Introduction to Complexity Theory
Lecture Notes for First Semester

Oded Goldreich
Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, ISRAEL.
Email: oded@wisdom.weizmann.ac.il

March 1999

©Copyright 1999 by Oded Goldreich.
Permission to make copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that new copies bear this notice and the full citation on the first page. Abstracting with credit is permitted.

II

Preface

Complexity Theory is a central field of Theoretical Computer Science, with a remarkable list of celebrated achievements as well as a very vibrant present research activity. The field is concerned with the study of the *intrinsic* complexity of computational tasks, and this study tend to aim at *generality*: It focuses on natural computational resources, and the effect of limiting those on the *class of problems* that can be solved.

These lecture notes were taken by students attending my year-long introductory course on Complexity Theory, given in 1998–99 at the Weizmann Institute of Science. The course was aimed at exposing the students to the basic results and research directions in the field. Specific topics included:

- revisiting NP and NPC (with emphasis on search vs decision);
- complexity classes defined by one resource-bound – hierarchies, gaps, etc;
- non-deterministic space complexity (with emphasis on NL);
- randomized computations (e.g., ZPP, RP and BPP);
- non-uniform complexity (e.g., P/poly, and lower bounds on restricted circuit classes);
- the Polynomial-time Hierarchy;
- the classes #P and approximate-#P and uniqueSAT;
- probabilistic proof systems (IP, PCP and ZK);
- pseudorandomness (generators and derandomization);
- time versus space (in Turing Machines);
- circuit-depth versus TM-space (e.g., NC, SC);
- average-case complexity;

It was assumed that students have taken a course in computability, and hence are familiar with Turing Machines.

State of these notes

These notes are neither complete nor fully proofread, let alone being far from uniformly well-written (although the notes of some lectures are quite good). Still, I do believe that these notes suggest a good outline for an *introduction to complexity theory* course.

The current notes include only lectures given in the first semester. These will be hopefully augmented within half a year, by notes for the second semester.

Using these notes

The first 13 lectures were given in the first semester. The pace was rather slow. In some circumstances (e.g., when the student body is uniformly more advanced) one should be able to cover much more in one semester.

Bibliographic Notes

There are several books which cover small parts of the material. These include:

1. Garey, M.R., and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979.

2. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. Algorithms and Combinatorics series (Vol. 17), *Springer*, 1998. Copies have been placed in the faculty's library.

3. J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

4. M. Sipser. *Introduction to the Theory of Computation*, PWS Publishing Company, 1997.

However, the presentation of material in these lecture notes does not necessarily follow these sources.

Each lecture is planned to include bibliographic notes, but this intension has been only partially fulfilled so far.

Acknowledgments

I am most grateful to the students who have attended the course and partipiated in the project of preparing the lecture notes. So thanks to *Sergey Benditkis, Reshef Eilon, Michael Elkin, Amiel Ferman, Dana Fisman, Danny Harnik, Tzvika Hartman, Tal Hassner, Hillel Kugler, Oded Lachish, Moshe Lewenstein, Yehuda Lindell, Yoad Lustig, Ronen Mizrahi, Leia Passoni, Guy Peer, Nir Piterman, Ely Porate, Yoav Rodeh, Alon Rosen, Vered Rosen, Noam Sadot, Il'ya Safro, Tamar Seeman, Ekaterina Sedletsky, Reuben Sumner, Yael Tauman, Boris Temkin. Erez Waisbard,* and *Gera Weiss.*

Thanks also to *Paul Beame*, *Ruediger Reischuk* and *Avi Wigderson* who have answered some questions I've had while preparing this course.

Lecture Summaries

Lecture 1: The P vs NP Question. We review the fundamental question of computer science, known as the $\mathcal{P}$ versus $\mathcal{NP}$ question: Given a problem whose solution can be verified efficiently (i.e., in polynomial time), is there necessarily an efficient method to actually *find* such a solution? Loosely speaking, the first condition (i.e., efficient verification) is captured in the definition of $\mathcal{NP}$, and the second in that of $\mathcal{P}$. The actual correspondence relies on the notion of *self-reducibility*, which relates the complexity of determining whether a solution exists to the complexity of actually finding one.

Notes taken by Eilon Reshef.

Lecture 2: NP-completeness and Self Reducibility. We prove that any relation defining an NP-complete language is self-reducible. This will be done using the SAT self-reducibility (proved in Lecture 1), and the fact that SAT is NP-Hard under Levin Reductions. The latter are Karp Reductions augmented by efficient transformations of NP-witnesses from the original instance to the reduced one, and vice versa. Along the way, we give a simple proof of the existence of NP-Complete languages (by proving that Bounded Halting is NP-Complete).

Notes taken by Nir Piterman and Dana Fisman.

Lecture 3: More on NP and some on DTIME. In the first part of this lecture we discuss two properties of the complexity classes P, NP and NPC: The property is that NP contains problems which are neither NP-complete nor in P (provided NP $\neq$ P), and the second one is that NP-relations have optimal search algorithms. In the second part we define new complexity classes based on exact time bounds, and consider some relations between them. We point out the sensitivity of these classes to the specific model of computation (e.g., one-tape versus two-tape Turing machines).

Notes taken by Michael Elkin and Ekaterina Sedletsky.

Lecture 4: Space Complexity. We define "nice" complexity bounds; these are bounds which can be computed within the resources they supposedly bound (e.g., we focus on time-constructible and space-constructible bounds). We define space complexity using an adequate model of computation in which one is not allowed to use the area occupied by the input for computation. Before dismissing sub-logarithmic space, we present two results regarding it (contrasting sub-loglog space with loglog space). We show that for "nice" complexity bounds, there is a hierarchy of complexity classes – the more resources one has the more tasks one can perform. One the other hand, we mention that this increase in power may not happen if the complexity bounds are not "nice".

Notes taken by Leia Passoni and Reuben Sumner.

Lecture 5: Non-Deterministic Space. We recall two basic facts about deterministic space complexity, and then define non-deterministic space complexity. Three alternative models for measuring non-deterministic space complexity are introduced: the standard non-deterministic model, the online model and the offline model. The equivalence between the non-deterministic and online models and their exponential relation to the offline model are proved. We then turn to investigate the relation between the non-deterministic and deterministic space complexity (i.e., Savitch's Theorem).

Notes taken by Yoad Lustig and Tal Hassner.

Lecture 6: Non-Deterministic Logarithmic Space We further discuss composition lemmas underlying previous lectures. Then we study the complexity class $\mathcal{NL}$ (the set of languages decidable within Non-Deterministic Logarithmic Space): We show that directed graph connectivity is complete for $\mathcal{NL}$. Finally, we prove that $\mathcal{NL} = \text{co}\mathcal{NL}$ (i.e., $\mathcal{NL}$ class is closed under complementation).

Notes taken by Amiel Ferman and Noam Sadot.

Lecture 7: Randomized Computations We extend the notion of efficient computation by allowing algorithms (Turing machines) to toss coins. We study the classes of languages that arise from various natural definitions of acceptance by such machines. We focus on probabilistic polynomial-time machines with one-sided, two-sided and zero error probability (defining the classes $\mathcal{RP}$ (and co$\mathcal{RP}$), $\mathcal{BPP}$ and $\mathcal{ZPP}$). We also consider probabilistic machines that uses logarithmic spaces (i.e., the class $\mathcal{RL}$).

Notes taken by Erez Waisbard and Gera Weiss.

Lecture 8: Non-Uniform Polynomial Time ($\mathcal{P}$/Poly). We introduce the notion of non-uniform polynomial-time and the corresponding complexity class $\mathcal{P}$/poly. In this (somewhat fictitious) computational model, Turing machines are provided an external advice string to aid them in their computation (on strings of certain length). The *non-uniformity* is expressed in the fact that an arbitrary advice string may be defined for every different length of input. We show that $\mathcal{P}$/poly "upper bounds" the notion of efficient computation (as $\mathcal{BPP} \subseteq \mathcal{P}$/poly), yet this upper bound is not tight (as $\mathcal{P}$/poly contains non-recursive languages). The effect of introducing *uniformity* is discussed, and shown to collapse $\mathcal{P}$/poly to $\mathcal{P}$. Finally, we relate the $\mathcal{P}$/poly versus $\mathcal{NP}$ question to the question of whether NP-completeness via Cook-reductions is more powerful that NP-completeness via Karp-reductions. This is done by showing, on one hand, that $\mathcal{NP}$ is Cook-reducible to a sparse set iff $\mathcal{NP} \subset \mathcal{P}$/poly, and on the other hand that $\mathcal{NP}$ is Karp-reducible to a sparse set iff $\mathcal{NP} = \mathcal{P}$.

Notes taken by Moshe Lewenstein, Yehuda Lindell and Tamar Seeman.

Lecture 9: The Polynomial Hierarchy (PH). We define a hierarchy of complexity classes extending $\mathcal{NP}$ and contained in PSPACE. This is done in two ways, shown equivalent: The first by generalizing the notion of Cook reductions, and the second by generalizing the definition of $\mathcal{NP}$. We then relate this hierarchy to complexity classes discussed in previous lectures such as $\mathcal{BPP}$ and $\mathcal{P}$/Poly: We show that $\mathcal{BPP}$ is in PH, and that if $\mathcal{NP} \subseteq \mathcal{P}$/poly then PH collapses to is second level.

Notes taken by Ronen Mizrahi.

Lecture 10: The counting class $\#\mathcal{P}$. The class $\mathcal{NP}$ captures the difficulty of determining whether a given input has a solution with respect to some (tractable) relation. A potentially harder question, captured by the class $\#\mathcal{P}$, refers to determining the number of such solutions. We first define the complexity class $\#\mathcal{P}$, and classify it with respect to other complexity classes. We then prove the existence of $\#\mathcal{P}$-complete problems, and mention some natural ones. Then we try to study the relation between $\#\mathcal{P}$ and $\mathcal{NP}$ more exactly, by showing we can probabilistically approximate $\#\mathcal{P}$ using an oracle in $\mathcal{NP}$. Finally, we refine this result by restricting the oracle to a weak form of SAT (called $uniqueSAT$).

Notes taken by Oded Lachish, Yoav Rodeh and Yael Tauman.

Lecture 11: Interactive Proof Systems. We introduce the notion of interactive proof systems and the complexity class IP, emphasizing the role of randomness and interaction in this model. The concept is demonstrated by giving an interactive proof system for Graph Non-Isomorphism. We discuss the power of the class IP, and prove that co$\mathcal{NP} \subseteq \mathcal{IP}$. We discuss issues regarding the number of rounds in a proof system, and variants of the model such as public-coin systems (a.k.a. Arthur-Merlin games).

Notes taken by Danny Harnik, Tzvika Hartman and Hillel Kugler.

Lecture 12: Probabilistically Checkable Proof (PCP). We introduce the notion of Probabilistically Checkable Proof (PCP) systems. We discuss some complexity measures involved, and describe the class of languages captured by corresponding PCP systems. We then demonstrate the alternative view of $\mathcal{NP}$ emerging from the PCP Characterization Theorem, and use it in order to prove non-approximability results for the problems $max3SAT$ and $maxCLIQUE$.

Notes taken by Alon Rosen and Vered Rosen.

Lecture 13: Pseudorandom Generators. Pseudorandom generators are defined as efficient deterministic algorithms which stretch short random seeds into longer pseudorandom sequences. The latter are indistiguishable from truely random sequences by any efficient observer. We show that, for efficiently sampleable distributions, computational indistiguishability is preserved under multiple samples. We related pseudorandom generators and one-way functions, and show how to increase the stretching of pseudorandom generators. The notes are augmented by an essay of Oded.

Notes taken by Sergey Benditkis, Il'ya Safro and Boris Temkin.

X

Contents

Preface **III**

Acknowledgments **V**

Lecture Summaries **VII**

1 The P vs NP Question **1**
1.1 Introduction . . . 1
1.2 The Complexity Class $\mathcal{NP}$ . . . 1
1.3 Search Problems . . . 3
1.4 Self Reducibility . . . 4
Bibliographic Notes . . . 6

2 NP-completeness and Self Reducibility **9**
2.1 Reductions . . . 9
2.2 All $\mathcal{NP}$-complete relations are Self-reducible . . . 11
2.3 $B_{ounded}H_{alting}$ is $\mathcal{NP}-$complete . . . 13
2.4 $C_{ircuit}S_{atisfiability}$ is $\mathcal{NP}-$complete . . . 14
2.5 $R_{\mathcal{SAT}}$ *is* $\mathcal{NP}-$complete . . . 17
Bibliographic Notes . . . 18
Appendix: Details for the reduction of BH to CS . . . 18

3 More on NP and some on DTIME **23**
3.1 Non-complete languages in NP . . . 23
3.2 Optimal algorithms for NP . . . 25
3.3 General Time complexity classes . . . 27
3.3.1 The DTime classes . . . 27
3.3.2 Time-constructibility and two theorems . . . 29
Bibliographic Notes . . . 31
Appendix: Proof of Theorem 3.5, via crossing sequences . . . 31

4 Space Complexity **35**
4.1 On Defining Complexity Classes . . . 35
4.2 Space Complexity . . . 35
4.3 Sub-Logarithmic Space Complexity . . . 36
4.4 Hierarchy Theorems . . . 39
4.5 Odd Phenumena (The Gap and Speed-Up Theorems) . . . 42
Bibliographic Notes . . . 42

5 Non-Deterministic Space **43**
5.1 Preliminaries . . . 43
5.2 Non-Deterministic space complexity . . . 44
5.2.1 Definition of models (online vs offline) . . . 45
5.2.2 Relations between $NSPACE_{on}$ and $NSPACE_{off}$ . . . 47
5.3 Relations between Deterministic and Non-Deterministic space . . . 53
5.3.1 Savitch's Theorem . . . 53
5.3.2 A translation lemma . . . 54
Bibliographic Notes . . . 56

6 Inside Non-Deterministic Logarithmic Space **57**
6.1 The composition lemma . . . 57
6.2 A complete problem for $\mathcal{NL}$ . . . 59
6.2.1 Discussion of Reducibility . . . 59
6.2.2 The complete problem: directed-graph connectivity . . . 61
6.3 Complements of complexity classes . . . 64
6.4 Immerman Theorem: $\mathcal{NL} = \text{co}\mathcal{NL}$ . . . 65
6.4.1 Theorem 6.9 implies $\mathcal{NL} = \text{co}\mathcal{NL}$ . . . 66
6.4.2 Proof of Theorem 6.9 . . . 68
Bibliographic Notes . . . 71

7 Randomized Computations **73**
7.1 Probabilistic computations . . . 73
7.2 The classes RP and $coRP$ – One-Sided Error . . . 75
7.3 The class BPP – Two-Sided Error . . . 79
7.4 The class PP . . . 83
7.5 The class ZPP – Zero error probability. . . . 86
7.6 Randomized space complexity . . . 87
7.6.1 The definition . . . 87
7.6.2 Undirected Graph Connectivity is in RL . . . 89
Bibliographic Notes . . . 90

8 Non-Uniform Polynomial Time ($\mathcal{P}$/Poly) **91**
8.1 Introduction . . . 91
8.1.1 The Actual Definition . . . 92
8.1.2 $\mathcal{P}$/poly and the $\mathcal{P}$ versus $\mathcal{NP}$ Question . . . 93
8.2 The Power of $\mathcal{P}$/poly . . . 93
8.3 Uniform Families of Circuits . . . 95
8.4 Sparse Languages and the $\mathcal{P}$ versus $\mathcal{NP}$ Question . . . 95
Bibliographic Notes . . . 99

9 The Polynomial Hierarchy (PH) **101**
9.1 The Definition of the class PH . . . 101
9.1.1 First definition for PH: via oracle machines . . . 101
9.1.2 Second definition for PH: via quantifiers . . . 104
9.1.3 Equivalence of definitions . . . 105
9.2 Easy Computational Observations . . . 107
9.3 BPP is contained in PH . . . 109

9.4 If NP has small circuits then PH collpases . . . 111
Bibliographic Notes . . . 112
Appendix: Proof of Proposition 9.2.3 . . . 113

10 The counting class $\#\mathcal{P}$ 115
10.1 Defining $\#\mathcal{P}$ . . . 115
10.2 Completeness in #P . . . 117
10.3 How close is $\#\mathcal{P}$ to $\mathcal{NP}$? . . . 122
10.3.1 Various Levels of Approximation . . . 123
10.3.2 Probabilistic Cook Reduction . . . 126
10.3.3 $Gap_8\#SAT$ Reduces to SAT . . . 127
10.4 Reducing to $uniqueSAT$ . . . 130
Bibliographic Notes . . . 133
Appendix A: A Family of $Universal_2$ Hash Functions . . . 133
Appendix B: Proof of Leftover Hash Lemma . . . 134

