it also holds for an expander.)[26]

The factor of $\Omega(\sqrt{t})$ gained in the second step makes up for the constant factor lost in the first step (as well as the constant factor to be lost in the last step). Furthermore, for a suitable choice of the constant t, the aforementioned gain yields an overall constant factor amplification (of $\mathtt{vlt}$). However, so far we obtained an instance of $\mathtt{gapCSP}^{\Sigma'}$ rather than an instance of $\mathtt{gapCSP}^{\Sigma}$, where $\Sigma' = \Sigma^{d^t}$. The purpose of the last step is to reduce the latter instance to an instance of $\mathtt{gapCSP}^{\Sigma}$. This is done by viewing the instance of $\mathtt{gapCSP}^{\Sigma'}$ as a (weak) PCP system and composing it with an inner-verifier, using the proof composition paradigm outlined in §9.3.2.2. We stress that the inner-verifier used here needs only handle instances of constant size (i.e., having description length $O(d^t \log|\Sigma|)$), and so the one presented in §9.3.2.1 will do. The resulting PCP-system uses randomness $r \stackrel{\text{def}}{=} \log_2|G_2| + O(d^t \log|\Sigma|)^2$ and a constant number of binary queries, and has rejection probability $\Omega(\mathtt{vlt}(G_2, \Phi_2))$, which is indepedent of the choice of the constant t. As in Exercise 9.15, for $\Sigma = \{0,1\}^{O(1)}$, we can easily obtain an instance of $\mathtt{gapCSP}^{\Sigma}$ that has a $\Omega(\mathtt{vlt}(G_2, \Phi_2))$ fraction of violated constraints. Furthermore, the size of the resulting instance is $O(2^r) = O(|G_2|)$, because d and t are constants. This completes the last step as well as the (outline of the) proof of the entire lemma. □

9.3.3 PCP and Approximation

The characterization of $\mathcal{NP}$ in terms of probabilistically checkable proofs plays a central role in the study of the complexity of approximation problems (cf., Section 10.1.1). To demonstrate this relationship, we first note that a PCP system V gives rise to a natural approximation problem; that is, on input x, the task is approximating the probability that V accepts x when given oracle access to the best possible π (i.e., we wish to approximate $\max_\pi\{\mathsf{Pr}[V^\pi(x) = 1]\}$). Thus, if $S \in \mathcal{PCP}(r,q)$ then deciding membership in S is reducible to approximating the maximum among $\exp(2^{r+q})$ quantities (corresponding to all effective oracles), where each quantity can be evaluated in time $2^r \cdot \mathrm{poly}$. Note that an approximation up to a constant factor (of 2) will do.

Let us take a closer look at the approximation problem associated with $\mathcal{PCP}(r,q)$. We focus, for simplicity, on the case of non-adaptive PCP systems (i.e., all the queries are determined beforehand by the input and the internal coin tosses of the verifier). On input x, we consider $2^{r(|x|)}$ formulae, each depending on $q(|x|)$ Boolean variables that represent the values of the corresponding bits in the proof oracle. Thus, if x is a yes-instance then there exists a truth assignment (to these variables) that satisfies all $2^{r(|x|)}$ formulae, whereas if x is a no-instance then there exists no

[26]We also note that due to a technical difficulty it is easier to establish the claimed bound of $\Omega(\sqrt{t} \cdot \mathtt{vlt}(G_1, \Phi_1))$ rather than $\Omega(t \cdot \mathtt{vlt}(G_1, \Phi_1))$.

truth assignment that satisfies more than $2^{r(|x|)-1}$ formulae. When focusing on the case that q is constant, this motivates the following definition.

Definition 9.20 (gap problems for SAT and generalized-SAT): *For constants $q \in \mathbb{N}$ and $\varepsilon > 0$, the promise problem* $\texttt{gapGSAT}_\varepsilon^q$ *consists of instances that are each a sequence of q-variable Boolean formulae. The* yes*-instances are sequences that are simultaneously satisfiable, whereas the* no*-instances are sequences for which no Boolean assignment satisfies more than a $1-\varepsilon$ fraction of the formulae in the sequence. The promise problem* $\texttt{gapSAT}_\varepsilon^q$ *is defined analogously, except that in this case each instance is a sequence of formulae that are each a single disjunctive clause.*

Indeed, each instance of $\texttt{gapSAT}_\varepsilon^q$ is naturally viewed as q-CNF formulae, and we consider an assignment that satisfies as many clauses (of the input CNF) as possible. As hinted, $\mathcal{NP} \subseteq \mathcal{PCP}(\texttt{log}, O(1))$ implies that $\texttt{gapGSAT}_{1/2}^{O(1)}$ is $\mathcal{NP}$-complete. The converse holds too, and (for some constant $\varepsilon > 0$) the same holds for $\texttt{gapSAT}_\varepsilon^3$.

Theorem 9.21 (equivalent formulations of the PCP Theorem). *The following three conditions are equivalent:*

1. The PCP Theorem: *there exists a constant q such that $\mathcal{NP} \subseteq \mathcal{PCP}(\texttt{log}, q)$.*
2. *There exists a constant q such that* $\texttt{gapGSAT}_{1/2}^q$ *is $\mathcal{NP}$-hard.*
3. *There exists a constant $\varepsilon > 0$ such that* $\texttt{gapSAT}_\varepsilon^3$ *is $\mathcal{NP}$-hard.*

Note that Items 2 and 3 make no reference to PCP. Instead, they manifest that the hardness of approximation lies at the heart of the PCP Theorem. In general, probabilistic checkable proof systems for $\mathcal{NP}$ yield strong inapproximability results for various classical optimization problems (cf., Exercise 9.10 and Section 10.1.1).

Proof: We first show that the PCP Theorem implies the NP-hardness of $\texttt{gapGSAT}$. We may assume, without loss of generality, that, for some constant q and every $S \in \mathcal{NP}$, it holds that $S \in \mathcal{PCP}(O(\log), q)$ via a non-adaptive verifier (because q adaptive queries can be emulated by 2^q non-adaptive queries). We reduce S to $\texttt{gapGSAT}$ as follows. On input x, we consider all $2^{O(\log|x|)}$ possible coin tosses of the verifier, and for each sequence of outcomes we determine the queries made as well as the residual decision predicate (of the PCP verifier). That is, for each random-outcome $\omega \in \{0,1\}^{O(\log|x|)}$, we consider the residual predicate determined by x and ω, and note that this predicate depends only on q variables (which represent the values of the q corresponding oracle answers). Thus, we obtain an instance of $\texttt{gapGSAT}^q$, and indeed $x \in S$ (resp., $x \notin S$) is mapped to a yes-instance (resp., no-instance) of $\texttt{gapGSAT}_{1/2}^q$.

Item 2 implies Item 3 by a standard reduction of GSAT to 3SAT. Specifically, $\texttt{gapGSAT}_{1/2}^q$ reduces to $\texttt{gapSAT}_{2^{-(q+1)}}^q$, which in turn reduces to $\texttt{gapSAT}_\varepsilon^3$ for $\varepsilon = 2^{-(q+1)}/(q-2)$. (Clearly, Item 3 implies Item 2.)

We complete the proof by showing that Item 3 implies Item 1. In fact, we show that $\texttt{gapGSAT}_\varepsilon^q$ is in $\mathcal{PCP}(O(\varepsilon^{-1}\log), O(q/\varepsilon))$, and do so by presenting a very natural PCP system. In this PCP system the proof oracle is supposed to be an satisfying assignment, and the verifier selects at random one of the (q-variable) formulae in the input sequence, and checks whether it is satisfied by the (assignment given by the) oracle. This amounts to tossing logarithmically many coins and making q queries. This verifier always accepts yes-instances (when given access to an adequate oracle), whereas each no-instances is rejected with probability at least ε (no matter which oracle is used). To amplify the rejection probability (to the desired threshold of 1/2), we invoke the foregoing verifier $O(\varepsilon^{-1})$ times. ■

Gap amplifying reductions – a reflection. Items 2 and 3 of Theorem 9.21 assert the existence of "gap amplifying" reductions of problems like 3SAT to themselves. These reductions map yes-instances to yes-instances (as usual), while mapping no-instances to no-instances of a special type such that a "gap" is created between the yes-instances and no-instances at the image of the reduction. For example, in the case of 3SAT, unsatisfiable formulae are mapped to formulae that are not merely unsatisfiable but rather have no assignment that satisfies more than a $1-\varepsilon$ fraction of the clauses. Thus, PCP constructions are essentially "gap amplifying" reductions.

9.3.4 More on PCP itself: an overview

We start by discussing variants of the PCP characterization of NP, and next turn to PCPs having expressing power beyond NP.

9.3.4.1 More on the PCP characterization of NP

Interestingly, the two complexity measures in the PCP-characterization of $\mathcal{NP}$ can be traded off such that at the extremes we get $\mathcal{NP} = \mathcal{PCP}(\texttt{log}, O(1))$ and $\mathcal{NP} = \mathcal{PCP}(0, \texttt{poly})$, respectively.

Proposition 9.22 *For every $S \in \mathcal{NP}$, there exist a constant α such that for every integer function ℓ that satisfies $0 \le \ell(n) \le \alpha \log_2 n$ it holds that $S \in \mathcal{PCP}(r,q) \subseteq \mathcal{NP}$, where $r(n) = \alpha \cdot \log_2 n - \ell(n)$ and $q(n) = O(2^{\ell(n)})$.*

Proof Sketch: Starting with Theorem 9.16, consider all possibilities for the $\ell(n)$-long prefix of the random tape of the verifier. □

Following the establishment of Theorem 9.16, numerous variants of the PCP Characterization of NP were explored. Following is a brief summary of some of these studies.[27]

[27] With the exception of works that appeared after [86], we provide no references for the results quoted here. We refer the interested reader to [86, Sec. 2.4.4].

The length of PCPs. Recall that the effective length of the oracle in any $\mathcal{PCP}(\mathtt{log},\mathtt{log})$ system is polynomial (in the length of the input). Furthermore, in the PCP systems underlying the proof of Theorem 9.16 the queries refer only to a polynomially long prefix of the oracle, and so the actual length of these PCPs for $\mathcal{NP}$ is polynomial. Remarkablly, the length of PCPs for $\mathcal{NP}$ can be made nearly-linear (in the combined length of the input and the standard NP-witness), while maintaining constant query complexity, where by nearly-linear we mean linear up to a poly-logarithmic factor. (For details see [34, 63].) This means that *the redundency required for supporting probabilistic verification of proofs is relatively modest.*

The number of queries in PCPs. Theorem 9.16 asserts that a constant number of queries suffice for PCPs with logarithmic randomness and soundness error 1/2 (for NP). It is currently known that this constant is at most 5, whereas with 3 queries one may get arbitrary close to error 1/2. The obvious trade-off between the number of queries and the soundness error gives rise to the robust notion of amortized query complexity, defined as the ratio between the number of queries and (minus) the logarithm (to based 2) of the soundness error. *For every $\varepsilon > 0$, any set in $\mathcal{NP}$ has a PCP system with logarithmic randomness and amortized query complexity* $1+\varepsilon$ (cf. [114]), whereas only sets in $\mathcal{P}$ have PCPs of logarithmic randomness and amortized query complexity 1 (or less).

The free-bit complexity. The motivation to the notion of free bits came from the PCP–to–MaxClique connection (see Exercise 9.10 and [27, Sec. 8]), but we believe that this notion is of independent interest. Intuitively, this notion distinguishes between queries for which the acceptable answer is determined by previously obtained answers (i.e., the verifier compares the answer to a value determined by the previous answers) and queries for which the verifier only records the answer for future usage. The latter queries are called free (because any answer to them is "acceptable"). For example, in the linearity test (see §9.3.2.1) the first two queries are free and the third is not (i.e., the test accepts if and only if $f(x) + f(y) = f(x + y)$). The amortized free-bit complexity is define analogously to the amortized query complexity. Interestingly, *$\mathcal{NP}$ has PCPs with logarithmic randomness and amortized free-bit complexity less than any positive constant.*

Adaptive versus non-adaptive verifiers. Recall that a PCP verifier is called non-adaptive if its queries are determined solely based on its input and the outcome of its coin tosses. (A general verifier, called adaptive, may determine its queries also based on previously received oracle answers.) Recall that the PCP Characterization of NP (i.e., Theorem 9.16) is established using a non-adaptive verifier; however, it turns out that *adaptive verifiers are more powerful than non-adaptive ones in terms of quantitative results*: Specifically, for PCP verifiers making three queries and having logarithmic randomness complexity, adaptive queries provide for soundness error at most 0.51 (actually $0.5 + \varepsilon$ for any $\varepsilon > 0$) for any set in $\mathcal{NP}$, whereas *non-adaptive* queries provide soundness error 5/8 (or less) only for sets in $\mathcal{P}$.

Non-binary queries. Our definition of PCP allows only binary queries. Certainly, non-binary queries can always be coded as binary ones, but the converse is not necessarily valid, in particular in adversarial settings. Note that the soundness condition constitutes an implicit adversarial setting, where a bad proof may be thought of as being selected by an adversary. Thus, when several binary queries are packed into one non-binary query, the adversary need not respect the packing (i.e., it may answer inconsistently on the same binary query depending on the other queries packed with it). For this reason, "parallel repetition" is highly non-trivial in the PCP setting. Still, a Parallel Repetition Theorem that refers to independent invocations of the same PCP is known [172], but it is not applicable for obtaining soundness error smaller than a constant (while preserving logarithmic randomness). Nevertheless, using adequate "consistency tests" one may construct PCP systems for $\mathcal{NP}$ using logarithmic randomness, a constant number of (non-binary) queries and *soundness error exponential in the length of the answers.* (Currently, this is known only for sub-logarithmic answer lengths.)

9.3.4.2 PCP with super-logarithmic randomness

The foregoing text has focused on the important case where the verifier tosses logarithmically many coins, and hence the "effective proof length" is polynomial. Here we mention that the PCP Theorem scales up.

Theorem 9.23 (Theorem 9.16 – Generalized): *Let $t(\cdot)$ be an integer function such that $n < t(n) < 2^{\mathrm{poly}(n)}$. Then,* $\mathrm{NTIME}(t) \subseteq \mathcal{PCP}(O(\log t), O(1))$.

Recall that $\mathcal{PCP}(r,q) \subseteq \mathrm{NTIME}(t)$, for $t(n) = \mathrm{poly}(n) \cdot 2^{r(n)}$.

Chapter Notes

(The following historical notes are quite long and still they fail to mention several important technical contributions that played an important role in the development of the area. For further details, the reader is referred to [86, Sec. 2.6.2].)

Motivated by the desire to formulate the most general type of "proofs" that may be used within cryptographic protocols, Goldwasser, Micali and Rackoff [105] introduced the notion of an *interactive proof system.* Although the main thrust of their work is the introduction of a special type of interactive proofs (i.e., ones that are *zero-knowledge*), the possibility that interactive proof systems may be more powerful from NP-proof systems has been pointed out in [105]. Independently of [105], Babai [16] suggested a different formulation of interactive proofs, which he called *Arthur-Merlin Games* (and conjectured to be "very close" to $\mathcal{NP}$). Syntactically, Arthur-Merlin Games are a restricted form of interactive proof systems, yet it was subsequently shown that these restricted systems are as powerful as the general ones (cf., [107]). The speed-up result (i.e., $\mathcal{AM}(2f) \subseteq \mathcal{AM}(f)$) is due to [20] (improving over [16]).

The first evidence of the power of interactive proofs was given by Goldreich, Micali, and Wigderson [96], who presented an interactive proof system for Graph

Non-Isomorphism (Construction 9.3). More importantly, they demonstrated the *generality and wide applicability of zero-knowledge proofs*: Assuming the existence of one-way function, they showed how to construct zero-knowledge interactive proofs for any set in $\mathcal{NP}$ (Theorem 9.11). This result has had a dramatic impact on the design of cryptographic protocols (cf., [97]). For further discussion of zero-knowledge and its applications to cryptography see Appendix C (and, in particular, Section C.4). Theorem 9.12 (i.e., $\mathcal{ZK} = \mathcal{IP}$) is due to [30, 122].

Probabilistically checkable proof (PCP) systems are related to *multi-prover interactive proof systems*, a generalization of interactive proofs that was suggested by Ben-Or, Goldwasser, Kilian and Wigderson [31]. Again, the main motivation came from the zero-knowledge perspective; specifically, introducing multi-prover zero-knowledge proofs for $\mathcal{NP}$ without relying on intractability assumptions. Yet, the complexity theoretic prospects of the new class, denoted $\mathcal{MIP}$, have not been ignored.

The amazing power of interactive proof systems has been demonstrated by using algebraic methods. The basic technique has been introduced by Lund, Fortnow, Karloff and Nisan [150], who applied it to show that the polynomial-time hierarchy (and actually $\mathcal{P}^{\#\mathcal{P}}$) is in $\mathcal{IP}$. Subsequently, Shamir [191] used the technique to show that $\mathcal{IP} = \mathcal{PSPACE}$, and Babai, Fortnow and Lund [17] used it to show that $\mathcal{MIP} = \mathcal{NEXP}$. (Our entire proof of Theorem 9.4 follows [191].)

The aforementioned multi-prover proof system of Babai, Fortnow and Lund [17] (hereafter referred to as the BFL proof system) has been the starting point for fundamental developments regarding $\mathcal{NP}$. The first development was the discovery that the BFL proof system can be "scaled-down" from $\mathcal{NEXP}$ to $\mathcal{NP}$. This important discovery was made independently by two sets of authors: Babai, Fortnow, Levin, and Szegedy [18] and Feige, Goldwasser, Lovász, and Safra [69]. However, the manner in which the BFL proof is scaled-down is different in the two papers, and so are the consequences of the scaling-down.

Babai *et. al.* [18] start by considering (only) inputs encoded using a special error-correcting code. The encoding of strings, relative to this error-correcting code, can be computed in polynomial time. They presented an almost-linear time algorithm that transforms NP-witnesses (to inputs in a set $S \in \mathcal{NP}$) into *transparent proofs* that can be verified (as vouching for the correctness of the encoded assertion) in (probabilistic) *poly-logarithmic time* (by a Random Access Machine). Babai *et. al.* [18] stress the practical aspects of transparent proofs; specifically, for rapidly checking transcripts of long computations. In contrast, in the proof system of Feige *et. al.* [69, 70] the verifier stays polynomial-time and only two more refined complexity measures (i.e., the randomness and query complexities) are reduced to poly-logarithmic. This eliminates the need to assume that the input is in a special error-correcting form, and yields a refinement of the complexity class introduced in [76], where the refinement is obtained by specifying the randomness and query complexities (see Definition 9.14). Hence, whereas the BFL proof system [17] can be reinterpreted as establishing

$$\mathcal{NEXP} = \mathcal{PCP}(\texttt{poly}, \texttt{poly}), \tag{9.9}$$

the work of Feige *et. al.* [70] establishes

$$\mathcal{NP} \subseteq \mathcal{PCP}(f, f), \quad \text{where } f(n) = O(\log n \cdot \log\log n). \tag{9.10}$$

We note that the work of Babai *et. al.* [18] implies

$$\mathcal{NP} \subseteq \mathcal{PCP}(\texttt{log}, \texttt{polylog}). \tag{9.11}$$

Interest in the new complexity class became immense since Feige *et. al.* [69, 70] demonstrated its relevance to proving the intractability of approximating some combinatorial problems (specifically, for MaxClique). When using the PCP–to–MaxClique connection established by Feige *et. al.*, the randomness and query complexities of the verifier (in a PCP system for an NP-complete set) relate to the strength of the negative results obtained for approximation problems. This fact provided a very strong motivation for trying to reduce these complexities and obtain a tight characterization of $\mathcal{NP}$ in terms of $\mathcal{PCP}(\cdot,\cdot)$. Once the work of Feige *et. al.* [70] had been presented, the challenge was clear: showing that $\mathcal{NP}$ equals $\mathcal{PCP}(\texttt{log}, \texttt{log})$. This challenge was met by Arora and Safra [14]. Actually, they showed that

$$\mathcal{NP} = \mathcal{PCP}(\texttt{log}, f(\cdot)), \qquad \text{where } f(n) = o(\log n). \tag{9.12}$$

Hence, a new challenge arose; namely, further reducing the query complexity – in particular to a constant – while maintaining the logarithmic randomness complexity. Again, additional motivation for this challenge came from the relevance of such a result to the study of approximation problems. The new challenge was met by Arora, Lund, Motwani, Sudan and Szegedy [13], and is captured by the equation

$$\mathcal{NP} = \mathcal{PCP}(\texttt{log}, O(1)). \tag{9.13}$$

Indeed the PCP Characterization Theorem (captured in Eq. (9.13)) is a culmination of a sequence of great works [150, 17, 18, 70, 14, 13]. These works are rich in innovative ideas (e.g., various arithmetizations of SAT as well as various forms of proof composition) and employ numerous techniques (e.g., low-degree tests, self-correction, and pseudorandomness). For further detail, the interested reader is referred to [86, Sec. 2.6.2].

Our overview of the original proof of the PCP Theorem (in §9.3.2.1–9.3.2.2) is based on [13, 14].[28] The alternative proof outlined in §9.3.2.3 is due to Dinur [63]. We also mention some of the ideas and techniques involved in deriving variants of the PCP Theorem. These include the Parallel Repetition Theorem [172], the use of the Long-Code [27], and the application of Fourier analysis in this setting [111, 112].

Computationally-Sound Proof Systems. Argument systems were defined by Brassard, Chaum and Crépeau [46], with the motivation of providing *perfect* zero-knowledge arguments (rather than zero-knowledge *proofs*) for $\mathcal{NP}$. A few years

[28] Our presentation also benefits from the notions of PCPs of proximity and robustness, put forward in [33, 64].

later, Kilian [135] demonstrated their significance beyond the domain of zero-knowledge by showing that, under some reasonable intractability assumptions, every set in $\mathcal{NP}$ has a computationally-sound proof in which the randomness and communication complexities are poly-logarithmic.[29] Interestingly, these argument systems rely on $\mathcal{NP} \subseteq \mathcal{PCP}(f, f)$, for $f(n) = \text{poly}(\log n)$. We mention that Micali [153] suggested a different type of computationally-sound proof systems (which he called CS-proofs).

Final comment: The current chapter is a revision of [86, Chap. 2]. In particular, more details are provided here for the main topics, whereas numerous secondary topics discussed in [86, Chap. 2] are not mentioned here (or are only briefly mentioned here). In addition, a couple of the research directions that were mentioned in [86, Sec. 2.4.4] received considerable attention in the period that elapsed, and improved results are currently known. In particular, the interested reader is referred to [33, 34, 63] (for a study of the length of PCPs) and to [114] (for a study of the amortized query complexity).

Exercises

Exercise 9.1 (parallel error-reduction for interactive proof systems) Prove that the error probability (in the soundness condition) can be reduced by parallel repetitions of the proof system.

Guideline: As a warm-up consider first the case of public-coin interactive proof systems. Next, note that the analysis generalizes to arbitrary interactive proof systems. (Extra hint: As a mental experiment, consider a "powerful verifier" that emulates the original verifier while behaining as in the public-coin model.) A proof appears in [86, Apdx. C.1].

Exercise 9.2 Complete the details of the proof that co$\mathcal{NP} \subseteq \mathcal{IP}$ (i.e., the first part of the proof of Theorem 9.4). In particular, regarding the proof of non-satisfiability of a CNF with n variables and m clauses, what is the length of the messages sent by the two parties? What is the soundness error?

Exercise 9.3 Present a $n/O(\log n)$-round interactive proof for the non-satisfiability of a CNF having n variables.
(Hint: Modify the (first part of the) proof of Theorem 9.4, by stripping $O(\log n)$ summations in each round.)

Exercise 9.4 Show that QBF can be reduced to a special form of QBF in which no variable appears both to the left and the right of more than one universal quantifier.

Guideline: Consider a process (which proceeds from left to right) of "refreshing" variables after each universal quantifier. Let $\phi(x_1, ..., x_s, y, x_{s+1}, ..., x_{s+t})$ be a quantifier-free boolean formula and let $Q_{s+1}, ..., Q_{s+t}$ be an arbitrary sequence of quantifiers. Then, we replace the quantified (sub-)formula

$$\forall y Q_{s+1} x_{s+1} \cdots Q_{s+t} x_{s+t}\, \phi(x_1, ..., x_s, y, x_{s+1}, ..., x_{s+t})$$

[29] We comment that interactive proofs are unlikely to have such low complexities; see [102].

by the (sub-)formula

$$\forall y \exists x'_1 \cdots \exists x'_s [(\wedge_{i=1}^{s}(x'_i = x_i)) \ \wedge \ Q_{s+1}x_{s+1} \ \cdots \ Q_{s+t}x_{s+t} \, \phi(x'_1, ..., x'_s, y, x_{s+1}, ..., x_{s+t})] .$$

Note that the variables $x_1, ..., x_s$ do not appear to the right of the quantifier Q_{s+1} in the replaced formula, and that the length of the replaced formula grows by an additive term of $O(s)$. This process of refreshing variables is applied from left to right on the entire sequence of universal quantifiers (except the inner one, for which this refreshing is useless).[30]

Exercise 9.5 (on interactive proofs with two-sided error (following [78])) Let $\mathcal{IP}'(f)$ denote the class of sets having a two-sided error interactive proof system in which a total of $f(|x|)$ messages are exchanged on common input x. Extending the ideas underlying the proof of Theorem 6.7, show that $\mathcal{IP}'(f) \subseteq \mathcal{IP}(f+3)$. Note that $\mathcal{IP}'(f) \subseteq \mathcal{AM}(f+1)$ follows.

Guideline: Consider the public-coin version of $\mathcal{IP}'$, denoted $\mathcal{AM}'$, and note that the proof of $\mathcal{IP}(f) \subseteq \mathcal{AM}(f+2)$ (cf. [107]) extends to $\mathcal{IP}'(f) \subseteq \mathcal{AM}'(f+2)$. Observe that the proof of Theorem 6.7 actually establishes that $\mathcal{BPP} \subseteq \mathcal{MA} = \mathcal{AM}(1)$, and that $\mathcal{BPP} = \mathcal{AM}'(0)$. Our aim is to establish $\mathcal{AM}'(f) \subseteq \mathcal{AM}(f+1)$ for arbitrary f (rather than for $f \equiv 0$). Consider an optimal prover strategy and the set of verifier coins that make the verifier accept any fixed yes-instance. Applying the ideas underlying the transformation of $\mathcal{BPP}$ to $\mathcal{MA}$, we obtain the desired result. For further details, see [78].

Exercise 9.6 In continuation to Exercise 9.5, show that $\mathcal{IP}'(f) = \mathcal{IP}(f)$ for every function $f : \mathbb{N} \to \mathbb{N}$ satisfying $f \geq 1$.
(Hint: If $f \geq 2$ then the claim follows by combining Exercise 9.5 with $\mathcal{IP}(O(f)) = \mathcal{IP}(f)$. For the case of $f = 1$, note that $\mathcal{IP}(1) = \mathcal{AM}(1)$, and that the single prover message added in Exercise 9.5 can be incorporated in the single message sent in $\mathcal{IP}(1)$.)

Exercise 9.7 (on the effective length of PCP oracles) Suppose that V is a PCP verifier of query complexity q and randomness complexity r. Show that for every fixed x, the number of possible locations examined on input x (when considering all possible internal coin tosses of V and all possible answers it may receive) is upper-bounded by $2^{q(|x|)+r(|x|)}$. Show that if V is non-adaptive then the upper-bound can be improved to $2^{r(|x|)} \cdot q(|x|)$.
(Hint: In the adaptive case, the i^{th} query is determined by V's internal coin tosses and the previous $i-1$ answers. In the non-adaptive case, all q queries are determined by V's internal coin tosses.)

---

[30] For example,

$$\exists z_1 \forall z_2 \exists z_3 \forall z_4 \exists z_5 \forall z_6 \, \phi(z_1, z_2, z_3, z_4, z_5, z_6)$$

is first replaced by

$$\exists z_1 \forall z_2 \exists z'_1 \, [(z'_1 = z_1) \ \wedge \ \exists z_3 \forall z_4 \exists z_5 \forall z_6 \, \phi(z'_1, z_2, z_3, z_4, z_5, z_6)]$$

and next (written as $\exists z_1 \forall z'_2 \exists z'_1 \, [(z'_1 = z_1) \ \wedge \ \exists z'_3 \forall z'_4 \exists z'_5 \forall z'_6 \, \phi(z'_1, z'_2, z'_3, z'_4, z'_5, z'_6)]$) is replaced by

$$\begin{aligned} &\exists z_1 \forall z'_2 \exists z'_1 \, [(z'_1 = z_1) \ \wedge \ \exists z'_3 \forall z'_4 \exists z''_1 \exists z''_2 \exists z''_3 \\ &\quad [(\wedge_{i=1}^{3}(z''_i = z'_i)) \ \wedge \ \exists z'_5 \forall z'_6 \phi(z''_1, z''_2, z''_3, z'_4, z'_5, z'_6)]]. \end{aligned}$$

Thus, in the resulting formula, no variable appears both to the left and to the right of more than a single universal quantifier.

Exercise 9.8 (upper-bounds on the complexity of PCPs) Suppose that a set S has a PCP of query complexity q and randomness complexity r. Show that S can be decided by a non-deterministic machine that, on input of length n, makes at most $2^{r(n)} \cdot q(n)$ non-deterministic[31] steps and halts within a total number of $2^{r(n)} \cdot \text{poly}(n)$ steps. Thus, $S \in \text{NTIME}(2^r \cdot \text{poly}) \cap \text{DTIME}(2^{2^r q} \cdot \text{poly})$.

(Hint: For each input $x \in S$ and each possible value $\omega \in \{0,1\}^{r(|x|)}$ of the random-tape, we consider a sequence of $q(|x|)$ bit values that represent a sequence of oracle answers that make the verifier accept. Indeed, for fixed x and $\omega \in \{0,1\}^{r(|x|)}$, each setting of the $q(|x|)$ oracle answers determine the computation of the corresponding verifier (including the queries it makes).)

Exercise 9.9 (on the effective randomness of PCPs) Suppose that a set S has a PCP of query complexity q that utilizes proof oracles of length ℓ. Show that, for every constant $\varepsilon > 0$, the set S has a "non-uniform" PCP of query complexity q, soundness error $0.5 + \varepsilon$ and randomness complexity r such that $r(n) = O(1) + \log_2(\ell(n) + n)$. By a "non-uniform PCP" we mean one in which the verifier is a probabilistic polynomial-time oracle machine that gets a non-uniform advice of length polynomial in ℓ.

Guideline: Consider a PCP verifier V as in the hypothesis, and denote its randomness complexity by r_V. We construct a non-uniform verifier V' that, on input of length n, obtains as advice a set $R_n \subseteq \{0,1\}^{r_V(n)}$ of cardinality $O((\ell(n) + n)/\varepsilon^2)$, and emulates V on a uniformly selected element of R_n. Show that for a random R_n, the verifier V' satisfies the claims of the exercise.

(Extra hint: Fixing any input $x \notin S$ and any oracle $\pi \in \{0,1\}^{\ell(|x|)}$, upper-bound the probability that a random set R_n causes V' to accept x with probability $0.5 + \varepsilon$ when using the oracle π.)

Exercise 9.10 (The FGLSS-reduction [70]) For any $S \in \mathcal{PCP}(r, q)$, consider the following mapping of instances for S to instances of the `Independent Set` problem. The instance x is mapped to a graph $G_x = (V_x, E_x)$, where $V_x \subseteq \{0,1\}^{r(|x|)+q(|x|)}$ consists of pairs (ω, α) such that the PCP verifier *accepts* the input x, when using coins $\omega \in \{0,1\}^{r(|x|)}$ and receiving the answers $\alpha = \alpha_1 \cdots \alpha_{q(|x|)}$ (to the oracle queries determined by x, r and the previous answers). The set E_x consists of edges that connect vertices that represents inconsistent view of the said verifier; that is, the vertex $v = (\omega, \alpha_1 \cdots \alpha_{q(|x|)})$ is connected to the vertex $v' = (\omega', \alpha'_1 \cdots \alpha'_{q(|x|)})$ if there exists i and i' such that $\alpha_i \neq \alpha'_{i'}$ and $\mathsf{q}_i^x(v) = \mathsf{q}_{i'}^x(v')$, where $\mathsf{q}_i^x(v)$ (resp., $\mathsf{q}_{i'}^x(v')$) denotes the i-th (resp., i'-th) query of the verifier on input x, when using coins ω (resp., ω') and receiving the answers $\alpha_1 \cdots \alpha_{i-1}$ (resp., $\alpha'_1 \cdots \alpha'_{i'-1}$). In particular, for every $\omega \in \{0,1\}^{r(|x|)}$ and $\alpha \neq \alpha'$, if $(\omega, \alpha), (\omega, \alpha') \in V_x$, then $((\omega, \alpha), (\omega, \alpha')) \in E_x$.

1. Prove that the mapping $x \mapsto G_x$ can be computed in time that is polynomial in $2^{r(|x|)+q(|x|)} \cdot |x|$.

   (Note that the number of vertices in G_x equals $2^{r(|x|)+f(|x|)}$, where f is the free-bit complexity of the PCP verifier.)

2. Prove that, for every x, the size of the maximum independent set in G_x is at most $2^{r(|x|)}$.

[31] See §4.2.1.3 for definition of non-deterministic machines.

3. Prove that if $x \in S$ then G_x has an independent set of size $2^{r(|x|)}$.

4. Prove that if $x \notin S$ then the size of the maximum independent set in G_x is at most $2^{r(|x|)-1}$.

In general, denoting the PCP verifier by V, prove that the size of the maximum independent set in G_x is exactly $2^{r(|x|)} \cdot \max_\pi\{\Pr[V^\pi(x) = 1]\}$. (Note the similarity to the proof of Proposition 2.25.) Show that the PCP Theorem implies that the size of the maximum independent set (resp., clique) in a graph is NP-hard to approximate to within any constant factor.

(Hint: Note that an independent set in G_x corresponds to a set of coins R and a partial oracle π' such that V accepts x when using coins in R and accessing any oracle that is consistent with π'. The FGLSS reduction creates a gap of a factor of 2 between yes and no-instances of S. Larger factors can be obtained by considering a PCP that results from repeating the given PCP for a constant number of times. The result for `Clique` follows by considering the complement graph.)

Exercise 9.11 In continuation to Exercise 9.10, prove that, for any $t(n) = o(\log n)$, it holds that $\mathcal{NP} \subseteq \mathcal{PCP}(t,t)$ implies that $\mathcal{P} = \mathcal{NP}$.

Guideline: We only use the fact that the said reduction actually reduces PCP to instances of the `Clique` problem (and ignore the fact that we actually get a stronger result that refers to a "gapClique" problem). Furthermore, when applies to problems in $\mathcal{NP} \subseteq \mathcal{PCP}(t,t)$, this reduction runs in polynomial-time. The key observation is that this reduction maps instances of the `Clique` problem (which is in $\mathcal{NP} \subseteq \mathcal{PCP}(o(\log), o(\log))$) to shorter instances of the same problem (because $2^{o(\log n)} \ll n$). Thus, iteratively applying the reduction, we can reduce instances of `Clique` to instances of constant size. Using the $\mathcal{NP}$-completeness of `Clique`, we reduce $\mathcal{NP}$ to a finite set, and $\mathcal{NP} = \mathcal{P}$ follows.

Exercise 9.12 (a simple but partial analysis of the BLR Linearity Test) For Abelian groups G and H, consider functions from G to H. For such a (generic) function f, consider the linearity (or rather homomorphism) test that selects uniformly $r, s \in G$ and checks that $f(r)+f(s) = f(r+s)$. Let $\delta(f)$ denote the distance of f from the set of homomorphisms (of G to H); that is, $\delta(f)$ is the minimum taken over all homomorphisms $h : G \to H$ of $\Pr_{x\in G}[f(x) \neq h(x)]$. Using the following guidelines, prove that the probability that the test rejects f, denoted $\varepsilon(f)$, is at least $3\delta(f) - 6\delta(f)^2$.

1. Suppose that h is the homomorphism closest to f (i.e., $\delta(f) = \Pr[f(x) \neq h(x)]$). Prove that $\varepsilon(f) = \Pr_{x,y\in G}[f(x) + f(y) \neq f(x,y)]$ is lower-bounded by $3 \cdot \Pr_{x,y}[f(x) \neq h(x) \wedge f(y) = h(y) \wedge f(x,y) = h(x+y)]$.

   (Hint: consider three out of four disjoint events that support the case $f(x)+f(y) \neq f(x,y)$.)

2. Prove that $\Pr_{x,y}[f(x) \neq h(x) \wedge f(y) = h(y) \wedge f(x,y) = h(x+y)] \geq \delta(f) - 2\delta(f)^2$.

   (Hint: lower-bound the said probability by $\Pr_{x,y}[f(x) \neq h(x)] - (\Pr_{x,y}[f(x) \neq h(x) \wedge f(y) \neq h(y)] + \Pr_{x,y}[f(x) \neq h(x) \wedge f(x+y) \neq h(x+y)])$.)

Note that the lower-bound $\varepsilon(f) \geq 3\delta(f) - 6\delta(f)^2$ increases with $\delta(f)$ only in the case that $\delta(f) \leq 1/4$. Furthermore, the lower-bound is useless in the case that

$\delta(f) \geq 1/2$. Thus an alternative lower-bound is needed in case $\delta(f)$ approaches $1/2$ (or is larger than it); see Exercise 9.13.

Exercise 9.13 (a better analysis of the BLR Linearity Test (cf. [38])) In continuation to Exercise 9.12, use the following guidelines in order to prove that $\varepsilon(f) \geq \min(1/7, \delta(f)/2)$. Specifically, focusing on the case that $\varepsilon(f) < 1/7$, show that f is $2\varepsilon(f)$-close to some homomorphism (and thus $\varepsilon(f) \geq \delta(f)/2$).

1. Define the vote of y regarding the value of f at x as $\phi_y(x) \stackrel{\text{def}}{=} f(x+y)-f(y)$, and define $\phi(x)$ as the corresponding plurality vote (i.e., $\phi(x) \stackrel{\text{def}}{=} \text{argmax}_{v\in H}\{|\{y \in G : \phi_y(x)=v\}|\}$).

   Prove that, for every $x \in G$, it holds that $\mathsf{Pr}_y[\phi_y(x) = \phi(x)] \geq 1 - 2\varepsilon(f)$.

   Extra guideline: Fixing x, call a pair (y_1, y_2) good if $f(y_1) + f(y_2 - y_1) = f(y_2)$ and $f(x+y_1)+f(y_2-y_1) = f(x+y_2)$. Prove that, for any x, a random pair (y_1, y_2) is good with probability at least $1 - 2\varepsilon(f)$. On the other hand, for a good (y_1, y_2), it holds that $\phi_{y_1}(x) = \phi_{y_2}(x)$. Show that the graph in which edges correspond to good pairs must have a connected component of size at least $(1 - 2\varepsilon(f)) \cdot |G|$. Note that $\phi_y(x)$ is identical for all vertices y in this connected component, which in turn contains a majority of all y's in G.

2. Prove that ϕ is a homomorphism; that is, prove that, for every $x, y \in G$, it holds that $\phi(x) + \phi(y) = \phi(x + y)$.

   Extra guideline: Prove that $\phi(x) + \phi(y) = \phi(x + y)$ holds by considering the somewhat ficticious expression $\mathsf{Pr}_{r\in G}[\phi(x) + \phi(y) \neq \phi(x + y)]$, and showing that it is structly smaller than 1 (and hence $\phi(x)+\phi(y) \neq \phi(x+y)$ is false). Upper-bound the probabilistic expression by

   $$\mathsf{Pr}_r[\phi(x)\neq f(x + r) - f(r) \vee \phi(y)\neq f(r) - f(r - y) \vee \phi(x + y)\neq f(x + r) - f(r - y)].$$

   Use the union bound (and Item 1), and note that $\mathsf{Pr}_r[\phi(x) \neq f(x + r) - f(r)] < 2\varepsilon(f) < 1/3$, whereas $\mathsf{Pr}_r[\phi(y) \neq f(r) - f(r - y)] = \mathsf{Pr}_{r'}[\phi(y) \neq f(y + r') - f(r')]$ and $\mathsf{Pr}_r[\phi(x + y) \neq f(x + r) - f(r - y)] = \mathsf{Pr}_{r'}[\phi(x + y) \neq f(x + y + r') - f(r')]$ (by substituting $r' = r - y$).

3. Prove that f is $2\varepsilon(f)$-close to ϕ.

   Extra guideline: By Item 1, for every x, it holds that $\mathsf{Pr}_y[f(x+y)-f(y) = \phi(x)] \geq 1-2\varepsilon(f)$. Thus, there exists $y \in G$ such that $\mathsf{Pr}_x[f(x+y)-f(y) = \phi(x)] \geq 1-2\varepsilon(f)$, and for $z = x + y$ we have $\mathsf{Pr}_z[f(z) - f(y) = \phi(z - y)] \geq 1 - 2\varepsilon(f)$. Using Item 2 and letting $\eta \stackrel{\text{def}}{=} f(y) - \phi(y)$, we have $\mathsf{Pr}_z[f(z) = \phi(z) + \eta] \geq 1 - 2\varepsilon(f)$. Prove that $\mathsf{Pr}_{x,y}[(\phi(x) + \eta) + (\phi(y) + \eta) = \phi(x + y) + \eta] \geq 1 - 7\varepsilon(f) > 0$, using $\varepsilon(f) < 1/7$, $\mathsf{Pr}_{x,y}[f(x)+f(y) = f(x+y)] = 1-\varepsilon(f)$, and $\mathsf{Pr}_x[f(x) = \phi(x)+\eta] \geq 1-2\varepsilon(f)$ (and the analogous statements for y and $x + y$). Using Item 2, $\eta = 0$ follows.

We comment that better bounds on the behavior of $\varepsilon(f)$ as a function of $\delta(f)$ are known.

Exercise 9.14 Let M be a non-zero m-by-n matrix over $\mathrm{GF}(p)$. Prove that $\mathsf{Pr}_{r,s}[r^\top Ms \neq 0] \geq (1-p^{-1})^2$, where r (resp., s) is a random m-ary (resp., n-ary) vector.

Guideline: Prove that if $v \neq 0^m$ then $\mathsf{Pr}_s[v^\top s = 0] = p^{-1}$, and that if M has rank ρ then $\mathsf{Pr}_r[r^\top M = 0^n] = p^{-\rho}$.

Exercise 9.15 Show that 3SAT is reducible to $\mathtt{gapCSP}_t^{\{1,\ldots,7\}}$ for $t(m) = 1/m$, where gapCSP is as in Definition 9.18. Furthermore, show that the size of the resulting gapCSP instance is linear in the length of the input formula.

Guideline: Given an instance ψ of 3SAT, consider the graph in which vertices correspond to clauses of ψ, edges correspond to pairs of clauses that share a variable, and the constraints represent the natural consistency condition regarding partial assignments that satisfy the clauses. See a similar construction in Exercise 9.10.

Exercise 9.16 Show that, for any fixed finite Σ and constant $c > 0$, the problem $\mathtt{gapCSP}_c^\Sigma$ is in $\mathcal{PCP}(\mathtt{log}, O(1))$.

Guideline: Consider an oracle that, for some satisfying assignment for the CSP-instance (G, Φ), provides a trivial encoding of the assignment; that is, for a satisfying assignment $\alpha : V \to \Sigma$, the oracle responds to the query (v, i) with the i^{th} bit in the binary representation of $\alpha(v)$. Consider a verifier that uniformly selects an edge (u, v) of G and checks the constraint $\phi_{(u,v)}$ when applied to the values $\alpha(u)$ and $\alpha(v)$ obtained from the oracle.

Exercise 9.17 For any constant Σ and $d \geq 14$, show that $\mathtt{gapCSP}^\Sigma$ can be reduced to itself such that the instance at the target of the reduction is a d-regular expander, and the fraction of violated constraints is preserved up to a constant factor. That is, the instance (G, Φ) is reduced to (G_1, Φ_1) such that G_1 is a d-regular expander graph and $\mathtt{vlt}(G_1, \Phi_1) = \Theta(\mathtt{vlt}(G, \Phi))$. Furthermore, make sure that $|G_1| = O(|G|)$ and that each vertex in G_1 has at least $d/2$ self-loops.

Guideline: First, replace each vertex of degree $d' > 3$ by a 3-regular expander of size d', and connect each of the original d' edges to a different vertex of this expander, thus obtaining a graph of maximum degree 4. Maintain the constraints associated with the original edges, and associate the equality constraint (i.e., $\phi(i, j) = 1$ if and only if $i = j$) to each new edge (residing in any of the added expanders). Next, denoting the number of vertices in the resulting graph by N_1, add to this graph a 3-regular expander of size N_1 (while associating with these edges the trivially satisfied constraint $\phi(i, j) = 1$ for all $i, j \in \Sigma$). Finally, add at least $d/2$ self-loops to each vertex (using again trivially satisfied constraints), so to obtain a d-regular graph. Prove that this sequence of modifications may only decrease the fraction of violated constraints, and that the decrease is only by a constant factor. The latter assertion relies on the equality constraints associated with the small expanders used in the first step.

Exercise 9.18 (free bit complexity zero) Note that only sets in $\mathcal{BPP}$ have PCPs of query complexity zero. Furthermore, Exercise 9.8 implies that only sets in $\mathcal{P}$ have PCP systems of logarithmic randomness and query complexity zero.

1. Show that only sets in $\mathcal{P}$ have PCP systems of logarithmic randomness and free-bit complexity zero.

   (Hint: Consider an application of the FGLSS-reduction to a set having a PCP of free-bit complexity zero.)

2. In contrast, show that Graph Non-Isomorphism has a PCP system of free-bit complexity zero (and linear randomness complexity).

Exercise 9.19 (free bit complexity one) In continuation to Exercise 9.18, prove that only sets in $\mathcal{P}$ have PCP systems of logarithmic randomness and free-bit complexity one.

Guideline: Consider an application of the FGLSS-reduction to a set having a PCP of free-bit complexity one and randomness complexity r. Note that the question of whether the resulting graph has an independent set of size 2^r can be expressed as a 2CNF formula of size poly(2^r), and see Exercise 2.21.

Chapter 10

Relaxing the Requirements

The philosophers have only interpreted the world, in various ways; the point is to change it.

Karl Marx, Theses on Feuerbach

In light of the apparent infeasibility of solving numerous natural computational problems, it is natural to ask whether these problems can be relaxed in a way that is both useful for applications and allows for feasible solving procedures. We stress two aspects about the foregoing question. On one hand, the relaxation should be sufficiently good for the intended applications; but, on the other hand, it should be significantly different from the original version so to escape the infeasibility of the latter. We note that whether a relaxation is adequate for an intended application depends on the application, and thus much of the material in this chapter is less robust (or generic) than the treatment of the non-relaxed computational problems.

Summary: We consider two types of relaxations. The first type of relaxation refers to the computational problems themselves; that is, for each problem instance we extend the set of admissible solutions. In the context of search problems this means settling for solutions that have a value that is "sufficiently close" to the value of the optimal solution (with respect to some value function). Needless to say, the specific meaning of 'sufficiently close' is part of the definition of the relaxed problem. In the context of decision problems this means that for some instances both answers are considered valid; put differently, we consider promise problems in which the no-instances are "far" from the yes-instances in some adequate sense (which is part of the definition of the relaxed problem).

The second type of relaxation deviates from the requirement that the solver provides an adequate answer on each valid instance. Instead, the behavior of the solver is analyzed with respect to a specific input distribution, and bad behavior may occur with negligible probability

where the probability is taken over the input distribution. Needless to say, a major component in this approach is limiting the class of distributions in a way that, on one hand, allows for various types of natural distributions and, on the other hand, prevents the collapse of average-case complexity to worst-case complexity.

10.1 Approximation

On the difficulty of being perfect and the ease of being almost perfect

Dana Ron, title of talk at Radcliffe (2003)

The notion of approximation is a natural one, and has arisen also in other disciplines. Its most common use is in references to quantities (e.g., the length of one meter is approximately forty inches), but it is also used when referring to qualities (e.g., an approximately correct account of a historical event). In the context of computation, the notion of approximation modifies computational tasks such as search and decision problems. (In fact, we have encountered it already as a modifier of counting problems; see Section 6.2.2.)

Two major questions regarding approximation are (1) what is a "good" approximation, and (2) can it be found easier than finding an exact solution. The answer to the first question seems intimately related to the specific computational task at stake and to its role in the wider context (i.e., the higher level application): a good approximation is one that suffices for the intended application. Indeed, the importance of certain approximation problems is much more subjective than the importance of the corresponding optimization problems. This fact seems to stand in the way of attempts at providing a comprehensive theory of approximation problems (e.g., classes of natural approximation problems that are shown to be computationally equivalent).

Turning to the second question, we note that in numerous cases natural approximation problems seem to be significantly easier than the corresponding original ("exact") problems. On the other hand, in numerous other cases, natural approximation problems are computationally equivalent to the original problems. We shall exemplify both cases by reviewing some specific results, but regret not being able to provide any systematic classification.

Mimicking the two standard uses of the word *approximation*, we shall distinguish between approximation problems that are of the "search type" and problems that are have a clear "decisional" flavor. In the first case we shall refer to a function that assigns values to possible solutions (of a search problem); whereas in the second case we shall refer to distances between instances (of a decision problem). Needless to say, at times the same computational problem may be cast in both ways, but for most natural approximation problems one of the two frameworks is more appealing than the other.

Teaching note: Most of the results presented in this section refer to specific computational problems and (with one exception) are presented without a proof. In view of the complexity of the corresponding proofs and the merely illustrative role of these results, we recommend doing the same in class.

10.1.1 Search or Optimization

As noted in Section 2.2.2, many search problems involve a set of potential solutions (per each problem instance) such that different solutions are assigned different "values" (resp., "costs") by some "value" (resp., "cost") function. In such a case, one is interested in finding a solution of maximum value (resp., minimum cost). A corresponding approximation problem may refer to finding a solution of approximately maximum value (resp., approximately minimum cost), where the specification of the desired level of approximation is part of the problem's definition. Let us elaborate.

For concreteness, we focus on the case of a value that we wish to maximize. For greater flexibility, we allow the value of the solution to depend also on the instance itself. Thus, for a (polynomially bounded) binary relation R and a *value function* $f:\{0,1\}^*\times\{0,1\}^*\to\mathbb{R}$, we consider the problem of finding solutions (with respect to R) that maximize the value of f. That is, given x (such that $R(x)\neq\emptyset$), the task is finding $y\in R(x)$ such that $f(x,y)=v_x$, where v_x is the maximum value of $f(x,y')$ over all $y'\in R(x)$. Typically, R is in $\mathcal{PC}$ and f is polynomial-time computable.[1] Indeed, without loss of generality, we may assume that for every x it holds that $R(x)=\{0,1\}^{\ell(|x|)}$ for some polynomial ℓ (see Exercise 2.8). Thus, the optimization problem is recast as *given x, find y such that $f(x,y)=v_x$, where $v_x=\max_{y'\in\{0,1\}^{\ell(|x|)}}\{f(x,y')\}$*.

We shall focus on relative approximation problems, where for some gap function $g:\{0,1\}\to\{r\in\mathbb{R}:r\geq 1\}$ the task is finding y such that $f(x,y)\geq v_x/g(x)$. Indeed, often the approximation factor is stated in terms of some structure of the input (e.g., the number of vertices in a graph) and not merely in terms of its length. Typically, g is polynomial-time computable. Approximation versions of minimization problems are defined analogously.

Definition 10.1 (g-factor approximation): *Let $f:\{0,1\}^*\times\{0,1\}^*\to\mathbb{R}$, $\ell:\mathbb{N}\to\mathbb{N}$, and $g:\{0,1\}\to\{r\in\mathbb{R}:r\geq 1\}$.*

Maximization version: *The g-factor approximation of maximizing f (w.r.t ℓ) is the search problem R such that $R(x)=\{y\in\{0,1\}^{\ell(|x|)}:f(x,y)\geq v_x/g(x)\}$, where $v_x=\max_{y'\in\{0,1\}^{\ell(|x|)}}\{f(x,y')\}$.*

Minimization version: *The g-factor approximation of minimizing f (w.r.t ℓ) is the search problem R such that $R(x)=\{y\in\{0,1\}^{\ell(|x|)}:f(x,y)\leq g(x)\cdot c_x\}$, where $c_x=\min_{y'\in\{0,1\}^{\ell(|x|)}}\{f(x,y')\}$.*

[1] In this case, we may assume without loss of generality that the function f depends only on the solution. This can be obtained by redefining the relation R such that each solution $y\in R(x)$ consists of a pair of the form (x,y'). Needless to say, this modification cannot be applied along with getting rid of R (as in Exercise 2.8).

We note that for numerous NP-complete optimization problems polynomial-time algorithms provide meaningful approximations. A few examples will be mentioned in §10.1.1.1. In contrast, for numerous other NP-complete optimization problems, natural approximation problems are computationally equivalent to the corresponding optimization problem. A few examples will be mentioned in §10.1.1.2, where we also introduce the notion of a *gap problem*, which is a promise problem (of the decision type) intended to capture the difficulty of the (approximate) search problem.

10.1.1.1 A few positive examples

Let us start with a trivial example. Considering a problem such as finding the maximum clique in a graph, we note that finding a linear factor approximation is trivial (i.e., given a graph $G = (V, E)$, we may output any vertex in V as a $|V|$-factor approximation of the maximum clique in G). A famous non-trivial example is presented next.

Proposition 10.2 (factor two approximation to `minimum Vertex Cover`): *There exists a polynomial-time approximation algorithm that given a graph $G = (V, E)$ outputs a vertex cover that is at most twice as large as the minimum vertex cover of G.*

We warn that an approximation algorithm for `minimum Vertex Cover` does not yield such an algorithm for the complementary problem (of `maximum Independent Set`). This phenomenon stands in contrast to the case of optimization, where an optimal solution for one problem (e.g., `minimum Vertex Cover`) yields an optimal solution for the complementary problem (`maximum Independent Set`).

Proof Sketch: The main observation is a connection between the set of maximal matchings and the set of vertex covers in a graph. Let M be any maximal matching in the graph $G = (V, E)$. Then, on one hand, the set of all vertices participating in M is a vertex cover of G, and, on the other hand, each vertex cover of G must contain at least one vertex of each edge of M. Thus, we can find the desired vertex cover by finding a maximal matching, which in turn can be found by a greedy algorithm. □

Another example. An instance of the traveling salesman problem (TSP) consists of a matrix of distances between pairs of points and the task is finding a shortest tour that passes through all points. In general, no reasonable approximation is feasible for this problem (see Exercise 10.1), but here we consider two special cases in which the distances satisfies some natural constraints.

Theorem 10.3 (approximations to special cases of TSP): *Polynomial-time algorithms exists for the following two cases.*

1. *Providing a 1.5-factor approximation for the special case of TSP in which the distances satisfy the triangle inequality.*

2. *For every $\varepsilon > 1$, providing a $(1+\varepsilon)$-factor approximation for the special case of Euclidean TSP* (i.e., the distances correspond to a k-dimensional Euclidean space, for some constant k (e.g., $k = 2$)).

A weaker version of Part 1 is given in Exercise 10.2. A detailed survey of Part 2 is provided in [12].

10.1.1.2 A few negative examples

Let us start again with a trivial example. Considering a problem such as finding the maximum clique in a graph, we note that given a graph $G = (V, E)$ finding a $(1 + |V|^{-1})$-factor approximation of the maximum clique in G is as hard as finding a maximum clique in G. Indeed, this "result" is not really meaningful. In contrast, building on the PCP Theorem (Theorem 9.16), one may prove that finding a $|V|^{1-o(1)}$-factor approximation of the maximum clique in G is as hard as finding a maximum clique in G. This follows from the fact that the approximation problem is NP-hard (cf. Theorem 10.5).

The statement of inapproximability results is made stronger by referring to a promise problem that consists of distinguishing instances of sufficiently far apart values. Such promise problems are called gap problems, and are typically stated with respect to two bounding functions $g_1, g_2 : \{0,1\} \to \mathbb{R}$ (which replace the gap g of Definition 10.1). Typically, g_1 and g_2 are polynomial-time computable.

Definition 10.4 (gap problem for approximation of f): *Let f be as in Definition 10.1 and $g_1, g_2 : \{0,1\} \to \mathbb{R}$.*

Maximization version: *For $g_1 \geq g_2$, the* gap_{g_1,g_2} problem of maximizing f *consists of distinguishing between $\{x : v_x \geq g_1(x)\}$ and $\{x : v_x < g_2(x)\}$, where $v_x = \max_{y \in \{0,1\}^{\ell(|x|)}}\{f(x,y)\}$.*

Minimization version: *For $g_1 \leq g_2$, the* gap_{g_1,g_2} problem of minimizing f *consists of distinguishing between $\{x : c_x \leq g_1(x)\}$ and $\{x : c_x > g_2(x)\}$, where $c_x = \min_{y \in \{0,1\}^{\ell(|x|)}}\{f(x,y)\}$.*

For example, the gap_{g_1,g_2} problem of maximizing the size of a clique in a graph consists of distinguishing between graphs G that have a clique of size $g_1(G)$ and graphs G that have no clique of size $g_2(G)$. In this case, we typically let $g_i(G)$ be a function of the number of vertices in $G = (V, E)$; that is, $g_i(G) = g'_i(|V|)$. Indeed, letting $\omega(G)$ denote the size of the largest clique in the graph G, we let $\mathtt{gapClique}_{L,s}$ denote the gap problem of distinguishing between $\{G = (V, E) : \omega(G) \geq L(|V|)\}$ and $\{G = (V, E) : \omega(G) < s(|V|)\}$, where $L \geq s$. Using this terminology, we restate (and strengthen) the aforementioned $|V|^{1-o(1)}$-factor inapproximation of the maximum clique problem.

Theorem 10.5 *For some $L(N) = N^{1-o(1)}$ and $s(N) = N^{o(1)}$, it holds that* $\mathtt{gapClique}_{L,s}$ *is NP-hard.*

As we shall show next, results of the type of Theorem 10.5 imply the hardness of a corresponding approximation problem; that is, the hardness of deciding a gap problem implies the hardness of a search problem that refers to an analogous factor of approximation.

Proposition 10.6 *Let f, g_1, g_2 be as in Definition 10.4 and suppose that these functions are polynomial-time computable. Then the* gap_{g_1,g_2} *problem of maximizing f* (resp., minimizing f) *is reducible to the g_1/g_2-factor* (resp., g_2/g_1-factor) *approximation of maximizing f* (resp., minimizing f).

Note that a reduction in the opposite direction does not necessarily exist (even in the case that the underlying optimization problem is self-reducible in some natural sense). Indeed, this is another difference between the current context (of approximation) and the context of optimization problems, where the search problem is reducible to a related decision problem.

Proof Sketch: We focus on the maximization version. On input x, we solve the gap_{g_1,g_2} problem, by making the query x, obtaining the answer y, and ruling that x has value exceeding $g_1(x)$ if and only if $f(x,y) \geq g_2(x)$. Recall that we need to analyze this reduction only on inputs that satisfy the promise. Thus, if $v_x \geq g_1(x)$ then the oracle must return a solution y that satisfies $f(x,y) \geq v_x/(g_1(x)/g_2(x))$, which implies that $f(x,y) \geq g_2(x)$. On the other hand, if $v_x < g_2(x)$ then $f(x,y) \leq v_x < g_2(x)$ holds for any possible solution y. □

Additional examples. Let us consider $\texttt{gapVC}_{s,L}$, the gap_{g_s,g_L} problem of minimizing the vertex cover of a graph, where s and L are constants and $g_s(G) = s \cdot |V|$ (resp., $g_L(G) = L \cdot |V|$) for any graph $G = (V, E)$. Then, Proposition 10.2 implies (via Proposition 10.6) that, for every constant s, the problem $\texttt{gapVC}_{s,2s}$ is solvable in polynomial-time. In contrast, sufficiently narrowing the gap between the two thresholds yields an inapproximability result. In particular:

Theorem 10.7 *For some constants $s < L$* (e.g., $s = 0.62$ and $L = 0.84$ will do), *the problem $\texttt{gapVC}_{s,L}$ is NP-hard.*

As stated in Section 9.3.3, the PCP Theorem and versions of it play a key role in establishing inapproximability results such as Theorems 10.5 and 10.7. In particular, recall that Theorem 9.21 establishes the equivalence of the PCP Theorem itself and the NP-hardness of a gap problem concerning the maximization of the number of clauses that are satisfies in a given 3-CNF formula. Specifically, $\texttt{gapSAT}^3_\varepsilon$ was defined (in Definition 9.20) as the gap problem consisting of distinguishing between satisfiable 3-CNF formulae and 3-CNF formulae for which each truth assignment violates at least an ε fraction of the clauses. Although Theorem 9.21 does not specify the quantitative relation that underlies its qualitative assertion, when combined with the best known PCP construction, it does yield the best possible bound.

Theorem 10.8 *For every $v < 1/8$, the problem $\texttt{gapSAT}^3_v$ is NP-hard.*

On the other hand, $\texttt{gapSAT}^3_{1/8}$ is solvable in polynomial-time.

Sharp threshold. The aforementioned results regarding $\texttt{gapSAT}_v^3$ exemplify a sharp threshold on the feasibly obtainable approximation factor. Another appealing example refers to the following maximization problem in which the instances are systems of linear equations over GF(2) and the task is finding an assignment that satisfies as many equations as possible. Note that by merely selecting an assignment at random, we expect to satisfy half of the equations. Also note that it is easy to determine whether there exists an assignment that satisfies all equations. Let $\texttt{gapLin}_{L,s}$ denote the problem of distinguishing between systems in which one can satisfy at least an L fraction of the equations and systems in which one cannot satisfy an s fraction (or more) of the equations. Then, as just noted, $\texttt{gapLin}_{L,0.5}$ is trivial and $\texttt{gapLin}_{1,s}$ is feasible (for every $s < 1$). In contrast, moving both thresholds towards one another yields an NP-hard gap problem:

Theorem 10.9 *For every constant $\varepsilon > 0$, the problem $\texttt{gapLin}_{1-\varepsilon,0.5+\varepsilon}$ is NP-hard.*

Gap location. Theorems 10.8 and 10.9 illustrate two opposite situations. In both cases there is an obvious upper-bound on the number of local conditions that can be satisfies (i.e., the number of clauses in the case of gapSAT and the number of equations in the case of gapLin). In one case (i.e., gapSAT) deciding whether an instance has a value that attains the upper-bound is NP-hard, whereas in the other case (i.e., gapLin) this decision problem is solvable in polynomial-time. Consequently, in the second case, the gap problem in which the higher threshold equals this upper-bound is easy (i.e., $\texttt{gapLin}_{1,s}$ is in P), whereas in the first case the corresponding gap problem may be hard. Indeed, $\texttt{gapSAT}_\varepsilon$ refers to distinguishing instances attaining the upper-bound from instances that have a value that is upper-bounded by a lower threshold (i.e., the $1-\varepsilon$ fraction), and in fact $\texttt{gapSAT}_\varepsilon$ is NP-hard.

A final comment. All the aforementioned inapproximability results refer to approximation (resp., gap) problems that are relaxations of optimization problems in NP (i.e., the optimization problem is computational equivalent to a decision problem in $\mathcal{NP}$; see Section 2.2.2). In these cases, the NP-hardness of the approximation (resp., gap) problem implies that the corresponding optimization problem is reducible to the approximation (resp., gap) problem. In other words, in these cases nothing is gained by relaxing the original optimization problem, since the relaxed version remains just as hard.

10.1.2 Decision or Property Testing

A natural notion of relaxation for decision problems arises when considering the distance between instances, where a natural notion of distance is the Hamming distance (i.e., fraction of bits on which two strings disagree). Loosely speaking, this relaxation (called *property testing*) refers to distinguishing inputs that reside in a predetermined set S from inputs that are "relatively far" from any input that resides in the set. Two natural types of promise problems emerge (with respect to the predetermined set S and the Hamming distance between strings):

1. *Relaxed decision w.r.t a fixed distance*: Fixing a distance parameter δ, we consider the problem of distinguishing inputs in S from inputs in $\Gamma_\delta(S)$, where

$$\Gamma_\delta(S) \stackrel{\rm def}{=} \{x : \forall z \in S \cap \{0,1\}^{|x|} \;\; \Delta(x,z) > \delta \cdot |x|\} \qquad (10.1)$$

   and $\Delta(x_1 \cdots x_m, z_1 \cdots z_m) = |\{i : x_i \neq z_i\}|$ denotes the number of bits on which $x = x_1 \cdots x_m$ and $z = z_1 \cdots z_m$ disagree. Thus, here we consider a promise problem that is a restriction (or a special case) of the problem of deciding membership in S.

2. *Relaxed decision w.r.t a variable distance*: Here instances are pairs (x, δ), where x is as in Type 1 and δ is a distance parameter. The yes-instances are pairs (x, δ) such that $x \in S$, whereas (x, δ) is a no-instance if $x \in \Gamma_\delta(S)$.

We shall focus on Type 1 formulation, which seems to capture the essential question of whether or not these relaxations lower the complexity of the original decision problem. The study of Type 2 formulation refers to a relatively secondary question, which assumes a positive answer to the first question; that is, assuming that the relaxed form is easier than the original form, we ask how is the complexity of the problem affected by making the distance parameter smaller (which means making the relaxed problem "tighter" and ultimately equivalent to the original problem).

We note that for numerous NP-complete problems there exist natural (Type 1) relaxations that are solvable in polynomial-time. Actually, these algorithms run in *sub-linear* time (specifically polylogarithmic time), when given direct access to the input. A few examples will be presented in §10.1.2.2. As indicated in §10.1.2.2, this is not a generic phenomenon. We start by discussing several key definitional issues (see §10.1.2.1).

10.1.2.1 Definitional issues

Property testing is concerned not only with solving relaxed versions of NP-hard problems, but rather solving these problems (as well as problems in $\mathcal{P}$) in *sub-linear time*. Needless to say, such results assume a model of computation in which algorithms have direct access to bits in the (representation of the) input (see Definition 10.10).

Definition 10.10 (a direct access model – conventions): *An algorithm with* direct access to its input *is given its* main input *on a special input device that is accessed as an oracle* (see §1.2.3.5). *In addition, the algorithm is given the length of the input and possibly other parameters on an* secondary input device. *The complexity of such an algorithm is stated in terms of the length of its main input.*

Indeed, the description in §5.2.4.2 refers to such a model, but there the main input is viewed as an oracle and the secondary input is viewed as the input.

Definition 10.11 (property testing for S): *The promise problem of distinguishing S from $\Gamma_\delta(S)$ is called* property testing for S (with respect to δ).

Recall that we say that a randomized algorithm solves a promise problem if it accepts every yes-instance (resp., rejects every no-instance) with probability at least 2/3. Thus, a (randomized) property testing for S accepts every input in S (resp., rejects every input in $\Gamma_\delta(S)$) with probability at least 2/3.

The question of representation. The specific representation of the input is of major concern in the current context. Firstly, the representation affects the distance measure (e.g., under an artificially padded input format, all strings will be deemed close to one another and property testing for S will become trivial). Secondly, since our focus is on sub-linear time algorithms, we cannot afford to transform the input from one natural format to another. Both issues will be clarified by the examples provided in §10.1.2.2.

The essential role of the promise. Recall that, for a fixed constant $\delta > 0$, we consider the promise problem of distinguishing S from $\Gamma_\delta(S)$. The promise means that all instances that are neither in S nor far from S (i.e., not in $\Gamma_\delta(S)$) are ignored, which is essential for sub-linear algorithms for natural problems. To demonstrate the point, we say that a set S is reasonably sensitive if for infinitely many $x \in \{0,1\}^* \cap S$ and every $I \subset [|x|]$ of cardinality $|x|/3$ there exists $z \in \{0,1\}^{|x|} \setminus S$ such that for every $i \in I$ it holds that $x_i = z_i$. We note that if S is reasonably sensitive then *deciding membership* in S requires linear time (even in the model of Definition 10.10). On the other hand, even for a reasonably sensitive set S, *property testing* for S (i.e., distinguishing S from $\Gamma_\delta(S)$) may be done in sub-linear (randomized) time. Consider, for example, the set S consisting of strings that have a majority of 1's. Then, S is reasonably sensitive, but the fraction of 1's in the input can be approximated in polylogarithmic time (which yields a property tester for S).

The essential role of randomization. Referring to the foregoing example, we note that randomization is essential for any sub-linear time algorithm that distinguishes this set S from, say, $\Gamma_{0.4}(S)$. In contrast, a sub-linear time deterministic algorithm cannot distinguish 1^n from any input that has 1's in each position probed by the algorithm on input 1^n. In general, on input x, a (sub-linear time) deterministic algorithm always reads the same bits of x and thus cannot distinguish x from any z that agrees with x on these bit locations.

Note that, in both cases, we are able to prove lower-bounds on the time complexity of algorithms. But these lower-bounds are actually information theoretic in nature and refer to the number of queries performed by these algorithms.

10.1.2.2 Two models for testing graph properties

In this subsection we consider the complexity of property testing for sets of graphs that are *closed under graph isomorphism*; such sets are called graph properties. In view of the foregoing comment, referring to the importance of representation, we consider two standard representations of graphs (see Appendix G.1).

1. The adjacency matrix representation. Here a graph $G = ([N], E)$ is represented (in a slightly redundant form) by an N-by-N Boolean matrix $M_G = (m_{i,j})_{i,j\in[N]}$ such that $m_{i,j} = 1$ if and only if $\{i, j\} \in E$.

2. Bounded incidence-lists representation. For a fixed parameter d, a graph $G = ([N], E)$ of degree at most d is represented (in a slightly redundant form) by a mapping $\mu_G : [N] \times [d] \to [N] \cup \{\perp\}$ such that $\mu_G(u, i) = v$ if v is the i^{th} neighbor of u and $\mu_G(u, i) = \perp$ if v has less than i neighbors.

We stress that the aforementioned representations determine both the notion of distance between graphs and the type of queries performed by the algorithm. As we shall see, the difference between these two representations yields a big difference on the complexity of corresponding property testing problems.

Theorem 10.12 (property testing in the adjacency matrix representation): *For any fixed $\delta > 0$ and each of the following sets, there exists a polylogarithmic time randomized algorithm that solves the corresponding property testing problem.*

- *The set of k-colorable graphs, for every fixed $k \geq 2$.*
- *The set of graphs having a clique* (resp. independent set) *of density ρ, for every fixed $\rho > 0$.*
- *The set of N-vertex graphs having a cut*[2] *with at least $\rho \cdot N^2$ edges, for every fixed $\rho > 0$.*
- *The set of N-vertex graphs having a bisection*[2] *with at most $\rho \cdot N^2$ edges, for every fixed $\rho > 0$.*

In contrast, for some $\delta > 0$, there exists a graph property in $\mathcal{NP}$ for which property testing requires linear time.

The algorithms use a constant number of queries, which in turn is polynomial in the constant $1/\delta$, and their running time hides a constant that is exponential in their query complexity, except for the case of 2-colorability where the hidden constant is polynomial in $1/\delta$. Note that such dependencies seem essential, since setting $\delta = 1/N^2$ regains the original (non-relaxed) decision problems (which, with the exception of 2-colorability, are all NP-complete). Again, the lower-bound on the time complexity follows from a lower-bound on the query complexity. We note that the graph property for which this lower-bound is proved is not a natural one.

Theorem 10.12 exhibits a dichotomy graph properties for which property testing is possible by a constant number of queries and graph properties for which property testing requires a linear number of queries. A combinatorial characterization of the graph properties for which property testing is possible (in the adjacency matrix representation) using a constant number of queries is known.[3] We note that the

[2] A cut in a graph $G = ([N], E)$ is a partition $(S, [N] \setminus S)$ of the set of vertices and the edges of the cut are the edges with exactly one endpoint in S. A bisection is a cut of the graph to two parts of equal cardinality.

[3] Describing this fascinating result of Alon *et. al.* [8], which refers to the notion of regular partitions (introduced by Szemerédi), is beyond the scope of the current text.

constant in this characterization may depend arbitrarily on δ (and indeed, in some cases, it a function growing faster than a tower of $1/\delta$ exponents).

Turning back to Theorem 10.12, we note that the results regarding property testing for the sets corresponding to max-cut and min-bisection yield approximation algorithms with an additive error term (of δN^2). For dense graphs (i.e., N-vertex graphs having $\Omega(N^2)$ edges), this yields a constant factor approximation for the standard approximation problem (as in Definition 10.1). That is, for every constant $c > 1$, we obtain a c-factor approximation of maximizing the size of a cut (resp., minimizing the size of the bisection) in dense graphs. On the other hand, the result regarding clique yields a so called dual-approximation for maximum clique; that is, we approximate the minimum number of missing edges in the densest induced graph of a given size.

Indeed, Theorem 10.12 is meaningful only for dense graphs. The same holds, in general, for the adjacency matrix representation.[4] Also note that for sets S satisfying $\Gamma_\delta(S) = \emptyset$ under the adjacency matrix representation (e.g., the set of connected graphs) property testing is trivial.

We now turn to the bounded incidence-lists representation, which is relevant only for bounded degree graphs. The problems of max-cut, min-bisection and clique (as in Theorem 10.12) are trivial under this representation, but graph connectivity becomes non-trivial and the complexity of property testing for the set of bipartite graphs changes dramatically.

Theorem 10.13 (property testing in the bounded incidence-lists representation): *The following assertions refer to representations of graphs of degree at most d.*

- *For any fixed d and $\delta > 0$, there exists a polylogarithmic time randomized algorithm that solves the property testing problem for the set of connected graphs of degree at most d.*

- *For any fixed d and $\delta > 0$, there exists a sub-linear randomized algorithm that solves the property testing problem for the set of bipartite graphs of degree at most d. Specifically, on input an N-vertex graph, the algorithm runs for $\widetilde{O}(\sqrt{N})$ time.*

- *For any fixed $d \geq 3$ and some $\delta > 0$, property testing for the set of N-vertex* (3-regular) *bipartite graphs requires $\Omega(\sqrt{N})$ queries.*

- *For some fixed d and $\delta > 0$, property testing for the set of N-vertex 3-colorable graphs requires $\Omega(N)$ queries.*

The running time of the algorithms hides a constant that is polynomial in $1/\delta$. Providing a characterization of graph properties according to the complexity of the corresponding tester (in the bounded incidence-lists representation) is an interesting open problem.

[4] In this model, all N-vertex graphs having less than δN^2 edges may be accepted if and only if there exists such a (non-dense) graph in the predetermined set. This trivial algorithm is correct because if set S contains an N-vertex graph with less than δN^2 then $\Gamma_\delta(S) = \emptyset$.

Decoupling the distance from the representation. So far we have confined our attention to the Hamming distance between the representations of graphs. This made the choice of representation even more important than usual (i.e., more crucial than is common in complexity theory). In contrast, it is natural to consider a notion of distance between graphs that is independent of their representation. For example, the distance between $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ can be defined as the size of symmetric difference between E_1 and the set of edges in a graph that is isomorphic to G_2. The corresponding relative distance may be defined as the distance divided by $|E_1| + |E_2|$ (or by $\max(|E_1|, |E_2|)$).

10.1.2.3 Beyond graph properties

Property testing has been applied to a variety of computational problems beyond the domain of graph theory. In fact, this area first emerged in the algebraic domain, where the instances (to be viewed as inputs to the testing algorithm) are functions and the relevant properties are sets of algebraic functions. The archetypical example is the set of low-degree polynomials; that is, m-variate polynomials of total (or individual) degree d over some finite field $\mathrm{GF}(q)$, where m, d and q are functions of the length of the input (which being the full description of a m-variate function over $\mathrm{GF}(q)$, has length $q^m \cdot \log_2 q$).

Note that we have referred to these computational problems, under a different terminology, in §9.3.2.2 and in §9.3.2.1. In particular, in §9.3.2.1 we refereed to the special case of linear Boolean functions (i.e., individual degree 1 and $q = 2$), whereas in §9.3.2.2 we used the setting $q = \mathrm{poly}(d)$ and $m = d/\log d$ (where d is a bound on the total degree).

Viewing the problem instance as a function suggests a natural measure of distance (i.e., the fraction of arguments on which the functions disagree) as well as a natural way of accessing the instance (i.e., querying the function for the value of selected arguments). However, as in §10.1.2.2, we may decouple the distance measure from the representation (i.e., a way of accessing the problem instance). This is done by introducing a representation-independent notion of distance between instances, which should be natural in the context of the problem at hand.

10.2 Average Case Complexity

Teaching note: We view average-case complexity as referring to the performance on average (or typical) instances, and not as the average performance on random instances. This choice is justified in §10.2.1.1. Thus, the current theory may be termed typical-case complexity.

Our approach so far (including in Section 10.1) is termed worst-case complexity, because it refers to the performance of potential algorithms on each legitimate instance (and hence to the performance on the worst possible instance). That is, computational problems were defined as referring to a set of instances and performance guarantees were required to hold for each instance in this set. In contrast, average-case complexity allows ignoring a negligible measure of the possible instances, where *the identity of the ignored instances is determined by the analysis of potential solvers and not by the problem's statement.*

A few comments are in place. Firstly, as just hinted, the standard statement of the worst-case complexity of a computational problem (especially one having a promise) may also ignores some instances (i.e., those considered inadmissible or violating the promise), but these instances are determined by the problem's statement. In contrast, the inputs ignored in average-case complexity are not inadmissible in any inherent sense (and are certainly not identified as such by the problem's statement). It is just that they are viewed as exceptional when claiming that a specific algorithm solve the problem; furthermore, these exceptional instances are determined by the analysis of that algorithm. Needless to say, these exceptional instances ought to be rare (i.e., occur with negligible probability).

The last sentence raises a couple of issues. Firstly, a distribution on the set of admissible instances has to be specified. In fact, we shall consider a new type of computational problems, each consisting of a standard computational problem coupled with a probability distribution on instances. Consequently, the question of which distributions should be considered arises. These and numerous other definitional issues will be addressed in §10.2.1.1.

Before proceeding, let us spell out the rather straightforward motivation to the study of the average-case complexity of computational problems. It is that, in real-life applications, one may be perfectly happy with an algorithm that solves the problem fast on almost all instances that arise in the application. That is, one may be willing to tolerate error provided that it occurs with negligible probability, where the probability is taken over the distribution of instances encountered in the application. We stress that a key aspect in this approach is a good modeling of the type of distributions of instances that are encountered in natural algorithmic applications.

At this point a natural question arises: *can natural computational problems be solve efficiently when considering typical instances?* The bottom-line of this section is that, for a well-motivated choice of definitions, our conjecture is that the "distributional version" of NP is not contained in the average-case (or typical-case) version of P. This means that some NP problems are not merely hard in the worst-case, but rather "typically hard" (i.e., hard on typical instances drawn from some simple distribution). Specifically, hard instances may occur in natural algorithmic applications (and not only in cryptographic (and other "adversarial") applications that are design on purpose to produce hard instances). This conjecture motivates the development of an average-case analogue of NP-completeness, which will be presented in this section. Indeed, the entire section may be viewed as an average-case analogue of Chapter 2.

Organization. A major part of our exposition is devoted to the definitional issues that arise when developing a general theory of average-case complexity. These issues are discussed in §10.2.1.1. In §10.2.1.2 we prove the existence of a distributional problem that is "NP-complete" in the average-case complexity sense. In

§10.2.1.3 we extend the treatment to randomized algorithms. Additional ramifications are presented in Section 10.2.2.

10.2.1 The basic theory

In this section we provide a basic treatment of the theory of average-case complexity, while postponing important ramifications to Section 10.2.2. The basic treatment contains the preferred definitional choices for the main notions as well as the identification of a complete problem for a natural class of average-case computational problems.

10.2.1.1 Definitional issues

The theory of average-case complexity is more subtle than may appear in first thought. In addition to the generic difficulty involved in defining relaxations, difficulties arise from the "interface" between standard probabilistic analysis and the conventions of complexity theory. This is most striking in the definition of the class of feasible average-case computations. Referring to the theory of worst-case complexity as a guideline, we shall address the following aspects of the analogous theory of average-case complexity.

1. *Setting the general framework.* We shall consider distributional problems, which are standard computational problems (see Section 1.2.2) coupled with distributions on the relevant instances.

2. *Identifying the class of feasible* (distributional) *problems.* Seeking an average-case analogue of classes such as $\mathcal{P}$, we shall reject the first definition (of "average polynomial time") that comes to mind, briefly discuss several related alternatives, and adopt one of them for the main treatment.

3. *Identifying the class of interesting* (distributional) *problems.* Seeking an average-case analogue of the class $\mathcal{NP}$, we shall avoid both the extreme of allowing arbitrary distributions (which collapses average-case complexity to worst-case complexity) and the opposite extreme of confining the treatment to the uniform distribution (which is misguided by the naive assumption that this distribution is the only one relevant to applications).

4. *Developing an adequate notion of reduction among* (distributional) *problems.* As in the theory of worst-case complexity, this notion should preserve feasible solveability (in the current distributional context).

We now turn to the actual treatment of each of the aforementioned aspects.

Distributional problems. Focusing on decision problems, we define distributional problems as pairs consisting of a decision problem and a probability ensemble.[5]

[5]We mention that even this choice is not evident. Specifically, Levin [144] (see discussion in [85]) advocates the use of a single probability distribution defined over the set of all strings.

For simplicity, here a probability ensemble $\{X_n\}_{n\in\mathbb{N}}$ is a sequence of random variables such that X_n ranges over $\{0,1\}^n$. Thus, $(S,\{X_n\}_{n\in\mathbb{N}})$ is the distributional problem consisting of the problem of deciding membership in the set S with respect to the probability ensemble $\{X_n\}_{n\in\mathbb{N}}$. (The treatment of search problem is similar; see §10.2.2.1.) We denote the uniform probability ensemble by $U = \{U_n\}_{n\in\mathbb{N}}$; that is, U_n is uniform over $\{0,1\}^n$.

Identifying the class of feasible problems. The first idea that comes to mind is defining the problem $(S,\{X_n\}_{n\in\mathbb{N}})$ as feasible (on the average) if there exists an algorithm A that solves S such that the *average running time* of A on X_n is bounded by a polynomial in n (i.e., there exists a polynomial p such that $\mathsf{E}[t_A(X_n)] \leq p(n)$, where $t_A(x)$ denotes the running-time of A on input x). The problem with this definition is that it very sensitive to the model of computation and is not closed under algorithmic composition. Both deficiencies are a consequence of the fact that t_A may be polynomial on the average with respect to $\{X_n\}_{n\in\mathbb{N}}$ but t_A^2 may fail to be so (e.g., consider $t_A(x'x'') = 2^{|x'|}$ if $x' = x''$ and $t_A(x'x'') = |x'x''|^2$ otherwise, coupled with the uniform distribution over $\{0,1\}^n$). We conclude that the average running-time of algorithms is not a robust notion. We also doubt the naive appeal of this notion, and view the typical running time of algorithms (as defined next) as a more natural notion. Thus, we shall consider an algorithm as feasible if its running-time is typically polynomial.[6]

We say that A is typically polynomial-time on $X = \{X_n\}_{n\in\mathbb{N}}$ if there exists a polynomial p such that the probability that A runs more that $p(n)$ steps on X_n is *negligible* (i.e., for every polynomial q and all sufficiently large n it holds that $\mathsf{Pr}[t_A(X_n) > p(n)] < 1/q(n)$). The question is what is required in the "untypical" cases, and two possible definitions follow.

1. The simpler option is saying that $(S,\{X_n\}_{n\in\mathbb{N}})$ is (typically) feasible if there exists an algorithm A that solves S such that A is typically polynomial-time on $X = \{X_n\}_{n\in\mathbb{N}}$. This effectively requires A to correctly solve S on each instance, which is more than was required in the motivational discussion. (Indeed, if the underlying reasoning is ignoring rare cases, then we should ignore them altogether rather than partially (i.e., only ignore their affect on the running-time).)

2. The alternative, which fits the motivational discussion, is saying that (S,X) is (typically) feasible if there exists an algorithm A such that typically A

His argument is that this makes the theory less representation-dependent. At the time we were convinced of his argument (see [85]), but currently we feel that the representation-dependent effects discussed in [85] are legitimate. Furthermore, the alternative formulation of [85] comes across as unnatural and tends to be confusing.

[6]An alternative choice, taken by Levin [144] (see discussion in [85]), is considering as feasible (w.r.t $X = \{X_n\}_{n\in\mathbb{N}}$) any algorithm that runs in time that is polynomial in a function that is linear on the average (w.r.t X); that is, requiring that there exists a polynomial p and a function $\ell : \{0,1\}^* \to \mathbb{N}$ such that $t(x) \leq p(\ell(x))$ and $\mathsf{E}[\ell(X_n)] = O(n)$. This definition is robust (i.e., it does not suffer from the aforementioned deficiencies) and is arguably as justified as the naive definition (i.e., $\mathsf{E}[t_A(X_n)] \leq \mathrm{poly}(n)$).

solves S on X in polynomial-time; that is, there exists a polynomial p such that the probability that on input X_n algorithm A either errs or runs more that $p(n)$ steps is negligible. This formulation totally ignores the untypical instances. Indeed, in this case we may assume, without loss of generality, that A always runs in polynomial-time (see Exercise 10.4), but we shall not do so here (in order to facilitate viewing the first option as a special case of the current option).

We note that both alternatives actually define typical feasibility and not average-case feasibility. To illustrate the difference between the two options, consider the distributional problem of deciding whether a uniformly selected (n-vertex) graph contains a Hamiltonian path. Intuitively, this problem is "typically trivial" because the algorithm may always say `yes` and be wrong with exponentially vanishing probability.[7] Indeed, this trivial algorithm is admissible by the second approach, but not by the first approach. In light of the foregoing, we adopt the second approach.

Definition 10.14 (the class tpc$\mathcal{P}$): *We say that A* typically solves $(S, \{X_n\}_{n\in\mathbb{N}})$ in polynomial-time *if there exists a polynomial p such that the probability that on input X_n algorithm A either errs or runs more that $p(n)$ steps is negligible.*[8] *We denote by* tpc$\mathcal{P}$ *the class of distributional problems that are typically solvable in polynomial-time.*

Clearly, for every $S \in \mathcal{P}$ and every probability ensemble X, it holds that $(S, X) \in$ tpc$\mathcal{P}$. However, tpc$\mathcal{P}$ contains also distributional problems (S, X) with $S \notin \mathcal{P}$ (see Exercises 10.5 and 10.6). The big question motivating the study of average-case complexity is whether natural distributional versions of $\mathcal{NP}$ are in tpc$\mathcal{P}$. Thus, we turn to identify such versions.

Identifying the class of interesting problems. Seeking to identify reasonable distributional versions of $\mathcal{NP}$, we note that two extreme choices should be avoided. On one hand, we must limit the class of admissible distributions so to prevent the collapse of average-case complexity to worst-case complexity (by a selection of a pathological distribution that resides on the "worst case" instances). On the other hand, we should allow for various types of natural distributions rather than confining attention merely to the uniform distribution. Recall that our aim is to address all possible input distributions that may occur in applications, and there is no justification to confining attention to the uniform distribution. Still, arguably, the distributions occuring in applications are "relatively simple" and so we seek to identify a class of simple distributions. One such notion (of simple distributions)

[7] In contrast, testing whether a given graph contains a Hamiltonian path seems "typically hard" for other distributions (see Exercise 10.15). Needless to say, in the latter distributions both yes-instances and no-instances appear with noticeable probability.

[8] Recall that a function $\mu : \mathbb{N} \to \mathbb{N}$ is negligible if for every positive polynomial q and all sufficiently large n it holds that $\mu(n) < 1/q(n)$. We say that A errs on x if $A(x)$ differs from the indicator value of the predicate $x \in S$.

underlies the following definition, while a more liberal notion will be presented in §10.2.2.2.

Definition 10.15 (the class dist$\mathcal{NP}$): *We say that a probability ensemble* $X = \{X_n\}_{n\in\mathbb{N}}$ *is* simple *if there exists a polynomial time algorithm that, on any input* $x \in \{0,1\}^*$, *outputs* $\Pr[X_{|x|} \leq x]$, *where the inequality refers to the standard lexicographic order of strings. We denote by* dist$\mathcal{NP}$ *the class of distributional problems consisting of decision problems in* $\mathcal{NP}$ *coupled with simple probability ensembles.*

Note that the uniform probability ensemble is simple, but so are many other "simple" probability ensembles. Actually, it makes sense to relax the definition such that the algorithm is only required to output an approximation of $\Pr[X_{|x|} \leq x]$, say, to within a factor of $1 \pm 2^{-|x|}$. We note that although Definition 10.15 does not refer explicitly to any structural property of the probability ensemble, it imposes a computational restriction on the admissible ensembles (which, in turn, limits their "complexity"). In §10.2.2.2 we shall consider the more intuitive and robust class of all polynomial-time sampleable ensembles (and show that it contains all simple ensembles). We believe that the results presented in §10.2.1.2 and §10.2.2.2 retrospectively justify the choice underlying Definition 10.15. We articulate this point next.

We note that a wider class of distributions weakens the conjecture that some distributional version of NP are not feasible. On the other hand, the conclusion that some distributional problem is not feasible becomes stronger when restricting the admissible class of distributions. The results presented in §10.2.1.2 and §10.2.2.2 assert that a conjecture that refers to polynomial-time sampleable ensembles implies a conclusion that refers to a (very) simple probability ensemble. The current setting in which both the conjecture and the conclusion refer to simple probability ensembles is thus but an intermediate step.

Indeed, the big question in the current context is whether dist$\mathcal{NP}$ is contained in tpc$\mathcal{P}$. A positive answer (especially if extended to sampleable ensembles) would deem the P-vs-NP Question of little practical significant. However, our daily experience as well as much research effort indicate that some NP problems are not merely hard in the worst-case, but rather "typically hard". This supports the *conjecture that* dist$\mathcal{NP}$ *is not contained in* tpc$\mathcal{P}$.

Needless to say, the latter conjecture implies $\mathcal{P} \neq \mathcal{NP}$, and thus we should not expect to see a proof of it. What we may hope to see is "dist$\mathcal{NP}$-complete" problems; that is, problems in dist$\mathcal{NP}$ that are not in tpc$\mathcal{P}$ unless the entire class dist$\mathcal{NP}$ is contained in tpc$\mathcal{P}$. An adequate notion of a reduction is used towards formulating this notion.

Reductions among (distributional) problems. Intuitively, such reductions must preserve average-case feasibility. Thus, in addition to the standard conditions (i.e., that the reduction be efficiently computable and yield a correct result), we require that the reduction "respects" the probability distribution of the corresponding distributional problems. Specifically, the reduction should not map very likely instances of the first ("starting") problem to rare instances of the second ("target") problem. Otherwise, having a typically polynomial-time algorithm for the second distributional problem does not necessarily yield such an algorithm for the first distributional problem. Following is the adequate analogue of a Cook reduction (i.e., general polynomial-time reduction), where the analogue of a Karp-reduction (many-to-one reduction) can be easily derived as a special case.

> **Teaching note:** One may prefer presenting in class only the special case of many-to-one reductions, which suffices for Theorem 10.17. See Footnote 10.

Definition 10.16 (reductions among distributional problems): *We say that the oracle machine M* reduces *the distributional problem (S,X) to the distributional problem (T,Y) if the following three conditions hold.*

1. Efficiency: *The machine M runs in polynomial-time.*[9]

2. Validity: *For every $x\in\{0,1\}^*$, it holds that $M^T(x)=1$ if an only if $x\in S$, where $M^T(x)$ denotes the output of the oracle machine M on input x and access to an oracle for T.*

3. Domination:[10] *There exists a polynomial p such that, for every $y\in\{0,1\}^*$, it holds that*

$$\max_{n\in\mathbb{N}}\{\mathsf{Pr}[Q(X_n)\ni y]\}\leq p(|y|)\cdot\mathsf{Pr}[Y_{|y|}=y], \tag{10.2}$$

where $Q(x)$ denotes the set of queries made by M on input x and oracle access to T. Furthermore, if $y\in Q(x)$ then $|x|\leq p(|y|)$.

The l.h.s. of Eq. (10.2) refers to the probability that, on input distributed as X_n, the reduction makes the query y. This probability is required not to exceed the probability that y occurs in the distribution $Y_{|y|}$ by more than a polynomial factor in $|y|$. In this case we say that the l.h.s. of Eq. (10.2) is dominated by $\mathsf{Pr}[Y_{|y|}=y]$.

Indeed, the domination condition is the only aspect of Definition 10.16 that extends beyond the worst-case treatment of reductions and refers to the distributional setting. The domination condition does not insist that the distribution induced by $Q(X)$ equals Y, but rather allows some slackness that, in turn, is bounded so to guarantee preservation of typical feasibility (see Exercise 10.7).

We note that the reducibility arguments extensively used in Chapters 7 and 8 (see discussion in Section 7.1.2) are actually reductions in the spirit of Definition 10.16 (except that they refer to a different type of computational tasks).

[9] In fact, one may relax the requirement and only require that M is typically polynomial-time with respect to X. The validity condition may also be relaxed similarly.

[10] For simplicity, we use the same polynomial for both bounds. Let us spell out the meaning of Eq. (10.2) in the special case of many-to-one reductions (i.e., $M^T(x)=1$ if and only if $f(x)\in T$, where f is a polynomial-time computable function): in this case $\mathsf{Pr}[Q(X_n)\ni y]$ is replaced by $\mathsf{Pr}[f(X_n)=y]$. Assuming that f is one-to-one, Eq. (10.2) simplifies to $\mathsf{Pr}[X_{|f^{-1}(y)|}=f^{-1}(y)]\leq p(|y|)\cdot\mathsf{Pr}[Y_{|y|}=y]$ for any y in the image of f. Indeed, nothing is required for y not in the image of f.

10.2.1.2 Complete problems

Recall that our conjecture is that dist$\mathcal{NP}$ is not contained in tpc$\mathcal{P}$, which in turn strengthens the conjecture $\mathcal{P} \neq \mathcal{NP}$ (making infeasibility a typical phenomenon rather than a worst-case one). Having no hope of proving that dist$\mathcal{NP}$ is not contained in tpc$\mathcal{P}$, we turn to the study of complete problems with respect to that conjecture. Specifically, we say that a distributional problem (S, X) is dist$\mathcal{NP}$-complete if $(S, X) \in$ dist$\mathcal{NP}$ and every $(S', X') \in$ dist$\mathcal{NP}$ is reducible to (S, X) (under Definition 10.16).

Recall that it is quite easy to prove the mere existence of NP-complete problems and many natural problems are NP-complete. In contrast, in the current context, establishing completeness results is quite hard. This should not be surprising in light of the restricted type of reductions allowed in the current context. The restriction (captured by the domination condition) requires that "typical" instances of one problem should not be mapped to "untypical" instances of the other problem. On the other hand, it is fair to say that standard Karp-reductions (used in establishing NP-completeness results) map "typical" instances of one problem to quite "bizarre" instances of the second problem. Thus, the current subsection may be viewed as a study of reductions that do not commit this sin.

Theorem 10.17 (dist$\mathcal{NP}$-completeness): *dist$\mathcal{NP}$ contains a distributional problem (S, X) such that each distributional problem in dist$\mathcal{NP}$ is reducible* (per Definition 10.16) *to (S, X). Furthermore, the reduction is deterministic and many-to-one.*

Proof: We start by introducing such a problem, which is a natural distributional version of the decision problem $S_{\mathtt{u}}$ (used in the proof of Theorem 2.18). Recall that $S_{\mathtt{u}}$ contains the instance $\langle M, x, 1^t \rangle$ if there exists $y \in \cup_{i \leq t} \{0,1\}^i$ such that M accepts the input pair (x, y) within t steps. We couple $S_{\mathtt{u}}$ with the "quasi-uniform" probability ensemble U' that assigns to the instance $\langle M, x, 1^t \rangle$ a probability mass proportional to $2^{-(|M|+|x|)}$. Specifically, for $|\langle M, x, 1^t \rangle| = n$ it holds that $\mathsf{Pr}[U'_n = \langle M, x, 1^t \rangle] = 2^{-(|M|+|x|)} / \binom{n}{2}$. Note that, under a suitable encoding, the ensemble U' is indeed simple.[11]

The reader can easily verify that the generic reduction used when reducing any set in $\mathcal{NP}$ to $S_{\mathtt{u}}$ (see the proof of Theorem 2.18), fails to reduce dist$\mathcal{NP}$ to $(S_{\mathtt{u}}, U')$. Specifically, in some cases (see next paragraph), these reductions do not satisfy the domination condition. Indeed, the difficulty is that we have to reduce all dist$\mathcal{NP}$ problems (i.e., pairs consisting of decision problems and simple distributions) to one single distributional problem (i.e., $(S_{\mathtt{u}}, U')$). Applying the aforementioned reductions, we end up with many distributional versions of $S_{\mathtt{u}}$, and furthermore the corresponding distributions are very different (and are not necessarily dominated by a single distribution).

Let us take a closer look at the aforementioned generic reduction, when applied to an arbitrary $(S, X) \in$ dist$\mathcal{NP}$. This reduction maps an instance x to a triple $(M_S, x, 1^{p_S(|x|)})$, where M_S is a machine verifying membership in S (while using

[11] For example, we may encode $\langle M, x, 1^t \rangle$, where $M = \sigma_1 \cdots \sigma_k \in \{0,1\}^k$ and $x = \tau_1 \cdots \tau_\ell \in \{0,1\}^\ell$, by the string $\sigma_1\sigma_1 \cdots \sigma_k\sigma_k 01 \tau_1\tau_1 \cdots \tau_\ell\tau_\ell 01^t$.

adequate NP-witnesses) and p_S is an adequate polynomial. The problem is that x may have relatively large probability mass (i.e., it may be that $\Pr[X_{|x|}=x] \gg 2^{-|x|}$) while $(M_S, x, 1^{p_S(|x|)})$ has "uniform" probability mass (i.e., $\langle M_S, x, 1^{p_S(|x|)}\rangle$ has probability mass smaller than $2^{-|x|}$). This violates the domination condition (see Exercise 10.10), and thus an alternative reduction is required.

The key to the alternative reduction is an (efficiently computable) encoding of strings taken from an arbitrary *simple* distribution by strings that have a similar probability mass under the uniform distribution. This means that the encoding should shrink strings that have relatively large probability mass under the original distribution. Specifically, this encoding will map x (taken from the ensemble $\{X_n\}_{n\in\mathbb{N}}$) to a codeword x' of length that is upper-bounded by the logarithm of $1/\Pr[X_{|x|}=x]$, ensuring that $2^{-|x'|} \geq \Pr[X_{|x|}=x]$. Accordingly, the reduction will map x to a triple $(M_{S,X}, x', 1^{p'(|x|)})$, where $|x'| < O(1) + \log_2(1/\Pr[X_{|x|}=x])$ and $M_{S,X}$ is an algorithm that first verifies that x' is a proper encoding of x and next applies the standard verification (i.e., M_S) of the problem S. Such a reduction will be shown to satisfy all three conditions (i.e., efficiency, validity, and domination). Thus, instead of forcing the structure of the original distribution X on the target distribution U', the reduction will incorporate the structure of X in the reduced instance. A key ingredient in making this possible is the fact that X is simple (as per Definition 10.15).

With the foregoing motivation in mind, we now turn to the actual proof; that is, proving that any (S, X) is reducible to $(S_{\mathtt{u}}, U')$. The following technical lemma is the basis of the reduction. In this lemma as well as in the sequel, it will be convenient to consider the (accumulative) distribution function of the probability ensemble X. That is, we consider $\mu(x) \stackrel{\text{def}}{=} \Pr[X_{|x|} \leq x]$, and note that $\mu : \{0,1\}^* \to [0,1]$ is polynomial-time computable (because X satisfies Definition 10.15).

Coding Lemma:[12] Let μ be a polynomial-time computable distribution function. Then there exist an encoding function C_μ satisfying the following three conditions.

1. Compression: For every x it holds that $|C_\mu(x)| \leq 1 + \min\{|x|, \log_2(1/\mu'(x))\}$, where $\mu'(x) \stackrel{\text{def}}{=} \Pr[X_{|x|}=x]$.

2. Efficient Encoding: The function C_μ is computable in polynomial-time.

3. Unique Decoding: For every $n \in \mathbb{N}$, when restricted to $\{0,1\}^n$, the function C_μ is one-to-one (i.e., if $C_\mu(x) = C_\mu(x')$ and $|x| = |x'|$ then $x = x'$).

Proof: The function C_μ is defined as follows. If $\mu'(x) \leq 2^{-|x|}$ then $C_\mu(x) = 0x$ (i.e., in this case x serves as its own encoding). Otherwise (i.e., $\mu'(x) > 2^{-|x|}$) then $C_\mu(x) = 1z$, where z is chosen such that $|z| \leq \log_2(1/\mu'(x))$ and the mapping of n-bit strings to their encoding is one-to-one. Loosely speaking, z is selected to

[12] The lemma actually refers to $\{0,1\}^n$, for a fixed value of n, but the efficiency condition is stated more easily when allowing n to vary (and using the standard asymptotic analysis of algorithms). Furthermore, the lemma holds for any monotonically non-decreasing function that is efficiently computable, and its proof is less cumbersome when stated for functions defined over $\{0,1\}^*$. See further discussion in Exercise 10.11.

equal the shortest binary expansion of a number in the interval $(\mu(x)-\mu'(x),\mu(x)]$. Bearing in mind that this interval has length $\mu'(x)$ and that the different intervals are disjoint, we obtain the desired encoding. Details follows.

We focus on the case that $\mu'(x) > 2^{-|x|}$, and detail the way that z is selected (for the encoding $C_\mu(x) = 1z$). If $x > 0^{|x|}$ and $\mu(x) < 1$, then we let z be the longest common prefix of the binary expansions of $\mu(x-1)$ and $\mu(x)$, where $x-1$ is the string preceding x in lexicographic order (e.g., if $\mu(1010) = 0.10010$ and $\mu(1011) = 0.10101111$ then $C_\mu(1011) = 1z$ with $z = 10$). Thus, in this case $0.z1$ is in the interval $(\mu(x-1),\mu(x)]$ (i.e., $\mu(x-1) < 0.z1 \le \mu(x)$). For $x = 0^{|x|}$, we let z be the longest common prefix of the binary expansions of 0 and $\mu(x)$ and again $0.z1$ is in the relevant interval (i.e., $(0,\mu(x)]$). Finally, for x such that $\mu(x) = 1$ and $\mu(x-1) < 1$, we let z be the longest common prefix of the binary expansions of $\mu(x-1)$ and $1-2^{-|x|-1} < \mu(x)$ (and again $0.z1$ is in $(\mu(x-1),\mu(x)]$). Note that if $\mu(x) = \mu(x-1) = 1$ then $\mu'(x) = 0 < 2^{-|x|}$.

We now verify that the foregoing C_μ satisfies the conditions of the lemma. We start with the compression condition. Clearly, if $\mu'(x) \le 2^{-|x|}$ then $|C_\mu(x)| = 1+|x| \le 1+\log_2(1/\mu'(x))$. On the other hand, suppose that $\mu'(x) > 2^{-|x|}$ and let us focus on the sub-case that $x > 0^{|x|}$ and $\mu(x) < 1$. Let $z = z_1\cdots z_\ell$ be the longest common prefix of the binary expansions of $\mu(x-1)$ and $\mu(x)$. Then, $\mu(x-1) = 0.z0u$ and $\mu(x) = 0.z1v$, where $u,v \in \{0,1\}^*$, and it follows that

$$\mu'(x) \;=\; \mu(x)-\mu(x-1) \;\le\; \left(\sum_{i=1}^{\ell} 2^{-i}z_i + \sum_{i=\ell+1}^{\mathrm{poly}(|x|)} 2^{-i}\right) - \sum_{i=1}^{\ell} 2^{-i}z_i \;<\; 2^{-|z|}.$$

Thus, $|z| < \log_2(1/\mu'(x)) \le |x|$ and it follows that $|C_\mu(x)| \le 1+\min(|x|,\log_2(1/\mu'(x)))$ holds in both cases. Clearly, C_μ can be computed in polynomial-time by computing $\mu(x-1)$ and $\mu(x)$. Finally, note that C_μ satisfies the unique decoding condition, by separately considering the two aforementioned cases (i.e., $C_\mu(x) = 0x$ and $C_\mu(x) = 1z$). Specifically, in the second case (i.e., $C_\mu(x) = 1z$), use the fact that $\mu(x-1) < 0.z1 \le \mu(x)$. $\square$

To obtain an encoding that is one-to-one when applied to strings of different lengths we augment C_μ in the obvious manner; that is, we consider $C'_\mu(x) \stackrel{\mathrm{def}}{=} (|x|, C_\mu(x))$, which may be implemented as $C'_\mu(x) = \sigma_1\sigma_1\cdots\sigma_\ell\sigma_\ell 01 C_\mu(x)$ where $\sigma_1\cdots\sigma_\ell$ is the binary expansion of $|x|$. Note that $|C'_\mu(x)| = O(\log|x|) + |C_\mu(x)|$ and that C'_μ is one-to-one.

The machine associated with (S,X). Let μ be the accumulative probability function associated with the probability ensemble X and M_S be the polynomial-time machine that verifies membership in S while using adequate NP-witnesses (i.e., $x \in S$ if and only if there exists $y \in \{0,1\}^{\mathrm{poly}(|x|)}$ such that $M(x,y) = 1$). Using the encoding function C'_μ, we introduce an algorithm $M_{S,\mu}$ with the intension of reducing the distributional problem (S,X) to $(S_{\mathtt{u}},U')$ such that all instances (of S) are mapped to triples in which the first element equals $M_{S,\mu}$. Machine $M_{S,\mu}$ is given an alleged encoding (under C'_μ) of an instance to S along with an alleged proof that the corresponding instance is in S, and verifies these claims in the obvious manner. That is, on input x' and $\langle x,y\rangle$, machine $M_{S,\mu}$ first verifies that $x' = C'_\mu(x)$, and next verifiers that $x \in S$ by running $M_S(x,y)$. Thus, $M_{S,\mu}$ verifies membership in the set $S' = \{C'_\mu(x) : x \in S\}$, while using proofs of the form $\langle x,y\rangle$ such that $M_S(x,y) = 1$ (for the instance $C'_\mu(x)$).[13]

The reduction. We maps an instance x (of S) to the triple $(M_{S,\mu}, C'_\mu(x), 1^{p(|x|)})$, where $p(n) \stackrel{\text{def}}{=} p_S(n)+p_C(n)$ such that p_S is a polynomial representing the running-time of M_S and p_C is a polynomial representing the running-time of the encoding algorithm. That is, on input (x,y), algorithm M_S makes at most $p_S(|x|)$ steps (and rejects (x,y) if $|y| > p(|x|) - |x|$).

Analyzing the reduction. Our goal is proving that *the foregoing mapping constitutes a reduction of* (S,X) *to* $(S_{\mathbf{u}}, U')$. We verify the corresponding three requirements (of Definition 10.16).

1. Using the fact that C_μ is polynomial-time computable (and noting that p is a polynomial), it follows that the foregoing mapping can be computed in polynomial-time.

2. Recall that, on input $(x', \langle x,y\rangle)$, machine $M_{D,\mu}$ accepts if and only if $x' = C'_\mu(x)$ and M_D accepts (x,y) within $p_S(|x|)$ steps. It follows that $x \in S$ if and only if there exists a string y of length at most $p(|x|)$ such that $M_{S,\mu}$ accepts $(C'_\mu(x), \langle x,y\rangle)$ in at most $p(|x|)$ steps. Thus, $x \in S$ if and only if $(M_{S,\mu}, C'_\mu(x), 1^{p(|x|)}) \in S_{\mathbf{u}}$, and the validity condition follows.

3. In order to verify the domination condition, we first note that the foregoing mapping is one-to-one (because the transformation $x \to C'_\mu(x)$ is one-to-one). Next, we note that it suffices to consider instances of $S_{\mathbf{u}}$ that have a preimage under the foregoing mapping (since instances with no preimage trivially satisfy the domination condition). Each of these instances (i.e., each image of this mapping) is a triple with the first element equal to $M_{S,\mu}$ and the second element being an encoding under C'_μ. By the definition of U', for every such image $\langle M_{S,\mu}, C'_\mu(x), 1^{p(|x|)}\rangle \in \{0,1\}^n$, it holds that

$$\begin{aligned} \mathsf{Pr}[U'_n = \langle M_{S,\mu}, C'_\mu(x), 1^{p(|x|)}\rangle] &= \binom{n}{2}^{-1} \cdot 2^{-(|M_{S,\mu}|+|C'_\mu(x)|)} \\ &> c \cdot n^{-2} \cdot 2^{-(|C_\mu(x)|+O(\log|x|))}, \end{aligned}$$

where $c = 2^{-|M_{S,\mu}|-1}$ is a constant depending only on S and μ (i.e., on the distributional problem (S,X)). Thus, for some positive polynomial p', we have

$$\mathsf{Pr}[U'_n = \langle M_{S,\mu}, C'_\mu(x), 1^{p(|x|)}\rangle] > p'(n)^{-1} \cdot 2^{-|C_\mu(x)|}. \tag{10.3}$$

[13] Note that $|y| = \mathrm{poly}(|x|)$, but $|x| = \mathrm{poly}(|C'_\mu(x)|)$ does not necessarily hold (and so S' is not necessarily in $\mathcal{NP}$). As we shall see, the latter point is immaterial.

By virtue of the compression condition (of the Coding Lemma), we have $2^{-|C_\mu(x)|} \geq 2^{-1-\min(|x|,\log_2(1/\mu'(x)))}$. It follows that

$$2^{-|C_\mu(x)|} \geq \Pr[X_{|x|} = x]/2. \tag{10.4}$$

Recalling that x is the only preimage that is mapped to $\langle M_{S,\mu}, C'_\mu(x), 1^{p(|x|)}\rangle$ and combining Eq. (10.3) & (10.4), we establish the domination condition.

The theorem follows. ■

Reflections. The proof of Theorem 10.17 demonstrates the fact that, unlike more advanced worst-case reductions, the reduction used in the proof of Theorem 2.18 does not introduce much structure in the reduced instances (i.e., does not reduce the original problem to a "highly structured special case" of the target problem). Put in other words, the latter reduction does not map "random" (i.e., uniformly distributed) instances to highly structured instances (which occur with negligible probability under the uniform distribution). Thus, the reduction used in the proof of Theorem 2.18 suffices for reducing any distributional problem in dist$\mathcal{NP}$ to a distributional problem consisting of S_{u} coupled with *some* simple probability ensemble.[14]

However, Theorem 10.17 states more than the latter assertion. That is, it states that any distributional problem in dist$\mathcal{NP}$ is reducible to the same distributional version of S_{u}. Indeed, the effort involved in proving Theorem 10.17 was due to the need for mapping instances taken from any simple probability ensemble (which may not be the uniform ensemble) to instances distributed in a manner that is dominated by a single probability ensemble (i.e., the quasi-uniform ensemble U').

Once we have established the existence of one dist$\mathcal{NP}$-complete problem, we may establish the dist$\mathcal{NP}$-completeness of other problems in dist$\mathcal{NP}$ by reducing any dist$\mathcal{NP}$-complete problem to them (and relying on the transitivity of reductions (see Exercise 10.9)). Thus, the difficulties encountered in the proof of Theorem 10.17 are no longer relevant. Unfortunately, a seemingly more severe difficulty arises: almost all know reductions in the theory of NP-completeness work by introducing much structure in the reduced instances (i.e., they actually reduce to highly structured special cases). Furthermore, this structure is too complex in the sense that the distribution of reduced instances does not seem simple (in the sense of Definition 10.15). Designing reductions that avoid the introduction of such structure has turned out to be quite difficult; still several such reductions are cited in [85].

10.2.1.3 Probabilistic versions

The definitions in §10.2.1.1 can be easily extended to refer to randomized algorithms. For example, extending Definition 10.14, we have:

[14]Note that this cannot be said of most known Karp-reductions.

Definition 10.18 (the class tpc$\mathcal{BPP}$): *For a probabilistic algorithm A, a Boolean function B, and $t : \mathbb{N} \to \mathbb{N}$, we say that the string x is* t-bad for A with respect to B *if with probability exceeding $1/3$, on input x, either $A(x) \neq B(x)$ or A runs more that $t(|x|)$ steps. We say that A* typically solves $(S, \{X_n\}_{n\in\mathbb{N}})$ in probabilistic polynomial-time *if there exists a polynomial p such that the probability that X_n is p-bad for A with respect to the characteristic function of S is negligible. We denote by* tpc$\mathcal{BPP}$ *the class of distributional problems that are typically solvable in probabilistic polynomial-time.*

The definition of reductions can be similarly extended. This means that in Definition 10.16, both $M^T(x)$ and $Q(x)$ (mentioned in Items 2 and 3, respectively) are random variables rather than fixed objects. Furthermore, validity is required to hold (for every input) only with probability 2/3, where the probability space refers only to the internal coin tosses of the reduction. Randomized reductions are closed under composition and preserve typical feasibility (see Exercise 10.12).

Randomized reductions allow the presentation of a dist$\mathcal{NP}$-complete problem that refers to the (perfectly) uniform ensemble. Recall that Theorem 10.17 establishes the dist$\mathcal{NP}$-completeness of $(S_{\mathtt{u}}, U')$, where U' is a quasi-uniform ensemble (i.e., $\Pr[U'_n = \langle M, x, 1^t\rangle] = 2^{-(|M|+|x|)}/\binom{n}{2}$, where $n = |\langle M, x, 1^t\rangle|$). We first note that $(S_{\mathtt{u}}, U')$ can be randomly reduced to $(S'_{\mathtt{u}}, U'')$, where $S'_{\mathtt{u}} = \{\langle M, x, z\rangle : \langle M, x, 1^{|z|}\rangle \in S_{\mathtt{u}}\}$ and $\Pr[U''_n = \langle M, x, z\rangle] = 2^{-(|M|+|x|+|z|)}/\binom{n}{2}$ for every $\langle M, x, z\rangle \in \{0,1\}^n$. The randomized reduction consists of mapping $\langle M, x, 1^t\rangle$ to $\langle M, x, z\rangle$, where z is uniformly selected in $\{0,1\}^t$. Recalling that $U = \{U_n\}_{n\in\mathbb{N}}$ denotes the uniform probability ensemble (i.e., U_n is uniformly distributed on strings of length n) and using a suitable encoding we get.

Proposition 10.19 *There exists $S \in \mathcal{NP}$ such that every $(S', X') \in$ dist$\mathcal{NP}$ is randomly reducible to (S, U).*

Proof Sketch: By the forgoing discussion, every $(S', X') \in$ dist$\mathcal{NP}$ is randomly reducible to $(S'_{\mathtt{u}}, U'')$. Thus, we focus on reducing $(S'_{\mathtt{u}}, U'')$ to $(S''_{\mathtt{u}}, U)$, where $S''_{\mathtt{u}} \in \mathcal{NP}$ is defined as follows. The string $\alpha\beta uvw$ is in $S''_{\mathtt{u}}$ if $\langle u, v, w\rangle \in S'_{\mathtt{u}}$ and α (resp., β) represents the binary encoding of the integer $|u|$ (resp., $|v|$), where the encoding is padded with zeros to a total length of $\log_2 |uvw|$. The reduction maps $\langle M, x, z\rangle$ to the string $\alpha \cdot \beta \cdot M \cdot x \cdot z$, where α (resp., β) represents the binary encoding of $|M|$ (resp., $|x|$) padded with zeros to a total length of $\log_2(|M| + |x| + |z|)$. Noting that this reduction satisfies all conditions of Definition 10.16, the proposition follows. □

10.2.2 Ramifications

In our opinion, the most problematic aspect of the theory described in Section 10.2.1 is the definition of simple probability ensembles, which in turn restricts the definition of dist$\mathcal{NP}$ (Definition 10.15). This restriction strengthens the conjecture that dist$\mathcal{NP}$ is not contained in tpc$\mathcal{BPP}$, which means that it weakens conditional results that are based on this conjecture. An appealing extension of the class dist$\mathcal{NP}$ is presented in §10.2.2.2, where it is shown that if the extended class is not contained in tpc$\mathcal{BPP}$ then dist$\mathcal{NP}$ itself is not contained in tpc$\mathcal{BPP}$. Thus, dist$\mathcal{NP}$-complete problems enjoy the benefit of both being in the more restricted class (i.e., dist$\mathcal{NP}$) and being hard as long as some problems in the extended class is hard.

In §10.2.2.1, we extend the treatment from decision problems to search problems. This extension is motivated by the realization that search problem are actually of greater importance to real-life applications (cf. Section 2.1.1), and hence a theory motivated by real-life applications must address such problems, as we do next.

10.2.2.1 Search versus Decision

Indeed, as in the case of worst-case complexity, search problems are at least as important as decision problems. Thus, an average-case treatment of search problems is indeed called for. We first present distributional versions of $\mathcal{PF}$ and $\mathcal{PC}$ (cf. Section 2.1.1), following the underlying principles of the definitions of tpc$\mathcal{P}$ and dist$\mathcal{NP}$.

Definition 10.20 (the classes tpc$\mathcal{PF}$ and dist$\mathcal{PC}$): *As in Section 2.1.1, we consider only polynomially bounded search problems; that is, binary relations $R \subseteq \{0,1\}^* \times \{0,1\}^*$ such that for some polynomial q it holds that $(x,y) \in R$ implies $|y| \leq q(|x|)$. Recall that $R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\}$.*

- *A* distributional search problem *consists of a polynomially bounded search problem coupled with a probability ensemble.*
- *The class* tpc$\mathcal{PF}$ *consists of all distributional search problems that are typically solvable in polynomial-time. That is, $(R, \{X_n\}_{n\in\mathbb{N}}) \in$ tpc$\mathcal{PF}$ if there exists an algorithm A and a polynomial p such that the probability that on input X_n algorithm A either errs or runs more that $p(n)$ steps is negligible, where A errs on x if $A(x) \notin R(x)$ in case $R(x) \neq \emptyset$ and $A(x) \neq \perp$ otherwise.*
- *A distributional search problem (R, X) is in* dist$\mathcal{PC}$ *if $R \in \mathcal{PC}$ and X is simple* (as in Definition 10.15)*.*

Likewise, the class tpc$\mathcal{BPPF}$ consists of all distributional search problems that are typically solvable in *probabilistic* polynomial-time (cf., Definition 10.18). The definitions of *reductions among distributional problems*, presented in the context of decision problem, extend to search problems.

Fortunately, as in the context of worst-case complexity, the study of distributional search problems "reduces" to the study of distributional decision problems.

Theorem 10.21 (reducing search to decision): dist$\mathcal{PC} \subseteq$ tpc$\mathcal{BPPF}$ *if and only if* dist$\mathcal{NP} \subseteq$ tpc$\mathcal{BPP}$*. Furthermore, every problem in* dist$\mathcal{NP}$ *is reducible to some problem in* dist$\mathcal{PC}$*, and every problem in* dist$\mathcal{PC}$ *is* randomly *reducible to some problem in* dist$\mathcal{NP}$*.*

Proof Sketch: The furthermore part is analogous to the actual contents of the proof of Theorem 2.6 (see also Step 1 in the proof of Theorem 2.15). Indeed the reduction of $\mathcal{NP}$ to $\mathcal{PC}$ presented in the proof of Theorem 2.6 extends to the current context. Specifically, for any $S \in \mathcal{NP}$, we consider a relation $R \in \mathcal{PC}$ such that $S = \{x : R(x) \neq \emptyset\}$, and note that, for any probability ensemble X, the identity transformation reduces (S, X) to (R, X).

A difficulty arises in the opposite direction. Recall that in the proof of Theorem 2.6 we reduced the search problem of $R \in \mathcal{PC}$ to deciding membership in $S'_R \stackrel{\text{def}}{=} \{\langle x, y'\rangle : \exists y'' \text{ s.t. } (x, y'y'') \in R\} \in \mathcal{NP}$. The difficulty encountered here is that, on input x, this reduction makes queries of the form $\langle x, y'\rangle$, where y' is a prefix of some string in $R(x)$. These queries may induce a distribution that is not dominated by any simple distribution. Thus, we seek an alternative reduction.

As a warm-up, let us assume for a moment that R has unique solutions (in the sense of Definition 6.26); that is, for every x it holds that $|R(x)| \leq 1$. In this case we may easily reduce the search problem of $R \in \mathcal{PC}$ to deciding membership in $S''_R \in \mathcal{NP}$, where $\langle x, i, \sigma\rangle \in S''_R$ if and only if $R(x)$ contains a string in which the i^{th} bit equals σ. Specifically, on input x, the reduction issues the queries $\langle x, i, \sigma\rangle$, where $i \in [\ell]$ (with $\ell = \text{poly}(|x|)$) and $\sigma \in \{0, 1\}$, which allows for determining the single string in $R(x)$ (whenever such a string exists). The point is that this reduction can be used to reduce any $(R, X) \in \text{dist}\mathcal{PC}$ (having unique solutions) to $(S''_R, X'') \in \text{dist}\mathcal{NP}$, where X'' equally distributes the probability mass of x (under X) to all the tuples $\langle x, i, \sigma\rangle$; that is, for every $i \in [\ell]$ and $\sigma \in \{0, 1\}$, it holds that $\mathsf{Pr}[X''_{|\langle x,i,\sigma\rangle|} = \langle x, i, \sigma\rangle]$ equals $\mathsf{Pr}[X_{|x|} = x]/2\ell$.

Unfortunately, in the general case, R may not have unique solutions. Nevertheless, applying the main idea that underlies the proof of Theorem 6.27, this difficulty can be overcome. We first note that the foregoing mapping of instances of the distributional problem $(R, X) \in \text{dist}\mathcal{PC}$ to instances of $(S''_R, X'') \in \text{dist}\mathcal{NP}$ satisfies the efficiency and domination condition even in the case that R does not have unique solutions. What may possibly fail (in the general case) is the validity condition (i.e., if $|R(x)| > 1$ then we may fail to recover any element of $R(x)$).

Recall that the main part of the proof of Theorem 6.27 is a randomized reduction that maps instances of R to triples of the form (x, m, h), where m is an integer and h is a hashing function that are uniformly distributed in some adequate sets $[\ell]$ and H^m_ℓ, where $\ell = \text{poly}(|x|)$ and H^m_ℓ is as in Appendix D.2. Furthermore, if $R(x) \neq \emptyset$ then, with probability $\Omega(1/\ell)$ over yje choices of $m \in [\ell]$ and $h \in H^m_\ell$, there exists a unique $y \in R(x)$ such that $h(y) = 0^m$. Defining $R'(x, m, h) \stackrel{\text{def}}{=} \{y \in R : h(y) = 0^m\}$, this yields a randomized reduction of the search problem of R to the search problem of R' such that with noticeable probability[15] the reduction maps instances that have solutions to instances having a unique solution. Furthermore, this reduction can be used to reduce any $(R, X) \in \text{dist}\mathcal{PC}$ to $(R', X') \in \text{dist}\mathcal{PC}$, where X' distributes the probability mass of x (under X) to all the triples (x, m, h) such that for every $m \in [\ell]$ and $h \in H^m_\ell$ it holds that $\mathsf{Pr}[X''_{|(x,m,h)|} = (x, m, h)]$

[15]Recall that the probability of an event is said to be noticeable (in a relevant parameter) if it is greater than the reciprocal of some positive polynomial. In the context of randomized reductions, the relevant parameter is the length of the input to the reduction.

equals $\mathsf{Pr}[X_{|x|} = x]/(\ell \cdot |H_\ell^m|)$. (Note that with a suitable encoding, X' is indeed simple.)

The theorem follows by combining the two aforementioned reductions. That is, we first apply the randomized reduction of (R, X) to (R', X'), and next reduce the resulting instance to an instance of the corresponding decision problem $(S''_{R'}, X'')$. The combined randomized mapping satisfies the efficiency and domination conditions, and is valid with noticeable probability. The error probability can be made negligible by straightforward amplification (see Exercise 10.12). □

10.2.2.2 Simple versus sampleable distributions

Recall that the definition of simple probability ensembles (underlying Definition 10.15) requires that the accumulating distribution function (as defined in the proof of Theorem 10.17) is polynomial-time computable. Recall that $\mu : \{0,1\}^* \to [0,1]$ is called the accumulating distribution function of $X = \{X_n\}_{n\in\mathbb{N}}$ if for every $n \in \mathbb{N}$ and $x \in \{0,1\}^n$ it holds that $\mu(x) \stackrel{\text{def}}{=} \mathsf{Pr}[X_n \leq x]$, where the inequality refers to the standard lexicographic order of n-bit strings.

As argued in §10.2.1.1, the requirement that the accumulating distribution function is polynomial-time computable imposes severe restrictions on the set of admissible ensembles. Furthermore, it seems that these simple ensembles are indeed "simple" in some intuitive sense and hence represent distributions that may occur in practice. However, a more robust definition of the latter is offered by the notion of polynomial-time sampleable ensembles (underlying Definition 10.22). We believe that the class of such ensembles contains all distributions that may occur in practice, because we believe that the real world should be modeled as a *feasible* (rather than an arbitrary) randomized process

Definition 10.22 (sampleable ensembles and the class samp$\mathcal{NP}$): *We say that a probability ensemble $X = \{X_n\}_{n\in\mathbb{N}}$ is* (polynomial-time) sampleable *if there exists a probabilistic polynomial-time algorithm A such that for every $x \in \{0,1\}^*$ it holds that $\mathsf{Pr}[A(1^{|x|}) = x] = \mathsf{Pr}[X_{|x|} = x]$. We denote by* samp$\mathcal{NP}$ *the class of distributional problems consisting of decision problems in $\mathcal{NP}$ coupled with sampleable probability ensembles.*

We first note that all simple probability ensembles are indeed sampleable (see Exercise 10.13), and thus dist$\mathcal{NP} \subseteq$ samp$\mathcal{NP}$. On the other hand, it seems that there are sampleable probability ensembles that are not simple (see Exercise 10.14). In fact, extending the scope of distributional problems (from dist$\mathcal{NP}$ to samp$\mathcal{NP}$) allows proving that every NP-complete problem has a distributional version in samp$\mathcal{NP}$ that is dist$\mathcal{NP}$-hard (see Exercise 10.15). Furthermore, it is possible to prove that all natural NP-complete problem have distributional versions that are samp$\mathcal{NP}$-complete.

Theorem 10.23 (samp$\mathcal{NP}$-completeness): *Suppose that $S \in \mathcal{NP}$ and that every set in $\mathcal{NP}$ is reducible to S by a Karp-reduction that does not shrink the input. Then there exists a polynomial-time sampleable ensemble X such that any problem in* samp$\mathcal{NP}$ *is reducible to (S, X)*

The proof of Theorem 10.23 is based on the observation that there exists a polynomial-time sampleable ensemble that dominates all polynomial-time sampleable ensembles. The existence of this ensemble is based on the notion of a universal (sampling) machine. For further details see Exercise 10.16. (Recall that when proving Theorem 10.17, we did not establish an analogous result for simple ensembles (but rather capitalized on the universal nature of $S_{\mathtt{u}}$).)

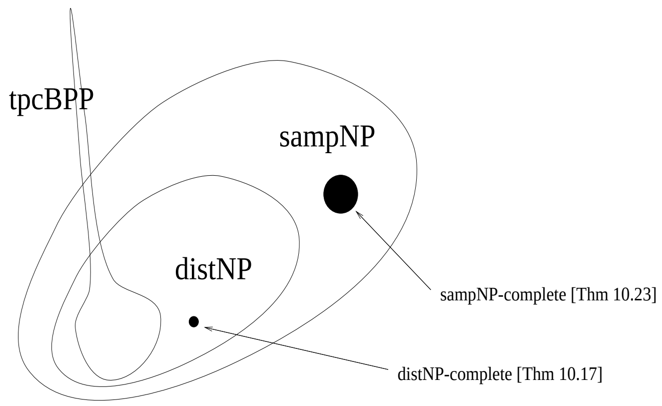


Figure 10.1: Two types of average-case completeness

Theorem 10.23 establishes a rich theory of samp$\mathcal{NP}$-completeness, but does not relate this theory to the previously presented theory of dist$\mathcal{NP}$-completeness (see Figure 10.1). This is done in the next theorem, which asserts that the existence of typically hard problems in samp$\mathcal{NP}$ implies their existence in dist$\mathcal{NP}$.

Theorem 10.24 (samp$\mathcal{NP}$-completeness versus dist$\mathcal{NP}$-completeness): *If* samp$\mathcal{NP}$ *is not contained in* tpc$\mathcal{BPP}$ *then* dist$\mathcal{NP}$ *is not contained in* tpc$\mathcal{BPP}$.

Thus, the two "typical-case complexity" versions of the P-vs-NP Question are equivalent. That is, if some "sampleable distribution" versions of NP are not typically feasible then some "simple distribution" versions of NP are not typically feasible. In particular, if samp$\mathcal{NP}$-complete problems are not in tpc$\mathcal{BPP}$ then dist$\mathcal{NP}$-complete problems are not in tpc$\mathcal{BPP}$.

The foregoing assertions would all follow if samp$\mathcal{NP}$ were (randomly) reducible to dist$\mathcal{NP}$ (i.e., if every problem in samp$\mathcal{NP}$ were reducible (under a randomized version of Definition 10.16) to some problem in dist$\mathcal{NP}$); but, unfortunately, we do not know whether such reductions exist. Yet, underlying the proof of Theorem 10.24 is a more liberal notion of a reduction among distributional problem.

Proof Sketch: We shall prove that if dist$\mathcal{NP}$ is contained in tpc$\mathcal{BPP}$ then the same holds for samp$\mathcal{NP}$ (i.e., samp$\mathcal{NP}$ is contained in tpc$\mathcal{BPP}$). Actually, we shall show that if dist$\mathcal{PC}$ is contained in tpc$\mathcal{BPPF}$ then the sampleable version of dist$\mathcal{PC}$, denoted samp$\mathcal{PC}$, is contained in tpc$\mathcal{BPPF}$ (and refer to Exercise 10.17).

Specifically, we shall show that under a relaxed notion of a randomized reduction, every problem in samp$\mathcal{PC}$ is reduced to some problem in dist$\mathcal{PC}$. Loosely speaking, this relaxed notion of a randomized reduction requires only a noticeable fraction of the probability space of the reduction to satisfies the validity and domination conditions (of Definition 10.16, when adapted to randomized reductions). We start by formulating this notion, when referring to distributional *search* problems.

A relaxed reduction of the distributional problem (R, X) to the distributional problem (T, Y) is a probabilistic polynomial-time oracle machine M that satisfies the following conditions:

Notation: For every $x \in \{0,1\}^*$, we denote by $m(|x|) = \text{poly}(|x|)$ the number of internal coin tosses of M on input x, and denote by $M^T(x, r)$ the execution of M on input x, internal coins $r \in \{0,1\}^m$, and oracle access to T.

Validity: For every $x \in \{0,1\}^*$, there exists a set $\Omega_x \subseteq \{0,1\}^{m(|x|)}$ of size at least $\rho(|x|) \cdot 2m(|x|)$, where $\rho(|x|) > 1/\text{poly}(|x|)$ such that for every $r \in \Omega_x$ the reduction yields a correct answer (i.e., $M^T(x, r) \in R(x)$ if $R(x) \neq \emptyset$ and $M^T(x, r) = \perp$ otherwise).

Domination: There exists a positive polynomial p such that, for every $y \in \{0,1\}^*$, it holds that

$$\max_{n \in \mathbb{N}}\{\mathsf{Pr}[Q'(X_n) \ni y]\} \leq p(|y|) \cdot \mathsf{Pr}[Y_{|y|} = y], \tag{10.5}$$

where $Q'(x)$ is a random variable, defined over the set Ω_x (as in the validity condition), representing the set of queries made by M on input x and oracle access to T. That is, $Q'(x)$ is defined by uniformly selecting $r \in \Omega_x$ and considering the set of queries made by M on input x, internal coins r, and oracle access to T.

The reader may verify that this relaxed notion of a reduction preserves typical feasibility; that is, for $R \in \mathcal{PC}$, if there exists a relaxed reduction of (R, X) to (T, Y) and (T, Y) is in tpc$\mathcal{BPPF}$ then (R, X) is in tpc$\mathcal{BPPF}$. The key observation is that the analysis may discard the case that, on input x, the reduction selects coins not in Ω_x. Indeed, the queries made in that case may be untypical and the answers received may be wrong, but this is immaterial (because correct solutions can be recognized using $R \in \mathcal{PC}$). That is, if x has a solution then with noticeable probability the reduction will find one and output it, whereas the reduction will never output a wrong solution.

Our goal is presenting, for every $(R, X) \in$ samp$\mathcal{PC}$, a relaxed reduction of (R, X) to a related problem $(R', X') \in$ dist$\mathcal{PC}$. (As usual, let $X = \{X_n\}_{n \in \mathbb{N}}$ and similarly for X'.) For starters, *suppose that X_n is uniformly distributed on some set $S_n \subseteq \{0,1\}^n$ and that there is a polynomial-time computable and invertible mapping μ of S_n to $\{0,1\}^{\ell(n)}$, where $\ell(n) = \log_2 |S_n|$*. Then, mapping x to $1^{|x| - \ell(|x|)}0\mu(x)$, we obtain a reduction of (R, X) to (R', X'), where X'_{n+1} is uniform over $\{1^{n-\ell(n)}0v : v \in \{0,1\}^{\ell(n)}\}$ and $R'(1^{n-\ell(n)}0v) = R(\mu^{-1}(v))$ (or, equivalently, $R(x) = R'(1^{|x|-\ell(|x|)}0\mu(x))$). Note that X' is a simple ensemble and $R' \in \mathcal{PC}$;

hence, $(R',X') \in \text{dist}\mathcal{PC}$. Also note that the foregoing mapping is indeed a valid reduction (i.e., it satisfies the efficiency, validity, and domination conditions). Thus, (R,X) is reduced to a problem in $\text{dist}\mathcal{PC}$ (and indeed the relaxation was not used here).

Next, we drop the assumption that there is a polynomial-time computable and invertible mapping μ of S_n to $\{0,1\}^{\ell(n)}$, but maintain the assumption that X_n is uniform on some set $S_n \subseteq \{0,1\}^n$ and assume that $|S_n| = 2^{\ell(n)}$ is easily computable (from n). In this case, we may map $x \in \{0,1\}^n$ to its image under a suitable randomly chosen hashing function h, which in particular maps n-bit strings to $\ell(n)$-bit strings. That is, we randomly map x to $(h, 1^{n-\ell(n)}0h(x))$, where h is uniformly selected in a set $H_n^{\ell(n)}$ of suitable hash functions (see Appendix D.2). This calls for redefining R' such that $R'(h, 1^{n-\ell(n)}0v)$ corresponds to the preimages of v under h that are in S_n. Assuming that h is a 1-1 mapping of S_n to $\{0,1\}^{\ell(n)}$, we may define $R'(h, 1^{n-\ell(n)}0v) = R(x)$ where x is the unique string satisfying $x \in S_n$ and $h(x) = v$, where the condition $x \in S_n$ may be *verified by providing the internal coins of the sampling procedure that generate x*. Denoting the sampling procedure of X by S, and letting $S(1^n, r)$ denote the output of S on input 1^n and internal coins r, we actually redefine R' as

$$R'(h, 1^{n-\ell(n)}0v) = \{\langle r,y\rangle : h(S(1^n,r))=v \wedge y \in R(S(1^n,r))\}. \tag{10.6}$$

We note that $\langle r,y\rangle \in R'(h, 1^{|x|-\ell(|x|)}0h(x))$ yields a solution $y \in R(x)$ if $S(1^{|x|},r) = x$, but otherwise "all bets are off" (as y will be a solution for $S(1^{|x|},r) \neq x$). Now, although typically h will not be a 1-1 mapping of S_n to $\{0,1\}^{\ell(n)}$, for each $x \in S_n$, with constant probability over the choice of h, it holds that $h(x)$ has a unique preimage in S_n under h. (See the proof of Theorem 6.27.) In this case $\langle r,y\rangle \in R'(h, 1^{|x|-\ell(|x|)}0h(x))$ implies $S(1^{|x|},r) = x$ (which, in turn, implies $y \in R(x)$). We claim that *the randomized mapping of x to $(h, 1^{n-\ell(n)}0h(x))$, where h is uniformly selected in $H_{|x|}^{\ell(|x|)}$, yields a relaxed reduction of (R,X) to (R',X'), where $X'_{n'}$ is uniform over $H_n^{\ell(n)} \times \{1^{n-\ell(n)}0v : v \in \{0,1\}^{\ell(n)}\}$.* (Needless to say, the claim refers to the reduction that makes the query $(h, 1^{n-\ell(n)}0h(x))$ and returns y if the oracle answer equals $\langle r,y\rangle$ and $y \in R(x)$.)

The claim is proved by considering the set Ω_x of choices of $h \in H_{|x|}^{\ell(|x|)}$ for which $x \in S_n$ is the only preimage of $h(x)$ under h that resides in S_n (i.e., $|\{x' \in S_n : h(x') = h(x)\}| = 1$). In this case (i.e., $h \in \Omega_x$) it holds that $\langle r,y\rangle \in R'(h, 1^{|x|-\ell(|x|)}0h(x))$ implies that $S(1^{|x|},r) = x$ and $y \in R(x)$, and the (relaxed) validity condition follows. The (relaxed) domination condition follows by noting that $\mathsf{Pr}[X_n = x] \approx 2^{-\ell(|x|)}$, that x is mapped to $(h, 1^{|x|-\ell(|x|)}0h(x))$ with probability $1/|H_{|x|}^{\ell(|x|)}|$, and that x is the only preimage of $(h, 1^{|x|-\ell(|x|)}0h(x))$ under the mapping (among $x' \in S_n$ such that $\Omega_{x'} \ni h$).

Before going any further, let us highlight the importance of hashing X_n to $\ell(n)$-bit strings. On one hand, this mapping is "typically" one-to-one, and thus (with constant probability) the solution provided for the hashed instance (i.e., $h(x)$) yield a solution for the original instance (i.e., x). This guarantees the validity of the reduction. On the other hand, for a typical h, the mapping of X_n to $h(X_n)$ covers the relevant range almost uniformly. This guarantees that the reduction satisfies the domination condition. Note that these two phenomena impose conflicting requirements that are both met at the correct value of ℓ; that is, the one-to-one condition requires $\ell(n) \geq \log_2 |S_n|$, whereas an almost uniform cover requires $\ell(n) \leq \log_2 |S_n|$. Also note that $\ell(n) = \log_2(1/\Pr[X_n = x])$ for every x in the support of X_n; the latter quantity will be in our focus in the general case.

Finally, we need to get rid of the assumption that X_n is *uniformly distributed* over some subset of $\{0,1\}^n$. All that we know is that there exists a probabilistic polynomial-time ("sampling") algorithm S such that $S(1^n)$ is distributed identically to X_n. In this (general) case, we map instances of (R, X) according to their probability mass such that x is mapped to an instance (of R') that consists of $(h, h(x))$ and additional information, where h is a random hash function mapping n-bit long string to strings of length $\ell_x \stackrel{\text{def}}{=} \lceil \log_2(1/\Pr[X_{|x|} = x]) \rceil$. Since (in the general case) there may be more than 2^{ℓ_x} strings in the support of X_n, we need to augment the reduced instance in order to ensure that it is uniquely associated with x. The basic idea is augmenting the mapping of x to $(h, h(x))$ with additional information that restricts X_n to strings that occur with probability at least $2^{-\ell_x}$.

Let $q(n)$ denote the randomness complexity of S and $S(1^n, r)$ denote the output of S on input 1^n and internal coin tosses $r \in \{0,1\}^{q(n)}$. Then, we randomly map x to $(h, h(x), h', v')$, where $h : \{0,1\}^{|x|} \to \{0,1\}^{\ell_x}$ and $h' : \{0,1\}^{q(|x|)} \to \{0,1\}^{q(|x|)-\ell_x}$ are random hash functions and $v' \in \{0,1\}^{q(|x|)-\ell_x}$ is uniformly distributed. The instance (h, v, h', v') of R' has solutions that consists of pairs $\langle r, y \rangle$ such that $h(S(1^n, r)) = v \wedge h'(r) = v'$ and $y \in R(S(1^n, r))$. As we shall see, this augmentation guarantees that, with constant probability (over the choice of h, h', v'), the solutions to the reduced instance $(h, h(x), h', v')$ correspond to the solutions to the original instance x.

The foregoing description assumes that, on input x, we can determine ℓ_x, which is an assumption that we cannot justify. Instead, we may just select ℓ uniformly in $\{0, 1, ..., q(|x|)\}$ and be correct with noticeable probability (i.e., $\Pr[\ell = \ell_x] = 1/(q(|x|) + 1) = 1/\text{poly}(|x|)$). Furthermore, for clarity, we make n and ℓ explicit in the reduced instance. Specifically, we randomly map $x \in \{0,1\}^n$, to $(1^n, 1^\ell, h, h(x), h', v') \in \{0,1\}^{n'}$, where $\ell \in \{0, 1, ..., q(n)\}$, $h \in H_n^\ell$, $h' \in H_{q(n)}^{q(n)-\ell}$, and $v' \in \{0,1\}^{q(n)-\ell}$ are uniformly distributed.[16] This mapping will be used to reduce (R, X) to (R', X'), where

$$R'(1^n, 1^\ell, h, v, h', v') = \{\langle r, y \rangle : h(S(1^n, r)) = v \wedge h'(r) = v' \wedge y \in R(S(1^n, r))\} \quad (10.7)$$

and $X'_{n'}$ assigns equal probability to each $X_{n',\ell}$ (for $\ell \in \{0, 1, ..., n\}$), and each $X_{n',\ell}$ is isomorphic to the uniform distribution over $H_n^\ell \times \{0,1\}^\ell \times H_{q(n)}^{q(n)-\ell} \times \{0,1\}^{q(n)-\ell}$. Note that indeed $(R', X') \in \text{dist}\mathcal{PC}$.

The aforementioned randomized mapping is analyzed by considering the correct choice for ℓ; that is, on input x, we focus on the choice $\ell = \ell_x$. Under this condi-

[16] As in other places, a suitable encoding will be used such that the reduction maps strings of the same length to strings of the same length (i.e., n-bit string are mapped to n'-bit strings, for $n' = \text{poly}(n)$). For example, we may encode $\langle 1^n, 1^\ell, h, h(x), h', v' \rangle$ as $1^n 0 1^\ell 0 1^{q(n)-\ell} 0 \langle h \rangle \langle h(x) \rangle \langle h' \rangle \langle v' \rangle$, where each $\langle w \rangle$ denotes an encoding of w by a string of length $(n' - (n + q(n) + 3))/4$.

tioning (as we shall show), *with constant probability over the choice of* h, h' *and* v', *the instance* x *is the only value* x' (of X_n) *that is mapped to* $(1^n, 1^\ell, h, h(x), h', v')$ *such that there exists* r *that satisfies* $S(1^n, r) = x'$ *and* $h'(r) = v'$. It follows that (for such h, h' and v') any solution $\langle r, y\rangle \in R'(1^n, 1^\ell, h, h(x), h', v')$ satisfies $S(1^n, r) = x$ and thus $y \in R(x)$, which means that the validity condition is satisfied. The domination condition is satisfied too, because (for such h, h' and v') the probability that X_n is mapped to $(1^n, 1^\ell, h, h(x), h', v')$ approximately equals $\Pr[X'_{n',\ell} = (1^n, 1^\ell, h, h(x), h', v')]$.

We now turn to analyze the probability, over the choice of h, h' and v', that the instance x is the only value x' (of X_n) that is mapped to $(1^n, 1^{\ell_x}, h, h(x), h', v')$ such that there exists r that satisfies $S(1^n, r) = x'$ and $h'(r) = v'$. Firstly, we note that $|\{r : S(1^n, r) = x\}| \geq 2^{q(n)-\ell_x}$, and thus, with constant probability over the choice of $h' \in H_{q(n)}^{q(n)-\ell_x}$ and $v' \in \{0,1\}^{q(n)-\ell_x}$, there exists r that satisfies $S(1^n, r) = x$ and $h'(r) = v'$. Next, we note that, with constant probability over the choice of $h \in H_n^{\ell_x}$, it holds that x is the only string having probability mass at least $2^{-\ell_x}$ (under X_n) that is mapped to $h(x)$ under h. Finally, we prove that, with constant probability over the choice of $h \in H_n^{\ell_x}$ and $h' \in H_{q(n)}^{q(n)-\ell_x}$ (and even when conditioning on the previous items), the mapping $r \mapsto (h(S(1^n, r)), h'(r))$ maps the set $\{r : \Pr[X_n = S(1^n, r)] \leq 2^{-\ell_x}\}$ almost uniformly to $\{0,1\}^{q(n)}$. Specifically, with constant probability, no other r is mapped to the aforementioned pair $(h(x), v')$. Thus, the claim follows and so does the theorem. □

Reflection. Theorem 10.24 implies that if samp$\mathcal{NP}$ is not contained in tpc$\mathcal{BPP}$ then every dist$\mathcal{NP}$-complete problem is not in tpc$\mathcal{BPP}$. This means that the hardness of some distributional problems that refer to sampleable distributions implies the hardness of some distributional problems that refer to simple distributions. Furthermore, by Proposition 10.19, this implies the hardness of distributional problems that refer to the uniform distribution. Thus, hardness with respect to some distribution in an utmost wide class (which arguably captures all distributions that may occur in practice) implies hardness with respect to a single simple distribution (which arguably is the simplest one).

Relation to one-way functions. We note that the existence of one-way functions (see Section 7.1) implies the existence of problems in samp$\mathcal{PC}$ that are not in tpc$\mathcal{BPPF}$ (which in turn implies the existence of such problems in dist$\mathcal{PC}$). Specifically, for a length-preserving one-way function f, consider the distributional search problem $(R_f, \{f(U_n)\}_{n\in\mathbb{N}})$, where $R_f = \{(f(r), r) : r \in \{0,1\}^*\}$.[17] On the other hand, it is not known whether the existence of a problem in samp$\mathcal{PC} \setminus$ tpc$\mathcal{BPPF}$ implies the existence of one-way functions. In particular, the existence of a problem (R, X) in samp$\mathcal{PC} \setminus$ tpc$\mathcal{BPPF}$ represents the feasibility of generating hard instances for the search problem R, whereas the existence of one-way function represents the feasibility of generating instance-solution pairs such that the instances

[17] Note that the distribution $f(U_n)$ is uniform in the special case that f is a permutation over $\{0,1\}^n$.

are hard to solve (see Section 7.1.1). Indeed, the gap refers to whether or not *hard instances can be efficiently generated together with corresponding solutions.* Our world view is thus depicted in Figure 10.2, where lower levels indicate seemingly weaker assumptions.