11 Interactive Proof Systems 135
11.1 Introduction . . . 135
11.2 The Definition of IP . . . 136
11.2.1 Comments . . . 137
11.2.2 Example – Graph Non-Isomorphism (GNI) . . . 138
11.3 The Power of IP . . . 140
11.3.1 IP is contained in PSPACE . . . 140
11.3.2 coNP is contained in IP . . . 142
11.4 Public-Coin Systems and the Number of Rounds . . . 145
11.5 Perfect Completeness and Soundness . . . 146
Bibliographic Notes . . . 148

12 Probabilistically Checkable Proof Systems 149
12.1 Introduction . . . 149
12.2 The Definition . . . 150
12.2.1 The basic model . . . 150
12.2.2 Complexity Measures . . . 150
12.2.3 Some Observations . . . 151
12.3 The PCP characterization of NP . . . 152
12.3.1 Importance of Complexity Parameters in PCP Systems . . . 152
12.3.2 The PCP Theorem . . . 152
12.3.3 The PCP Theorem gives rise to "robust" $\mathcal{NP}$-relations . . . 154
12.3.4 Simplifying assumptions about $\mathcal{PCP}(\log, O(1))$ verifiers . . . 155
12.4 PCP and non-approximability . . . 156
12.4.1 Amplifying Reductions . . . 156
12.4.2 PCP Theorem Rephrased . . . 157
12.4.3 Connecting PCP and non-approximability . . . 159
Bibliographic Notes . . . 163

13 Pseudorandom Generators **165**
13.1 Instead of an introduction . . . 165
13.2 Computational Indistinguishability . . . 165
13.2.1 Two variants . . . 166
13.2.2 Relation to Statistical Closeness . . . 166
13.2.3 Computational indistinguishability and multiple samples . . . 167
13.3 PRG: Definition and amplification of the stretch function . . . 168
13.4 On Using Pseudo-Random Generators . . . 171
13.5 Relation to one-way functions . . . 172
Bibliographic Notes . . . 175
Appendix: An essay by O.G. . . . 176
13.6.1 Introduction . . . 176
13.6.2 The Definition of Pseudorandom Generators . . . 177
13.6.3 How to Construct Pseudorandom Generators . . . 180
13.6.4 Pseudorandom Functions . . . 182
13.6.5 The Applicability of Pseudorandom Generators . . . 183
13.6.6 The Intelectual Contents of Pseudorandom Generators . . . 184
13.6.7 A General Paradigm . . . 185
References . . . 185

Lecture 1

The P vs NP Question

Notes taken by Eilon Reshef

Summary: We review the fundamental question of computer science, known as $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$: given a problem whose solution can be verified efficiently (i.e., in polynomial time), is there necessarily an efficient method to actually *find* such a solution? First, we define the notion of $\mathcal{NP}$, i.e., the class of all problems whose solution can be verified in polynomial-time. Next, we discuss how to represent search problems in the above framework. We conclude with the notion of *self-reducibility*, relating the hardness of determining whether a feasible solution exists to the hardness of actually finding one.

1.1 Introduction

Whereas the research in complexity theory is still in its infancy, and many more questions are open than closed, many of the concepts and results in the field have an extreme conceptual importance and represent significant intellectual achievements.

Of the more fundamental questions in this area is the relation between different flavors of a problem: the *search* problem, i.e., finding a feasible solution, the *decision* problem, i.e., determining whether a feasible solution exists, and the *verification* problem, i.e., deciding whether a given solution is correct.

To initiate a formal discussion, we assume basic knowledge of elementary notions of computability, such as Turning machines, reductions, polynomial-time computability, and so on.

1.2 The Complexity Class $\mathcal{NP}$

In this section we recall the definition of the complexity class $\mathcal{NP}$ and overview some of its basic properties. Recall that the complexity class $\mathcal{P}$ is the collection of all languages L that can be recognized "efficiently", i.e., by a deterministic polynomial-time Turing machine. Whereas the traditional definition of $\mathcal{NP}$ associates the class $\mathcal{NP}$ with the collection of languages that can be efficiently recognized by a *non-deterministic* Turning machine, we provide an alternative definition, that in our view better captures the conceptual contents of the class.

Informally, we view $\mathcal{NP}$ as the class of all languages that admit a short "certificate" for membership in the language. Given this certificate, called a *witness*, membership in the language can be verified efficiently, i.e., in polynomial time.

For the sake of self-containment, we recall that a (binary) relation R is *polynomial-time decidable* if there exists a polynomial-time Turing machine that accepts the language $\{E(x,y) \mid (x,y) \in R\}$, where $E(x,y)$ is a unique encoding of the pair (x,y). An example of such an encoding is $E(\sigma_1 \cdots \sigma_n, \tau_1 \cdots \tau_m) \triangleq \sigma_1\sigma_1 \cdots \sigma_n\sigma_n 01\tau_1\tau_1 \cdots \tau_n\tau_n$.

We are now ready to introduce a definition of $\mathcal{NP}$.

Definition 1.1 *The complexity class $\mathcal{NP}$ is the class of all languages L for which there exists a relation $R_L \subseteq \{0,1\}^* \times \{0,1\}^*$, such that*

- *R_L is polynomial-time decidable.*
- *There exists a polynomial b_L such that $x \in L$ if and only if there exists a* witness *w, $|w| \leq b_L(|x|)$ for which $(x,w) \in R_L$.*

Note that the polynomial bound in the second condition is required despite the fact that R_L is polynomial-time decidable, since the polynomiality of R_L is measured with respect to the length of the pair (x,y), and not with respect to $|x|$ only.

It is important to note that if x is not in L, there is no polynomial-size witness w for which $(x,w) \in R_L$. Also, the fact that $(x,y) \notin R_L$ does *not* imply that $x \notin L$, but rather that y is not a proper witness for x.

A slightly different definition may sometimes be convenient. This definition allows only *polynomially-bounded* relations, i.e.,

Definition 1.2 *A relation R is* polynomially bounded *if there exists a polynomial $p(\cdot)$, such that for every $(x,y) \in R$, $|y| \leq p(|x|)$.*

Since a composition of two polynomials is also a polynomial, any polynomial in $p(|x|)$, where p is a polynomial, is also polynomial in $|x|$. Thus, if a polynomially-bounded relation R can be decided in polynomial-time, it can also be decided in time polynomial in the size of first element in the pair $(x,y) \in R$.

Now, definition 1.1 of $\mathcal{NP}$ can be also formulated as:

Definition 1.3 *The complexity class $\mathcal{NP}$ is the class of all languages L for which there exists a* polynomially-bounded *relation $R_L \subseteq \{0,1\}^* \times \{0,1\}^*$, such that*

- *R_L is polynomial-time decidable.*
- *$x \in L$ if and only if there exists a* witness *w, for which $(x,w) \in R_L$.*

In this view, the fundamental question of computer science, i.e., $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ can be formulated as the question whether the *existence* of a short witness (as implied by membership in $\mathcal{NP}$) necessarily brings about an efficient algorithm for finding such a witness (as required for membership in $\mathcal{P}$).

To relate our definitions to the traditional definition of $\mathcal{NP}$ in terms of a non-deterministic Turning machine, we show that the definitions above indeed represent the same complexity class.

Proposition 1.2.1 *$\mathcal{NP}$ (as in definition 1.1) = $\mathcal{NP}$ (as in the traditional definition).*

Proof: First, we show that if a language L is in $\mathcal{NP}$ according to the traditional definition, then it is also in $\mathcal{NP}$ according to definition 1.1.

Consider a non-deterministic Turing machine $\tilde{M}_L$ that decides on L after at most $p_L(|x|)$ steps, where p_L is some polynomial depending on L, and x is the input to $\tilde{M}_L$. The idea is that one can

encode the non-deterministic choices of $\tilde{M}_L$, and to use this encoding as a witness for membership in L. Namely, $\tilde{M}_L$ can always be assumed to first make all its non-deterministic choices (e.g., by writing them on a separate tape), and then execute deterministically, branching according to the choices that had been made in the first step. Thus, $\tilde{M}_L$ is equivalent to a deterministic Turning machine M_L accepting as input the pair (x, y) and executing exactly as $\tilde{M}_L$ on x with a predetermined sequence of non-deterministic choices encoded by y. An input x is accepted by $\tilde{M}_L$ if and only if there exists a y for which (x, y) is accepted by M_L.

The relation R_L is defined to be the set of all pairs (x, y) accepted by M_L.

Thus, $x \in L$ if and only if there exists a y such that $(x, y) \in R_L$, namely if there exists an accepting computation of M_L. It remains to see that R_L is indeed polynomial-time decidable and polynomially bounded. For the first part, observe that R_L can be decided in polynomial time simply by simulating the Turing machine M_L on (x, y). For the second part, observe that $\tilde{M}_L$ is guaranteed to terminate in polynomial time, i.e., after at most $p_L(|x|)$ steps, and therefore the number of non-deterministic choices is also bounded by a polynomial, i.e., $|y| \leq p_L(|x|)$. Hence, the relation R_L is polynomially bounded.

For the converse, examine the witness relation R_L as in definition 1.1. Consider the polynomial-time deterministic Turing machine M_L that decides on R_L, i.e., accepts the pair (x, y) if and only if $(x, y) \in R_L$. Construct a non-deterministic Turning machine $\tilde{M}_L$ that given an input x, guesses, non-deterministically, a witness y of size $b_L(|x|)$, and then executes M_L on (x, y). If $x \in L$, there exists a polynomial-size witness y for which $(x, y) \in R_L$, and thus there exists a polynomial-time computation of $\tilde{M}_L$ that accepts x. If $x \notin L$, then for every polynomial-size witness y, $(x, y) \notin R_L$ and therefore $\tilde{M}_L$ always rejects x. ∎

1.3 Search Problems

Whereas the definition of computational power in terms of languages may be mathematically convenient, the main computational goal of computer science is to solve "problems". We abstract a computation problem Π by a *search problem* over some binary relation R_Π: the input of the problem at hand is some x and the task is to find a y such that $(x, y) \in R_\Pi$ (we ignore the case where no such y exists).

A particularly interesting subclass of these relations is the collection of *polynomially verifiable* relations R for which

- R is polynomially bounded. Otherwise, the mere writing of the solution cannot be carried out efficiently.
- R is polynomial-time recognizable. This captures the intuitive notion that once a solution to the problem is given, one should be able to verify its correctness efficiently (i.e., in polynomial time). The lack of such an ability implies that even if a solution is provided "by magic", one cannot efficiently determine its validness.

Given a polynomially-verifiable relation R, one can define the corresponding language $L(R)$ as the set of all words x for which there *exists* a solution y, such that $(x, y) \in R$, i.e.,

$$L(R) \triangleq \{x \mid \exists y \ (x, y) \in R\}. \tag{1.1}$$

By the above definition, $\mathcal{NP}$ is exactly the collection of the languages $L(R)$ that correspond to search problems over polynomially verifiable relations, i.e.,

$$\mathcal{NP} \triangleq \{L(R) \mid R \text{ is polynomially verifiable}\}$$

Thus, the question $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ can be rephrased as the question whether for every polynomially verifiable relation R, its corresponding language $L(R)$ can be decided in polynomial time.

Following is an example of a computational problem and its formulation as a search problem.

PROBLEM: 3-Coloring Graphs
INPUT: An undirected graph $G = (V, E)$.
TASK: Find a 3-coloring of G, namely a mapping $\varphi : V \to \{1, 2, 3\}$ such that no adjacent vertices have the same color, i.e., for every $(u, v) \in E$, $\varphi(u) \neq \varphi(v)$.

The natural relation R_{3COL} that corresponds to 3-Coloring is defined over the set of pairs (G, φ), such that $(G, \varphi) \in R_{3COL}$ if

- φ is indeed a mapping $\varphi : V \to \{1, 2, 3\}$.
- For every $(u, v) \in E$, $\varphi(u) \neq \varphi(v)$.

Clearly, with any reasonable representation of φ, its size is polynomial in the size of G. Further, it is easy to determine in polynomial time whether a pair (G, φ) is indeed in R_{3COL}.

The corresponding language $L(R_{3COL})$ is the set of all 3-colorable graphs, i.e., all graphs G that have a legal 3-coloring.

Jumping ahead, it is $\mathcal{NP}$-hard to determine whether such a coloring exists, and hence, unless $\mathcal{P} = \mathcal{NP}$, no efficient algorithm for this problem exists.

1.4 Self Reducibility

Search problems as defined above are "harder" than the corresponding decision problem in the sense that if the former can be carried out efficiently, so can the latter. Given a polynomial-time search algorithm $\mathcal{A}$ for a polynomially-verifiable relation R, one can construct a polynomial-time decision algorithm for $L(R)$ by simulating $\mathcal{A}$ for polynomially many steps, and answering "yes" if and only if $\mathcal{A}$ has terminated and produced a proper y for which $(x, y) \in R$.

Since much of the research in complexity theory evolves around decision problems, a fundamental question that naturally arises is whether an efficient procedure for solving the *decision* problem guarantees an efficient procedure for solving the *search* problem. As will be seen below, this is not known to be true in general, but can be shown to be true for any $\mathcal{NP}$-complete problem.

We begin with a definition that captures this notion:

Definition 1.4 *A relation R is* self-reducible *if solving the search problem for R is Cook-reducible to deciding the corresponding language $L(R) \stackrel{\triangle}{=} \{x \mid \exists y \ (x, y) \in R\}$.*

Recall that a Cook reduction from a problem Π_1 to Π_2 allows a Turing machine for Π_1 to use Π_2 as an oracle (polynomially many times).

Thus, if a relation R is self-reducible, then there exists a polynomial-time Turing machine that solves the search problem (i.e., for each input x finds a y such that $(x, y) \in R$), except that the Turning machine is allowed to access an oracle that decides $L(R)$, i.e., for each input x' outputs whether there exists a y' such that $(x', y') \in R$. For example, in the case of 3-colorability, the search algorithm is required to *find* a 3-coloring for an input graph G, given as an oracle a procedure that tells *whether* a given graph G' is 3-colorable. The search algorithm is not limited to ask the oracle only about G, but rather may query the oracle on a (polynomially long) sequence of graphs G', where the sequence itself may depend upon answers to previous invocations of the oracle.

We consider the example of SAT.

PROBLEM: SAT
INPUT: A CNF formula φ over $\{x_1, \ldots, x_n\}$.
TASK: Find a satisfying assignment σ, i.e., a mapping $\sigma : \{1, \ldots, n\} \to \{T, F\}$, such that $\varphi(\sigma(1), \ldots, \sigma(n))$ is true.

The relation R_{SAT} corresponding to SAT is the set of all pairs (φ, σ) such that σ is a satisfying assignment for φ. It can be easily verified that the length of σ is indeed polynomial in n and that the relation can be recognized in polynomial time.

Proposition 1.4.1 *R_{SAT} is self-reducible.*

Proof: We show that R_{SAT} is self-reducible by showing an algorithm that solves the search problem over R_{SAT} using an oracle $\mathcal{A}$ for deciding $SAT \stackrel{\triangle}{=} L(R_{SAT})$. The algorithm incrementally constructs a solution by building partial assignments. At each step, the invariant guarantees that the partial assignment can be completed into a full satisfying assignment, and hence when the algorithm terminates, the assignment satisfies φ. The algorithm proceeds as follows.

- Query whether $\varphi \in$ SAT. If the answer is "no", the input formula φ has no satisfying assignment.
- For i ranging from 1 to n, let $\varphi_i(x_{i+1}, \ldots, x_n) \stackrel{\triangle}{=} \varphi(\sigma_1, \ldots, \sigma_{i-1}, 1, x_{i+1}, \ldots, x_n)$. Using the oracle, test whether $\varphi_i \in SAT$. If the answer is "yes", assign $\sigma_i \leftarrow 1$. Otherwise, assign $\sigma_i \leftarrow 0$. Clearly, the partial assignment $\sigma(1) = \sigma_1, \ldots, \sigma(i) = \sigma_i$ can still be completed into a satisfying assignment, and hence the algorithm terminates with a true assignment.

■

Consequently, one may deduce that if SAT is decidable in polynomial time, then there exists an efficient algorithm that solves the search problem for R_{SAT}. On the other hand, if SAT is not decidable in polynomial time (which is the more likely case), there is no efficient algorithm for solving the search problem. Therefore, research on the complexity of deciding SAT relates directly to the complexity of searching R_{SAT}.

In the next lecture we show that every $\mathcal{NP}$-complete language has a self-reducible relation. However, let us first discuss the problem of graph isomorphism, which can be easily shown to be in $\mathcal{NP}$, but is not known to be $\mathcal{NP}$-hard. We show that nevertheless, graph isomorphism has a self-reducible relation.

PROBLEM: Graph Isomorphism
INPUT: Two simple[1] graphs $G_1 = (V, E_1)$, $G_2 = (V, E_2)$. We may assume, without loss of generality, that none of the input graphs has any isolated vertices
TASK: Find an isomorphism between the graphs, i.e. a permutation $\varphi : V \to V$, such that $(u, v) \in E_1$ if and only if $(\varphi(u), \varphi(v)) \in E_2$.

The relation R_{GI} corresponding to the graph isomorphism problem is the set of all pairs $((G_1, G_2), \varphi)$ for which φ is an isomorphism between G_1 and G_2.

Proposition 1.4.2 *R_{GI} is self-reducible.*

Proof: To see that graph isomoprphism is self-reducible, consider an algorithm that uses a graph-isomorphism membership oracle along the lines of the algorithm for SAT. Again, the algorithm fixes the mapping $\varphi(\cdot)$ vertex by vertex.

[1]Such graphs have no self-loops and no parallel edges, and so each vertex has degree at most $|V| - 1$.

At each step, the algorithm fixes a single vertex u in G_1, and finds a vertex v such that the mapping $\varphi(u) = v$ can be completed into a graph isomorphism. To find such a vertex v, the algorithm tries all candidate mappings $\varphi(u) = v$ for all unmapped $v \in V$, using the oracle to tell whether the mapping can still be completed into a complete isomorphism. If there exists an isomorphism to begin with, such a mapping must exist, and hence the algorithm terminates with a complete isomorphism.

We now show how a partial assignment can be decided by the oracle. The trick here is that in order to check if u can be mapped to v, one can "mark" both vertices by a unique pattern, say by rooting a star of $|V|$ leaves at both u and v, resulting in new graphs G_1', G_2'. Next, query the oracle whether there is an isomorphism φ between G_1' and G_2'. Since the degrees of u and v are strictly larger than the degrees of other vertices in G_1' and G_2', an isomorphism φ' between G_1' and G_2' would exist if and only if there exists an isomorphism φ between G_1 and G_2 that maps u to v.

After the mapping of u is determined, proceed by incrementally marking vertices in V with stars of $2|V|$ leaves, $3|V|$ leaves, and so on, until the complete mapping is determined. ■

A point worth mentioning is that the definition of self-reducibility applies to relations and not to languages. A particular language $L \in \mathcal{NP}$ may be associated with more than one search problem, and the self-reducibility of a given relation R (or the lack thereof) does not immediately imply self-reducibility (or the lack thereof) for a different relation R' associated with the same language L.

It is believed that not every language in $\mathcal{NP}$ admits a self-reducible relation. Below we present an example of a language in $\mathcal{NP}$ for which the "natural" search problem is believed not to be self-reducible. Consider the language of composite numbers, i.e.,

$$L_{COMP} \stackrel{\triangle}{=} \{N \mid N = n_1 \cdot n_2 \quad n_1, n_2 > 1\}.$$

The language L_{COMP} is known to be decidable in polynomial time by a randomized algorithm. A natural relation R_{COMP} corresponding to L_{COMP} is the set of all pairs $(N, (n_1, n_2))$ such that $N = n_1 \cdot n_2$, where $n_1, n_2 > 1$. Clearly, the length of (n_1, n_2) is polynomial in the length of N, and since R_{COMP} can easily be decided in polynomial time, L_{COMP} is in $\mathcal{NP}$.

However, the search problem over R_{COMP} requires finding a pair (n_1, n_2) for which $N = n_1 \cdot n_2$. This problem is computationally equivalent to factoring, which is believed not to admit any (probabilistic) polynomial-time algorithm. Thus, it is very unlikely that R_{COMP} is (random) self-reducible.

Another language whose natural relation is believed not to be self-reducible is L_{QR}, the set of all quadratic residues. The language L_{QR} contains all pairs (N, x) in which x is a quadratic residue modulo N, namely, there exists a y for which $y^2 \equiv x \pmod{N}$. The natural search problem associated with L_{QR} is R_{QR}, the set of all pairs $((N, x), y)$ such that $y^2 \equiv x \pmod{N}$. It is well-known that the search problem over R_{QR} is equivalent to factoring under randomized reductions. Thus, under the assumption that factoring is "harder" than deciding L_{QR}, the natural relation R_{QR} is not (random) self-reducible.

Bibliographic Notes

For a historical account of the "P vs NP Question" see [2].

The discussion regarding Quadratic Resideousity is taken from [1]. This paper contains also further evidence to the existence of NP-relations which are not self-reducible.

1. M. Bellare and S. Goldwasser, "The Complexity of Decision versus Search", *SIAM Journal on Computing*, Vol. 23, pages 97–119, 1994.

2. Sipser, M., "The history and status of the P versus NP problem", *Proc. 24th ACM Symp. on Theory of Computing*, pages 603–618, 1992.

Lecture 2

NP-completeness and Self Reducibility

Notes taken by Nir Piterman and Dana Fisman

Summary: It will be proven that the relation R of any $\mathcal{NP}-$complete language is Self-reducible. This will be done using the $\mathcal{SAT}$ self reducibility proved previously and the fact that $\mathcal{SAT}$ is $\mathcal{NP}-$hard (under Levin reduction). Prior to that, a simpler proof of the existence of $\mathcal{NP}-$complete languages will be given.

2.1 Reductions

The notions of self reducibility and $\mathcal{NP}-$completeness require a definition of the term reduction. The idea behind reducing problem Π_1 to problem Π_2, is that if Π_2 is known to be easy, so is Π_1 or vice versa, if Π_1 is known to be hard so is Π_2

Definition 2.1 (*Cook Reduction*):
A Cook reduction *from problem Π_1 to problem Π_2 is a polynomial oracle machine that solves problem Π_1 on input x while getting oracle answers for problem Π_2.*

For example:
Let Π_1 and Π_2 be decision problems of languages L_1 and L_2 respectively and χ_L the characteristic function of L defined to be $\chi_L(x) = \begin{cases} 1 & x \in L \\ 0 & x \notin L \end{cases}$
Then Π_1 will be Cook reducible to Π_2 if exists an oracle machine that on input x asks query q, gets answer $\chi_{L_2}(q)$ and gives as output $\chi_{L_1}(x)$ (May ask multiple queries).

Definition 2.2 (*Karp Reduction*):
A Karp reduction (many to one reduction) *of language L_1 to language L_2 is a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ such that $x \in L_1$ if and only if $f(x) \in L_2$.*

Claim 2.1.1 *A Karp reduction is a special case of a Cook reduction.*

Proof: Given a Karp reduction $f(\cdot)$ from L_1 to L_2 and an input x to be decided whether x belongs to L_1, define the following oracle machine:

1. On input x compute the value $f(x)$.

2. Present $f(x)$ to the oracle of L_2.

3. The oracle's answer is the desired decision.

The machine runs polynomial time since Step 1 is polynomial as promised by Karp reduction and both Steps 2 and 3 require constant time.
Obviously M accepts x if and only if x is in L_1. ■

Hence a Karp reduction can be viewed as a Cook reduction.

Definition 2.3 (*Levin Reduction*):
A Levin reduction *from relation R_1 to relation R_2 is a triplet of polynomial time computable functions f, g and h such that:*

1. $x \in L(R_1) \iff f(x) \in L(R_2)$
2. $\forall (x,y) \in R_1 \ , \ (f(x), g(x,y)) \in R_2$
3. $\forall x,z \ \ (f(x),z) \in R_2 \Longrightarrow (x, h(x,z)) \in R_1$

Note: A Levin reduction from R_1 to R_2 implies a Karp reduction of the decision problem (using condition 1) and a Cook reduction of the search problem (using conditions 1 and 3).

Claim 2.1.2 *Karp reduction is transitive.*

Proof: Let $f_1 : \Sigma^* \longrightarrow \Sigma^*$ be a Karp reduction from L_a to L_b and $f_2 : \Sigma^* \longrightarrow \Sigma^*$ be a Karp reduction from L_b to L_c
The function $f_1 \circ f_2(\cdot)$ is a Karp reduction from L_a to L_c:

- $x \in L_a \iff f_1(x) \in L_b \iff f_2(f_1(x)) \in L_c$.
- f_1 and f_2 are polynomial time computable, so the composition of the functions is again polynomial time computable.

■

Claim 2.1.3 *Levin reduction is transitive.*

Proof: Let (f_1, g_1, h_1) be a Levin reduction from R_a to R_b and (f_2, g_2, h_2) be a Levin reduction from R_b to R_c. Define:

- $f_3(x) \stackrel{\triangle}{=} f_2(f_1(x))$
- $g_3(x,y) \stackrel{\triangle}{=} g_2(f_1(x), g_1(x,y))$
- $h_3(x,y) \stackrel{\triangle}{=} h_1(x, h_2(f_1(x), y))$

We show that the triplet (f_3, g_3, h_3) is a Levin reduction from R_a to R_c:

- $x \in L(R_a) \iff f_3(x) \in L(R_c)$
  Since:
  $x \in L(R_a) \iff f_1(x) \in L(R_b) \iff f_2(f_1(x)) \in L(R_c) \iff f_3(x) \in L(R_c)$

- $\forall (x,y) \in R_a$, $(f_3(x), g_3(x,y)) \in R_c$
  Since:
  $(x,y) \in R_a \Longrightarrow (f_1(x), g_1(x,y)) \in R_b \Longrightarrow (f_2(f_1(x)), g_2(f_1(x), g_1(x,y))) \in R_c \Longrightarrow$
  $(f_3(x), g_3(x,y)) \in R_c$

- $\forall x,z$ $(f_3(x), z) \in R_c \Longrightarrow (x, h_3(x,z)) \in R_a$
  Since:
  $(f_3(x), z) \in R_c \Longrightarrow (f_2(f_1(x)), z) \in R_c \Longrightarrow (f_1(x), h_2(f_1(x), z)) \in R_b \Longrightarrow$
  $(x, h_1(x, h_2(f_1(x), z))) \in R_a \Longrightarrow (x, h_3(x,z)) \in R_a$

■

Theorem 2.4 *If* Π_1 *Cook reduces to* Π_2 *and* $\Pi_2 \in \mathcal{P}$ *then* $\Pi_1 \in \mathcal{P}$.

Here class $\mathcal{P}$ denotes not only languages but also any problem that can be solved in polynomial time.
Proof: We shall build a deterministic polynomial time Turing machine that recognizes Π_1:
As Π_1 Cook reduces to Π_2, there exists a polynomial oracle machine M_1 that recognizes Π_1 while asking queries to an oracle of Π_2.
As $\Pi_2 \in \mathcal{P}$, there exists a deterministic polynomial time Turing machine M_2 that recognizes Π_2.
Now build a machine M, recognizer for Π_1 that works as following:

- On input x, emulate M_1 until it poses a query to the oracle.
- Present the query to the machine M_2 and return the answer to M_1.
- Proceed until no more queries are presented to the oracle.
- The output of M_1 is the required answer.

Since the oracle and M_2 give the same answers to the queries, correctness is obvious.
Considering the fact that M_1 is polynomial, the number of queries and the length of each query are polynomial in $|x|$. Hence the delay caused by introducing the machine M_2 is polynomial in $|x|$.
Therefore the total run time of M is polynomial. ■

2.2 All $\mathcal{NP}$-complete relations are Self-reducible

Definition 2.5 ($\mathcal{NP}-$complete *language*):
A language L *is* $\mathcal{NP}$-complete *if:*

1. $L \in \mathcal{NP}$
2. *For every language* L' *in* $\mathcal{NP}$, L' *Karp reduces to* L.

These languages are the hardest problems in $\mathcal{NP}$, in the sense that if we knew how to solve an $\mathcal{NP}-$complete problem efficiently we could have efficiently solved any problem in $\mathcal{NP}$. $\mathcal{NP}-$completeness can be defined in a broader sense by Cook reductions. There are not many known $\mathcal{NP}-$complete problems by Cook reductions that are not $\mathcal{NP}-$complete by Karp reductions.

Definition 2.6 *1. R is a $\mathcal{NP}$* relation *if $L(R) \in \mathcal{NP}$*

*2. A relation R is $\mathcal{NP}$-*hard **under Levin reduction** *if any $\mathcal{NP}$ relation R' is Levin reducible to R.*

Theorem 2.7 *For every $\mathcal{NP}$ relation R, if $L(R)$ is $\mathcal{NP}-$*complete *then R is Self-reducible.*

Proof: To prove the theorem we shall use two facts:

1. $\mathcal{SAT}$ is Self-reducible (was proved last lecture).

2. $R_{\mathcal{SAT}}$ is $\mathcal{NP}-$hard under Levin reduction (will be proven later).

Given an $\mathcal{NP}$ relation R of an $\mathcal{NP}-$complete language, a Levin reduction (f, g, h) from R to $R_{\mathcal{SAT}}$, a Karp reduction k from $\mathcal{SAT}$ to $L(R)$ and x, the following algorithm will find y such that $(x, y) \in R$ (provided that $x \in L(R)$).

The idea behind the proof is very similar to the self reducibility of $R_{\mathcal{SAT}}$:

1. Ask $L(R)$'s oracle whether $x \in L(R)$.

2. On answer $'no'$ declare: $x \notin L(R)$ and abort.

3. On answer $'yes'$ use the function f, that preserves the property of belonging to the language, to translate the input x for $L(R)$ into a satisfiable CNF formula $\varphi = f(x)$.

4. Compute $(\sigma_1, ..., \sigma_n)$ a satisfying assignment for φ as follows:

   (a) Given a partial assignment $\sigma_1, ..., \sigma_i$ such that $\varphi_i(x_{i+1}, ..., x_n) = \varphi(\sigma_1, ..., \sigma_i, x_{i+1}, x_{i+2}, ..., x_n) \in \mathcal{SAT}$, where $x_{i+1}, ..., x_n$ are variables and $\sigma_1, ..., \sigma_i$ are constants.

   (b) Assign $x_{i+1} = 1$ and compute $\varphi_i(1, x_{i+2}, ..., x_n) = \varphi(\sigma_1, ..., \sigma_i, 1, x_{i+2}, ..., x_n)$

   (c) Use the function k to translate the CNF formula $\varphi_i(1, x_{i+2}, ..., x_n)$ into an input to the language $L(R)$. Ask $L(R)$'s oracle wheather $k(\varphi_i(1, x_{i+2}, ..., x_n)) \in L(R)$.

   (d) On answer $'yes'$ assign $\sigma_{i+1} = 1$, otherwise assign $\sigma_{i+1} = 0$.

   (e) Iterate until $i = n - 1$.

5. Use the function h that translates a pair x and a satisfying assignment $\sigma_1, ..., \sigma_n$ to $\varphi = f(x)$ into a witness $y = h(x, (\sigma_1, ..., \sigma_n))$ such that $(x, y) \in R$.

Clearly $(x, y) \in R$. ■

Note: The above argument uses a Karp reduction of $\mathcal{SAT}$ to $L(R)$ (guaranteed by the NP-completeness of the latter). One may extend the argument to hold also for the case one is only given a Cook reduction of $\mathcal{SAT}$ to $L(R)$. Specifically in stage 4(c) instead of getting the answer to whether $\varphi_i(1, x_{i+2}, ..., x_n)$ is in $\mathcal{SAT}$ by quering on whether $k(\varphi_i)$ is in $L(R)$, we can get the answer by running the oracle machine given in the Cook reduction (which makes queries to $L(R)$).

2.3 $B_{ounded}H_{alting}$ is $\mathcal{NP}-$complete

In order to show that indeed exist problems in $\mathcal{NP}-$complete (i.e. the class $\mathcal{NP}-$complete is not empty) the language BH will be introduced and proved to be $\mathcal{NP}-$complete.

Definition 2.8 (Bounded Halting):

1. $BH \triangleq \left\{ (\langle M \rangle, x, 1^t) \;\middle|\; \begin{array}{l} \langle M \rangle \textit{ is the description of a non-deterministic machine} \\ \textit{that accepts input } x \textit{ within } t \textit{ steps.} \end{array} \right\}$
2. $BH \triangleq \left\{ (\langle M \rangle, x, 1^t) \;\middle|\; \begin{array}{l} \langle M \rangle \textit{ is the description of a deterministic machine and exists } y \textit{ whose} \\ \textit{length is polynomial in } |x| \textit{ such that } M \textit{ accepts } (x, y) \textit{ within } t \textit{ steps.} \end{array} \right\}$

The two definitions are equivalent if we consider the y wanted in the second as the sequence of non deterministic choices of the first. The computation is bounded by t hence so is y's length.