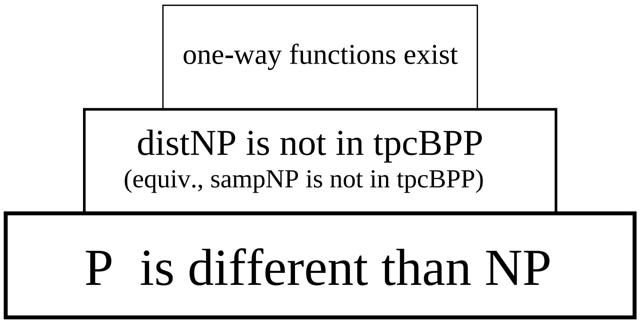


Figure 10.2: Worst-case vs average-case assumptions

Chapter Notes

In this chapter, we presented two different approaches to the relaxation of computational problems. The first approach refers to the concept of approximation, while the second approach refers to average-case analysis. We demonstrated that various natural notions of approximation can be cast within the framework of promise problems (see Section 2.4.1). In contrast, the study of average-case complexity requires the introduction of a new conceptual framework and addressing various definitional issues.

A natural question at this point is what have we gained by relaxing the requirements. In the context of approximation, the answer is mixed: in some natural cases we gain alot (i.e., we obtained feasible relaxations of hard problems), while in other natural cases we gain nothing (i.e., even extreme relaxations remain as intractable as the original version). In the context of average-case complexity, the answer appears to be negative. In particular, assuming the existence of one-way functions, every natural NP-complete problem has a distributional version that is hard (and refers to a sampleable ensemble). Furthermore, in this case, some problems in NP have hard distributional versions that refer to the uniform distribution.

Another difference between the two approaches is that the theory of approximation seems to lack a comprehensive structure, whereas the theory of average-case complexity seems to have a too rigid structure (which foils attempts to present more appealing dist$\mathcal{NP}$-complete problems).

Approximation

The following bibliographic comments are quite laconic and neglect mentioning various important works (including credits for some of the results mentioned in our text). As usual, the interested reader is referred to corresponding surveys.

Search or Optimization. The interest in approximation algorithms increased considerably following the demonstration of the NP-completeness of many natural optimization problems. But, with some exceptions (most notably [166]), the systematic study of the complexity of such problems stalled till the discovery of the "PCP connection" (see Section 9.3.3) by Feige, Goldwasser, Lovász, and Safra [69]. Indeed the relatively "tight" inapproximation results for max-Clique, max-SAT, and the maximization of linear equations, due to Håstad [111, 112], build on previous work regarding PCP and their connection to approximation (cf., e.g., [70, 14, 13, 27, 172]). Specifically, Theorem 10.5 is due to [111], while Theorems 10.8 and 10.9 are due to [112]. The best known inapproximation result for minimum Vertex Cover (see Theorem 10.7) is due to [65], but we doubt it is tight (see, e.g., [133]). Reductions among approximation problems were defined and presented in [166]; see Exercise 10.3, which presents a major technique introduced in [166]. For general texts on approximation algorithms and problems (as discussed in Section 10.1.1), the interested reader is referred to the surveys collected in [117]. A compendium of NP optimization problems is available at [61].

Property testing. The study of property testing was initiated by Rubinfeld and Sudan [182] and re-initiated by Goldreich, Goldwasser, and Ron [93]. While the focus of [182] was on algebraic properties such as low-degree polynomials, the focus of [93] was on graph properties (and Theorem 10.12 is taken from [93]). The model of bounded-degree graphs was introduced in [99] and Theorem 10.13 combines results from [99, 100, 39]. For surveys of the area, the interested reader is referred to [73, 181].

Average-case complexity

The theory of average-case complexity was initiated by Levin [144], who in particular proved Theorem 10.17. In light of the laconic nature of the original text [144], we refer the interested reader to a survey [85], which provides a more detailed exposition of the definitions suggested by Levin as well as a discussion of the considerations underlying these suggestions. (This survey [85] provides also a brief account of further developments.)

As noted in §10.2.1.1, the current text uses a variant of the original definitions. In particular, our definition of "typical feasibility" differs from the original definition of "average feasibility" in totally discarding exceptional instances and in even allowing the algorithm to fail on them (and not merely run for an excessive amount of time). The alternative definition was suggested by several researchers and appears as a special case of the general treatment provided in [41].

Section 10.2.2 is based on [28, 120]. Specifically, Theorem 10.21 (or rather the reduction of search to decision) is due to [28] and so is the introduction of the class samp$\mathcal{NP}$. A version of Theorem 10.24 was proven in [120], and our proof follows their ideas, which in turn are closely related to the ideas underlying the proof of Theorem 8.11 (proved in [113]).

Recall that we know of the existence of problems in dist$\mathcal{NP}$ that are hard provided samp$\mathcal{NP}$ contains hard problems. However, these problems refer to somewhat generic decision problems such as $S_{\mathtt{u}}$. The presetation of dist$\mathcal{NP}$-complete problems that combine a more natural decision problem (like SAT or Clique) with a simple probability ensemble is an open problem.

Exercises

Exercise 10.1 (general TSP) For any function g, prove that the following approximation problem is NP-Hard. Given a general TSP instance I, the task is finding a tour of length that is at most a factor $g(I)$ of the minimum. In the case that all distances are required to be positive, show that the result holds with $g(I) = \exp(\text{poly}(|I|))$.

Guideline: By reduction from Hamiltonian path. Specifically, reduce the instance $G = ([n], E)$ to an n-by-n distance matrix $D = (d_{i,j})_{i,j\in[n]}$ such that $d_{i,j} = 1$ if $\{i,j\} \in E$ and $d_{i,j} = 0$ otherwise. In case that all distances are required to be positive, we may set $d_{i,j} = \exp(\text{poly}(n))$ if $\{i,j\} \in E$ and $d_{i,j} = 1$.

Exercise 10.2 (TSP with triangle inequalities) Provide a polynomial-time 2-factor approximation for the special case of TSP in which the distances satisfy the triangle inequality.

Guideline: First note that the length of any tour is lower-bounded by the weight of a minimum spanning tree in the corresponding weighted graph. Next note that such a tree yields a tour (of length twice the weight of this tree) that may visit some points several times. The triangle inequality guarantees that the tour does not become longer by "shortcuts" that eliminate multiple visits at the same point.

Exercise 10.3 (enforcing multi-way equalities via expanders) The aim of this exercise is presenting a major technique of Papadimitriou and Yannakakis [166], which is useful for designing reductions among approximation problems. Recalling that $\mathtt{gapSAT}^3_{0.1}$ is NP-hard, our goal is proving NP-hard of the following gap problem, denoted $\mathtt{gapSAT}^{3,c}_{\varepsilon}$, which is a special case of $\mathtt{gapSAT}^3_{\varepsilon}$. Specifically, the instances are restricted to 3CNF formulae with each variable appearing in at most c clauses, where c (as ε) is a fixed constant. Note that the standard reduction of 3SAT to the corresponding special case (see proof of Proposition 2.22) does not preserve an approximation gap.[18] The idea is enforcing equality of the values assigned

[18] Recall that in this reduction each occurrence of each Boolean variable is replaced by a new copy of this variable, and clauses are added for enforcing the assignment of the same value to all these copies. Specifically, the m occurrence of variable z are replaced by the variables $z^{(1)},...,z^{(m)}$, while adding the clauses $z^{(i)} \vee \neg z^{(i+1)}$ and $z^{(i+1)} \vee \neg z^{(i)}$ (for $i = 1,...,m-1$). The problem is that almost all clauses of the reduced formula may be satisfied by an assignment in which half of the copies of each variable are assigned one value and the rest are assigned an opposite value. That is, an assignment in which $z^{(1)} = \cdots = z^{(i)} \neq z^{(i+1)} = \cdots = z^{(m)}$ violates only one of the auxiliary clauses introduced for enforcing equality among the copies of z. Using an alternative reduction that adds the clauses $z^{(i)} \vee \neg z^{(j)}$ for every $i,j \in [m]$ will not do either, because the number of added clauses may be quadratic in the number of original clauses.

to the auxiliary variables (i.e., the copies of each original variable) by introducing equality constraints only for pairs of variables that correspond to edges of an expander graph (see Appendix E.2). For example, we enforce equality among the values of $z^{(1)},...,z^{(m)}$ by adding the clauses $z^{(i)} \vee \neg z^{(j)}$ for every $\{i,j\} \in E$, where E is the set of edges of a m-vertex expander graph. Prove that, for some constants c and $\varepsilon > 0$, the corresponding mapping reduces $\texttt{gapSAT}^3_{0.1}$ to $\texttt{gapSAT}^{3,c}_{\varepsilon}$.

Guideline: Using a d-regular expander, we map 3CNF to instances in which each variable appears in at most $2d+1$ clauses. Note that if the original formula is satisfiable then so is the reduced one. On the other hand, consider an arbitrary assignment τ' to the reduced formula ϕ' (i.e., the formula obtained by mapping ϕ). For each original variable z, if τ' assigns the same value to almost all copies of z then we consider the corresponding assignment in ϕ. Otherwise, τ' does not satisfy a constant fraction of the clauses containing a copy of z.

Exercise 10.4 (an equivalent definition of tpc$\mathcal{P}$**)** Prove that $(S,X) \in$ tpc$\mathcal{P}$ if and only if there exists a polynomial-time algorithm A such that the probability that $A(X_n)$ errs (in determining membership in S) is a negligible function in n.

Exercise 10.5 (tpc$\mathcal{P}$ **versus** $\mathcal{P}$ **– Part 1)** Prove that tpc$\mathcal{P}$ contains a problem (S,X) such that S is not even recursive. Furthermore, use $X = U$.

Guideline: Let $S = \{0^{|x|}x : x \in S'\}$, where S' is an arbitrary (non-recursive) set.

Exercise 10.6 (tpc$\mathcal{P}$ **versus** $\mathcal{P}$ **– Part 2)** Prove that there exists a distributional problem (S,X) such that $S \notin \mathcal{P}$ and yet there exists an algorithm solving S (correctly on all inputs) in time that is typically polynomial with respect to X. Furthermore, use $X = U$.

Guideline: For any time-constructible function $t : \mathbb{N} \to \mathbb{N}$ that is super-polynomial and sub-exponential, use $S = \{0^{|x|}x : x \in S'\}$ for any $S' \in \mathrm{DTIME}(t) \setminus \mathcal{P}$.

Exercise 10.7 (reductions preserve typical polynomial-time solveability) Prove that if the distributional problem (S,X) is reducible to the distributional problem (S',X') and $(S',X') \in$ tpc$\mathcal{P}$, then (S,X) is in tpc$\mathcal{P}$.

Guideline: Let B' denote the set of exceptional instances for the distributional problem (S',X') (i.e., the set of instances on which the solver in the hypothesis errs or exceeds the typical running-time). Prove that $\Pr[Q(X_n) \cap B' \neq \emptyset]$ is a negligible function (in n), using both $\Pr[y \in Q(X_n)] \leq p(|y|) \cdot \Pr[X'_{|y|} = y]$ and $|x| \leq p(|y|)$ for every $y \in Q(x)$. Specifically, use the latter condition for inferring that $\sum_{y \in B'} \Pr[y \in Q(X_n)]$ equals $\sum_{y \in \{y' \in B' : p(|y'|) \geq n\}} \Pr[y \in Q(X_n)]$, which guarantees that a negligible function in $|y|$ for any $y \in Q(X_n)$ is negligible in n.

Exercise 10.8 (reductions preserve error-less solveability) In continuation to Exercise 10.7, prove that reductions preserve error-less solveability (i.e., solveability by algorithms that never err and typically run in polynomial-time).

Exercise 10.9 (transitivity of reductions) Prove that reductions among distributional problems (as in Definition 10.16) are transitive.

Guideline: The point is establishing the domination property of the composed reduction. The hypothesis that reductions do not make too short queries is instrumental here.

Exercise 10.10 For any $S \in \mathcal{NP}$ present a simple probability ensemble X such that the generic reduction used in the proof of Theorem 2.18 violates the domination condition regarding a possible reduction of (S, X) to $(S_{\mathtt{u}}, U')$.

Guideline: Consider $X = \{X_n\}_{n\in\mathbb{N}}$ such that X_n is uniform over $\{0^{n/2}x' : x' \in \{0,1\}^{n/2}\}$.

Exercise 10.11 (variants of the Coding Lemma) Prove the following two variants of the Coding Lemma (which is stated in the proof of Theorem 10.17).

1. A variant that refers to any monotonically non-decreasing function $\mu : \{0,1\}^* \to [0,1]$ that is efficiently computable, where here we refer to the lexicographic order over $\{0,1\}^*$.

2. As in Part 1, except that in this variant the function μ is strictly increasing and the compression condition requires that $|C_\mu(x)| \leq \log_2(1/\mu'(x))$ rather than $|C_\mu(x)| \leq 1 + \min\{|x|, \log_2(1/\mu'(x))\}$, where $\mu'(x) \stackrel{\text{def}}{=} \mu(x) - \mu(x-1)$.

Guideline: In both cases, the proof is less cumbersome than the one presented in the main text.

Exercise 10.12 (randomized reductions) Following the outline in §10.2.1.3, provide a definition of randomized reductions among distributional problems.

1. Prove that randomized reductions are transitive (cf. Exercise 10.9).

2. In analogy to Exercise 10.7, prove that randomized reductions preserve feasible solveability (i.e., typical solveability in probabilistic polynomial-time). That is, if the distributional problem (S, X) is randomly reducible to the distributional problem (S', X') and $(S', X') \in \text{tpc}\mathcal{BPP}$, then (S, X) is in $\text{tpc}\mathcal{BPP}$.

3. In analogy to Exercise 10.8, prove that randomized reductions preserve solveability by probabilistic algorithms that err with probability at most $1/3$ on each input and typically run in polynomial-time.

4. Show that the error probability of such reductions can be reduced (while preserving the domination condition).

Extend the foregoing to reductions that involve distributional *search* problems.

Exercise 10.13 (simple vs sampleable ensembles – Part 1) Prove that any simple probability ensemble is polynomial-time sampleable.

Guideline: Let μ be the accumulating distribution function of $X = \{X_n\}_{n\in\mathbb{N}}$ (i.e., $\mu(x) = \Pr[X_{|x|} \leq x]$), and let p be a polynomial such that (without loss of generality) $|\mu(x)| = p(|x|)$ for every x. Consider the algorithm that, on input 1^n, uniformly selects $i \in [2^{p(n)}]$ and outputs x if and only if x is the lexicographically first string such that $i \leq \mu(x) \cdot 2^{p(n)}$. Note that this x can be found by binary search, using the fact that μ is polynomial-time computable.

Exercise 10.14 (simple vs sampleable ensembles – Part 2) Assuming that $\#\mathcal{P}$ contains functions that are not computable in polynomial-time, prove that there exists polynomial-time sampleable ensembles that are not simple.

Guideline: Consider any $R \in \mathcal{PC}$ and suppose that p is a polynomial such that $(x, y) \in R$ implies $|y| = p(|x|)$. Then consider the sampling algorithm A that, on input 1^n, uniformly selects $(x, y) \in \{0,1\}^{n-1} \times \{0,1\}^{p(n-1)}$ and outputs $x1$ if $(x, y) \in R$ and $x0$ otherwise. Note that $\#R(x) = 2^{p(|x|-1)} \cdot \Pr[A(1^{|x|-1}) = x1]$.

Exercise 10.15 (distributional versions of NPC problems – Part 1 [28]) Prove that for any NP-complete problem S there exists a polynomial-time sampleable ensemble X such that any problem in dist$\mathcal{NP}$ is reducible to (S, X). We actually assume that the many-to-one reductions establishing the NP-completeness of S do not shrink the length of the input.

Guideline: Prove that the guaranteed reduction (of $S_{\mathtt{u}}$ to S) also reduces $(S_{\mathtt{u}}, U')$ to (S, X), for some sampleable probability ensemble X. Specifically, note that U' is sampleable (by Exercise 10.13) and prove that the standard reduction of $S_{\mathtt{u}}$ to S, when applied to a sampleable probability ensemble, induces a sampleable distribution on the instances of S. Consider first the case that the standard reduction is length preserving, and next extend the treatment to the general case.

Exercise 10.16 (distributional versions of NPC problems – Part 2 [28]) Prove Theorem 10.23 (i.e., for any NP-complete problem S there exists a polynomial-time sampleable ensemble X such that any problem in samp$\mathcal{NP}$ is reducible to (S, X)). As in Exercise 10.15, we actually assume that the many-to-one reductions establishing the NP-completeness of S do not shrink the length of the input.

Guideline: We establish the result for $S_{\mathtt{u}}$, and the rest follows as in Exercise 10.15. Thus, we focus on showing that, for a fixed sampleable X, we can reduce any $(S', X') \in$ samp$\mathcal{NP}$ to $(S_{\mathtt{u}}, X)$. Loosely speaking, X will be an adequate convex combination of all sampleable distributions (and thus X will not equal U' or U). Specifically, $X = \{X_n\}_{n\in\mathbb{N}}$ is defined such that X_n selects $i \in [n]$ with probability $\approx 1/i^2$, emulate the execution of the i^{th} algorithm (in lexicographic order) on input 1^n for n^3 steps,[19] and outputs whatever the latter has output (or 0^n in case the said algorithm has not halted within n^3 steps). Prove that, for any $(S'', X'') \in$ samp$\mathcal{NP}$ such that X'' is sampleable in cubic time, the standard reduction of S'' to $S_{\mathtt{u}}$ reduces (S'', X'') to $(S_{\mathtt{u}}, X)$ (as per Definition 10.15; i.e.,

[19]Needless to say, the choice to consider n algorithms in the definition of X_n is quite arbitrary. Any other unbounded function of n that is at most a polynomial (and is computable in polynomial-time) will do. Likewise, the choice to emulate each algorithm for a cubic number of steps (rather some other polynomial number of steps) is quite arbitrary.

in particular, it satisfies the domination condition).[20] Finally, using adequate padding, reduce any $(S', X') \in \mathrm{samp}\mathcal{NP}$ to some $(S'', X'') \in \mathrm{samp}\mathcal{NP}$ such that X'' is sampleable in cubic time.

Exercise 10.17 (search vs decision in the context of sampleable ensembles) Prove that every problem in $\mathrm{samp}\mathcal{NP}$ is reducible to some problem in $\mathrm{samp}\mathcal{PC}$, and every problem in $\mathrm{samp}\mathcal{PC}$ is *randomly* reducible to some problem in $\mathrm{samp}\mathcal{NP}$.

Guideline: See proof of Theorem 10.21.

[20]Note that applying this reduction, denoted f, to X'' yields an ensembles that is also sampleable in cubic time. This uses the fact that the standard reduction runs in time that is almost linear in its output, which in turn is longer than the input.

Epilogue

Farewell, Hans – whether you live or end where you are! Your chances are not good. The wicked dance in which you are caught up will last a few more sinful years, and we would not wager much that you will come out whole. To be honest, we are not really bothered about leaving the question open. Adventures in the flesh and spirit, which enhanced and heightened your ordinariness, allowed you to survive in the spirit what you probably will not survive in the flesh. There were majestic moments when you saw the intimation of a dream of love rising up out of death and the carnal body. Will love someday rise up out of this world-wide festival of death, this ugly rutting fever that inflames the rainy evening sky all round?

Thomas Mann, The Magic Mountain, The Thunderbolt.

We hope that this work succeeds in conveying the fascinating flavour of the concepts, results and open problems that dominate the field of computational complexity. We believe that the new century will witness even more exciting developments in this field, and urge the reader to try to contribute to them. But before bidding goodbye, we wish to express a few more thoughts.

As noted in Section 1.1.1, so far complexity theory has been far more successful in relating fundamental computational phenumena than in providing definite answers regarding fundamental questions. Consider, for example, the theory of NP-completeness versus the P-versus-NP Question, or the theory of pseudorandomness versus establishing the existence of one-way function (even under $\mathcal{P} \neq \mathcal{NP}$). The failure to resolve questions of the "absolute" type is the source of common frustration and one often wonders about the reasons for this failure.

Our feeling is that many of these failures are really due to the difficulty of the questions asked, and that it is the appealing and natural flavor of these questions that makes us underestimate their hardness. Indeed, the underlying assumption is that if a question is appealing and natural then answering it should not be hard. We doubt this assumption. Our feeling is that the more intuitive a question is, the harder it may be to answer. Our view is that intuitive questions arise from an encounter with the raw and chaotic reality of life, rather than from an artificial construct which is typically endowed with a rich internal structure. Indeed, natural complexity classes and natural questions regarding computation arise from looking at the reality of computation from the outside and lack any internal structure. That is, complexity classes are defined in terms of the "external behavior" of potential algorithms rather than in terms of the "internal structure" (of the problem).

Appendix A

Glossary of Complexity Classes

Summary: This glossary includes self-contained definitions of most complexity classes mentioned in the book. Needless to say, the glossary offers a very minimal discussion of these classes and the reader is referred to the main text for further discussion. The items are organized by topics rather than by alphabetic order. Specifically, the glossary is partitioned into two parts, dealing seperately with complexity classes that are defined in terms of algorithms and their resources (i.e., time and space complexity of Turing machines) and complexity classes defined in terms of non-uniform circuit (and referring to their size and depth). The algorithmic classes include time-complexity based classes (such as $\mathcal{P}$, $\mathcal{NP}$, co$\mathcal{NP}$, $\mathcal{BPP}$, $\mathcal{RP}$, co$\mathcal{RP}$, $\mathcal{PH}$, $\mathcal{E}$, $\mathcal{EXP}$ and $\mathcal{NEXP}$) and the space complexity classes $\mathcal{L}$, $\mathcal{NL}$, $\mathcal{RL}$ and $\mathcal{PSPACE}$. The non-uniform classes include the circuit classes $\mathcal{P}$/poly as well as $\mathcal{NC}^k$ and $\mathcal{AC}^k$.

The definition (and basic results) regarding many other complexity classes is available at the constantly evolving *Complexity Zoo* [1].

A.1 Introduction

Complexity classes are sets of computational problems, where each class contains problems that can be solved with specific computational resources. To define a complexity class one specifies a model of computation, a complexity measure (like time or space), which is always measured as a function of the input length, and a bound on the complexity (of problems in the class).

We follow the tradition of focusing on decision problems, but refer to these problems using the terminology of promise problems (see Section 2.4.1). That is, we will refer to the problem of distinguishing inputs in $\Pi_{\rm yes}$ from inputs in $\Pi_{\rm no}$, and denote the corresponding decision problem by $\Pi = (\Pi_{\text{yes}}, \Pi_{\text{no}})$. Standard decision problems are viewed as a special case in which $\Pi_{\text{yes}} \cap \Pi_{\text{no}} = \{0,1\}^*$, and the standard formulation of complexity classes is obtained by postulating that this is the case. We refer to this case as the case of a trivial promise.

The prevailing model of computation is that of Turing machines, a model which captures the notion of (uniform) algorithms (see Section 1.2.3). Another important model is the one of non-uniform circuits (see Section 1.2.4). The term uniformity refers to whether the algorithm is the same one for every input length or whether a different "algorithm" (or rather a "circuit") is considered for each input length.

We focus on natural complexity classes, obtained by considering natural complexity measures and bounds. Typically, these classes contain natural computational problems (which are defined in Appendix G). Furthermore, almost all of these classes can be "characterized" by natural problems, which capture every problem in the class. Such problems are called complete for the class, which means that they are in the class and every problem in the class can be "easily" reduced to them, where "easily" means that the reduction takes less resources than what each of the problems seems to require individually. Efficient algorithm for a complete problem implies an algorithm of similar efficiency for *all* problems in the class.

Organization: The glossary is organized by topics (rather than by alphabetic order of the various items). Specifically, we partition the glossary to classes defined in terms of algorithmic resources (i.e., time and space complexity of Turing machines) and classes defined in terms of circuit (size and depth). The former include the time-complexity classes $\mathcal{P}$, $\mathcal{NP}$, co$\mathcal{NP}$, $\mathcal{BPP}$, $\mathcal{RP}$, co$\mathcal{RP}$, $\mathcal{PH}$, $\mathcal{E}$, $\mathcal{EXP}$ and $\mathcal{NEXP}$ as well as the space-complexity classes $\mathcal{L}$, $\mathcal{NL}$, $\mathcal{RL}$ and $\mathcal{PSPACE}$, which are all reviewed in Section A.2. The latter include the circuit classes $\mathcal{P}$/poly as well as $\mathcal{NC}^k$ and $\mathcal{AC}^k$, which are reviewed in Section A.3.

A.2 Algorithm-based classes

The two main complexity measures considered in the context of (uniform) algorithms are the number of steps taken by the algorithm (i.e., its time complexity) and the amount of "memory" or "work-space" consumed by the computation (i.e., its space complexity). We review the time complexity based classes $\mathcal{P}$, $\mathcal{NP}$, co$\mathcal{NP}$, $\mathcal{BPP}$, $\mathcal{RP}$, co$\mathcal{RP}$, $\mathcal{ZPP}$, $\mathcal{PH}$, $\mathcal{E}$, $\mathcal{EXP}$ and $\mathcal{NEXP}$ as well as the space complexity classes $\mathcal{L}$, $\mathcal{NL}$, $\mathcal{RL}$ and $\mathcal{PSPACE}$.

By prepending the name of a complexity class (of decision problems) with the prefix "co" we mean the class of complement problems; that is, the problem $\Pi = (\Pi_{\text{yes}}, \Pi_{\text{no}})$ is in co$\mathcal{C}$ if and only if $(\Pi_{\text{no}}, \Pi_{\text{yes}})$ is in $\mathcal{C}$. Specifically, deciding membership in the set S is in the class co$\mathcal{C}$ if and only if deciding membership in the set $\{0,1\}^* \setminus S$ is in the class $\mathcal{C}$. Thus, the definition of co$\mathcal{NP}$ and co$\mathcal{RP}$ can be easily derived from the definitions of $\mathcal{NP}$ and $\mathcal{RP}$, respectively. Complexity classes defined in terms of symmetric acceptance criteria (e.g., deterministic and two-sided error randomized classes) are trivially closed under complementation (e.g., co$\mathcal{P} = \mathcal{P}$ and co$\mathcal{BPP} = \mathcal{BPP}$) and so we don't even present their "co"-classes. In other cases (most notablly $\mathcal{NL}$ and $\mathcal{SZK}$), the closure property is highly non-trivial and we comment about it.

A.2.1 Time complexity classes

We start with classes that are closely related to polynomial-time computations (i.e., $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{BPP}$, $\mathcal{RP}$ and $\mathcal{ZPP}$), and latter consider the classes $\mathcal{PH}$, $\mathcal{E}$, $\mathcal{EXP}$ and $\mathcal{NEXP}$.

A.2.1.1 Classes closely related to polynomial time

The most prominent complexity classes are $\mathcal{P}$ and $\mathcal{NP}$, which are extensively discussed in Section 2.1. We also consider classes related to randomized polynomial-time, which are discussed in Section 6.1.

P and NP. The class $\mathcal{P}$ consists of all decision problem that can be solved in (deterministic) polynomial-time. A decision problem $\Pi = (\Pi_{\text{yes}}, \Pi_{\text{no}})$ is in $\mathcal{NP}$ if there exists a polynomial p and a (deterministic) polynomial-time algorithm V such that the following two conditions hold

1. For every $x \in \Pi_{\text{yes}}$ there exists $y \in \{0,1\}^{p(|x|)}$ such that $V(x,y) = 1$.
2. For every $x \in \Pi_{\text{no}}$ and every $y \in \{0,1\}^*$ it holds that $V(x,y) = 0$.

A string y satisfying Condition 1 is called an NP-witness (for x).

Reductions and NP-completeness (NPC). A problem is $\mathcal{NP}$-complete if it is in $\mathcal{NP}$ and every problem in $\mathcal{NP}$ is polynomial-time reducible to it, where polynomial-time reducibility is defined and discussed in Section 2.2. Loosely speaking, a polynomial-time reduction of problem Π to problem Π' is a polynomial-time algorithm that solves Π by making queries to a subroutine that solves problem Π', where the running-time of the subroutine is not counted in the algorithm's time complexity. Typically, NP-completeness is defined while restricting the reduction to make a single query and output its answer. Such a reduction, called a Karp-reduction, is represented by a polynomial-time computible mapping that maps yes-instances of Π to yes-instances of Π' (and no-instances of Π to no-instances of Π').

Probabilistic polynomial-time (BPP, RP and ZPP). A decision problem $\Pi = (\Pi_{\text{yes}}, \Pi_{\text{no}})$ is in $\mathcal{BPP}$ if there exists a probabilistic polynomial-time algorithm A such that the following two conditions hold

1. For every $x \in \Pi_{\text{yes}}$ it holds that $\Pr[A(x)=1] \geq 2/3$.
2. For every $x \in \Pi_{\text{no}}$ it holds that $\Pr[A(x)=0] \geq 2/3$.

That is, the algorithm has two-sided error probability (of $1/3$), which can be further reduced by repetitions. We stress that due to the two-sided error probability of $\mathcal{BPP}$, it is not known whether or not $\mathcal{BPP}$ is contained in $\mathcal{NP}$. In contrast to $\mathcal{BPP}$, there are one-sided and zero-error classes, denoted $\mathcal{RP}$ and $\mathcal{ZPP}$, respectively. A problem $\Pi = (\Pi_{\text{yes}}, \Pi_{\text{no}})$ is in $\mathcal{RP}$ if there exists a probabilistic polynomial-time algorithm A such that the following two conditions hold

1. For every $x \in \Pi_{\text{yes}}$ it holds that $\Pr[A(x)=1] \geq 1/2$.
2. For every $x \in \Pi_{\text{no}}$ it holds that $\Pr[A(x)=0] = 1$.

Again, the error probability can be reduced by repetitions, and thus $\mathcal{RP} \subseteq \mathcal{BPP} \cap \mathcal{NP}$. A problem $\Pi = (\Pi_{\text{yes}}, \Pi_{\text{no}})$ is in $\mathcal{ZPP}$ if there exists a probabilistic polynomial-time algorithm A, which may output a special ("don't know") symbol $\perp$, such that the following two conditions hold

1. For every $x \in \Pi_{\text{yes}}$ it holds that $\Pr[A(x) \in \{1, \perp\}] = 1$ and $\Pr[A(x)=1] \geq 1/2$.
2. For every $x \in \Pi_{\text{no}}$ it holds that $\Pr[A(x) \in \{0, \perp\}] = 1$ and $\Pr[A(x)=0] \geq 1/2$.

Note that $\mathcal{P} \subseteq \mathcal{ZPP} = \mathcal{RP} \cap \text{co}\mathcal{RP}$. When defined in terms of promise problems, these classes have complete problems (w.r.t Karp-reductions), but the same is not known when considering only standard decision problems (with trivial promise).

The counting class $\#\mathcal{P}$. Functions in $\#\mathcal{P}$ count the number of solutions to an NP-type search problem (or, equivalently, the number of NP-witnesses for a yes-instance of a decision problem in $\mathcal{NP}$). Formally, a function f is in $\#\mathcal{P}$ if there exists a polynomial p and a (deterministic) polynomial-time algorithm V such that $f(x) = |\{y \in \{0,1\}^{p(|x|)} : V(x,y)=1\}|$. Indeed, p and V are as in the definition of $\mathcal{NP}$, and it follows that deciding membership in the set $\{x : f(x) \geq 1\}$ is in $\mathcal{NP}$. Clearly, $\#\mathcal{P}$ problems are solvable in polynomial space. Surprisingly, the permanent of positive integer matrices is $\#\mathcal{P}$-complete (i.e., it is in $\#\mathcal{P}$ and any function in $\#\mathcal{P}$ is polynomial-time reducible to it).

A.2.1.2 Other time complexity classes

The classes $\mathcal{E}$ and $\mathcal{EXP}$ corresponding to problems that can be solved (by a deterministic algorithm) in time $2^{O(n)}$ and $2^{\text{poly}(n)}$, respectively, for n-bit long inputs. Clearly, $\mathcal{NP} \subseteq \mathcal{EXP}$.

In general, one may define a complexity class for every time bound and every type of machine (i.e., deterministic, probabilistic and non-deterministic), but polynomial and exponential bounds seem most natural and very robust. Another robust type of time bounds that is sometimes used is quasi-polynomial time (e.g., $\widetilde{\mathcal{P}}$ denotes the class of problems solvable by deterministic machines of time complexity $\exp(\text{poly}(\log n))$). Another class that occurs is $\mathcal{NEXP}$ the class of problems that can be solved by a non-deterministic machine in $2^{\text{poly}(n)}$ steps.[1]

[1]Alternatively, analogously to the definition of $\mathcal{NP}$, a problem $\Pi = (\Pi_{\text{yes}}, \Pi_{\text{no}})$ is in $\mathcal{NEXP}$

The Polynomial-time hierarchy, $\mathcal{PH}$. For any natural number k, the k^{th} level of the polynomial-time hierarchy consists of problems $\Pi = (\Pi_{\text{yes}}, \Pi_{\text{no}})$ such that there a polynomial p and a polynomial-time algorithm V that satisfies the following two conditions:

1. For every $x \in \Pi_{\text{yes}}$ there exists $y_1 \in \{0,1\}^{p(|x|)}$ such that for every $y_2 \in \{0,1\}^{p(|x|)}$ there exists $y_3 \in \{0,1\}^{p(|x|)}$ such that for every $y_4 \in \{0,1\}^{p(|x|)}$ ... (going for k quantifier alternations) ... it holds that $V(x, y_1, y_2, y_3, y_4, ..., y_k) = 1$.

2. For every $x \in \Pi_{\text{no}}$, the above condition does not hold. That is, for every $y_1 \in \{0,1\}^{p(|x|)}$ there exists $y_2 \in \{0,1\}^{p(|x|)}$ such that for every $y_3 \in \{0,1\}^{p(|x|)}$ there exists $y_4 \in \{0,1\}^{p(|x|)}$ ... it holds that $V(x, y_1, y_2, y_3, y_4, ..., y_k) = 0$.

Such a problem Π is said to be in Σ_k (and $\Pi_k \stackrel{\text{def}}{=} \text{co}\Sigma_k$). Indeed, $\mathcal{NP} = \Sigma_1$ corresponds to the special case where $k = 1$. Interestingly, $\mathcal{PH}$ is polynomial-time reducible to $\#\mathcal{P}$.

A.2.2 Space complexity

When defining space-complexity classes, one counts *only* the space consumed by the actual computation, and *not* the space occupied by the input and output. This is formalized by postulating that the input is read from a read-only device (resp., the output is written on a write-only device). Four important classes of decision problems are:

- The class $\mathcal{L}$ consists of problems solvable in logarithmic space. That is, a problem Π is in $\mathcal{L}$ if there exists a standard (i.e., deterministic) algorithm of logarithmic space-complexity for solving Π. This class contains some simple computational problems (e.g., matrix multiplication), and arguably captures the most space-efficient computations.

- Classes of problems solvable by randomized algorithms of logarithmic space-complexity include $\mathcal{RL}$ and $\mathcal{BPL}$, which are defined analogously to $\mathcal{RP}$ and $\mathcal{BPP}$. That is, $\mathcal{RL}$ corresponds to algorithms with one-sided error probability, whereas $\mathcal{BPL}$ allows two-sided error.

- The class $\mathcal{NL}$ is the non-deterministic analogue of $\mathcal{L}$, and is traditionally defined in terms of non-deterministic machines of logarithmic space-complexity.[2] The class $\mathcal{NL}$ contains the problem of deciding whether there exists a directed

---

if there exists a polynomial p and a polynomial-time algorithm V such that the two conditions hold

1. For every $x \in \Pi_{\text{yes}}$ there exists $y \in \{0,1\}^{2^{p(|x|)}}$ such that $V(x,y) = 1$.
2. For every $x \in \Pi_{\text{no}}$ and every $y \in \{0,1\}^*$ it holds that $V(x,y) = 0$.

[2] See further discussion of this definition in Section 5.3. In particular, note that we refrained here from presenting a definition analogous to the definition of $\mathcal{NP}$.

path between two given vertexes in a given directed graph. In fact, the latter problem is complete for the class (under logarithmic-space reductions). Interestingly, co$\mathcal{NL}$ equals $\mathcal{NL}$.

- The class $\mathcal{PSPACE}$ consists of problems solvable in polynomial space. This class contains very difficult problems, including the computation of winning strategies for any "efficient 2-party games" (see Section 5.4).

Clearly, $\mathcal{L} \subseteq \mathcal{RL} \subseteq \mathcal{NL} \subseteq \mathcal{P}$ and $\mathcal{NP} \subseteq \mathcal{PSPACE} \subseteq \mathcal{EXP}$.

A.3 Circuit-based classes

We refer the reader to Section 1.2.4 for a definition of Boolean circuits as computing devices. The two main complexity measures considered in the context of (non-uniform) circuits are the number of gates (or wires) in the circuit (i.e., the circuit's size) and the length of the longest directed path from an input to an output (i.e., the circuit's depth).

Throughout this section, when we talk of circuits, we actually refer to families of circuits conatining a circuit for each instance length, where the n-bit long instances of the computational problem are handled by the n^{th} circuit in the family. Similarly, the size and depth of the circuit actually refers to the family of circuits and to the dependence (on n) of the size and depth of the n^{th} circuit in the family.

General polynomial-size circuits (P/poly). The main motivation for the introduction of complexity classes based on (non-uniform) circuits is the development of lower-bounds. For example, the class of problems solvable by polynomial-size circuits, denoted $\mathcal{P}$/poly, is a super-set of $\mathcal{P}$ (because it clearly contains $\mathcal{P}$ as well as deciding membership in any subset of $\{1\}^*$, whereas there exists such sets that represents decision problems that are not solvable (i.e., by any uniform algorithm)). Thus, showing that $\mathcal{NP}$ is not contained in $\mathcal{P}$/poly would imply $\mathcal{P} \neq \mathcal{NP}$. For further discussion see Section B.2.

The subclasses AC0 and TC0. The class $\mathcal{AC}^0$, discussed in Section B.2, consists of problems solvable by constant-depth polynomial-size circuits of *unbounded fan-in*. The analogue class that allows also (unbounded fan-in) majority-gates (or, equivalently, threshold-gates) is denoted $\mathcal{TC}^0$.

The subclasses AC and NC. Turning back to the standard basis (of $\neg$, $\vee$ and $\wedge$), for any non-negative integer k, we denote by $\mathcal{NC}^k$ (resp., $\mathcal{AC}^k$) the class of problems solvable by polynomial-size circuits of *bounded fan-in* (resp., unbounded fan-in) having depth $O(\log^k n)$, where n is the input length. Clearly, $\mathcal{NC}^k \subseteq \mathcal{AC}^k \subseteq \mathcal{NC}^{k+1}$. A commonly referred class is $\mathcal{NC} \stackrel{\text{def}}{=} \cup_{k \in \mathbb{N}} \mathcal{NC}^k$.

We mention that the class $\mathcal{NC}^2 \supseteq \mathcal{NL}$ is the habitat of most natural computational problems of Linear Algebra: solving a linear system of equations as well as computing the rank, inverse and determinant of a matrix. The class $\mathcal{NC}^1$

contains all symmetric functions, regular languages as well as word problems for finite groups and monoids. The class $\mathcal{AC}^0$ contains all properties of finite objects expressible by first-order logic.

Uniformity. The above classes make no reference to the complexity of constructing the adequate circuits, and it is plausible that there is no effective way of constructing these circuits (e.g., as in case of circuits that trivially solve undecidable problem regarding unary instances). A minimal notion of constructability of such (polynomial-size) circuits is the existance of a polynomial time algorithm that given 1^n produces the n^{th} relevant circuit (i.e., the circuit that solves the problem on instances of length n). Such a notion of constructability means that the family of circuits is "uniform" in some sense (rather than consisting of circuits that have no relation between one another). Stronger notions of uniformity (e.g., log-space constructability) are more adequate for subclasses such as AC and NC. We mention that log-space uniform NC circuits correspond to parallel algorithms that use polynomially many processors and run in polylogarithmic time.

Appendix B

On the Quest for Lower Bounds

Alas, Philosophy, Medicine, Law, and unfortunately also Theology, have I studied in detail, and still remained a fool, not a bit wiser than before. Magister and even Doctor am I called, and for a decade am I sick and tired of pulling my pupils by the nose and understanding that we can know nothing.[1]

J.W. Goethe, Faust, Lines 354–364

Summary: In this appendix we survey some attempts at proving lower bounds on the complexity of natural computational problems. In the first part, devoted to Circuit Complexity, we describe lower bounds for the *size* of (restricted) circuits that solve natural computational problems. This can be viewed as a program whose long-term goal is proving that $\mathcal{P} \neq \mathcal{NP}$. In the second part, devoated to Proof Complexity, we describe lower bounds on the length of (restricted) propositional proofs of natural tautologies. This can be viewed as a program whose long-term goal is proving that $\mathcal{NP} \neq \text{co}\mathcal{NP}$.

The current activity in these areas is aimed towards developing proof techniques that may be applied to the resolution of the "big problems" (such as P versus NP), but the current achievements (though very impressive) seem far from reaching this goal. Current crown-jewel achievements in these areas take the form of tight (or strong) lower bounds on the complexity of computing (resp., proving) "relatively simple" functions (resp., claims) in *restricted* models of computation (resp., proof systems).

[1]This quote reflects a common sentiment, not shared by the author of the current book.

B.1 Preliminaries

Circuit complexity refers to a non-uniform model of computation; specifically the model of Boolean circuits, focusing on the size of such circuits, while ignoring the complexity of constructing adequate circuits. Similarly, proof complexity refers to proofs of tautologies, focusing on the length of such proofs, while ignoring the complexity of generating such proofs. Both circuits and proofs are finite objects that are defined on top of the notion of a *directed acyclic graph* (dag), which we review next.

A dag $G(V,E)$ consists of a finite set of vertices V, and a set of ordered pairs called directed edges $E \subseteq V \times V$, in which there are no directed cycles. The vertices with no incoming edges are called the inputs of the dag G, and the vertices with no outgoing edges are called the outputs. We will restrict ourselves to dags in which the number of *incoming* edges to every vertex is at most 2. If the number of *outgoing* edges from every node is at most 1, the dag is called a tree. Finally, we assume that every vertex can be reached from some input via a directed path. The size of a dag will be its number of edges.

To make a dag into a computational device (or a proof), each non-input vertex will be marked by a rule, converting values in its predecessors to values at that vertex. It is easy to see that the vertices of every dag can be linearly ordered, such that predecessors of every vertex (if any) appear before it in the ordering. Thus, if the input vertices are labeled with some values, we can label the remaining vertices (in that order), one at a time, till all vertices (and in particular all outputs) are labeled.

For computation devices, the non-input vertices will be marked by functions (called gates), which make the dag a circuit. If we label the input vertices by specific values from some domain, the outputs will be determined by them, and the circuit will naturally define a function (from input values to output values). For more details see Section 1.2.4.

For proofs, the non-input vertices will be marked by sound deduction (or inference) rules, which make the dag a proof. If we label the inputs by formulae that are axioms in a given proof system, the output again will be determined by them, and will yield the tautology proved by this proof.

We note that both settings fit the paradigm of simplicity shared by computational models discussed in Section 1.2; the rules are simple by definition – they are applied to at most 2 previous values. The main difference is that this model is finite – each dag can compute only functions/proofs with a fixed input length. To allow all input lengths, one must consider families of dags, one for each, thus significantly extending the power of the computation model beyond that of the notion of *algorithm* (discussed in Section 1.2.3). However, as we are interested in lower bounds here, this is legitimate, and one can hope that the finiteness of the model will potentially allow for combinatorial techniques to analyze its power and limitations. Furthermore, these models allow for the introduction (and study) of meaningful restricted classes of computations.

B.2 Boolean Circuit Complexity

In Boolean circuits all inputs, outputs, and values at intermediate nodes of the dag are bits. The set of allowed gates is naturally taken to be a *complete basis* – one that allows the circuit to compute *all* Boolean functions. The specific choice of a complete basis hardly effects the study of circuit complexity. A typical choice is the set $\{\wedge, \vee, \neg\}$ of (respectively) conjunction, disjunction (each on 2 bits) and negation (on 1 bit).

For a finite function f, we denote by $\mathcal{S}(f)$ the size of the smallest Boolean circuit computing f. We will be interested in sequences of functions $\{f_n\}$, where f_n is a function on n input bits, and will study (their size complexity) $\mathcal{S}(f_n)$ asymptotically as a function of n. With some abuse of notation, for $f(x) \stackrel{\text{def}}{=} f_{|x|}(x)$, we let $\mathcal{S}(f)$ denote the integer function that assigns to n the value $\mathcal{S}(f_n)$. Thus, we refer to the following definition.

Definition B.1 (circuit complexity): *Let $f : \{0,1\}^* \to \{0,1\}^*$ and $\{f_n\}$ be such that $f(x) = f_{|x|}(x)$ for every x. The* complexity *of a function f* (resp., $\{f_n\}$), *denoted $\mathcal{S}(f)$* (resp., $n \mapsto \mathcal{S}(f_n)$), *is a function of n that represents the size of the smallest Boolean circuit computing f_n.*

We note that different circuits (in particular having a different number of inputs) are used for each f_n. Still there may be a simple description of this sequence of circuits, say, an algorithm that on input n produces a circuit computing f_n. In case such an algorithm exists and works in time polynomial in the size of its output, we say that the corresponding sequence of circuits is uniform. Note that if f has a uniform sequence of polynomial-size circuits then $f \in \mathcal{P}$. On the other hand, it can be shown that any $f \in \mathcal{P}$ has (a uniform sequence of) polynomial-size circuits. Consequently, a super-polynomial size lower-bound on any function in $\mathcal{NP}$ would imply that $\mathcal{P} \neq \mathcal{NP}$.

Definition B.1 makes no reference to "uniformity" and indeed the sequence of smallest circuits computing $\{f_n\}$ may be highly "nonuniform". Indeed, non-uniformity makes the circuit model stronger than Turing machines (or, equivalently, than the model of uniform circuits): there exist functions f that cannot be computed by Turing machines (regardless of their running time), but do have linear-size circuits. So isn't proving circuit lower bounds a much harder task than we need to resolve the P vs. NP question?

The answer is that there is a strong sentiment that the extra power provided by non-uniformity is irrelevant to the P vs. NP question; that is, it is conjectured that NP-complete sets do not have polynomial-size circuits. This conjecture is supported by the fact that its failure will yield an unexpected collapse in the complexity world of standard computations (see Section 3.2). Furthermore, the hope is that abstracting away the (supposedly irrelevant) uniformity condition will allow for combinatorial techniques to analyze the power and limitations of polynomial-size circuits (w.r.t NP-sets). This hope has materialized in the study of restricted classes of circuits (see Sections B.2.2 and B.2.3).

We also mention that Boolean circuits are a natural computational model, corresponding to "hardware complexity" (which was indeed the original motivation for their introduction by Shannon [188]), and so their study is of independent interest. Moreover, some of the techniques for analyzing Boolean functions found applications elsewhere (e.g., in computational learning theory, combinatorics and game theory).

B.2.1 Basic Results and Questions

We have already mentioned several basic facts about Boolean circuits, in particular the fact that they can efficiently simulate Turing Machines. Another basic fact is that *most Boolean functions require exponential size circuits*, which is due to the gap between the number of functions and the number of small circuits.

Thus, hard functions (i.e., function that require large circuits and thus have no efficient algorithms) do exist, to say the least. However, the aforementioned hardness result is proved via a counting argument, and so provides no way of pointing to one hard function. Using more conventional language, we cannot prove analogous hardness results for any *explicit* function f (e.g., for an NP-complete function like SAT or even for functions in $\mathcal{EXP}$). The situation is even worse: no *nontrivial* lower-bound is known for any explicit function. Note that for any function f on n bits (which depends on all its inputs), we trivially must have $\mathcal{S}(f) \geq n$, just to read the inputs. The main open problem of circuit complexity is beating this trivial bound.

Open Problem B.2 *Find an explicit Boolean function f (or even a length-preserving function f) for which $\mathcal{S}(f)$ is not $O(n)$.*

A particularly basic special case of this problem, is the question whether addition is easier to perform than multiplication. Let $\mathtt{ADD}:\{0,1\}^n \times \{0,1\}^n \to \{0,1\}^{n+1}$ and $\mathtt{MULT}:\{0,1\}^n \times \{0,1\}^n \to \{0,1\}^{2n}$, denote the addition and multiplication functions, respectively, applied to a pair of integers (presented in binary). For addition we have an optimal upper bound; that is, $\mathcal{S}(\mathtt{ADD}) = O(n)$. For multiplication, the standard (elementary school) quadratic-time algorithm can be greatly improved (via Discrete Fourier Transforms) to slightly super-linear, yielding $\mathcal{S}(\mathtt{MULT}) = O(n \cdot (\log n)^2)$. Now, the question is *whether or not there exist linear-size circuits for multiplication* (i.e., is $\mathcal{S}(\mathtt{MULT}) = O(n)$)?

Unable to report on any nontrivial lower bound (for an explicit function), we turn to restricted models. There has been some remarkable successes in developing techniques for proving strong lower bounds for natural restricted classes of circuits. We describe the most important ones, and refer the reader to [43, 218] for further detail.

General Boolean circuits, as described above, can compute every function and can do it at least as efficiently as general (uniform) algorithms. Restricted circuits may be only able to compute a subclass of all functions (e.g., monotone functions). The restriction makes sense when the related classes of functions and the computations represented by the restricted circuits are natural (from a conceptual or practical viewpoint). The models discussed below satisfy this condition.

B.2.2 Monotone Circuits

An extremely natural restriction comes by forbidding negation from the set of gates, namely allowing only $\{\wedge, \vee\}$. The resulting circuits are called monotone circuits and it is easy to see that they can compute every function $f : \{0,1\}^n \to \{0,1\}$ that is monotone with respect to the standard partial order on n-bit strings ($x \preceq y$ iff for every bit position i we have $x_i \leq y_i$).

It is as easy to see that most monotone functions require exponential size monotone circuits. Still, proving a super-polynomial lower bound on an explicit monotone function was open for over 40 years, till the invention of the so-called *approximation method* (by Razborov [174]).

Let CLIQUE be the function that, given a graph on n vertices (by its adjacency matrix), outputs 1 iff it contains a complete subgraph of size (say) $\sqrt{n}$ (namely, all pairs of vertices in some $\sqrt{n}$ subset are connected by edges). This function is clearly monotone. Moreover, it is known to be NP-complete.

Theorem B.3 ([174], improved in [7]): *There are no polynomial-size monotone circuits for* CLIQUE.

We note that the lower bounds are sub-exponential in the number of vertices (i.e., size $\exp(\Omega(n^{1/8}))$ for n vertices), and that similar lower-bounds are known for functions in $\mathcal{P}$. Thus, there exists an *exponential separation of the monotone and non-monotone circuit complexities* (referring, of course, to the computation of monotone functions).

B.2.3 Bounded-Depth Circuits

The next restriction is structural: *we allow all gates, but limit the depth of the circuit.* The depth of a dag is simply the length of the longest directed path in it. So in a sense, depth captures the *parallel time* to compute the function: if a circuit has depth d, then the function can be evaluated by enough processors in d phases (where in each phase many gates are evaluated at once). Parallel time is another important computational resource.

We will restrict d to be a constant, which still is interesting not only as parallel time, but also due to the relation of this model to expressibility in first order logic as well as to complexity classes above NP called the *Polynomial-time Hierarchy* (see Section 3.2). In the current setting (of constant-depth circuits), we allow *unbounded fan-in* (i.e., $\wedge$-gates and $\vee$-gates taking any number of incoming edges), as otherwise each output bit can depend only on a constant number of input bits.

Let PAR (for parity) denote the sum modulo two of the input bits, and MAJ (for majority) be 1 iff there are more 1's than 0's among the input bits. The invention of the *random restriction method* (by Furst, Saxe, and Sipser [79]) led to the following basic result.

Theorem B.4 ([79], improved in [222, 110]): *For all constant d, the functions* PAR *and* MAJ *have no polynomial size circuit of depth d.*

The aforementioned improvement (of Håstad [110], following Yao [110]) gives a relatively tight lower bound of $\exp(\Omega(n^{1/(d-1)}))$ on the size of n-input PAR circuits of depth d.

Interestingly, MAJ remains hard (for constant-depth polynomial-size circuits) even if the circuits are also allowed (unbounded fan-in) PAR-gates (this result is based on yet another proof technique: *approximation by polynomials* [195, 175]). However, the "converse" does not hold (i.e., constant-depth polynomial-size circuits with MAJ-gates can compute PAR), and in general the class of constant-depth polynomial-size circuits with MAJ-gates (denoted $\mathcal{TC}^0$) seems quite powerful.

B.2.4 Formula Size

The final restriction is again structural – we require the dag to be a tree. Intuitively, this forbids the computation from reusing a previously computed partial result (and if it is needed again, it has to be recomputed). The resulting circuits are simply formulae, which are natural not only for their prevalent mathematical use, but also since their size can be related to the *memory* requirements of a Turing machine. (Here we go back to the standard basis of negation ($\neg$), and $\wedge, \vee$ gates *of fan-in 2.*)

One of the oldest results on Circuit Complexity, is that PAR and MAJ are non-trivial in this model. The proof follows a simple combinatorial (or information theoretic) argument.

Theorem B.5 [134]: *Boolean formuale for n-bit* PAR *and* MAJ *require size $\Omega(n^2)$ size.*

This should be contrasted with the linear-size circuits that exist for both functions. We comment that $\mathcal{S}(\texttt{PAR}) = O(n)$ is trivial, but $\mathcal{S}(\texttt{MAJ}) = O(n)$ is not. Encouraged by Theorem B.5, one may ask whether we can hope to provide super-polynomial lower bounds on the formula size of explicit functions. This is indeed a famous open problem.

Open Problem B.6 *Find an explicit Boolean function f for which $\mathcal{S}(f)$ is super-polynomial.*

One of the cleanest methods suggested is the *communication complexity method* (of Karchmer and Wigderson [131]). This method asserts that the depth of a formula for a Boolean function f equals the communication complexity in the following two party game, G_f. The first party is given $x \in f^{-1}(1) \cap \{0,1\}^n$, the second party is given $y \in f^{-1}(0) \cap \{0,1\}^n$, and their goal is to find a bit location on which x and y disagree (i.e., i such that $x_i \neq y_i$, which cleraly exists). To that end, the party exchange messages, according to a predetermined protocol, and the question is what is the communication complexity (in terms of total number of bits exchanged on the worst-case input pair) of the best such protocol.

Note that proving a super-logarithmic lower-bound of the communication complexity of the aforementioned game G_f will establish a super-polynomial lower-bound on the size of formulae computing f (becuase formula depth can be made logarithmic in their size). We stress that a lower bound of purely information theoretic nature (no computational restriction were paced on the parties in the game) implies a computational lower bound!

We mention that the communication complexity method has a *monotone version* in which the depth of monotone circuits is related to the communication complexity of protocols that are required to find an i such that $1 = x_i \neq y_i = 0$ (rather than any i such that $x_i \neq y_i$).[2] In fact, the montone version is better known than the general one, due to its success in establishing linear lower bounds on monotone depth of natural problems such as perfect matching (by Raz and Wigderson [173]).

B.3 Arithmetic Circuits

We now leave the Boolean rind, and discuss circuits over general fields. Fix any field F. The gates of the dag will now be the standard $+$ and $\times$ operations in the field. This requires two immediate clarifications. First, to allow using constants of the field, one adds a special input vertex whose value is the constant '1' of the field. Moreover, multiplication by any field element (e.g., -1) is free. Second, one may wonder about division. However, we will be mainly interested in computing polynomials, and for computing polynomials (over infinite fields) division can be efficiently emulated by the other operations.

Now the inputs of the dag will hold elements of the field F, and hence so will all computed values at vertices. Thus an arithmetic circuit computes a polynomial map $p : F^n \to F^m$, and every such polynomial map is computed by some circuit. We denote by $\mathcal{S}_F(p)$ the size of a smallest circuit computing p (when no subscript is given, $F = \mathcal{Q}$ the field of rational numbers). As usual, we'll be interested in sequences of polynomials, one for every input size, and will study size asymptotically.

It is easy to see that over any *fixed* finite field, arithmetic circuits can simulate Boolean circuits on Boolean inputs with only constant factor loss in size. Thus the study of arithmetic circuits focuses more on infinite fields, where lower bounds may be easier to obtain.

As in the Boolean case, the existence of hard functions is easy to establish (via dimension considerations, rather than counting argument), and we will be interested in *explicit* (families of) polynomials. However, the notion of explicitness is more delicate here (e.g., allowing polynomials with algebraically independent coefficients would yield strong lower bounds, which are of no interest whatsoever). Very roughly speaking, polynomials are called explicit if the mapping from monomials to (a finite description of) their coefficients has an efficient program.

An important parameter, which is absent in the Boolean model, is the *degree* of the polynomial(s) computed. It is obvious, for example, that a degree d polynomial (even in one variable, i.e., $n = 1$) requires size at least $\log d$. We briefly consider the univariate case (in which d is the only measure of input size), which already contains

[2]Note that since f is monotone, $f(x) = 1$ and $f(y) = 0$ implies the existence of an i such that $1 = x_i \neq y_i = 0$.

striking and important problems. Then we move to the general multivariate case, in which as usual n, the number of inputs will be the main parameter. (We refer the reader to [201] for further detail.)

B.3.1 Univariate Polynomials

How tight is the $\log d$ lower bounds for the size of an arithmetic circuit computing a degree d polynomial? A simple dimension argument shows that for most degree d polynomials p, $\mathcal{S}(p) = \Omega(d)$. However, we know of no explicit one:

Open Problem B.7 *Find an explicit polynomial p of degree d, such that $\mathcal{S}(p)$ is not $O(\log d)$.*

To illustrate the question, we consider the following two concrete polynomials $p_d(x) = x^d$, and $q_d(x) = (x+1)(x+2)\cdots(x+d)$. Clearly, $\mathcal{S}(p_d) \leq 2\log d$ (via repeated squaring), so the trivial lower bound is tight. On the other hand, it is a major open problem to determine $\mathcal{S}(q_d)$, and the conjecture is that $\mathcal{S}(q_d)$ is not polynomial in $\log d$. To realize the importance of this question, we state the following proposition:

Proposition B.8 *If given d one can find in polynomial-time an arithmetic circuit computing q_d then the integer factorization problem can be solved in polynomial-time.*

Recall that it is widely believed that the integer factorization problem is intractable (i.e., cannot be solved in polynomial-time). Proposition B.8 follows by observing that $q_d(0) \equiv (d!) \pmod{N}$ for any integer N. Thus, the proposition's hypothesis provides a polynomial time algorithm for computing $(d!) \bmod N$, when given d and N. Next, observe that $\gcd(N, (d!) \bmod N)$ yields a non-trivial factor of N if some prime factors of N are smaller than d and some are larger than d. Thus, using a binary search for a suitable d, we can obtain such a non-trivial factor.

B.3.2 Multivariate Polynomials

We are now back to polynomials with n variables. To make n our only input size parameter, it is convenient to restrict ourselves to polynomials whose total degree is at most n.

Once again, almost every polynomial p in n variables requires size $\mathcal{S}(p) \geq \exp(n/2)$, via a dimension argument, and we seek explicit polynomial (families) that are hard. Unlike in the Boolean world, here there are slightly nontrivial lower bounds (via elementary tools from algebraic geometry).

Theorem B.9 [23]: $\mathcal{S}(x_1^n + x_2^n + \cdots + x_n^n) = \Omega(n \log n)$.

The same techniques extend to prove a similar lower-bound for other natural polynomials such as the symmetric polynomials and the determinant. Establishing a stronger lower-bound for any explicit polynomial is a major open problem. Another open problem is obtaining a super-linear lower bound for a polynomial map of constant (even 1) total degree. Outstanding candidates for the latter open problem are the *linear* maps computing the Discrete Fourier Transform over the Complex numbers, or the Walsh transform over the Rationals (for both $O(n \log n)$ algorithms are known, but no super-linear lower bounds are known).

We now focus on specific polynomials of central importance. The most natural and well studied candidate for the last open problem is the matrix multiplication function MM: let A, B be two $m \times m$ matrices of variables over F, and define $\mathtt{MM}(A, B)$ to be the $n = m^2$ entries of the matrix $A \times B$. Thus, MM is a set of n explicit bilinear forms over the $2n$ input variables. It is known that $\mathcal{S}_{\mathrm{GF}(2)}(\mathtt{MM}) \geq 3n$ (cf., [192]). On the other hand, the obvious $m^3 = n^{3/2}$ algorithm can be improved.

Theorem B.10 [59]: *For every field F, $\mathcal{S}_F(\mathtt{MM}) = O(n^{1.19})$.*

So what is the complexity of MM (even if one counts only multiplication gates)? Is it linear or almost-linear or is it the case that $\mathcal{S}(\mathtt{MM}) > n^\alpha$ for some $\alpha > 1$? This is indeed a famous open problem.

We next consider the determinant and permanent polynomials (DET and PER, resp.) over the $n = m^2$ variables representing an $m \times m$ matrix. While DET plays a major role in classical mathematics, PER is somewhat esoteric (though it appears in Statistical Mechanics and Quantum Mechanics). In the context of complexity theory both polynomials are of great importance, because they capture natural complexity classes. The function DET has relatively low complexity (and is closely related to the class of polynomials having polynomial-sized arithmetic formulae), whereas PER seems to have high complexity (and it is complete for the counting class $\#\mathcal{P}$ (see §6.2.1), which contains $\mathcal{NP}$). Thus, it is conjectured that PER is *not* polynomial-time reducible to DET. A specific type of reduction that makes sense in this algebraic context is a reduction by projection.

Definition B.11 (projections): *Let X and Y be two disjoint finite sets of variables, and let p and q be polynomial maps over $F[X]$ and $F[Y]$, respectively. We say that there is a* projection from p to q over F, *denoted $p \propto q$ if there exists a function $h : X \to Y \cup F$ such that $p(\bar{x}) \equiv q(h(\bar{x}))$.*

Clearly, if $p \propto q$ then $\mathcal{S}_F(p) \leq \mathcal{S}_F(q)$. Let $\mathtt{DET}_m$ and $\mathtt{PER}_m$ denote the functions DET and PER restricted to m-by-m matrices. It is known that $\mathtt{PER}_m \propto \mathtt{DET}_{3^m}$, but to yield a polynomial-time reduction one would need a projection of $\mathtt{PER}_m$ to $\mathtt{DET}_{\mathrm{poly}(m)}$. It is conjectured that no such projection exists.

B.4 Proof Complexity

The concept of *proof* is what distinguishes the study of Mathematics from all other fields of human inquiry. Mathematicians have gathered millennia of experience to attribute such adjectives to proofs as "insightful, original, deep" and most notably, "difficult". Can one quantify, mathematically, the difficulty of proving various theorems? This is exactly the task undertaken in Proof Complexity. It seeks to classify theorems according to the difficulty of proving them, much like Circuit Complexity seeks to classify functions according to the difficulty of computing them. In proofs, just like in computation, there will be a number of models, called *proof systems* capturing the power of reasoning allowed to the prover.

We will consider only propositional proof systems, and so our theorems will be *tautologies*. We will see soon why the complexity of proving tautologies is highly nontrivial and amply motivated.

The formal definition of a proof system spells out what we take for granted: the efficiency of the verification procedure. In the following definition the efficiency of the verification procedure refers to its running-time measured in terms of the *total length of the alleged theorem and proof*. In contrast, in Chapter 9, we consider the running-time as a function of the *length of the alleged theorem*. (Both approaches were mentioned in Section 2.1, where the two approaches coincide because in Section 2.1 we mandated proofs of length polynomial in the alleged theorem.)

Definition B.12 [58]: *A* (propositional) proof system *is a polynomial-time Turing machine M such that a formula T is a tautology of and only if exists a string π, called a* proof*, such that $M(\pi,T)=1$.*

In agreement with standard formalisms (see below), the proof is viewed as coming before the theorem. Note that Definition B.12 guarantees the completeness and soundness of the proof system, as well as verification efficiency (relative to the total length of the alleged proof–theorem pair). Definition B.12 judiciously ignores the length of the proof π (of the tautology T), viewing the length of the proof as a measure of the complexity of the tautology T with respect to the proof system M.

For each tautology T, let $\mathcal{PS}_M(T)$ denote the size of the shortest proof of T in M (i.e., the length of the shortest string π such that M accepts (π,T)). That is, $\mathcal{PS}_M$ captures the *proof complexity* of various tautologies with respect to the proof system M.

Abusing notation, we let $\mathcal{PS}_M(n)$ denotes the maximum $\mathcal{PS}_M(T)$ over all tautologies T of length n. The following simple theorem provides a basic connection between proof complexity (with respect to any propositional proof system) and computational complexity (i.e., the NP-vs-coNP Question).

Theorem B.13 [58]: *There exists a propositional proof system M such that $\mathcal{PS}_M$ is polynomial if and only if $\mathcal{NP}=\mathrm{co}\mathcal{NP}$.*

In particular, a propositional proof system M such that $\mathcal{PS}_M$ is polynomial coincides with a NP-proof system (as in Definition 2.5) for the set of propositional tautologies, which is a co$\mathcal{NP}$-complete set.

The long-term goal of Proof Complexity is to establish super polynomial lower bounds on the length of proofs in any propositional proof system (and thus establish $\mathcal{NP}\neq\mathrm{co}\mathcal{NP}$). It is natural to start this formidable project by considering first simple (and thus weaker) proof systems, and then move on to more and more complex ones. Moreover, various natural proof systems, capturing basic (restricted) types and "primitives" of reasoning as well as natural tautologies, suggest themselves as objects for this study. In the rest of this section we focus on such restricted proof systems. Different branches of Mathematics such as logic, algebra and geometry provide different such systems, often implicitly. A typical system would have a set of axioms, and a set of deduction rules. A proof would proceed to derive the desired tautology in a sequence of steps, each producing a formula (often called a line of the proof), which is either an axiom, or follows from previous formulae via one of the deduction rules.

Regarding these proof systems, we make two observations. First, proofs in these systems can be easily verified by an algorithm (and thus they fit the general framework of Definition B.12). Second, these proof systems perfectly fit our dag model. The inputs will be labeled by the axioms, the internal vertices by deduction rules, which in turn "infer" a formula for that vertex from the formulae at the vertices pointing to it.[3]

An important observation: For various proof systems Π, we turn to study the proof length $\mathcal{PS}_{\Pi}(T)$ of tautologies T in proof system Π. The first observation, revealing a major difference between proof complexity and circuit complexity, is that the trivial counting argument *fails*. The reason is that, while the number of functions on n bits is 2^{2^n}, there are at most 2^n tautologies of this length. Thus, in proof complexity, even the *existence* of a hard tautology, not necessarily an explicit one, would be of interest (and, in particular, if established for any propositional proof system then it would yield $\mathcal{NP} \neq \text{co}\mathcal{NP}$). (Note that here we refer to hard instances of of a problem and not to hard problems.) Anyhow, as we shall see, most known lower bounds (in restricted proof systems) apply to very natural (let alone explicit) tautologies.

Conventions: There is an equivalent and somewhat more convenient view of (simple) proof systems, namely as (simple) refutation systems. First, recalling that 3SAT is NP-complete, note that every (negation of a) tautology can be written as a conjunction of clauses, with each clause being a disjunction of only 3 literals (variables or their negation). Now, if we take these clauses as axioms, and derive (using the rules of the system) a contradiction (e.g., the negation of an axiom, or better yet the empty clause), then we have proved the tautology (since we have proved that its negation yields a contradiction). Proof complexity often takes the refutation viewpoint, and often exchanges "tautology" with its negation ("contradiction").

The rest of this section is divided to three parts, referring to logical, algebraic and geometric proof systems. We will briefly describe important representative and basic results in each of these domains, and refer the reader to [24] for further detail (and, in particular, to adequate references).

[3]General proof systems as in Definition B.12 can also be adapted to this formalism, by considering a deduction rule that corresponds to a single step of the machine M. However, the deduction rules considered below are even simpler, and more importantly they are natural.

B.4.1 Logical Proof Systems

The proof systems in this section will all have lines that are Boolean formulae, and the differences will be in the structural limits imposed on these formulae.

The most basic proof system, called Frege system, puts no restriction on the formulae manipulated by the proof. It has one derivation rule, called the cut rule: $A \vee C, B \vee \neg C \vdash A \vee B$ (adding any other sound rule, like *modus ponens*, has little effect on the length of proofs in this system). Frege systems are basic in the sense that they (in several variants) are the most common in Logic, and in that polynomial length proofs in these systems naturally corresponds to "polynomial-time reasoning" about feasible objects.

The major open problem in proof complexity is to find any tautology (as usual, we mean a family of tautologies) that has no polynomial-size proof in the Frege system.

Since lower bounds for Frege are hard, we turn to subsystems of Frege which are interesting and natural. The most widely studied system is Resolution, whose importance stems from its use by most propositional (as well as first order) automated theorem provers. The formulae allowed in Resolution refutations are simply clauses (disjunctions), and so the derivation cut rule simplifies to the "resolution rule": $A \vee x, B \vee \neg x \vdash A \vee B$, for clauses A, B and variable x.

An example of a tautology that is easy for Frege and hard for Resolution, is the pigeonhole principle, $\mathtt{PHP}_n^m$, expressing the fact that there is no one-to-one mapping of m pigeons to $n < m$ holes.

Theorem B.14 $\mathcal{PS}_{\rm Frege}(\mathtt{PHP}_n^{n+1}) = n^{O(1)}$ *but* $\mathcal{PS}_{\rm Resolution}(\mathtt{PHP}_n^{n+1}) = 2^{\Omega(n)}$

B.4.2 Algebraic Proof Systems

Just as a natural contradiction in the Boolean setting is an unsatisfiable collection of clauses, a natural contradiction in the algebraic setting is a system of polynomials without a common root. Moreover, CNF formulae can be easily converted to a system of polynomials, one per clause, over any field. One often adds the polynomials $x_i^2 - x_i$ which ensure Boolean values.

A natural proof system (related to Hilbert's Nullstellensatz, and to computations of Grobner bases in symbolic algebra programs) is Polynomial Calculus, abbreviated PC. The lines in this system are polynomials (represented explicitly by all coefficients), and it has two deduction rules: For any two polynomials g, h, the rule $g, h \vdash g + h$, and for any polynomial g and variable x_i, the rule $g, x_i \vdash x_i g$. Strong size lower bounds (obtained from degree lower bounds) are known for this system. For example, encoding the pigeonhole principle as a contradicting set of constant degree polynomials, we have

Theorem B.15 *For every n and every $m > n$, $\mathcal{PS}_{\rm PC}(\mathtt{PHP}_n^m) \geq 2^{n/2}$, over every field.*

B.4.3 Geometric Proof Systems

Yet another natural way to represent contradictions is a by a set of regions in space that have empty intersection. Again, we care mainly about discrete (say, Boolean) domains, and a wide source of interesting contradictions are Integer Programs from Combinatorial Optimization. Here, the constraints are (affine) linear inequalities with integer coefficients (so the regions are subsets of the Boolean cube carved out by halfspaces). The most basic system is called Cutting Planes (CP). Its lines are linear inequalities with integer coefficients. Its deduction rules are (the obvious) addition of inequalities, and the (less obvious) division of the coefficients by a constant (and rounding, taking advantage of the integrality of the solution space).

While $\mathtt{PHP}_n^m$ is easy in this system, exponential lower bounds are known for other tautologies. We mention that they are obtained from the *monotone circuit* lower bounds of Section B.2.2.

Appendix C

On the Foundations of Modern Cryptography

It is possible to build a cabin with no foundations, but not a lasting building.

Eng. Isidor Goldreich (1906–1995)

Summary: Cryptography is concerned with the construction of computational systems that withstand any abuse: Such a system is constructed so to maintain a desired functionality, even under malicious attempts aimed at making it deviate from this functionality.

This appendix is aimed at presenting the foundations of cryptography, which are the paradigms, approaches and techniques used to conceptualize, define and provide solutions to natural security concerns. It presents some of these conceptual tools as well as some of the fundamental results obtained using them. The emphasis is on the clarification of fundamental concepts, and on demonstrating the feasibility of solving several central cryptographic problems. The presentation assumes basic knowledge of algorithms, probability theory and complexity theory, but nothing beyond this.

The appendix augments the treatment of one-way functions, pseudorandom generators and zero-knowledge proofs, which is given in Sections 7.1, 8.3 and 9.2, respectively. (These augmentations are important for cryptography, but are less central to the main context of this book and thus were omitted from the main text.) Using these basic tools, the appendix provides a treatment of basic cryptographic applications such as Encryption, Signatures, and General Cryptographic Protocols.