Definition 2.9 $R_{BH} \triangleq \left\{ ((\langle M \rangle, x, 1^t), y) \;\middle|\; \begin{array}{l} \langle M \rangle \textit{ is the description of a deterministic machine} \\ \textit{that accepts input } (x, y) \textit{ within } t \textit{ steps.} \end{array} \right\}$

Once again the length of the witness y is bounded by t, hence it is polynomial in the length of the input $(\langle M \rangle, x, 1^t)$.

Directly from $\mathcal{NP}$'s definition: $BH \in \mathcal{NP}$.

Claim 2.3.1 *Any language L in $\mathcal{NP}$, Karp reduces to BH*

Proof:
Given a language L in $\mathcal{NP}$, implies the following:

- A witness relation R_L exists and has a polynomial bound $b_L(\cdot)$ such that:
  $\forall\ (x, y) \in R_L, \quad |y| \leq b_L(|x|)$
- A recognizer machine M_L for R_L exists and its time is bounded by another polynomial $p_L(\cdot)$.

The reduction maps x to $f(x) \triangleq (\langle M_L \rangle, x, 1^{p_L(|x|+b_L(|x|))})$, which is an instance to BH by Version 2 of Definition 2.8 above.

Notice that the reduction is indeed polynomial since $\langle M_L \rangle$ is a constant string for the reduction from L to BH. All the reduction does is print this constant string, concatenate the input x to it and then concatenate a polynomial number of ones.
We will show now that $x \in L$ if and only if $f(x) \in BH$:

$x \in L \iff$
Exists a witness y whose length is bounded by $b_L(|x|)$ such that $(x, y) \in R_L \iff$
Exists a computation of M_L with $t \triangleq P_L(|x| + b_L(|x|))$ steps accepting $(x, y) \iff$
$(\langle M \rangle, x, 1^t) \in BH$
■

Note: The reduction can be easily transformed into Levin reduction of R_L to R_{BH} with the identity function supplying the two missing functions.
Thus $BH \in \mathcal{NP}-$complete.

Corollary 2.10 *There exist $\mathcal{NP}-$complete sets.*

2.4 $C_{ircuit}S_{atisfiability}$ is $\mathcal{NP}-$complete

Definition 2.11 (Circuit Satisfiability):

1. *A* **Circuit** *is a directed a-cyclic graph $G = (V, E)$ with vertices labeled:*
   output, $\bigwedge, \bigvee, \neg, x_1, ..., x_m, 0, 1$
   With the following restrictions:
   - *a vertex labeled by $\neg$ has in-degree 1.*
   - *a vertex labeled by x_i has in-degree 0 (i.e. is a source).*
   - *a vertex labeled by 0 (or 1) has in-degree 0.*
   - *there is a single sink (vertex of out-degree 0), it has in-degree 1 and is labeled 'output'.*
   - *in-degree of vertices labeled $\bigwedge, \bigvee$ can be restricted to 2.*

   Given an assignment $\sigma \in \{0,1\}^m$ to the variables $x_1, ..., x_m$, $C(\sigma)$ will denote the value of the circuit's output. The value is defined in the natural manner, by setting the value of each vertex according to the boolean operation it is labeled with. For example, if a vertex is labelled $\bigwedge$ and the vertices with a directed edge to it have values a and b, then the vertex has valu $a \bigwedge b$.
2. **Circuit Satisfiability**
   $CS \stackrel{\triangle}{=} \{C : C \text{ is a circuit and exists } \sigma, \text{ an input to circuit } C \text{ such that } C(\sigma) = 1\}$
3. $R_{CS} \stackrel{\triangle}{=} \{(C, \sigma) : C(\sigma) = 1\}$

The relation defined above is indeed an $\mathcal{NP}$ relation since:

1. σ contains assignment for all variables $x_1, x_2, ..., x_m$ appearing in C and hence its length is polynomial in $|C|$.
2. Given a couple (C, σ) evaluating one gate takes $O(1)$ (since in-degree is restricted to 2) and in view that the number of gates is at most $|C|$, total evaluation time is polynomial in $|C|$.

Hence $CS \in \mathcal{NP}$.

Claim 2.4.1 $C_{ircuit}S_{atisfiability}$ *is* $\mathcal{NP}-$complete

Proof: As mentioned previously $CS \in \mathcal{NP}$.
We will show a Karp reduction from BH to CS, and since Karp reductions are transitive and BH is $\mathcal{NP}-$complete, the proof will be completed. In this reduction we shall use the second definition of BH as given in Definition 2.8.
Thus we are given a triplet $(\langle M \rangle, x, 1^t)$. This triplet is in BH if exists a y such that the deterministic machine M accepts (x, y) within t steps. The reduction maps such a triplet into an instance of CS.
The idea is building a circuit that will simulate the run of M on (x, y), for the given x and a generic y (which will be given as an input to the circuit). If M does not accept (x, y) within the first t steps of the run, we are ensured that $(\langle M \rangle, x, 1^t)$ is not in BH. Hence it suffices to simulate only the first t steps of the run.
Each one of these first t configurations is completely described by the letters written in the first t tape cells, the head's location and the machine's state.

Hence the whole computation can be encoded in a matrix of size $t \times t$. The entry (i, j) of the matrix will consist of the contents of cell j at time i, an indicator whether the head is on this cell at time i and in case the head is indeed there the state of the machine is also encoded. So every matrix entry will hold the following information:

- $a_{i,j}$ the letter written in the cell
- $h_{i,j}$ an indicator to head's presence in the cell
- $q_{i,j}$ the machine's state in case the head indicator is 1 (0 otherwise)

The contents of matrix entry (i, j) is determined only by the three matrix entries $(i-1, j-1), (i-1, j)$ and $(i-1, j+1)$. If the head indicator of these three entries is off, entry (i, j) will be equal to entry $(i-1, j)$.
The following constructs a circuit that implements the idea of the matrix and this way emulates the run of machine M on input x. The circuit consists of t levels of t triplets $(a_{i,j}, h_{i,j}, q_{i,j})$ where $0 \leq i \leq t,\ 1 \leq j \leq t$. Level i of gates will encode the configuration of the machine at time i. The wiring will make sure that if level i represents the correct configuration, so will level $i+1$.
The (i, j)-th triplet, $(a_{i,j}, h_{i,j}, q_{i,j})$, in the circuit is a function of the three triplets $(i-1, j-1), (i-1, j)$ and $(i-1, j+1)$.
Every triplet consists of $O(log\ n)$ bits, where $n \stackrel{\triangle}{=} |(\langle M \rangle, x, 1^t)|$:

- Let G denote the size of M's alphabet. Representing one letter requires $log\ G$ many bits: $log\ G = O(log\ n)$ bits.
- The head indicator requires one bit.
- Let K denote the number of states of M. Representing a state requires $log\ K$ many bits: $log\ K = O(log\ n)$ bits.

Note that the machine's description is given as input. Hence the number of states and the size of the alphabet are smaller than input's size and can be represented in binary by $O(log\ n)$ many bits (Albeit doing the reduction directly from any $\mathcal{NP}$ language to CS, the machine M_L that accepts the language L wouldn't have been given as a parameter but rather as a constant, hence a state or an alphabet letter would have required a constant number of bits).
Every bit in the description of a triplet is a boolean function of the bits in the description of three other triplets, hence it is a boolean function of $O(log\ n)$ bits.

Claim 2.4.2 *Any boolean function on m variables can be computed by a circuit of size $m2^m$*

Proof: Every boolean function on m variables can be represented by a $(m+1) \times 2^m$ matrix. The first m columns will denote a certain input and the last column will denote the value of the function. The 2^m rows are required to describe all different inputs.
Now the circuit that will calculate the function is:
For line l in the matrix in which the function value is 1 ($f(l) = 1$), build the following circuit:
$C_l = (\bigwedge_{\text{input } y_i=1} y_i) \bigwedge (\bigwedge_{\text{input } y_i=0} \neg y_i)$
Now take the OR of all lines (value 1):
$C = \bigvee_{f(l)=1} C_l$
The circuit of each line is of size m and since there are at most 2^m lines of value 1, the size of the whole circuit is at most $m2^m$. ■

So far the circuit emulates a generic computation of M. Yet the computation we care about refers to one specific input. Similarly the initial state should be q_0 and the head should be located at time 0 in the first location. This will be done by setting all triplets $(0, j)$ as following:
Let $x = x_1x_2x_3...x_m$ and $n \stackrel{\triangle}{=} |(\langle M\rangle, x, 1^t)|$ the length of the input.

- $a_{0,j} = \begin{cases} x_j & 1 \le j \le m \quad \text{constants set by input } x \\ y_{j-m} & m < j \le t \quad \text{these are the inputs to the circuit} \end{cases}$
- $h_{0,j} = \begin{cases} 1 & j = 1 \\ 0 & j \ne 1 \end{cases}$
- $q_{0,j} = \begin{cases} q_0 & j = 1 \quad \text{where } q_0 \text{ is the initial state of } M \\ 0 & j \ne 1 \end{cases}$

The y elements are the variables of the circuit. The circuit belongs to CS if and only if there exists an assignment σ for y such that $C(\sigma) = 1$. Note that y, the input to the circuit plays the same role as the short witness y to the fact that $(\langle M\rangle, x, 1^t)$ is a member of BH. Note that (by padding y with zeros), we may assume without loss of generality that $|y| = t - |x|$.

So far (on input y) the circuit emulates a running of M on input (x, y), it is left to ensure that M accepts (x, y). The output of the circuit will be determined by checking whether at any time the machine entered the $'accept'$ state. This can be done by checking whether in any of the $t \times t$ triplets in the circuit the state is $'accept'$.

Since every triplet (i, j) consists of $O(log\ n)$ bits we have $O(log\ n)$ functions associated with each triplet. Every function can be computed by a circuit of size $O(n\ log\ n)$, so the circuit attached to triplet (i, j) is of size $O(n\ log^2\ n)$.
There are $t \times t$ such triplets so the size of the circuit is $O(n^3\ log^2\ n)$.
Checking for a triplet (i, j) whether $q_{i,j}$ is $'accept'$ requires a circuit of size $O(log\ n)$. This check is implemented for $t \times t$ triplets, hence the overall size of the output check is $O(n^2\ log\ n)$ gates.
The overall size of the circuit will be $O(n^3\ log^2\ n)$.

Since the input level of the circuit was set to represent the right configuration of machine M when operated on input (x, y) at time 0, and the circuit correctly emulates with its i^{th} level the configuration of the machine at time i, the value of the circuit on input y indicates whether or not M accepts (x, y) within t steps. Thus, the circuit is satisfiable if and only if there exists a y so that M accepts (x, y) within t steps, i.e. $(\langle M\rangle, x, 1^t)$ is in BH.

For a detailed description of the circuit and full proof of correction see Appendix.

The above description can be viewed as instructions for constructing the circuit. Assuming that building one gate takes constant time, constructing the circuit following these instructions will be linear to the size of the circuit. Hence, construction time is polynomial to the size of the input $(\langle M\rangle, x, 1^t)$.

■

Once again the missing functions for Levin reduction of R_{BH} to R_{CS} are the identity functions.

2.5 $R_{\mathcal{SAT}}$ is $\mathcal{NP}-$complete

Claim 2.5.1 *$R_{\mathcal{SAT}}$ is $\mathcal{NP}-$*hard *under Levin reduction.*

Proof: Since Levin reduction is transitive it suffices to show a reduction from R_{CS} to $R_{\mathcal{SAT}}$:
The reduction will map a circuit C to an CNF expression φ_{C} and an input y for the circuit to an assignment y' to the expression and vice versa.
We begin by describing how to construct the expression φ_{C} from C.
Given a circuit C we allocate a variable to every vertex of the graph. Now for every one of the vertices v build the CNF expression φ_v that will force the variables to comply to the gate's function:

1. For a $\neg$ vertex v with edge entering from vertex u:
   - Write $\varphi_v(v,u) = ((v \bigvee u) \bigwedge (\neg u \bigvee \neg v))$
   - It follows that $\varphi_v(v,u) = 1$ if and only if $v = \neg u$
2. For a $\bigvee$ vertex v with edges entering from vertices u, w:
   - Write $\varphi_v(v,u,w) = ((u \bigvee w \bigvee \neg v) \bigwedge (u \bigvee \neg w \bigvee v) \bigwedge (\neg u \bigvee w \bigvee v) \bigwedge (\neg u \bigvee \neg w \bigvee v))$
   - It follows that $\varphi_v(v,u,w) = 1$ if and only if $v = u \bigvee w$
3. For a $\bigwedge$ vertex v with edges entering from vertices u, w:
   - Similarly write $\varphi_v(v,u,w) = ((u \bigvee w \bigvee \neg v) \bigwedge (u \bigvee \neg w \bigvee \neg v) \bigwedge (\neg u \bigvee w \bigvee \neg v) \bigwedge (\neg u \bigvee \neg w \bigvee v))$
   - It follows that $\varphi_v(v,u,w) = 1$ if and only if $v = u \bigwedge w$
4. For the vertex marked *output* with edge entering from vertex u:
   Write $\varphi_{output}(u) = u$

We are ready now to define $\varphi_{\mathrm{C}} = \bigwedge_{v \in V} \varphi_v$, where V is the set of all vertices of in-degree at least one (i.e. the constant inputs and variable inputs to the circuit are not included).
The length of φ_{C} is linear to the size of the circuit. Once again the instructions give a way to build the expression in linear time to the circuit's size.

We next show that $C \in CS$ if and only if $\varphi_{\mathrm{C}} \in \mathcal{SAT}$. Actually, to show that the reduction is a Levin-reduction, we will show how to efficiently transform witnesses for one problem into witnesses for the other. That is, we describe how to construct the assignment y' to φ_{C} from an input y to the circuit C (and vice versa):
Let C be a circuit with m input vertices labeled $x_1, ..., x_m$ and d vertices labeled $\bigvee, \bigwedge$ and $\neg$ namely, $v_1, ..., v_d$. An assignment $y = y_1, ..., y_m$ to the circuit's input vertices will propagate into the circuit and set the value of all the vertices. Considering that the expression φ_{C} has a variable for every vertex of the circuit C, the assignment y' to the expression should consist of a value for every one of the circuit vertices. We will build $y' = y'_{x_1}, ..., y'_{x_m}, y'_{v_1}, y'_{v_2}, ..., y'_{v_d}$ as following:

- The variables of the expression that correspond to input vertices of the circuit will have the same assignment: $y'_{x_h} = y_h,\ 1 \leq h \leq m$.
- The assignment y'_{v_l} to every other expression variable v_l will have the value set to the corresponding vertex in the circuit, $1 \leq l \leq d$.

Similarly given an assignment to the expression, an assignment to the circuit can be built. This will be done by using only the values assigned to the variables corresponding to the input vertices of the circuit. It is easy to see that:
$C \in CS \iff$ exists y such that $C(y) = 1 \iff \varphi_{\mathrm{C}}(y') = 1 \iff \varphi_{\mathrm{C}} \in \mathcal{SAT}$ ■

Corollary 2.12 *$\mathcal{SAT}$ is $\mathcal{NP}-$*complete

Bibliographic Notes

The initiation of the theory NP-completeness is attributed to Cook [1], Levin [4] and Karp [3]: Cook has initiated this theory in the West by showing that SAT is NP-complete, and Karp demonstrated the wide scope of the phenumena (of NP-completeness) by demonstrating a wide variety of NP-complete problems (about 20 in number). Independently, in the East, Levin has shown that half a dozen different problems (including SAT) are NP-complete. The three types of reductions discussed in the lecture are indeed the corresponding reductions used in these papers. Whereas the Cook–Karp exposition is in terms of decision problems, Levin's exposition is in terms of search problems – which explains why he uses the stronger notion of reduction.

Currently thousands of problems, from an even wider range of sciences and technologies, are known to be NP-complete. A partial (out-dated) list is provided in Garey and Johnson's book [2].

Interestingly, almost all reductions used to establish NP-completeness are much more restricted than allowed in the definition (even according to Levin). In particular, they are all computable in logarithmic space (see next lectures for definition of space).

1. Cook, S.A., "The Complexity of Theorem Proving Procedures", *Proc. 3rd ACM Symp. on Theory of Computing*, pp. 151–158, 1971.

2. Garey, M.R., and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979.

3. Karp, R.M., "Reducibility among Combinatorial Problems", *Complexity of Computer Computations*, R.E. Miller and J.W. Thatcher (eds.), Plenum Press, pp. 85–103, 1972.

4. Levin, L.A., "Universal Search Problems", *Problemy Peredaci Informacii 9*, pp. 115–116, 1973. Translated in *problems of Information Transmission 9*, pp. 265–266.

Appendix: Details for the reduction of BH to CS

We present now the details of the reduction from BH to CS. The circuit that will emulate the run of machine M on input x can be constructed in the following way:
Let $(\langle M\rangle, x, t)$ be the input to be determined whether is in BH, where $x = x_1x_2...x_m$ and $n \stackrel{\triangle}{=} |(\langle M\rangle, x, t)|$ the length of the input.
We will use the fact that every gate of in-degree r can be replaced by r gates of in-degree 2. This can be done by building a balanced binary tree of depth $log\ r$. In the construction $'and'$,$'$ or' gates of varying in-degree will be used. When analyzing complexity, every such gate will be weighed as its in-degree.
The number of states of machine M is at most n, hence $log\ n$ bits can represent a state. Similarly the size of alphabet of machine M is at most n, and therfore $log\ n$ bits can represent a letter.

1. Input Level
   y is the witness to be entered at a later time (assume y is padded by zeros to complete length t as explained earlier).

   - $a_{0,j} = \begin{cases} x_j & 1 \le j \le m \quad \text{constants set by input } x \\ y_{j-m} & m < j \le t \quad \text{these are the inputs to the circuit} \end{cases}$
   - $h_{0,j} = \begin{cases} 1 & j = 1 \\ 0 & j \ne 1 \end{cases}$
   - $q_{0,j} = \begin{cases} q_0 & j = 1 \quad \text{where } q_0 \text{ is the initial state of } M \\ 0 & j \ne 1 \end{cases}$

   As said before this represents the configuration at time 0 of the run of M on (x, y).
   This stage sets $O(n\ log\ n)$ wires.

2. For $0 < i < t$, $h_{i+1,j}$ will be wired as shown in figure 1:

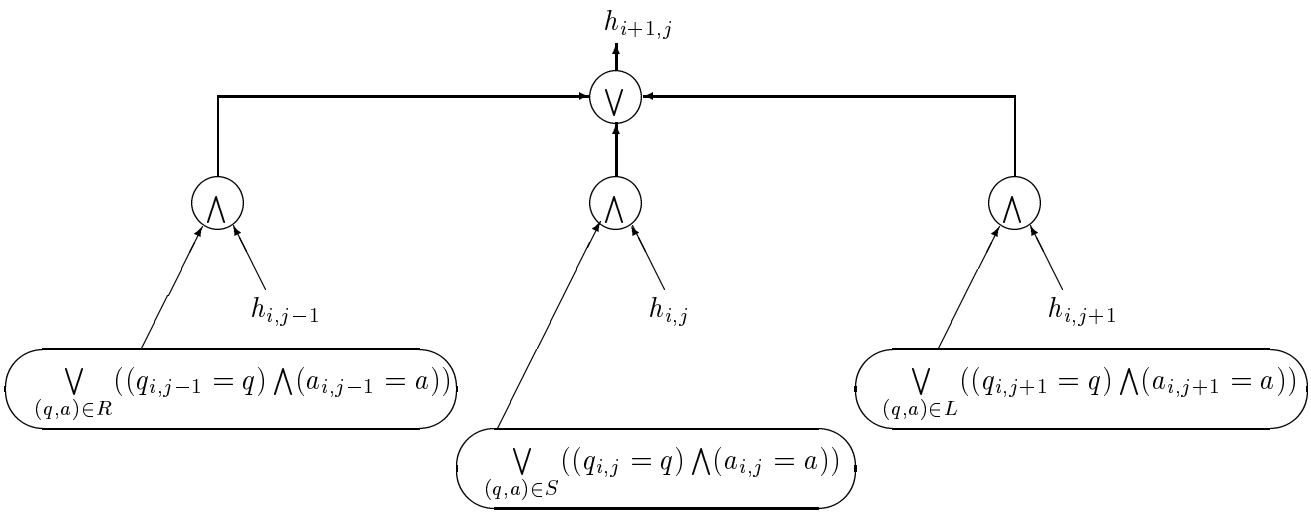


figure 1

The definition of groups R, S, L is:
$R \stackrel{\triangle}{=} \{(q,a) : q \in K \bigwedge a \in \{0,1\} \bigwedge \delta(q,a) = (*,*,R)\}$
$S \stackrel{\triangle}{=} \{(q,a) : q \in K \bigwedge a \in \{0,1\} \bigwedge \delta(q,a) = (*,*,S)\}$
$L \stackrel{\triangle}{=} \{(q,a) : q \in K \bigwedge a \in \{0,1\} \bigwedge \delta(q,a) = (*,*,L)\}$
The equations are easily wired using an $'and'$ gate for every equation.
The size of this component:
The last item on every entry in the relation δ is either R, L or S. For every one of these entries there is one comparison above. Since δ is bounded by n there are at most n such comparisons. A comparison of the state requires $O(log\ n)$ gates. Similarly a comparison of the letter requires $O(log\ n)$ gates. Hence the total number of gates in figure 1 is $O(n\ log\ n)$

3. For $0 < i < t$, $q_{i+1,j}$ will be wired as shown in figure 2:

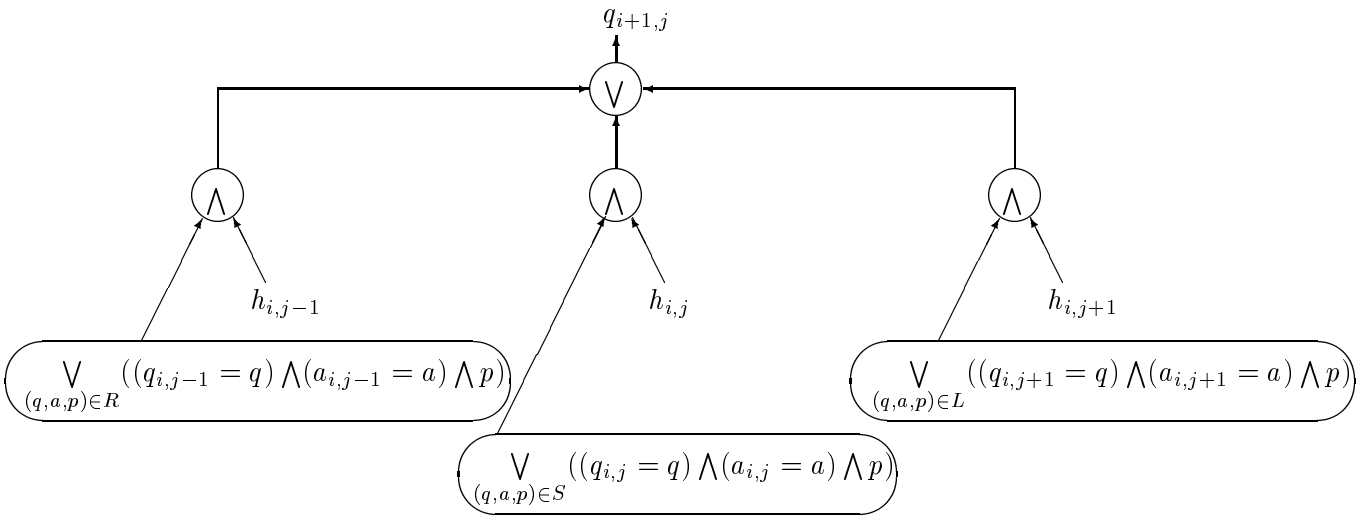


figure 2

The definition of groups R, S, L in:
$R \stackrel{\triangle}{=} \{(q,a,p) : q,p \in K \bigwedge a \in \{0,1\} \bigwedge \delta(q,a) = (p,*,R)\}$
$S \stackrel{\triangle}{=} \{(q,a,p) : q,p \in K \bigwedge a \in \{0,1\} \bigwedge \delta(q,a) = (p,*,S)\}$
$L \stackrel{\triangle}{=} \{(q,a,p) : q,p \in K \bigwedge a \in \{0,1\} \bigwedge \delta(q,a) = (p,*,L)\}$
Once again every comparison requires $O(log\ n)$ gates. Every state is represented by $log(n)$ bits so the figure has to be multiplied for every bit.
Overall complexity of the component in figure 2 is $O(n\ log^2\ n)$.

4. For $0 < i < t$, $a_{i+1,j}$ will be wired as shown in figure 3:

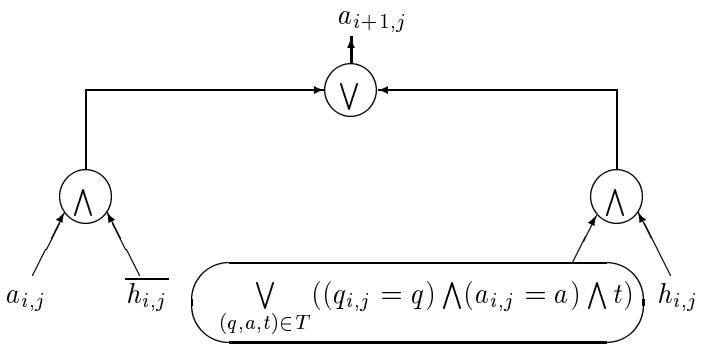


figure 3

The definition of T is:
$T \stackrel{\triangle}{=} \{(q,a,t) : q \in K \bigwedge a,t \in \{0,1\} \bigwedge \delta(q,a) = (*,t,*)\}$
Once again all entries of the relation δ have to be checked, hence there are $O(n)$ comparisons of size $O(log\ n)$.
Since the letter is represented by $O(log\ n)$ bits, the overall complexity of the component in figure 3 is $O(n\ log^2\ n)$.

5. Finally the output gate of the circuit will be a check whether at any level of the circuit the state was *accept*. This will be done by comparing $q_{i,j},\ 1 \le j \le t,\ 0 \le i \le t$ to $'accept'$. There are $t \times t$ such comparisons, each of them takes $O(log\ n)$ gates. Taking an OR on all these comparisons costs $O(n^2\ log\ n)$ gates.

For every cell in the $t \times t$ matrix we used $O(n^3)$ gates. The whole circuit can be built with $O(n^5)$ gates. With this description, building the circuit is linear to circuit's size. Hence, this can be done in polynomial time.

Correctness: We will show now that $(\langle M\rangle, x, 1^t) \in BH$ if and only if $C_{(\langle M\rangle,x,t)} \in CS$

Claim 2.5.2 *Gates at level i of the circuit represent the exact configuration of M at time i on input (x,y).*

Proof: By induction on time i.

- $i = 0$, stage 1 of the construction ensures correctness.
- Assume C's gates on level i correctly represent M's configuration at time i and prove for $i+1$:
  Set j as the position of the head at time i $(h_{i,j} = 1)$.
  - The letter contents of all cells $(i+1,k), k \neq j$ does not change. Same happens in the circuit since $(a_{i,k} \bigwedge(\neg h_{i,k})) = a_{i,k}$.
  - Likewise the head can not reach cells $(i+1,k)$ where $k < (j-1)$ or $k > (j+1)$. Respectively $h_{i,k} = 0$ since $h_{i,k-1} = h_{i,k} = h_{i,k+1} = 0$.

– The same argument shows that state bits for all gates of similar $k's$ will be reset to zero.

Let $\delta(q_{i,j}, a_{i,j}) = (q, a, m)$
We shall look into what happens when machine's head stays in place, i.e. $m = S$. The other two posibilities for movement of the head are similar.

– Cell (i, j) on the tape will change into a. Since $h_{i,j} = 1$ and correspondingly $(q_{i,j} = q_{i,j} \bigwedge a_{i,j} = a_{i,j} \bigwedge a)$ will return a
– The head stays in place and respectively:
  1. $h_{i+1,j-1} = 0$ since $h_{i,j} = 1$ but $\delta(q_{i,j}, a_{i,j})$ is not $(*, *, L)$.
  2. $h_{i+1,j} = 1$ since $h_{i,j} = 1$ and one $\delta(q_{i,j}, a_{i,j}) = (*, *, S)$ returns 1.
  3. $h_{i+1,j+1} = 0$ since $h_{i,j} = 1$ but $\delta(q_{i,j}, a_{i,j})$ is not $(*, *, R)$.
– The machine's next state is q, and respectively:
  1. Similarly $q_{i+1,j-1}$ and $q_{i+1,j+1}$ will be reset to zero.
  2. $q_{i+1,j}$ will change into q since $h_{i,j} = 1$ and $(q_{i,j} = q_{i,j} \bigwedge a_{i,j} = a_{i,j} \bigwedge q)$ will return q.

So at any time $0 \leq i \leq t$, gate level i correctly represents M's configuration at time i.
■

Lecture 3

More on NP and some on DTIME

Notes taken by Michael Elkin and Ekaterina Sedletsky

Summary: In this lecture we discuss some properties of the complexity classes P, NP and NPC (theorems of Ladner and Levin). We also define new complexity classes (DTime_i), and consider some relations between them.

3.1 Non-complete languages in NP

In this lecture we consider several items, that describe more closely the picture of the complexity world. We already know that $P \subseteq NP$ and we conjecture that it's a strict inequality, although we can't prove it. Another important class that we have considered is $NP - complete$ (NPC) problems, and as we have already seen, if there is a gap between P and NP then the class of NPC problems is contained in this gap ($NPC \subseteq NP \backslash P$).

The following theorem of Ladner tells us additional information about this gap $NP\backslash P$, namely that NPC is strictly contained in $NP\backslash P$.

Formally,

Theorem 3.1 *If $P \neq NP$ then there exists a language $L \in (NP\backslash P)\backslash NPC$.*

That is, NP (or say SAT) is not (Karp) reducible to L. Actually, one can show that SAT is not even Cook-reducible to L.

Oded's Note: *Following is a proof sketch.*

We start with any $B \in NP\backslash P$, and modify it to $B' = B \cap S$, where $S \in P$, so that B' is neither NP-complete nor in P. The fact that S is in P implies that B' is in NP. The "seiving" set S will be constructed to to foil both all possible polynomial-time algorithms for B' and all possible reductions. At the extreme, setting $S = \{0,1\}^*$, foil all algorithms since in this case $B' = B \notin P$. On the other hand, setting $S = \emptyset$, foils all possible reductions since in this case $B' = \emptyset$ and so (under $P \neq NP$) cannot be NP-complete (as reducing to it gives nothing). Note that the above argument extends to the case S (resp., $\overline{S}$) is a finite set.

The "seiving" set S is constructed iteratively so that in odd iterations is fails machines from a standard enumeration of polynomial-time machines (so that in iteration $2i-1$ we fail the i^{th} machine). (Here we don't need to emulate these machines in polynomial-time

in length of their inputs.) In even iterations we fail oracle-machines from a standard enumeration of such machines (which correspond to Cook-reductions of B to B') so that in iteration $2i$ we fail the i^{th} oracle-machine.

The iteration number is determined by a deciding algorithm for S which is operates as follows. For simplicity, the algorithm puts z in S iff it puts $1^{|z|}$ in S. The decision whether to put 1^n in S is taken as follows. Starting with the first iteration, and using a time-out mechanism with a fixed polynomial bound $b(n)$ (e.g., $b(n) = n^2$ or $b(n) = \log n$ will do), we try to find a input $z \in \{0,1\}^*$ so that the first polynomial-time algorithm, A_1, fails on z (i.e., $A_1(z) \neq \chi_B(z)$). In order to decide $\chi_B(z)$ we run the obvious exponential-time algorithm, but z is expected to be much shorter than n (or else we halt by time-out before). We scan all possible z's in lexicographic order until reaching time-out. Once we reach time-out while not finding such bad z, we let $1^n \in S$. Eventually, for a sufficiently large n we will find a bad z within the time-out. In such a case we let $1^n \notin S$ and continue to the second iteration, where we consider the first polynomial-time oracle-machine, M_1. Now we try to find an input z for B on which M_1 with oracle S. Note that we know the value $n' < n$ so that $1^{n'}$ was the first string not put in S. So currently, S is thought of as containing only strings of length smaller than n'. We emulate M_1 while answering its queries to $B' = B \cap S$ accordingly, using the exponential-time algorithm for deciding B (and our knowledge of the portion of S determined so far). We also use the exponential-time algorithm for B to check if the recuction of z to B' is correct for each z. Once we reach time-out while not finding such bad z, we let $1^n \notin S$. Again, eventually, for a sufficiently large n we will find a bad z within the time-out. In such a case we let $1^n \in S$ and continue to the third iteration, where we consider the second polynomial-time algorithm, and so on.

Some implementation details are provided below. Specifically, the algorithm T below computes the number of iterations completed wrt input $x \in \{0,1\}^n$.