C.1 Introduction and Preliminaries

The vast expansion and rigorous treatment of cryptography is one of the major achievements of theoretical computer science. In particular, concepts such as computational indistinguishability, pseudorandomness and zero-knowledge interactive proofs were introduced, classical notions such as secure encryption and unforgeable signatures were placed on sound grounds, and new (unexpected) directions and connections were uncovered. Indeed, modern cryptography is strongly linked to complexity theory (in contrast to "classical" cryptography which is strongly related to information theory).

C.1.1 Modern cryptography

Modern cryptography is concerned with the construction of information systems that are robust against malicious attempts to make these systems deviate from their prescribed functionality. The prescribed functionality may be the private and authenticated communication of information through the Internet, the holding of incoercible and secret electronic voting, or conducting any "fault-resilient" multi-party computation. Indeed, the scope of modern cryptography is very broad, and it stands in contrast to "classical" cryptography (which has focused on the single problem of enabling secret communication over insecure communication media).

The design of cryptographic systems is a very difficult task. One cannot rely on intuitions regarding the "typical" state of the environment in which the system operates. For sure, the adversary attacking the system will try to manipulate the environment into "untypical" states. Nor can one be content with counter-measures designed to withstand specific attacks, since the adversary (which acts after the design of the system is completed) will try to attack the schemes in ways that are different from the ones the designer had envisioned. Although the validity of the foregoing assertions seems self-evident, still some people hope that in practice ignoring these tautologies will not result in actual damage. Experience shows that these hopes rarely come true; cryptographic schemes based on make-believe are broken, typically sooner than later.

In view of the foregoing, we believe that it makes little sense to make assumptions regarding the specific *strategy* that the adversary may use. The only assumptions that can be justified refer to the computational *abilities* of the adversary. Furthermore, the design of cryptographic systems has to be based on *firm foundations*; whereas ad-hoc approaches and heuristics are a very dangerous way to go. A heuristic may make sense when the designer has a very good idea regarding the environment in which a scheme is to operate, yet a cryptographic scheme has to operate in a maliciously selected environment that typically transcends the designer's view.

This appendix is aimed at presenting the foundations for cryptography. The foundations of cryptography are the paradigms, approaches and techniques used to conceptualize, define and provide solutions to natural "security concerns". We will present some of these paradigms, approaches and techniques as well as some of the fundamental results obtained using them. Our emphasis is on the clarification of fundamental concepts and on demonstrating the feasibility of solving several central cryptographic problems.

Solving a cryptographic problem (or addressing a security concern) is a two-stage process consisting of a *definitional stage* and a *constructive stage*. First, in the definitional stage, the functionality underlying the natural concern is to be identified, and an adequate cryptographic problem has to be defined. Trying to list all undesired situations is infeasible and prone to error. Instead, one should define the functionality in terms of operation in an imaginary ideal model, and require a candidate solution to emulate this operation in the real, clearly defined, model (which specifies the adversary's abilities). Once the definitional stage is completed, one proceeds to construct a system that satisfies the definition. Such a construction may use some simpler tools, and its security is proved relying on the features of these tools. In practice, of course, such a scheme may need to satisfy also some *specific* efficiency requirements.

This appendix focuses on several archetypical cryptographic problems (e.g., encryption and signature schemes) and on several central tools (e.g., computational difficulty, pseudorandomness, and zero-knowledge proofs). For each of these problems (resp., tools), we start by presenting the natural concern underlying it (resp., its intuitive objective), then define the problem (resp., tool), and finally demonstrate that the problem may be solved (resp., the tool can be constructed). In the latter step, our focus is on demonstrating the feasibility of solving the problem, not on providing a practical solution.

Computational Difficulty

The aforementioned tools and applications (e.g., secure encryption) exist only if some sort of computational hardness exists. Specifically, all these problems and tools require (either explicitly or implicitly) the ability to generate instances of hard problems. Such ability is captured in the definition of one-way functions. Thus, one-way functions are the very minimum needed for doing most natural tasks of cryptography. (It turns out, as we shall see, that this necessary condition is "essentially" sufficient; that is, the existence of one-way functions (or augmentations and extensions of this assumption) suffices for doing most of cryptography.)

Our current state of understanding of efficient computation does not allow us to prove that one-way functions exist. In particular, if $\mathcal{P} = \mathcal{NP}$ then no one-way functions exist. Furthermore, the existence of one-way functions implies that $\mathcal{NP}$ is not contained in $\mathcal{BPP} \supseteq \mathcal{P}$ (not even "on the average"). Thus, proving that one-way functions exist is not easier than proving that $\mathcal{P} \neq \mathcal{NP}$; in fact, the former task seems significantly harder than the latter. Hence, we have no choice (at this stage of history) but to assume that one-way functions exist. As justification to this assumption we can only offer the combined beliefs of hundreds (or thousands) of researchers. Furthermore, these beliefs concern a simply stated assumption, and their validity follows from several widely believed conjectures which are central to various fields (e.g., the conjectured intractability of integer factorization is central to computational number theory).

Since we need assumptions anyhow, why not just assume what we want (i.e., the existence of a solution to some natural cryptographic problem)? Well, first we need to know what we want: as stated above, we must first clarify what exactly we want; that is, go through the typically complex definitional stage. But once this stage is completed, can we just assume that the definition derived can be met? Not really: once a definition is derived, how can we know that it can at all be met? The way to demonstrate that a definition is viable (and that the corresponding intuitive security concern can be satisfied at all) is to construct a solution based on a *better understood* assumption (i.e., one that is more common and widely believed). For example, looking at the definition of zero-knowledge proofs, it is not a-priori clear that such proofs exist at all (in a non-trivial sense). The non-triviality of the notion was first demonstrated by presenting a zero-knowledge proof system for statements, regarding Quadratic Residuosity, which are believed to be hard to verify (without extra information). Furthermore, contrary to prior beliefs, it was later shown that the existence of one-way functions implies that any NP-statement can be proved in zero-knowledge. Thus, facts that were not known at all to hold (and even believed to be false), were shown to hold by reduction to widely believed assumptions (without which most of modern cryptography collapses anyhow). To summarize, not all assumptions are equal, and so reducing a complex, new and doubtful assumption to a widely-believed and simple (or even merely simpler) assumption is of great value. Furthermore, reducing the solution of a new task to the assumed security of a well-known primitive typically means providing a construction that, using the known primitive, solves the new task. This means that we do not only know (or assume) that the new task is solvable but we also have a solution based on a primitive that, being well-known, typically has several candidate implementations.

C.1.2 Preliminaries

Modern Cryptography, as surveyed here, is concerned with the construction of *efficient* schemes for which it is *infeasible* to violate the security feature. Thus, we need a notion of efficient computations as well as a notion of infeasible ones. The computations of the legitimate users of the scheme ought be efficient, whereas violating the security features (by an adversary) ought to be infeasible. We stress that we do not identify feasible computations with efficient ones, but rather view the former notion as potentially more liberal. Let us elaborate.

Efficient Computations and Infeasible ones

Efficient computations are commonly modeled by computations that are polynomial-time in the security parameter. The polynomial bounding the running-time of the legitimate user's strategy is *fixed and typically explicit* (and *small*). Indeed, our aim is to have a notion of efficiency that is as strict as possible (or, equivalently, develop strategies that are as efficient as possible). Here (i.e., when referring to the complexity of the legitimate users) we are in the same situation as in any algorithmic setting. Things are different when referring to our assumptions regarding the computational resources of the adversary, where we refer to the notion of feasible, which we wish to be as wide as possible. A common approach is to postulate that feasible computations are polynomial-time too, but here the polynomial is *not a-priori specified* (and is to be thought of as arbitrarily large). In other words, the adversary is restricted to the class of polynomial-time computations and anything beyond this is considered to be infeasible.

Although many definitions explicitly refer to the convention of associating feasible computations with polynomial-time ones, this convention is *inessential* to any of the results known in the area. In all cases, a more general statement can be made by referring to a general notion of feasibility, which should be preserved under standard algorithmic composition, yielding theories that refer to adversaries of running-time bounded by any specific super-polynomial function (or class of functions). Still, for sake of concreteness and clarity, we shall use the former convention in our formal definitions (but our motivational discussions will refer to an unspecified notion of feasibility that covers at least efficient computations).

Randomized (or probabilistic) Computations

Randomized computations play a central role in cryptography. One fundamental reason for this fact is that randomness is essential for the existence (or rather the generation) of secrets. Thus, we must allow the legitimate users to employ randomized computations, and certainly (since we consider randomization as feasible) we must consider also adversaries that employ randomized computations. This brings up the issue of success probability: typically, we require that legitimate users succeed (in fulfilling their legitimate goals) with probability 1 (or negligibly close to this), whereas adversaries succeed (in violating the security features) with negligible probability. Thus, the notion of a negligible probability plays an important role in our exposition. One requirement of the definition of negligible probability is to provide a robust notion of rareness: A rare event should occur rarely even if we repeat the experiment for a feasible number of times. That is, in case we consider any polynomial-time computation to be feasible, a function $\mu:\mathbb{N}\to\mathbb{N}$ is called negligible if $1-(1-\mu(n))^{p(n)} < 0.01$ for every polynomial p and sufficiently big n (i.e., μ is negligible if for every positive polynomial p' the function $\mu(\cdot)$ is upper-bounded by $1/p'(\cdot)$). However, if we consider the function $T(n)$ to provide our notion of infeasible computation then functions upper-bounded by $1/T(n)$ are considered negligible (in n).

We will also refer to the notion of noticeable probability. Here the requirement is that events that occur with noticeable probability, will occur almost surely (i.e., except with negligible probability) if we repeat the experiment for a polynomial number of times. Thus, a function $\nu:\mathbb{N}\to\mathbb{N}$ is called noticeable if for some positive polynomial p' the function $\nu(\cdot)$ is lower-bounded by $1/p'(\cdot)$.

C.1.3 Prerequisites, Organization, and Beyond

Our aim is to present the basic concepts, techniques and results in cryptography. As stated, our emphasis is on the clarification of fundamental concepts and the relationship among them. This is done in a way independent of the particularities of some popular number theoretic examples. These particular examples played a central role in the development of the field and still offer the most practical implementations of all cryptographic primitives, but this does not mean that the presentation has to be linked to them. On the contrary, we believe that concepts are best clarified when presented at an abstract level, decoupled from specific implementations. Thus, the most relevant background for this appendix is provided by basic knowledge of algorithms (including randomized ones), computability and elementary probability theory.

The appendix is organized in two main parts, corresponding to the Basic Tools of Cryptography and the Basic Applications of Cryptography. The basic tools consist of computational difficulty (one-way functions), pseudorandomness and zero-knowledge proofs. These basic tools are used for the basic applications, which in turn consist of Encryption Schemes, Signature Schemes, and General Cryptographic Protocols.

The basic tools: The most basic tool is computational difficulty, which in turn is captured by the notion of one-way functions. Another notion of key importance is that of computational indistinguishability, underlying the theory of pseudorandomness as well as much of the rest of cryptography. Pseudorandom generators and functions are important tools that are frequently used. So are zero-knowledge proofs, playing a key role in the design of secure cryptographic protocols.

The basic applications: Encryption and signature schemes are the most basic applications of Cryptography. Their main utility is in providing secret and reliable communication over insecure communication media. Loosely speaking, encryption schemes are used for ensuring the secrecy (or privacy) of the actual information being communicated, whereas signature schemes are used to ensure its reliability (or authenticity). Another basic topic is the construction of secure cryptographic protocols for the implementation of arbitrary functionalities.

The presentation of the basic tools in Sections C.2–C.4 augments (and sometimes repeats parts of) Sections 7.1, 8.3, and 9.2 (which provide a basic treatment of one-way functions, pseudorandom generators, and zero-knowledge proofs, respectively). Sections C.5–C.7, provide a overview of the basic applications; that is, Encryption Schemes, Signature Schemes, and General Cryptographic Protocols.

Suggestions for further reading. This appendix is a brief summary of the author's two-volume work on the subject [87, 88]. Furthermore, the first part (i.e., Basic Tools) corresponds to [87], whereas the second part (i.e., Basic Applications) corresponds to [88]. Needless to say, the interested reader is referred to these textbooks for further detail (and, in particular, for missing references).

Practice. The aim of this appendix is to introduce the reader to the *theoretical foundations* of cryptography. As argued, such foundations are necessary for *sound* practice of cryptography. Indeed, practice requires more than theoretical foundations, whereas the current text makes no attempt to provide anything beyond the latter. However, given a sound foundation, one can learn and evaluate various practical suggestions that appear elsewhere. On the other hand, lack of sound foundations results in inability to critically evaluate practical suggestions, which in turn leads to unsound decisions. *Nothing could be more harmful to the design of schemes that need to withstand adversarial attacks than misconceptions about such attacks.*

C.2 Computational Difficulty

Modern Cryptography is concerned with the construction of systems that are easy to operate (properly) but hard to foil. Thus, a complexity gap (between the ease of proper usage and the difficulty of deviating from the prescribed functionality) lies at the heart of Modern Cryptography. However, gaps as required for Modern Cryptography are not known to exist; they are only widely believed to exist. Indeed, almost all of Modern Cryptography rises or falls with the question of whether one-way functions exist. We mention that the existence of one-way functions implies that $\mathcal{NP}$ contains search problems that are hard to solve *on the average*, which in turn implies that $\mathcal{NP}$ is not contained in $\mathcal{BPP}$ (i.e., a worst-case complexity conjecture).

Loosely speaking, one-way functions are functions that are easy to evaluate but hard (on the average) to invert. Such functions can be thought of as an efficient way of generating "puzzles" that are infeasible to solve (i.e., the puzzle is a random image of the function and a solution is a corresponding preimage). Furthermore, the person generating the puzzle knows a solution to it and can efficiently verify the validity of (possibly other) solutions to the puzzle. Thus, one-way functions have, by definition, a clear cryptographic flavor (i.e., they manifest a gap between the ease of one task and the difficulty of a related one).

C.2.1 One-Way Functions

One-way functions are functions that are efficiently computable but infeasible to invert (in an average-case sense). That is, a function $f:\{0,1\}^* \to \{0,1\}^*$ is called one-way if there is an efficient algorithm that on input x outputs $f(x)$, whereas any feasible algorithm that tries to find a preimage of $f(x)$ under f may succeed only with negligible probability (where the probability is taken uniformly over the choices of x and the algorithm's coin tosses). Associating feasible computations with probabilistic polynomial-time algorithms, we obtain the following definition.

Definition C.1 (one-way functions, Definition 7.1 restated): *A function $f:\{0,1\}^* \to \{0,1\}^*$ is called* one-way *if the following two conditions hold:*

1. easy to evaluate: *There exist a polynomial-time algorithm A such that $A(x) = f(x)$ for every $x \in \{0,1\}^*$.*

2. hard to invert: *For every probabilistic polynomial-time algorithm A', every polynomial p, and all sufficiently large n,*

$$\mathsf{Pr}[A'(f(x), 1^n) \in f^{-1}(f(x))] \; < \; \frac{1}{p(n)}$$

where the probability is taken uniformly over $x \in \{0,1\}^n$ and all the internal coin tosses of algorithm A'.

Some of the most popular candidates for one-way functions are based on the conjectured intractability of computational problems in number theory. One such conjecture is that it is infeasible to factor large integers. Consequently, the function that takes as input two (equal length) primes and outputs their product is widely believed to be a one-way function. Furthermore, factoring such a composite is infeasible if and only if squaring modulo such a composite is a one-way function (see [170]). For certain composites (i.e., products of two primes that are both congruent to 3 mod 4), the latter function induces a permutation over the set of quadratic residues modulo this composite. A related permutation, which is widely believed to be one-way, is the RSA function [180]: $x \mapsto x^e \bmod N$, where $N = P \cdot Q$ is a composite as above, e is relatively prime to $(P-1) \cdot (Q-1)$, and $x \in \{0, ..., N-1\}$. The latter examples (as well as other popular suggestions) are better captured by the following formulation of a collection of one-way functions (which is indeed related to Definition C.1):

Definition C.2 (collections of one-way functions): *A collection of functions, $\{f_i : D_i \to \{0,1\}^*\}_{i \in \overline{I}}$, is called* one-way *if there exists three probabilistic polynomial-time algorithms, I, D and F, so that the following two conditions hold*

1. easy to sample and compute: *On input 1^n, the output of* (the index selection) *algorithm I is distributed over the set $\overline{I} \cap \{0,1\}^n$ (i.e., is an n-bit long index of some function). On input (an index of a function) $i \in \overline{I}$, the output of* (the domain sampling) *algorithm D is distributed over the set D_i (i.e., over the domain of the function). On input $i \in \overline{I}$ and $x \in D_i$,* (the evaluation) *algorithm F* always *outputs $f_i(x)$.*

2. hard to invert:[1] *For every probabilistic polynomial-time algorithm, A', every positive polynomial $p(\cdot)$, and all sufficiently large n's*

$$\mathsf{Pr}\left[A'(i, f_i(x)) \in f_i^{-1}(f_i(x))\right] < \frac{1}{p(n)}$$

where $i \leftarrow I(1^n)$ and $x \leftarrow D(i)$.

The collection is said to be a collection of permutations *if each of the f_i's is a permutation over the corresponding D_i, and $D(i)$ is almost uniformly distributed in D_i.*

[1] Note that this condition refers to the distributions $I(1^n)$ and $D(i)$, which are merely required to range over $\overline{I} \cap \{0,1\}^n$ and D_i, respectively. (Typically, the distributions $I(1^n)$ and $D(i)$ are (almost) uniform over $\overline{I} \cap \{0,1\}^n$ and D_i, respectively.)

For example, in case of the RSA, one considers $f_{N,e} : D_{N,e} \to D_{N,e}$ that satisfies $f_{N,e}(x) = x^e \bmod N$, where $D_{N,e} = \{0, ..., N-1\}$. Definition C.2 is also a good starting point for the definition of a trapdoor permutation.[2] Loosely speaking, the latter is a collection of one-way permutations augmented with an efficient algorithm that allows for inverting the permutation when given adequate auxiliary information (called a trapdoor).

Definition C.3 (trapdoor permutations): *A collection of permutations as in Definition C.2 is called a* trapdoor permutation *if there are two auxiliary probabilistic polynomial-time algorithms I' and F^{-1} such that (1) the distribution $I'(1^n)$ ranges over pairs of strings so that the first string is distributed as in $I(1^n)$, and (2) for every (i,t) in the range of $I'(1^n)$ and every $x \in D_i$ it holds that $F^{-1}(t, f_i(x)) = x$.* (That is, t is a trapdoor that allows to invert f_i.)

For example, in case of the RSA, $f_{N,e}$ can be inverted by raising to the power d (modulo $N = P \cdot Q$), where d is the multiplicative inverse of e modulo $(P-1)\cdot(Q-1)$. Indeed, in this case, the trapdoor information is (N, d).

Strong versus weak one-way functions (summary of Section 7.1.2). Recall that the foregoing definitions require that any feasible algorithm *succeeds in inverting* the function *with negligible probability.* A weaker notion only requires that any feasible algorithm *fails to invert* the function *with noticeable probability.* It turns out that the existence of such weak one-way functions implies the existence of strong one-way functions (as defined above). The construction itself is straightforward: one just parses the argument to the new function into sufficiently many blocks, and applies the weak one-way function on the individual blocks. We warn that the hardness of inverting the resulting function is not established by mere "combinatorics" (i.e., considering the relative volume of S^t in U^t, for $S \subset U$, where S represents the set of "easy to invert" images). Specifically, one may *not* assume that the potential inverting algorithm works independently on each block. Indeed this assumption seems reasonable, but we should not assume that the adversary behaves in a reasonable way (unless we can actually prove that it gains nothing by behaving in other ways, which seem unreasonable to us). Instead, the hardness of inverting the resulting function is proved via a so called "reducibility argument" (which is used to prove all conditional results in the area). Specifically, we show that any algorithm that inverts the resulting function F with non-negligible success probability can be used to construct an algorithm that inverts the original function f with success probability that violates the hypothesis (regarding f). The reduction is presented in Section 7.1.2.

C.2.2 Hard-Core Predicates

Recall that saying that a function f is one-way implies that given y (in the range of f) it is infeasible to find a preimage of y under f. This does not mean that it

[2] Indeed, a more adequate term would be a collection of trapdoor permutations, but the shorter (and less precise) term is the commonly used one.

is infeasible to find out partial information about the preimage(s) of y under f. Specifically it may be easy to retrieve half of the bits of the preimage (e.g., given a one-way function f consider the function g defined by $g(x,r) \stackrel{\rm def}{=} (f(x),r)$, for every $|x|=|r|$). As will become clear in subsequent sections, hiding partial information (about the function's preimage) plays an important role in more advanced constructs (e.g., secure encryption). This partial information can be considered as a "hard core" of the difficulty of inverting f. Loosely speaking, a *polynomial-time computable* (Boolean) predicate b, is called a hard-core of a function f if no feasible algorithm, given $f(x)$, can guess $b(x)$ with success probability that is non-negligibly better than one half. The actual definition is presented in Section 7.1.3 (i.e., Definition 7.6).

Note that if b is a hard-core of a 1-1 function f that is polynomial-time computable then f is a one-way function. On the other hand, recall that Theorem 7.7 asserts that *for any one-way function f, the inner-product mod 2 of x and r is a hard-core of $f'(x,r) = (f(x),r)$.*

C.3 Pseudorandomness

In practice "pseudorandom" sequences are often used instead of truly random sequences. The underlying belief is that if an (efficient) application performs well when using a truly random sequence then it will perform essentially as well when using a "pseudorandom" sequence. However, this belief is not supported by ad-hoc notions of "pseudorandomness" such as passing the statistical tests in [136] or having large linear-complexity (as in [108]). Needless to say, using such "pseudorandom" sequences in a cryptographic application (instead of truly random sequences) is very dangerous.

In contrast, truly random sequences can be safely replaced by pseudorandom sequences provided that pseudorandom distributions are defined as being computationally indistinguishable from the uniform distribution. Such a definition makes the soundness of the replacement an easy corollary. Loosely speaking, pseudorandom generators are then defined as efficient procedures for creating long pseudorandom sequences based on few truly random bits (i.e., a short random seed). The relevance of such constructs to cryptography is in providing legitimate users that share short random seeds a method for creating large objects that look random to any feasible adversary (which does not know the said seed).

C.3.1 Computational Indistinguishability

A central notion in Modern Cryptography is that of "effective similarity" (a.k.a computational indistinguishability; cf. [104, 221]). The underlying thesis is that we do not care whether or not objects are equal, all we care about is whether or not a difference between the objects can be observed by a feasible computation. In case the answer is negative, the two objects are equivalent as far as any practical application is concerned. Indeed, in the sequel we will often interchange such (computationally indistinguishable) objects. In this section we reproduce the definition of computational indistinguishability (presented in Section 8.3), and consider two variants.

Definition C.4 (computational indistinguishability, Definition 8.4 restated): *We say that* $X = \{X_n\}_{n\in\mathbb{N}}$ *and* $Y = \{Y_n\}_{n\in\mathbb{N}}$ *are* computationally indistinguishable *if for every probabilistic polynomial-time algorithm* D *every polynomial* p*, and all sufficiently large* n*,*

$$|\Pr[D(X_n)=1] - \Pr[D(Y_n)=1]| \;<\; \frac{1}{p(n)}$$

where the probabilities are taken over the relevant distribution (i.e., either X_n or Y_n) *and over the internal coin tosses of algorithm* D.

See further discussion in Section 8.3.3. Recall that, for "efficiently constructible" distributions, indistinguishability by a single sample (as defined above) implies indistinguishability by multiple samples (as in Definition 8.5).

Extension to ensembles indexed by strings. Here we refer to a natural extension of Definition C.4: Rather than referring to ensembles indexed by $\mathbb{N}$, we refer to ensembles indexed by a set $S \subseteq \{0,1\}^*$. Typically, for an ensemble $\{Z_\alpha\}_{\alpha\in S}$, it holds that Z_α ranges over strings of length that is polynomially-related to the length of α.

Definition C.5 *We say that* $\{X_\alpha\}_{\alpha\in S}$ *and* $\{Y_\alpha\}_{\alpha\in S}$ *are* computationally indistinguishable *if for every probabilistic polynomial-time algorithm* D *every polynomial* p*, and all sufficiently long* $\alpha \in S$*,*

$$|\Pr[D(\alpha, X_\alpha)=1] - \Pr[D(\alpha, Y_\alpha)=1]| \;<\; \frac{1}{p(|\alpha|)}$$

where the probabilities are taken over the relevant distribution (i.e., either X_α *or* Y_α*) and over the internal coin tosses of algorithm* D.

Note that a minor (natural) variant of Definition C.4 is obtained as a special case by setting $S = \{1^n : n \in \mathbb{N}\}$. In this case, the distinguisher is explicitly given the index n of the distribution that it inspects. (In typical applications, the difference is immaterial because the index n is easily determined from any sample of the corresponding distributions.)

A non-uniform version. A non-uniform definition of computational indistinguishability can be derived from Definition C.5 by artificially augmenting the indices of the distributions. That is, $\{X_\alpha\}_{\alpha\in S}$ and $\{Y_\alpha\}_{\alpha\in S}$ are computationally indistinguishable in a non-unform sense if for every polynomial p the ensembles $\{X'_{\alpha'}\}_{\alpha'\in S'}$ and $\{Y'_{\alpha'}\}_{\alpha'\in S'}$ are computationally indistinguishable (as in Definition C.5), where $S' = \{\alpha\beta : \alpha \in S \wedge \beta \in \{0,1\}^{p(|\alpha|)}\}$ and $X'_{\alpha\beta} = X_\alpha$ (resp., $Y'_{\alpha\beta} = Y_\alpha$) for every $\beta \in \{0,1\}^{p(|\alpha|)}$. An equivalent (alternative) formulation underlies Definition 8.12.

C.3.2 Pseudorandom Generators

Loosely speaking, a pseudorandom generator is an efficient (deterministic) algorithm that on input a short random *seed* outputs a (typically much) longer sequence that is computationally indistinguishable from a uniformly chosen sequence.

Definition C.6 (pseudorandom generator, Definition 8.1 restated): *Let* $\ell:\mathbb{N}\rightarrow\mathbb{N}$ *satisfy* $\ell(n) > n$*, for all* $n \in \mathbb{N}$*. A* pseudorandom generator*, with* stretch function ℓ*, is a* (deterministic) *polynomial-time algorithm* G *satisfying the following:*

1. *For every* $s \in \{0,1\}^*$*, it holds that* $|G(s)| = \ell(|s|)$*.*
2. $\{G(U_n)\}_{n\in\mathbb{N}}$ *and* $\{U_{\ell(n)}\}_{n\in\mathbb{N}}$ *are computationally indistinguishable, where* U_m *denotes the uniform distribution over* $\{0,1\}^m$*.*

Indeed, the probability ensemble $\{G(U_n)\}_{n\in\mathbb{N}}$ *is called* pseudorandom*.*

We stress that pseudorandom sequences can replace truly random sequences not only in "standard" algorithmic applications but also in cryptographic ones. That is, *any* cryptographic application that is secure when the legitimate parties use truly random sequences, is also secure when the legitimate parties use pseudorandom sequences. The benefit in such a substitution (of random sequences by pseudorandom ones) is that the latter sequences can be efficiently generated using much less true randomness. Furthermore, *in an interactive setting*, it is possible to eliminate all random steps from the on-line execution of a program, by replacing them with the generation of pseudorandom bits based on a random seed selected and fixed off-line (or at set-up time).

Various cryptographic applications of pseudorandom generators will be presented in the sequel, but let us first recall that *pseudorandom generators exist if and only if one-way functions exist* (see Theorem 8.11). For further treatment of pseudorandom generators, the interested reader is referred to Section 8.3.

C.3.3 Pseudorandom Functions

Pseudorandom generators provide a way to efficiently generate long pseudorandom sequences from short random seeds. Pseudorandom functions, introduced and constructed by Goldreich, Goldwasser, and Micali [91], are even more powerful: they provide efficient direct access to bits of a huge pseudorandom sequence (which is not feasible to scan bit-by-bit). More precisely, a pseudorandom function is an efficient (deterministic) algorithm that given an n-bit *seed*, s, and an n-bit *argument*, x, returns an n-bit string, denoted $f_s(x)$, such that it is infeasible to distinguish the values of f_s, for a uniformly chosen $s \in \{0,1\}^n$, from the values of a truly random function $F : \{0,1\}^n \rightarrow \{0,1\}^n$. That is, the (feasible) testing procedure is given oracle access to the function (but not its explicit description), and cannot distinguish the case it is given oracle access to a pseudorandom function from the case it is given oracle access to a truly random function.

Definition C.7 (pseudorandom functions): *A* pseudorandom function (ensemble), *is a collection of functions* $\{f_s:\{0,1\}^{|s|}\rightarrow\{0,1\}^{|s|}\}_{s\in\{0,1\}^*}$ *that satisfies the following two conditions:*

1. (efficient evaluation) *There exists an efficient* (deterministic) *algorithm that given a* seed, *s, and an* argument, $x \in \{0,1\}^{|s|}$, *returns* $f_s(x)$.

2. (pseudorandomness) *For every probabilistic polynomial-time oracle machine, M, every positive polynomial p and all sufficiently large n's*

$$\left|\,\mathsf{Pr}[M^{f_{U_n}}(1^n)=1]-\mathsf{Pr}[M^{F_n}(1^n)=1]\,\right| \;<\; \frac{1}{p(n)}$$

   where F_n *denotes a uniformly selected function mapping* $\{0,1\}^n$ *to* $\{0,1\}^n$.

One key feature of the foregoing definition is that pseudorandom functions can be generated and shared by merely generating and sharing their seed; that is, a "random looking" function $f_s : \{0,1\}^n \rightarrow \{0,1\}^n$, is determined by its n-bit seed s. Parties wishing to share a "random looking" function f_s (determining 2^n-many values), merely need to generate and share among themselves the n-bit seed s. (For example, one party may randomly select the seed s, and communicate it, via a secure channel, to all other parties.) Sharing a pseudorandom function allows parties to determine (by themselves and without any further communication) random-looking values depending on their current views of the environment (which need not be known a priori). To appreciate the potential of this tool, one should realize that sharing a pseudorandom function is essentially as good as being able to agree, on the fly, on the association of random values to (on-line) given values, where the latter are taken from a huge set of possible values. We stress that this agreement is achieved without communication and synchronization: Whenever some party needs to associate a random value to a given value, $v \in \{0,1\}^n$, it will associate to v the (same) random value $r_v \in \{0,1\}^n$ (by setting $r_v = f_s(v)$, where f_s is a pseudorandom function agreed upon beforehand).

Theorem C.8 (How to construct pseudorandom functions): *Pseudorandom functions can be constructed using any pseudorandom generator.*

Proof Sketch:[3] Let G be a pseudorandom generator that stretches its seed by a factor of two (i.e., $\ell(n) = 2n$), and let $G_0(s)$ (resp., $G_1(s)$) denote the first (resp., last) $|s|$ bits in $G(s)$. Define

$$G_{\sigma_{|s|}\cdots\sigma_2\sigma_1}(s) \stackrel{\rm def}{=} G_{\sigma_{|s|}}(\cdots G_{\sigma_2}(G_{\sigma_1}(s))\cdots).$$

We consider the function ensemble $\{f_s:\{0,1\}^{|s|}\rightarrow\{0,1\}^{|s|}\}_{s\in\{0,1\}^*}$, where $f_s(x) \stackrel{\rm def}{=} G_x(s)$. Pictorially, the function f_s is defined by n-step walks down a full binary tree of depth n having labels at the vertices. The root of the tree, hereafter referred to as the level 0 vertex of the tree, is labeled by the string s. If an internal vertex is

[3] See details in [87, Sec. 3.6.2].

labeled r then its left child is labeled $G_0(r)$ whereas its right child is labeled $G_1(r)$. The value of $f_s(x)$ is the string residing in the leaf reachable from the root by a path corresponding to the string x.

We claim that this function ensemble $\{f_s\}_{s\in\{0,1\}^*}$ is pseudorandom. The proof uses the hybrid technique (cf. Section 8.3.3): The i^{th} hybrid, H_n^i, is a function ensemble consisting of $2^{2^i\cdot n}$ functions $\{0,1\}^n \to \{0,1\}^n$, each determined by 2^i random n-bit strings, denoted $\overline{s} = \langle s_\beta\rangle_{\beta\in\{0,1\}^i}$. The value of such function $h_{\overline{s}}$ at $x = \alpha\beta$, where $|\beta| = i$, is defined to equal $G_\alpha(s_\beta)$. (Pictorially, the function $h_{\overline{s}}$ is defined by placing the strings in $\overline{s}$ in the corresponding vertices of level i, and labeling vertices of lower levels using the very rule used in the definition of f_s.) The extreme hybrids correspond to our indistinguishability claim (i.e., $H_n^0 \equiv f_{U_n}$ and H_n^n is a truly random function), and neighboring hybrids can be related to our indistinguishability hypothesis (specifically, to the indistinguishability of $G(U_n)$ and U_{2n} under multiple samples). $\square$

Variants. Useful variants (and generalizations) of the notion of pseudorandom functions include Boolean pseudorandom functions that are defined for all strings (i.e., $f_s : \{0,1\}^* \to \{0,1\}$) and pseudorandom functions that are defined for other domains and ranges (i.e., $f_s : \{0,1\}^{d(|s|)} \to \{0,1\}^{r(|s|)}$, for arbitrary polynomially bounded functions $d, r : \mathbb{N} \to \mathbb{N}$). Various transformations between these variants are known (cf. [87, Sec. 3.6.4] and [88, Apdx. C.2]).

Applications and a generic methodology. Pseudorandom functions are a very useful cryptographic tool: One may first design a cryptographic scheme assuming that the legitimate users have black-box access to a random function, and next implement the random function using a pseudorandom function. The usefulness of this tool stems from the fact that having (black-box) access to a random function gives the legitimate parties a potential advantage over the adversary (which does not have free access to this function).[4] The security of the resulting implementation (which uses a pseudorandom function) is established in two steps: First one proves the security of an idealized scheme that uses a truly random function, and next one argues that the actual implementation (which uses a pseudorandom function) is secure (as otherwise one obtains an efficient oracle machine that distinguishes a pseudorandom function from a truly random one).

C.4 Zero-Knowledge

Zero-knowledge proofs provide a powerful tool for the design of cryptographic protocols. Loosely speaking, zero-knowledge proofs are proofs that yield nothing beyond the validity of the assertion. That is, a verifier obtaining such a proof only gains conviction in the validity of the assertion (as if it was told by a trusted party

[4]The aforementioned methodology is sound provided that the adversary does not get the description of the pseudorandom function (i.e., the seed) in use, but has only (possibly limited) oracle access to it. This is different from the so-called Random Oracle Methodology.

that the assertion holds). This is formulated by saying that anything that is feasibly computable from a zero-knowledge proof is also feasibly computable from the (valid) assertion itself. The latter formulation follows the simulation paradigm, which is discussed next, while reproducing part of the discussion in §9.2.1.1 and making additional comments regarding the use of this paradigm in cryptography.

C.4.1 The Simulation Paradigm

A key question regarding the modeling of security concerns is how to express the intuitive requirement that an adversary "gains nothing substantial" by deviating from the prescribed behavior of an honest user. Our approach is that the adversary *gains nothing* if whatever it can obtain by unrestricted adversarial behavior can also be obtained within essentially the same computational effort by a benign behavior. The definition of the "benign behavior" captures what we want to achieve in terms of security, and is specific to the security concern to be addressed. For example, in the previous paragraph, we said that a proof is zero-knowledge if it yields nothing (to the adversarial verifier) beyond the validity of the assertion (i.e., the benign behavior is any computation that is based (only) on the assertion itself, while assuming that the latter is valid). Other examples are discussed in Sections C.5.1 and C.7.1.

A notable property of the simulation paradigm, as well as of the entire definitional approach surveyed here, is that this approach is overly liberal with respect to its view of the abilities of the adversary as well as to what might constitute a gain for the adversary. Thus, the approach may be considered overly cautious, because it prohibits also "non-harmful" gains of some "far fetched" adversaries. We warn against this impression. Firstly, there is nothing more dangerous in cryptography than to consider "reasonable" adversaries (a notion which is almost a contradiction in terms): typically, the adversaries will try exactly what the system designer has discarded as "far fetched". Secondly, it seems impossible to come up with definitions of security that distinguish "breaking the scheme in a harmful way" from "breaking it in a non-harmful way": what is harmful is application-dependent, whereas a good definition of security ought to be application-independent (as otherwise using the scheme in any new application will require a full re-evaluation of its security). Furthermore, even with respect to a specific application, it is typically very hard to classify the set of "harmful breakings".

C.4.2 The Actual Definition

In Section 9.2.1.2 zero-knowledge was defined as a property of some prover strategies (within the context of interactive proof systems, as defined in Section 9.1.1). More generally, the term may apply to any interactive machine, regardless of its goal. A strategy A is zero-knowledge on (inputs from) the set S if, for every feasible strategy B^*, there exists a feasible computation C^* such that the following two probability ensembles are computationally indistinguishable (according to Definition C.5):

1. $\{(A, B^*)(x)\}_{x \in S} \stackrel{\text{def}}{=}$ the output of B^* after interacting with A on common input $x \in S$; and

2. $\{C^*(x)\}_{x \in S} \stackrel{\text{def}}{=}$ the output of C^* on input $x \in S$.

Recall that the first ensemble represents an actual execution of an interactive protocol, whereas the second ensemble represents the computation of a stand-alone procedure (called the "simulator"), which does not interact with anybody.

The foregoing definition does *not* account for auxiliary information that an adversary B^* may have prior to entering the interaction. Accounting for such auxiliary information is essential for using zero-knowledge proofs as subprotocols inside larger protocols. This is taken care of by a stricter notion called auxiliary-input zero-knowledge, which was not presented in Section 9.2.

Definition C.9 (zero-knowledge, revisited): *A strategy A is* auxiliary-input zero-knowledge *on inputs from S if, for every probabilistic polynomial-time strategy B^* and every polynomial p, there exists a probabilistic polynomial-time algorithm C^* such that the following two probability ensembles are computationally indistinguishable:*

1. $\{(A, B^*(z))(x)\}_{x \in S\,,\, z \in \{0,1\}^{p(|x|)}} \stackrel{\text{def}}{=}$ *the output of B^* when having auxiliary-input z and interacting with A on common input $x \in S$; and*

2. $\{C^*(x, z)\}_{x \in S\,,\, z \in \{0,1\}^{p(|x|)}} \stackrel{\text{def}}{=}$ *the output of C^* on inputs $x \in S$ and $z \in \{0,1\}^{p(|x|)}$.*

Almost all known zero-knowledge proofs are in fact auxiliary-input zero-knowledge. As hinted, *auxiliary-input zero-knowledge is preserved under sequential composition.* A simulator for the multiple-session protocol can be constructed by iteratively invoking the single-session simulator that refers to the residual strategy of the adversarial verifier in the given session (while feeding this simulator with the transcript of previous sessions). Indeed, the residual single-session verifier gets the transcript of the previous sessions as part of its auxiliary input (i.e., z in Definition C.9). For details, see [87, Sec. 4.3.4].

C.4.3 A construction and a generic application

A question avoided so far is whether zero-knowledge proofs exist at all. Clearly, every set in $\mathcal{P}$ (or rather in $\mathcal{BPP}$) has a "trivial" zero-knowledge proof (in which the verifier determines membership by itself); however, what we seek is zero-knowledge proofs for statements that the verifier cannot decide by itself.

Assuming the existence of "commitment schemes" (see below), which in turn exist if one-way functions exist [157, 113], *there exist* (auxiliary-input) *zero-knowledge proofs of membership in any NP-set.* These zero-knowledge proofs, abstractly depicted in Construction 9.10, have the following important property: the prescribed prover strategy is efficient, provided it is given as auxiliary-input an NP-witness to the assertion (to be proved).[5] Indeed, by using the standard Karp-reductions to 3-Colorability, the protocol of Construction 9.10 can be used for obtaining zero-knowledge proofs for any set in $\mathcal{NP}$. Implementing the abstract boxes refered to in Construction 9.10 by commitment schemes, we get:

Theorem C.10 (On the applicability of zero-knowledge proofs): *If one-way functions exist then every set $S \in \mathcal{NP}$ has an auxiliary-input zero-knowledge interactive proof. Furthermore, the prescribed prover strategy can be implemented in probabilistic polynomial-time, provided it is given as auxiliary-input an NP-witness for membership of the common input in S.*

Theorem C.10 makes zero-knowledge a very powerful tool in the design of cryptographic schemes and protocols (see below). We comment that the intractability assumption used in Theorem C.10 seems essential.

Commitment schemes. Loosely speaking, commitment schemes are two-stage (two-party) protocols allowing for one party to commit itself (at the first stage) to a value while keeping the value secret. In a (second) latter stage, the commitment is "opened" and it is guaranteed that the "opening" can yield only a single value determined in the committing phase. Thus, the (first stage of the) commitment scheme is both *binding* and *hiding*. A simple (uni-directional communication) commitment scheme can be constructed based on any one-way 1-1 function f (with a corresponding hard-core b). To commit to a bit σ, the sender uniformly selects $s \in \{0,1\}^n$, and sends the pair $(f(s), b(s) \oplus \sigma)$. Note that this is both binding and hiding. An alternative construction, which can be based on any one-way function, uses a pseudorandom generator G that stretches its seed by a factor of three (cf. Theorem 8.11). A commitment is established, via two-way communication, as follows (cf. [157]): The receiver selects uniformly $r \in \{0,1\}^{3n}$ and sends it to the sender, which selects uniformly $s \in \{0,1\}^n$ and sends $r \oplus G(s)$ if it wishes to commit to the value one and $G(s)$ if it wishes to commit to zero. To see that this is binding, observe that there are at most 2^{2n} "bad" values r that satisfy $G(s_0) = r \oplus G(s_1)$ for some pair (s_0, s_1), and with overwhelmingly high probability the receiver will not pick one of these bad values. The hiding property follows by the pseudorandomness of G.

Efficiency considerations. The number of rounds in a protocol is commonly considered the most important efficiency criterion (or complexity measure), and typically one desires to have it be a constant. However, in order to obtain negligible soundness error, the protocol of Construction 9.10 has to be invoked for a non-constant number of times (and the analysis of the resulting protocol relies on the preservation of zero-knowledge under sequential composition). At first

[5] The auxiliary-input given to the prescribed prover (in order to allow for an efficient implementation of its strategy) is not to be confused with the auxiliary-input that is given to malicious verifiers (in the definition of auxiliary-input zero-knowledge). The former is typically an NP-witness for the common input, which is available to the user that invokes the prover strategy (cf. the generic application discussed below). In contrast, the auxiliary-input that is given to malicious verifiers models arbitrary partial information that may be available to the adversary.

glance, it seems that one can derive a constant-round zero-knowledge proof system (of negligible soundness error) by performing these invocations in parallel (rather than sequentially). Unfortunately, it is not clear that the resulting interactive proof is zero-knowledge. Still, under standard intractability assumptions (e.g., the intractability of factoring), constant-round zero-knowledge proofs (of negligible soundness error) do exist for every set in $\mathcal{NP}$.

A generic application. As mentioned, Theorem C.10 makes zero-knowledge a very powerful tool in the design of cryptographic schemes and protocols. This wide applicability is due to two important aspects regarding Theorem C.10: Firstly, Theorem C.10 provides a zero-knowledge proof for every NP-set, and secondly the prescribed prover can be implemented in probabilistic polynomial-time when given an adequate NP-witness. We now turn to a typical application of zero-knowledge proofs. In a typical cryptographic setting, a user U has a secret and is supposed to take some action based on its secret. The question is how can other users verify that U indeed took the correct action (as determined by U's secret and publicly known information). Indeed, if U discloses its secret then anybody can verify that U took the correct action. However, U does not want to reveal its secret. Using zero-knowledge proofs we can satisfy both conflicting requirements (i.e., having other users verify that U took the correct action without violating U's interest in not revealing its secret). That is, U can prove in zero-knowledge that it took the correct action. Note that U's claim to having taken the correct action is an NP-assertion (since U's legal action is determined as a polynomial-time function of its secret and the public information), and that U has an NP-witness to its validity (i.e., the secret is an NP-witness to the claim that the action fits the public information). Thus, by Theorem C.10, it is possible for U to efficiently prove the correctness of its action without yielding anything about its secret. Consequently, it is fair to ask U to prove (in zero-knowledge) that it behaves properly, and so to force U to behave properly. Indeed, "forcing proper behavior" is the canonical application of zero-knowledge proofs (see §C.7.3.2).

This paradigm (i.e., "forcing proper behavior" via zero-knowledge proofs), which in turn is based on the fact that zero-knowledge proofs can be constructed for any NP-set, has been utilized in numerous different settings. Indeed, this paradigm is the basis for the wide applicability of zero-knowledge protocols in Cryptography.

C.4.4 Variants and Issues

In this section we consider numerous variants on the notion of zero-knowledge and the underlying model of interactive proofs. These include black-box simulation and other variants of zero-knowledge (cf. Section C.4.4.1), as well as notions such as proofs of knowledge, non-interactive zero-knowledge, and witness indistinguishable proofs (cf. Section C.4.4.2).

Before starting, we call the reader's attension to the notion of computational soundness and to the resulting argument systems, discussed in §9.1.4.2. We mention that argument systems may be more efficient than interactive proofs as well as provide stronger zero-knowledge guarantees. Specifically, perfect zero-knowledge arguments for $\mathcal{NP}$ can be constructed based on some reasonable assumptions [46], where perfect zero-knowledge means that the simulator's output is distributed *identically* to the verifier's view in the real interaction (see Definition 9.7 or a discussion in §C.4.4.1). Note that stronger security for the prover (as provided by perfect zero-knowledge) comes at the cost of weaker security for the verifier (as provided by computational soundness). The answer to the question of whether or not this trade-off is worthwhile seems to be application dependent, and one should also take into account the availability and complexity of the corresponding protocols.

C.4.4.1 Definitional variations

We consider several definitional issues regarding the notion of zero-knowledge (as defined in Definition C.9).

Universal and black-box simulation. Further strengthening of Definition C.9 is obtained by requiring the existence of a universal simulator, denoted $\mathcal{C}$, that is given the program of the verifier (i.e., B^*) as an auxiliary-input; that is, in terms of Definition C.9, one should replace $C^*(x, z)$ by $\mathcal{C}(x, z, \langle B^* \rangle)$, where $\langle B^* \rangle$ denotes the description of the program of B^* (which may depend on x and on z). That is, we effectively restrict the simulation by requiring that it be a uniform (feasible) function of the verifier's program (rather than arbitrarily depend on it). This restriction is very natural, because it seems hard to envision an alternative way of establishing the zero-knowledge property of a given protocol. Taking another step, one may argue that since it seems infeasible to reverse-engineer programs, the simulator may as well just use the verifier strategy as an oracle (or as a "black-box"). This reasoning gave rise to the notion of black-box simulation, which was introduced and advocated in [94] and further studied in numerous works. The belief was that inherent limitations regarding black-box simulation represent inherent limitations of zero-knowledge itself. For example, it was believed that the *fact* that the parallel version of the interactive proof of Construction 9.10 cannot be simulated in a black-box manner (unless $\mathcal{NP}$ is contained in $\mathcal{BPP}$) *implies* that this version is not zero-knowledge (as per Definition C.9 itself). However, the (underlying) belief that *any* zero-knowledge protocol can be simulated in a black-box manner was refuted recently by Barak [22].

Honest verifier versus general cheating verifier. Definition C.9 refers to all feasible verifier strategies, which is most natural (in the cryptographic setting) because zero-knowledge is supposed to capture the robustness of the prover under *any feasible* (i.e., adversarial) attempt to gain something by interacting with it. A weaker and still interesting notion of zero-knowledge refers to what can be gained by an "honest verifier" (or rather a semi-honest verifier)[6] that interacts

[6]The term "honest verifier" is more appealing when considering an alternative (equivalent) formulation of Definition C.9. In the alternative definition (see [87, Sec. 4.3.1.3]), the simulator is "only" required to generate the verifier's view of the real interaction, where the verifier's view includes its (common and auxiliary) inputs, the outcome of its coin tosses, and all messages it has received.

with the prover as directed, with the exception that it may maintain (and output) a record of the entire interaction (i.e., even if directed to erase all records of the interaction). Although such a weaker notion is not satisfactory for standard cryptographic applications, it yields a fascinating notion from a conceptual as well as a complexity-theoretic point of view. Furthermore, every proof system that is *zero-knowledge with respect to the honest-verifier* can be transformed into a *standard zero-knowledge* proof (without using intractability assumptions and in case of "public-coin" proofs this is done without significantly increasing the prover's computational effort; see [212]).

Statistical versus Computational Zero-Knowledge. Recall that Definition C.9 postulates that for every probability ensemble of one type (i.e., representing the verifier's output after interaction with the prover) there exists a "similar" ensemble of a second type (i.e., representing the simulator's output). One key parameter is the interpretation of "similarity". Three interpretations, yielding different notions of zero-knowledge, have been commonly considered in the literature:

1. Perfect Zero-Knowledge requires that the two probability ensembles be identical.[7]

2. Statistical Zero-Knowledge requires that these probability ensembles be statistically close (i.e., the variation distance between them is negligible).

3. Computational (or rather general) Zero-Knowledge requires that these probability ensembles be computationally indistinguishable.

Indeed, Computational Zero-Knowledge is the most liberal notion, and is the notion considered in Definition C.9. We note that the class of problems having statistical zero-knowledge proofs contains several problems that are considered intractable. The interested reader is referred to [211].

Strict versus expected probabilistic polynomial-time. So far, we did not specify what we exactly mean by the term probabilistic polynomial-time. Two common interpretations are:

1. Strict probabilistic polynomial-time. That is, there exist a (polynomial in the length of the input) bound on the *number of steps in each possible run* of the machine, regardless of the outcome of its coin tosses.

2. Expected probabilistic polynomial-time. The standard approach is to look at the running-time as a random variable and *bound its expectation* (by a polynomial in the length of the input). As observed by Levin (cf. [87, Sec. 4.3.1.6]), this definitional approach is quite problematic (e.g., it is not model-independent and is not closed under algorithmic composition), and an alternative treatment of this random variable is preferable.

[7] The actual definition of Perfect Zero-Knowledge allows the simulator to fail (while outputting a special symbol) with negligible probability, and the output distribution of the simulator is conditioned on its not failing.

Consequently, the notion of expected polynomial-time raises a variety of conceptual and technical problems. For that reason, whenever possible, one should prefer the more robust (and restricted) notion of strict (probabilistic) polynomial-time. Thus, with the *exception of constant-round* zero-knowledge protocols, whenever we talked of a probabilistic polynomial-time verifier (resp., simulator) we mean one in the strict sense. In contrast, with a couple of exceptions (e.g., [22]), all results regarding *constant-round* zero-knowledge protocols refer to a strict polynomial-time verifier and an expected polynomial-time simulator, which is indeed a small cheat.

C.4.4.2 Related notions: POK, NIZK, and WI

We briefly discuss the notions of proofs of knowledge (POK), non-interactive zero-knowledge (NIZK), and witness indistinguishable proofs (WI).

Proofs of Knowledge. Loosely speaking, proofs of knowledge (cf. [105]) are interactive proofs in which the prover asserts "knowledge" of some object (e.g., a 3-coloring of a graph), and not merely its existence (e.g., the existence of a 3-coloring of the graph, which in turn is equivalent to the assertion that the graph is 3-colorable). See further discussion in Section 9.2.3. We mention that "proofs of knowledge", and in particular zero-knowledge "proofs of knowledge", have many applications to the design of cryptographic schemes and cryptographic protocols. One famous application of zero-knowledge proofs of knowledge is to the construction of identification schemes (e.g., the Fiat-Shamir scheme).

Non-Interactive Zero-Knowledge. The model of non-interactive zero-knowledge proof systems consists of three entities: a prover, a verifier and a uniformly selected reference string (which can be thought of as being selected by a trusted third party). Both the verifier and prover can read the reference string (as well as the common input), and each can toss additional coins. The interaction consists of a single message sent from the prover to the verifier, who is then left with the final decision (whether or not to accept the common input). The (basic) zero-knowledge requirement refers to a simulator that outputs pairs that should be computationally indistinguishable from the distribution (of pairs consisting of a uniformly selected reference string and a random prover message) seen in the real model.[8] Non-interactive zero-knowledge proof systems have numerous applications (e.g., to the construction of public-key encryption and signature schemes, where the reference string may be incorporated in the public-key). Several different definitions of non-interactive zero-knowledge proofs were considered in the literature (see [87, Sec. 4.10] and [88, Sec. 5.4.4.4]). Constructing non-interactive zero-knowledge proofs seems more difficult than constructing interactive zero-knowledge proofs. Still, based on standard intractability assumptions (e.g., intractability of factoring), it is known how to construct a non-interactive zero-knowledge proof for any NP-set.

[8] Note that the verifier does not effect the distribution seen in the real model, and so the basic definition of zero-knowledge does not refer to it. The verifier (or rather a process of adaptively selecting assertions to be proved) is referred to in the adaptive variants of the definition.

Witness Indistinguishability. The notion of witness indistinguishability was suggested in [72] as a meaningful relaxation of zero-knowledge. Loosely speaking, for any NP-relation R, a proof (or argument) system for the corresponding NP-set is called witness indistinguishable if no feasible verifier may distinguish the case in which the prover uses one NP-witness to x (i.e., w_1 such that $(x, w_1) \in R$) from the case in which the prover is using a different NP-witness to the same input x (i.e., w_2 such that $(x, w_2) \in R$). Clearly, any zero-knowledge protocol is witness indistinguishable, but the converse does not necessarily hold. Furthermore, it seems that witness indistinguishable protocols are easier to construct than zero-knowledge ones. Another advantage of witness indistinguishable protocols is that they are closed under arbitrary concurrent composition, whereas in general zero-knowledge protocols are not closed even under parallel composition. Witness indistinguishable protocols turned out to be an *important tool in the construction of more complex protocols*. We refer, in particular, to the technique of [71] for constructing zero-knowledge proofs (and arguments) based on witness indistinguishable proofs (resp., arguments).

C.5 Encryption Schemes

The problem of providing *secret communication over insecure media* is the traditional and most basic problem of cryptography. The setting of this problem consists of two parties communicating through a channel that is possibly tapped by an adversary. The parties wish to exchange information with each other, but keep the "wire-tapper" as ignorant as possible regarding the contents of this information. The canonical solution to this problem is obtained by the use of encryption schemes. Loosely speaking, an encryption scheme is a protocol allowing these parties to communicate *secretly* with each other. Typically, the encryption scheme consists of a pair of algorithms. One algorithm, called encryption, is applied by the sender (i.e., the party sending a message), while the other algorithm, called decryption, is applied by the receiver. Hence, in order to send a message, the sender first applies the encryption algorithm to the message, and sends the result, called the ciphertext, over the channel. Upon receiving a ciphertext, the other party (i.e., the receiver) applies the decryption algorithm to it, and retrieves the original message (called the plaintext).

In order for the foregoing scheme to provide secret communication, the communicating parties (at least the receiver) must know something that is not known to the wire-tapper. (Otherwise, the wire-tapper can decrypt the ciphertext exactly as done by the receiver.) This extra knowledge may take the form of the decryption algorithm itself, or some parameters and/or auxiliary inputs used by the decryption algorithm. We call this extra knowledge the decryption-key. Note that, without loss of generality, we may assume that the decryption algorithm is known to the wire-tapper, and that the decryption algorithm operates on two inputs: a ciphertext and a decryption-key. (This description implicitly presupposes the existence of an efficient algorithm for generating (random) keys.) We stress that the existence of a decryption-key, not known to the wire-tapper, is merely a necessary condition for secret communication.

Evaluating the "security" of an encryption scheme is a very tricky business. A preliminary task is to understand what is "security" (i.e., to properly define what is meant by this intuitive term). Two approaches to defining security are known. The first ("classical") approach, introduced by Shannon [190], is *information theoretic*. It is concerned with the "information" about the plaintext that is "present" in the ciphertext. Loosely speaking, if the ciphertext contains information about the plaintext then the encryption scheme is considered insecure. It has been shown that such high (i.e., "perfect") level of security can be achieved only if the key in use is at least as long as the *total* amount of information sent via the encryption scheme [190]. This fact (i.e., that the key has to be longer than the information exchanged using it) is indeed a drastic limitation on the applicability of such (perfectly-secure) encryption schemes.

The second ("modern") approach, followed in the current text, is based on *computational complexity*. This approach is based on the thesis that it *does not matter* whether the ciphertext contains information about the plaintext, but rather whether this information can be *efficiently extracted*. In other words, instead of asking whether it is *possible* for the wire-tapper to extract specific information, we ask whether it is *feasible* for the wire-tapper to extract this information. It turns out that the new (i.e., "computational complexity") approach can offer security even when the key is much shorter than the total length of the messages sent via the encryption scheme.

The computational complexity approach enables the introduction of concepts and primitives that cannot exist under the information theoretic approach. A typical example is the concept of *public-key encryption schemes*, introduced by Diffie and Hellman [62] (with the most popular candidate suggested by Rivest, Shamir, and Adleman [180]). Recall that in the foregoing discussion we concentrated on the decryption algorithm and its key. It can be shown that the encryption algorithm must get, in addition to the message, an auxiliary input that depends on the decryption-key. This auxiliary input is called the encryption-key. Traditional encryption schemes, and in particular all the encryption schemes used in the millennia until the 1980's, operate with an encryption-key that equals the decryption-key. Hence, the wire-tapper in these schemes must be ignorant of the encryption-key, and consequently the *key distribution* problem arises; that is, how can two parties wishing to communicate over an insecure channel agree on a secret encryption/decryption key. (The traditional solution is to exchange the key through an alternative channel that is secure though (much) more expensive to use.) The computational complexity approach allows the introduction of encryption schemes in which the encryption-key may be given to the wire-tapper without compromising the security of the scheme. Clearly, the decryption-key in such schemes is different from the encryption-key, and furthermore it is infeasible to obtain the decryption-key from the encryption-key. Such encryption schemes, called public-key schemes, have the advantage of trivially resolving the key distribution problem (because the encryption-key can be publicized). That is, once some Party X generates a pair of keys and publicizes the encryption-key, any party can send encrypted messages to

Party X so that Party X can retrieve the actual information (i.e., the plaintext), whereas nobody else can learn anything about the plaintext.

In contrast to public-key schemes, traditional encryption schemes in which the encryption-key equals the description-key are called private-key schemes, because in these schemes the encryption-key must be kept secret (rather than be public as in public-key encryption schemes). We note that a full specification of either schemes requires the specification of the way in which keys are generated; that is, a (randomized) key-generation algorithm that, given a security parameter, produces a (random) pair of corresponding encryption/decryption keys (which are identical in case of private-key schemes).

Thus, both private-key and public-key encryption schemes consist of three efficient algorithms: a key generation algorithm denoted G, an encryption algorithm denoted E, and a decryption algorithm denoted D. For every pair of encryption and decryption keys (e, d) generated by G, and for every plaintext x, it holds that $D_d(E_e(x)) = x$, where $E_e(x) \stackrel{\text{def}}{=} E(e, x)$ and $D_d(y) \stackrel{\text{def}}{=} D(d, y)$. The difference between the two types of encryption schemes is reflected in the definition of security: the security of a public-key encryption scheme should hold also when the adversary is given the encryption-key, whereas this is not required for a private-key encryption scheme. Below we focus on the public-key case (and the private-key case can be obtained by omitting the encryption-key from the sequence of inputs given to the adversary).

C.5.1 Definitions

A good disguise should not reveal the person's height.

Shafi Goldwasser and Silvio Micali, 1982

For simplicity, we first consider the encryption of a single message (which, for further simplicity, is assumed to be of length n).[9] As implied by the foregoing discussion, a public-key encryption scheme is said to be secure if it is infeasible to gain any information about the plaintext by looking at the ciphertext (and the encryption-key). That is, whatever information about the plaintext one may compute from the ciphertext and some a-priori information, can be essentially computed as efficiently from the a-priori information alone. This fundamental definition of security (called semantic security) turns out to be equivalent to saying that, for any two messages, it is infeasible to distinguish the encryption of the first message from the encryption of the second message, even when given the encryption-key. Both definitions were introduced by Goldwasser and Micali [104].

Definition C.11 (semantic security): *A* public-key *encryption scheme* (G, E, D) *is* semantically secure *if for every probabilistic polynomial-time algorithm, A, there exists a probabilistic polynomial-time algorithm B so that for every two functions*

[9] In the case of public-key schemes no generality is lost by these simplifying assumptions, but in the case of private-key schemes one should consider the encryption of polynomially-many messages (as we do below).

$f, h : \{0,1\}^* \to \{0,1\}^*$ *such that* $|h(x)| = \text{poly}(|x|)$, *and all probability ensembles* $\{X_n\}_{n\in\mathbb{N}}$, *where* X_n *is a random variable ranging over* $\{0,1\}^n$, *it holds that*

$$\Pr[A(e, E_e(x), h(x)) = f(x)] \; < \; \Pr[B(1^n, h(x)) = f(x)] + \mu(n)$$

where the plaintext x *is distributed according to* X_n, *the encryption-key* e *is distributed according to* $G(1^n)$, *and* μ *is a negligible function.*

That is, it is feasible to predict $f(x)$ from $h(x)$ as successfully as it is to predict $f(x)$ from $h(x)$ and $(e, E_e(x))$, which means that nothing is gained by obtaining $(e, E_e(x))$. Note that no computational restrictions are made regarding the functions h and f. We stress that the above definition (as well as the next one) refers to public-key encryption schemes, and in the case of private-key schemes algorithm A is not given the encryption-key e.

A good disguise should not allow a mother to distinguish her own children.

Shafi Goldwasser and Silvio Micali, 1982

The following technical interpretation of security states that it is infeasible to distinguish the encryptions of two plaintexts (of the same length).

Definition C.12 (indistinguishability of encryptions): *A public-key encryption scheme* (G, E, D) *has* indistinguishable encryptions *if for every probabilistic polynomial-time algorithm,* A, *and all sequences of triples,* $(x_n, y_n, z_n)_{n\in\mathbb{N}}$, *where* $|x_n| = |y_n| = n$ *and* $|z_n| = \text{poly}(n)$,

$$|\Pr[A(e, E_e(x_n), z_n) = 1] - \Pr[A(e, E_e(y_n), z_n) = 1]| = \mu(n)$$

Again, e *is distributed according to* $G(1^n)$, *and* μ *is a negligible function.*

In particular, z_n may equal (x_n, y_n). Thus, it is infeasible to distinguish the encryptions of any two fixed messages (such as the all-zero message and the all-ones message).

Definition C.11 is more appealing in most settings where encryption is considered the end goal. Definition C.12 is used to establish the security of candidate encryption schemes as well as to analyze their application as modules inside larger cryptographic protocols. Thus, their equivalence is of major importance.

Equivalence of Definitions C.11 and C.12 – proof ideas. Intuitively, indistinguishability of encryptions (i.e., of the encryptions of x_n and y_n) is a special case of semantic security; specifically, it corresponds to the case that X_n is uniform over $\{x_n, y_n\}$, the function f indicates one of the plaintexts and h does not distinguish them (i.e., $f(w) = 1$ iff $w = x_n$ and $h(x_n) = h(y_n) = z_n$, where z_n is as in Definition C.12). The other direction is proved by considering the algorithm B that, on input $(1^n, v)$ where $v = h(x)$, generates $(e, d) \leftarrow G(1^n)$ and outputs $A(e, E_e(1^n), v)$, where A is as in Definition C.11. Indistinguishability of encryptions is used to prove that B performs as well as A (i.e., for every h, f and $\{X_n\}_{n\in\mathbb{N}}$, it holds that $\Pr[B(1^n, h(X_n)) = f(X_n)] = \Pr[A(e, E_e(1^n), h(X_n)) = f(X_n)]$ approximately equals $\Pr[A(e, E_e(X_n), h(X_n)) = f(X_n)]$).

Probabilistic Encryption: A secure *public-key* encryption scheme must employ a probabilistic (i.e., randomized) encryption algorithm. Otherwise, given the encryption-key as (additional) input, it is easy to distinguish the encryption of the all-zero message from the encryption of the all-ones message.[10] This explains the association of the robust security definitions and the method of *probabilistic encryption*, an association that goes back to the title of the pioneering work of Goldwasser and Micali [104].

Further discussion: We stress that (the equivalent) Definitions C.11 and C.12 go way beyond saying that it is infeasible to recover the plaintext from the ciphertext. The latter statement is indeed a minimal requirement from a secure encryption scheme, but is far from being a sufficient requirement. Typically, encryption schemes are used in applications where even obtaining partial information on the plaintext may endanger the security of the application. When designing an application-independent encryption scheme, we do not know which partial information endangers the application and which does not. Furthermore, even if one wants to design an encryption scheme tailored to a specific application, it is rare (to say the least) that one has a precise characterization of all possible partial information that endanger this application. Thus, we need to require that it is infeasible to obtain any information about the plaintext from the ciphertext. Furthermore, in most applications the plaintext may not be uniformly distributed and some a-priori information regarding it may be available to the adversary. We require that the secrecy of all partial information is preserved also in such a case. That is, even in presence of a-priori information on the plaintext, it is infeasible to obtain any (new) information about the plaintext from the ciphertext (beyond what is feasible to obtain from the a-priori information on the plaintext). The definition of semantic security postulates all of this. The equivalent definition of indistinguishability of encryptions is useful in demonstrating the security of candidate constructions as well as for arguing about their effect as part of larger protocols.

Security of multiple messages: Definitions C.11 and C.12 refer to the security of an encryption scheme that is used to encrypt a single plaintext (per a generated key). Since the plaintext may be longer than the key[11], these definitions are already non-trivial, and an encryption scheme satisfying them (even in the private-key model) implies the existence of one-way functions. Still, in many cases, it is desirable to encrypt many plaintexts using the same encryption-key. Loosely speaking, an encryption scheme is secure in the multiple-messages setting if conditions as in Definition C.11 (resp., Definition C.12) hold when polynomially-

[10] The same holds for (stateless) *private-key* encryption schemes, when considering the security of encrypting several messages (rather than a single message as done above). For example, if one uses a deterministic encryption algorithm then the adversary can distinguish two encryptions of the same message from the encryptions of a pair of different messages.

[11] Recall that for sake of simplicity we have considered only messages of length n, but the general definitions refer to messages of arbitrary (polynomial in n) length. We comment that, in the general form of Definition C.11, one should provide the length of the message as an auxiliary input to both algorithms (A and B).

many plaintexts are encrypted using the same encryption-key (cf. [88, Sec. 5.2.4]). *In the public-key model*, security in the single-message setting implies security in the multiple-messages setting. We stress that this is not necessarily true *for the private-key model.*

C.5.2 Constructions

It is common practice to use "pseudorandom generators" as a basis for private-key encryption schemes. We stress that this is a very dangerous practice when the "pseudorandom generator" is easy to predict (such as the linear congruential generator or some modifications of it that output a constant fraction of the bits of each resulting number). However, this common practice becomes sound provided one uses pseudorandom generators (as defined in Section C.3.2). An alternative and more flexible construction follows.

Private-Key Encryption Scheme based on Pseudorandom Functions: We present a simple construction that uses pseudorandom functions as defined in Section C.3.3. The key generation algorithm consists of selecting a seed, denoted s, for a (pseudorandom) function, denoted f_s. To encrypt a message $x \in \{0,1\}^n$ (using key s), the encryption algorithm uniformly selects a string $r \in \{0,1\}^n$ and produces the ciphertext $(r, x \oplus f_s(r))$, where $\oplus$ denotes the exclusive-or of bit strings. To decrypt the ciphertext (r, y) (using key s), the decryption algorithm just computes $y \oplus f_s(r)$. The proof of security of this encryption scheme consists of two steps (suggested as a general methodology in Section C.3.3):

1. Proving that an idealized version of the scheme, in which one uses a uniformly selected function $F:\{0,1\}^n \to \{0,1\}^n$, rather than the pseudorandom function f_s, is secure.

2. Concluding that the real scheme is secure (because, otherwise one could distinguish a pseudorandom function from a truly random one).

Note that we could have gotten rid of the randomization (in the encryption process) if we had allowed the encryption algorithm to be history dependent (e.g., use a counter in the role of r). This can be done if all parties that use the same key (for encryption) coordinate their encryption actions (by maintaining a joint state (e.g., counter)). Indeed, when using a private-key encryption scheme, a common situation is that the same key is only used for communication between two specific parties, which update a joint counter during their communication. Furthermore, if the encryption scheme is used for FIFO communication between the parties and both parties can reliably maintain the counter value, then there is no need (for the sender) to send the counter value. (The resulting scheme is related to "stream ciphers" that are commonly used in practice.)

We comment that the use of a counter (or any other state) in the encryption process is not reasonable in the case of public-key encryption schemes, because it is incompatible with the canonical usage of such schemes (i.e., allowing all parties to send encrypted messages to the "owner of the encryption-key" without engaging in any type of further coordination or communication). Furthermore (unlike in the case of private-key schemes), probabilistic encryption is essential for a secure public-key encryption scheme *even in the case of encrypting a single message.* Following Goldwasser and Micali [104], we now demonstrate the use of *probabilistic encryption* in the construction of a public-key encryption scheme.

Public-Key Encryption Scheme based on Trapdoor Permutations: We present two constructions that employ a collection of trapdoor permutations, as defined in Definition C.3. Let $\{f_i : D_i \to D_i\}_i$ be such a collection, and let b be a corresponding hard-core predicate. The key generation algorithm consists of selecting a permutation f_i along with a corresponding trapdoor t, and outputting (i, t) as the key-pair. To encrypt a (*single*) bit σ (using the encryption-key i), the encryption algorithm uniformly selects $r \in D_i$, and produces the ciphertext $(f_i(r), \sigma \oplus b(r))$. To decrypt the ciphertext (y, τ) (using the decryption-key t), the decryption algorithm computes $\tau \oplus b(f_i^{-1}(y))$ (using the trapdoor t of f_i). Clearly, $(\sigma \oplus b(r)) \oplus b(f_i^{-1}(f_i(r))) = \sigma$. Indistinguishability of encryptions can be easily proved using the fact that b is a hard-core of f_i. We comment that this scheme is quite wasteful in bandwidth; nevertheless, the paradigm underlying its construction (i.e., applying the trapdoor permutation to a randomized version of the plaintext rather than to the actual plaintext) is valuable in practice.

A more efficient construction of a public-key encryption scheme, which uses the same key-generation algorithm, follows. To encrypt an ℓ-bit long string x (using the encryption-key i), the encryption algorithm uniformly selects $r \in D_i$, computes $y \leftarrow b(r) \cdot b(f_i(r)) \cdots b(f_i^{\ell-1}(r))$ and produces the ciphertext $(f_i^{\ell}(r), x \oplus y)$. To decrypt the ciphertext (u, v) (using the decryption-key t), the decryption algorithm first recovers $r = f_i^{-\ell}(u)$ (using the trapdoor t of f_i), and then obtains $v \oplus b(r) \cdot b(f_i(r)) \cdots b(f_i^{\ell-1}(r))$. Note the similarity to the Blum-Micali Construction (depicted in Eq. (8.8)), and the fact that the proof of the pseudoradomness of Eq. (8.8) can be extended to establish the computational indistinguishability of $(b(r) \cdots b(f_i^{\ell-1}(r)), f_i^{\ell}(r))$ and $(r', f_i^{\ell}(r))$, for random and independent $r \in D_i$ and $r' \in \{0,1\}^{\ell}$. Indistinguishability of encryptions follows, and thus the second scheme is secure. We note that, assuming the intractability of factoring integers, this scheme has a concrete implementation with efficiency comparable to that of RSA.

C.5.3 Beyond Eavesdropping Security

Our treatment so far has referred only to a "passive" attack in which the adversary merely eavesdrops on the line over which ciphertexts are sent. Stronger types of attacks, culminating in the so-called Chosen Ciphertext Attack, may be possible in various applications. Specifically, in some settings it is feasible for the adversary to make the sender encrypt a message of the adversary's choice, and in some settings the adversary may even make the receiver decrypt a ciphertext of the adversary's choice. This gives rise to *chosen plaintext attacks* and to *chosen ciphertext attacks*, respectively, which are not covered by the security definitions considered in Sections C.5.1 and C.5.2. Here we briefly discuss such "active" attacks, focusing on chosen ciphertext attacks (of the strongest type known as "a posteriori" or "CCA2").

Loosely speaking, in a chosen ciphertext attack, the adversary may obtain the decryptions of ciphertexts of its choice, and is deemed successful if it learns something regarding the plaintext that corresponds to some different ciphertext (see [88, Sec. 5.4.4]). That is, the adversary is given oracle access to the decryption function corresponding to the decryption-key in use (and, in the case of private-key schemes, it is also given oracle access to the corresponding encryption function). The adversary is allowed to query the decryption oracle on any ciphertext except for the "test ciphertext" (i.e., the very ciphertext for which it tries to learn something about the corresponding plaintext). It may also make queries that do not correspond to legitimate ciphertexts, and the answer will be accordingly (i.e., a special 'failure' symbol). Furthermore, the adversary may effect the selection of the test ciphertext (by specifying a distribution from which the corresponding plaintext is to be drawn).