Proof: Let $B \in NP \backslash P$. Let $A_0, A_1, ...$ be the enumeration of all polynomial time bounded Turing machines, that solve decision problems and $M_0, M_1, ...$ be the enumeration of polynomial time bounded oracle Turing machines. Let $L(A_i)$ denote the set recognized by A_i and for every set S let $L(M_i^S)$ to be the set recognized by machine M_i with oracle S.

We construct a polynomial time bounded Turing machine T with range $\{1\}^*$ in such a way that, for $B' = \{x \in B : |T(x)| \text{ is even}\}$, both $B' \notin P$ and $B \not\propto_C B'$ (i.e., B is not Cook reducible to B'). It follows that $B' \notin P \cup NPC$.

We show that for any such T, $B' \propto_K B$ (B' is Karp reducible to B) and $B' \in NP$ follows, since diciding B' can be done by deciding B and $B \in NP$.The Karp-reduction (of B' to B), denoted f, is defined as follows. Let $x_0 \notin B$. (We assume that $B \neq \Sigma^*$, because otherwise $B \in P$). The function f will compute $|T(x)|$ and if $|T(x)|$ is even it will return x and otherwise it will return x_0. Now if $x \in B'$, then $x \in B$ and $|T(x)|$ is even, hence $f(x) = x \in B$. Otherwise $x \notin B'$ and then there are two possibilities.

1. If $x \notin B$, then for even $|T(x)|$ holds $f(x) = x$, and for odd $|T(x)|$ holds $f(x) = x_0$, and so for any x holds $f(x) \notin B$.

2. If $x \in B$ and $|T(x)|$ is odd, then $f(x) = x_0 \notin B$.

(Recall that $x \notin B'$ rules out "$x \in B$ and even $|T(x)|$")

To complete the construction we have to build a Turing machine T such that

(1) $B' \stackrel{\text{def}}{=} \{x \in B : |T(x)| \text{ is even}\} \neq L(A_i)$, for any $i = 0, 1, 2, ...$ (and so $B' \notin P$)

(2) $L(M_i^{B'}) \neq B$, for any $i = 0, 1, 2, ...$ (and so $B \not\propto_C B'$).

A machine T that satisfies these conditions may be constructed in the following way:

On input λ (empty string), T prints λ. On an input x of length n where $x \neq 0^n$ (unary), T prints $T(0^n)$. It remains to say what T does on inputs of the form 0^n where $n \geq 1$.

On input 0^n where $n \geq 1$, T does the following:

1. For n moves, try to reconstruct the sequence $T(\lambda), T(0), T(0^2), ...$ Let $T(0^m)$ be the last number of this sequence that is computed.

2. We consider two cases depending on the parity of $|T(0^m)|$. We associate B with an exponential-time algorithm decoding B (by scanning all possible NP-witnesses).

   Case(i): $|T(0^m)|$ is even. Let $i = |T(0^m)|/2$. For n moves, try to find a z such that $B'(z) \neq A_i(z)$. This is done by successively simulating B and T (to see what B' is) and A_i, on inputs $\lambda, 0, 1, 00, 01, ...$ If no such z is found, print 1^{2*i}; otherwise, print 1^{2*i+1}.

   Case(ii): $|T(0^m)|$ is odd. Let $i = (|T(0^m)| - 1)/2$. For n moves, try to find a z such that $B(z) \neq M_i^{B'}(z)$. This is done by simulating an algorithm B and the procedure M_i successively on inputs $\lambda, 0, 1, 00, 01, ...$ In simulating M_i on some input, whenever M_i makes a query of length at most m, we answer it according to B' determined by B and the values of T computed in Step (1). In case the query has lengths exceeding m, we behave as if we have already completed n steps. (The moves in this side calculation are counted among the n steps allowed.) If no such z is found, print 1^{2*i+1}; otherwise, print 1^{2*i+2}.

Such a machine can be programmed in a Turing machine that runs in polynomial time. For this specific machine T we obtain that $B' = \{x \in B : |T(x)| \text{ is even}\}$, satisfies: $B' \in NP \backslash P$ and $B \not\propto_C B' \Longrightarrow B' \notin NPC \Longrightarrow B' \in (NP \backslash P) \backslash NPC$. ∎

The set B' constructed in the above proof is certainly not a natural one (even in case B is). We mention that there are some natural problems conjectured to be in $(NP \backslash P) \backslash NPC$: For example, Graph Isomorphism (the problem of whether two graphs are isomorphic).

3.2 Optimal algorithms for NP

The following theorem, due to Levin, states an interesting property of the class NP. Informally, Levin's Theorem tells us that exists optimal algorithm for any NP search problem.

Theorem 3.2 *For any NP-relation R there exist a polynomial $P_R(\cdot)$ and an algorithm $A_R(\cdot)$ which finds solutions whenever they exist, so that for every algorithm A which finds solutions and for any input x*

$$time_{A_R}(x) \leq O(time_A(x) + P_R(|x|)), \tag{3.1}$$

where $time_A(x)$ is the running time of the algorithm A on input x.

This means that for every algorithm A exists a constant c such that for any sufficiently long x

$$time_{A_R}(x) \leq c * (time_A(x) + P_R(|x|)).$$

This c is not a universal constant, but an algorithm-specific one (i.e., depends on A). The algorithm A_R is optimal in the following sense: for any other algorithm A there exists a constant c such that for any sufficiently long x

$$time_A(x) \geq \frac{1}{c} * time_{A_R}(x) - Q_R(|x|),$$

where $Q_R(|x|) = \frac{1}{c} * P_R(|x|)$.

The algorithms we are talking about are not TM's. For proving the theorem, we should define exactly the computational model we are working with. Either it will be a one-tape machine or two-tape one and etc. Depending on the exact model the constant c may be replaced by some low-n function like $logn$. A constant may be achieved only in more powerful/flexible models of computation that are not discussed here.

We observe also that although the proof of Levin's Theorem is a constructive one, it's completely impractical, since as we'll see it incorporates a huge constant in its running time. On the other hand, it illustrates an important asymptotic property of the NP class.

Proof: The basic idea of the proof is to have an enumeration of all possible algorithms. This set is countable, since the set of all possible deterministic TM's is countable. Using this enumeration we would like to run all the algorithms in parallel. It's, of course, impossible since we can't run a countable set of TM's in parallel, but the solution to the problem is to run them in different rates.

There are several possible implementations of this idea. One of the possibilities is as following. Let us divide the execution of the simulating machine into rounds. The simulating machine runs machine i at round r if and only if $r \equiv 0 \pmod{i^2}$. That is, we let i'th machine to make t steps during $i^2 * t$ rounds of the simulating machine. Also the number of steps made in these $r = i^2 * t$ rounds is $\sum_{j \geq 1} |\frac{r}{j^2}| < r$.

Such a construction would fail if some of these machines would provide wrong answers. To solve this difficulty we "force" the machines to verify the validity of their outputs. So, without loss of generality, we assume that each machine is augmented by a verifying component that checks the validity of machine's output. Since the problem is in NP, verifying the output takes polynomial amount of time. When we estimate the running time of the algorithm A_R, we take into account this polynomial amount of time and it is the reason that P_R arises in Eq. (3.1). So, without loss of generality, the outputs of the algorithms are correct.

Another difficulty, is that some of these machines could not have sufficient amount of time to halt. On the other hand, since each of the machines solves the problem, it is sufficient for us that one of them will halt.

Levin's optimal algorithm A_R is this construction running interchangeably all these machines.

We'll claim the following property of the construction.

Claim 3.2.1 *Consider A that solves the problem. Let i be the position of A in the enumeration of all the machines. Let $time_A(\cdot\ )$ be the running time of A. It is run in A_R in $\frac{1}{f(i)}$ rate. Then A_R runs at most $f(i) * time_A(\cdot\ )$.*

We took $f(i) = (i+1)^2$, but that is not really important: we observe that i is a constant (and thus, so is $f(i)$). Of course, it's a huge constant, because each machine needs millions of bits to be encoded (even the simplest deterministic TM that does nothing needs several hundreds of bits to be encoded) and the index of machine M in the enumeration (i.e. i) is $i \approx 2^{|M|}$, where $|M|$ is the number of bits needed to encode machine M and then $f(i)$ will be $(i+1)^2 = (2^{|M|}+1)^2$. (This constant makes the algorithm completely impractical.) ■

3.3 General Time complexity classes

The class P was introduced to capture our notion of efficient computation. This class has some "robustness" properties which make it convenient for investigation.

1. P is not model-dependent: Remain the same if we consider one tape TM or two tape TM. This remains valid for any "reasonable" and "general" enough model of computation.
2. P is robust under "Reasonable" changes to an algorithm: Closed classes under "reasonable" changes of the algorithm, like flipping the output and things like that. This holds for P, but probably not for NP and NPC. The same applies also for (3) and (4).
3. Closed under serial composition: Concatenation of two (efficient) algorithms from a class will produce another (efficient) algorithm from the same class.
4. Closed under subroutine operation: Using one algorithm from a class as a subroutine of another algorithm from the same class provides an algorithm from the class (the class is set of problems and when we are talking about an algorithm from the class we mean an algorithm for solving a problem from the class and the existence of this algorithm is evidence that the problem indeed belongs to this class).

None of these nice properties holds for classes that we will now define.

3.3.1 The DTime classes

Oded's Note: *DTime denotes the class of languages which are decideable within a specific time-bound. Since this class specifies one time-bound rather than a family of such bounds (like polinomial-time), we need to be soecific with respect to the model of computation.*

Definition 3.3 *$DTime_i(t(\cdot))$ is the class of languages decidable by a deterministic i-tape Turing Machine within $t(\cdot)$ steps. That is, $L \in DTime_i(t(\cdot))$ if there exists a deterministic i-tape Turing Machine M which accepts L so that for any $x \in \{0,1\}^*$, on input x machine M makes at most $t(|x|)$ steps.*

Usually, we consider $i = 1$ or $i = 2$ talking about one- and two-tape TM's respectively. When we'll consider space complexity we'll find it very natural to deal with 3-tape TM. If there is no index in DTime, then the default is $i = 1$.

Using this new notation we present the following theorem:

Theorem 3.4 *For every function $t(\cdot)$ that is at least linear*

$$DTime_2(t(\cdot)) \subseteq DTime_1(t(\cdot)^2)$$

The theorem is important in the sense that it enables us sometimes to skip the index, since with respect to polynomial-time computations both models (one-tape and two-tape) coincide. The proof is by simulating the two-tape TM on a one-tape one.

Proof: Consider a language $L \in DTime_2(t(\cdot))$. Therefore, there exists a two-tape TM M_1 which accepts L in $O(t(\cdot))$. We can imagine that the tape of a one-tape TM as divided into 4 tracks. We

can construct M_2, a one-tape TM with 4 tracks, 2 tracks for each of M_1's tapes. One track records the contents of the corresponding tape of M_1 and the other is blank, except for a marker in the cell that holds the symbol scanned by the corresponding head of M_1. The finite control of M_2 stores the state of M_1, along with a count of the number of head markers to the right of M_2's tape head.

Each move of M_1 is simulated by a sweep from left to right and then from right to left by the tape head of M_2, which takes $O(t(\cdot))$ time. Initially, M_2's head is at the leftmost cell containing a head marker. To simulate a move of M_1, M_2 sweeps right, visiting each of the cells with head markers and recording the symbol scanned by both heads of M_1. When M_2 crosses a head marker, it must update the count of head markers to the right. When no more head markers are to the right, M_2 has seen the symbols scanned by both of M_1's heads, so M_2 has enough information to determine the move of M_1. Now M_2 makes a pass left, until it reaches the leftmost head marker. The count of markers to the right enables M_2 to tell when it has gone far enough. As M_2 passes each head marker on the leftward pass, it updates the tape symbol of M_1 "scanned" by that head marker, and it moves the head marker one symbol left or right to simulate the move of M_1. Finally, M_2 changes the state of M_1 recorded in M_2's control to complete the simulation of one move of M_1. If the new state of M_1 is accepting, then M_2 accepts.

Finding the mark costs $O(t(\cdot))$, and as there are not more than $t(\cdot)$ moves, totally the execution costs at most $O(t^2(\cdot))$. ■

The next theorem is less important and brought from elegancy considerations, and says that in general one cannot do a better simulation than one in Theorem 3.
We recall the definition of "O", "Ω" and "o" notations:

- $f(n) = O(g(n))$ means that exists c such that for any sufficiently large n $f(n) \leq c * g(n)$.
- $f(n) = \Omega(g(n))$ means that exists $c > 0$ such that for any sufficiently large n $f(n) \geq c * g(n)$.
- $f(n) = o(g(n))$ means that for any c exists N such that for all $n > N$ it holds $f(n) < c * g(n)$.

Theorem 3.5 *$DTime_2(O(n))$ is not contained in $DTime_1(o(n^2))$.*

We note that it's much harder to prove that some things are "impossible" or "can not be done", than the opposite, because for the latter, constructive proofs can be used.

There are several possible ways to prove the theorem. The following one uses the notion of communication complexity.

Proof: Define language $L = \{xx : x \in \{0,1\}^*\}$. This language L is clearly in $DTime_2(O(n))$. We will show that $L \notin DTime_1(o(n^2))$ by "reduction" to a communication complexity problem.

Introduce, for the sake of proof, computational complexity model: two parties A and B have two strings, A has $\alpha \in \{0,1\}^n$ and B has $\beta \in \{0,1\}^n$, respectively. Their goal is to calculate $f(\alpha, \beta)$, where f is a function from $\{0,1\}^n * \{0,1\}^n$ to $\{0,1\}$. At the end of computation both parties should know $f(\alpha, \beta)$.

Let us also introduce a notation $R_0(f)$ to be the minimum expected cost of a randomized protocol, that computes f with zero error.

In our case it is sufficient to consider Equality (EQ) function defined by

$$EQ(\alpha, \beta) \stackrel{\text{def}}{=} 1 \text{ if } \alpha = \beta \text{ and } 0 \text{ otherwise.}$$

We state without proof a lower bound on the randomized zero-error communication complexity of EQ.

$$R_0(EQ) = \Omega(n), \tag{3.2}$$

The lower bound can be proven by invoking the lower bound on the "nondeterministic communication complexity" of EQ, and from the observation that nondeterminism is generally stronger than (zero-error!) randomization: Intuitively, if a randomized algorithm reaches the solution with some non-zero probability, then there is a sequence of values of flipped coins that causes the randomized algorithm to reach the solution. The nondeterministic version of the same algorithm could just guess correctly all these coins' values and to reach the solution as well.

We now get back to the proof. Suppose for contradiction, that there exists a one-tape Turing machine M which decides L in $o(n^2)$ time. Then we will build a zero-error randomized protocol Π that solves EQ in expected complexity $o(n)$, contradicting Eq. (3.2).

This protocol Π, on input α and β each of length n simulates the Turing machine on input $\alpha 0^n \beta 0^n$ in the following way. They output 1 or 0 depending on whether the machine accepts or rejects this input. They first choose together, uniformly at random, a location at the first 0-region of the tape. The party A simulates the machine whenever the head is to the left of this location, and the party B whenever the head is to the right of this location. Each time the head crosses this location only the state of the finite control ($O(1)$ bits) need to be sent. If the total running time of the machine is $o(n^2)$, then the expected number of times it crosses this location (which has been chosen at random among n different possibilities) is at most $o(n)$, contradicting Eq. (3.2).

Therefore, we have proved that a one-tape Turing machine which decides L runs $\Omega(n^2)$ time. ■

An alternative way of proving the theorem, a direct proof, based on the notion of a crossing sequence, is given in the Appendix.

3.3.2 Time-constructibility and two theorems

Definition 3.6 *A function $f : N \longrightarrow N$ is time-constructible if there exists an algorithm A st. on input 1^n, A runs at most $f(n)$ steps and outputs $f(n)$ (in binary).*

One motivation to the definition of time-constructible function is the following: if the machine's running time is this specific function of input length, then we can calculate the running time within the time required to perform the whole calculation. This notion is important for simulating results, when we want to "efficiently" run all machines which have time bound $t(\cdot)$. We cannot enumerate these machines. Instead we enumerate all machines and run each with time bound $t(\cdot)$. Thus, we need to be able to compute $t(\cdot)$ within time $t(\cdot)$. Otherwise, just computing $t(\cdot)$ will take too much time.

For example, n^2, 2^n, n^n are all time-constructible functions.