Private-key and public-key encryption schemes secure against chosen ciphertext attacks can be constructed under (almost) the same assumptions that suffice for the construction of the corresponding passive schemes. Specifically:

Theorem C.13 *Assuming the existence of* one-way functions, *there exist* private-key *encryption schemes that are secure against chosen ciphertext attack.*

Theorem C.14 *Assuming the existence of* enhanced[12] trapdoor permutations, *there exist* public-key *encryption schemes that are secure against chosen ciphertext attack.*

Both theorems are proved by constructing encryption schemes in which the adversary's gain from a chosen ciphertext attack is eliminated by making it infeasible (for the adversary) to obtain any useful knowledge via such an attack. In the case of private-key schemes (i.e., Theorem C.13), this is achieved by making it infeasible (for the adversary) to produce legitimate ciphertexts (other than those explicitly given to it, in response to its request to encrypt plaintexts of its choice). This, in turn, is achieved by augmenting the ciphertext with an "authentication tag" that is hard to generate without knowledge of the encryption-key; that is, we use a message-authentication scheme (as defined in Section C.6). In the case of public-key schemes (i.e., Theorem C.14), the adversary can certainly generate ciphertexts by itself, and the aim is to to make it infeasible (for the adversary) to produce legitimate ciphertexts without "knowing" the corresponding plaintext. This, in turn, will be achieved by augmenting the plaintext with a non-interactive zero-knowledge "proof of knowledge" of the corresponding plaintext.

Security against chosen ciphertext attack is related to the notion of *non-malleability* of the encryption scheme. Loosely speaking, in a non-malleable encryption scheme it is infeasible for an adversary, given a ciphertext, to produce a valid ciphertext

[12] Loosely speaking, the enhancement refers to the hardness condition of Definition C.2, and requires that it be hard to recover $f_i^{-1}(y)$ also when given the coins used to sample y (rather than merely y itself). See [88, Apdx. C.1].

for a related plaintext (e.g., given a ciphertext of a plaintext $1x$, for an unknown x, it is infeasible to produce a ciphertext to the plaintext $0x$). For further discussion see [88, Sec. 5.4.5].

C.6 Signatures and Message Authentication

Both signature schemes and message authentication schemes are methods for "validating" data; that is, verifying that the data was approved by a certain party (or set of parties). The difference between signature schemes and message authentication schemes is that signatures should be "universally verifiable", whereas authentication tags are only required to be verifiable by parties that are also able to generate them.

Signature Schemes: The need to discuss "digital signatures" (cf. [62, 169]) has arisen with the introduction of computer communication to the business environment (in which parties need to commit themselves to proposals and/or declarations that they make). Discussions of "unforgeable signatures" did take place also in previous centuries, but the objects of discussion were handwritten signatures (and not digital ones), and the discussion was not perceived as related to "cryptography". Loosely speaking, a scheme for unforgeable signatures should satisfy the following:

- each user can *efficiently produce its own signature* on documents of its choice;
- every user can *efficiently verify* whether a given string is a signature of another (specific) user on a specific document; but
- *it is infeasible to produce signatures of other users* to documents they did not sign.

We note that the formulation of unforgeable digital signatures provides also a clear statement of the essential ingredients of handwritten signatures. The ingredients are each person's ability to sign for itself, a universally agreed verification procedure, and the belief (or assertion) that it is infeasible (or at least hard) to forge signatures (i.e., produce some other person's signatures to documents that were not signed by it such that these "unauthentic" signatures are accepted by the verification procedure).

Message authentication schemes: Message authentication is a task related to the setting considered for encryption schemes; that is, communication over an insecure channel. This time, we consider an active adversary that is monitoring the channel and may alter the messages sent over it. The parties communicating through this insecure channel wish to authenticate the messages they send such that their counterpart can tell an original message (sent by the sender) from a modified one (i.e., modified by the adversary). Loosely speaking, a scheme for message authentication should satisfy the following:

- each of the communicating parties can *efficiently produce an authentication tag* to any message of its choice;
- each of the communicating parties can *efficiently verify* whether a given string is an authentication tag of a given message; but
- *it is infeasible for an external adversary* (i.e., a party other than the communicating parties) *to produce authentication tags* to messages not sent by the communicating parties.

Note that in contrast to the specification of signature schemes we do not require universal verification: only the designated receiver is required to be able to verify the authentication tags. Furthermore, we do not require that the receiver can not produce authentication tags by itself (i.e., we only require that *external parties* can not do so). Thus, message authentication schemes cannot convince *a third party* that the sender has indeed sent the information (rather than the receiver having generated it by itself). In contrast, signatures can be used to convince third parties: in fact, a signature to a document is typically sent to a second party so that in the future this party may (by merely presenting the signed document) convince third parties that the document was indeed generated (or sent or approved) by the signer.

C.6.1 Definitions

Formally speaking, both signature schemes and message authentication schemes consist of three efficient algorithms: key generation, signing and verification. As in the case of encryption schemes, the key-generation algorithm, denoted G, is used to generate a pair of corresponding keys, one is used for signing (via algorithm S) and the other is used for verification (via algorithm V). That is, $S_s(\alpha)$ denotes a signature produced by algorithm S on input a signing-key s and a document α, whereas $V_v(\alpha, \beta)$ denotes the verdict of the verification algorithm V regarding the document α and the alleged signature β relative to the verification-key v. Needless to say, for any pair of keys (s, v) generated by G and for every α, it holds that $V_v(\alpha, S_s(\alpha)) = 1$.

The difference between the two types of schemes is reflected in the definition of security. In the case of *signature schemes*, the adversary is given the verification-key, whereas in the case of *message authentication schemes* the verification-key (which may equal the signing-key) is not given to the adversary. Thus, schemes for message authentication can be viewed as a private-key version of signature schemes. This difference yields different functionalities (even more than in the case of encryption): In typical use of a signature scheme, each user generates a pair of signing and verification keys, publicizes the verification-key and keeps the signing-key secret. Subsequently, each user may sign documents using its own signing-key, and these signatures are *universally verifiable* with respect to its public verification-key. In contrast, message authentication schemes are typically used to authenticate information sent among a set of *mutually trusting* parties that agree on a secret key, which is being used both to produce and verify authentication-tags. (Indeed, it is assumed that the mutually trusting parties have generated the key together or have exchanged the key in a secure way, prior to the communication of information that needs to be authenticated.)

We focus on the definition of secure signature schemes, and consider very powerful attacks on the signature scheme as well as a very liberal notion of breaking it. Specifically, the attacker is allowed to obtain signatures to any message of its choice. One may argue that in many applications such a general attack is not possible (because messages to be signed must have a specific format). Yet, our view is that it is impossible to define a general (i.e., application-independent) notion of admissible messages, and thus a general/robust definition of an attack seems to have to be formulated as suggested here. (Note that at worst, our approach is overly cautious.) Likewise, the adversary is said to be successful if it can produce a valid signature to *any* message for which it has not asked for a signature during its attack. Again, this refers to the ability to form signatures to possibly "nonsensical" messages as a breaking of the scheme. Yet, again, we see no way to have a general (i.e., application-independent) notion of "meaningful" messages (so that only forging signatures to them will be considered a breaking of the scheme).

Definition C.15 (secure signature schemes – a sketch): *A* chosen message attack *is a process that, on input a verification-key, can obtain signatures* (relative to the corresponding signing-key) *to messages of its choice. Such an attack is said to* succeed (in existential forgery) *if it outputs a valid signature to a message for which it has* NOT *requested a signature during the attack. A signature scheme is* secure (or unforgeable) *if every* feasible *chosen message attack succeeds with at most negligible probability, where the probability is taken over the initial choice of the key-pair as well as over the adversary's actions.*

We stress that *plain* RSA (alike plain versions of Rabin's scheme [170] and the DSS) is not secure under the above definition. However, it may be secure if the message is "randomized" before RSA (or the other schemes) is applied.

C.6.2 Constructions

Secure *message authentication schemes* can be constructed using pseudorandom functions. Specifically, the key-generation algorithm consists of selecting a seed $s \in \{0,1\}^n$ for such a function, denoted $f_s:\{0,1\}^* \to \{0,1\}^n$, and the (only valid) tag of message x with respect to the key s is $f_s(x)$. As in the case of our private-key encryption scheme, the proof of security of the current message authentication scheme consists of two steps:

1. Proving that an idealized version of the scheme, in which one uses a uniformly selected function $F:\{0,1\}^* \to \{0,1\}^n$, rather than the pseudorandom function f_s, is secure (i.e., unforgeable).

2. Concluding that the real scheme is secure (because, otherwise one could distinguish a pseudorandom function from a truly random one).

Note that this message authentication scheme makes an "extensive use of pseudorandom functions" (i.e., the pseudorandom function is applied directly to the message, which requires a generalized notion of pseudorandom functions (cf. Section C.3.3)). More efficient schemes may be obtained either based on a more restricted use of a pseudorandom function or based on other cryptographic primitives.

Constructing secure *signature schemes* seems more difficult than constructing message authentication schemes. Nevertheless, secure signature schemes can be constructed based on the same assumptions.

Theorem C.16 *The following three conditions are equivalent:*

1. *One-way functions exist.*
2. *Secure signature schemes exist.*
3. *Secure message authentication schemes exist.*

We stress that, unlike in the case of public-key encryption schemes, the construction of signature schemes (which may be viewed as a public-key analogue of message authentication) does not require a trapdoor property.

How to construct secure signature schemes

Three central paradigms used in the construction of secure *signature schemes* are the "refreshing" of the "effective" signing-key, the usage of an "authentication tree", and the "hashing paradigm" (all to be discussed in the sequel). In addition to being used in the proof of Theorem C.16, all three paradigms are of independent interest.

The refreshing paradigm. Introduced in [106], the *refreshing paradigm* is aimed at limiting the potential dangers of chosen message attacks. This is achieved by signing the actual document using a newly (randomly) generated instance of the signature scheme, and authenticating (the verification-key of) this random instance with respect to the fixed public-key. That is, consider a basic signature scheme (G, S, V) used as follows. Suppose that the user U has generated a key-pair, $(s, v) \leftarrow G(1^n)$, and has placed the verification-key v on a public-file. When a party asks U to sign some document α, the user U generates a new ("fresh") key-pair, $(s', v') \leftarrow G(1^n)$, signs v' using the original signing-key s, signs α using the new signing-key s', and presents $(S_s(v'), v', S_{s'}(\alpha))$ as a signature to α. An alleged signature, (β_1, v', β_2), is verified by checking whether both $V_v(v', \beta_1) = 1$ and $V_{v'}(\alpha, \beta_2) = 1$ hold. Intuitively, the gain in terms of security is that a full-fledged chosen message attack cannot be launched on a fixed instance of (G, S, V) (i.e., on the fixed verification-key that resides in the public-file and is known to the attacker). All that an attacker may obtain (via a chosen message attack on the new scheme) is signatures, relative to the original signing-key s of (G, S, V), to random strings (distributed according to $G(1^n)$) as well as additional signatures that are each relative to a random and independently distributed signing-key.

Authentication trees. The security benefits of the refreshing paradigm are increased when combining it with the use of *authentication trees*. The idea is to use the public verification-key in order to authenticate several (e.g., two) fresh instances of the signature scheme, use each of these instances to authenticate several additional fresh instances, and so on. We obtain a tree of fresh instances of the basic signature scheme, where each internal node authenticates its children. We can now use the leaves of this tree in order to sign actual documents, where each leaf is used at most once. Thus, a signature to an actual document consists of (1) a signature to this document authenticated with respect to the verification-key associated with some leaf, and (2) a sequence of verification-keys associated with the nodes along the path from the root to this leaf, where each such verification-key is authenticated with respect to the verification-key of its parent. We stress that (by suitable implementation to be discussed below) each instance of the signature scheme is used to sign at most one string (i.e., a single sequence of verification-keys if the instance resides in an internal node, and an actual document if the instance resides in a leaf). Thus, it suffices to use a signature scheme that is secure as long as it is used to legitimately sign a *single* string. Such signature schemes, called one-time signature schemes, are easier to construct than standard signature schemes, especially if one only wishes to sign strings that are significantly shorter than the signing-key (resp., than the verification-key). For example, using a one-way function f, we may let the signing-key consist of a sequence of n pairs of strings, let the corresponding verification-key consist of the corresponding sequence of images of f, and sign an n-bit long message by revealing the adequate pre-images.[13]

The hashing paradigm. Note, however, that in the foregoing authentication-tree, the instances of the signature scheme (associated with internal nodes) are used to sign a pair of verification-keys. Thus, we need a one-time signature scheme that can be used for signing messages that are longer than the verification-key. Here is where the *hashing paradigm* comes into play. This paradigm refers to the common practice of signing documents via a two stage process: First the actual document is hashed to a (relatively) short string, and next the basic signature scheme is applied to the resulting string. This practice (as well as other usages of the hashing paradigm) is sound provided that the hashing function belongs to a family of *collision-free hashing* (a.k.a *collision-resistant hashing*) functions. Loosely speaking, given a hash function that is randomly selected in such a family, it is infeasible to find two different strings that are hashed by this function to the same value. We also refer the interested reader to a variant of the *hashing paradigm* that uses the seemingly weaker notion of a family of *Universal One-Way Hash Functions* (see [159] or [88, Sec. 6.4.3]).

Implementation details. In order to implement the foregoing (full-fledged) signature scheme one needs to store in (secure) memory all the instances of the basic

[13]That is, the signing-key consist of a sequence $((s_1^0, s_1^1), ..., (s_n^0, s_n^1)) \in \{0,1\}^{2n^2}$, the corresponding verification-key is $(f(s_1^0), f(s_1^1)), ..., (f(s_n^0), f(s_n^1)))$, and the signature of the message $\sigma_1 \cdots \sigma_n$ is $(s_1^{\sigma_1}, ..., s_n^{\sigma_n})$.

(one-time) signature scheme that are generated throughout the entire signing process (which refers to numerous documents). This can be done by extending the model so to allow for memory-dependent signature schemes. Alternatively, we note that all that we need to store are the random-coins used for generating each of these instances, and the former can be determined by a pseudorandom function (applied to the name of the corresponding vertex in the tree). Indeed, the seed of this pseudorandom function will be part of the signing-key of the resulting (full-fledged) signature scheme.

C.6.3 Public-Key Infrastructure

The standard use of public-key encryption schemes (resp., signature schemes) in real-life communication requires a mechanism for providing the sender (resp., signature verifier) with the receiver's authentic encryption-key (resp., signer's authentic verification-key). Specifically, this problem arises in large-scale systems, where typically the sender (resp., verifier) does not have a local record of the receiver's encryption-key (resp., signer's verification-key), and so must obtain this key in a "reliable" way (i.e., typically, certified by some trusted authority). In most theoretical works, one assumes that the keys can be retrieved from a public-file, which is maintained by a trusted party that makes sure that each user can post only keys bearing its own identity. In practice, maintaining such a public-file is a major problem, and mechanisms that implement this abstraction are typically referred to by the generic term "public-key infrastructure" (PKI).

C.7 General Cryptographic Protocols

The design of secure protocols that implement arbitrary desired functionalities is a major part of modern cryptography. Taking the opposite perspective, the design of any cryptographic scheme may be viewed as the design of a secure protocol for implementing a corresponding functionality. Still, we believe that it makes sense to differentiate between basic cryptographic primitives (which involve little interaction) like encryption and signature schemes on one hand, and general cryptographic protocols on the other hand.

We survey *general* results concerning secure *multi*-party computations, where the *two*-party case is an important special case. In a nutshell, these results assert that one can construct protocols for securely computing *any* desirable multi-party functionality. Indeed, what is striking about these results is their generality, and we believe that the wonder is not diminished by the (various alternative) conditions under which these results hold.

Our focus on the *general* study of secure multi-party computation (rather than on protocols for solving specific problems) is natural in the context of the theoretical treatment of the subject matter. We wish to highlight the importance of this *general* study to practice. Firstly, this study clarifies fundamental issues regarding security in a multi-party environment. Secondly, it draws the lines between what is possible in principle and what is not. Thirdly, it develops general techniques for designing secure protocols. And last, sometimes, it may even yield schemes (or modules) that may be incorporated in practical systems.

A general framework for casting (m-party) cryptographic (protocol) problems consists of specifying a random process[14] that maps m inputs to m outputs. The inputs to the process are to be thought of as the local inputs of m parties, and the m outputs are their corresponding local outputs. The random process describes the desired functionality. That is, if the m parties were to trust each other (or trust some external party), then they could each send their local input to the trusted party, who would compute the outcome of the process and send to each party the corresponding output. A pivotal question in the area of cryptographic protocols is to what extent can this (imaginary) trusted party be "emulated" by the mutually distrustful parties themselves.

The results surveyed in this section describe a variety of models in which such an "emulation" is possible. The models vary by the underlying assumptions regarding the communication channels, numerous parameters governing the extent of adversarial behavior, and the desired level of emulation of the trusted party (i.e., level of "security"). All these refers to the security of stand-alone executions, and the preservation of security in an environment in which many executions of many protocols are attacked is beyond the scope of this section.[15]

C.7.1 The Definitional Approach and Some Models

Before describing the aforementioned results, we further discuss the notion of "emulating a trusted party", which underlies the definitional approach to secure multi-party computation. The approach can be traced back to the definition of zero-knowledge and even to the definition of secure encryption. The underlying paradigm (called the simulation paradigm (cf. Section C.4.1)) is that a scheme is secure if whatever a feasible adversary can obtain after attacking it, is also feasibly attainable "from scratch". In the case of zero-knowledge this amounts to saying that whatever a (feasible) verifier can obtain after interacting with the prover on a prescribed valid assertion, can be (feasibly) computed from the assertion itself. In the case of multi-party computation we compare the effect of adversaries that participate in the execution of the actual protocol to the effect of adversaries that participate in an imaginary execution of a trivial (ideal) protocol for computing the desired functionality with the help of a trusted party. If whatever the adversaries can feasibly obtain in the real setting can also be feasibly obtained in the ideal setting then the actual protocol "emulates the ideal setting" (i.e., "emulates a trusted party"), and so is deemed secure. This basic approach can be applied in

[14] That is, we consider the secure evaluation of randomized functionalities, rather than "only" the secure evaluation of functions. Specifically, we consider an arbitrary (randomized) process F that on input $(x_1, ..., x_m)$, first selects at random (depending only on $\ell \stackrel{\rm def}{=} \sum_{i=1}^m |x_i|$) an m-ary function f, and then outputs the m-tuple $f(x_1, ..., x_m) = (f_1(x_1, ..., x_m), ..., f_m(x_1, ..., x_m))$. In other words, $F(x_1, ..., x_m) = F'(r, x_1, ..., x_m)$, where r is uniformly selected in $\{0,1\}^{\ell'}$ (with $\ell' = {\rm poly}(\ell)$), and F' is a function mapping $(m+1)$-long sequences to m-long sequences.

[15] The interested reader is referred to [88, Sec. 7.7.2].

a variety of models, and is used to define the goals of security in these models.[16] We first discuss some of the parameters used in defining various models, and next demonstrate the application of this approach in two important cases. For further details, see [88, Sec. 7.2 and 7.5.1].

C.7.1.1 Some parameters used in defining security models

The following parameters are described in terms of the actual (or real) computation. In *some cases*, the corresponding definition of security is obtained by imposing some restrictions or provisions on the ideal model. For example, in the case of two-party computation (see §C.7.1.3), secure computation is possible only if premature termination is *not* considered a breach of security. In that case, the suitable security definition is obtained (via the simulation paradigm) by allowing (an analogue of) premature termination in the ideal model. In *all cases*, the desired notion of security is defined by requiring that for any adequate adversary in the real model, there exist a corresponding adversary in the corresponding ideal model that obtains essentially the same impact (as the real-model adversary).

The communication channels: The standard assumption in cryptography is that the adversary may tap all communication channels (between honest parties), but cannot modify (or omit or insert) messages sent over them. In contrast, one may *postulate* that the adversary cannot obtain messages sent between a pair of honest parties, yielding the so-called private-channel model. Most works in the area assume that communication is *synchronous* and that point-to-point channels exist between every pair of processors (i.e., a *complete network*).

Set-up assumptions: Unless stated differently, we make no set-up assumptions (except for the obvious assumption that all parties have identical copies of the protocol's program).

Computational limitations: Typically, we consider computationally-bounded adversaries (e.g., probabilistic polynomial-time adversaries). However, the private-channel model allows for the (meaningful) consideration of computationally-unbounded adversaries.[17]

[16] A few technical comments are in place. Firstly, we assume that the inputs of all parties are of the same length. We comment that as long as the lengths of the inputs are polynomially related, the foregoing convention can be enforced by padding. On the other hand, some length restriction is essential for the security results, because in general it is impossible to hide all information regarding the length of the inputs to a protocol. Secondly, we assume that the desired functionality is computable in probabilistic polynomial-time, because we wish the secure protocol to run in probabilistic polynomial-time (and a protocol cannot be more efficient than the corresponding centralized algorithm). Clearly, the results can be extended to functionalities that are computable within any given (time-constructible) time bound, using adequate padding.

[17] We stress that, also in the case of computationally-unbounded adversaries, security should be defined by requiring that for every real adversary, whatever the adversary can compute after participating in the execution of the actual protocol is computable *within comparable time* by an imaginary adversary participating in an imaginary execution of the trivial ideal protocol (for computing the desired functionality with the help of a trusted party). That is, although no

Restricted adversarial behavior: The parameters of the model include questions like whether the adversary is "active" or "passive" (i.e., whether a dishonest party takes active steps to disrupt the execution of the protocol or merely gathers information) and whether or not the adversary is "adaptive" (i.e., whether the set of dishonest parties is fixed before the execution starts or is adaptively chosen by an adversary during the execution).

Restricted notions of security: One important example is the willingness to tolerate "unfair" protocols in which the execution can be suspended (at any time) by a dishonest party, provided that it is detected doing so. We stress that in case the execution is suspended, the dishonest party does not obtain more information than it could have obtained when not suspending the execution. (What may happen is that the honest parties will not obtain their desired outputs, but will detect that the execution was suspended.) We stress that the motivation to this restricted model is the impossibility of obtaining general secure two-party computation in the unrestricted model.

Upper bounds on the number of dishonest parties: In some models, secure multi-party computation is possible only if a majority of the parties is honest.

C.7.1.2 Example: Multi-party protocols with honest majority

Here we consider an active, non-adaptive, computationally-bounded adversary, and do not assume the existence of private channels. Our aim is to define multi-party protocols that remain secure provided that the honest parties are in majority. (The reason for requiring an honest majority will be discussed at the end of this subsection.)

Consider any multi-party protocol. We first observe that each party may change its local input before even entering the execution of the protocol. However, this is unavoidable also when the parties utilize a trusted party. Consequently, such an effect of the adversary on the real execution (i.e., modification of its own input prior to entering the actual execution) is not considered a breach of security. In general, whatever cannot be avoided when the parties utilize a trusted party, is not considered a breach of security. We wish secure protocols (in the real model) to suffer only from whatever is unavoidable also when the parties utilize a trusted party. Thus, the basic paradigm underlying the definitions of *secure multi-party computations* amounts to requiring that the only situations that may occur in the real execution of a secure protocol are those that can also occur in a corresponding ideal model (where the parties may employ a trusted party). In other words, the "effective malfunctioning" of parties in secure protocols is restricted to what is postulated in the corresponding ideal model.

When defining secure multi-party protocols with honest majority, we need to pin-point what cannot be avoided in the ideal model (i.e., when the parties utilize a trusted party). Since we are interested in executions in which the majority of

computational restrictions are made on the real-model adversary, it is required that the ideal-model adversary that obtains the same impact does so within comparable time (i.e., within time that is polynomially related to the running time of the real-model adversary being simulated).

parties are honest, we consider an ideal model in which any minority group (of the parties) may collude as follows:

1. Firstly this dishonest minority shares its original inputs and decides together on replaced inputs to be sent to the trusted party. (The other parties send their respective original inputs to the trusted party.)

2. Upon receiving inputs from all parties, the trusted party determines the corresponding outputs and sends them to the corresponding parties. (We stress that the information sent between the honest parties and the trusted party is not seen by the dishonest colluding minority.)

3. Upon receiving the output-message from the trusted party, each honest party outputs it locally, whereas the dishonest colluding minority may determine their outputs based on all they know (i.e., their initial inputs and their received outputs).

Note that the foregoing behavior of the minority group is unavoidable in any execution of any protocol (even in presence of trusted parties). This is the reason that the ideal model was defined as above. Now, a *secure multi-party computation with honest majority* is required to emulate this ideal model. That is, the effect of any feasible adversary that controls a minority of the parties in a real execution of the actual protocol, can be essentially simulated by a (different) feasible adversary that controls the corresponding parties in the ideal model.

Definition C.17 (secure protocols – a sketch): *Let f be an m-ary functionality and Π be an m-party protocol operating in the real model.*

- *For a real-model adversary A, controlling some minority of the parties* (and tapping all communication channels)*, and an m-sequence $\overline{x}$, we denote by $\text{REAL}_{\Pi,A}(\overline{x})$ the sequence of m outputs resulting from the execution of Π on input $\overline{x}$ under the attack of the adversary A.*

- *For an ideal-model adversary A', controlling some minority of the parties, and an m-sequence $\overline{x}$, we denote by $\text{IDEAL}_{f,A'}(\overline{x})$ the sequence of m outputs resulting from the foregoing three-step ideal process, when applied to input $\overline{x}$ under the attack of the adversary A'.*

We say that Π securely implements f with honest majority *if for every feasible real-model adversary A, controlling some minority of the parties, there exists a feasible ideal-model adversary A', controlling the same parties, such that the probability ensembles $\{\text{REAL}_{\Pi,A}(\overline{x})\}_{\overline{x}}$ and $\{\text{IDEAL}_{f,A'}(\overline{x})\}_{\overline{x}}$ are computationally indistinguishable (as in Definition C.5).*

Thus, security means that the effect of each minority group in a real execution of a secure protocol is "essentially restricted" to replacing its own local inputs (independently of the local inputs of the majority parties) before the protocol starts, and replacing its own local outputs (depending only on its local inputs and outputs) after the protocol terminates. (We stress that in the real execution the minority parties do obtain additional pieces of information; yet in a secure protocol they gain nothing from these additional pieces of information, because they can actually reproduce those by themselves.)

The fact that Definition C.17 refers to a model without private channels is reflected in the fact that our (sketchy) definition of the real-model adversary allowed it to tap the channels, which in turn effects the set of possible ensembles $\{\text{REAL}_{\Pi,A}(\overline{x})\}_{\overline{x}}$. When defining security in the private-channel model, the real-model adversary is not allowed to tap channels between honest parties, and this again effects the possible ensembles $\{\text{REAL}_{\Pi,A}(\overline{x})\}_{\overline{x}}$. On the other hand, when we wish to define security with respect to passive adversaries, both the scope of the real-model adversaries and the scope of the ideal-model adversaries changes. In the real-model execution, all parties follow the protocol but the adversary may alter the output of the dishonest parties arbitrarily depending on their intermediate internal states (during the entire execution). In the corresponding ideal-model, the adversary is not allowed to modify the *inputs* of dishonest parties (in Step 1), but is allowed to modify their outputs (in Step 3).

We comment that a definition analogous to Definition C.17 can be presented also in the case that the dishonest parties are not in minority. In fact, such a definition seems more natural, but the problem is that such a definition cannot be satisfied. That is, most (natural) functionalities do not have a protocol for computing them securely in the case that at least half of the parties are dishonest and employ an adequate adversarial strategy. This follows from an impossibility result regarding two-party computation, which essentially asserts that there is no way to prevent a party from prematurely suspending the execution. On the other hand, secure multi-party computation with dishonest majority is possible if premature suspension of the execution is not considered a breach of security (see §C.7.1.3).

C.7.1.3 Another example: Two-party protocols allowing abort

In light of the last paragraph, we now consider multi-party computations in which premature suspension of the execution is not considered a breach of security. For simplicity, we focus on the special case of two-party computations.[18]

Intuitively, in any two-party protocol, each party may suspend the execution at any point in time, and furthermore it may do so as soon as it learns the desired output. Thus, in case the output of each parties depends on both inputs, it is always possible for one of the parties to obtain the desired output while preventing the other party from fully determining its own output.[19] The same phenomenon occurs even in case the two parties just wish to generate a common random value. Thus, when considering active adversaries in the two-party setting, we do not consider such premature suspension of the execution a breach of security. Consequently, we consider an ideal model where each of the two parties may "shut-down" the trusted (third) party at any point in time. In particular, this may happen after the trusted

[18] As in §C.7.1.2, we consider a non-adaptive, active, computationally-bounded adversary.

[19] In the case of an honest majorty (cf., §C.7.1.2), the honest party that fails to obtain its output is not alone. It may seek help from the other honest parties, which together and being in majority can do things that dishonest minorities cannot do: See §C.7.3.2.

party has supplied the outcome of the computation to one party but before it has supplied it to the other. That is, an execution in the corresponding ideal model proceeds as follows:

1. Each party sends its input to the trusted party, where the dishonest party may replace its input or send no input at all (which can be treated as sending a default value).
2. Upon receiving inputs from both parties, the trusted party determines the corresponding outputs, and sends the first output to the first party.
3. In case the first party is dishonest, it may instruct the trusted party to halt, otherwise it always instructs the trusted party to proceed. If instructed to proceed, the trusted party sends the second output to the second party.
4. Upon receiving the output-message from the trusted party, the honest party outputs it locally, whereas the dishonest party may determine its output based on all it knows (i.e., its initial input and its received output).

A secure two-party computation allowing abort is required to emulate this ideal model. That is, as in Definition C.17, security is defined by requiring that for every feasible real-model adversary A, there exists a feasible ideal-model adversary A', controlling the same party, such that the probability ensembles representing the corresponding (real and ideal) executions are computationally indistinguishable. This means that each party's "effective malfunctioning" in a secure protocol is restricted to supplying an initial input of its choice and aborting the computation at any point in time. (Needless to say, the choice of the initial input of each party may *not* depend on the input of the other party.)

We mention that an alternative way of dealing with the problem of premature suspension of execution (i.e., abort) is to restrict our attention to single-output functionalities; that is, functionalities in which only one party is supposed to obtain an output. The definition of secure computation of such functionalities can be made identical to Definition C.17, with the exception that no restriction is made on the set of dishonest parties (and in particular one may consider a single dishonest party in the case of two-party protocols). For further details, see [88, Sec. 7.2.3].

C.7.2 Some Known Results

We next list some of the models for which general secure multi-party computation is known to be attainable (i.e., models in which one can construct secure multi-party protocols for computing any desired functionality). We mention that the first results of this type were obtained by Goldreich, Micali, Wigderson and Yao [96, 223, 97].

In the standard channel model. *Assuming the existence of enhanced*[20] *trapdoor permutations*, secure multi-party computation is possible in the following three models (cf. [96, 223, 97] and details in [88]):

[20] See Footnote 12.

1. Passive adversary, for any number of dishonest parties (see [88, Sec. 7.3]).
2. Active adversary that may control only a minority of the parties (see [88, Sec. 7.5.4]).
3. Active adversary, for any number of bad parties, provided that suspension of execution is not considered a violation of security (i.e., as discussed in §C.7.1.3). (See [88, Sec. 7.4 and 7.5.5].)

In all these cases, the adversary is computationally-bounded and non-adaptive. On the other hand, the adversary may tap the communication lines between honest parties (i.e., we do not assume "private channels" here). The results for active adversaries assume a broadcast channel. Indeed, the latter can be implemented (while tolerating any number of bad parties) using a signature scheme and assuming a public-key infrastructure (or that each party knows the verification-key corresponding to each of the other parties).

In the private channels model. Making no computational assumptions and allowing computationally-unbounded adversaries, but *assuming private channels*, secure multi-party computation is possible in the following two models (cf. [32, 50]):

1. Passive adversary that may control only a minority of the parties.
2. Active adversary that may control only less than one third of the parties.

In both cases the adversary may be adaptive.

Theory versus practice (or general versus specific): Recall that this appendix is focused on presenting general notions and general feasibility results. Needless to say, practical solutions to specific problems (e.g., voting, secure payment systems, and threshold cryptosystems) are typically derived by direct constructions (and not by applying general results of the foregoing type). Still, these general results are of great importance to practice because they characterize a wide class of security problems that are solvable in principle, and provide techniques that may be useful also towards constructing reasonable solutions to specific problems.

C.7.3 Construction Paradigms and Two Simple Protocols

We briefly sketch a couple of paradigms used in the construction of secure multi-party protocols. We focus on the construction of secure protocols for the model of computationally-bounded and non-adaptive adversaries [96, 223, 97]. These constructions proceed in two steps (see details in [88]). First a secure protocol is presented for the model of passive adversaries (for any number of dishonest parties), and next such a protocol is "compiled" into a protocol that is secure in one of the two models of active adversaries (i.e., either in a model allowing the adversary to control only a minority of the parties or in a model in which premature suspension of the execution is not considered a violation of security). These two steps are presented in the following two corresponding subsections, in which we also present two relatively simple protocols for two specific tasks, which in turn are used extensively in the general protocols.

Recall that in the model of passive adversaries, all parties follow the prescribed protocol, but at termination the adversary may alter the outputs of the dishonest parties depending on their intermediate internal states (during the entire execution). Below, we refer to protocols that are secure in the model of passive (resp., active) adversaries by the term passively-secure (resp., actively-secure).

C.7.3.1 Passively-secure computation with shares

For any $m \geq 2$, suppose that m parties, each having a private input, wish to obtain the value of a predetermined m-argument function evaluated at their sequence of inputs. Below, we outline a passively-secure protocol for achieving this goal. We mention that the design of passively-secure multi-party protocol for any functionality (allowing different outputs to different parties as well as handling also randomized computations) reduces easily to the aforementioned task.

We assume that the parties hold a circuit for computing the value of the function on inputs of the adequate length, and that the circuit contains only `and` and `not` gates. The key idea is having each party "secretly share" its input with everybody else, and having the parties "secretly transform" shares of the input wires of the circuit into shares of the output wires of the circuit, thus obtaining shares of the outputs (which allows for the reconstruction of the actual outputs). The value of each wire in the circuit is shared such that all shares yield the value, whereas lacking even one of the shares keeps the value totally undetermined. That is, we use a simple secret sharing scheme such that a bit b is shared by a random sequence of m bits that sum-up to $b \bmod 2$. First, each party shares each of its input bits with all parties (by secretly sending each party a random value and setting its own share accordingly). Next, all parties jointly scan the circuit from its input wires to its output wires, processing each gate as follows:

- When encountering a gate, the parties already hold shares of the values of the wires entering the gate, and their aim is to obtain shares of the value of the wires exiting the gate.
- For a `not`-gate this is easy: the first party just flips the value of its share, and all other parties maintain their shares.
- Since an `and`-gate corresponds to multiplication modulo 2, the parties need to securely compute the following randomized functionality (in which the x_i's denote shares of one entry-wire, the y_i's denote shares of the second entry-wire, the z_i's denote shares of the exit-wire, and the shares indexed by i belongs to Party i):

$$((x_1, y_1), ..., (x_m, y_m)) \quad \mapsto \quad (z_1, ..., z_m)\,, \text{ where} \tag{C.1}$$

$$\sum_{i=1}^{m} z_i = \left(\sum_{i=1}^{m} x_i\right) \cdot \left(\sum_{i=1}^{m} y_i\right). \tag{C.2}$$

  That is, the z_i's are random subject to Eq. (C.2).

Finally, the parties send their shares of each circuit-output wire to the designated party, which reconstructs the value of the corresponding bit. Thus, the parties have propagated shares of the circuit-input wires into shares of the ocircuit-utput wires, by repeatedly conducting privately-secure computation of the m-ary functionality of Eq. (C.1) & (C.2). That is, securely evaluating the entire (arbitrary) circuit "reduces" to securely conducting a specific (very simple) multi-party computation. But things get even simpler: the key observation is that

$$\left(\sum_{i=1}^{m} x_i\right) \cdot \left(\sum_{i=1}^{m} y_i\right) = \sum_{i=1}^{m} x_i y_i + \sum_{1\le i<j\le m} (x_i y_j + x_j y_i) \qquad \text{(C.3)}$$

Thus, the m-ary functionality of Eq. (C.1) & (C.2) can be computed as follows (where all arithmetic operations are mod 2):

1. Each Party i locally computes $z_{i,i} \stackrel{\text{def}}{=} x_i y_i$.

2. Next, each pair of parties (i.e., Parties i and j) securely compute random shares of $x_i y_j + y_i x_j$. That is, Parties i and j (holding (x_i, y_i) and (x_j, y_j), respectively), need to securely compute the randomized two-party functionality $((x_i, y_i), (x_j, y_j)) \mapsto (z_{i,j}, z_{j,i})$, where the z's are random subject to $z_{i,j} + z_{j,i} = x_i y_j + y_i x_j$. Equivalently, Party j uniformly selects $z_{j,i} \in \{0,1\}$, and Parties i and j securely compute the deterministic functionality $((x_i, y_i), (x_j, y_j, z_{j,i})) \mapsto (z_{j,i} + x_i y_j + y_i x_j, \lambda)$, where λ denotes the empty string.

   The latter simple two-party computation can be securely implemented using a 1-out-of-4 Oblivious Transfer (see below), which in turn can be implemented using enhanced trapdoor permutations (see further below). Loosely speaking, a 1-out-of-k Oblivious Transfer is a protocol enabling one party to obtain one of k secrets held by another party, without the second party learning which secret was obtained by the first party. That is, we refer to the two-party functionality

   $$(i, (s_1, ..., s_k)) \mapsto (s_i, \lambda). \qquad \text{(C.4)}$$

   Note that any function $f : [k] \times \{0,1\}^* \to \{0,1\}^*$ can be privately-computed by invoking a 1-out-of-k Oblivious Transfer on inputs i and $(f(1, y), ..., f(k, y))$, where i (resp., y) is the initial input of the first (resp., second) party.

3. Finally, for every $i = 1, ..., m$, summing-up all the $z_{i,j}$'s yields the desired share of Party i.

The above construction is analogous to a construction that was briefly described in [97]. A detailed description and full proofs appear in [88, Sec. 7.3.4 and 7.5.2].

A passively-secure 1-out-of-k Oblivious Transfer. Using a collection of enhanced trapdoor permutations, $\{f_\alpha : D_\alpha \to D_\alpha\}_{\alpha \in \overline{I}}$ and a corresponding hard-core predicate b, we outline a passively-secure implementation of the functionality of Eq. (C.4).

Inputs: The sender has input $(\sigma_1, \sigma_2, ..., \sigma_k) \in \{0,1\}^k$, the receiver has input $i \in \{1, 2, ..., k\}$.

Step S1: The sender selects at random a permutation f_α along with a corresponding trapdoor, denoted t, and sends the permutation f_α (i.e., its index α) to the receiver.

Step R1: The receiver uniformly and independently selects $x_1, ..., x_k \in D_\alpha$, sets $y_i = f_\alpha(x_i)$ and $y_j = x_j$ for every $j \neq i$, and sends $(y_1, y_2, ..., y_k)$ to the sender.

Thus, the receiver knows $f_\alpha^{-1}(y_i) = x_i$, but cannot predict $b(f_\alpha^{-1}(y_j))$ for any $j \neq i$. Needless to say, the last assertion presumes that the receiver follows the protocol (i.e., is semi-honest).

Step S2: Upon receiving $(y_1, y_2, ..., y_k)$, using the inverting-with-trapdoor algorithm and the trapdoor t, the sender computes $z_j = f_\alpha^{-1}(y_j)$, for every $j \in \{1, ..., k\}$. It sends the k-tuple $(\sigma_1 \oplus b(z_1), \sigma_2 \oplus b(z_2), ..., \sigma_k \oplus b(z_k))$ to the receiver.

Step R2: Upon receiving $(c_1, c_2, ..., c_k)$, the receiver locally outputs $c_i \oplus b(x_i)$.

We first observe that this protocol correctly computes 1-out-of-k Oblivious Transfer; that is, the receiver's local output (i.e., $c_i \oplus b(x_i)$) indeed equals $(\sigma_i \oplus b(f_\alpha^{-1}(f_\alpha(x_i)))) \oplus b(x_i) = \sigma_i$. Next, we offer some intuition as to why this protocol constitutes a privately-secure implementation of 1-out-of-k Oblivious Transfer. Intuitively, the sender gets no information from the execution because, for any possible value of i, the senders sees the same distribution; specifically, a sequence of k uniformly and independently distributed elements of D_α. (Indeed, the key observation is that applying f_α to a uniformly distributed element of D_α yields a uniformly distributed element of D_α.) As for the receiver, intuitively, it gains no computational knowledge from the execution because, for $j \neq i$, the only information that the receiver has regarding σ_j is the triplet $(\alpha, x_j, \sigma_j \oplus b(f_\alpha^{-1}(x_j)))$, where x_j is uniformly distributed in D_α, and from this information it is infeasible to predict σ_j better than by a random guess.[21] (See [88, Sec. 7.3.2] for a detailed proof of security.)

C.7.3.2 From passively-secure protocols to actively-secure ones

We show how to transform any passively-secure protocol into a corresponding actively-secure protocol. The communication model in both protocols consists of a single broadcast channel. Note that the messages of the original protocol may be assumed to be sent over a broadcast channel, because the adversary may see them anyhow (by tapping the point-to-point channels), and because a broadcast

[21] The latter intuition presumes that sampling D_α is trivial (i.e., that there is an easily computable correspondence between the coins used for sampling and the resulting sample), whereas in general the coins used for sampling may be hard to compute from the corresponding outcome (which is the reason that an enhanced hardness assumption is used in the general analysis of the foregoing protocol).

channel is trivially implementable in the case of passive adversaries. As for the resulting actively-secure protocol, the broadcast channel it uses can be implemented via an (authenticated) Byzantine Agreement protocol, thus providing an emulation of this model on the standard point-to-point model (in which a broadcast channel does not exist). We mention that authenticated Byzantine Agreement is typically implemented using a signature scheme (and assuming that each party knows the verification-key corresponding to each of the other parties).

Turning to the transformation itself, the main idea is using zero-knowledge proofs (as described in Section C.4.3) in order to force parties to behave in a way that is consistent with the (passively-secure) protocol. Actually, we need to confine each party to a unique consistent behavior (i.e., according to some fixed local input and a sequence of coin tosses), and to guarantee that a party cannot fix its input (and/or its coins) in a way that depends on the inputs (and/or coins) of honest parties. Thus, some preliminary steps have to be taken before the step-by-step emulation of the original protocol may start. Specifically, the compiled protocol (which like the original protocol is executed over a broadcast channel) proceeds as follows:

1. *Committing to the local input*: Prior to the emulation of the original protocol, each party commits to its input (using a commitment scheme as defined in Section C.4.3). In addition, using a zero-knowledge proof-of-knowledge, each party also proves that it knows its own input; that is, it proves that it can decommit to the commitment it sent. (These zero-knowledge proof-of-knowledge prevent dishonest parties from setting their inputs in a way that depends on inputs of honest parties.)

2. *Generation of local random tapes*: Next, all parties jointly generate a sequence of random bits for each party such that only this party knows the outcome of the random sequence generated for it, and everybody else gets a commitment to this outcome. These sequences will be used as the random-inputs (i.e., sequence of coin tosses) for the original protocol. Each bit in the random-sequence generated for Party X is determined as the exclusive-or of the outcomes of instances of an (augmented) coin-tossing protocol (cf. [88, Sec. 7.4.3.5]) that Party X plays with each of the other parties. The latter protocol provides the other parties with a commitment to the outcome obtained by Party X.

3. *Effective prevention of premature termination*: In addition, when compiling (the passively-secure protocol to an actively-secure protocol) *for the model that allows the adversary to control only a minority of the parties*, each party shares its input and random-input with all other parties using a "Verifiable Secret Sharing" (VSS) protocol (cf. [88, Sec. 7.5.5.1]). Loosely speaking, a VSS protocol allows sharing a secret in a way that enables each participant to verify that the share it got fits the publicly posted information, which includes commitments to all shares. The use of VSS guarantees that if Party X prematurely suspends the execution, then the honest parties can together reconstruct all Party X's secrets and carry on the execution while playing its

role. This step effectively prevents premature termination, and is not needed in a model that does not consider premature termination a breach of security.

4. *Step-by-step emulation of the original protocol*: After all the foregoing steps are completed, the new protocol emulates the steps of the original protocol. In each step, each party augments the message determined by the original protocol with a zero-knowledge proof that asserts that the message was indeed computed correctly. Recall that the next message (as determined by the original protocol) is a function of the sender's own input, its random-input, and the messages it has received so far (where the latter are known to everybody because they were sent over a broadcast channel). Furthermore, the sender's input is determined by its commitment (as sent in Step 1), and its random-input is similarly determined (in Step 2). Thus, the next message (as determined by the original protocol) is a function of publicly known strings (i.e., the said commitments as well as the other messages sent over the broadcast channel). Moreover, the assertion that the next message was indeed computed correctly is an NP-assertion, and the sender knows a corresponding NP-witness (i.e., its own input and random-input as well as the corresponding decommitment information). Thus, the sender can prove in zero-knowledge (to each of the other parties) that the message it is sending was indeed computed according to the original protocol.

The above compilation was first outlined in [96, 97]. A detailed description and full proofs appear in [88, Sec. 7.4 and 7.5].

A secure coin-tossing protocol. Using a commitment scheme, we outline a secure (ordinary as opposed to augmented) coin-tossing protocol.

Step C1: Party 1 uniformly selects $\sigma \in \{0,1\}$ and sends Party 2 a commitment, denoted c, to σ.

Step C2: Party 2 uniformly selects $\sigma' \in \{0,1\}$, and sends σ' to Party 1.

Step C3: Party 1 outputs the value $\sigma \oplus \sigma'$, and sends σ along with the decommitment information, denoted d, to Party 2.

Step C4: Party 2 checks whether or not (σ, d) fit the commitment c it has obtained in Step 1. It outputs $\sigma \oplus \sigma'$ if the check is satisfied and halts with output $\perp$ otherwise (indicating that Party 1 has effectively aborted the protocol prematurely).

Outputs: Party 1 always outputs $b \stackrel{\rm def}{=} \sigma \oplus \sigma'$, whereas Party 2 either outputs b or $\perp$.

Intuitively, Steps C1–C2 may be viewed as "tossing a coin into the well". At this point (i.e., after Step C2) the value of the coin is determined (essentially as a random value), but only one party (i.e., Party 1) "can see" (i.e., knows) this value. Clearly, if both parties are honest then they both output the same uniformly chosen bit, recovered in Steps C3 and C4, respectively. Intuitively, each party can guarantee that the outcome is uniformly distributed, and Party 1 can cause premature termination by improper execution of Step 3. Formally, we have to show how the effect of every real-model adversary can be simulated by an adequate ideal-model adversary (which is allowed premature termination). This is done in [88, Sec. 7.4.3.1].

C.7.4 Concluding Remarks

In Sections C.7.1-C.7.2 we have mentioned numerous definitions and results regarding secure multi-party protocols, where some of these definitions are incomparable to others (i.e., they neither imply the others nor are implies by them). For example, in §C.7.1.2 and §C.7.1.3, we have presented two alternative definitions of "secure multi-party protocols", one requiring an honest majority and the other allowing abort. These definitions are incomparable and there is no generic reason to prefer one over the other. Actually, as mentioned in §C.7.1.2, one could formulate a natural definition that implies both definitions (i.e., waiving the bound on the number of dishonest parties in Definition C.17). Indeed, the resulting definition is free of the annoying restrictions that were introduced in each of the two aforementioned definitions; the "only" problem with the resulting definition is that it cannot be satisfied (in general). Thus, for the first time in this appendix, we have reached a situation in which a natural (and general) definition cannot be satisfied, and we are forced to choose between two weaker alternatives, where each of these alternatives carries fundamental disadvantages.

In general, Section C.7 carries a stronger flavor of compromise (i.e., recognizing inherent limitations and settling for a restricted meaningful goal) than previous sections. In contrast to the impression given in other parts of this appendix, it is now obvious that we cannot get all that we may want (and this is without mentioning the problems involved in preserving security under concurrent composition; cf. [88, Sec. 7.7.2]). Instead, we should study the alternatives, and go for the one that best suits our real needs.

Indeed, as stated in Section C.1, the fact that we can define a cryptographic goal does not mean that we can satisfy it as defined. In case we cannot satisfy the initial definition, we should search for relaxations that can be satisfied. These relaxations should be defined in a clear manner such that it would be obvious what they achieve (and what they fail to achieve). Doing so will allow a sound choice of the relaxation to be used in a specific application. This seems to be a good point to end the current appendix.

A good compromise is one in which he most important interests of all parties are satisfied.

Adv. Klara Goldreich-Ingwer (1912–2004)

Appendix D

Probabilistic Preliminaries and Advanced Topics in Randomization

What is this? Chicken Quesadilla and Seafood Salad?
Fine, but in the same plate? This is disgusting!

Johan Håstad at Grendel's, Cambridge (1985)

Summary: This appendix lumps together some preliminaries regarding probability theory and some advanced topics related to the role and use of randomness in computation. Needless to say, each of these appears in a separate section.

The probabilistic preliminaries include our conventions regarding random variables, which are used throughout the book. Also included are overviews of three useful inequalities: Markov Inequality, Chebyshev's Inequality, and Chernoff Bound.

The advanced topics include hashing, sampling and randomness extraction. For hashing, we describe constructions of pairwise (and t-wise independent) hashing functions, and variants of the Leftover Hashing Lemma (which are used a couple of times in the main text). We then review the "complexity of sampling": that is, the number of samples and the randomness complexity involved in estimating the average value of an arbitrary function defined over a huge domain. Finally, we provide an overview on the question of extracting almost perfect randomness from sources of weak (or defected) randomness.

D.1 Probabilistic preliminaries

Probability plays a central role in complexity theory (see, for example, Chapters 6–9). We assume that the reader is familiar with the basic notions of probability theory. In this section, we merely present the probabilistic notations that are used throughout the book, and three useful probabilistic inequalities.

D.1.1 Notational Conventions

Throughout the entire book we will refer only to *discrete* probability distributions. Specifically, the underlying probability space will consist of the set of all strings of a certain length ℓ, taken with uniform probability distribution. That is, the sample space is the set of all ℓ-bit long strings, and each such string is assigned probability measure $2^{-\ell}$. Traditionally, *random variables* are defined as functions from the sample space to the reals. Abusing the traditional terminology, we use the term random variable also when referring to functions mapping the sample space into the set of binary strings. We often do not specify the probability space, but rather talk directly about random variables. For example, we may say that X is a random variable assigned values in the set of all strings such that $\mathsf{Pr}[X=00] = \frac{1}{4}$ and $\mathsf{Pr}[X=111] = \frac{3}{4}$. (Such a random variable may be defined over the sample space $\{0,1\}^2$, so that $X(11) = 00$ and $X(00) = X(01) = X(10) = 111$.) One important case of a random variable is the output of a randomized process (e.g., a probabilistic polynomial-time algorithm as in Section 6.1).

All our probabilistic statements refer to functions of random variables that are defined beforehand. Typically, we may write $\mathsf{Pr}[f(X)=1]$, where X is a random variable defined beforehand (and f is a function). An important convention is that *all occurrences of the same symbol in a probabilistic statement refer to the same* (unique) *random variable*. Hence, if $B(\cdot,\cdot)$ is a Boolean expression depending on two variables, and X is a random variable then $\mathsf{Pr}[B(X,X)]$ denotes the probability that $B(x,x)$ holds when x is chosen with probability $\mathsf{Pr}[X=x]$. For example, for every random variable X, we have $\mathsf{Pr}[X=X] = 1$. We stress that if we wish to discuss the probability that $B(x,y)$ holds when x and y are chosen independently with identical probability distribution, then we will define *two* independent random variables each with the same probability distribution. Hence, if X and Y are two independent random variables then $\mathsf{Pr}[B(X,Y)]$ denotes the probability that $B(x,y)$ holds when the pair (x,y) is chosen with probability $\mathsf{Pr}[X=x] \cdot \mathsf{Pr}[Y=y]$. For example, for every two independent random variables, X and Y, we have $\mathsf{Pr}[X=Y] = 1$ only if both X and Y are trivial (i.e., assign the entire probability mass to a single string).

Throughout the entire book, U_n denotes a random variable uniformly distributed over the set of strings of length n. Namely, $\mathsf{Pr}[U_n = \alpha]$ equals 2^{-n} if $\alpha \in \{0,1\}^n$ and equals 0 otherwise. We will often refer to the distribution of U_n as the uniform distribution (neglecting to qualify that it is uniform over $\{0,1\}^n$). In addition, we will occasionally use random variables (arbitrarily) distributed over $\{0,1\}^n$ or $\{0,1\}^{\ell(n)}$, for some function $\ell:\mathbb{N}\rightarrow\mathbb{N}$. Such random variables are typically denoted by X_n, Y_n, Z_n, etc. We stress that in some cases X_n is distributed over $\{0,1\}^n$, whereas in other cases it is distributed over $\{0,1\}^{\ell(n)}$, for some function $\ell(\cdot)$, which is typically a polynomial. We will often talk about probability ensembles, which are infinite sequence of random variables $\{X_n\}_{n\in\mathbb{N}}$ such that each X_n ranges over strings of length bounded by a polynomial in n.

Statistical difference. The statistical distance (a.k.a variation distance) between the random variables X and Y is defined as

$$\frac{1}{2} \cdot \sum_v |\mathsf{Pr}[X = v] - \mathsf{Pr}[Y = v]| \;=\; \max_S\{\mathsf{Pr}[X \in S] - \mathsf{Pr}[Y \in S]\}.$$

We say that X is δ-close (resp., δ-far) to Y if the statistical distance between them is at least (resp., at most) δ.

D.1.2 Three Inequalities

The following probabilistic inequalities are very useful. All inequalities refer to random variables that are assigned real values and provide upper-bounds on the probability that the random variable deviates from its expectation.

Markov Inequality. The most basic inequality is Markov Inequality that applies to any random variable with bounded maximum or minimum value. For simplicity, it is stated for random variables that are lower-bounded by zero, and reads: *Let X be a non-negative random variable and v a real number. Then*

$$\mathsf{Pr}\left[X \geq v\right] \leq \frac{\mathsf{E}(X)}{v} \tag{D.1}$$

Equivalently, $\mathsf{Pr}[X \geq r \cdot \mathsf{E}(X)] \leq \frac{1}{r}$. The proof amounts to the following sequence.

$$\begin{aligned}
\mathsf{E}(X) &= \sum_x \mathsf{Pr}[X = x] \cdot x \\
&\geq \sum_{x<v} \mathsf{Pr}[X = x] \cdot 0 + \sum_{x\geq v} \mathsf{Pr}[X = x] \cdot v \\
&= \mathsf{Pr}[X \geq v] \cdot v
\end{aligned}$$

Chebyshev's Inequality: Using Markov's inequality, one gets a potentially stronger bound on the deviation of a random variable from its expectation. This bound, called Chebyshev's inequality, is useful when having additional information concerning the random variable (specifically a good upper bound on its variance). For a random variable X of finite expectation, we denote by $\mathsf{Var}(X) \stackrel{\text{def}}{=} \mathsf{E}[(X - \mathsf{E}(X))^2]$ the variance of X, and observe that $\mathsf{Var}(X) = \mathsf{E}(X^2) - \mathsf{E}(X)^2$. Chebyshev's inequality then reads as follows: *Let X be a random variable, and $\delta > 0$. Then*

$$\mathsf{Pr}\left[|X - \mathsf{E}(X)| \geq \delta\right] \;\leq\; \frac{\mathsf{Var}(X)}{\delta^2} \tag{D.2}$$

Proof: We define a random variable $Y \stackrel{\rm def}{=} (X - \mathsf{E}(X))^2$, and apply Markov inequality. We get

$$\begin{aligned} \mathsf{Pr}\left[|X - \mathsf{E}(X)| \geq \delta\right] &= \mathsf{Pr}\left[(X - \mathsf{E}(X))^2 \geq \delta^2\right] \\ &\leq \frac{\mathsf{E}[(X - \mathsf{E}(X))^2]}{\delta^2} \end{aligned}$$

and the claim follows. ■

Corollary (Pairwise Independent Sampling)**:** Chebyshev's inequality is particularly useful in the analysis of the error probability of approximation via repeated sampling. It suffices to assume that the samples are picked in a pairwise independent manner, where $X_1, X_2, ..., X_n$ are pairwise independent if for every $i \neq j$ and every α, β it holds that $\mathsf{Pr}[X_i = \alpha \wedge X_j = \beta] = \mathsf{Pr}[X_i = \alpha] \cdot \mathsf{Pr}[X_j = \beta]$. The corollary reads: *Let $X_1, X_2, ..., X_n$ be pairwise independent random variables with identical expectation, denoted μ, and identical variance, denoted σ^2. Then, for every $\varepsilon > 0$,*

$$\mathsf{Pr}\left[\left|\frac{\sum_{i=1}^n X_i}{n} - \mu\right| \geq \varepsilon\right] \leq \frac{\sigma^2}{\varepsilon^2 n} \tag{D.3}$$

Proof: Define the random variables $\overline{X}_i \stackrel{\rm def}{=} X_i - \mathsf{E}(X_i)$. Note that the $\overline{X}_i$'s are pairwise independent, and each has zero expectation. Applying Chebyshev's inequality to the random variable $\sum_{i=1}^n \frac{X_i}{n}$, and using the linearity of the expectation operator, we get

$$\begin{aligned} \mathsf{Pr}\left[\left|\sum_{i=1}^n \frac{X_i}{n} - \mu\right| \geq \varepsilon\right] &\leq \frac{\mathsf{Var}\left[\sum_{i=1}^n \frac{X_i}{n}\right]}{\varepsilon^2} \\ &= \frac{\mathsf{E}\left[\left(\sum_{i=1}^n \overline{X}_i\right)^2\right]}{\varepsilon^2 \cdot n^2} \end{aligned}$$

Now (again using the linearity of expectation)

$$\mathsf{E}\left[\left(\sum_{i=1}^n \overline{X}_i\right)^2\right] = \sum_{i=1}^n \mathsf{E}\left[\overline{X}_i^2\right] + \sum_{1 \leq i \neq j \leq n} \mathsf{E}\left[\overline{X}_i \overline{X}_j\right]$$

By the pairwise independence of the $\overline{X}_i$'s, we get $\mathsf{E}[\overline{X}_i \overline{X}_j] = \mathsf{E}[\overline{X}_i] \cdot \mathsf{E}[\overline{X}_j]$, and using $\mathsf{E}[\overline{X}_i] = 0$, we get

$$\mathsf{E}\left[\left(\sum_{i=1}^n \overline{X}_i\right)^2\right] = n \cdot \sigma^2$$

The corollary follows. ■

Chernoff Bound: Using pairwise independent sampling, the error probability in the approximation is decreasing linearly with the number of sample points (see Eq. (D.3)). Using totally independent sampling points, the error probability in the approximation can be shown to decrease exponentially with the number of sample points. (The random variables $X_1, X_2, ..., X_n$ are said to be totally independent if for every sequence $a_1, a_2, ..., a_n$ it holds that $\mathsf{Pr}[\wedge_{i=1}^n X_i = a_i] = \prod_{i=1}^n \mathsf{Pr}[X_i = a_i]$.) Probability bounds, supporting the above statement, are given next. The first bound, commonly referred to as Chernoff Bound, concerns 0-1 random variables (i.e., random variables that are assigned as values either 0 or 1), and asserts the following. *Let $p \leq \frac{1}{2}$, and $X_1, X_2, ..., X_n$ be independent 0-1 random variables so that $\mathsf{Pr}[X_i = 1] = p$, for each i. Then for all ε, $0 < \varepsilon \leq p(1-p)$, we have*

$$\mathsf{Pr}\left[\left|\frac{\sum_{i=1}^n X_i}{n} - p\right| > \varepsilon\right] < 2 \cdot e^{-\frac{\varepsilon^2}{2p(1-p)} \cdot n} \tag{D.4}$$

Proof Sketch: We upper-bound $\mathsf{Pr}[\sum_{i=1}^n X_i - pn > \varepsilon n]$ and $\mathsf{Pr}[pn - \sum_{i=1}^n X_i > \varepsilon n]$ is bounded similarly. Letting $\overline{X}_i \stackrel{\text{def}}{=} X_i - \mathsf{E}(X_i)$, we apply Markov Inequality to the random variable $e^{\lambda \sum_{i=1}^n \overline{X}_i}$, where $\lambda > 0$ is determined to optimize the expressions that we derive (hint: $\lambda = \theta(\varepsilon/p(1-p))$ will do). Thus, $\mathsf{Pr}[\sum_{i=1}^n \overline{X}_i > \varepsilon n]$ is upper-bounded by

$$\frac{\mathsf{E}[e^{\lambda \sum_{i=1}^n \overline{X}_i}]}{e^{\lambda \varepsilon n}} = e^{-\lambda \varepsilon n} \cdot \prod_{i=1}^n \mathsf{E}[e^{\lambda \overline{X}_i}]$$

where the equality is due to the independence of the random variables. To simplify the rest of the proof, we establish a sub-optimal bound as follows. Using a Taylor expansion of e^x (e.g., $e^x < 1 + x + 2x^2$) and observing that $\mathsf{E}[\overline{X}_i] = 0$, we get $\mathsf{E}[e^{\lambda \overline{X}_i}] < 1 + 2\lambda^2 \mathsf{E}[\overline{X}_i^2]$, which equals $1 + 2\lambda^2 p(1-p)$. Thus, $\mathsf{Pr}[\sum_{i=1}^n X_i - pn > \varepsilon n]$ is upper-bounded by $(1 + 2\lambda^2 p(1-p))^n < \exp(-\lambda \varepsilon n + 2\lambda^2 p(1-p)n)$, which is optimized at $\lambda = \varepsilon/(4p(1-p))$ yielding $\exp(-\frac{\varepsilon^2}{8p(1-p)} \cdot n)$. Needless to say, this method can be applied in more general settings (e.g., for $X_i \in [0,1]$ rather than $X_i \in \{0,1\}$). $\square$

A more general bound, useful in the approximations of the expectation of a general random variable (not necessarily 0-1), is given next (and is commonly referred to as Hoefding Inequality).[1] *Let $X_1, X_2, ..., X_n$ be n independent random variables with identical probability distribution, each ranging over the (real) interval $[a, b]$, and let μ denote the expected value of each of these variables. Then, for every $\varepsilon > 0$,*

$$\mathsf{Pr}\left[\left|\frac{\sum_{i=1}^n X_i}{n} - \mu\right| > \varepsilon\right] < 2 \cdot e^{-\frac{2\varepsilon^2}{(b-a)^2} \cdot n} \tag{D.5}$$

Hoefding Inequality is useful in estimating the average value of a function defined over a large set of values, especially when the desired error probability needs to

[1] A more general form requires the X_i's to be independent, but not necessarily identical, and uses $\mu \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^n \mathsf{E}(X_i)$. See [10, Apdx. A].

be negligible (i.e., decrease faster than any polynomial in the relevant parameter). Such an estimate can be obtained provided that we can efficiently sample the set and have a bound on the possible values (of the function).

Pairwise independent vs totally independent sampling. Consider, for simplicity the case that $a = 0 < \mu < b = 1$. In this case, n independent samples give an approximation that deviates by ε from the expectation with probability, denoted δ, that is exponentially decreasing with $\varepsilon^2 n$. Such an approximation is called an (ε, δ)-approximation, and can be achieved using $n = O(\varepsilon^{-2} \cdot \log(1/\delta))$ sample points. Thus, the number of sample points is polynomially related to ε^{-1} and logarithmically related to δ^{-1}. It follows that, when using poly(n) many samples,[2] the error probability (i.e., δ) can be made negligible (as a function in n), but the accuracy of the estimation (i.e., ε) can be only bounded above by any fixed polynomial fraction (and cannot be made negligible). We stress that the dependency of the number of samples on ε is not better than in the case of pairwise independent sampling; the advantage of totally independent samples is only in the dependency of the number of samples on δ.

D.2 Hashing

Hashing is extensively used in complexity theory in order to map arbitrary (unstructured) sets "almost uniformly" to a smaller structured set of adequate size. Specifically, hashing is supposed to map an arbitrary 2^m-subset (of $\{0,1\}^n$) to $\{0,1\}^m$ in an "almost uniform" manner.

For a fixed set S of cardinality 2^m, a 1-1 mapping $f_S : S \to \{0,1\}^m$ does exist, but it is not necessarily an efficient one (e.g., it may require "knowing" the entire set S). Clearly, no fixed function $f : \{0,1\}^n \to \{0,1\}^m$ can map every 2^m subset of $\{0,1\}^n$ to $\{0,1\}^m$ in a 1-1 manner (or even approximately so). However, a random function $f : \{0,1\}^n \to \{0,1\}^m$ has the property that, for every 2^m-subset $S \subset \{0,1\}^n$, with overwhelmingly high probability f maps S to $\{0,1\}^m$ such that no point in the range has many f-preimages in S. The problem is that a truly random function is unlikely to have a succinct representation (let alone an efficient evaluation algorithm). We seek families of functions that have a similar property, but do have a succinct representation as well as an efficient evaluation algorithm.

D.2.1 Definitions

Motivated by the foregoing discussion, we consider families of functions $\{H_n^m\}_{m<n}$ Such that the following properties hold:

1. For every $S \subset \{0,1\}^n$, with high probability, a function h selected uniformly in H_n^m maps S to $\{0,1\}^m$ in an "almost uniform" manner. For example, for any $|S| = 2^m$ and each point y, with high probability over the choice of h, it holds that $|\{x \in S : h(x) = y\}| \leq \text{poly}(n)$.

[2]Here and in the rest of the book, we denote by poly() some fixed but unspecified polynomial.

2. The functions in H_n^m have succinct representation. For example, we may require that $H_n^m \equiv \{0,1\}^{\ell(n,m)}$, for some polynomial ℓ.

3. The functions in H_n^m can be efficiently evaluated. That is, there exists a polynomial-time algorithm that, on input a representation of a function, h (in H_n^m), and a string $x \in \{0,1\}^n$, returns $h(x)$. In some cases we make even more stringent requirements regarding the the algorithm (e.g., that it runs in linear space).

Condition 1 was left vague on purpose. At the very least, we require that the expected size of $\{x \in S : h(x) = y\}$ equals $|S|/2^m$. We shall see (in Section D.2.3) that different (stronger) interpretations of Condition 1 are satisfied by different types of hashing functions. We focus on t-wise independent hashing functions, defined next.

Definition D.1 (t-wise independent hashing functions): *A family H_n^m of functions from n-bit strings to m-bit strings is called t-wise independent if for every t distinct domain elements $x_1, ..., x_t \in \{0,1\}^n$ and every $y_1, ..., y_t \in \{0,1\}^m$ it holds that*

$$\mathsf{Pr}_{h \in H_n^m}[\wedge_{i=1}^t h(x_i) = y_1] \;=\; 2^{-t \cdot m}$$

That is, every t domain elements are mapped by a uniformly chosen $h \in H_n^m$ in a totally uniform manner. Note that for $t \geq 2$, it follows that the probability that a random $h \in H_n^m$ maps two distinct domain elements to the same image is 2^{-m}. Such (families of) functions are called universal (cf. [47]), but we will focus on the stronger condition of t-wise independence.

D.2.2 Constructions

The following constructions are merely a re-interpretation of the constructions presented in §8.6.1.1. (Alternatively, one may view the constructions presented in §8.6.1.1 as a re-interpretation of the following two constructions.)

Construction D.2 (t-wise independent hashing): *For $t, m, n \in \mathbb{N}$ such that $m \leq n$, consider the following family of hashing functions mapping n-bit strings to m-bit strings. Each t-sequence $\overline{s} = (s_0, s_1, ..., s_{t-1}) \in \{0,1\}^{t \cdot n}$ describes a function $h_{\overline{s}} : \{0,1\}^n \to \{0,1\}^m$ such that $h_{\overline{s}}(x)$ equals the m-bit prefix of the binary representation of $\sum_{j=0}^{t-1} s_j x^j$, where the arithmetic is that of* GF(2^n)*, the finite field of 2^n elements.*

Proposition 8.24 implies that Construction D.2 constitutes a family of t-wise independent hash functions. Typically, we will use either $t = 2$ or $t = \Theta(n)$. To make the construction totally explicit, we need an explicit representation of GF(2^n); see details following Proposition 8.24. An alternative construction for the case of $t = 2$ may be obtained analogously to the pairwise independent generator of Proposition 8.25. Recall that a Toeplitz matrix is a matrix with all diagonals being homogeneous; that is, $T = (t_{i,j})$ is a Toeplitz matrix if $t_{i,j} = t_{i+1,j+1}$, for all i, j.

Construction D.3 (Alternative pairwise independent hashing): *For $m \leq n$, consider the family of hashing functions in which each n-by-m Toeplitz matrix T and an m-dimensional vector b describes a function $h_{T,b} : \{0,1\}^n \to \{0,1\}^m$ such that $h_{T,b}(x) = Tx + b$.*

Proposition 8.25 implies that Construction D.3 constitutes a family of pairwise independent hash functions. Note that a n-by-m Toeplitz matrix can be specified by $n+m-1$ bits, yielding description length $n+2m-1$. An alternative construction (analogous to Eq. (8.17) and requiring $m \cdot n+m$ bits of representation) uses arbitrary n-by-m matrices rather than Toeplitz matrices.

D.2.3 The Leftover Hash Lemma

We now turn to the "almost uniform" cover condition (i.e., Condition 1) mentioned in Section D.2.1. One concrete interpretation of this condition is implied by the following lemma.

Lemma D.4 *Let $m < n$ be integers, H_n^m be a family of pairwise independent hash functions, and $S \subseteq \{0,1\}^n$. Then, for every $y \in \{0,1\}^m$ and every $\varepsilon > 0$, for all but at most an $\frac{2^m}{\varepsilon^2|S|}$ fraction of $h \in H_n^m$ it holds that*

$$|\{x \in S : h(x) = y\}| = (1 \pm \varepsilon) \cdot \frac{|S|}{2^m}. \tag{D.6}$$

By pairwise independence (or rather even by "1-wise independence"), the expected size of $\{x \in S : h(x) = y\}$ is $|S|/2^m$, where the expectation is taken uniformly over all $h \in H_n^m$. The lemma upper bounds the fraction of h's that deviate from the expected value. Needless to say, the bound is meaningful only in case $|S| > 2^m$ (or alternatively for $\varepsilon > 1$). Setting $\varepsilon = \sqrt[3]{2^m/|S|}$ (and focusing on the case that $|S| > 2^m$), we infer that *for all but at most an ε fraction of $h \in H_n^m$ it holds that* $|\{x \in S : h(x) = y\}| = (1 \pm \varepsilon) \cdot |S|/2^m$. Thus, each range element has approximately the right number of h-preimages in the set S under almost all $h \in H_n^m$.

Proof: Fixing an arbitrary set $S \subseteq \{0,1\}^n$ and an arbitrary $y \in \{0,1\}^m$, we estimate the probability that a uniformly selected $h \in H_n^m$ violates Eq. (D.6). We define random variables ζ_x, over the aforementioned probability space, such that $\zeta_x = \zeta_x(h)$ equal 1 if $h(x) = y$ and 0 otherwise. The expected value of $\sum_{x \in S} \zeta_x$ is $\mu \stackrel{\text{def}}{=} |S| \cdot 2^{-m}$, and we are interested in the probability that this sum deviates from the expectation. Applying Chebyshev's Inequality, we get

$$\mathsf{Pr}\left[\left|\mu - \sum_{x \in S} \zeta_x\right| > \varepsilon \cdot \mu\right] \; < \; \frac{\mu}{\varepsilon^2 \mu^2}$$

because $\mathsf{Var}(\sum_{x \in S} \zeta_x) < |S| \cdot 2^{-m}$ by the pairwise independence of the ζ_x's and the fact that $\mathsf{E}[\zeta_x] = 2^{-m}$. The lemma follows. ∎

A generalization (called mixing). The proof of Lemma D.4 can be easily extended to show that *for every set $T \subset \{0,1\}^m$ and every $\varepsilon > 0$, for all but at most an $\frac{2^m}{|T|\cdot|S|\varepsilon^2}$ fraction of $h \in H_n^m$ it holds that $|\{x \in S : h(x) = y\}| = (1 \pm \varepsilon) \cdot |T| \cdot |S|/2^m$.* (Hint: just define $\zeta_x = \zeta(h) = 1$ if $h(x) \in T$ and 0 otherwise.) In the case that $m = n$, this is called a mixing property, and is meaningfull provided $|T| \cdot |S| > 2^m/\varepsilon$.

An extremely useful corollary. The aforementioned generalization of Lemma D.4 asserts that most functions behave well with respect to any fixed sets of preimages $S \subset \{0,1\}^n$ and images $T \subset \{0,1\}^m$. A seemingly stronger statement, which is (non-trivially) implied by Lemma D.4, is that for all adequate sets S most functions $h \in H_n^m$ map S to $\{0,1\}^m$ in an almost uniform manner.[3] This is a consequence of the following theorem.

Theorem D.5 (a.k.a Leftover Hash Lemma): *Let H_n^m and $S \subseteq \{0,1\}^n$ be as in Lemma D.4, and define $\varepsilon = \sqrt[3]{2^m/|S|}$. Consider random variable X and H that are uniformly distributed on S and H_n^m, respectively. Then, the statistical distance between $(H, H(X))$ and (H, U_m) is at most 2ε.*

Using the terminology of Section D.4, we say that H_n^m yields a strong extractor (with parameters to be spelled out there).

Proof: Let V denote the set of pairs (h, y) that violate Eq. (D.6), and $\overline{V} \stackrel{\rm def}{=} (H_n^m \times \{0,1\}^m) \setminus V$. Then for every $(h, y) \in \overline{V}$ it holds that

$$\begin{aligned} \mathsf{Pr}[(H, H(X)) = (h, y)] &= \mathsf{Pr}[H = h] \cdot \mathsf{Pr}[h(X) = y] \\ &= (1 \pm \varepsilon) \cdot \mathsf{Pr}[(H, U_m) = (h, y)]. \end{aligned}$$

On the other hand, by Lemma D.4 (which asserts $\mathsf{Pr}[(H, y) \in V] \leq \varepsilon$ for every $y \in \{0,1\}^m$), we have $\mathsf{Pr}[(H, U_m) \in V] \leq \varepsilon$. Using

$$\begin{aligned} \mathsf{Pr}[(H, H(X)) \in V] &= 1 - \mathsf{Pr}[(H, H(X)) \in \overline{V}] \\ &\leq 1 - \mathsf{Pr}[(H, U_n)) \in \overline{V}] + \varepsilon \;\leq\; 2\varepsilon \end{aligned}$$

we upper-bounded the statistical difference between $(H, H(X))$ and (H, U_m) by

$$\begin{aligned} &\frac{1}{2} \cdot \sum_{(h,y) \in H_n^m \times \{0,1\}^m} |\mathsf{Pr}[(H, H(X)) = (h, y)] - \mathsf{Pr}[(H, U_m) = (h, y)]| \\ &\leq \frac{\varepsilon}{2} + \frac{1}{2} \cdot \sum_{(h,y) \in V} |\mathsf{Pr}[(H, H(X)) = (h, y)] - \mathsf{Pr}[(H, U_m) = (h, y)]| \\ &\leq \frac{\varepsilon}{2} + \frac{1}{2} \cdot \sum_{(h,y) \in V} (\mathsf{Pr}[(H, H(X)) = (h, y)] + \mathsf{Pr}[(H, U_m) = (h, y)]) \\ &\leq \frac{\varepsilon}{2} + \frac{1}{2} \cdot (2\varepsilon + \varepsilon) \end{aligned}$$

[3]That is, for X as in Theorem D.5 and any $\alpha > 0$, for all but at most an α fraction of the functions $h \in H_n^m$ it holds that $h(X)$ is $(2\varepsilon/\alpha)$-close to U_m.

and the claim follows. ■

An alternative proof of Theorem D.5. Define the collision probability of a random variable Z, denote $\mathsf{cp}(Z)$, as the probability that two independent samples of Z yield the same result. Alternatively, $\mathsf{cp}(Z) \stackrel{\rm def}{=} \sum_z \Pr[Z = z]^2$. Theorem D.5 follows by combining the following two facts:

1. A general fact: *If $Z \in [N]$ and $\mathsf{cp}(Z) \le (1 + 4\epsilon^2)/N$ then Z is ϵ-close to the uniform distribution on $[N]$.*

   We prove the contra-positive: Assuming that the statistical distance between Z and the uniform distribution on $[N]$ equals δ, we show that $\mathsf{cp}(Z) \ge (1+4\delta^2)/N$. This is done by defining $L \stackrel{\rm def}{=} \{z : \Pr[Z = z] < 1/N\}$, and lower-bounding $\mathsf{cp}(Z)$ by using the fact that the collision probability minimizes on uniform distributions. Specifically,

   $$\mathsf{cp}(Z) \ge |L| \cdot \left(\frac{\Pr[Z \in L]}{|L|}\right)^2 + (N - |L|) \cdot \left(\frac{\Pr[Z \in [N] \setminus L]}{N - |L|}\right)^2,$$

   which equals $1 + (\delta^2/(1 - \rho)\rho) \ge 1 + 4\delta^2$, where $\rho = |L|/N$.

2. The collision probability of $(H, H(X))$ is at most $(1 + (2^m/|S|))/(|H_n^m| \cdot 2^m)$. (Furthermore, this holds even if H_n^m is only universal.)

   The proof is by a straightforward calculation. Specifically, note that $\mathsf{cp}(H, H(X)) = |H_n^m|^{-1} \cdot \mathsf{E}_{h\in H_n^m}[\mathsf{cp}(h(X))]$, whereas $\mathsf{E}_{h\in H_n^m}[\mathsf{cp}(h(X)] = |S|^{-2} \sum_{x_1,x_2\in S} \Pr[H(x_1) = H(x_2)]$. The sum equals $|S| + (|S|^2 - |S|) \cdot 2^{-m}$, and so $\mathsf{cp}(H, H(X)) < |H_n^m|^{-1} \cdot (2^{-m} + |S|^{-1})$.

Note that it follows that $(H, H(X))$ is $\sqrt{2^m/4|S|}$-close to (H, U_m), which is a stronger bound than the one provided in Theorem D.5.

Stronger uniformity via higher independence. Recall that Lemma D.4 asserts that for each point in the range of the hash function, with high probability over the choice of the hash function, *this fixed point* has approximately the expected number of preimages in S. A stronger condition asserts that, with high probability over the choice of the hash function, *every point in its range* has approximately the expected number of preimages in S. Such a guarantee can be obtained when using n-wise independent hashing functions.

Lemma D.6 *Let $m < n$ be integers, H_n^m be a family of n-wise independent hash functions, and $S \subseteq \{0,1\}^n$. Then, for every $\varepsilon \in (0,1)$, for all but at most an $2^m \cdot (n \cdot 2^m/\varepsilon^2|S|)^{n/2}$ fraction of $h \in H_n^m$, it holds that $|\{x \in S : h(x) = y\}| = (1 \pm \varepsilon) \cdot |S|/2^m$ for every $y \in \{0,1\}^m$.*

Indeed, the lemma should be used with $2^m < \varepsilon^2|S|/4n$. In particular, using $m = \log_2 |S| - \log_2(5n/\varepsilon^2)$ guarantees that with high probability each range elements has

$(1 \pm \varepsilon) \cdot |S|/2^m$ preimages in S. Under this setting of parameters $|S|/2^m = 5n/\varepsilon^2$, which is $\text{poly}(n)$ whenever $\varepsilon = 1/\text{poly}(n)$. Needless to say, this guarantee is stronger than the conclusion of Theorem D.5.

Proof: The proof follows the footsteps of the proof of Lemma D.4, taking advantage of the fact that the random variables (i.e., the ζ_x's) are now $2t$-wise independent, where $t = n/2$. This allows for the use of a so-called $2t^{\text{th}}$ moment analysis, which generalizes the analysis of pairwise independent samplying (presented in Section D.1.2). As in the proof of Lemma D.4, we fix any S and y, and define $\zeta_x = \zeta_x(h) = 1$ if and only if $h(x) = y$. Letting $\mu = \mathsf{E}[\sum_{x \in S} \zeta_x] = |S|/2^m$ and $\overline{\zeta}_x = \zeta_x - \mathsf{E}(\zeta_x)$, we start with Markov inequality:

$$\begin{aligned} \Pr\left[\left|\mu - \sum_{x \in S} \zeta_x\right| > \varepsilon \cdot \mu\right] &< \frac{\mathsf{E}[(\sum_{x \in S} \overline{\zeta}_x)^{2t}]}{\varepsilon^{2t}\mu^{2t}} \\ &= \frac{\sum_{x_1,\ldots,x_{2t} \in S} \mathsf{E}[\prod_{i=1}^{2t} \overline{\zeta}_{x_i}]}{\varepsilon^{2t} \cdot (|S|/2^m)^{2t}} \end{aligned} \tag{D.7}$$

Using $2t$-wise independence, we note that only the terms in Eq. (D.7) that do not vanish are those in which each variable appears with multiplicity. This mean that only terms having less than t distinct variables contribute to Eq. (D.7). Now, for every $j \leq t$, we have less than $\binom{|S|}{j} \cdot (2t!) < (2t!/j!) \cdot |S|^j$ terms with j distinct variables, and each contributes less than $(2^{-m})^j$ to the sum. Thus, Eq. (D.7) is upper-bounded by

$$\frac{2t!}{(\varepsilon^{2t}|S|/2^m)^{2t}} \cdot \sum_{j=1}^{t} \frac{(|S|/2^m)^j}{j!} < \frac{2t!/t!}{(\varepsilon^2|S|/2^m)^t} < \left(\frac{2t \cdot 2^m}{\varepsilon^2|S|}\right)^t$$

where the first inequality assumes $|S| > n2^m$ (since the claim hold vacuously otherwise). This upper-bounds the probability that a random $h \in H_n^m$ violates the mapping condition regarding a fixed y. Using a union bound on all $y \in \{0,1\}^m$, the lemma follows. ■

D.3 Sampling

In many settings repeated sampling is used to estimate the average of a huge set of values. Namely, a "value" function $\nu : \{0,1\}^n \to \mathbb{R}$ is defined over a huge domain, and one wishes to approximate $\bar{\nu} \stackrel{\text{def}}{=} \frac{1}{2^n} \sum_{x \in \{0,1\}^n} \nu(x)$ without having to inspect the value of ν at each point in the domain. The obvious thing to do is to sample the domain at random, and obtain an approximation to $\bar{\nu}$ by taking the average of the values of ν on the sample points. It turns out that certain "pseudorandom" sequences of sample points may serve almost as well as truly random sequences of sample points, and thus the current problem is indeed related to Section 8.6.

Formal Setting. It is essential to have the range of ν be bounded (or else no reasonable approximation is possible). For simplicity, we adopt the convention of having $[0,1]$ be the range of ν, and the problem for other (predetermined) ranges can be treated analogously. Our notion of approximation depends on two parameters: *accuracy* (denoted ε) and *error probability* (denoted δ). We wish to have an algorithm that, with probability at least $1-\delta$, gets within ε of the correct value. This leads to the following definition.

Definition D.7 (sampler): *A* sampler *is a randomized algorithm that on input parameters* n (length)*,* ε (accuracy) *and* δ (error)*, and oracle access to* any *function* $\nu:\{0,1\}^n \to [0,1]$*, outputs, with probability at least* $1-\delta$*, a value that is at most* ε *away from* $\bar{\nu} \stackrel{\rm def}{=} \frac{1}{2^n}\sum_{x\in\{0,1\}^n} \nu(x)$*. Namely,*

$$\Pr[|\text{sampler}^{\nu}(n,\varepsilon,\delta) - \bar{\nu}| > \varepsilon] \; < \; \delta$$

where the probability is taken over the internal coin tosses of the sampler, also called its random seed.

A non-adaptive sampler *is a sampler that consists of two deterministic algorithms: a* sample generating *algorithm,* G*, and a* evaluation algorithm*,* V*. On input* n,ε,δ *and a random* seed*, algorithm* G *generates a sequence of queries, denoted* $s_1,...,s_m \in \{0,1\}^n$*. Algorithm* V *is given the corresponding* ν*-values (i.e.,* $\nu(s_1),...,\nu(s_m)$*) and outputs an estimate to* $\bar{\nu}$*.*

We are interested in "the complexity of sampling" quantified as a function of the parameters n, ε and δ. Specifically, we will consider three complexity measures: The sample complexity (i.e., the number of oracle queries made by the sampler); the randomness complexity (i.e., the length of the random seed used by the sampler); and the computational complexity (i.e., the running-time of the sampler). We say that a sampler is efficient if its running-time is polynomial in the total length of its queries (i.e., polynomial in both its sample complexity and in n). We will focus on efficient samplers. Furthermore, we will focus on efficient samplers that have optimal (up-to a constant factor) sample complexity, and will wish the randomness complexity to be as low as possible.

We note that all positive results to be reviewed refer to non-adaptive samplers, whereas the lower bound hold also for general samplers. For more details see [86, Sec. 3.6.4].

The naive sampler. The straightforward method (or the naive sampler) consists of *uniformly and independently* selecting sufficiently many sample points (queries), and outputting the average value of the function on these points. Using Chernoff Bound it follows that $O(\frac{\log(1/\delta)}{\varepsilon^2})$ sample points suffice. The naive sampler is optimal (up-to a constant factor) in its sample complexity, but is quite wasteful in randomness.

It is known that $\Omega(\frac{\log(1/\delta)}{\varepsilon^2})$ samples are needed in any sampler, and that that samplers that make $s(n,\varepsilon,\delta)$ queries require randomness at least $n+\log_2(1/\delta) -$

$\log_2 s(n, \varepsilon, \delta) - O(1)$. These lower bounds are tight (as demonstrated by non-explicit and inefficient samplers). These facts guide our quest for improvements, which is aimed at finding more randomness-efficient ways of *efficiently* generating sample sequences that can be used in conjunction with an appropriate evaluation algorithm V. (We stress that V need not necessarily take the average of the values of the sampled points.)

The pairwise-independent sampler. Using the pairwise-independence generator (of §8.6.1.1) for generating sample points, along with the natural evaluation algorithm (which outputs the average of the values of these points), we obtain a great saving in the randomness complexity: pairwise-independent sampling uses $2n$ random bits rather than the $\Omega((\log(1/\delta))\varepsilon^{-2} \cdot n)$ coins used by the naive sampler. Using Eq. (D.3) it follows that $O(1/\delta\varepsilon^2)$ samples are sufficient to get accuracy ε with error δ. Thus, for constant $\delta > 0$, the Pairwise-Independent Sampler is optimal up-to a constant factor in both its sample and randomness complexities. However, for small δ (i.e., $\delta = o(1)$), this sampler is wasteful in sample complexity.

The Median-of-Averages sampler. A new idea is required for going further, and a relevant tool – random walks on expander graphs (see Sections 8.6.3 and E.2) – is needed too. Specifically, we combine the Pairwise-Independent Sampler with the Expander Random Walk Generator (see Proposition 8.29) to obtain a new sampler. The new sampler uses a random walk on an expander with vertex set $\{0,1\}^{2n}$ to generate a sequence of $t \stackrel{\text{def}}{=} O(\log(1/\delta))$ related seeds for t invocations of the Pairwise-Independent Sampler, where each of these invocations uses the corresponding $2n$ bits to generate a sequence of $O(1/\varepsilon^2)$ samples in $\{0,1\}^n$. Furthermore, each of these invocations returns a value that, with probability at least 0.9, is ε-close to $\bar{\nu}$. Theorem 8.28 (see also Exercise 8.31) is used to show that, with probability at least $1-\exp(-t) = 1-\delta$, most of these t invocations return an ε-close approximation. Hence, the median among these t values is an (ε, δ)-approximation to the correct value. The resulting sampler, called the Median-of-Averages Sampler, has sample complexity $O(\frac{\log(1/\delta)}{\varepsilon^2})$ and randomness complexity $2n + O(\log(1/\delta))$, which is optimal up-to a constant factor in both complexities.

Further improvements. The randomness complexity of the Median-of-Averages Sampler can be improved from $2n + O(\log(1/\delta))$ to $n + O(\log(1/\delta\varepsilon))$, while maintaining its (optimal) sample complexity (of $O(\frac{\log(1/\delta)}{\varepsilon^2})$). This is done by replacing the Pairwise Independent Sampler by a sampler that picks a random vertex in a suitable expander and samples all its neighbors.

Averaging Samplers. Averaging (a.k.a. Oblivious) samplers are non-adaptive samplers in which the evaluation algorithm is the natural one: that is, it merely outputs the average of the values of the sampled points. Indeed, the Pairwise-Independent Sampler is an averaging sampler, whereas the Median-of-Averages Sampler is not. Interestingly, averaging samplers have applications for which ordinary non-adaptive samplers do not suffice. Averaging samplers are closely related to randomness extractors, defined and discussed in Section D.4.

An odd perspective. Recall that a non-adaptive sampler consists of a sample generator G and an evaluator V such that for every $\nu:\{0,1\}^n \to [0,1]$ it holds that

$$\mathsf{Pr}_{(s_1,\ldots,s_m)\leftarrow G(U_k)}[|V(\nu(s_1),\ldots,\nu(s_m)) - \bar{\nu}| > \varepsilon] \;<\; \delta.$$

Thus, we may view G as a pseudorandom generator that is subjected to a distinguishability text that is determined by a fixed algorithm V and an arbitrary function $\nu:\{0,1\}^n \to [0,1]$, where we assume that $\mathsf{Pr}[|V(\nu(U_n^{(1)}),\ldots,\nu(U_n^{(m)})) - \bar{\nu}| > \varepsilon] \;<\; \delta$. What is a bit odd here is that, except for the case of averaging samplers, the distinguishability test contains a central component (i.e., the evaluator V) that is potentially custom-made to help the generator G pass the test.[4]

D.4 Randomness Extractors

Extracting almost-perfect randomness from sources of weak (i.e., defected) randomness is crucial for the actual use of randomized algorithms, procedures and protocols. The latter are analyzed assuming that they are given access to a perfect random source, while in reality one typically has access only to sources of weak randomness. Randomness extractors are efficient procedures that (possibly with the help of little extra randomness) enhance the quality of random sources, converting any source of weak randomness to an almost perfect one. In addition, randomness extractors are related to several other fundamental problems, to be further discuss later.

One key parameter, which was avoided in the foregoing discussion, is the class of weak random sources from which we need to extract almost perfect randomness. It is preferable to make as little assumptions as possible regarding the weak random source. In other words, we wish to consider a wide class of such sources, and require the randomness extractor (often referred to as the extractor) to "work well" for any source in this class. A general class of such sources is defined in §D.4.1.1, but first we wish to mention that even for very restricted classes of sources no deterministic extractor can work. To overcome this impossibility result, two approach are used:

Seeded extractors: The first approach consists of considering randomized extractors that use a relatively small amount of randomness (in addition to the weak random source). That is, these extractors obtain two input: a short truly random seed and a relatively long sequence generated by an arbitrary source that belong to the specified class of sources. This suggestion is motivated in two different ways:

[4] Another aspect in which samplers differ from the various pseudorandom generators discussed in Chapter 8 is in the aim to minimize, rather than maximize, the number of blocks (denoted here by m) in the output sequence. However, also in case of samplers the aim is to maximize the block-length (denoted here by n).

1. The application may have access to an almost-perfect random source, but bits from this source are much more expensive than bits from the weak/defected random source. Thus, it makes sense to obtain few high-quality bits from the almost-perfect source and use them to "purify" the cheap bits obtained from the defected source.
2. In some applications (e.g., when using randomized algorithms), it may be possible to scan over all possible values of the seed and run the algorithm using the corresponding extracted randomness. That is, we obtain a sample r from the weak random source, and invoke the algorithm on $\texttt{extract}(s,r)$, for every possible seed s, ruling by majority. (This alternative is typically not applicable to distributed settings.)

Few independent sources: The second approach consists of considering deterministic extractors that obtain samples from a few (say two) *independent* sources of weak randomness. Such extractors are applicable in any setting (including in cryptography), provided that the application has access to the required number of independent weak random sources.

In this section we focus on the first type of extractors (i.e., the *seeded extractors*). This choice is motivated both by the relatively more mature state of the research in that direction and the closer connection between this direction and other topics in complexity.

D.4.1 Definitions and various perspectives

We first present a definition that corresponds to the foregoing motivational discussion, and later discuss its relation to other topics in complexity.

D.4.1.1 The Main Definition

A very wide class of weak random sources corresponds to sources for which no specific output is too probable (cf. [52]). That is, the class is parameterized by a bound β and consists of all sources X such that for every x it holds that $\mathsf{Pr}[X = x] \leq \beta$. In such a case, we say that X has min-entropy[5] at least $\log_2(1/\beta)$. Indeed, we represent sources as random variables, and assume that they are distributed over strings of a fixed length, denoted n. A (n,k)-source is a source that is distributed over $\{0,1\}^n$ and has min-entropy at least k.

An interesting special case of (n,k)-sources is that of sources that are uniform over a subset of 2^k strings. Such sources are called (n,k)-flat. A simple but useful observation is that each (n,k)-source is a convex combination of (n,k)-flat sources.

Definition D.8 (extractor for (n,k)-sources):

[5]Recall that the entropy of a random variable X is defined as $\sum_x \mathsf{Pr}[X = x] \log_2(1/\mathsf{Pr}[X = x])$. Indeed the min-entropy of X equals $\min_x\{\log_2(1/\mathsf{Pr}[X = x])\}$, and is always upper-bounded by its entropy.

1. *An algorithm* $\mathrm{Ext} : \{0,1\}^d \times \{0,1\}^n \to \{0,1\}^m$ *is called an* extractor with error ε for the class $\mathcal{C}$ *if for every source* X *in* $\mathcal{C}$ *it holds that* $\mathrm{Ext}(U_d, X)$ *is* ε*-close to* U_m.

   An algorithm Ext *is called a* (k,ε)-extractor *if it is an extractor with error* ε *for the class of* (n,k)*-sources.*

2. *An algorithm* Ext *is called a* strong extractor with error ε for $\mathcal{C}$ *if for every source* X *in* $\mathcal{C}$ *it holds that* $(U_d, \mathrm{Ext}(U_d, X))$ *is* ε*-close to* (U_d, U_m)*. A* strong (k,ε)-extractor *is defined analogously.*

Using the "decomposition" of (n,k)-sources to (n,k)-flat sources, it follows that Ext *is a* (k,ε)*-extractor if and only if it is an extractor with error* ε *for the class of* (n,k)*-flat sources.* (A similar claim holds for strong extractors.) Thus, much of the technical analysis is conducted with respect to the class of (n,k)-flat sources. For example, it is easy to see that, for $d = \log_2(n/\varepsilon^2) + O(1)$, there exists a (k,ε)-extractor $\mathrm{Ext} : \{0,1\}^d \times \{0,1\}^n \to \{0,1\}^k$. (The proof is by the Probabilistic Method and uses a union bound on the set of all (n,k)-flat sources.)[6]

We seek, however, explicit extractors; that is, extractors that are implementable by polynomial-time algorithms. We note that the evaluation algorithm of any family of pairwise independent hash functions mapping n-bit strings to m-bit strings constitutes a (strong) (k,ε)-extractor for $\varepsilon = 2^{-(k-m)/2}$ (see the alternative proof of Theorem D.5). However, these extractors necessarily use a long seed (i.e., $d \geq 2m$ must hold (and in fact $d = n+2m-1$ holds in Construction D.3)). In Section D.4.2 we survey constructions of efficient (k,ε)-extractors that obtain logarithmic seed length (i.e., $d = O(\log(n/\varepsilon))$). But before doing so, we provide a few alternative perspectives on extractors.

An important note on logarithmic seed length. The case of logarithmic seed length is of particular importance for a variety of reasons. Firstly, when emulating a randomized algorithm using a defected random source (as in Item 2 of the motivational discussion of seeded extractors), the overhead is exponential in the length of the seed. Thus, the emulation of a generic probabilistic polynomial-time algorithm can be done in polynomial time only if the seed length is logarithmic.[7] Similarly, the applications discussed in §D.4.1.2 and §D.4.1.3 are feasible only if the seed length is logarithmic. Lastly, we note that logarithmic seed length is an absolute lower-bound for (k,ε)-extractors, whenever $n > k + k^{\Omega(1)}$ (and $m \geq 1$ and $\varepsilon < 1/2$).

D.4.1.2 Extractors as averaging samplers

There is a close relationship between extractors and averaging samplers (discussed towards the end of Section D.3). We first show that any averaging sampler gives rise

[6] The probability that a random function $\mathrm{Ext} : \{0,1\}^d \times \{0,1\}^n \to \{0,1\}^k$ is not an extractor with error ε for a fixed (n,k)-flat source is upper-bounded by $2^{2^k} \cdot \exp(-\Omega(2^{d+k}\varepsilon^2))$, which is smaller than $1/\binom{2^n}{2^k}$.

[7]

to an extractor. Let $G : \{0,1\}^n \to (\{0,1\}^m)^t$ be the sample generating algorithm of an averaging sampler having accuracy ε and error probability δ. That is, G uses n bits of randomness and generates t sample points in $\{0,1\}^m$ such that for every $f : \{0,1\}^m \to [0,1]$ with probability at least $1-\delta$ the average f-values of these points is in the interval $[\overline{f} \pm \varepsilon]$, where $\overline{f} \stackrel{\rm def}{=} \mathsf{E}[f(U_m)]$. Define $\mathrm{Ext} : [t] \times \{0,1\}^n \to \{0,1\}^m$ such that $\mathrm{Ext}(i,r)$ is the $i^{\rm th}$ sample generated by $G(r)$. We shall prove that Ext is a $(k, 2\varepsilon)$-extractor, for $k = n - \log_2(\varepsilon/\delta)$.

Suppose towards the contradiction that there exists a $(k, 2\varepsilon)$-flat source X such that for some $S \subset \{0,1\}^m$ it is the case that $\mathsf{Pr}[\mathrm{Ext}(U_d, X) \in S] > \mathsf{Pr}[U_m \in S] + 2\varepsilon$, where $d = \log_2 t$ and $[t] \equiv \{0,1\}^d$. Define

$$B = \{x \in \{0,1\}^n : \mathsf{Pr}[\mathrm{Ext}(U_d, x) \in S] > (|S|/2^m) + \varepsilon\}.$$

Then, $|B| > \varepsilon \cdot 2^k = \delta \cdot 2^n$. Defining $f(z) = 1$ if $z \in S$ and $f(z) = 0$ otherwise, we have $\overline{f} \stackrel{\rm def}{=} \mathsf{E}[f(U_m)] = |S|/2^m$. But, for every $r \in B$ the f-average of the sample $G(r)$ is greater than $\overline{f} + \varepsilon$, in contradiction to the hypothesis that the sampler has error probability δ (with respect to accuracy ε).

We now turn to show that extractors give rise to averaging samplers. Let $\mathrm{Ext} : \{0,1\}^d \times \{0,1\}^n \to \{0,1\}^m$ be a (k,ε)-extractor. Consider the sample generation algorithm $G : \{0,1\}^n \to (\{0,1\}^m)^{2^d}$ define by $G(r) = (\mathrm{Ext}(s,r))_{s \in \{0,1\}^d}$. We prove that it corresponds to an averaging sampler with accuracy ε and error probability $\delta = 2^{-(n-k-1)}$.

Suppose towards the contradiction that there exists a function $f : \{0,1\}^m \to [0,1]$ such that for $\delta 2^n = 2^{k+1}$ strings $r \in \{0,1\}^n$ the average f-value of the sample $G(r)$ deviates from $\overline{f} \stackrel{\rm def}{=} \mathsf{E}[f(U_m)]$ by more than ε. Suppose, without loss of generality, that for at least half of these r's the average is greater than $\overline{f} + \varepsilon$, and let B denote the set of these r's. Then, for X that is uniformly distributed on B and is thus a (n, k)-source, we have

$$\mathsf{E}[f(\mathrm{Ext}(U_d, X))] > \mathsf{E}[f(U_m)] + \varepsilon,$$

which (using $|f(z)| \leq 1$ for every z) contradicts the hypothesis that $\mathrm{Ext}(U_d, X)$ is ε-close to U_m.

D.4.1.3 Extractors as randomness-efficient error-reductions

As may be clear from the foregoing discussion, extractors yield randomness-efficient methods for error-reduction. Indeed, error-reduction is a special case of the sampling problem, obtained by considering Boolean functions. Specifically, for a two-sided error decision procedure A, consider the function $f_x : \{0,1\}^{\ell(|x|)} \to \{0,1\}$ such that $f_x(r) = 1$ if $A(x,r) = 1$ and $f_x(r) = 0$ otherwise. Assuming that the probability that A is correct is at least $0.5+\varepsilon$ (say $\varepsilon = 1/6$), error reduction amounts to providing a sampler with accuracy ε and any desired error probability δ for the Boolean function f_x. In particular, any (k,ε)-extractor $\mathrm{Ext} : \{0,1\}^d \times \{0,1\}^n \to \{0,1\}^{\ell(|x|)}$ with $k = n - \log(1/\delta) - 1$ will do, provided 2^d is feasible (e.g., $2^d = \mathrm{poly}(\ell(|x|))$, where $\ell(\cdot)$ represents the randomness complexity of the original algorithm A). The question of interest here is how does n (which represents the randomness complexity of the corresponding sampler) grow as a function of $\ell(|x|)$ and δ.

Error-reduction using the extractor $\mathrm{Ext}:[\mathrm{poly}(\ell(|x|))]\times\{0,1\}^n\to\{0,1\}^{\ell(|x|)}$

	error probability	randomness complexity
original algorithm	$1/3$	$\ell(\vert x\vert)$
resulting algorithm	δ (may depend on $\vert x\vert$)	n (function of $\ell(\vert x\vert)$ and δ)

Jumping ahead, we note that for every $\alpha > 1$, one can obtain $n = O(\ell(|x|)) + \alpha\log_2(1/\delta)$ (cf. [224, 225]).[8] Note that this bound on the randomness-complexity of error-reduction is better than the one provided (for the reduction of one-sided error) by the Expander Random Walk Generator (of Section 8.6.3), albeit the number of samples here is larger (but still polynomial in n).

Mentioning the reduction of *one-sided* error probability, brings us to a corresponding relaxation of the notion of an extractor, which is called a disperser. Loosely speaking, a (k,ε)-disperser is only required to hit (with positive probability) any set of density greater than ε in its image, rather than produce a distribution that is ε-close to uniform.

Definition D.9 (dispersers): *An algorithm* $\mathrm{Dsp} : \{0,1\}^d \times \{0,1\}^n \to \{0,1\}^m$ *is called a* (k,ε)-disperser *if for every* (n,k)*-source* X *the support of* $\mathrm{Dsp}(U_d,X)$ *covers at least* $(1-\varepsilon)\cdot 2^m$ *points. Alternatively, for every set* $S \subset \{0,1\}^m$ *of size greater than* $\varepsilon 2^m$ *it holds that* $\mathsf{Pr}[\mathrm{Dsp}(U_d,X)\in S] > 0$.

Dispersers can be used for the reduction of one-sided error analogously to the use of extractors for the reduction of two-sided error. Specifically, regarding the aforementioned function f_x (and assuming that either $\mathsf{Pr}[f_x(U_{\ell(|x|)}) = 1] > \varepsilon$ or $f_x(U_{\ell(|x|)}) = 0$), we may use any (k,ε)-disperser $\mathrm{Dsp} : \{0,1\}^d\times\{0,1\}^n \to \{0,1\}^{\ell(|x|)}$ in attempt to find a point z such that $f_x(z) = 1$. Indeed, if $\mathsf{Pr}[f_x(U_{\ell(|x|)}) = 1] > \varepsilon$ then $|\{z : (\forall s\in\{0,1\}^d)\, f_x(\mathrm{Dsp}(s,z)) = 0\}| < 2^k$, and thus the one-sided error can be reduced from $1-\varepsilon$ to $2^{-(n-k)}$ while using n random bits.

D.4.1.4 Other perspectives

Extractors and dispersers have an appealing interpretation in terms of bipartite graphs. Starting with dispersers, we view a disperser $\mathrm{Dsp} : \{0,1\}^d \times \{0,1\}^n \to \{0,1\}^m$ as a bipartite graph $G = ((\{0,1\}^n,\{0,1\}^m),E)$ such that $E = \{(x,\mathrm{Dsp}(s,x)) : x \in \{0,1\}^n, s \in \{0,1\}^d\}$. This graph has the property that *any* subset of 2^k vertices on the left (i.e., in $\{0,1\}^n$) has a neighborhood that contains more than a $1-\varepsilon$ fraction of the vertices of the right, which is remarkable in the typical case where d is small (e.g., $d = O(\log n/\varepsilon)$) and $n \gg k \geq m$ whereas $m = \Omega(k)$ (or at least $m = k^{\Omega(1)}$). Furthermore, if Dsp is efficiently computable then this bipartite graph is strongly constructible in the sense that, given a vertex on the left, one can efficiently find all its neighbors. An extractor $\mathrm{Ext} : \{0,1\}^d \times \{0,1\}^n \to \{0,1\}^m$ yields

[8] In general, if $n = \mathrm{poly}(\ell(|x|)) + \alpha\log_2(1/\delta)$ then for sufficiently small $\delta > \exp(-\mathrm{poly}\ell(|x|)))$, we have $n = (\alpha+o(1))\log_2(1/\delta) = \mathrm{poly}(\ell(|x|))$, which means $\delta < 2^{-n/\alpha'}$ for every $\alpha' > \alpha$.

an analogous graph with a even stronger property: the neighborhood multi-set of *any* subset of 2^k vertices on the left covers the vertices on the right in an almost uniform manner.

An odd perspective. In addition to viewing extractors as averaging samplers, which in turn may be viewed within the scope of the pseudorandomness paradigm, we mention here an even more odd perspective. Specifically, randomness extractors may be viewed as randomized (by the seed) algorithms designed on purpose such that to be fooled by any weak random source (but not by an even worse source). Consider an (k,ε)-extractor $\mathrm{Ext}:\{0,1\}^d\times\{0,1\}^n\to\{0,1\}^m$, for say $\varepsilon\leq 1/100$, $m=k=\omega(\log n/\varepsilon)$ and $d=O(\log n/\varepsilon)$, and a potential test T_S, parameterized by a set $S\subset\{0,1\}^m$, such that $\mathsf{Pr}[T_S(x)=1]=\mathsf{Pr}[\mathrm{Ext}(U_d,x)\in S]$ (i.e., on input $x\in\{0,1\}^n$, the test uniformly selects $s\in\{0,1\}^d$ and outputs 1 if and only if $\mathrm{Ext}(s,x)\in S$). Then, for every (n,k)-source X the test T_S does not distinguish X from U_n (i.e., $\mathsf{Pr}[T_S(X)]=\mathsf{Pr}[T_S(U_n)]\pm 2\varepsilon$, because $\mathrm{Ext}(U_d,X)$ is 2ε-close to $\mathrm{Ext}(U_d,U_n)$). On the other hand, for every $(n,k-d-4)$-flat source Y there exists a set S such that T_S distinguish Y from U_n with gap 0.9 (e.g., for S that equals the support of $\mathrm{Ext}(U_d,Y)$, it holds that $\mathsf{Pr}[T_S(Y)]=1$ and $\mathsf{Pr}[T_S(U_n)]\leq|S|\cdot 2^{-m}+\varepsilon=2^{-4}+\varepsilon<0.1$). Furthermore, this class of tests detects as defected, with probability 2/3, any source that has entropy below $(k/4)-d$.[9] Thus, these weird class of tests views each (n,k)-source as "pseudorandom" while detecting sources of lower entropy (e.g., entropy lower than $(k/4)-d$) as non-pseudorandom. Indeed, this perspective stretches the pseudorandomness paradigm quite far.

D.4.2 Constructions

Recall that we seek explicit constructions of extractors; that is, functions $\mathrm{Ext}:\{0,1\}^d\times\{0,1\}^n\to\{0,1\}^m$ that can be computed in polynomial-time. The question, of course, is of parameters; that is, having (k,ε)-extractors with m as large as possible and d as small as possible. We first note that $m\leq k+d-(2\log_2(1/\varepsilon)-O(1))$ and $d\geq\log_2((n-k)/\varepsilon^2)-O(1)$, regardless of explicitness. The aforementioned bounds are in fact tight; that is, there exists (non-explicit) (k,ε)-extractors with $m=k+d-2\log_2(1/\varepsilon)-O(1)$ and $d=\log_2((n-k)/\varepsilon^2)+O(1)$. The obvious goal is to meet these bounds via explicit constructions.

Despite tremendous progress on this problem (and occasional claims regarding "optimal" explicit constructions), the ultimate goal was not reached yet. Yet, we are pretty close. In particular, we have the following.

Theorem D.10 (explicit constructions of extractors): *Explicit (k,ε)-extractors of the form $\mathrm{Ext}:\{0,1\}^d\times\{0,1\}^n\to\{0,1\}^m$ exist in the following cases:*

- *For any constants $\varepsilon,\alpha>0$, with $d=O(\log n)$ and $m=(1-\alpha)\cdot k$.*

[9] For any such source Y, the distribution $Z=\mathrm{Ext}(U_d,Y)$ has entropy at most $k/4=m/4$, and thus is 0.7-far from U_m (and 2/3-far from $\mathrm{Ext}(U_d,U_n)$). The lower-bound on the statistical distance of Z to U_m can be proven by the contra-positive: if Z is δ-close to U_m then its entropy is at least $(1-\delta)\cdot m-1$ (e.g., by using Fano's inequality, see [60, Thm. 2.11.1]).

- *For any constants* $\varepsilon, \alpha > 0$*, with* $d = (1-\alpha)\cdot\log_2 n$ *and* $m = k/\mathrm{poly}(\log n)$*.*
- *For any* $\varepsilon > \exp(-k/\log k)$*, with* $d = O(\log n/\varepsilon)$ *and* $m = \Omega(k/\log k)$*.*

The second item is due to [187], and the other two are due to [147], where these works build on previous ones (which are not cited here). We note that, for sake of simplicity, we did not quote the best possible bounds. Furthermore, we did not mention additional incomparable results (which are relevant for different ranges of parameters). In general, it seems that the "last word" has not been said yet: indeed the current results are close to optimal, but this cannot be said about the way that they are achieved. In view of the foregoing, we refrain from trying to provide an overview of the proof of Theorem D.10, and review instead a conceptual insight that opened the door to much of the recent developments in the area.

The pseudorandomness connection

We conclude this section with an overview of a fruitful connection between extractors and certain pseudorandom generators. The connection, discovered by Trevisan [208], is surprising in the sense that it goes in a non-standard direction: it transforms certain pseudorandom generators into extractors. As argued throughout this book (most conspicuously at the end of Section 7.1.2), computational objects are typically more complex than the corresponding information theoretical objects. Thus, if pseudorandom generators and extractors are at all related (which was not suspected before [208]) then this relation should not be expected to help in the construction of extractors, which seem an information theoretic object. Nevertheless, the discovery of this relation did yield a breakthrough in the study of extractors.[10]

But before describing the connection, let us wonder for a moment. Just looking at the syntax, we note that pseudorandom generators have a single input (i.e., the seed), while extractors have two inputs (i.e., the n-bit long source and the d-bit long seed). But taking a second look at the Nisan–Wigderson Generator (i.e., the combination of Construction 8.17 with an amplification of worst-case to average-case hardness), we note that this construction can be viewed as taking two inputs: a d-bit long seed and a "hard" predicate on d'-bit long strings (where $d' = \Omega(d)$).[11] Now, an appealing idea is to use the n-bit long source as a description of a (worse-case) hard predicate (which indeed means setting $n = 2^{d'}$). The key observation is that even if the source is only weakly random we expect it to represent a function that is hard on the worst-case.

Recall that the aforementioned construction is supposed to yield a pseudorandom generator whenever it starts with a hard predicate. In the current context, where there are no computational restrictions, pseudorandomness is supposed to hold against any (computationally unbounded) distinguisher, and thus here pseudorandomness means being statistically close to the uniform distribution (on strings

[10] We note that once the connection became better understood, influence started going in the "right" direction: from extractors to pseudorandom generators.

[11] Indeed, to fit the current context, we have modified some notations. In Construction 8.17 the length of the seed is denoted by k and the length of the input for the predicate is denoted by m.

of the adequate length, denoted ℓ). Intuitively, this makes sense only if the observed sequence is shorter that the amount of randomness in the source (and seed), which is indeed the case (i.e., $\ell < k + d$, where k denotes the min-entropy of the source). Hence, there is hope to obtain a good extractor this way.

To turn the hope into a reality, we need a proof. Looking again at the Nisan–Wigderson Generator, we note that the proof of indistinguishability of the generator provides a black-box procedure for computing the underlying predicate when given oracle access to any potential distinguisher. In the proofs of Theorems 7.19 and 8.18, this black-box procedure was implemented by a relatively small circuit, so it is clear that this procedure contains relatively little information on top of the observed ℓ-bit long output of the extractor/generator. Specifically, a crude estimate yields an upper-bound of $b \stackrel{\text{def}}{=} \text{poly}(\ell)$ on the amount of information available to the procedure, which is suppose to compute the predicate correctly on every input. That is, this amount of information is supposed to fully determine the predicate in use, which in turn is identical to the n-bit long source. Thus, if the source has min-entropy exceeding b, then it cannot be fully determine using only b bits of information. It follows that the foregoing construction constitutes a $(b+O(1), 1/6)$-extractor (outputting ℓ bits), where the constant $1/6$ is the one used in the proof of Theorem 8.18.[12] Note that this was obtained with a seed of length $d = O(d') = O(\log n)$. The argument can be extended to obtain $(k, \text{poly}(1/k))$-extractors that output $m = k^{\Omega(1)}$ bits using a seed of length $d = O(\log n)$, provided that $k = n^{\Omega(1)}$. (For further details, the interested reader is referred to either [208] or [186].)

We note that the foregoing description has only referred to two abstract properties of the Nisan–Wigderson Generator: (1) the fact that it uses any worst-case hard predicate as a black-box, and (2) the fact that its analysis uses any distinguisher as a black-box. In particular, we viewed the amplification of worst-case hardness to inapproximability (performed in Theorem 7.19) as part of the construction of the pseudorandom generator. An alternative presentation, which is more self-contained, replaces the amplification step of Theorem 7.19 by a direct argument in the current (information theoretic) context and plugs the resulting predicate directly into Construction 8.17.

The alternative presentation. Recall that we viewed the violation of worst-case hardness as an ability to fully determine the predicate, and used the upper-bound on the information available to the "universal" evaluation procedure to upper-bound the number of sources that may be bad. From this perspective, the violation of inapproximability may be viewed as ability to generate a function that is close to the predicate. If the predicates of interests are far apart, then this directly yields the desired bound (on the number of bad sources that are encoded in such predicates). Thus, the idea is to encode the n-bit long source by an error correcting code of length $n' = \text{poly}(n)$ and relative distance $0.5 - (1/n)^2$, and use the resulting codeword as a truth-table of a predicate for Construction 8.17.

[12] Note that the argument assumes $\ell = \exp(\Omega(d'))$, but the polynomial in $b = \text{poly}(\ell)$ is independent of the constant in the Omega-notation.

Such codes (coupled with efficient encoding algorithms) do exist (see Section E.1), and the benefit in using them is that each n'-bit long string (determined by the information available to the aforementioned "universal" evaluation procedure) may be $(0.5 - (1/n))$-close to at most $O(n^2)$ codewords.

Recommended reading

The interested reader is referred to a truly excellent survey of Shaltiel [186]. This survey contains a comprehensive introduction to the area, including an overview of the ideas that underly the various constructions. In particular, the survey describes the approaches used before the discovery of the pseudorandomness connection, the connection itself (and the constructions that arise from it), and the "third generation" of constructions that followed.

Appendix E

Explicit Constructions

It is easier for a camel to go through the eye of a needle, than for a rich man to enter into the kingdom of God.

Matthew, 19:24.

Complexity theory provides a clear definition of the intuitive notion of an explicit construction. Furthermore, it even suggests a hierarchy of different levels of explicitness, referring to the ease of constructing the said object. The basic levels of explicitness are provided by considering the complexity of fully constructing the object. Stronger levels of explicitness emerge when considering the complexity of answering natural queries regarding the object. The aforementioned themes are demonstrated in our brief overview of explicit constructions of error correcting codes and expander graphs. These constructed are, in turn, used in various parts of the main text.

Summary: We review several popular constructions of error correcting codes, culminating with the construction of a concatenated code that combines a Reed-Solomon code with a "mildly explicit" construction of a small code. We also review briefly the notions of locally testable and locally decodable codes, and a useful "list decoding bound" (i.e., bounding the number of codewords that are close to any single sequence).

We review the two standard definitions of expanders, two levels of explicitness, and two properties of expanders that are related to (single-step and multi-step) random walks. We then review two explicit constructions of expander graphs.

E.1 Error Correcting Codes

In this section we highlight some issues and aspects of coding theory that are most relevant to the current book. The interested reader is referred to [203] for a more comprehensive treatment of the computational aspects of coding theory. Structural aspects of coding theory, which are at the traditional focus of that field, are covered in standard textbook such as [151].

Loosely speaking, an error correcting code is a mapping of strings to longer strings such that any two different strings are mapped to a corresponding pair of strings that are far apart (and not merely different). Specifically, $C:\{0,1\}^k \to \{0,1\}^n$ is a (binary) code of distance d if for every $x \neq y \in \{0,1\}^k$ it holds that $C(x)$ and $C(y)$ differ on at least d bit positions.

It will be useful to extend this definition to sequences over an arbitrary alphabet Σ, and to use some notations. Specifically, for $x \in \Sigma^m$, we denote the i^{th} symbol of x by x_i (i.e., $x = x_1 \cdots x_m$), and consider codes over Σ (i.e., mappings of Σ-sequences to Σ-sequences). The mapping (code) $C : \Sigma^k \to \Sigma^n$ has distance d if for every $x \neq y \in \Sigma^k$ it holds that $|\{i : C(x)_i \neq C(y)_i\}| \geq d$. The members of $\{C(x) : x \in \Sigma^k\}$ are called codewords (and in some texts this set itself is called a code).

In general, we define a metric, called Hamming distance, over the set of n-long sequences over Σ. The Hamming distance between y and z, where $y, z \in \Sigma^n$, is defined as the number of locations on which they disagree (i.e., $|\{i : y_i \neq z_i\}|$). The Hamming weight of such sequences is defined as the number of non-zero elements (assuming that one element of Σ is viewed as zero). Typically, Σ is associated with an additive group, and in this case the distance between y and z equals the Hamming weight of $w = y - z$, where $w_i = y_i - z_i$ (for every i).

Asymptotics. We will actually consider infinite families of codes; that is, $\{C_k : \Sigma_k^k \to \Sigma_k^{n(k)}\}_{k\in S}$, where $S \subseteq \mathbb{N}$ (and typically $S = \mathbb{N}$). (N.B., we allow Σ_k to depend on k.) We say that such a family has distance $d : \mathbb{N} \to \mathbb{N}$ if for every $k \in S$ it holds that C_k has distance $d(k)$. Needless to say, both $n = n(k)$ (called the block-length) and $d(k)$ depend on k, and the aim is to have a linear dependence (i.e., $n(k) = O(k)$ and $d(k) = \Omega(n(k))$). In such a case, one talks of the relative rate of the code (i.e., the constant $k/n(k)$) and its relative distance (i.e., the constant $d(k)/n(k)$).

In general, we will often refer to relative distances between sequences. For example, for $y, z \in \Sigma^n$, we say that y and z are ε-close (resp., ε-far) if $|\{i : y_i \neq z_i\}| \leq \varepsilon \cdot n$ (resp., $|\{i : y_i \neq z_i\}| \geq \varepsilon \cdot n$).

Explicitness. A mild notion of explicitness refers to constructing the list of all codewords in time that is polynomial in its length (which is exponential in k). A more standard notion of explicitness refers to generating a specific codeword (i.e., producing $C(x)$ when given x), which coincides with the encoding task mentioned next. Stronger notions of explicitness refer to other computational problems concerning codes (see next).

Computational problems. The most basic computational tasks associated with codes are encoding and decoding (under noise). The definition of the encoding task is straightforward (i.e., map $x \in \Sigma_k^k$ to $C_k(x)$), and an efficient algorithm is required to compute each symbol in $C_k(x)$ in poly$(k, |\Sigma_k|)$-time.[1] When defining the decoding task we note that "minimum distance decoding" (i.e., given $w \in \Sigma_k^{n(k)}$, find x such that $C_k(x)$ is closest to y (in Hamming distance)) is just one natural possibility. Two related variants, regarding a code of distance d, are:

Unique decoding: Given $w \in \Sigma_k^{n(k)}$ that is at Hamming distance less than $d(k)/2$ from some codeword $C_k(x)$, retrieve the corresponding decoding of $C_k(x)$ (i.e., retrieve x).

Needless to say, this task is well-defined because there cannot be two different codewords that are each at Hamming distance less than $d(k)/2$ from w.

List decoding: Given $w \in \Sigma_k^{n(k)}$ and a parameter $d' \geq d(k)/2$, output a list of all $x \in \Sigma_k^k$ that are at Hamming distance at most d' from w.

Typically, one considers the case that $d' < d(k)$. See Section E.1.3 for discussion of upper-bounds on the number of codewords that are within a certain distance from a generic sequence.

Two additional computational tasks are considered in Section E.1.2.

Linear codes. Associating Σ_k with some finite field, we call a code $C_k : \Sigma_k^k \to \Sigma_k^{n(k)}$ linear if it satisfies $C_k(x+y) = C_k(x) + C_k(y)$, where x and y (resp., $C_k(x)$ and $C_k(y)$) are viewed as k-dimensional (resp., $n(k)$-dimensional) vectors over Σ_k, and the arithmetic is of the corresponding vector space. A useful property of linear codes is that their distance equals the Hamming weight of the lightest codeword other than $C_k(0^k)$; that is, $\min_{x \neq y}\{|\{i : C_k(x)_i \neq C_k(y)_i\}|\}$ equals $\min_{x \neq 0^k}\{|\{i : C_k(x)_i \neq 0\}|\}$. Another useful property is that the code is fully specified by a k-by-$n(k)$ matrix, called the generating matrix, that consists of the codewords of some fixed basis of Σ_k^k. That is, the set of all codewords is obtained as the $|\Sigma_k|^k$ different linear combination of the rows of the generating matrix.

E.1.1 A few popular codes

Our focus will on explicitly constructible codes; that is, (families of) codes of the form $\{C_k : \Sigma_k^k \to \Sigma_k^{n(k)}\}_{k \in S}$ that are coupled with efficient encoding and decoding algorithms. But before presenting a few such codes, let us consider a non-explicit construction.

Proposition E.1 (random linear codes): *Let $c > 1$ and $n, d : \mathbb{N} \to \mathbb{N}$ be such that for every k it holds that $n(k) > c \cdot k/(1 - H_2(d(k)/n(k)))$ and $d(k) < n(k)/2c$, where $H_2(\alpha) \stackrel{\rm def}{=} \alpha \log_2(1/\alpha) + (1-\alpha)\log_2(1/(1-\alpha))$. Then, with high probability, a random linear transformation of $\{0,1\}^k$ to $\{0,1\}^{n(k)}$ constitutes a code of distance $d(k)$.*

[1] This formulation is not the one common in coding theory, but it is the most natural one for our applications. On one hand, this formulation is applicable also to codes with super-polynomial block-length. On the other hand, this formulation does not support a discussion of practical algorithms that compute the codeword faster than by computing each of its bits separately.

Thus, *for every constant* $\delta \in (0, 0.5)$ *there exists a constant* $\rho > 0$ *and an infinite family of codes* $\{C_k : \{0,1\}^k \to \{0,1\}^{\rho k}\}_{k \in \mathbb{N}}$ *of relative distance* δ. Specifically, $\rho = c/(1 - H_2(\delta))$ will do.

Proof: We consider a uniformly selected k-by-$n(k)$ generating matrix over GF(2), and upper-bound the probability that it yields a linear code of distance less than $d(k)$. We use a union bound on all possible $2^k - 1$ linear combinations of the rows of the generating matrix, where for each such combination we compute the probability that it yields a vector of Hamming weight less than $d(k)$. Observe that the result of each such linear combination is uniformly distributed over $\{0,1\}^{n(k)}$, and thus has Hamming weight less than $d(k)$ with probability $\sum_{i=0}^{d(k)-1} \binom{n(k)}{i} \cdot 2^{-n(k)} \approx 2^{-(1-H_2(d(k)/n(k))) \cdot n(k))}$. Using $(1 - H_2(d(k)/n(k))) \cdot n(k) > c \cdot k$, the proposition follows. ■

E.1.1.1 A mildly explicit version of Proposition E.1

Note that Proposition E.1 yields a (deterministic) $\exp(k \cdot n(k))$-time algorithm that finds a linear code of distance $d(k)$. The time bound can be improved to $\exp(k + n(k))$, by observing that we may choose the rows of the generating matrix one by one, making sure that all non-empty linear combinations of the current rows have weight at least $d(k)$. Note that the proof of Proposition E.1 can be adapted to assert that as long as we have less than k rows a random choice of the next row will do with high probability. Note that in the case that $n(k) = O(k)$, this yields an algorithm that runs in time that is polynomial in the size of the code (i.e., the number of codewords). Needless to say, this mild level of explicitness is inadequate for most coding applications; however, it will be useful to us in §E.1.1.5.

E.1.1.2 The Hadamard Code

The Hadamard code is the longest (non-repetitive) linear code over $\{0,1\} \equiv \mathrm{GF}(2)$. That is, $x \in \{0,1\}^k$ is mapped to the sequence of all $n(k) = 2^k$ possible linear combinations of its bits (i.e., bit locations in the codewords are associated with k-bit strings, and location $\alpha \in \{0,1\}^k$ in the codeword of x holds the value $\sum_{i=1}^{k} \alpha_i x_i$). It can be verified that each non-zero codeword has weight 2^{k-1}, and thus this code has relative distance $d(k)/n(k) = 1/2$ (albeit its block-length $n(k)$ is exponential in k).

Turning to the computational aspects, we note that encoding is very easy. As for decoding, the warm-up discussion at the beginning of the proof of Theorem 7.7 provides a very fast probabilistic algorithm for unique decoding, whereas Theorem 7.8 provides a very fast probabilistic algorithm for list decoding.

We note that the Hadamard code has played a key role in the proof of the PCP Theorem (Theorem 9.16); see §9.3.2.1.

A propos long codes. We note that the longest (non-repetitive) binary code (called the Long-Code and introduced in [27]) is extensively used in the design of "advanced" PCP systems (see, e.g., [111, 112]). In this code, a k-bit long string

x is mapped to the sequence of $n(k) = 2^{2^k}$ values, each corresponding to the evaluation of a different Boolean function at x; that is, bit locations in the codewords are associated with Boolean functions such that the location associated with $f:\{0,1\}^k \to \{0,1\}$ in the codeword of x holds the value $f(x)$.

E.1.1.3 The Reed–Solomon Code

A Reed-Solomon code is defined for a non-binary alphabet, which is associated with a finite field of n elements, denoted $\mathrm{GF}(n)$. For any $k < n$, we consider the mapping of univariate degree $k-1$ polynomials over $\mathrm{GF}(n)$ to their evaluation at all field elements. That is, $p \in \mathrm{GF}(n)^k$ (viewed as such a polynomial), is mapped to the sequence $(p(\alpha_1), ..., p(\alpha_n))$, where $\alpha_1, ..., \alpha_n$ is a canonical enumeration of the elements of $\mathrm{GF}(n)$.[2]

The Reed-Solomon code offers infinite families of codes with constant rate and constant relative distance (e.g., by taking $n(k) = 3k$ and $d(k) = 2k$), but the alphabet size grows with k (or rather with $n(k) > k$). Efficient algorithms for unique decoding and list decoding are known (see [202] and references therein). These computational tasks correspond to the extrapolation of polynomials based on a noisy version of their values at all possible evaluation points.

E.1.1.4 The Reed–Muller Code

Reed-Muller codes generalize Reed-Solomon codes by considering multi-variate polynomials rather than univariate polynomials. Consecutively, the alphabet may be any finite field, and in particular the two-element field $\mathrm{GF}(2)$. Reed-Muller codes (and variants of them) are extensively used in complexity theory; for example, they underly Construction 7.11 and the PCP constructed at the end of §9.3.2.2. The relevant property of these codes is that, under a suitable setting of parameters that satisfies $n(k) = \mathrm{poly}(k)$, they allow super fast "codeword testing" and "self-correction" (see discussion in Section E.1.2).

For any prime power q and parameters m and r, we consider the set, denoted $P_{m,r}$, of all m-variate polynomials of total degree at most r. Each polynomial in $P_{m,r}$ is represented by the $k = \log_q |P_{m,r}|$ coefficients of all relevant monomials, where in the case that $r < q$ it holds that $k = \binom{m+r}{m}$. We consider the code $C : \mathrm{GF}(q)^k \to \mathrm{GF}(q)^n$, where $n = q^m$, mapping m-variate polynomials of total degree at most r to their values at all q^m evaluation points. That is, the m-variate polynomial p of total degree at most r is mapped to the sequence of values $(p(\overline{\alpha}_1), ..., p(\overline{\alpha}_n))$, where $\overline{\alpha}_1, ..., \overline{\alpha}_n$ is a canonical enumeration of all the m-tuples of $\mathrm{GF}(q)$. The relative distance of this code is lower-bounded by $(q-r)/q$.

In typical applications one sets $r = \Theta(m^2 \log m)$ and $q = \mathrm{poly}(r)$, which yields $k > m^m$ and $n = \mathrm{poly}(r)^m = \mathrm{poly}(m^m)$. Thus we have $n(k) = \mathrm{poly}(k)$ but not $n(k) = O(k)$. As we shall see in Section E.1.2, the advantage (in comparison to the Reed-Solomon code) is that codeword testing and self-correction can be performed

[2] Alternatively, we may map $(v_1, ..., v_k) \in \mathrm{GF}(n)^k$ to $(p(\alpha_1), ..., p(\alpha_n))$, where p is the unique univariate polynomial of degree $k-1$ that satisfies $p(\alpha_i) = v_i$ for $i = 1, ..., k$. Note that this modification amounts to a linear transformation of the generating matrix.

at complexity related to $q = \text{poly}(\log n)$. Actually, in most complexity applications, a variant in which only m-variate polynomials of individual degree $r' = r/m$ are used. In this case, an alternative presentation analogous to the one presented in Footnote 2 is preferred: The information is viewed as a function $H^m \to \text{GF}(q)$, where $H \subset \text{GF}(q)$ is of size $r' + 1$, and is encoded by a m-variate polynomial of individual degree r' evaluated at all points in $\text{GF}(q)^m$.

E.1.1.5 Binary codes of constant relative distance and constant rate

Recall that we seek binary codes of constant relative distance and constant rate. Proposition E.1 asserts that such codes exists, but does not provide an explicit construction. The Hadamard code is explicit but does not have a constant rate (to say the least (since $n(k) = 2^k$)).[3] The Reed-Solomon code has constant relative distance and constant rate but uses a non-binary alphabet (which grows at least linearly with k). We achieve the desired construction by using the paradigm of concatenated codes [74], which is of independent interest. (Indeed, concatenated codes may be viewed as a simple version of the proof composition paradigm presented in §9.3.2.2.)

Intuitively, concatenated codes are obtained by first encoding information, viewed as a sequence over a large alphabet, by some code and next encoding each resulting symbol, which is viewed as a sequence of over a smaller alphabet, by a second code. Formally, consider $\Sigma_1 \equiv \Sigma_2^{k_2}$ and two codes, $C_1 : \Sigma_1^{k_1} \to \Sigma_1^{n_1}$ and $C_2 : \Sigma_2^{k_2} \to \Sigma_2^{n_2}$. Then, the concatenated code of C_1 and C_2, maps $(x_1, ..., x_{k_1}) \in \Sigma_1^{k_1} \equiv \Sigma_2^{k_1 k_2}$ to $(C_2(y_1), ..., C_2(y_{n_1}))$, where $(y_1, ..., y_{n_1})) = C_1(x_1, ..., x_{k_1})$.

Note that the resulting code $C : \Sigma_2^{k_1 k_2} \to \Sigma_2^{n_1 n_2}$ has constant rate and constant relative distance if both C_1 and C_2 have these properties. Encoding in the concatenated code is straightforward. To decode a corrupted codeword of C, we first apply the decoder of C_2 to the n_1 blocks (which are each a n_2-long sequence over Σ_2), obtaining n_1 sequences over Σ_2, which are each interpreted as a symbol of Σ_1. Next we apply the decoder of C_1 to the resulting n_1-long sequence (over Σ_1), and finally we interpret the resulting k_1-long sequence (over Σ_1) as a $k_1 k_2$-long sequence over Σ_2. The key observation is that if we are given a sequence that is $\varepsilon_1 \varepsilon_2$-close to $C(x_1, ..., x_{k_1}) = (C_2(y_1), ..., C_2(y_{n_1}))$ then at least $1 - \varepsilon_1$ of the blocks are ε_2-close to the corresponding $C_2(y_i)$.

We are going to consider the concatenated code obtained by using the Reed-Solomon Code $C_1 : \text{GF}(n_1)^{k_1} \to \text{GF}(n_1)^{n_1}$ as the large code, setting $k_2 = \log_2 n_1$, and using the mildly explicit version of Proposition E.1, $C_2 : \{0,1\}^{k_2} \to \{0,1\}^{n_2}$ as the small code. We use $n_1 = 3k_1$ and $n_2 = O(k_2)$, and so the concatenated code is $C : \{0,1\}^k \to \{0,1\}^n$, where $k = k_1 k_2$ and $n = n_1 n_2 = O(k)$. The key observation is that C_2 can be constructed in $\exp(k_2)$-time, whereas here $\exp(k_2) = \text{poly}(k)$. Furthermore, both encoding and decoding with respect to C_2 can be performed in time $\exp(k_2) = \text{poly}(k)$. Thus, we get:

Theorem E.2 (an explicit good code): *There exists constants $\delta, \rho > 0$ and an*

[3] Binary Reed-Muller codes also fail to simultaneously provide constant relative distance and constant rate.

explicit family of binary codes of rate ρ and relative distance at least δ.[4] *That is, there exists a polynomial-time* (encoding) *algorithm C such that $|C(x)| = |x|/\rho$* (for every x) *and a polynomial-time* (decoding) *algorithm D such that for every y that is $\delta/2$-close to some $C(x)$ it holds that $D(y) = x$. Furthermore, C is a linear code.*

The linearity of C is justified by using a Reed-Solomon code over the extension field $F = \mathrm{GF}(2^{k_2})$, and noting that this code induces a linear transformation over $\mathrm{GF}(2)$. Specifically, the value of a polynomial p over F at a point $\alpha \in F$ can be obtained as a linear transformation of the coefficient of p, when viewed as k_2-dimensional vectors over $\mathrm{GF}(2)$.

Relative distance approaching one half. Starting with a Reed-Solomon code of relative distance δ_1, we obtain a concatenated code of relative distance $\delta_1/2$. Note that, for any constant $\delta_1 < 1$, there exists a Reed-Solomon code $C_1 : \mathrm{GF}(n_1)^{k_1} \to \mathrm{GF}(n_1)^{n_1}$ of relative distance δ_1 and constant rate (i.e., $1 - \delta_1$). Giving up on constant rate, we may start with a Reed-Solomon code of block-length $n_1(k_1) = \mathrm{poly}(k_1)$ and distance $n_1(k_1) - k_1$ over $[n_1(k_1)]$, which yields a (concatenated) binary code of block length $n(k) \approx n_1(k)^2$ and distance approximately $(n(k) - k \cdot n_1(k))/2$. Thus, the relative distance is approximately $(1/2) - (k/\sqrt{n(k)})$.

E.1.2 Two additional computational problems

In this section we briefly review relaxations of two traditional coding theoretic tasks. The purpose of these relaxations is enabling super-fast (randomized) algorithms that provide meaningful information. Specifically, these algorithms may run in sub-linear (e.g., poly-logarithmic) time, and thus cannot possibly solve the unrelaxed version of the problem.

Local testability. This task refers to testing whether a given word is a codeword (in a predetermine code), based on (randomly) inspecting few locations in the word. Needless to say, we can only hope to make an approximately correct decision; that is, accept each codeword and reject with high probability each word that is *far* from the code. (Indeed, this task is within the framework of property testing; see Section 10.1.2.)

Local decodability. Here the task is to recover a specified bit in the plaintext by (randomly) inspecting few locations in a mildly corrupted codeword. This task is somewhat related to the task of self-correction (i.e., recovering a specified bit in the codeword itself, by inspecting few locations in the mildly corrupted codeword).

Note that the Hadamard code is both locally testable and locally decodable as well as self-correctable (based on a constant number of queries into the word); these facts were demonstrated and extensively used in §9.3.2.1. However, the Hadamard code

[4]The relative distance of the code may be actually larger than δ. We set δ such that we can guarantee efficient decoding of any input that is $\delta/2$-close to some codeword.

has an exponential block-length (i.e., $n(k) = 2^k$), and the question is whether one can achieve analogous results with respect to a shorter code (e.g., $n(k) = \mathrm{poly}(k)$). As hinted in §E.1.1.4, the answer is positive (when referring to performing these operations in time that is poly-logarithmic in k):

Theorem E.3 *For some constant $\delta > 0$ and polynomials $n, q : \mathbb{N} \to \mathbb{N}$, there exists an explicit family of codes $\{C_k : [q(k)]^k \to [q(k)]^{n(k)}\}_{k\in\mathbb{N}}$ of relative distance δ that can be locally testable and locally decodable in* $\mathrm{poly}(\log k)$*-time. That is, the following three conditions hold.*

1. Encoding: *There exists a polynomial time algorithm that on input $x \in [q(k)]^k$ returns $C_k(x)$.*
2. Local Testing: *There exists a probabilistic polynomial-time oracle machine T that given k* (in binary)[5] *and oracle access to $w \in [q(k)]^{n(k)}$ distinguishes the case that w is a codeword from the case that w is $\delta/2$-far from any codeword. Specifically:*
   (a) *For every $x \in [q(k)]^k$ it holds that $\Pr[T^{C_k(x)}(k)=1] = 1$.*
   (b) *For every $w \in [q(k)]^{n(k)}$ that is $\delta/2$-far from any codeword of C_k it holds that $\Pr[T^w(k)=1] \leq 1/2$.*

   As usual, the error probability can be reduced by repetitions.
3. Local Decoding: *There exists a probabilistic polynomial-time oracle machine D that given k and $i \in [k]$* (in binary) *and oracle access to any $w \in [q(k)]^{n(k)}$ that is $\delta/2$-close to $C_k(x)$ returns x_i; that is, $\Pr[D^w(k,i) = x_i] \geq 2/3$.*

   Self correction *holds too: there exists a probabilistic polynomial-time oracle machine M that given k and $i \in [n(k)]$* (in binary) *and oracle access to any $w \in [q(k)]^{n(k)}$ that is $\delta/2$-far from $C_k(x)$ returns $C_k(x)_i$; that is, $\Pr[D^w(k,i) = C_k(x)_i] \geq 2/3$.*

We stress that both oracle machines work in time that is polynomial in the binary representation of k, which means that they run in time that is poly-logarithmic in k. The code asserted in Theorem E.3 is a (small modification of a) Reed-Muller code, for $r = m^2 \log m < q(k) = \mathrm{poly}(r)$ and $[n(k)] \equiv \mathrm{GF}(q(k))^m$ (see §E.1.1.4).[6] The aforementioned oracle machines query the oracle $w : [n(k)] \to \mathrm{GF}(q(k))$ at a non-constant number of locations. Specifically, self-correction for location $i \in \mathrm{GF}(q(k))^m$ is performed by selecting a random line (over $\mathrm{GF}(q(k))^m$) that passes through i, recovering the values assigned by w to all $q(k)$ points on this line, and performing univariate polynomial extrapolation (under mild noise). Local testability is easily reduced to self-correction, and local decodability is enabled by the aforementioned modification.

[5] Thus, the running time of T is $\mathrm{poly}(|k|) = \mathrm{poly}(\log k)$.

[6] The modification is analogous to the one presented in Footnote 2: For a suitable choice of k points $\overline{\alpha}_1, ..., \overline{\alpha}_k \mathrm{GF}(q(k))^m$, we map $v_1, ..., v_k$ to $(p(\overline{\alpha}_1), ..., p(\overline{\alpha}_n))$, where p is the unique m-variate polynomial of degree at most r that satisfies $p(\overline{\alpha}_i) = v_i$ for $i = 1, ..., k$.

Constant number of queries. The local testing and decoding algorithms asserted in Theorem E.3 make a polylogarithmic number of queries into the oracle. In contrast, the Hadamard code supports these operation using a *constant number of queries. Can this be obtained with much shorter codewords?* For local testability the answer is definitely positive. One can obtain such locally testable codes with length that is nearly linear (see [34, 63]). For local decodability based on q queries the shortest known code has codewords of length $n(k) = \exp(k^{O(\log\log q)/q\log q})$; see [25]. We conjecture that for every constant q there exists a constant $c_q > 0$ such that $n(k) \geq \exp(k^{c_q})$ must hold (for local decodability based on q queries). In light of this conjecture, we advocate a relaxation of the local decodability task (e.g., the one studied in [33]).

The interested reader is referred to [89], which includes more details on locally testable and decodable codes as well as a wider perspective. (Note, however, that this survey was written prior to [63], which resolves the main open problem posed in [89].)

E.1.3 A list decoding bound

A necessary condition for the feasibility of the list decoding task is that the list of codewords that are close to the given word is short. In this section we present an upper-bound on the length of such lists, noting that this bound has found several applications in complexity theory (and specifically to studies related to the contents of this book). In contrast, we do not present far more famous bounds (which typically refer to the relation among the main parameters of codes (i.e., k, n and d)), because they seem irrelevant to the contents of this book.

We start with a general statement that refers to any alphabet $\Sigma \equiv [q]$, and later specialize it to the case that $q = 2$. Especially in the general case, it is natural and convenient to consider the agreement (rather than the distance) between sequences over $[q]$. Furthermore, it is natural to focus on agreement rate of at least $1/q$, and convenient to state the following result in terms of the "excessive agreement rate" (i.e., the excess beyond $1/q$).[7]

Lemma E.4 (Part 2 of [101, Thm. 15]): *Let $C : [q]^k \to [q]^n$ be an arbitrary code of distance $d \leq n - (n/q)$, and let $\gamma_{\mathtt{C}} \stackrel{\rm def}{=} 1 - (d/n) - (1/q) \geq 0$ denote the corresponding upper-bound on the excessive agreement rate between codewords. Suppose that $\gamma_{\mathtt{W}} \in (0,1)$ satisfies*

$$\gamma_{\mathtt{W}} > \sqrt{\left(1 - \frac{1}{q}\right) \cdot \gamma_{\mathtt{C}}} \qquad \text{(E.1)}$$

Then, for any $w \in [q]^n$, the number of codewords that are at distance at most $(1 - ((1/q) + \gamma_{\mathtt{W}})) \cdot n$ from w (i.e., agree with w on at least $((1/q) + \gamma_{\mathtt{W}}) \cdot n$ positions)

[7] Indeed, we only consider codes with distance $d \leq (1 - 1/q) \cdot n$ and words that are at distance at most d from the code. Note that $1/q$ is a natural threshold for an upper-bound on the relative agreement between sequences over $[q]$, because a random sequence is expected to agree with any fixed sequence on a $1/q$ fraction of the locations.

is upper-bounded by

$$\frac{(1-(1/q))^2 - (1-(1/q)) \cdot \gamma_{\mathtt{C}}}{\gamma_{\mathtt{W}}{}^2 - (1-(1/q)) \cdot \gamma_{\mathtt{C}}} \tag{E.2}$$

In the binary case (i.e., $q=2$), Eq. (E.1) requires $\gamma_{\mathtt{W}} > \sqrt{\gamma_{\mathtt{C}}/2}$ and Eq. (E.2) yields the upper-bound $(1-2\gamma_{\mathtt{C}})/(4\gamma_{\mathtt{W}}{}^2 - 2\gamma_{\mathtt{C}})$. We note that at the end of Section D.4.2, we refer to this bound (for the binary case) while setting $\gamma_{\mathtt{C}} = (1/k)^2$ and $\gamma_{\mathtt{W}} = 1/k$ and using $(1-2\gamma_{\mathtt{C}})/(4\gamma_{\mathtt{W}}{}^2 - 2\gamma_{\mathtt{C}}) = O(k^2)$. In the case of the Hadamard code, we have $\gamma_{\mathtt{C}} = 0$, and so (for every $w \in \{0,1\}^n$ and every $\gamma_{\mathtt{W}} > 0$) the number of codewords that are $(0.5-\gamma_{\mathtt{W}})$-close to w is at most $1/(4\gamma_{\mathtt{W}}{}^2)$. In the general case (and specifically for $q \gg 2$) it is useful to simplify Eq. (E.1) by $\gamma_{\mathtt{W}} > \min\{\sqrt{\gamma_{\mathtt{C}}}, (1/q) + \sqrt{\gamma_{\mathtt{C}} - (1/q)}\}$ and Eq. (E.2) by $\frac{1}{\gamma_{\mathtt{W}}{}^2 - \gamma_{\mathtt{C}}}$.

E.2 Expander Graphs

Loosely speaking, expander graphs are graphs of small degree that exhibit various properties of cliques. In particular, we refer to properties such as the relative sizes of cuts in the graph, and the rate at which a random walk converges to the uniform distribution (relative to the logarithm of the graph size to the base of its degree).

Some technicalities. Typical presentation of expander graphs refer to one of several variants. For example, in some sources, expanders are defined as bipartite graphs, whereas in others they are very far from being bipartite. We shall follow the latter convention. Furthermore, at times we implicitly consider an augmentation of these graphs where self-loops are added to each vertex. For simplicity, we also allow parallel edges.

We often talk of expander graphs while we actually mean an infinite collection of graphs such that each graph in this collection satisfies the same property (which is informally attributed to the collection). For example, when talking of a d-regular expander (graph) we actually refer to an infinite collection of graphs such that each of these graphs is d-regular. Typically, such a collection (or family) contains a single N-vertex graph for every $N \in S$, where S is an infinite subset of $\mathbb{N}$. Throughout this section, we denote such a collection by $\{G_N\}_{N\in S}$, with the understanding that G_N is a graph with N vertices and S is an infinite set of natural numbers.