Time Hierarchy: A special case of the Time Hierarchy Theorem asserts that

for every constant $c \in N$, for any $i \in \{1, 2\}$

$$DTime_i(n^c) \subset DTime_i(n^{c+1})$$

(where $A \subset B$ denotes that A is strictly contained in B)

That is, in this case there is no "complexity gaps" and the set of problems that can be solves grows whewn allowing more time: There are computational tasks that can be done in $O(n^{c+1})$, but can not be done in $O(n^c)$. The function n^{c+1} (above) can be replaced even by $n^{c+\frac{1}{2}}$ etc. The general case of the Time Hierarchy Theorem is

Theorem 3.7 (Time Hierarchy): *For every time-constructible functions $t_1, t_2 : N \longrightarrow N$ such that*

$$\lim_{n\to\infty} \frac{t_1(n) * \log t_2(n)}{t_2(n)} = 0$$

then

$$DTime_i(t_1(n)) \subset DTime_i(t_2(n)).$$

The proof for an analogous space hierarchy is easier, and therefore we'll present it first, but in following lectures.

Linear Speed-up: The following *Linear Speed-up Theorem* allows us to discard constant factors in running-time. Intuitively, there is no point in holding such an accurate account when one does not specify other components of the algorithm (like the size of its finite control and work-tape alphabet).

Theorem 3.8 (Linear Speed-up): *For every function $t : N \longrightarrow N$ and for every i*

$$DTime_i(t(n)) \subseteq DTime_i(\frac{t(n)}{2} + O(n)).$$

The proof idea is as following: let Γ be the original work alphabet. We reduce the time complexity by a constant factor by working with larger alphabet $\Gamma_k = \underbrace{\Gamma * \Gamma * ... * \Gamma}_{k \text{ times}}$, which enables us to process adjacent symbols simultaneously. Then we construct a new machine with alphabet Γ_k. Using this alphabet, any k adjacent cells of the original tape are replaced by one cell of the new tape.

So the new input will be processed almost k times faster, but dealing with the input will produce $O(n)$ overhead.

Let M_1 be an i-tape $t(n)$ time-bounded Turing machine. Let L be a language accepted by M_1. Then L is accepted by a i-tape $(\frac{t(n)}{2} + O(n))$ time-bounded TM M_2.

Proof: A Turing machine M_2 can be constructed to simulate M_1 in the following manner. First M_2 copies the input onto a storage tape, encoding 16 symbols into one. From this point on, M_2 uses this storage tape as the input tape and uses the old input tape as a storage tape. M_2 will encode the contents of M_1's storage tape by combining 16 symbols into one. During the course of the simulation, M_2 simulates a large number of moves of M_1 in one basic step consisting of eight moves of M_2. Call the cells currently scanned by each of M_2's heads the home cells. The finite control of M_2 records, for each tape, which of the 16 symbols of M_1 represented by each home cell is scanned by the corresponding head of M_2.

To begin a basic step, M_2 moves each head to the left once, to the right twice, and to the left once, recording the symbols to the left and right of the home cells in its finite control. Four moves of M_2 are required, after which M_2 has returned to its home cells.

Next, M_2 determines the contents of all of M_1's tape cells represented by the home cells and their left and right neighbors at the time when some tape head of M_1 first leaves the region represented

by the home cell and its left and right neighbors. (Note that this calculation by M_2 takes no time. It is built into the transition rules of M_2.) If M_1 accepts before some tape head leaves the represented region, M_2 accepts. If M_1 halts, M_2 halts. Otherwise M_2 then visits, on each, the two neighbors of the home cell, changing these symbols and that of the home cell if necessary. M_2 positions each of it heads at the cell that represents the symbol that M_1's corresponding head is scanning at the end of the moves simulated. At most four moves of M_2 are needed.

It takes at least 16 moves for M_1 to move a head out of the region represented by a home cell and its neighbors. Thus in eight moves, M_2 has simulated at least 16 moves of M_1. ■

Bibliographic Notes

For a detailed proof of Ladner's Theorem, the reader is referred to the original paper [3]. The existence of an optimal algorithm for any NP-problem (referred to above as Levin's Theorem) was proven in [4].

The separtion of one-tape Turing machines from two-tape ones (i.e., $\text{DTime}_2(O(n))$ is NOT contained in $\text{DTime}_1(o(n^2))$) can be proved in various ways. Our presentation follows the two-step proof in [2], while omitting the second step (i.e., proving a lower bound on the error-free randomized communication complexity of `equality`). The alternative (direct) proof, presented in the appendix to this lecture, is adapted from Exercise 12.2 (for which a solution is given) in the textbook [1].

1. J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

2. E. Kushilevitz and N. Nisan. *Communication Complexity*, Cambridge University Press, 1996.

3. R.E. Ladner, "On the Structure of Polynomial Time Reducibility", *Jour. of the ACM*, 22, 1975, pp. 155–171.

4. Levin, L.A., "Universal Search Problems", *Problemy Peredaci Informacii 9*, pp. 115–116, 1973. Translated in *problems of Information Transmission 9*, pp. 265–266.

Appendix: Proof of Theorem 3.5, via crossing sequences

Consider one-tape TM M with transition function δ, input w of length m, such that M accepts w and an integer i, $0 < i < m$.

Denote by w_j, $0 \leq j \leq m$, the j'th symbol of the word w.

Consider the computation of the machine M on input w. This computation is uniquely determined by the machine description and the input, since the machine is deterministic.

The computation is a sequence of ID's (instantaneous descriptions), starting with $q_0 w_1 ... w_n$ and ending with $w_1 ... w_n p$ for some $p \in F$ (accepting or rejecting state).

Denote the elements of the computation sequence by $(ID_j)_{j=1}^{r}$ for some finite r.

Consider a sequence $(L_j)_{j=1}^{r-1}$ of pairs $L_j = (ID_j, ID_{j+1})$. For some $0 \leq i \leq m$ let $(L_{j_l})_{l=1}^{t}$ $t \leq r-1$ be the subsequence of $(L_j)_{j=1}^{r-1}$ of elements (ID_{j_l}, ID_{j_l+1}) of the form:

either

$ID_{j_l} = w_1 ... w_{i-1} p w_i ... w_n$ and $ID_{j_l+1} = w_1 ... w_i q w_{i+1} ... w_n$, for some $p, q \in Q$ and this specific i,

or

$ID_{j_l} = w_1 ... w_i p w_{i+1} ... w_n$ and $ID_{j_l+1} = w_1 ... w_{i-1} q w_i ... w_n$, for some $p, q \in Q$ and this specific i.

By definition of Turing machine computation $ID_{j_l} \vdash^M ID_{j_l+1}$ and therefore in the first case $\delta(p, w_i) = (q, w_{i+1}, R)$ and in the second case $\delta(p, w_{i+1}) = (q, w_i, L)$.

For every $1 \leq l \leq t$, let q_l be the state recorded in $ID_{j_{l+1}}$, where $ID_{j_{l+1}}$ are as above. Then the sequence $(q_l)_{l=1}^t$ is defined to be the **crossing sequence** of the triple (M, w, i) (machine M on input w in the boundary i).

Consider

$$L = \{wcw : w \in \{0,1\}^*\},$$

where c is a special symbol.

Clearly, a 2-tape TM will decide the language in $O(n)$ just by copying all the symbols before c to another tape and comparing symbol by symbol.

Let's prove that 1-tape TM will need $\Omega(n^2)$ steps to decide this language.

For $w \in \{0,1\}^m$, and $1 \leq i \leq m-1$, let $l_{w,i}$ be the length of the crossing sequence of (M, wcw, i). Denote by s the number of states of M.

Denote the average of $l_{w,i}$ over all m-long words w by $p(i)$.

Then from counting considerations, at least for half of w's it holds $l_{w,i} \leq 2 * p(i)$.

Let $N = 2^m$.So there are at least 2^{m-1} words w for which holds $l_{w,i} \leq 2 * p(i)$.

The number of possible crossing sequences of length $\leq 2 * p(i)$ is

$$\sum_{j=0}^{2*p(i)} s^j < s^{2*p(i)+1},$$

where s is the number of states of M.

So there is at least $\frac{2^{m-1}}{s^{2*p(i)+1}}$ words w of length m with the same crossing sequence for boundary $(i, i+1)$ (by pigeonhole principle). We are interested in such words w with the same suffix ($i+1, ..., m$ symbols). The number of such different suffixes is 2^{m-i}. Therefore if for some i holds

$$\frac{2^{m-1}}{s^{2*p(i)+1}} / 2^{m-i} > 1, \tag{3.3}$$

then by pigeonhole principle there are two different w's with the same suffix and the same crossing sequence between i and $i+1$ positions. We'll show that this leads to contradiction.

Denote the differing i-prefixes by α_1 and α_2 and the common $(m-i)$-bit suffix by β. Consider the input word $\alpha_1 \beta c \alpha_1 \beta$ and the input word $\alpha_2 \beta c \alpha_1 \beta$. Since the crossing sequence between α_1 and β is the same as the one between α_2 and β, the machine will not be able to distinguish between the two cases, and will accept the second word too, contradiction.

So Eq. (3.3) can not hold for any i and therefore for every i it holds

$$\frac{2^{m-1}}{2^{2*p(i)+1} 2^{m-i}} \leq 1$$

and so

$$2^{i-1} \leq 2^{2*p(i)+1},$$

implying

$$i - 1 \leq 2 * p(i) + 1,$$

and

$$p(i) \geq \frac{i-2}{2} \tag{3.4}$$

follows.

Denote by $T_m(w)$ the time needed to machine M to accept the word wcw. Let us compute the average time $Av_M(w)$ needed for M to accept wcw for an m-long word w.

$$Av_M(w) = \frac{1}{N}\sum_w T_M(w) \geq \frac{1}{N}\sum_w \sum_{i=1}^{m} l_{w,i},$$

because the running time of a TM on input w is the sum of lengths of the crossing sequences of (M, w, i) for $0 < i < m$. We have inequality here since there are crossing sequences for $i > m$ in the word wcw. Now, we have

$$\frac{1}{N}\sum_w \sum_{i=1}^{m} l_{w,i} = \sum_{i=1}^{m}\sum_w \frac{l_{w,i}}{N} = \sum_{i=1}^{m} p(i)$$

and so, by Eq. (3.4),

$$Av_M(w) \geq \sum_{i=1}^{m} p(i) \geq \sum_{i=1}^{m} \frac{i-2}{2} = \Omega(m^2).$$

So the average running time of M on wcw is $\Omega(m^2)$ implying that there exists an input wcw of length $2 * m + 1$ on which M runs $\Omega(m^2)$ steps.

Therefore, we have proved a lower bound of the worst case complexity of the language-decision problem for $L = \{wcw : w \in \{0,1\}^*\}$ on one-tape Turing machine. This lower bound is $\Omega(m^2)$ for $O(m)$-long input. On the other hand, this problem is decidable in $O(m)$ time by two-tape TM. Therefore

$$DTime_2(O(n)) \not\subseteq DTime_1(o(n^2)).$$

And so Theorem 3 is tight in the sense that there are functions $t(\cdot)$ such that

$$DTime_2(O(t(\cdot))) \not\subseteq DTime_1(o(t^2(\cdot))).$$

Lecture 4

Space Complexity

Notes taken by Leia Passoni and Reuben Sumner

Summary: In this lecture we introduce space complexity and discuss how a proper complexity function should behave. We see that properly choosing complexity functions yields as a result well-behaved hierarchies of complexity classes. We also discuss space complexity below logarithm.

4.1 On Defining Complexity Classes

So far two main complexity classes have been considered: $\mathcal{NP}$ and $\mathcal{P}$. We now consider general complexity measures. In order to specify a complexity class, we first have to set the *model of computation* we are going to use, the specific *resource* we want to bound – time or space – and finally the bound itself, that is the *function* with respect to which we want complexity to be measured.

What kind of functions $f : \mathsf{N} \mapsto \mathsf{N}$ should be considered appropriate in order to define "adequate" complexity classes? Such functions should be computable within a certain amount of the resource they bound, and *that* amount has to be a value of the function itself. In fact, choosing a too complicated function as a complexity function could give as a result that the function itself is not computable within the amount of time or space it permits. These functions are not good in order to understand and classify usual computational problems: even though we can use *any* such function in order do formally define its complexity class, strange things can happen between complexity classes if we don't choose these functions properly. This is the reason why we have defined *time constructible functions* when dealing with time complexity. For the same reason we will here define *space constructible functions*.

4.2 Space Complexity

In space complexity we are concerned with the amount of space that is needed for a computation. The model of computation we will use is a 3-*tape Turing Machine*. We use this model because it is easier to deal with it. We remind that any multi-tape TM can be simulated by an ordinary TM with a loss of efficiency that is only polynomial. For the reminder of this lecture notes, " Turing Machine" will refer to a 3-tape Turing Machine. The 3 tapes are:

1. *input tape*. Read-only

2. *output tape.* Write-only. Usually considered unidirectional: this assumption is not essential but useful. For decision problems, as considered below, one can omit the output-tape altogether and have the decision in the machine's state.

3. *work tape.* Read and write. Space complexity is measured by the bounds on the machine's position on this tape.

Writing is not allowed on the input tape: this way space is measured only on the worktape. If we allowed writing on the input tape then the length of the input itself should be taken into account when measuring space. Thus we could only measure space complexities which are at least linear. In order to consider also sublinear space bounds we restrict the input tape to be read-only. Define $W_M(x)$ to be the index of the rightmost cell on the worktape scanned by M on input x. Define $S_M(n) = max_{|x|=n} W_M(x)$. For any language L define $\chi_L(x)$ so that if $x \in L$ then $\chi_L(x) = 1$ otherwise $\chi_L(x) = 0$

Definition 4.1 (Dspace):

$$\text{Dspace}(s(n)) = \{L \,|\exists \text{a Turing machine } M, M(x) = \chi_L(x) \text{and } \forall n \; S_M(n) \leq s(n)\}$$

We may multiply $s(\cdot)$ by $\log_2 |\Gamma_M|$ where Γ_M is the alphabet used by M. Otherwise, we could always linearly compress the number of space cells using a bigger alphabet. We may also add $\log_2(|x|)$ to $s(\cdot)$, where x is the input. (However, this convention disallow treatment of sub-logarithmic space, and therefore will not be done when discussing such space bounds.) This is done in order to have a correspondence to the number of configurations.

Definition 4.2 (Configuration) : *A* configuration *of M is an instantaneous representation of the computation carried on by M on a given input x. Therefore if $|x| = n$ a configuration gives information about the following:*

- *state of M* (O(1) bits)
- *contents of the work tape* ($s(n)$ bits)
- *head position in the input tape* ($\log(n)$ bits)
- *head position in the work tape* ($\log(s(n))$ bits)

4.3 Sub-Logarithmic Space Complexity

Working with sublogarithmic space is not so useful. One may be tempted to think that whatever can be done in $o(\log(n))$ space can also be done in constant space. Formally this would mean

$$\text{Dspace}(o(\log(n))) \subseteq \text{Dspace}(O(1))$$

and since obviously $\text{Dspace}(O(1)) \subseteq \text{Dspace}(o(\log(n)))$, we may also (incorrectly) argue that in fact

$$\text{Dspace}(o(\log(n))) = \text{Dspace}(O(1))$$

This intuition comes from the following imprecise observation: if space is not constant, machine M must determine how much space to use. Determining how much space to use seems to require the machine counting up to at least $|x| = n$ which needs $O(\log(n))$ space. Therefore any M that uses less than $O(\log(n))$ cells, is forced to use constant space. It turns out that this intuition is wrong and the reason is that the language itself can help in deciding how much space to use.

Oded's Note: *This should serve as warning against making statements based on vague intuitions on how a "reasonable" algorithm should behave. In general, trying to make claims about "reasonable" algorithms is a very dangerous attitude to proving lower bounds and impossibility results. It is rarely useful and quite often misleading.*

Note: It is known that Dspace($O(1)$) equal the set of regular languages. This fact will be used to prove the following

Theorem 4.3 Dspace($o(\log(n))$) *is a proper superset of* Dspace($O(1)$).

Proof: We will show that Dspace($o(\log(n))$) $\supseteq$ Dspace($\log\log(n)$) is not contained in Dspace($O(1)$). In fact there is a language L so that $L \in$ Dspace($\log\log(n)$) but $L \notin$ Dspace($O(1)$). For simplicity, we define a language, L, over the alphabet $\{0, 1, \$\}$:

$$L = \left\{ w = 0\cdots0\$0\cdots01\$0\cdots010\$\cdots\$1\cdots1\$ \;\middle|\; \forall k \in \mathsf{N} \begin{array}{r}\text{the } l\text{-th substring of } w \text{ delimited by \$ has}\\ \text{length } k \text{ and is the binary representation}\\ \text{of the number } l-1, \text{ where } 0 \le l < 2^k\end{array} \right\}$$

It can be easily shown that L is not regular using standard pumping lemma techniques. We then prove that $L \in$ Dspace($\log\log(n)$). Note that $L = \{x_k : k \in \mathsf{N}\}$, where

$$x_k = 0^{k-2}\$0^{k-2}01\$0^{k-2}10\$0^{k-2}11\$\ldots\$1^k\$$$

First consider a simplified case where we only measure space when in fact $x = x_k \in L$, $|x_k| = (k+1)2^k$, but we need to check if $x \in L$. We have to

1. Check the first block is all 0's and the last block is all 1's
2. For any two consecutive intermediate blocks in x_k, check that the second is the binary increment by 1 of the first one.