E.2.1 Definitions and Properties

We consider two definitions of expander graphs, two different notions of explicit constructions, and two useful properties of expanders.

E.2.1.1 Two Mathematical Definitions

We start with two different definitions of expander graphs. These definitions are qualitatively equivalent and even quantitatively related. We start with the algebraic definition, and later present the combinatorial definition.

The algebraic definition (spectral gap). Identifying graphs with their adjacency matrix, we consider the eigenvalues (and eigenvectors) of a graph (or rather of its adjacency matrix). Any d-regular graph $G=(V,E)$ has the uniform vector as an eigenvector corresponding to the eigenvalue d, and if G is connected and not bipartite then all other eigenvalues are strictly smaller than d. The second eigenvalue, denoted $\lambda_2(G)<d$, of such a graph G is thus a tight upper-bound on the *absolute value* of all the other eigenvalues. Using the connection to the combinatorial definition, it follows that $\lambda_2(G)<d-\Omega(1/|V|^2)$ holds (for every connected non-bipartite d-regular graph G). The algebraic definition of expanders refers to an infinite family of d-regular graphs and requires the existence of a *constant* eigenvalue bound that holds for all the graphs in the family.

Definition E.5 *An infinite family of d-regular graphs, $\{G_N\}_{N\in S}$, where $S\subseteq\mathbb{N}$,* satisfies the eigenvalue bound λ *if for every $N\in S$ it holds that $\lambda_2(G_N)\leq\lambda$.*

In such a case we say that the family has spectral gap $d-\lambda$. It will be often convenient to consider relative (or normalized) versions of these quantities, obtained by division by d. This technical definition is the one typically used in complexity theoretic applications, since it directly implies various "mixing properties" (see §E.2.1.3).

The combinatorial definition (expansion). This algebraic definition is related to the combinatorial definition of expansion. Loosely speaking, expansion requires that any (not too big) set of vertices of the graph has a relatively large set of neighbors. Specifically, a graph $G=(V,E)$ is c-expanding if, for every set $S\subset V$ of cardinality at most $|V|/2$, it holds that

$$\Gamma_G(S)\stackrel{\rm def}{=}\{v:\exists u\in S \text{ s.t. } (u,v)\in E\} \tag{E.3}$$

has cardinality at least $(1+c)\cdot|S|$. Equivalently (assuming the existence of self-loops on all vertices), we may require that $|\Gamma_G(S)\setminus S|\geq c\cdot|S|$. Clearly, every connected graph $G=(V,E)$ is $(1/|V|)$-expanding. The combinatorial definition of expanders refers to an infinite family of d-regular graphs and requires the existence of a *constant* expansion bound that holds for all the graphs in the family.

Definition E.6 *An infinite family of d-regular graphs, $\{G_N\}_{N\in S}$* is c-expanding *if for every $N\in S$ it holds that G_N is c-expanding.*

The two definitions of expander graphs are related (see [10, Sec. 9.2]).

Theorem E.7 *Let G be a non-bipartite d-regular graph.*

1. *The graph G is c-expanding for $c\geq(d-\lambda_2(G))/2d$.*
2. *If G is c-expanding then $d-\lambda_2(G)\geq c^2/(4+2c^2)$.*

Thus, any non-zero bound on combinatorial expansion of a family of d-regular graphs yields a non-zero bound on its spectral gap, and vice versa. Note, however, that the back-and-forth translation between these definitions is not tight. Our applications refer to the algebraic definition, and the loss incurred in Theorem E.7 is immaterial for them.

Amplification. The quality of expander graphs improves by raising them to any power $t > 1$ (i.e., raising their adjacency matrix to the t^{th} power), which corresponds to considering graphs in which t-paths are replaced by edges. Using the algebraic definition, we have $\lambda_2(G^t) = \lambda_2(G)^t$, but indeed the degree also gets raised to the power t. Still, the ratio $\lambda_2(G^t)/d^t$ deceases with t. An analogous phenomenon occurs also under the combinatorial definition, provided that some suitable modifications are applied. For example, if $G = (V, E)$ is c-expanding (i.e., for every $S \subseteq V$ it holds that $|\Gamma_G(S)| \geq \min((1 + c) \cdot |S|, |V|/2))$, then for every $S \subseteq V$ it holds that $|\Gamma_{G^t}(S)| \geq \min((1 + c)^t \cdot |S|, |V|/2)$.

The optimal eigenvalue bound. For every d-regular graph $G = (V, E)$, it holds that $\lambda_2(G) \geq 2\gamma_G \cdot \sqrt{d-1}$, where $\gamma_G = 1 - O(1/\log_d |V|)$. Thus, $2\sqrt{d-1}$ is a lower-bound on the eigenvalue bound of any infinite family of d-regular graphs.

E.2.1.2 Two levels of explicitness

A mild level of explicit constructions refer to the complexity of constructing the entire graph. An infinite family of graphs $\{G_N\}_{N\in S}$ is said to be explicitly constructible if there exists a polynomial-time algorithm that, on input 1^N (where $N \in S$), outputs the list of the edges in the N-vertex graph G_N.

The aforementioned level of explicitness suffices when the application requires holding the entire graph and/or runs in time that is lower-bounded by the size of the graph. In contrast, other applications only refer to a huge virtual graph (which is much bigger than their running time), and only require the computation of the neighborhood relations in such a graph. In this case, the following stronger level of explicitness is relevant.

A strongly explicit construction of an infinite family of (d-regular) graphs $\{G_N\}_{N\in S}$ is a polynomial-time algorithm that on input N (in binary), a vertex v in the N-vertex graph G_N and an index i ($i \in \{1, ..., d\}$), returns the i^{th} neighbor of v. That is, the neighbor is determined in time that is polylogarithmic in the size of the graph. Needless to say, the strong level of explicitness implies the basic level.

An additional requirement, which is often forgotten but is very important, refers to the "tractability" of the set S. Specifically, we require the existence of an efficient algorithm that given any $n \in \mathbb{N}$ finds an $s \in S$ such that $n \leq s < 2n$. Corresponding to the foregoing definitions, efficient may mean either running in time $\text{poly}(n)$ or running in time $\text{poly}(\log n)$. The requirement that $n \leq s < 2n$ suffices in most applications, but in some cases a smaller interval (e.g., $n \leq s < n+\sqrt{n}$) is required, whereas in others a larger interval (e.g., $n \leq s < \text{poly}(n)$) suffices.

Greater flexibility. In continuation to the foregoing paragraph, we comment that expanders can be combined in order to obtain expanders for a wider range of sizes. For example, two d-regular c-expanding graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ where $|V_1| \leq |V_2|$ and $c \leq 1$, can be combined into a $(d + 1)$-regular $c/2$-expanding graph on $|V_1|+|V_2|$ vertices by connecting the two graphs with a perfect matching of V_1 and $|V_1|$ of the vertices of V_2 (and adding self-loops to the remaining vertices of V_2). More generally, the d-regular c-expanding graphs, $G_1 = (V_1, E_1)$

through $G_t = (V_t, E_t)$, where $N \stackrel{\rm def}{=} \sum_{i=1}^{t-1} |V_i| \le |V_2|$, yield a $(d+1)$-regular $c/2$-expanding graph on $\sum_{i=1}^{t} |V_i|$ vertices by using a perfect matching of $\cup_{i=1}^{t-1} V_i$ and N of the vertices of V_2.

E.2.1.3 Two properties

The following two properties provide a quantitative interpretation to the statement that expanders approximate the complete graph. The deviation from the latter is represented by an error term that is linear in λ/d.

The mixing lemma. The following lemma is folklore and has appeared in many papers. Loosely speaking, the lemma asserts that expander graphs (for which $d \gg \lambda$) have the property that the fraction of edges between two large sets of vertices approximately equals the product of the densities of these sets. This property is called *mixing*.

Lemma E.8 (Expander Mixing Lemma): *For every d-regular graph $G = (V, E)$ and for every two subsets $A, B \subseteq V$ it holds that*

$$\left| \frac{|(A \times B) \cap E_2|}{|E_2|} - \frac{|A|}{|V|} \cdot \frac{|B|}{|V|} \right| \le \frac{\lambda_2(G)\sqrt{|A| \cdot |B|}}{d \cdot |V|} \le \frac{\lambda_2(G)}{d}$$

where E_2 denotes the set of directed edges that correspond to the undirected edges of G (i.e., $E_2 = \{(u,v) : \{u,v\} \in E\}$ and $|E_2| = d|V|$).

Proof: Let $N \stackrel{\rm def}{=} |V|$ and $\lambda \stackrel{\rm def}{=} \lambda_2(G)$. For any subset of the vertices $S \subseteq V$, we denote its density in V by $\rho(S) \stackrel{\rm def}{=} |S|/N$. Hence, the claim of the lemma is restated as

$$\left| \frac{|(A \times B) \cap E_2|}{d \cdot N} - \rho(A) \cdot \rho(B) \right| \le \frac{\lambda\sqrt{\rho(A) \cdot \rho(B)}}{d}.$$

We proceed by providing bounds on the value of $|(A \times B) \cap E_2|$. To this end we let $\overline{a}$ denote the N-dimensional Boolean vector having 1 in the $i^{\rm th}$ component if and only if $i \in A$. The vector $\overline{b}$ is defined similarly. Denoting the adjacency matrix of the graph G by $M = (m_{i,j})$, we note that $|(A \times B) \cap E_2|$ equals $\overline{a}^\top M \overline{b}$ (because $(i,j) \in (A \times B) \cap E_2$ if and only if it holds that $i \in A$, $j \in B$ and $m_{i,j} = 1$). We consider the *orthogonal eigenvector basis*, $\overline{e_1}, ..., \overline{e_N}$, where $\overline{e_1} = (1, ..., 1)^\top$ and $\overline{e_i}^\top \overline{e_i} = N$ for each i, and write each vector as a linear combination of the vectors in the basis. Specifically, we denote by a_i the coefficient of $\overline{a}$ in the direction of $\overline{e_i}$; that is, $a_i = (\overline{a}^\top \overline{e_i})/N$ and $\overline{a} = \sum_i a_i \overline{e_i}$. Note that $a_1 = (\overline{a}^\top (1, ..., 1)^\top)/N = \rho(A)$ and $\sum_{i=1}^N a_i^2 = (\overline{a}^\top \overline{a})/N = \rho(A)$. Similarly for $\overline{b}$. It now follows that

$$\begin{aligned} |(A \times B) \cap E_2| &= \overline{a}^\top M \left(b_1 \overline{e_1} + \sum_{i=2}^N b_i \overline{e_i} \right) \\ &= \rho(B) \cdot d \cdot |A| + \sum_{i=2}^N b_i \lambda_i \overline{a}^\top \overline{e_i} \end{aligned}$$

where λ_i denotes the $i^{\rm th}$ eigenvalue of M (and indeed $\lambda_1 = d$). Thus,

$$\begin{aligned}\frac{|(A\times B)\cap E|}{dN} &= \rho(B)\rho(A) + \sum_{i=2}^{N}\frac{\lambda_i a_i b_i}{d} \\ &\in \left[\rho(B)\rho(A) \pm \frac{\lambda}{d}\cdot\sum_{i=2}^{N} a_i b_i\right]\end{aligned}$$

Using $\sum_{i=1}^N a_i^2 = \rho(A)$ and $\sum_{i=1}^N b_i^2 = \rho(B)$, and applying Cauchy-Schwartz Inequality, we bound $\sum_{i=2}^N a_i b_i$ by $\sqrt{\rho(A)\rho(B)}$. The lemma follows. ■

The random walk lemma. Loosely speaking, the first part of the following lemma asserts that, as far as remaining trapped in some subset of the vertex set is concerned, a random walk on an expander approximates a random walk on the complete graph.

Lemma E.9 (Expander Random Walk Lemma): *Let $G = ([N], E)$ be a d-regular graph, and consider walks on G that start from a uniformly chosen vertex and take $\ell - 1$ additional random steps, where in each such step we uniformly selects one out of the d edges incident at the current vertex and traverses it.*

Theorem 8.28 (restated): *Let W be a subset of $[N]$ and $\rho \stackrel{\rm def}{=} |W|/N$. Then the probability that such a random walk stays in W is at most*

$$\rho\cdot\left(\rho + (1-\rho)\cdot\frac{\lambda_2(G)}{d}\right)^{\ell-1} \tag{E.4}$$

Exercise 8.30 (restated): *For any $W_0, ..., W_{\ell-1} \subseteq [N]$, the probability that a random walk of length ℓ intersects $W_0 \times W_1 \times \cdots \times W_{\ell-1}$ is at most*

$$\sqrt{\rho_0}\cdot\prod_{i=1}^{\ell-1}\sqrt{\rho_i + (\lambda/d)^2}, \tag{E.5}$$

where $\rho_i \stackrel{\rm def}{=} |W_i|/N$.

The basic principle underlying Lemma E.9 was discovered by Ajtai, Komlos, and Szemerédi [4], who proved a bound as in Eq. (E.5). The better analysis yielding Theorem 8.28 is due to Kahale [125, Cor. 6.1]. More general bounds that refer to the probability of staying in W for a number of times that approximates $|W|/N$ are given in [82], which actually considers an even more general problem (i.e., obtaining Chernoff-type bounds for random variables that are generated by a walk on a Markov Chain).

Proof of Equation (E.5): The basic idea is to view the random walk as the evolution of a corresponding probability vector under suitable transformations.

The transformations correspond to taking a random step in G and to passing through a "sieve" that keeps only the entries that correspond to the current set W_i. The key observation is that the first transformation shrinks the component that is orthogonal to the uniform distribution, whereas the second transformation shrinks the component that is in the direction of the uniform distribution. Details follow.

Let A be a matrix representing the random walk on G (i.e., A is the adjacency matrix of G divided by d). Let $\hat{\lambda}$ denote the absolute value of the second largest eigenvalue of A (i.e., $\hat{\lambda} \stackrel{\text{def}}{=} \lambda_2(G)/d$), and note that $\overline{u} = (N^{-1}, ..., N^{-1})^\top$ (which represents the uniform distribution) is the eigenvector of A that is associated with the largest eigenvalue (which is 1). Let P_i be a 0-1 matrix that has 1-entries only on its diagonal, and furthermore entry (j, j) is set to 1 if and only if $j \in W_i$. Then, the probability that a random walk of length ℓ intersects $W_0 \times W_1 \times \cdots \times W_{\ell-1}$ is the sum of the entries of the vector

$$\overline{v} \stackrel{\text{def}}{=} P_{\ell-1}A \cdots P_2AP_1AP_0\overline{u}. \tag{E.6}$$

We are interested in upper-bounding $\|\overline{v}\|_1$, and use $\|\overline{v}\|_1 \leq \sqrt{N} \cdot \|\overline{v}\|$, where $\|\overline{z}\|_1$ and $\|\overline{z}\|$ denote the L_1-norm and L_2-norm of $\overline{z}$, respectively (e.g., $\|\overline{u}\|_1 = 1$ and $\|\overline{u}\| = N^{-1/2}$). The key observation is that the linear transformation P_iA shrinks every vector.

Main Claim. For every $\overline{z}$, it holds that $\|P_iA\overline{z}\| \leq (\rho_i + \hat{\lambda}^2)^{1/2} \cdot \|\overline{z}\|$.

Proof. Intuitively, A shrinks the component of $\overline{z}$ that is orthogonal to $\overline{u}$, whereas P_i shrinks the component of $\overline{z}$ that is in the direction of $\overline{u}$. Specifically, we decompose $\overline{z} = \overline{z_1} + \overline{z_2}$ such that $\overline{z_1}$ is the projection of $\overline{z}$ on $\overline{u}$ and $\overline{z_2}$ is the component orthogonal to $\overline{u}$. Then, using the triangle inequality and other obvious facts (i.e., $\|P_iA\overline{z_1}\| = \|P_i\overline{z_1}\|$ and $\|P_iA\overline{z_2}\| \leq \|A\overline{z_2}\|$), we have

$$\begin{aligned}
\|P_iA\overline{z_1} + P_iA\overline{z_2}\| &\leq \|P_iA\overline{z_1}\| + \|P_iA\overline{z_2}\| \\
&\leq \|P_i\overline{z_1}\| + \|A\overline{z_2}\| \\
&\leq \sqrt{\rho_i} \cdot \|\overline{z_1}\| + \hat{\lambda} \cdot \|\overline{z_2}\|
\end{aligned}$$

where the last inequality uses the fact that P_i shrinks any uniform vector by eliminating $1-\rho_i$ of its elements, whereas A shrinks the length of any eigenvector except $\overline{u}$ by a factor of at least $\hat{\lambda}$. Using the Cauchy-Schwartz inequality[8], we get

$$\begin{aligned}
\|P_iA\overline{z}\| &\leq \sqrt{\rho_i + \hat{\lambda}^2} \cdot \sqrt{\|\overline{z_1}\|^2 + \|\overline{z_2}\|^2} \\
&= \sqrt{\rho_i + \hat{\lambda}^2} \cdot \|\overline{z}\|
\end{aligned}$$

where the equality is due to the fact that $\overline{z_1}$ is orthogonal to $\overline{z_2}$. □

[8] That is, we get $\sqrt{\rho_i}\|z_1\| + \hat{\lambda}\|z_2\| \leq \sqrt{\rho_i + \hat{\lambda}^2} \cdot \sqrt{\|z_1\|^2 + \|z_2\|^2}$, by using $\sum_{i=1}^n a_i \cdot b_i \leq \left(\sum_{i=1}^n a_i{}^2\right)^{1/2} \cdot \left(\sum_{i=1}^n b_i{}^2\right)^{1/2}$.

Recalling Eq. (E.6) and using the Main Claim (and $\|\overline{v}\|_1 \leq \sqrt{N} \cdot \|\overline{v}\|$), we get

$$\begin{aligned}\|\overline{v}\|_1 &\leq \sqrt{N} \cdot \|P_{\ell-1}A \cdots P_2AP_1AP_0\overline{u}\| \\ &\leq \sqrt{N} \cdot \left(\prod_{i=1}^{\ell-1} \sqrt{\rho_i + \hat{\lambda}^2}\right) \cdot \|P_0\overline{u}\|.\end{aligned}$$

Finally, using $\|P_0\overline{u}\| = \sqrt{\rho_0 N \cdot (1/N)^2} = \rho_0/\sqrt{N}$, we establish Eq. (E.5). ■

Rapid mixing. A property related to Lemma E.9 is that a random walk starting at any vertex converges to the uniform distribution on the expander vertices after a logarithmic number of steps. Using notation as in the proof of Eq. (E.5), we claim that for every starting distribution $\overline{s}$ (including one that assigns all weight to a single vertex), it holds that $\|A^\ell\overline{s} - \overline{u}\|_1 \leq \sqrt{N} \cdot \hat{\lambda}^\ell$, which is meaningful for any $\ell > 0.5 \cdot \log_{1/\hat{\lambda}} N$. The claim is proved by recalling that $\|A^\ell\overline{s} - \overline{u}\|_1 \leq \sqrt{N} \cdot \|A^\ell\overline{s} - \overline{u}\|$ and using the fact that $\overline{s} - \overline{u}$ is orthogonal to $\overline{u}$ (because the former is a zero-sum vector). Thus, $\|A^\ell\overline{s} - \overline{u}\| \leq \hat{\lambda}^\ell \|\overline{s} - \overline{u}\|$ and using $\|\overline{s} - \overline{u}\| < 1$ the claim follows.

E.2.2 Constructions

Many explicit constructions of expanders were given, starting in [152] and culminating in the optimal construction of [148] where $\lambda = 2\sqrt{d-1}$. Most of these constructions are quite simple (see, e.g., §E.2.2.1), but their analysis is based on non-elementary results from various branches of mathematics. In contrast, the construction of Reingold, Vadhan, and Wigderson [178], presented in §E.2.2.2, is based on an iterative process, and its analysis is based on a relatively simple algebraic fact regarding the eigenvalues of matrices.

Before turning to these explicit constructions we note that it is relatively easy to prove the existence of 3-regular expanders, by using the Probabilistic Method (cf. [10]) and referring to the combinatorial definition of expansion.[9]

[9] As a warm-up, one may establish the existence of d-regular expanders, for some constant d. In particular, towards dealing with the case of $d = 3$, consider a random graph G on the vertex set $V = \{0, ..., n-1\}$ constructed by augmenting the fixed edge set $\{\{i, i+1 \bmod n\} : i = 0, ..., n-1\}$ with $d-2$ uniformly (and independently) chosen perfect matchings of the vertices of $F \stackrel{\text{def}}{=} \{0, ..., (n/2)-1\}$ to the vertices of $L \stackrel{\text{def}}{=} \{n/2, ..., n-1\}$. Noting that $|(\Gamma_G(S\cap F)\cap F)| \geq |S\cap F| - 1$ (and similarly for L), and assuming without loss of generality that $|S \cap F| \geq |S \cap L|$, we focus on upper-bounding the probability that, for some $\varepsilon > 0$, there exists a set $S \subset V$ of size at most $n/2$ such that $|(\Gamma_G(S\cap F)\cap L)\setminus\Gamma_G(S\cap L)| < \varepsilon|S|$. Fixing such a set S, the corresponding probability is upper-bounded by p_S^{d-2}, where

$$p_S \stackrel{\text{def}}{=} \frac{\binom{(n/2)-\ell}{\varepsilon|S|} \cdot \binom{\ell+\varepsilon|S|}{|S\cap F|}}{\binom{n/2}{|S\cap F|}}$$

where $\ell = |\Gamma_G(S \cap L) \cap L|$. Indeed, we may focus on the case that $|S \cap F| \leq \ell + \varepsilon|S|$ (because in the other case $p_S = 0$), and observe that for sufficiently small $\varepsilon > 0$ it holds that $p_S < \binom{n}{|S|}^{-2/5}$. This suffices if $d \geq 5$. To deal with the case $d = 3$, we reduce the analysis of the expansion of sets S to the case that the set S consists of relatively long arithmetic sequences that use an increment

E.2.2.1 The Margulis–Gabber–Galil Expander

For every natural number m, consider the graph with vertex set $\mathbb{Z}_m \times \mathbb{Z}_m$ and edge set in which every $\langle x, y\rangle \in \mathbb{Z}_m \times \mathbb{Z}_m$ is connected to the vertices $\langle x \pm y, y\rangle$, $\langle x \pm (y+1), y\rangle$, $\langle x, y \pm x\rangle$, and $\langle x, y \pm (x+1)\rangle$, where the arithmetic is modulo m. This yields an extremely simple and explicit 8-regular graph with second eigenvalue that is bounded by a constant $\lambda < 8$ that is independent of m. Thus we get:

Theorem E.10 *For some constant $\lambda < 8$ there exists a strongly explicit construction of a family of $(8, \lambda)$-expanders for $\{m^2 : m \in \mathbb{N}\}$. Furthermore, the neighbors of a vertex can be computed in logarithmic-space.*[10]

An appealing property of Theorem E.10 is that, for every $n \in \mathbb{N}$, it directly yields expanders with vertex set $\{0,1\}^n$. This is obvious in case n is even, but can be easily achieved also for odd n (e.g., use two copies of the graph for $n-1$, and connect the two copies by the obvious perfect matching).

Theorem E.10 is due to Gabber and Galil [80], building on the basic approach suggested by Margulis [152]. It was later shown that $\lambda = 5\sqrt{2} < 7.1$. Recall, however, that the optimal construction of [148] achieves $\lambda = 2\sqrt{d-1}$ but there are restrictions on d (i.e., $d-1$ should be a prime congruent to 1 modulo 4) and on the graph sizes (i.e., the set S) for which this construction works.

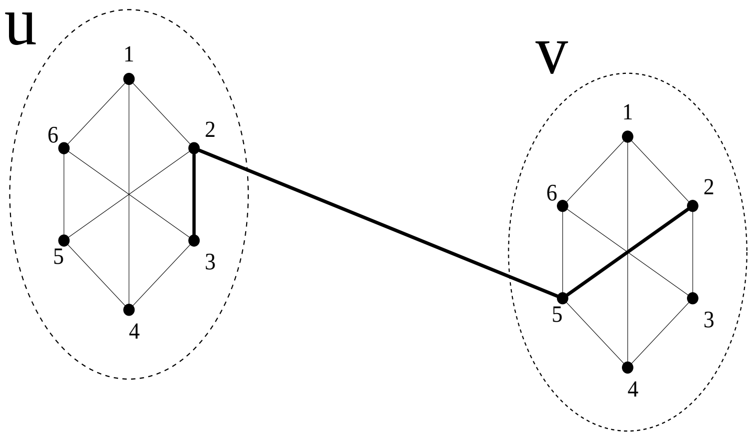
E.2.2.2 The Iterated Zig-Zag Construction

The starting point of the construction is a very good expander G of *constant size*, which may be found by an exhaustive search. The construction of a large expander graph proceeds in iterations, where in the i^{th} iteration the current graph G_i and the fixed graph G are combined, resulting in a larger graph G_{i+1}. The combination step guarantees that the expansion property of G_{i+1} is at least as good as the expansion of G_i, while G_{i+1} maintains the degree of G_i and is a constant times larger than G_i. The process is initiated with $G_1 = G^2$ and terminates when we obtain a graph G_t of approximately the desired size (which requires a logarithmic number of iterations).

The Zig-Zag product. The heart of the combination step is a new type of "graph product" called *Zig-Zag product*. This operation is applicable to any pair of graphs $G = ([D], E)$ and $G' = ([N], E')$, provided that G' (which is typically larger than G) is D-regular. For simplicity, we assume that G is d-regular (where typically $d \ll D$). The Zig-Zag product of G' and G, denoted $G' \textcircled{z} G$, is defined as a graph with vertex set $[N] \times [D]$ and an edge set that includes an edge between

of either 1 or 2, where relatively long means longer than some sufficiently large constant $t = 1/\sqrt{\varepsilon}$. (Note that if S contains more that $2\sqrt{\varepsilon} \cdot |S|$ elements in relatively short arithmetic sequences then $|\Gamma_G(S)| > 2\sqrt{\varepsilon}|S|/2t$, and prove expansion factor at least $\varepsilon' = 4\sqrt{\varepsilon}$ for the rest.) The reduction allows the application of the union bound, while considering only $2^{|S|/t} \cdot \binom{n}{|S|/t}$ possible sets S (rather than $\binom{n}{|S|}$ such sets).

[10]In fact, under a suitable encoding of the vertices and for m that is a power of two, the neighbors can be computed by a on-line algorithm that uses a constant amount of space.



In this example G' is 6-regular and G is a 3-regular graph having six vertices. In the graph G' (not shown), the 2nd edge of vertex u is incident at v, as its 5th edge. The wide 3-segment line shows one of the corresponding edges of G'ⓩG.

Figure E.1: Detail of the zig-zag product of G' and G.

$\langle u,i\rangle \in [N]\times[D]$ and $\langle v,j\rangle$ if and only if $(i,k),(\ell,j)\in E$ and the k^{th} edge incident at u equals the ℓ^{th} edge incident at v. (See Figure E.1, where vertex $\langle u,3\rangle$ is connected in G'ⓩG to $\langle v,2\rangle$, using the G-edges $(3,2)$ and $(5,2)$.)

It will be convenient to represent graphs like G' by their edge rotation function, denoted $R' : [N]\times[D] \to [N]\times[D]$, such that $R'(u,i)=(v,j)$ if (u,v) is the i^{th} edge incident at u as well as the j^{th} edge incident at v. For simplicity, we assume that G is edge-colorable with d colors, which in turn yields a natural edge rotation function (i.e., $R(i,\alpha)=(j,\alpha)$ if the edge (i,j) is colored α). We will denote by $E_\alpha(i)$ the vertex reached from $i\in[D]$ by following the edge colored α (i.e., $E_\alpha(i)=j$ iff $R(i,\alpha)=(j,\alpha)$). The Zig-Zag product of G' and G, denoted G'ⓩG, is then defined as a graph with the vertex set $[N]\times[D]$ and the edge rotation function

$$(\langle u,i\rangle,\langle\alpha,\beta\rangle)\mapsto(\langle v,j\rangle,\langle\beta,\alpha\rangle) \quad \text{if } R'(u,E_\alpha(i))=(v,E_\beta(j)). \tag{E.7}$$

That is, edges are labeled by pairs over $[d]$, and the $\langle\alpha,\beta\rangle^{\text{th}}$ edge out of vertex $\langle u,i\rangle\in[N]\times[D]$ is incident at the vertex $\langle v,j\rangle$ (as its $\langle\beta,\alpha\rangle^{\text{th}}$ edge) if $R(u,E_\alpha(i))=(v,E_\beta(j))$. (That is, based on $\langle\alpha,\beta\rangle$, we take a G-step from $\langle u,i\rangle$ to $\langle u,E_\alpha(i)\rangle$, then viewing $\langle u,E_\alpha(i)\rangle\equiv(u,E_\alpha(i))$ as an edge of G' we rotate it to $(v,j')\stackrel{\text{def}}{=}R'(u,E_\alpha(i))$, and take a G-step from $\langle v,j'\rangle$ to $\langle v,E_\beta(j')\rangle$, while defining $j=E_\beta(j')$ and using $j'=E_\beta(E_\beta(j'))=E_\beta(j)$.)

Clearly, the graph G'ⓩG is d^2-regular and has $D\cdot N$ vertices. The key fact, proved in [178], is that the relative eigenvalue of the zig-zag product is upper-bounded by the sum of the relative eigenvalues of the two graphs (i.e., $\bar\lambda_2(G'ⓩG)\le\bar\lambda_2(G')+\bar\lambda_2(G)$, where $\bar\lambda_2(\cdot)$ denotes the relative eigenvalue of the relevant graph).

The iterated construction. The iterated expander construction uses the aforementioned zig-zag product as well as graph squaring. Specifically, the construction starts with the d^2-regular graph $G_1 = G^2 = ([D], E^2)$, where $D = d^4$ and $\bar{\lambda}_2(G) < 1/4$, and proceeds in iterations such that $G_{i+1} = G_i^2 \textcircled{z} G$ for $i = 1, 2, ..., t-1$. That is, in each iteration, the current graph is first squared and then composed with the fixed graph G via the zig-zag product. This process maintains the following two invariants:

1. The graph G_i is d^2-regular and has D^i vertices.

   (The degree bound follows from the fact that a zig-zag product with the d-regular graph always yields a d^2-regular graph.)

2. The relative eigenvalue of G_i is smaller than one half.

   (Here we use the fact that $\bar{\lambda}_2(G_{i-1}^2 \textcircled{z} G) \leq \bar{\lambda}_2(G_{i-1}^2) + \bar{\lambda}_2(G)$, which in turn equals $\bar{\lambda}_2(G_{i-1})^2 + \bar{\lambda}_2(G) < (1/2)^2 + (1/4)$. Note that graph squaring is used to reduce the relative eigenvalue of G_i before increasing it by zig-zag product with G.)

To ensure that we can construct G_i, we should show that we can actually construct the edge rotation function that correspond to its edge set. This boils down to showing that, given the edge rotation function of G_{i-1}, we can compute the edge rotation function of G_{i-1}^2 as well as of its zig-zag product with G. Note that this computation amounts to two recursive calls to computations regarding G_{i-1} (and two computations that correspond to the constant graph G). But since the recursion depth is logarithmic in the size of the final graph, the time spend in the recursive computation is polynomial in the size of the final graph. This suffices for the minimal notion of explicitness, but not for the stronger one.

The strongly explicit version. To achieve a *strongly explicit construction*, we slightly modify the iterative construction. Rather than letting $G_{i+1} = G_i^2 \textcircled{z} G$, we let $G_{i+1} = (G_i \times G_i)^2 \textcircled{z} G$, where $G' \times G'$ denotes the *tensor product of G' with itself*; that is, if $G' = (V', E')$ then $G' \times G' = (V' \times V', E'')$, where

$$E'' = \{(\langle u_1, u_2 \rangle, \langle v_1, v_2 \rangle) : (u_1, v_1), (u_2, v_2) \in E'\}$$

with an edge rotation function

$$R''(\langle u_1, u_2 \rangle, \langle i_1, i_2 \rangle) = (\langle v_1, v_2 \rangle, \langle j_1, j_2 \rangle)$$

where $R'(u_1, i_1) = (v_1, j_1)$ and $R'(u_2, i_2) = (v_2, j_2)$. (We still use $G_1 = G^2$.) Using the fact that tensor product preserves the relative eigenvalue and using a d-regular $G = ([D], E)$ with $D = d^8$, we note that the modified $G_i = (G_{i-1} \times G_{i-1})^2 \textcircled{z} G$ is a d^2-regular graph with $(D^{2^{i-1}-1})^2 \cdot D = D^{2^i-1}$ vertices, and $\bar{\lambda}_2(G_i) < 1/2$ (because $\bar{\lambda}_2((G_{i-1} \times G_{i-1})^2 \textcircled{z} G) \leq \bar{\lambda}_2(G_{i-1})^2 + \bar{\lambda}_2(G)$). Computing the neighbor of a vertex in G_i boils down to a constant number of such computations regarding G_{i-1}, but due to the tensor product operation the depth of the recursion is only double-logarithmic in the size of the final graph (and hence logarithmic in the length of the description of vertices in it).

Digest. In the first construction, the zig-zag product was used both in order to increase the size of the graph and to reduce its degree. However, as indicated by the second construction (where the tensor product of graphs is the main vehicle for increasing the size of the graph), the primary effect of the zig-zag product is to reduce the degree, and the increase in the size of the graph is merely a side-effect (which is actually undesired in Section 5.2.4). In both cases, graph squaring is used in order to compensate for the modest increase in the relative eigenvalue caused by the zig-zag product. In retrospect, the second construction is the "correct" one, because it decouples three different effects, and uses a natural operation to obtain each of them: Increasing the size of the graph is obtained by tensor product of graphs (which in turn increases the degree), a degree reduction is obtained by the zig-zag product (which in turn increases the relative eigenvalue), and graph squaring is used in order to reduce the relative eigenvalue.

Stronger bound regarding the effect of the zig-zag product. In the foregoing description we relied on the fact, proved in [178], that the relative eigenvalue of the zig-zag product is upper-bounded by the sum of the relative eigenvalues of the two graphs. Actually, a stronger upper-bound is proved in [178]: For $g(x,y) = (1-y^2)\cdot x/2$, it holds that

$$\begin{aligned}\bar{\lambda}_2(G' \text{ⓩ} G) &\leq g(\bar{\lambda}_2(G'),\bar{\lambda}_2(G)) + \sqrt{g(\bar{\lambda}_2(G'),\bar{\lambda}_2(G))^2 + \bar{\lambda}_2(G)^2} \qquad \text{(E.8)}\\ &\leq 2g(\bar{\lambda}_2(G'),\bar{\lambda}_2(G)) + \bar{\lambda}_2(G)\\ &= (1-\bar{\lambda}_2(G)^2)\cdot\bar{\lambda}_2(G') + \bar{\lambda}_2(G).\end{aligned}$$

Thus, we get $\bar{\lambda}_2(G' \text{ⓩ} G) \leq \bar{\lambda}_2(G') + \bar{\lambda}_2(G)$. Furthermore, Eq. (E.8) yields a non-trivial bound for any $\bar{\lambda}_2(G'), \bar{\lambda}_2(G) < 1$, even in case $\bar{\lambda}_2(G')$ is very close to 1 (as in [177]; see proof of Theorem 5.6). Specifically, Eq. (E.8) is upper-bounded by

$$\begin{aligned} g(\bar{\lambda}_2(G'),\bar{\lambda}_2(G)) &+ \sqrt{\left(\frac{1-\bar{\lambda}_2(G)^2}{2}\right)^2 + \bar{\lambda}_2(G)^2}\\ &= \frac{(1-\bar{\lambda}_2(G)^2)\cdot\bar{\lambda}_2(G')}{2} + \frac{1+\bar{\lambda}_2(G)^2}{2}\\ &= 1 - \frac{(1-\bar{\lambda}_2(G)^2)\cdot(1-\bar{\lambda}_2(G'))}{2} \qquad \text{(E.9)}\end{aligned}$$

Thus, $1-\bar{\lambda}_2(G' \text{ⓩ} G) \geq (1-\bar{\lambda}_2(G)^2)\cdot(1-\bar{\lambda}_2(G'))/2$. In particular, if $\bar{\lambda}_2(G) < 1/\sqrt{3}$ then $1-\bar{\lambda}_2(G' \text{ⓩ} G) > (1-\bar{\lambda}_2(G'))/3$. This fact plays an important role in the proof of Theorem 5.6).

Appendix F

Some Omitted Proofs

A word of a Gentleman is better than a proof,
but since you are not a Gentleman – please provide a proof.

Leonid A Levin (1986)

The proofs presented in this appendix were not included in the main text for a variety of reasons (e.g., they were deemed too technical and/or out-of-pace for the corresponding location). On the other hand, our presentation of these proofs is sufficiently different from the original and/or standard presentation, and so we see a benefit in including these proofs in the current book.

Summary: We present proofs of the following results:

1. $\mathcal{PH}$ is reducible to $\#\mathcal{P}$ (and in fact to $\oplus\mathcal{P}$) via randomized Karp-reductions. The proof follows the underlying ideas of Toda's original proof, but the actual presentation is quite different.
2. For any integral function f that satisfies $f(n) \in \{2, ..., \mathrm{poly}(n)\}$, it holds that $\mathcal{IP}(f) \subseteq \mathcal{AM}(O(f))$ and $\mathcal{AM}(O(f)) \subseteq \mathcal{AM}(f)$. The proofs differ from the original proofs (provided in [107] and [20], respectively) only in secondary details.

F.1 Proving that $\mathcal{PH}$ reduces to $\#\mathcal{P}$

Recall that Theorem 6.14 asserts that $\mathcal{PH}$ is Cook-reducible to $\#\mathcal{P}$ (via deterministic reductions). Here we prove a closely related result (also due to Toda [206]), which relaxes the requirement from the reduction (allowing it to be randomized) but uses an oracle to a seemingly weaker class. The latter class is denoted $\oplus\mathcal{P}$ and is the "modulo 2 analogue" of $\#\mathcal{P}$. Specifically, a Boolean function f is in $\oplus\mathcal{P}$ if there exists a function $g \in \#\mathcal{P}$ such that for every x it holds that $f(x) = g(x) \bmod 2$. Equivalently, f is in $\oplus\mathcal{P}$ if there exists a search problem

$R \in \mathcal{PC}$ such that $f(x) = |R(x)| \bmod 2$, where $R(x) = \{y : (x,y) \in R\}$. (The $\oplus$ symbol in the notation $\oplus\mathcal{P}$ actually represents parity, which is merely addition modulo 2. Indeed, a notation such as $\#_2\mathcal{P}$ would have been more appropriate.)

Theorem F.1 *Every set in $\mathcal{PH}$ is reducible to $\oplus\mathcal{P}$ via a probabilistic polynomial-time reduction. Furthermore, the reduction is many-to-one and fails with negligible error probability.*

The proof follows the underlying ideas of the original proof [206], but the actual presentation is quite different. Alternative proofs of Theorem F.1 can be found in [126, 198].

> **Teaching note:** It is quite easy to prove a non-uniform analogue of Theorem F.1, which asserts that AC0 circuits can be approximated by the circuits consisting of an unbounded parity of conjunctions, where each conjunction has polylogarithmic size. Turning this argument into a proof of Theorem F.1 requires careful implementation as well as using transitions of the type presented in Exercise 3.6. Furthermore, such a proof tends to obscure the conceptual steps that underly the entire argument.

Proof Sketch: The proof uses three main ingredients. The first ingredient is the fact that $\mathcal{NP}$ is reducible to $\oplus\mathcal{P}$ via a probabilistic polynomial-time Karp-reduction, and that this reduction in "highly structured" (see Footnote 2). The second ingredient is the fact that error-reduction is available in the correct context, resulting in reductions that have exponentially vanishing error probability.[1] The third ingredient may be schematically paraphrased by the Boolean equality $\oplus_i(\zeta_i \wedge (\oplus_j X_{i,j})) = \oplus_{i,j}(\zeta_i \wedge X_{i,j})$. These ingredients correspond to the three main steps of the proof.

Rather than presenting the actual proof at an abstract level (while using suitable definitions), we prefer a concrete presentation in which the third step is performed by an extension of the first step. In particular, this allows performing the third step at a level that clarifies what exactly is going on. In addition, it offers the opportunity for revisiting the standard presentations of the first step, while correcting what we consider to be a conceptual error in these presentations. Thus, we begin by dealing with the easy case of $\mathcal{NP}$ (and co$\mathcal{NP}$), and then turn the implementation of error-reduction (in the current context). *Such error-reduction is crucial as a starting point for the third step, which deals with the case of* Σ_2. When completing the third step, we will have all the ingredients needed for the general case (of dealing with Σ_k for any $k \geq 2$), and we will thus conclude with a few comments regarding the latter case. Admittingly, the description of the last part is very sketchy and an actual implementation would be quite cumbersome; however, the ideas are all present in the case of Σ_2. Furthermore, we believe that the case of Σ_2 is of significant interest per se.

[1] We comment that such an error-reduction is not available in the context of reductions to unique solution problems. This comment is made in view of the similarity between the reduction of $\mathcal{NP}$ to $\oplus\mathcal{P}$ and the reduction of $\mathcal{NP}$ to problems of unique solution.

Let us first prove that every set in $\mathcal{NP}$ is reducible to $\oplus\mathcal{P}$ via a probabilistic polynomial-time Karp-reduction. Indeed, this follows immediately from the NP-hardness of deciding unique solution for some relations $R \in \mathcal{PC}$ (i.e., Theorem 6.27), because the corresponding modulo 2 counter (i.e., $\#R$ mod 2) solves the unique solution problem associated with this relation (i.e., deciding the existence of unique solutions for R). Specifically, Theorem 6.27 asserts that, for some complete problems $R \in \mathcal{PC}$, deciding membership in any NP-set is reducible in probabilistic polynomial-time to the promise problem $(\mathtt{US}_R, \overline{S}_R)$, where $\mathtt{US}_R = \{x : |R(x)| = 1\}$ and $\overline{S}_R = \{x : |R(x)| = 0\}$. The point is that the function $\oplus R(x) \stackrel{\rm def}{=} |R(x)| \bmod 2$ solves the latter promise problem; that is, $(\mathtt{US}_R, \overline{S}_R)$ is reducible to $\oplus R$ by the identity mapping. Thus, any reduction to the promise problem $(\mathtt{US}_R, \overline{S}_R)$ constitutes a reduction to $\oplus R$. Still, for the sake of self-containment and concreteness, let us consider an alternative proof.[2]

Step 1: a direct proof for the case of $\mathcal{NP}$. As in the proof of Theorem 6.27, we start with any $R \in \mathcal{PC}$ and our goal is reducing $S_R = \{x : |R(x)| \geq 1\}$ to $\oplus\mathcal{P}$ by a randomized Karp-reduction.[3] The standard way of obtaining such a reduction (e.g., in [126, 165, 198, 206]) consists of just using the reduction presented in the proof of Theorem 6.27, but we believe that this way is conceptually wrong. Recall that the proof of Theorem 6.27 consists of implementing a randomized sieve that has the following property. For any $x \in S_R$, with noticeable probability, a single element of $R(x)$ passes the sieve (and this event can be detected by an oracle to a unique solution problem). Indeed, an oracle in $\oplus\mathcal{P}$ correctly detects the case in which a single element of $R(x)$ passes the sieve. However, by definition, an oracle in $\oplus\mathcal{P}$ correctly detects the more general case in which any odd number of elements of $R(x)$ pass the sieve. Thus, insisting on a random sieve that allows the passing of a single element of $R(x)$ seems an over-kill (or at least is conceptually wrong). Instead, *we should just apply a less stringent random sieve that, with noticeable probability, allows the passing of an odd number of elements of* $R(x)$. The adequate tool for this sieve is a small-bias generator (see Section 8.6.2). Specifically, we use a strongly efficient generator (see §8.6.2.1) that given a seed s and index i produces the adequate bit, denoted $G(s,i)$, in the $\ell(|s|)$-bit generator sequence $G(s)$, where $G(U_k)$ has small bias and $\ell(k) = \exp(\Omega(k))$. Assuming, without loss of generality, that $R(x) \subseteq \{0,1\}^{p(|x|)}$ for some polynomial p, we consider the relation

$$R_2 = \{(\langle x,s\rangle, y) : (x,y) \in R \wedge G(s,y) = 1\} \tag{F.1}$$

where $y \in \{0,1\}^{p(|x|)} \equiv [2^{p(|x|)}]$ and $s \in \{0,1\}^{O(|y|)}$ such that $\ell(|s|) = 2^{|y|}$. Then, for every $x \in S_R$, with probability at least 1/3, a uniformly selected $s \in$

[2] Indeed, the presentation can be modified such that the following direct proof is omitted. In this case, we shall only use the fact that each set in $\mathcal{NP}$ is reducible to $\oplus\mathcal{P}$ by a randomized Karp-reduction. Actually, we will have to rely on the fact that the reduction is "highly structured" in the sense that for any polynomially bounded relation R it reduces S_R to $\oplus R_2$ such that x is mapped to $\langle x,s\rangle$ and $y \in R_2(\langle x,s\rangle)$ if and only if $y \in R(x) \wedge \psi(x,s,y)$, where ψ is some polynomial-time computable predicate.

[3] As in Theorem 6.27, if any search problem in $\mathcal{PC}$ is reducible to R via a parsimonious reduction, then we can reduce S_R to $\oplus R$. Specifically, we shall show that S_R is randomly reducible to $\oplus R_2$, for some $R_2 \in \mathcal{PC}$, and a reduction of S_R to $\oplus R$ follows (by using the parsimonious reduction of R_2 to R).

$\{0,1\}^{O(|y|)}$ satisfies $|R_2(\langle x,s\rangle)| \equiv 1 \pmod 2$, whereas for every $x \notin S_R$ and every $s \in \{0,1\}^{O(|y|)}$ it holds that $|R_2(\langle x,s\rangle)| = 0$. A key observation is that $R_2 \in \mathcal{PC}$ (and thus $\oplus R_2$ is in $\oplus\mathcal{P}$). Thus, deciding membership in S_R is randomly reducible to $\oplus R_2$ (by the many-to-one randomized mapping of x to $\langle x,s\rangle$, where s is uniformly selected in $\{0,1\}^{O(|y|)}$). Since the foregoing holds for any $R \in \mathcal{PC}$, it follows that $\mathcal{NP}$ is reducible to $\oplus\mathcal{P}$ via randomized Karp-reductions.

Dealing with co$\mathcal{NP}$. We may Cook-reduce co$\mathcal{NP}$ to $\mathcal{NP}$ and thus prove that co$\mathcal{NP}$ is randomly reducible to $\oplus\mathcal{P}$, but we wish to highlight the fact that a randomized Karp-reduction will also do. Starting with the reduction present for the case of sets in $\mathcal{NP}$, we note that for $S \in$ co$\mathcal{NP}$ we obtain a relation R_2 such that $x \in S$ is indicated by $|R_2(\langle x,\cdot\rangle)| \equiv 0 \pmod 2$. We wish to flip the parity such that $x \in S$ will be indicated by $|R_2(\langle x,\cdot\rangle)| \equiv 1 \pmod 2$, and this can be done by augmenting the relation R_2 with a single dummy solution per each x. For example, we may redefine $R_2(\langle x,s\rangle)$ as $\{0y : y \in R_2(\langle x,s\rangle)\} \cup \{10^{p(|x|)}\}$. Indeed, we have demonstrated and used the fact that $\oplus\mathcal{P}$ is closed under complementation.

We note that dealing with the cases of $\mathcal{NP}$ and co$\mathcal{NP}$ is of interest only because we reduced these classes to $\oplus\mathcal{P}$ rather than to $\#\mathcal{P}$. In contrast, even a reduction of Σ_2 to $\#\mathcal{P}$ is of interest, and thus the reduction of Σ_2 to $\oplus\mathcal{P}$ (presented in Step 3) is interesting. This reduction relies heavily on the fact that error-reduction is applicable in the context of randomized Karp-reductions to $\oplus\mathcal{P}$.

Step 2: error reduction. An important observation, towards the core of the proof, is that it is possible to drastically reduce the (one-sided) error probability in randomized Karp-reductions to $\oplus\mathcal{P}$. Specifically, let R_2 be as in Eq. (F.1) and t be any polynomial. Then, a binary relation R_2' that satisfies

$$|R_2'(\langle x,s_1,...,s_{t(|x|)}\rangle)| \;=\; 1+\prod_{i=1}^{t(|x|)}(1+|R_2(\langle x,s_i\rangle)|) \tag{F.2}$$

offers such an error reduction, because $|R_2'(\langle x,s_1,...,s_{t(|x|)}\rangle)|$ is odd if and only if for some $i \in [t(|x|)]$ it holds that $|R_2(\langle x,s_i\rangle)|$ is odd. Thus,

$$\begin{aligned}\mathsf{Pr}_{s_1,...,s_{t(|x|)}}&[|R_2'(\langle x,s_1,...,s_{t(|x|)}\rangle)| \equiv 0 \pmod 2]\\ &= \mathsf{Pr}_s[|R_2(\langle x,s\rangle)| \equiv 0 \pmod 2]^{t(|x|)}\end{aligned}$$

where $s,s_1,....,s_{t(|x|)}$ are uniformly and independently distributed in $\{0,1\}^{O(p(|x|))}$ (and p is such that $R(x) \subseteq \{0,1\}^{p(|x|)}$). This means that the one-sided error probability of a randomized reduction of S_R to $\oplus R_2$ (which maps x to $\langle x,s\rangle$) can be reduced by reducing S_R to $\oplus R_2'$, where the reduction maps x to $\langle x,s_1,...,s_{t(|x|)}\rangle$. Specifically (for $S_R \in \mathcal{NP}$), error probability ε (e.g., $\varepsilon = 2/3$) in the case that we desire an "odd outcome" (i.e., $x \in S_R$) is reduced to error probability ε^t, whereas zero error probability in the case of a desired "even outcome" (i.e., $x \in \overline{S}_R$) is preserved. A key question is whether this yields error-reduction for reductions to $\oplus\mathcal{P}$; that is, whether R_2' (as postulated in Eq. (F.2)) can be implemented in $\mathcal{PC}$ (and so imply $\oplus R_2' \in \oplus\mathcal{P}$). The answer is positive, and this can be shown by using a Cartesian product construction (and adding some dummy solutions). For example, let $R_2'(\langle x, s_1, ..., s_{t(|x|)}\rangle)$ consists of tuples $\langle \sigma_0, y_1, ..., y_{t(|x|)}\rangle$ such that either $\sigma_0 = 1$ and $y_1 = \cdots = y_{t(|x|)} = 0^{p(|x|)+1}$ or $\sigma_0 = 0$ and for every $i \in [t(|x|)]$ it holds that $y_i \in (\{0\} \times R_2(\langle x, s_i\rangle)) \cup \{10^{p(|x|)}\}$.

We wish to stress that, when starting with R_2 as in Eq. (F.1), the forgoing process of error-reduction can be used for obtaining error probability that is upper-bounded by $\exp(-q(|x|))$ for any desired polynomial q. The importance of this comment will become clear shortly.

Step 3: the case of Σ_2. With the foregoing preliminaries, we are now ready to handle the case of $S \in \Sigma_2$. By Proposition 3.9, there exists a polynomial p and a set $S' \in \Pi_1 = \mathrm{co}\mathcal{NP}$ such that $S = \{x : \exists y \in \{0,1\}^{p(|x|)} \text{ s.t. } (x, y) \in S'\}$. Using $S' \in \mathrm{co}\mathcal{NP}$, we apply the forgoing reduction of S' to $\oplus\mathcal{P}$ as well as an adequate error-reduction that yields an upper-bound of $\varepsilon \cdot 2^{-p(|x|)}$ on the error probability, where $\varepsilon \le 1/7$ is unspecified at this point. (For the case of Σ_2 the setting $\varepsilon = 1/7$ will do, but for the dealing with Σ_k we will need a much smaller value of $\varepsilon > 0$.) Thus, we obtain a relation $R_2' \in \mathcal{PC}$ such that the following holds: *for every x and $y \in \{0,1\}^{p(|x|)}$, with probability at least $1 - \varepsilon \cdot 2^{-p(|x|)}$ over the random choice of $s' \in \{0,1\}^{O(p(|x|))^2}$, it holds that $x' \stackrel{\rm def}{=} (x, y) \in S'$ if and only if $|R_2'(\langle x', s'\rangle)|$ is odd.*[4] Using a union bound (over all possible $y \in \{0,1\}^{p(|x|)}$), it follows that, with probability at least $1 - \varepsilon$ over the choices of s', it holds that $x \in S$ if and only if there exists a y such that $|R_2'(\langle (x, y), s'\rangle)|$ is odd. Now, as in the treatment of $\mathcal{NP}$, we wish to reduce the latter "existential problem" to $\oplus\mathcal{P}$. That is, we wish to define a relation $R_3 \in \mathcal{PC}$ such that for a randomly selected s the value $|R_3(\langle x, s, s'\rangle)| \bmod 2$ provides an indication to whether or not $x \in S$ (by indicating whether or not there exists a y such that $|R_2'(\langle (x, y), s'\rangle)|$ is odd). Analogously to Eq. (F.1), consider the binary relation

$$I_3 = \{(\langle x, s, s'\rangle, y) : |R_2'(\langle (x, y), s'\rangle| \equiv 1 (\bmod\ 2)\ \wedge G(s, y) = 1\}. \tag{F.3}$$

Indeed, if $x \in S$ then, with probability at least $1 - \varepsilon$ over the random choice of s' and probability at least $1/3$ over the random choice of s, it holds that $|I_3(\langle x, s, s'\rangle)|$ is odd, whereas for every $x \notin S$ and every choice of s it holds that $\mathsf{Pr}_{s'}[|I_3(\langle x, s, s'\rangle)| = 0] \ge 1 - \varepsilon$. (For $\varepsilon \le 1/7$, it follows for every $x \in S$ we have $\mathsf{Pr}_{s,s'}[|I_3(\langle x, s, s'\rangle)| \equiv 1 \ (\bmod\ 2)] \ge (1 - \varepsilon)/3 \ge 2/7$, whereas for every $x \notin S$ we have $\mathsf{Pr}_{s,s'}[|I_3(\langle x, s, s'\rangle)| \equiv 1 \ (\bmod\ 2)] \le \varepsilon \le 1/7$.) Thus, $|I_3(\langle x, \cdot, \cdot\rangle)| \bmod 2$ provides a randomized indication to whether or not $x \in S$, but it is not clear whether I_3 is in $\mathcal{PC}$ (and in fact I_3 is likely not to be in $\mathcal{PC}$). The key observation is that

$$\begin{array}{rcl} |R_3(\langle x, s, s'\rangle)| & \equiv & |I_3(\langle x, s, s'\rangle)| \pmod 2 \\ \text{where} & & R_3(\langle x, s, s'\rangle) \stackrel{\rm def}{=} \{\langle y, z\rangle : (\langle (x, y), s'\rangle, z) \in R_2' \wedge G(s, y) = 1\} \end{array} \tag{F.4}$$

(with $\langle y, z\rangle \in \{0,1\}^{p(|x|)} \times \{0,1\}^{p'(|x|)}$), where Eq. (F.4) is justified by noting that $\oplus_{y,z}(\chi_{y,z} \wedge \xi_y)$ equals $\oplus_y((\oplus_z \chi_{y,z}) \wedge \xi_y)$, where $\chi_{y,z} = 1$ (resp., ξ_y) indicates the

[4] Note that $R_2' \subseteq \{0,1\}^{|x|+p(|x|)+O(p(|x|)^2)} \times \{0,1\}^{p'(|x|)}$, where p' is some polynomial that may depend on p. In particular, the specific implementation of R_2', which uses $t = O(p)$, yields $p' = O(p^2)$.

event $(\langle\langle(x,y),s'\rangle,z\rangle \in R_2'$ (resp., the event $G(s,y)=1$). The punch-line is that $R_3 \in \mathcal{PC}$. It follows that S is randomly Karp-reducible to $\oplus\mathcal{P}$ (by the many-to-one randomized mapping of x to $\langle x,s,s'\rangle$, where (s,s') is uniformly selected in $\{0,1\}^{O(p(|x|))} \times \{0,1\}^{O(p'(|x|))}$).

Again, error-reduction may be applied to this reduction (of Σ_2 to $\oplus\mathcal{P}$) such that it can be used for dealing with Σ_3. A technical difficulty arises since the foregoing reduction has two-sided error probability, where one type (or "side") of error is due to the error in the reduction of $S' \in \mathrm{co}\mathcal{NP}$ to $\oplus R_2'$ (which occurs on no-instances of S') and the second type (or "side") of error is due to the (new) reduction of S to $\oplus R_3$ (and occurs on the yes-instances of S). However, the error probability in the first reduction is (or can be made) very small and can be ignored when applying error-reduction to the second reduction. See following comments.

The general case. First note that, as in the case of $\mathrm{co}\mathcal{NP}$, we can obtain a similar reduction for $\Pi_2 = \mathrm{co}\Sigma_2$. It remains to extend the treatment of Σ_2 to Σ_k, for every $k \geq 2$. Indeed, $S \in \Sigma_k$ is treated by considering a polynomial p and a set $S' \in \Pi_{k-1}$ such that $S = \{x : \exists y \in \{0,1\}^{p(|x|)} \text{ s.t. } (x,y) \in S'\}$. Next, we use a relation R_k' such that, with overwhelmingly high probability over the choice of s' the value $|R_k'(\langle(x,y),s'\rangle)| \bmod 2$ indicates whether or not $(x,y) \in S'$. Using the ideas underlies the treatment of $\mathcal{NP}$ (and Σ_2) we check whether for some y it holds that $|R_k'(\langle(x,y),s'\rangle)| \equiv 1 \pmod 2$. This yields a relation R_{k+1} such that for random s,s' the value $|R_{k+1}(\langle x,s,s'\rangle)| \bmod 2$ indicates whether or not $x \in S$. Finally, we apply error reduction, while ignoring the probability that s' is bad, and obtain the desired relation R_{k+1}'. This means that if we wish to upper-bound the error probability in the reduction (of S) to $\oplus R_{k+1}'$ by ε_{k+1}, then the error probability in the reduction (of S') to $\oplus R_k'$ should be upper-bounded by $\varepsilon_k = \varepsilon_{k+1} \cdot 2^{-p(|x|)}$. Thus, the proof that $\mathcal{PH}$ is randomly reducible to $\oplus\mathcal{P}$ actually proceed "top down" (at least partially); that is, starting with an arbitrary $S \in \Sigma_k$, we first determine the auxiliary sets (as per Proposition 3.9) as well as the error-bounds that should be proved for the reductions of these sets (which reside in lower levels of $\mathcal{PH}$), and only then we establish the existence of such reductions. Indeed, this latter (and main) step is done "bottom up" using the reduction (to $\oplus\mathcal{P}$) of the set in the i^{th} level when reducing (to $\oplus\mathcal{P}$) the set in the $i+1^{\text{st}}$ level. □

F.2 Proving that $\mathcal{IP}(f) \subseteq \mathcal{AM}(O(f)) \subseteq \mathcal{AM}(f)$

Using the notations presented in §9.1.3.3. we restate two results mentioned there.

Theorem F.2 (Round-efficient emulation of $\mathcal{IP}$ by $\mathcal{AM}$): *Let $f : \mathbb{N}\to\mathbb{N}$ be a polynomially bounded function. Then $\mathcal{IP}(f) \subseteq \mathcal{AM}(f+3)$.*

We comment that, in light of the following linear speed-up in round-complexity for $\mathcal{AM}$, it suffices to establish $\mathcal{IP}(f) \subseteq \mathcal{AM}(O(f))$.

Theorem F.3 (Linear speed-up for $\mathcal{AM}$): *Let $f : \mathbb{N}\to\mathbb{N}$ be a polynomially bounded function. Then $\mathcal{AM}(2f) \subseteq \mathcal{AM}(f+1)$.*

Combining these two theorems, we obtain a linear speed-up for $\mathcal{IP}$; that is, for any polynomially bounded $f : \mathbb{N} \to (\mathbb{N} \setminus \{1\})$, it holds that $\mathcal{IP}(O(f)) \subseteq \mathcal{AM}(f) \subseteq \mathcal{IP}(f)$. In this appendix we prove both theorems.

We mention that the proof of Theorem F.2 relies on the fact that, for every f, error-reduction is possible for $\mathcal{IP}(f)$. Specifically, error-reduction can be obtained via *parallel* repetitions (see [86, Apdx. C.1]). We note that error-reduction (in the context of $\mathcal{AM}(f)$) is implicit also in the proof of Theorem F.3 (and is explicit in the original proof of [20]).

F.2.1 Emulating general interactive proofs by AM-games

In this section we prove Theorem F.2. Our proof differs from the original proof of Goldwasser and Sipser [107] only in the conceptualization and implementation of the iterative emulation process.

F.2.1.1 The basic approach

Our aim is to transform a general interactive proof system (P, V) into a public-coin interactive proof system for the same problem. Suppose, without loss of generality, that P constitutes an optimal prover with respect to V (i.e., P maximizes the acceptance probability of V on any input). Then, for any yes-instance, the set of coin sequences that make V accept when interacting with this optimal prover contains all possible outcomes, whereas for a no-instance this set is very small. The idea is having a public-coin system in which the prover prove to the verifier that the said set is big. Such a proof system can be constructed using ideas as in the case of approximate counting (see the proof of Theorem 6.25), while replacing the NP-oracle with a prover that is required to prove the correctness of its answers. Implementing this idea requires taking a closer look at the set of coin sequences that make V accept an input.

We demonstrate the implementation of the foregoing approach by considering an interactive proof system (such as Construction 9.3) in which the verifier V sends a single message to which the prover P responses. Further suppose that, when the common input is a yes-instance, each possible message of V is equally likely (which holds for a minor modification of Construction 9.3).[5] Specifically, suppose that on input x, the verifier V tosses $\ell = \ell(|x|)$ coins and sends one out of M possible messages (as determined by the input and the coin sequence). Then, in the *public-coin system*, the prover will claim that in the original proof there are M possible V-messages such that the original prover can respond to each of them in a way that is accepted by $2^\ell/M$ corresponding coin sequences of V. To prove this claim, the prover lets the verifier select at random one of the possible M messages (e.g., by selecting coins for V), denoted α, and the prover send back an adequate P-message, denoted β, and proves that β would have been accepted by $2^\ell/M$ possible coin sequences of V. The latter proof follows the idea of the proof

[5]In the original protocol, the verifier selects at random one of the two input graphs, and sends a random isomorphic copy of it. In the modification, the verifier creates a random isomorphic copy of each of the two input graphs, and sends them in a random order.

of Theorem 6.25: The verifier applies a random sieve that lets only a $(2^\ell/M)^{-1}$ fraction of the elements pass, and the prover proves that some adequate sequence of V-coins has passed this sieve. The latter claim is proved by merely presenting such a sequence, denoted r, and the verifier can check whether indeed r passes the sieve as well as fits the initial message α and would have made V accept the prover message β (i.e., V would have accepted the input, on coins r, when receiving the prover message β). We stress that the foregoing interaction (and in particular the random sieve) can be implemented in the public-coin model.

A few technical problems arise. Firstly, recall that the random sieve only allows for an approximation of set sizes. However, since the gap between the acceptance probability of yes-instances and no-instances is big enough (or can be made big enough by parallel repetition), this suffices. Secondly, in general, it is not necessarily the case that each possible message of V is equally likely. However, the prover may cluster the V-messages into few (say ℓ) clusters such that the messages in each cluster are sent (by V) with roughly the same probability (say, up to a factor of two). Focusing on the cluster having the largest probability weight, the prover can proceed as in the simple case. This has a potential of cutting the probabilistic gap[6] between yes-instances and no-instances by a factor related to the number of clusters times the approximation level within clusters (e.g., a factor of $O(\ell)$), but this loss is negligible in comparison to the initial gap (which can be obtained via error-reduction). Lastly, there is the fact that we only dealt with a two-message system (i.e., $\mathcal{IP}(2)$).

It is tempting to say that the general case of $\mathcal{IP}(f)$ can be dealt by recursion (or rather iterations), and indeed this is almost the case. Recall that our treatment of the case of $\mathcal{IP}(2)$ boils down to having the verifier choose a random V-message, α, and having the prover send a P-response, β, and finally prove that β is acceptable by many V-coins. In other words, the prover should prove that in the conditional probability space defined by V-message α, the original verifier V accepts with high probability. In the general case (of $\mathcal{IP}(f)$), the latter claim refers to the probability of accepting in the residual interaction, which consists of $f-2$ messages, and thus the very same protocol can be applied iteratively (until we get to the last message, which is dealt as in the case of $\mathcal{IP}(2)$). The only problem is that in the residual interactions, it may not be easy for the verifier to select a random V-message (as done in the case of $\mathcal{IP}(2)$). Instead, the verifier will be assisted by the prover, while making sure that it is not being fooled by the prover. Indeed, this calls for an adequate "random selection" protocol, which need to be implemented in the public-coin model. For simplicity, we may consider the problem of selecting a uniform sequence of coins in the residual probability space, because such a sequence determines the desired random V-message.

F.2.1.2 Random selection

Various types of "random selection" protocols have appeared in the literature (see, e.g., [211, Sec. 6.4]). The common theme in these protocols is that they allow

[6]The point is that in one case all clusters may have equal weight, and thus a corresponding factor is lost, while in the other case all probability mass may be concentrated in a single cluster.

for a probabilistic polynomial-time player (called the *verifier*) to sample a set, denoted $S \subset \{0,1\}^\ell$, while being assisted by a second player (called the *prover*) that is powerful but not trustworthy. These nicknames fit the common conventions regarding interactive proofs and are further justified by the typical applications of such protocols as subroutines within an interactive proof system (where indeed the first party is played by the higher-level verifier while the second party is played by the higher-level prover). The various types of random selection protocols differ by what is known about the set S and what is required from the protocol.

Here we will assume that the verifier is given a parameter N, which is supposed to equal $|S|$, and the performance guarantee of the protocol will be meaningful only for sets of size at most N. We desire a constant-round (preferably two-round) public-coin protocol for this setting such that the following holds, with respect to a security parameter $\varepsilon = 1/\text{poly}(\ell)$.

1. If both players are honest and $N = |S|$ then the verifier's output is ε-close to the uniform distribution over S. Furthermore, the verifier always outputs an element of S.

2. For any set $S' \subseteq \{0,1\}^\ell$ if the verifier follows the protocol then, no matter how the prover behaves, the verifier's output resides in S' with probability at most $\text{poly}(\ell/\varepsilon) \cdot (|S'|/N)$.

Note that the second property is meaningful only for sets S' of size (significantly) smaller than N.

A three-round public-coin protocol can be obtained by using the ideas that underly Construction 6.30. Specifically, we set $m' = \max(0, \log_2 N - O(\log \ell/\varepsilon))$ in order to guarantee that if $|S| = N$ then, with overwhelmingly high probability, each cell defined by the hashing function contains $(1 \pm \varepsilon) \cdot |S|/2^{m'}$ elements of S. In the protocol, the prover selects a good hashing function (i.e., one defining such a good partition of S) and sends it to the verifier, which answers with a uniformly selected cell, to which the prover responds with a uniformly selected element of S that resides in this cell.[7]

Note that this protocol satisfies the aforementioned properties. In particular, the second property follows because for every possible hashing function, the fraction of cells containing an element of S' is at most $|S'|/2^{m'}$, which is upper-bounded by $\text{poly}(\ell/\varepsilon) \cdot |S'|/N$. We stress that the protocol is indeed in the public-coin model, and comment that the fact that it uses three messages rather than two will have a minor effect on our application.

[7] A more natural version of this protocol consists of having the verifier select at random a hashing function as well as a cell, and asks the prover for a list of $(1-\varepsilon) \cdot N/2^{m'}$ elements in this cell. The verifier then outputs an element that is uniformly selected in the list. This protocol provides a stronger guarantee with respect to cheating provers: the verifier's output resides in S' with probability at most $(|S'|/N) + \varepsilon$. However, even in case the prover is honest, this protocol does not guarantee that the verifier always outputs an element of S, because it may happen (rarely) that the hashing function selected by the verifier is not good. For this reason, we preferred the version presented in the main text.

F.2.1.3 The iterated partition protocol

The random selection protocol discussed in §F.2.1.2 is meaningful only with respect to sets (i.e., S') that are smaller than the given parameter N. Here we explain why this suffices for our goals. We start with some notations.

Fixing any input x to (P, V), we denote by $t = t(|x|)$ the number of pairs of communication rounds (assuming that the verifier takes the first move in (P, V))[8] and by $\ell = \ell(|x|) > t$ the number of coins tossed by V. Recall that we assume that P is an optimal prover (with respect to V), and that (without loss of generality) P is deterministic. Let us denote by $\langle P, V(r)\rangle(x)$ the full transcript of the interaction of P and V on input x, when V uses coins r; that is, $\langle P, V(r)\rangle(x) = (\alpha_1, \beta_1, ..., \alpha_t, \beta_t, \sigma)$ if $\sigma = V(x, r, \beta_1, ..., \beta_t) \in \{0,1\}$ is V's final verdict and for every $i = 1, ..., t$ it holds that $\alpha_i = V(x, r, \beta_1, ..., \beta_{i-1})$ and $\beta_i = P(x, \alpha_1, ..., \alpha_i)$. For any partial transcript ending with a P-message, $\gamma = (\alpha_1, \beta_1, ..., \alpha_{i-1}, \beta_{i-1})$, we denote by $\mathrm{ACC}_x(\gamma)$ the set of coin sequences that are consistent with the partial transcript γ and lead V to accept x when interacting with P; that is, $r \in \mathrm{ACC}_x(\gamma)$ if and only if for some γ' it holds that $\langle P, V(r)\rangle(x) = (\alpha_1, \beta_1, ..., \alpha_{i-1}, \beta_{i-1}, \gamma', 1)$. The same notation is also used for a partial transcript ending with a V-message; that is, $r \in \mathrm{ACC}_x(\alpha_1, \beta_1, ..., \alpha_i)$ if and only if $\langle P, V(r)\rangle(x) = (\alpha_1, \beta_1, ..., \alpha_i, \gamma', 1)$ for some γ'.

Motivation. By suitable error reduction, we may assume that (P, V) has soundness error $\mu = \mu(|x|)$ that is smaller than $\mathrm{poly}(\ell)^{-t}$. Thus, for any yes-instance x it holds that $|\mathrm{ACC}_x(\lambda)| = 2^\ell$, whereas for any no-instance x it holds that $|\mathrm{ACC}_x(\lambda)| \leq \mu \cdot 2^\ell$. Indeed, the gap between the set sizes is huge, and it will be preserved as long as we lose at most a factor of $\mathrm{poly}(\ell)$ per each round. The key observations is that, for any partial transcript $\gamma = (\alpha_1, \beta_1, ..., \alpha_{i-1}, \beta_{i-1})$, it holds that

$$|\mathrm{ACC}_x(\gamma)| = \sum_\alpha |\mathrm{ACC}_x(\gamma, \alpha)|, \tag{F.5}$$

whereas $|\mathrm{ACC}_x(\gamma, \alpha)| = \max_\beta\{|\mathrm{ACC}_x(\gamma, \alpha, \beta)|\}$. Clearly, we can prove that $|\mathrm{ACC}_x(\gamma, \alpha)|$ is big by providing an adequate β and proving that $|\mathrm{ACC}_x(\gamma, \alpha, \beta)|$ is big. Likewise, proving that $|\mathrm{ACC}_x(\gamma)|$ is big reduces to proving that the sum $\sum_\alpha |\mathrm{ACC}_x(\gamma, \alpha)|$ is big. The problem is that this sum may contain exponentially many terms, and so we cannot even afford reading the value of each of these terms.[9] As hinted in §F.2.1.1, we may cluster these terms into ℓ clusters, such that the j^{th} cluster contains sets of cardinality approximately 2^j (i.e., α's such that $2^j \leq |\mathrm{ACC}_x(\gamma, \alpha)| < 2^{j+1}$). One of these clusters must account for a $1/2\ell$ fraction of the claimed size of $|\mathrm{ACC}_x(\gamma)|$, and so we focus on this cluster; that is, the prover we construct will identify a suitable j and prove that there are at least $N = |\mathrm{ACC}_x(\gamma)|/(2\ell \cdot 2^{j+1})$ sets (i.e., the $\mathrm{ACC}_x(\gamma, \alpha)$ sets) each of size at least 2^j. Note that this establishes that $|\mathrm{ACC}_x(\gamma)|$

[8] We note if the prover takes the first move in (P, V) then its first message can be emulated with no cost (in the number of rounds).

[9] Furthermore, we cannot afford verifying more than a single claim regarding the value of one of these terms, because examining at least two values per round will yield an exponential blow-up (i.e., time complexity that is exponential in the number of rounds).

is bigger than $N \cdot 2^j = |\mathrm{ACC}_x(\gamma)|/O(\ell)$, which means that we lost a factor of $O(\ell)$ of the size of $\mathrm{ACC}_x(\gamma)$. But as stated before, we may afford such a lost.

Before we turn to the actual protocol, let us discuss the method of proving that that there are at least N sets (i.e., $\mathrm{ACC}_x(\gamma, \alpha)$'s) each of size at least 2^j. This claim is proved by employing the random selection protocol (with size parameter set to N) with the goal of selecting such a set (or rather its index α). If indeed N such sets exists then the first property of the protocol guarantees that such a set is always chosen, and we will proceed to the next iteration with this set, which has size at least 2^j (and so we should be able to establish a corresponding lower-bound there). Thus,, entering the current iteration with a valid claim, we proceed to the next iteration with a new valid claim. On the other hand, suppose that $|\mathrm{ACC}_x(\gamma)| \ll N \cdot 2^j$. Then, the second property of the protocol guarantees that, with probability at least $1 - (1/3t)$, the selected α is such that $|\mathrm{ACC}_x(\gamma, \alpha)| < \mathrm{poly}(\ell) \cdot |\mathrm{ACC}_x(\gamma)|/N \ll 2^j$, whereas at the next iteration we will need to prove that the selected set has size at least 2^j. Thus, entering the current iteration with a false claim that is wrong by a factor F, with probability at least $1 - (1/3t)$, we proceed to the next iteration with a claim that is wrong by a factor of at least $F/\mathrm{poly}(\ell)$.

We note that, although the foregoing motivational discussion refers to proving lower-bounds on various set sizes, the actual implementation refers to randomly selecting elements in such sets. If the sets are smaller than claimed, the selected elements are likely to reside outside these sets, which will be eventually detected.

Construction F.4 (the actual protocol). *On common input x, the $2t$-round interaction of P and V is* "quasi-emulated" *in t iterations, where $t = t(|x|)$. The i^{th} iteration starts with a partial transcript $\gamma_{i-1} = (\alpha_1, \beta_1, ..., \alpha_{i-1}, \beta_{i-1})$ and a claimed bound M_{i-1}, where in the first iteration γ_0 is the empty sequence and $M_0 = 2^\ell$. The i^{th} iteration proceeds as follows.*

1. *The prover determines an index j such that the cluster $C_j = \{\alpha : 2^j \leq |\mathrm{ACC}_x(\gamma_{i-1}, \alpha)| < 2^{j+1}\}$ has size at least $N \stackrel{\text{def}}{=} M_{i-1}/(2^{j+2}\ell)$, and sends j to the verifier. Note that if $|\mathrm{ACC}_x(\gamma_{i-1})| \geq M_{i-1}$ then such a j exists.*

2. *The prover invokes the random selection protocol with size parameter N in order to select $\alpha \in C_j$, where for simplicity we assume that $C_j \subseteq \{0,1\}^\ell$. Recall that this public-coin protocol involves three messages with the first and last message being sent by the prover. Let use denote the outcome of this protocol by α_i.*

3. *The prover determines β_i such that $\mathrm{ACC}_x(\gamma_{i-1}, \alpha_i, \beta_i) = \mathrm{ACC}_x(\gamma_{i-1}, \alpha_i)$ and sends β_i to the verifier.*

   Towards the next iteration $M_i \leftarrow 2^j$ and $\gamma_i = (\alpha_1, \beta_1, ..., \alpha_i, \beta_i) \equiv (\gamma_{i-1}, \alpha_i, \beta_i)$.

After the last iteration,[10] *the prover invokes the random selection protocol with size parameter $N = M_t$ in order to select $r \in \mathrm{ACC}_x(\alpha_1, \beta_1, ..., \alpha_t, \beta_t)$. Upon obtain-*

[10] Alternatively, we may modify (P, V) by adding a last V-message in which V sends its internal coin tosses (i.e., r). In this case, the additional invocation of the random selection protocol occurs as a special case of handling the added $t + 1^{\text{st}}$ iteration.

ing this r, the verifier accepts if and only if $V(x, r, \beta_1, ..., \beta_t) = 1$ and for every $i = 1, ..., t$ it holds that $\alpha_i = V(x, r, \beta_1, ..., \beta_{i-1})$, where the α_i's and β_i's are as determined in the aforementioned iterations.

Note that the three steps of each iteration involve a single message by the public-coin verifier, and thus the foregoing protocol can be implemented using $2t + 3$ messages.

Clearly, if x is a yes-instance then the prover can make the verifier accept with probability one (because an adequately large cluster exists at each iteration, and the random selection protocol guarantees that the selected α_i will reside in this cluster). Thus, at the last invocation of the random selection protocol, the verifier always obtains $r \in \mathrm{ACC}_x(\gamma_t)$ and accepts. On the other hand, if x is a no-instance then by using the low soundness error of (P, V) we can establish the soundness of Construction F.4. This is proved in the *following claim, which refers to a polynomial p that is sufficiently large.*

Claim F.5 *Suppose that $|\mathrm{ACC}_x(\lambda)| < \delta^{t+1} \cdot 2^\ell$, where $\delta = 1/p(\ell)$. Then, the verifier of Construction F.4 accepts x with probability smaller than* $1/2$.

Proof Sketch: We first prove that, for every $i = 1, ..., t$, if $|\mathrm{ACC}_x(\gamma_{i-1})| < \delta^{t+1-(i-1)} \cdot M_{i-1}$ then, with probability at least $1-(1/3t)$, it holds that $|\mathrm{ACC}_x(\gamma_i)| < \delta^{t+1-i} \cdot M_i$. Let j be the value selected by the prover in Step 1 (of iteration i), and define $S' = \{\alpha : |\mathrm{ACC}_x(\gamma_{i-1}, \alpha)| \geq \delta^{t+1-i} \cdot 2^j\}$. Then $|S'| \cdot \delta^{t+1-i} 2^j < \delta^{t+1-(i-1)} \cdot M_{i-1}$, and so $|S'| < \delta \cdot (M_{i-1}/2^j) = 4\ell\delta \cdot N$, where $N = M_{i-1}/(2^{j+2}\ell)$ is as used in Step 2 (of this iteration). By the second property of the random selection protocol it follows that $\mathsf{Pr}[\alpha_i \in S] \leq \mathrm{poly}(\ell) \cdot \delta = \mathrm{poly}(\ell)/p(\ell)$, which is smaller than $1/3t$ provided that the aforementioned polynomial p is sufficiently large. Thus, with probability at least $1 - (1/3t)$, it holds that $|\mathrm{ACC}_x(\gamma_{i-1}, \alpha_i)| < \delta^{t+1-i} \cdot 2^j$. The i^{th} claim follows by recalling that $M_i = 2^j$ (in Step 3) and that for every β it holds that $|\mathrm{ACC}_x(\gamma_{i-1}, \alpha_i, \beta)| \leq |\mathrm{ACC}_x(\gamma_{i-1}, \alpha_i)|$.

Recalling that $|\mathrm{ACC}_x(\gamma_0)| < \delta^{t+1} \cdot M_0$, with probability at least $2/3$, we have $|\mathrm{ACC}_x(\gamma_t)| < \delta \cdot M_t$. In this case, the random selection protocol produces an element of $\mathrm{ACC}_x(\gamma_t)$ with probability at most $1/6$, and the verifier rejects otherwise (because the conditions that the verifier checks regarding the output r of the random selection protocol are logically equivalent to $r \in \mathrm{ACC}_x(\gamma_t)$). The main claim follows. □

F.2.2 Linear speed-up for $\mathcal{AM}$

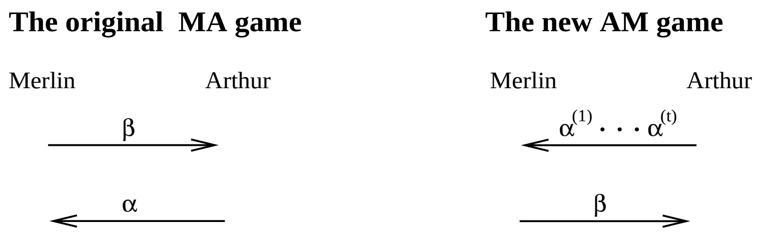
In this section we prove Theorem F.3. Our proof differs from the original proof of Babai and Moran [20] in the way we analyze the basic switch (of MA to AM).

We assume that the reader is familiar with the terminology of public-coin (a.k.a Arthur-Merlin) interactive proofs, where the verifier is called Arthur and the prover is called Merlin. The *execution* of such a proof system, on any fixed common input x, can be viewed as a game (indexed by x) between an honest Arthur and powerful Merlin. These parties alternate in taking moves such that Arthur takes random moves and Merlin takes optimal moves with respect to a fixed (polynomial-time computable) predicate v_x that is *evaluated on the full transcript of the game's execution.* The value of the game is defined as the expected value of an execution of the game, where the expectation is taken over Arthur's moves (and Merlin's moves are assumed to be optimal).

Recall that $\mathcal{AM} = \mathcal{AM}(2)$ denotes a two-round system in which Arthur moves first and does not toss coins after receiving Merlin's answer, whereas $\mathcal{MA} = \mathcal{AM}(1)$ denotes a one-round system in which Merlin sends a single message and Arthur tosses additional coins after receiving this message. We may assume, without loss of generality, that all messages of Arthur are of the same length, denoted $\ell = \ell(|x|)$. Similarly, each of Merlin's messages is of length $m = m(|x|)$.

F.2.2.1 The basic switch (from MA to AM)

The basic idea is to transform an MA-game (i.e., a two-move game in which Merlin moves first and Arthur follows) into an AM-game (in which Arthur moves first and Merlin follows). Recall that, in the original game, first Merlin sends a message $\beta \in \{0,1\}^m$, then Arthur responds with a random $\alpha \in \{0,1\}^\ell$, and the value of this execution of the game is given by $v_x(\beta, \alpha) \in \{0,1\}$. In the new game (see Figure F.1), the order of these moves will be switched, but to limit Merlin's potential gain from the switch we require it to provide a single answer that should "fit" several random messages of Arthur. That is, for a parameter t to be specified, first Arthur send a random sequence $(\alpha^{(1)}, ..., \alpha^{(t)}) \in \{0,1\}^{t \cdot \ell}$, then Merlin responds with a string $\beta \in \{0,1\}^m$, and the value of this transcript of the new game is defined as the conjunction of the values $v_x(\beta, \alpha^{(i)})$, for $i = 1, ..., t$. Intuitively, Merlin gets the advantage of choosing its move after seeing Arthur's move(s), but Merlin's choice must fit the t choices of Arthur's move, which intuitively leaves Merlin with little gain (if t is sufficiently large).



The value of the transcript (β, α) of the original MA-game is given by $v_x(\beta, \alpha)$, whereas the value of the transcript $((\alpha^{(1)}, ..., \alpha^{(t)}), \beta)$ of the new AM-game is given by $\wedge_{i=1}^t v_x(\beta, \alpha^{(i)})$.

Figure F.1: The transformation of an MA-game into an AM-game.

Recall that the value, v'_x, of the transcript $(\overline{\alpha}, \beta)$ of the new game, where $\overline{\alpha} = (\alpha^{(1)}, ..., \alpha^{(t)})$, is defined as $\wedge_{i=1}^t v_x(\beta, \alpha^{(i)})$. Thus, the value of the new game is defined as

$$\mathsf{E}_{\overline{\alpha}}[\max_{\beta}\{\wedge_{i=1}^{t} v_x(\beta,\alpha^{(i)})\}], \tag{F.6}$$

which is upper-bounded by

$$\mathsf{E}_{\overline{\alpha}}\left[\max_{\beta}\left\{\frac{1}{t}\sum_{i=1}^{t} v_x(\beta,\alpha^{(i)})\right\}\right]. \tag{F.7}$$

Note that the upper-bound provided in Eq. (F.7) is tight in the case that the value of the original MA-game equals one (i.e., if x is a yes-instance), and that in this case the value of the new game is one (because in this case there exists a move β such that $v_x(\beta,\alpha) = 1$ holds for every α). However, the interesting case, where Merlin may gain something by the switch is when the value of the original MA-game is strictly smaller than one (i.e., when x is a no-instance). For this case, we upper-bound the probability that Merlin can obtain a significant gain by selecting β based on the sequence $\overline{\alpha}$ chosen by Arthur (in the new game, rather than obliviously of Arthur's move as in the original game). Specifically, we upper-bound the probability that Merlin's gain from the switch exceeds a parameter δ as follows.

$$\begin{aligned}
p_{x,\delta} &\stackrel{\text{def}}{=} \mathsf{Pr}_{(\alpha^{(1)},\ldots,\alpha^{(t)})}\left[\max_{\beta}\left\{\frac{1}{t}\cdot\sum_{i=1}^{t} v_x(\beta,\alpha^{(i)})\right\} \geq \max_{\beta}\{\mathsf{E}_\alpha(v_x(\beta,\alpha))\} + \delta\right] \\
&\leq \mathsf{Pr}_{(\alpha^{(1)},\ldots,\alpha^{(t)})}\left[\exists\beta\in\{0,1\}^m \text{ s.t. } \left|\frac{1}{t}\cdot\sum_{i=1}^{t} v_x(\beta,\alpha^{(i)}) - \mathsf{E}_\alpha(v_x(\beta,\alpha))\right| \geq \delta\right] \\
&\leq 2^m\cdot\exp(-\Omega(\delta^2\cdot t)),
\end{aligned}$$

where the last inequality is due to combining the Union Bound with Chernoff Bound. Denoting by $V_x = \max_\beta\{\mathsf{E}_\alpha(v_x(\beta,\alpha))\}$ the value of the original game, we upper-bound Eq. (F.7) by $p_{x,\delta} + V_x + \delta$. Using $t = O((m+k)/\delta^2)$ we have $p_{x,\delta} \leq 2^{-k}$, and thus

$$V_x' \stackrel{\text{def}}{=} \mathsf{E}_{\overline{\alpha}}\left[\max_{\beta}\left\{\frac{1}{t}\sum_{i=1}^{t} v_x(\beta,\alpha^{(i)})\right\}\right] \leq \max_{\beta}\{\mathsf{E}_\alpha(v_x(\beta,\alpha))\} + \delta + 2^{-k}. \tag{F.8}$$

Needless to say, Eq. (F.7) is lower-bounded by V_x (since Merlin may just use the optimal move of the MA-game). In particular, using $\delta = 2^{-k} = 1/8$ and assuming that $V_x \leq 1/4$, we obtain $V_x' < 1/2$. Thus, starting from an MA proof system for some set, we obtain an AM proof system for the same set; that is, we just proved that $\mathcal{MA} \subseteq \mathcal{AM}$.

Extension. We note that the foregoing transformation as well as its analysis does not refer to the fact that $v_x(\beta,\alpha)$ is efficiently computable from (β,α). Furthermore, it does not refer to the fact that $v_x(\beta,\alpha)$ is in $\{0,1\}$. Thus, we may apply the foregoing transformation to the any two consecutive Merlin-Arthur moves in any public-coin interactive proof, provided that all subsequent moves are performed in t copies, where each copy corresponds to a different $\alpha^{(i)}$ used in the switch. That is, if the j^{th} move is by Merlin then we can switch the players in the j and $j+1$ moves, by letting Arthur take the j^{th} move, sending $(\alpha^{(1)}, ..., \alpha^{(t)})$, followed by Merlin's move, answering β. Subsequent moves will be played in t copies such that the i^{th} copy corresponds to the moves $\alpha^{(i)}$ and β. The value of the new game may increase by at most $2^{-k} + \delta < 1/4$, and so we obtain an "equivalent" game with the two steps switched. Schematically, acting on the middle MA (indicated in bold font), we can replace $[\mathrm{AM}]^{j_1}\mathrm{A}\mathbf{MA}[\mathrm{MA}]^{j_2}$ by $[\mathrm{AM}]^{j_1}\mathrm{A}\mathbf{AM}[\mathrm{MA}]^{j_2}$, which (for $j_2 \geq 1$) may be written as $[\mathrm{AM}]^{j_1}\mathrm{AAMMA}[\mathrm{MA}]^{j_2-1} = [\mathrm{AM}]^{j_1}\mathrm{AMA}[\mathrm{MA}]^{j_2-1}$ (and for $j_2 = 0$ yields $[\mathrm{AM}]^{j_1}\mathrm{AAM} = [\mathrm{AM}]^{j_1}\mathrm{AM}$). In particular, we get $\mathrm{A}[\mathrm{MA}]^{j+1} = \mathrm{A}[\mathrm{MA}]^{j} = \cdots = \mathrm{AMA} = \mathrm{AM}$. Thus, for any constant f, we get $\mathcal{AM}(f) = \mathcal{AM}(2)$.

We stress that the foregoing switching process can be obtained only a constant number of times, because each time we apply the switch the length of messages increases by a factor of $t = \Omega(m)$. Thus, a different approach is required to deal with a non-constant number of messages (i.e., unbounded function f).

F.2.2.2 The augmented switch (from $[MAMA]^j$ to $[AMA]^j A$)

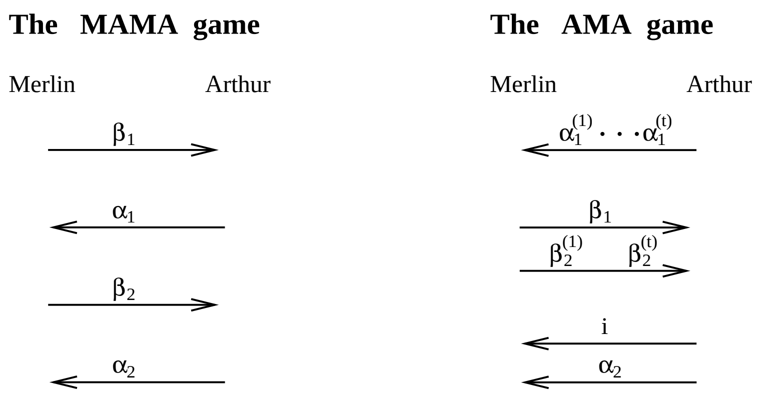
Sequential applications of the "MA-to-AM switch" allows for reducing the number of rounds by any additive constant. However, each time this switch is applied, all subsequent moves are performed t times (in parallel). That is, the "MA-to-AM switch" splits the rest of the game to t independent copies, and thus this switch cannot be performed more than a constant number of times. Fortunately, Eq. (F.7) suggests a way of shrinking the game back to a single copy: just have Arthur select $i \in [t]$ uniformly and have the parties continue with the i^{th} copy.[11] In order to avoid introducing an Arthur-Merlin alternation, the extra move of Arthur is postpone to after the following move of Merlin (see Figure F.2). Schematically (indicating the action by bold font), we replace **MA**MA by **AM**MA**A**=**AMA** (rather than replacing **MA**MA by **AMA**MA and obtaining no reduction in the number of move-alternations).

The value of game obtained via the aforementioned augmented switch is given by Eq. (F.7), which can be written as

$$\mathsf{E}_{\alpha^{(1)},\ldots,\alpha^{(t)}}[\max_{\beta}\{\mathsf{E}_{i\in[t]}(v_x(\beta,\alpha^{(i)}))\}],$$

which in turn is upper-bounded (in Eq. (F.8)) by $V_x + \delta + 2^{-k}$. As in §F.2.2.1, the argument applies to any two consecutive Merlin-Arthur moves in any public-coin interactive proof. Recall that in order to avoid the introduction of an extra Arthur move, we actually postpone the last move of Arthur to after the next move of Merlin. Thus, we apply the augmented switch to any block of four consecutive moves that start with a Merlin move, transforming the schematic sequence MAMA into AMMAA=AMA (see Figure F.2). The key point is that the moves that take place after the said block remain intact. Thus, we may apply the augmented

[11] Indeed, the relaxed form of Eq. (F.7) plays a crucial role here (in contrast to Eq. (F.6)).



The value of the transcript $(\beta_1, \alpha_1, \beta_2, \alpha_2)$ *of the original MAMA-game is given by* $v_x(\beta_1, \alpha_1, \beta_2, \alpha_2)$*, whereas the value of the transcript* $((\alpha_1^{(1)}, ..., \alpha_1^{(t)}), (\beta_1, \beta_2^{(1)}, ..., \beta_2^{(t)}), (i, \alpha_2))$ *of the new AMA-game is given by* $v_x(\beta_1, \alpha_1^{(i)}, \beta_2^{(i)}, \alpha_2)$.

Figure F.2: The transformation of MAMA into AMA.

"MA-to-AM switch" (which is actually an "MAMA-to-AMA switch") concurrently to disjoint segments of the game. Schematically, we can replace $[\mathrm{MAMA}]^j$ by $[\mathrm{AMA}]^j = A[\mathrm{MA}]^j$. Note that Merlin's gain from each such switch is upper-bounded by $\delta + 2^{-k}$, but selecting $t = \widetilde{O}(f(|x|)^2 \cdot m(|x|)) = \mathrm{poly}(|x|)$ allows to upper-bound the total gain by a constant (using, say, $\delta = 2^{-k} = 1/8f(|x|)$). We thus obtain $\mathcal{AM}(4f) \subseteq \mathcal{AM}(2f+1)$, and Theorem F.3 follows.

Appendix G

Some Computational Problems

Although we view specific (natural) computational problems as secondary to (natural) complexity classes, we do use the former for clarification and illustration. This appendix provides definitions of these computational problems.

We start by addressing the central issue of the representation of the various objects. The general principle is that elements of all sets are "compactly" represented as binary strings (without much redundancy). For example, the elements of a finite set S (e.g., the set of vertices in a graph or the set of variables appearing in a Boolean formula) will be represented as binary strings of length $\log_2 |S|$.

G.1 Graphs

> *Graph theory has long become recognized as one of the more useful mathematical subjects for the computer science student to master. The approach which is natural in computer science is the algorithmic one; our interest is not so much in existence proofs or enumeration techniques, as it is in finding efficient algorithms for solving relevant problems, or alternatively showing evidence that no such algorithms exist. Although algorithmic graph theory was started by Euler, if not earlier, its development in the last ten years has been dramatic and revolutionary.*
>
> Shimon Even, Graph Algorithms [67]

A simple graph $G = (V, E)$ consists of a *finite* set of vertices V and a finite set of edges E, where each edge is an *unordered pair* of vertices; that is, $E \subseteq \{\{u, v\} : u, v \in V \wedge u \neq v\}$. This formalism does not allow self-loops and parallel edges, which are allowed in general (i.e., non-simple) graphs, where E is a multi-set that may contain singletons (i.e., self-loops). The vertex u is called an end-point of the edge

$\{u,v\}$, and the edge $\{u,v\}$ is said to be incident at v. In such a case we say that u and v are adjacent in the graph, and that u is a neighbor of v. The degree of a vertex in G is defined as the number of edges that are incident at this vertex.

We will consider various sub-structures of graphs, the simplest one being paths. A path in a graph $G=(V,E)$ is a sequence of vertices $v_0,...,v_\ell$ such that for every $i\in[\ell]\stackrel{\rm def}{=}\{1,...,\ell\}$ it holds that v_{i-1} and v_i are adjacent in G. Such a path is said to have length ℓ. A simple path is a path in which each vertex appears at most once, which implies that the longest possible simple path in G has length $|V|-1$. The graph is called connected if there exists a path between each pair of vertices in it.

A cycle is a path in which the last vertex equals the first one (i.e., $v_\ell=v_0$). The cycle $v_0,...,v_\ell$ is called simple if $\ell>2$ and $|\{v_0,...,v_\ell\}|=\ell$. A graph is called acyclic if it has no simple cycles, and if it is also connected then it is called a tree. Note that $G=(V,E)$ is a tree if and only if it is connected and $|E|=|V|-1$.

A subgraph of the graph $G=(V,E)$ is any graph $G'=(V',E')$ satisfying $V'\subseteq V$ and $E'\subseteq E$. Note that a simple cycle in G is a connected subgraph of G in which each vertex has degree exactly two. An induced subgraph of the graph $G=(V,E)$ is any subgraph $G'=(V',E')$ that contain all edges of E that are contained in V'. In such a case, we say that G' is the subgraph induced by V'.

Directed graphs. We will also consider (simple) directed graphs (a.k.a digraphs), where edges are *ordered pairs* of vertices. In this case the set of edges is a subset of $V\times V\setminus\{(v,v):v\in V\}$, and the edges (u,v) and (v,u) are called anti-parallel. General (i.e., non-simple) directed graphs are defined analogously. The edge (u,v) is viewed as going from u to v, and thus is called an out-going edge of u (resp., in-coming edge of v). The out-degree (resp., in-degree) of a vertex is the number of its out-going edges (resp., in-coming edges). Directed paths and the related objects are defined analogously; for example, $v_0,...,v_\ell$ is a directed path if for every $i\in[\ell]$ it holds that (v_{i-1},v_i) is a directed edge (which is directed from v_{i-1} to v_i). In this case it is common to consider also a pair of anti-parallel edges as a simple directed cycle.

A directed acyclic graph (DAG) is a digraph that has no directed cycles. Every DAG has at least one vertex having out-degree (resp., in-degree) zero, called a sink (resp., a source). A simple directed acyclic graph $G=(V,E)$ is called an inward (resp., outward) directed tree if $|E|=|V|-1$ and there exists a unique vertex, called the root, having out-degree (resp., in-degree) zero. Note that each vertex in an inward (resp., outward) directed tree can reach the root (resp., is reachable from the root) by a unique directed path.[1]

[1]Note that in any DAG, there is a directed path from each vertex v to some sink (resp., from some source to each vertex v). In an inward (resp., outward) directed tree this sink (resp., source) must be unique. The condition $|E|=|V|-1$ enforces the uniqueness of these paths, because (combined with the reachability condition) it implies that the underlying graph (obtained by disregarding the orientation of the edges) is a tree.

Representation. Graphs are commonly represented by their adjacency matrix and/or their incidence lists. The adjacency matrix of a simple graph $G=(V,E)$ is a $|V|$-by-$|V|$ Boolean matrix in which the (i,j)-th entry equals 1 if and only if i and j are adjacent in G. The incidence list representation of G consists of $|V|$ sequences such that the $i^{\rm th}$ sequence is an ordered list of the set of edges incident at vertex i.

Computational problems. Simple computational problems regarding graphs include determining whether a given graph is connected (and/or acyclic) and finding shortest paths in a given graph. Another simple problem is determining whether a given graph is bipartite, where a graph $G=(V,E)$ is bipartite (or 2-colorable) if there exists a 2-coloring of its vertices that does not assign neighboring vertices the same color. All these problems are easily solvable by BFS.

Moving to more complicated tasks that are still solvable in polynomial-time, we mention the problem of finding a perfect matching (or a maximum matching) in a given graph, where a matching is a subgraph in which all vertices have degree 1, a perfect matching is a matching that contains all the graph's vertices, and a maximum matching is a matching of maximum cardinality (among all matching of the said graph).

Turning to NP-complete problems, we mention the problem of determining whether a given graph is 3-colorable (i.e., G3C). A few additional NP-complete problems follow.

- A Hamiltonian path (resp., Hamiltonian cycle) in the graph $G=(V,E)$ is a *simple* path (resp., cycle) that passes through all the vertices of G. Such a path (resp., cycle) has length $|V|-1$ (resp., $|V|$). The problem is to determine whether a given graph contains a Hamiltonian path (resp., cycle).

- An independent set (resp., clique) of the graph $G=(V,E)$ is a set of vertices $V' \subseteq V$ such that the subgraph induced by V' contains no edges (resp., contains all possible edges). The problem is to determine whether a given graph has an independent set (resp., a clique) of a given size.

  A vertex cover of the graph $G=(V,E)$ is a set of vertices $V' \subseteq V$ such that each edge in E has at least one end-point in V'. Note that V' is a vertex cover of G if and only if $V \setminus V'$ is an independent set of V.

A natural computational problem which is believed to be neither in $\mathcal{P}$ nor NP-complete is the graph isomorphism problem. The input consists of two graphs, $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$, and the question is whether there exist a 1-1 and onto mapping $\phi : V_1 \to V_2$ such that $\{u,v\}$ is in E_1 if and only if $\{\phi(u),\phi(v)\}$ is in E_2. (Such a mapping is called an isomorphism.)

G.2 Formulae

In §1.2.4.3, Boolean formulae are defined as a special case of Boolean circuits (§1.2.4.1). Here we take the more traditional approach, and define Boolean formulae as structured sequences over an alphabet consisting of variable names and various connectives. It is most convenient to define Boolean formulae recursively as follows:

- A variable is a Boolean formula.
- If $\phi_1, ..., \phi_t$ are Boolean formulae and ψ is a t-ary Boolean operation then $\psi(\phi_1, ..., \phi_t)$ is a Boolean formula.

Typically, we consider three Boolean operations: the unary operation of negation (denoted `neg` or $\neg$), and the (bounded or unbounded) conjunction and disjunction (denoted $\wedge$ and $\vee$, respectively). Furthermore, the convention is to shorthand $\neg(\phi)$ by $\neg\phi$, and to write $(\wedge_{i=1}^{t} \phi_i)$ or $(\phi_1 \wedge \cdots \wedge \phi_t)$ instead of $\wedge(\phi_1, ..., \phi_t)$, and similarly for $\vee$.

Two important special cases of Boolean formulae are CNF and DNF formulae. A CNF formula is a conjunction of disjunctions of variables and/or their negation; that is, $\wedge_{i=1}^{t} \phi_i$ is a CNF if each ϕ_i has the form $(\wedge_{j=1}^{t_i} \phi_{i,j})$, where each $\phi_{i,j}$ is either a variable or a negation of a variable (and is called a literal). If for every i it holds that $t_i \leq 3$ then we say that the formula is a 3CNF. Similarly, DNF formulae are defined as disjunctions of conjunctions of literals.

The value of a Boolean formula under a truth assignment to its variables is defined recursively along its structure. For example, $\wedge_{i=1}^{t} \phi_i$ has the value `true` under an assignment τ if and only if every ϕ_i has the value `true` under τ. We say that a formula ϕ is satisfiable if there exists a truth assignment τ to its variables such that the value of ϕ under τ is `true`.

The set of satisfiable CNF (resp., 3CNF) formulae is denoted `SAT` (resp., `3SAT`), and the problem of deciding membership in it is NP-complete. The set of tautologies (i.e., formula that have the value `true` under any assignment), even when restricted to 3DNF formulae, is coNP-complete.

Quantified Boolean Formulae. In contrast to the above that refers to unquantified Boolean formulae, a quantified Boolean formula is a formula augmented with quantifiers that refer to each variable appearing in it. That is, if ϕ is a formula in the Boolean variables $x_1, ..., x_n$ and $Q_1, ..., Q_n$ are Boolean quantifiers (i.e., each Q_i is either $\exists$ or $\forall$) then $Q_1 x_1 \cdots Q_n x_n \, \phi(x_1, ..., x_n)$ is a quantified Boolean formula. A k-alternating quantified Boolean formula is a quantified Boolean formula with up to k alternations between existential and universal quantifiers, starting with an existential quantifier. For example, $\exists x_1 \exists x_2 \forall x_3 \phi(x_1, x_2, x_3)$ is a 2-alternating quantified Boolean formula.

The set of satisfiable k-alternating quantified Boolean formulae is denoted `kQBF` and is Σ_k-complete, whereas the set of all satisfiable quantified Boolean formulae is denoted `QBF` and is $\mathcal{PSPACE}$-complete.

The foregoing definition refers to the canonical form of quantified Boolean formulae, in which all the quantifiers appear at the leftmost side of the formula. A more general definition allows each variable to be quantified at an arbitrary place left to its leftmost occurrence in the formula (e.g., $(\forall x_1)(\exists x_2)\,(x_1 = x_2) \wedge (\exists x_3)(x_3 = x_1)$). Note that such generalized formulae (used in the proof of Theorems 5.15

and 9.4) can be transformed to the canonical form by "pulling" all quantifiers to the left of the formula (e.g., $\forall x_1 \exists x_2 \exists x_3 \left((x_1 = x_2) \wedge (x_3 = x_1)\right)$).

G.3 Algebra

Various algebraic objects, computational problems and technique play an important role in complexity theory. The most dominant objects are finite fields as well as vector spaces and polynomials over such fields.

We denote by $\mathrm{GF}(q)$ the finite field of q elements and note that q may be either a prime or a prime power. In the first case, $\mathrm{GF}(q)$ is viewed as consisting of the elements $\{0, ..., q-1\}$ with addition and multiplication being defined modulo q. Indeed, $\mathrm{GF}(2)$ is an important special case. In the case that $q = p^e$, where p is a prime and $e > 1$, the standard representation of $\mathrm{GF}(p^e)$ refers to an irreducible polynomial of degree e over $\mathrm{GF}(p)$. Finding such polynomials is a non-trivial computational problem (see [21]). Fortunately, for $e = 2 \cdot 3^{e'}$ (with e' being an integer), the polynomial $x^e + x^{e/2} + 1$ is irreducible over $\mathrm{GF}(2)$, which means that we can avoid trouble in this important case. In any case, if f is an irreducible polynomial of degree e over $\mathrm{GF}(p)$ then $\mathrm{GF}(p^e)$ can be represented as the set of polynomial with degree at most $e-1$ over $\mathrm{GF}(p)$ with addition and multiplication defined modulo the polynomial f.

The determinant and the permanent. These two computational problems, seem to have a vastly different complexity. Recall that the permanent of an n-by-n matrix $M = (a_{i,j})$ is defined as the sum $\sum_\pi \prod_{i=1}^n a_{i,\pi(j)}$ taken over all permutations of the set $\{1, ..., n\}$. This is related to the definition of the determinant in which the same sum is used except that some elements are negated; that is, the determinant of $M = (a_{i,j})$ is defined as $\sum_\pi (-1)^{\sigma(\pi)} \prod_{i=1}^n a_{i,\pi(j)}$, where $\sigma(\pi) = 1$ if π is an even permutation (i.e., can be expressed by an even number of transpositions) and $\sigma(\pi) = -1$ otherwise.

G.4 Number Theory

A prime is a natural number that is not divisible by any natural number other than itself and 1. For simplicity, say that 1 is NOT a prime. A natural number that is not a prime is called composite, and its prime factorization is the set of primes that divide it; that is, if $N = \prod_{i=1}^t P_i^{e_i}$, where the P_i's are distinct primes and $e_i \geq 1$, then $\{P_i : i = 1, ..., t\}$ is the prime factorization of N. (If $t = 1$ then N is a prime power.)

Two famous computational problems, identified by Gauss as fundamental ones, are testing primality (i.e., given a natural number, determine whether it is prime or composite) and factoring composite integers (i.e., given a composite number, find its prime factorization). Needless to say, in both cases, the input is presented in binary representation. Whereas testing primality is in $\mathcal{P}$ (see [3] and §6.1.1.2 showing that the problem is in $\mathcal{BPP}$), it is conjectured that factoring composite integers is intractable. In fact, many popular candidates for various cryptographic systems are based on this conjecture.

Extracting modular square roots. Two related computational problems are extracting (modular) square roots (for prime and composite moduli). Specifically, a quadratic residue modulo a prime P is an integer s such that there exists a number r satisfying $s \equiv r^2 \pmod{P}$. The corresponding search problem (i.e., given such P and s, find r) can be solved in probabilistic polynomial-time (see Exercise 6.14). The corresponding problem for composite moduli is computationally equivalent to factoring (see [170]); furthermore, extracting square roots modulo N is easily reducible to factoring N, and factoring N is randomly reducible to extracting square roots modulo N (even in a typical-case sense). We mention that even the problem of deciding whether or not a given integer s has a modular square root modulo a given composite is conjectured to be hard.

Bibliography

[1] S. Aaronson. Complexity Zoo. A continueously updated web-site at `http://qwiki.caltech.edu/wiki/Complexity_Zoo/`.

[2] L.M. Adleman and M. Huang. *Primality Testing and Abelian Varieties Over Finite Fields*. Springer-Verlag Lecture Notes in Computer Science (Vol. 1512), 1992. Preliminary version in *19th STOC*, 1987.

[3] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, Vol. 160 (2), pages 781–793, 2004.

[4] M. Ajtai, J. Komlos, E. Szemerédi. Deterministic Simulation in LogSpace. In *19th ACM Symposium on the Theory of Computing*, pages 132–140, 1987.

[5] R. Aleliunas, R.M. Karp, R.J. Lipton, L. Lovász and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *20th IEEE Symposium on Foundations of Computer Science*, pages 218–223, 1979.

[6] N. Alon, L. Babai and A. Itai. A fast and Simple Randomized Algorithm for the Maximal Independent Set Problem. *J. of Algorithms*, Vol. 7, pages 567–583, 1986.

[7] N. Alon and R. Boppana. The monotone circuit complexity of Boolean functions. *Combinatorica*, Vol. 7 (1), pages 1–22, 1987.

[8] N. Alon, E. Fischer, I. Newman, and A. Shapira. A Combinatorial Characterization of the Testable Graph Properties: It's All About Regularity. In *38th ACM Symposium on the Theory of Computing*, to appear, 2006.

[9] N. Alon, O. Goldreich, J. Håstad, R. Peralta. Simple Constructions of Almost k-wise Independent Random Variables. *Journal of Random structures and Algorithms*, Vol. 3, No. 3, (1992), pages 289–304.

[10] N. Alon and J.H. Spencer. *The Probabilistic Method*. John Wiley & Sons, Inc., 1992.

[11] R. Armoni. On the derandomization of space-bounded computations. In the proceedings of *Random98*, Springer-Verlag, Lecture Notes in Computer Science (Vol. 1518), pages 49–57, 1998.

[12] S. Arora. Approximation schemes for NP-hard geometric optimization problems: A survey. *Math. Programming*, Vol. 97, pages 43–69, July 2003.

[13] S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szegedy. Proof Verification and Intractability of Approximation Problems. *Journal of the ACM*, Vol. 45, pages 501–555, 1998. Preliminary version in *33rd FOCS*, 1992.

[14] S. Arora and S. Safra. Probabilistic Checkable Proofs: A New Characterization of NP. *Journal of the ACM*, Vol. 45, pages 70–122, 1998. Preliminary version in *33rd FOCS*, 1992.

[15] H. Attiya and J. Welch: *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, 1998.

[16] L. Babai. Trading Group Theory for Randomness. In *17th ACM Symposium on the Theory of Computing*, pages 421–429, 1985.

[17] L. Babai, L. Fortnow, and C. Lund. Non-Deterministic Exponential Time has Two-Prover Interactive Protocols. *Computational Complexity*, Vol. 1, No. 1, pages 3–40, 1991. Preliminary version in *31st FOCS*, 1990.

[18] L. Babai, L. Fortnow, L. Levin, and M. Szegedy. Checking Computations in Polylogarithmic Time. In *23rd ACM Symposium on the Theory of Computing*, pages 21–31, 1991.

[19] L. Babai, L. Fortnow, N. Nisan and A. Wigderson. BPP has Subexponential Time Simulations unless EXPTIME has Publishable Proofs. *Complexity Theory*, Vol. 3, pages 307–318, 1993.

[20] L. Babai and S. Moran. Arthur-Merlin Games: A Randomized Proof System and a Hierarchy of Complexity Classes. *Journal of Computer and System Science*, Vol. 36, pp. 254–276, 1988.

[21] E. Bach and J. Shallit. *Algorithmic Number Theory* (Volume I: Efficient Algorithms). MIT Press, 1996.

[22] B. Barak. Non-Black-Box Techniques in Crypptography. PhD Thesis, Weizmann Institute of Science, 2004.

[23] W. Baur and V. Strassen. The Complexity of Partial Derivatives. *Theor. Comput. Sci.* 22, pages 317–330, 1983.

[24] P. Beame and T. Pitassi. Propositional Proof Complexity: Past, Present, and Future. In *Bulletin of the European Association for Theoretical Computer Science*, Vol. 65, June 1998, pp. 66–89.

[25] A. Beimel, Y. Ishai, E. Kushilevitz, and J.F. Raymond. Breaking the $O(n^{1/(2k-1)})$ barrier for information-theoretic private information retrieval. In *43rd IEEE Symposium on Foundations of Computer Science*, pages 261–270, 2002.

[26] M. Bellare, O. Goldreich, and E. Petrank. Uniform Generation of NP-witnesses using an NP-oracle. *Information and Computation*, Vol. 163, pages 510–526, 2000.

[27] M. Bellare, O. Goldreich and M. Sudan. Free Bits, PCPs and Non-Approximability – Towards Tight Results. *SIAM Journal on Computing*, Vol. 27, No. 3, pages 804–915, 1998. Extended abstract in *36th FOCS*, 1995.

[28] S. Ben-David, B. Chor, O. Goldreich, and M. Luby. On the Theory of Average Case Complexity. *Journal of Computer and System Science*, Vol. 44 (2), pages 193–219, 1992.

[29] A. Ben-Dor and S. Halevi. In *2nd Israel Symp. on Theory of Computing and Systems*, IEEE Computer Society Press, pages 108-117, 1993.

[30] M. Ben-Or, O. Goldreich, S. Goldwasser, J. Håstad, J. Kilian, S. Micali and P. Rogaway. Everything Provable is Probable in Zero-Knowledge. In *Crypto88*, Springer-Verlag Lecture Notes in Computer Science (Vol. 403), pages 37–56, 1990

[31] M. Ben-Or, S. Goldwasser, J. Kilian and A. Wigderson. Multi-Prover Interactive Proofs: How to Remove Intractability. In *20th ACM Symposium on the Theory of Computing*, pages 113–131, 1988.

[32] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *20th ACM Symposium on the Theory of Computing*, pages 1–10, 1988.

[33] E. Ben-Sasson, O. Goldreich, P. Harsha, M. Sudan, and S. Vadhan. Robust PCPs of proximity, Shorter PCPs and Applications to Coding. In *36th ACM Symposium on the Theory of Computing*, pages 1–10, 2004. Full version in *ECCC*, TR04-021, 2004.

[34] E. Ben-Sasson and M. Sudan. Simple PCPs with Poly-log Rate and Query Complexity. *ECCC*, TR04-060, 2004.

[35] L. Berman and J. Hartmanis. On isomorphisms and density of NP and other complete sets. *SIAM Journal on Computing*, Vol. 6 (2), 1977, pages 305–322. Extended abstract in *8th STOC*, 1976.

[36] M. Blum. A Machine-Independent Theory of the Complexity of Recursive Functions. *Journal of the ACM*, Vol. 14 (2), pages 290–305, 1967.

[37] M. Blum and S. Micali. How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. *SIAM Journal on Computing*, Vol. 13, pages 850–864, 1984. Preliminary version in *23rd FOCS*, 1982.

[38] M. Blum, M. Luby and R. Rubinfeld. Self-Testing/Correcting with Applications to Numerical Problems. *Journal of Computer and System Science*, Vol. 47, No. 3, pages 549–595, 1993.

[39] A. Bogdanov, K. Obata, and L. Trevisan. A lower bound for testing 3-colorability in bounded-degree graphs. In *43rd IEEE Symposium on Foundations of Computer Science*, pages 93–102, 2002.

[40] A. Bogdanov and L. Trevisan. On worst-case to average-case reductions for NP problems. In *Proc. 44th IEEE Symposium on Foundations of Computer Science*, pages 308–317, 2003.

[41] A. Bogdanov and L. Trevisan. Average-case complexity: a survey. In preparation, 2005.

[42] R. Boppana, J. Håstad, and S. Zachos. Does Co-NP Have Short Interactive Proofs? *Information Processing Letters*, 25, May 1987, pages 127-132.

[43] R. Boppana and M. Sipser. The complexity of finite functions. In *Handbook of Theoretical Computer Science: Volume A – Algorithms and Complexity*, J. van Leeuwen editor, MIT Press/Elsevier, 1990, pages 757–804.

[44] A. Borodin. Computational Complexity and the Existence of Complexity Gaps. *Journal of the ACM*, Vol. 19 (1), pages 158–174, 1972.

[45] A. Borodin. On Relating Time and Space to Size and Depth. *SIAM Journal on Computing*, Vol. 6 (4), pages 733–744, 1977.

[46] G. Brassard, D. Chaum and C. Crépeau. Minimum Disclosure Proofs of Knowledge. *Journal of Computer and System Science*, Vol. 37, No. 2, pages 156–189, 1988. Preliminary version by Brassard and Crépeau in *27th FOCS*, 1986.

[47] L. Carter and M. Wegman. Universal Hash Functions. *Journal of Computer and System Science*, Vol. 18, 1979, pages 143–154.

[48] G.J. Chaitin. On the Length of Programs for Computing Finite Binary Sequences. *Journal of the ACM*, Vol. 13, pages 547–570, 1966.

[49] A.K. Chandra, D.C. Kozen and L.J. Stockmeyer. Alternation. *Journal of the ACM*, Vol. 28, pages 114–133, 1981.

[50] D. Chaum, C. Crépeau and I. Damgård. Multi-party unconditionally Secure Protocols. In *20th ACM Symposium on the Theory of Computing*, pages 11–19, 1988.

[51] B. Chor and O. Goldreich. On the Power of Two–Point Based Sampling. *Jour. of Complexity*, Vol 5, 1989, pages 96–106. Preliminary version dates 1985.

[52] B. Chor and O. Goldreich. Unbiased Bits from Sources of Weak Randomness and Probabilistic Communication Complexity. *SIAM Journal on Computing*, Vol. 17, No. 2, pages 230–261, 1988.

[53] A. Church. An Unsolvable Problem of Elementary Number Theory. *Amer. J. of Math.*, Vol. 58, pages 345–363, 1936.

[54] A. Cobham. The Intristic Computational Difficulty of Functions. In *Proc. 1964 Iternational Congress for Logic Methodology and Philosophy of Science*, pages 24–30, 1964.

[55] S.A. Cook. The Complexity of Theorem Proving Procedures. In *3rd ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[56] S.A. Cook. A overview of Computational Complexity. Turing Award Lecture. *CACM*, Vol. 26 (6), pages 401–408, 1983.

[57] S.A. Cook. A Taxonomy of Problems with Fast Parallel Algorithms. *Information and Control*, Vol. 64, pages 2–22, 1985.

[58] S.A. Cook and R.A. Reckhow. Stephen A. Cook, Robert A. Reckhow: The Relative Efficiency of Propositional Proof Systems. *J. of Symbolic Logic*, Vol. 44 (1), pages 36–50, 1979.

[59] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9, pages 251–280, 1990.

[60] T.M. Cover and G.A. Thomas. *Elements of Information Theory.* John Wiley & Sons, Inc., New-York, 1991.

[61] P. Crescenzi and V. Kann. A compendium of NP Optimization problems. Available at `http://www.nada.kth.se/~viggo/wwwcompendium/`

[62] W. Diffie, and M.E. Hellman. New Directions in Cryptography. *IEEE Trans. on Info. Theory*, IT-22 (Nov. 1976), pages 644–654.

[63] I. Dinur. The PCP Theorem by Gap Amplification. *ECCC*, TR05-046, 2005.

[64] I. Dinur and O. Reingold. Assignment-testers: Towards a combinatorial proof of the PCP-Theorem. In *45th IEEE Symposium on Foundations of Computer Science*, pages 155–164, 2004.

[65] I. Dinur and S. Safra. The importance of being biased. In *34th ACM Symposium on the Theory of Computing*, pages 33–42, 2002.

[66] J. Edmonds. Paths, Trees, and Flowers. *Canad. J. Math.*, Vol. 17, pages 449–467, 1965.

[67] S. Even. *Graph Algorithms.* Computer Science Press, 1979.

[68] S. Even, A.L. Selman, and Y. Yacobi. The Complexity of Promise Problems with Applications to Public-Key Cryptography. *Information and Control*, Vol. 61, pages 159–173, 1984.

[69] U. Feige, S. Goldwasser, L. Lovász and S. Safra. On the Complexity of Approximating the Maximum Size of a Clique. Unpublished manuscript, 1990.

[70] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating Clique is almost NP-complete. *Journal of the ACM*, Vol. 43, pages 268–292, 1996. Preliminary version in *32nd FOCS*, 1991.

[71] U. Feige, D. Lapidot, and A. Shamir. Multiple Non-Interactive Zero-Knowledge Proofs Under General Assumptions. *SIAM Journal on Computing*, Vol. 29 (1), pages 1–28, 1999.

[72] U. Feige and A. Shamir. Witness Indistinguishability and Witness Hiding Protocols. In *22nd ACM Symposium on the Theory of Computing*, pages 416–426, 1990.

[73] E. Fischer. The art of uninformed decisions: A primer to property testing. *Bulletin of the European Association for Theoretical Computer Science*, Vol. 75, pages 97–126, 2001.

[74] G.D. Forney. *Concatenated Codes.* MIT Press, Cambridge, MA 1966.

[75] L. Fortnow, R. Lipton, D. van Melkebeek, and A. Viglas. Time-space lower bounds for satisfiability. *Journal of the ACM*, Vol. 52 (6), pages 835–865, November 2005.

[76] L. Fortnow, J. Rompel and M. Sipser. On the power of multi-prover interactive protocols. In *3rd IEEE Symp. on Structure in Complexity Theory*, pages 156–161, 1988. See errata in *5th IEEE Symp. on Structure in Complexity Theory*, pages 318–319, 1990.

[77] S. Fortune. A Note on Sparse Complete Sets. *SIAM Journal on Computing*, Vol. 8, pages 431–433, 1979.

[78] M. Fürer, O. Goldreich, Y. Mansour, M. Sipser, and S. Zachos. On Completeness and Soundness in Interactive Proof Systems. *Advances in Computing Research: a research annual*, Vol. 5 (Randomness and Computation, S. Micali, ed.), pages 429–442, 1989.

[79] M.L. Furst, J.B. Saxe, and M. Sipser. Parity, Circuits, and the Polynomial-Time Hierarchy. *Mathematical Systems Theory*, Vol. 17 (1), pages 13–27, 1984. Preliminary version in *22nd FOCS*, 1981.

[80] O. Gaber and Z. Galil. Explicit Constructions of Linear Size Superconcentrators. *Journal of Computer and System Science*, Vol. 22, pages 407–420, 1981.

[81] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

[82] D. Gillman. A chernoff bound for random walks on expander graphs. In *34th IEEE Symposium on Foundations of Computer Science*, pages 680–691, 1993.

[83] O. Goldreich. *Foundation of Cryptography – Class Notes*. Computer Science Dept., Technion, Israel, Spring 1989. Superseded by [87, 88].

[84] O. Goldreich. A Note on Computational Indistinguishability. *Information Processing Letters*, Vol. 34, pages 277–281, May 1990.

[85] O. Goldreich. Notes on Levin's Theory of Average-Case Complexity. *ECCC*, TR97-058, Dec. 1997.

[86] O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. Algorithms and Combinatorics series (Vol. 17), Springer, 1999.

[87] O. Goldreich. *Foundation of Cryptography: Basic Tools*. Cambridge University Press, 2001.

[88] O. Goldreich. *Foundation of Cryptography: Basic Applications*. Cambridge University Press, 2004.

[89] O. Goldreich. Short Locally Testable Codes and Proofs (Survey). *ECCC*, TR05-014, 2005.

[90] O. Goldreich. On Promise Problems (a survey in memory of Shimon Even [1935-2004]). *ECCC*, TR05-018, 2005.

[91] O. Goldreich, S. Goldwasser, and S. Micali. How to Construct Random Functions. *Journal of the ACM*, Vol. 33, No. 4, pages 792–807, 1986.

[92] O. Goldreich, S. Goldwasser, and A. Nussboim. On the Implementation of Huge Random Objects. In *44th IEEE Symposium on Foundations of Computer Science*, pages 68–79, 2002.

[93] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *Journal of the ACM*, pages 653–750, July 1998.

[94] O. Goldreich and H. Krawczyk. On the Composition of Zero-Knowledge Proof Systems. *SIAM Journal on Computing*, Vol. 25, No. 1, February 1996, pages 169–192. Preliminary version in *17th ICALP*, 1990.

[95] O. Goldreich and L.A. Levin. Hard-core Predicates for any One-Way Function. In *21st ACM Symposium on the Theory of Computing*, pages 25–32, 1989.

[96] O. Goldreich, S. Micali and A. Wigderson. Proofs that Yield Nothing but their Validity or All Languages in NP Have Zero-Knowledge Proof Systems. *Journal of the ACM*, Vol. 38, No. 3, pages 691–729, 1991. Preliminary version in *27th FOCS*, 1986.

[97] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In *19th ACM Symposium on the Theory of Computing,* pages 218–229, 1987.

[98] O. Goldreich, N. Nisan and A. Wigderson. On Yao's XOR-Lemma. *ECCC*, TR95-050, 1995.

[99] O. Goldreich and D. Ron. Property testing in bounded degree graphs. *Algorithmica*, pages 302–343, 2002.

[100] O. Goldreich and D. Ron. A sublinear bipartite tester for bounded degree graphs. *Combinatorica*, Vol. 19 (3), pages 335–373, 1999.

[101] O. Goldreich, R. Rubinfeld and M. Sudan. Learning polynomials with queries: the highly noisy case. *SIAM J. Discrete Math.*, Vol. 13 (4), pages 535–570, 2000.

[102] O. Goldreich, S. Vadhan and A. Wigderson. On interactive proofs with a laconic provers. *Computational Complexity*, Vol. 11, pages 1–53, 2002.

[103] O. Goldreich and A. Wigderson. Computational Complexity. In *The Princeton Companion to Mathematics*, to appear.

[104] S. Goldwasser and S. Micali. Probabilistic Encryption. *Journal of Computer and System Science*, Vol. 28, No. 2, pages 270–299, 1984. Preliminary version in *14th STOC*, 1982.

[105] S. Goldwasser, S. Micali and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing*, Vol. 18, pages 186–208, 1989. Preliminary version in *17th STOC*, 1985. Earlier versions date to 1982.

[106] S. Goldwasser, S. Micali, and R.L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM Journal on Computing*, April 1988, pages 281–308.

[107] S. Goldwasser and M. Sipser. Private Coins versus Public Coins in Interactive Proof Systems. *Advances in Computing Research: a research annual*, Vol. 5 (Randomness and Computation, S. Micali, ed.), pages 73–90, 1989. Extended abstract in *18th STOC*, 1986.

[108] S.W. Golomb. *Shift Register Sequences.* Holden-Day, 1967. (Aegean Park Press, revised edition, 1982.)

[109] J. Hartmanis and R.E. Stearns. On the Computational Complexity of of Algorithms. *Transactions of the AMS*, Vol. 117, pages 285–306, 1965.

[110] J. Håstad. Almost Optimal Lower Bounds for Small Depth Circuits. *Advances in Computing Research: a research annual*, Vol. 5 (Randomness and Computation, S. Micali, ed.), pages 143–170, 1989. Extended abstract in *18th STOC*, pages 6–20, 1986.

[111] J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, Vol. 182, pages 105–142, 1999. Preliminary versions in *28th STOC* (1996) and *37th FOCS* (1996).

[112] J. Håstad. Getting optimal in-approximability results. In *29th ACM Symposium on the Theory of Computing*, pages 1–10, 1997.

[113] J. Håstad, R. Impagliazzo, L.A. Levin and M. Luby. A Pseudorandom Generator from any One-way Function. *SIAM Journal on Computing*, Volume 28, Number 4, pages 1364–1396, 1999. Preliminary versions by Impagliazzo *et. al.* in *21st STOC* (1989) and Håstad in *22nd STOC* (1990).

[114] J. Håstad and S. Khot. Query efficient PCPs with pefect completeness. In *42nd IEEE Symposium on Foundations of Computer Science*, pages 610–619, 2001.

[115] A. Healy, S. Vadhan and E. Viola. Using nondeterminism to amplify hardness. In *36th ACM Symposium on the Theory of Computing*, pages 192–201, 2004.

[116] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[117] D. Hochbaum (ed.). *Approximation Algorithms for NP-Hard Problems.* PWS, 1996.

[118] N. Immerman. Nondeterministic Space is Closed Under Complementation. *SIAM Journal on Computing*, Vol. 17, pages 760–778, 1988.

[119] R. Impagliazzo. Hard-core Distributions for Somewhat Hard Problems. In *36th IEEE Symposium on Foundations of Computer Science*, pages 538–545, 1995.

[120] R. Impagliazzo and L.A. Levin. No Better Ways to Generate Hard NP Instances than Picking Uniformly at Random. In *31st IEEE Symposium on Foundations of Computer Science*, pages 812–821, 1990.

[121] R. Impagliazzo and A. Wigderson. P=BPP if E requires exponential circuits: Derandomizing the XOR Lemma. In *29th ACM Symposium on the Theory of Computing*, pages 220–229, 1997.

[122] R. Impagliazzo and M. Yung. Direct Zero-Knowledge Computations. In *Crypto87*, Springer-Verlag Lecture Notes in Computer Science (Vol. 293), pages 40–51, 1987.

[123] M. Jerrum, A. Sinclair, and E. Vigoda. A Polynomial-Time Approximation Algorithm for the Permanent of a Matrix with Non-Negative Entries. *Journal of the ACM*, Vol. 51 (4), pages 671–697, 2004.

[124] M. Jerrum, L. Valiant, and V.V. Vazirani. Random Generation of Combinatorial Structures from a Uniform Distribution. *Theoretical Computer Science*, Vol. 43, pages 169–188, 1986.

[125] N. Kahale, Eigenvalues and Expansion of Regular Graphs. *Journal of the ACM*, Vol. 42 (5), pages 1091–1106, September 1995.

[126] R. Kannan, H. Venkateswaran, V. Vinay, and A.C. Yao. A Circuit-based Proof of Toda's Theorem. *Information and Computation*, Vol. 104 (2), pages 271–276, 1993.

[127] R.M. Karp. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, R.E. Miller and J.W. Thatcher (eds.), Plenum Press, pages 85–103, 1972.

[128] R.M. Karp and R.J. Lipton. Some connections between nonuniform and uniform complexity classes. In *12th ACM Symposium on the Theory of Computing*, pages 302-309, 1980.

[129] R.M. Karp and M. Luby. Monte-Carlo algorithms for enumeration and reliability problems. In *24th IEEE Symposium on Foundations of Computer Science*, pages 56-64, 1983.

[130] R.M. Karp and V. Ramachandran: Parallel Algorithms for Shared-Memory Machines. In *Handbook of Theoretical Computer Science, Vol A: Algorithms and Complexity*, 1990.

[131] M. Karchmer and A. Wigderson. Monotone Circuits for Connectivity Require Super-logarithmic Depth. *SIAM J. Discrete Math.*, Vol. 3 (2), pages 255–265, 1990. Preliminary version in *20th STOC*, 1988.

[132] M.J. Kearns and U.V. Vazirani. *An introduction to Computational Learning Theory*. MIT Press, 1994.

[133] S. Khot and O. Regev. Vertex Cover Might be Hard to Approximate to within $2-\varepsilon$. In *18th IEEE Conference on Computational Complexity*, pages 379–386, 2003.

[134] V.M. Khrapchenko. A method of determining lower bounds for the complexity of Pi-schemes. In *Matematicheskie Zametki* 10 (1),pages 83–92, 1971 (in Russian). English translation in *Mathematical Notes of the Academy of Sciences of the USSR* 10 (1) 1971, pages 474–479.

[135] J. Kilian. A Note on Efficient Zero-Knowledge Proofs and Arguments. In *24th ACM Symposium on the Theory of Computing*, pages 723–732, 1992.

[136] D.E. Knuth. *The Art of Computer Programming*, Vol. 2 (*Seminumerical Algorithms*). Addison-Wesley Publishing Company, Inc., 1969 (first edition) and 1981 (second edition).

[137] A. Kolmogorov. Three Approaches to the Concept of "The Amount Of Information". *Probl. of Inform. Transm.*, Vol. 1/1, 1965.

[138] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1996.

[139] R.E. Ladner. On the Structure of Polynomial Time Reducibility. *Journal of the ACM*, Vol. 22, 1975, pages 155–171.

[140] C. Lautemann. BPP and the Polynomial Hierarchy. *Information Processing Letters*, 17, pages 215–217, 1983.

[141] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[142] L.A. Levin. Universal Search Problems. *Problemy Peredaci Informacii 9*, pages 115–116, 1973. Translated in *problems of Information Transmission 9*, pages 265–266.

[143] L.A. Levin. Randomness Conservation Inequalities: Information and Independence in Mathematical Theories. *Information and Control*, Vol. 61, pages 15–37, 1984.

[144] L.A. Levin. Average Case Complete Problems. *SIAM Journal on Computing*, Vol. 15, pages 285–286, 1986.

[145] L.A. Levin. Fundamentals of Computing. *SIGACT News*, Education Forum, special 100-th issue, Vol. 27 (3), pages 89–110, 1996.

[146] M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and its Applications.* Springer Verlag, August 1993.

[147] C.-J. Lu, O. Reingold, S. Vadhan, and A. Wigderson. Extractors: optimal up to constant factors. In *35th ACM Symposium on the Theory of Computing*, pages 602–611, 2003.

[148] A. Lubotzky, R. Phillips, and P. Sarnak. Ramanujan Graphs. *Combinatorica*, Vol. 8, pages 261–277, 1988.

[149] M. Luby and A. Wigderson. Pairwise Independence and Derandomization. TR-95-035, International Computer Science Institute (ICSI), Berkeley, 1995. ISSN 1075-4946.

[150] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic Methods for Interactive Proof Systems. *Journal of the ACM*, Vol. 39, No. 4, pages 859–868, 1992. Preliminary version in *31st FOCS*, 1990.

[151] F. MacWilliams and N. Sloane. *The theory of error-correcting codes.* North-Holland, 1981.

[152] G.A. Margulis. Explicit Construction of Concentrators. (In Russian.) *Prob. Per. Infor.*, Vol. 9 (4), pages 71–80, 1973. English translation in *Problems of Infor. Trans.*, pages 325–332, 1975.

[153] S. Micali. Computationally Sound Proofs. *SIAM Journal on Computing*, Vol. 30 (4), pages 1253–1298, 2000. Preliminary version in *35th FOCS*, 1994.

[154] G.L. Miller. Riemann's Hypothesis and Tests for Primality. *Journal of Computer and System Science*, Vol. 13, pages 300–317, 1976.

[155] P.B. Miltersen and N.V. Vinodchandran. Derandomizing Arthur-Merlin Games using Hitting Sets. *Journal of Computational Complexity*, to appear. Preliminary version in *40th FOCS*, 1999.

[156] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[157] M. Naor. Bit Commitment using Pseudorandom Generators. *Journal of Cryptology*, Vol. 4, pages 151–158, 1991.

[158] J. Naor and M. Naor. Small-bias Probability Spaces: Efficient Constructions and Applications. *SIAM Journal on Computing*, Vol 22, 1993, pages 838–856.

[159] M. Naor and M. Yung. Universal One-Way Hash Functions and their Cryptographic Application. In *21st ACM Symposium on the Theory of Computing*, 1989, pages 33–43.

[160] N. Nisan. Pseudorandom bits for constant depth circuits. *Combinatorica*, Vol. 11 (1), pages 63–70, 1991.

[161] N. Nisan. Pseudorandom Generators for Space Bounded Computation. *Combinatorica*, Vol. 12 (4), pages 449–461, 1992.

[162] N. Nisan. $\mathcal{RL} \subseteq \mathcal{SC}$. *Journal of Computational Complexity*, Vol. 4, pages 1-11, 1994.

[163] N. Nisan and A. Wigderson. Hardness vs Randomness. *Journal of Computer and System Science*, Vol. 49, No. 2, pages 149–167, 1994.

[164] N. Nisan and D. Zuckerman. Randomness is Linear in Space. *Journal of Computer and System Science*, Vol. 52 (1), pages 43–52, 1996.

[165] C.H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

[166] C.H. Papadimitriou and M. Yannakakis. Optimization, Approximation, and Complexity Classes. In *20th ACM Symposium on the Theory of Computing*, pages 229–234, 1988.

[167] N. Pippenger and M.J. Fischer. Relations among complexity measures. *Journal of the ACM*, Vol. 26 (2), pages 361–381, 1979.

[168] E. Post. A Variant of a Recursively Unsolvable Problem. *Bull. AMS*, Vol. 52, pages 264–268, 1946.

[169] M.O. Rabin. Digitalized Signatures. In *Foundations of Secure Computation* (R.A. DeMillo et. al. eds.), Academic Press, 1977.

[170] M.O. Rabin. Digitalized Signatures and Public Key Functions as Intractable as Factoring. MIT/LCS/TR-212, 1979.

[171] M.O. Rabin. Probabilistic Algorithm for Testing Primality. *Journal of Number Theory*, Vol. 12, pages 128–138, 1980.

[172] R. Raz. A Parallel Repetition Theorem. *SIAM Journal on Computing*, Vol. 27 (3), pages 763–803, 1998. Extended abstract in *27th STOC*, 1995.

[173] R. Raz and A. Wigderson. Monotone Circuits for Matching Require Linear Depth. *Journal of the ACM*, Vol. 39 (3), pages 736–744, 1992. Preliminary version in *22nd STOC*, 1990.

[174] A. Razborov. Lower bounds for the monotone complexity of some Boolean functions. In *Doklady Akademii Nauk SSSR*, Vol. 281, No. 4, 1985, pages 798–801. English translation in *Soviet Math. Doklady*, 31, pages 354–357, 1985.

[175] A. Razborov. Lower bounds on the size of bounded-depth networks over a complete basis with logical addition. In *Matematicheskie Zametki*, Vol. 41, No. 4, pages 598–607, 1987. English translation in *Mathematical Notes of the Academy of Sci. of the USSR*, Vol. 41 (4), pages 333–338, 1987.

[176] A.R. Razborov and S. Rudich. Natural Proofs. *Journal of Computer and System Science*, Vol. 55 (1), pages 24–35, 1997.

[177] O. Reingold. Undirected ST-Connectivity in Log-Space. In *37th ACM Symposium on the Theory of Computing*, pages 376–385, 2005.

[178] O. Reingold, S. Vadhan, and A. Wigderson. Entropy Waves, the Zig-Zag Graph Product, and New Constant-Degree Expanders and Extractors. *Annals of Mathematics*, Vol. 155 (1), pages 157–187, 2001. Preliminary version in *41st FOCS*, pages 3–13, 2000.

[179] H.G. Rice. Classes of Recursively Enumerable Sets and their Decision Problems. *Trans. AMS*, Vol. 89, pages 25–59, 1953.

[180] R.L. Rivest, A. Shamir and L.M. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *CACM*, Vol. 21, Feb. 1978, pages 120–126.

[181] D. Ron. Property testing. In *Handbook on Randomization, Volume II*, pages 597–649, 2001. (Editors: S. Rajasekaran, P.M. Pardalos, J.H. Reif and J.D.P. Rolim.)

[182] R. Rubinfeld and M. Sudan. Robust characterization of polynomials with applications to program testing. *SIAM Journal on Computing*, Vol. 25 (2), pages 252–271, 1996.

[183] M. Saks and S. Zhou. $\mathrm{RSPACE}(S) \subseteq \mathrm{DSPACE}(S^{3/2})$. In 36th *IEEE Symposium on Foundations of Computer Science*, pages 344–353, 1995.

[184] W.J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *JCSS*, Vol. 4 (2), pages 177-192, 1970.

[185] A. Selman. On the structure of NP. *Notices Amer. Math. Soc.*, Vol. 21 (6), page 310, 1974.

[186] R. Shaltiel. Recent Developments in Explicit Constructions of Extractors. *Bulletin of the EATCS 77*, pages 67–95, 2002.

[187] R. Shaltiel and C. Umans. Simple Extractors for All Min-Entropies and a New Pseudo-Random Generator. In *42nd IEEE Symposium on Foundations of Computer Science*, pages 648–657, 2001.

[188] C.E. Shannon. A Symbolic Analysis of Relay and Switching Circuits. *Trans. American Institute of Electrical Engineers*, Vol. 57, pages 713–723, 1938.

[189] C.E. Shannon. A mathematical theory of communication. *Bell Sys. Tech. Jour.*, Vol. 27, pages 623–656, 1948.

[190] C.E. Shannon. Communication Theory of Secrecy Systems. *Bell Sys. Tech. Jour.*, Vol. 28, pages 656–715, 1949.

[191] A. Shamir. IP = PSPACE. *Journal of the ACM*, Vol. 39, No. 4, pages 869–877, 1992. Preliminary version in *31st FOCS*, 1990.

[192] A. Shpilka. Lower Bounds for Matrix Product. *SIAM Journal on Computing*, pages 1185-1200, 2003.

[193] M. Sipser. A Complexity Theoretic Approach to Randomness. In *15th ACM Symposium on the Theory of Computing*, pages 330–335, 1983.

[194] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[195] R. Smolensky. Algebraic Methods in the Theory of Lower Bounds for Boolean Circuit Complexity. In *19th ACM Symposium on the Theory of Computing* pages 77–82, 1987.

[196] R.J. Solomonoff. A Formal Theory of Inductive Inference. *Information and Control*, Vol. 7/1, pages 1–22, 1964.

[197] R. Solovay and V. Strassen. A Fast Monte-Carlo Test for Primality. *SIAM Journal on Computing*, Vol. 6, pages 84–85, 1977. Addendum in *SIAM Journal on Computing*, Vol. 7, page 118, 1978.

[198] D.A. Spielman. *Advanced Complexity Theory*, Lectures 10 and 11. Notes (by D. Lewin and S. Vadhan), March 1997. Available from `http://www.cs.yale.edu/homes/spielman/AdvComplexity/1998/` as `lect10.ps` and `lect11.ps`.

[199] L.J. Stockmeyer. The Polynomial-Time Hierarchy. *Theoretical Computer Science*, Vol. 3, pages 1–22, 1977.

[200] L. Stockmeyer. The Complexity of Approximate Counting. In *15th ACM Symposium on the Theory of Computing*, pages 118–126, 1983.

[201] V. Strassen. Algebraic Complexity Theory. In *Handbook of Theoretical Computer Science: Volume A – Algorithms and Complexity*, J. van Leeuwen editor, MIT Press/Elsevier, 1990, pages 633–672.

[202] M. Sudan. Decoding of Reed Solomon codes beyond the error-correction bound. *Journal of Complexity*, Vol. 13 (1), pages 180–193, 1997.

[203] M. Sudan. Algorithmic introduction to coding theory. Lecture notes, Available from `http://theory.csail.mit.edu/~madhu/FT01/`, 2001.

[204] , M. Sudan, L. Trevisan, and S. Vadhan. Pseudorandom generators without the XOR Lemma. *Journal of Computer and System Science*, Vol. 62, No. 2, pages 236–266, 2001.

[205] R. Szelepcsenyi. A Method of Forced Enumeration for Nondeterministic Automata. *Acta Informatica*, Vol. 26, pages 279–284, 1988.

[206] S. Toda. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, Vol. 20 (5), pages 865–877, 1991.

[207] B.A. Trakhtenbrot. A Survey of Russian Approaches to *Perebor* (Brute Force Search) Algorithms. *Annals of the History of Computing*, Vol. 6 (4), pages 384–398, 1984.

[208] L. Trevisan. Constructions of Near-Optimal Extractors Using Pseudo-Random Generators. In *31st ACM Symposium on the Theory of Computing*, pages 141–148, 1998.

[209] C.E. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. Londom Mathematical Soceity*, Ser. 2, Vol. 42, pages 230–265, 1936. A Correction, *ibid.*, Vol. 43, pages 544–546.

[210] C. Umans. Pseudo-random generators for all hardness. *Journal of Computer and System Science*, Vol. 67 (2), pages 419–440, 2003.

[211] S. Vadhan. A Study of Statistical Zero-Knowledge Proofs. PhD Thesis, Department of Mathematics, MIT, 1999. Available from `http://www.eecs.harvard.edu/~salil/papers/phdthesis-abs.html`.

[212] S. Vadhan. An Unconditional Study of Computational Zero Knowledge. In *45th IEEE Symposium on Foundations of Computer Science*, pages 176–185, 2004.

[213] L.G. Valiant. The Complexity of Computing the Permanent. *Theoretical Computer Science*, Vol. 8, pages 189–201, 1979.

[214] L.G. Valiant. A theory of the learnable. *CACM*, Vol. 27/11, pages 1134–1142, 1984.

[215] L.G. Valiant and V.V. Vazirani. NP Is as Easy as Detecting Unique Solutions. *Theoretical Computer Science*, Vol. 47 (1), pages 85–93, 1986.

[216] J. von Neumann, First Draft of a Report on the EDVAC, 1945. Contract No. W-670-ORD-492, Moore School of Electrical Engineering, Univ. of Penn., Philadelphia. Reprinted (in part) in *Origins of Digital Computers: Selected Papers*, Springer-Verlag, Berlin Heidelberg, pages 383–392, 1982.

[217] J. von Neumann, Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100, pages 295–320, 1928.

[218] I. Wegener. *The Complexity of Boolean Functions.* Wiley-Teubner, 1987.

[219] I. Wegener. *Branching Programs and Binary Decision Diagrams – Theory and Applications.* SIAM Monographs on Discrete Mathematics and Applications, 2000.

[220] A. Wigderson. The amazing power of pairwise independence. In *26th ACM Symposium on the Theory of Computing*, pages 645–647, 1994.

[221] A.C. Yao. Theory and Application of Trapdoor Functions. In *23rd IEEE Symposium on Foundations of Computer Science*, pages 80–91, 1982.

[222] A.C. Yao. Separating the Polynomial-Time Hierarchy by Oracles. In *26th IEEE Symposium on Foundations of Computer Science*, pages 1-10, 1985.

[223] A.C. Yao. How to Generate and Exchange Secrets. In *27th IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.

[224] D. Zuckerman. Simulating BPP Using a General Weak Random Source. *Algorithmica*, Vol. 16, pages 367–391, 1996.

[225] D. Zuckerman. Randomness-Optimal Oblivious Sampling. *Journal of Random Structures and Algorithms*, Vol. 11, Nr. 4, December 1997, pages 345–367.

Index