Step (1) can be done in constant space. In Step (2) we count the number of 1's in the first block, starting from the right delimiter \$ and going left until we reach the first 0. If the number of 1's in the first block is i, we then check that in the second block there are exactly i 0's followed by 1. Then we check the remaining $k - i - 1$ digits in the two consecutive blocks are the same. On input x_k, step 2 can be done in $O(\log(k))$ space, which in terms of $n = |x_k| = (k+1)2^k$, means $O(\log\log(n))$ space.

Handling the case where $x \notin L$ while still using space $O(\log\log(n))$ is slightly trickier. If we only proceeded as above then we might be tricked by an input of the form "$0^n\$$" into using space $O(\log(n))$. We think of x being "parsed" into blocks separated by \$, doing this requires only constant space. We avoid using too much space by making k passes on the input. On the first pass we make sure that the last bit of every block is 0 then 1 then 0 and so on. On the second pass we make sure that the last two bits of every block are 00 then 01 then 10 then 11 and then back to 00 and so on. In general on the i^{th} pass we check that the last i bits of each block form an increasing sequence modulo 2^i. If we ever detect consecutive blocks of different length then we reject. Otherwise, we accept if in some (i.e., i^{th}) pass, the first block is of length i, and the entire sequence is increasing mod 2^i. This multi-pass approach, while requiring more time, is guaranteed never to use too much space. Specifically, on any input x, we use space $O(1 + \log i)$, where $i = O(\log|x|)$ is the index of the last pass performed before termination. ■

Going further on, we can consider Dspace($o(\log\log(n))$ and Dspace($O(1)$). We will show that these two complexity classes are equivalent. The kind of argument used to prove their equivalence extends the one used to prove the following simpler fact.

Theorem 4.4 *For any* $s(n) : s(n) \geq \log(n)$ $\quad$ Dspace($s(n)$) $\subseteq$ Dtime($2^{o(s(n))}$)

Proof: Fix an input $x : |x| = n$ and a deterministic machine M that accepts x in space $s(n)$. Let be $\mathcal{C}$ the number of possible configurations of M on input x. Then an upper bound for $\mathcal{C}$ is:

$$\mathcal{C} \leq |Q_M| \cdot n \cdot s(n) \cdot 2^{o(s(n))}$$

where $Q_M$is the set of states of M, n is the number of possible locations of the head on the input tape, $s(n)$ is the number of possible locations of the head on the worktape and $2^{o(s)}$ is the number of possible contents in the work tape – the exponent is $o(s)$ because the alphabet is not necessarily binary. We can write $s(n) \cdot 2^{o(s(n))} = 2^{o(s(n))}$ and since s is at least logarithmic , $n \leq 2^{o(s(n))}$. Therefore

$$\mathcal{C} \leq 2^{o(s(n))}$$

M cannot run on input x for a time $t(n) > 2^{s(n)}$. Otherwise, M will go through the same configuration at least twice, entering an infinite loop and never stop. Then necessarily M has to run in time $t(n) \leq 2^{o(s)}$. ■

Theorem 4.5 Dspace($o(\log_2\log_2(n))$ = Dspace($O(1)$)

Proof: Consider a $s(\cdot)$-bounded machine M on the alphabet $\{0,1\}$.

Claim: given input $x : |x| = n$ such that M accepts x, then M can be on every cell on the input tape at most $k = 2^{s(n)} \cdot s(n) \cdot |Q_M| = O\left(2^{s(n)}\right)$ times. The reason being that if M were to be on the cell more than k times then it would be in the same configuration twice, and thus never terminate.

We define a semi-configuration as a configuration with the position on the input tape replaced by the symbol at the current input tape position. For every location i on the input tape, we consider all possible semi-configurations of M when passing location i. If the sequence of such configurations is $\mathcal{C}^i = \mathcal{C}^i_1, \ldots, \mathcal{C}^i_r$ then by the above claim its length is bounded: $r \leq O\left(2^{s(n)}\right)$. The number of possible different sequences of semi-configurations of M, associated with any position on the input tape, is bounded by

$$\left(2^{s(n)}\right)^{\left(2^{s(n)}\right)} = 2^{2^{O(s(n))}}$$

Since $s(n) = o(\log_2\log_2 n)$ then $2^{2^{O(s(n))}} = o(n)$ and therefore there exists $n_0 \in \mathsf{N}$ such that $\forall n \geq n_0$, $2^{2^{O(s(n))}} < \frac{n}{3}$. We then show that $\forall n \geq n_0$, $s(n) = s(n_0)$. Thus $L \in$ Dspace($s(n_0)$) = Dspace($O(1)$) proving the theorem.

Assume to the contrary that there exists an n' such that $s(n') > s(n_0)$. Let $n_1 = \min_{|x|>n_0}\{W_M(x) > s(n_0)\}$ and let $x_1 \in \{0,1\}^{n_1}$ be such that $W_M(x_1) > s(n_0)$. That is, x_1 is the shortest input on which M uses space more than $s(n_0)$.

The number of sequences of semi-configurations at any position in the input tape is $< \frac{n_1}{3}$. So labelling n_1 positions on the input tape by at most $\frac{n_1}{3}$ sequences means that there must be at least three positions with the same sequence of semi-configurations. Say $x_1 = \alpha a \beta a \gamma a \sigma$. Where each of the positions with symbol a has the same sequence of semi-configurations attached to it.

Claim: The machine produces the same final semi-configuration with either βa or γa eliminated from the input. For the sake of argument consider cutting β leaving $x_1' = \alpha a \gamma a \sigma$. On x_1' the machine proceeds on the input exactly as with x_1 until it first reaches the a. This is the first entry in our sequence of semi-configurations. Locally, M will make the same decision to go left or right on x_1' as it did on x_1 since all information stored in the machine at at the current read head position is identical. If the machine goes left then its computation will proceed identically on x_1' as on x_1 because it still hasn't seen any difference in input and will either terminate or once again come to the first a. On the other hand consider the case of the machine going right. Say this is the ith time at the first a. We now compare the computation of M to what it did following the ith time going past the second a (after the now nonexistent β). Since the semi-configuration is the same in both cases then on input x_1 the machine M also went right on the ith time seeing the second a. The machine proceeded and either terminated or came back for the $i+1$st time to the second a. In either case on input x_1' the machine M is going to do the same thing but now on the first a. Continuing this argument as we proceed through the sequence of semi-configurations (arguing each time that on x_1' we will have the same sequence of semi-configurations) we see that the final semi-configuration on x_1' will be same as for x_1. The case in which γa is eliminated is identical.

Now consider the space usage of M on x_1. Let $x_2 = \alpha a \beta a \sigma$ and $x_3 = \alpha a \gamma a \sigma$. If peak space usage processing x_1 was in αa or σ then $W_M(x_2) = W_M(x_3) = W_M(x_1)$. If peak space usage was in βa then $W_M(x_3) \leq W_M(x_2) = W_M(x_1)$. If peak space usage was in γa then $W_M(x_2) \leq W_M(x_3) = W_M(x_1)$. Choose $x_1' \in \{x_2, x_3\}$ to maximize $W_M(x_1')$. Then $W_M(x_1') = W_M(x_1)$ and $|x_1'| < |x_1|$. This contradicts our assumption that x_1 was a minimal length string that used more than $s(n_0)$ space. Therefore no such x_1 exists. ∎

Discussion: Note that the proof of Theorem 4.4 actually establishes $\text{Dspace}(O(\log\log n)) \neq \text{Dspace}(O(1))$. Thus, combined with Theorem 4.5 we have a separation between $\text{Dspace}(O(\log\log n))$ and $\text{Dspace}(o(\log\log n))$.

The rest of our treatment focuses on space complexity classes with space bound which is at least logarithmic. Theorem 4.5 says that we can really dismiss space bounds below double-logarithmic (alas Theorem 4.4 says there are some things beyond finite-automata that one may do with sub-logarithmic space).

4.4 Hierarchy Theorems

As we did for time, we give now

Definition 4.6 (Space Constructible Function): *A* space constructible function *is a function* $s : \mathsf{N} \mapsto \mathsf{N}$ *for which there exists a machine* M *of space complexity at most* $s(\cdot)$ *such that* $\forall n \quad M(1^n) = s(n)$

For sake of simplicity, we consider only machines which halt on every input. Little generality is lost by this –

Lemma 4.4.1 *For any space bounded Turing Machine* M *using space* $s(n)$*, where* $s(n)$ *is at least* $\log(n)$ *and space constructible we can construct* $M' \in \text{Dspace}(O(s(n)))$ *such that* $L(M') = L(M)$ *and machine* M' *halts on all inputs.*

Proof: Machine M' first calculates a time bound equal to the number of possible configurations of M which is $2^{s(n)} \cdot s(n) \cdot n \cdot |Q_M|$. This takes space $s(n)$, and same holds for the counter to be

maintained in the sequel. Now we simulate the computation of M on input x and check at every step that we have not exceeded the calculated time bound. If the simulated machine halts before reaching its time bound we accept or reject reflecting the decision of the simulated machine. If we reach the time bound before the simulated machine terminates that we are assured that the simulated machine will never terminate, in particular never accept, and we reject the input. ■

Theorem 4.7 (Space Hierarchy Theorem): *For any space-constructible* $s_2 : \mathsf{N} \mapsto \mathsf{N}$ *and every at least logarithmic function* $s_1 : \mathsf{N} \mapsto \mathsf{N}$ *so that* $s_1(n) = o(s_2(n))$*, the class* $\mathrm{Dspace}(s_1(n))$ *is strictly contained in* $\mathrm{Dspace}(s_2(n))$.

We prove the theorem only for machines which halt on every input. By the above lemma, this does not restrict the result in case s_1 is space-constructible. Alternatively, the argument can be extended to deal also with non-halting machines.

Proof: The idea is to construct a language L in $\mathrm{Dspace}(s_2(n))$ such that any machine M using space s_1 will fail recognizing L. We will enumerate all machines running in space s_1 and we will use a diagonalization technique.

- Compute the allowed bound on input x: for instance let it be $\frac{1}{10}s_2(|x|)$.
- Write the language:

$$L = \left\{ x \in \{0,1\}^* \;\middle|\; \begin{array}{l} x \text{ is of the form } \langle M \rangle 01^* \text{ and such that} \\ \text{- } |\langle M \rangle| < \frac{1}{10}s_2(|x|) \\ \text{- on input } x, M \text{ rejects } x \text{ while using at most space } \frac{1}{10}s_2(|x|) \end{array} \right\}$$

  Here $\langle M \rangle$ is a binary encoding of the machine M, so we can see $x \in L$ as a description of M itself.

- Show that $L \in \mathrm{Dspace}(s_2(n))$ and $L \notin \mathrm{Dspace}(s_1(n))$.

To see that $L \in \mathrm{Dspace}(s_2(n))$, we write an algorithm that recognizes L:

On input x:

1. Check if x is of the right form
2. Compute the space bound $S \leftarrow \frac{1}{10}s_2(|x|)$
3. Check the length of $\langle M \rangle$ is correct: $|\langle M \rangle| < \frac{1}{10}s_2(|x|)$.
4. Emulate the computation of machine M on input x. If M exceeds the space bound then $x \notin L$ so we reject.
5. If M rejects x then accept. Else, reject.

The computation in Step (1) can be done in $O(1)$ space. The computation of S in Step (2) can be done in space $s_2(|x|)$ because s_2 is space constructible. Step (3) needs $\log(S)$ space. In Step(4) we have to make sure that (# cells M scans) $\times$ $(\log_2 |\Gamma_M|) < S$. Checking that M does not exceed the space bound needs space S. As for the implementation of Step(4), on the work tape we first copy the description $\langle M \rangle$ and then mark a specific area in which we are allowed to operate. Then it is possible to emulate the behavior of M going back and forth on the work tape from $\langle M \rangle$ to the

simulated machine's work area, stopping when we are out of space. The algorithm then is running in $\text{Dspace}(s_2(n))$.
Note: Since we want to measure space, we are not concerned on how much time is "wasted" going back and forth on the worktape from the description of M to the operative area.

Now we have to see that $L \notin \text{Dspace}(s_1(n))$.
We will show that for every machine M of space complexity s_1, $L(M) \neq L$.

There exists $n : s_1(n) < \frac{1}{10}s_2(n)$ since $s_1(n) = o(s_2(n))$. We then consider $M : |\langle M\rangle| < \frac{1}{10}s_2(n)$ and see how M acts on input x of the form $x = \langle M\rangle 01^{n-(|\langle M\rangle|+1)}$ – note that it is always possible to find inputs of the above form for any sufficiently large n. There are two cases:

1. if M accepts x, then (by definition of L) $x \notin L$.
2. if M rejects x, then since $|\langle M\rangle| < \frac{1}{10}s_2(n)$ and $M(x)$ uses at most $s_1(|x|) < \frac{1}{10}s_2(|x|)$ space, $x \in L$.

In either case $L(M) \neq L$. Therefore any M using space s_1 cannot recognize L. ■

Theorem 4.8 (Time Hierarchy Theorem): *For any time-constructible* $t_2 : \mathsf{N} \mapsto \mathsf{N}$ *and every at least linear function* $t_1 : \mathsf{N} \mapsto \mathsf{N}$ *so that* $\lim_{n\to\infty} \frac{t_1(n)\log(t_2(n))}{t_2(n)} = 0$, *the class* $\text{Dtime}(t_1)$ *is strictly contained in* $\text{Dtime}(t_2)$.

Proof: It is analogous to the previous one used for space. The only difference is in the definition of the language L:

$$L = \left\{ x \in \{0,1\}^* \;\middle|\; \begin{array}{l} x \text{ is of the form } \langle M\rangle 01^* \text{ and such that} \\ \text{- } |\langle M\rangle| < \frac{1}{10}\log(t_2(|x|)) \\ \text{- on input } x, M \text{ rejects } x \text{ while using at most time } \frac{1}{10\log t_2(|x|)}t_2(|x|) \end{array} \right\}$$

Dealing with time, we require $|\langle M\rangle| < \log(t_2(|x|))$. The reason for requiring a small description for M is that we cannot implement Step (4) of the algorithm as it has been done with space: scanning $\langle M\rangle$ and going back and forth from $\langle M\rangle$ to the operative area would blow up time. In order to save time we copy $\langle M\rangle$ in the operative area on the worktape, shifting $\langle M\rangle$ while moving on the worktape to the right. If $|\langle M\rangle| < \log(t_2(|x|))$ it takes then time $\log t_2(|x|)$ to copy $\langle M\rangle$ when needed and time $\log(t_2(|x|))$ to scan it. In Step (4) each step of the simulated machine takes time $O(\log(t_2(|x|)))$ so the total execution time will be

$$\log(t_2(|x|)) \cdot \frac{t_2(|x|)}{10 \cdot \log(t_2(|x|))} = O(t_2(|x|))$$

The logarithmic factor we have to introduce in Step (4) for the simulation of M is thus the reason why in Time Hierarchy Theorem we have to increase the time bound by a logarithmic factor in order to get a bigger complexity class. ■

The Hierarchy Theorems show that increasing the time-space bounding functions by any small amount, gives as a result bigger time-space complexity classes – which is what we intuitively would expect: given more resources, we should be able to recognize more languages.

However, it is also clear that the complexity classes hierarchy is strict only if we use proper time/space bounding functions, namely *time and space constructible* functions. This is not the case if we allow any recursive function for defining complexity classes, as it can be seen in the following theorems.

4.5 Odd Phenumena (The Gap and Speed-Up Theorems)

The following theorems are given without proofs, which can be found in [1].

Theorem 4.9 (Borodin's Gap Theorem): *For any recursive function* $g : \mathsf{N} \mapsto \mathsf{N}$ *with* $g(n) \geq n$, *there exists a recursive function* $s_1 : \mathsf{N} \mapsto \mathsf{N}$ *so that for* $s_2(n) = g(s_1(n))$, *the class* $\mathrm{Dspace}(s_1(n))$ *equals* $\mathrm{Dspace}(s_2(n))$.

Theorem 4.9 is in a sense the opposite of the Space Hierarchy Theorem: between space bounds $s_1(n)$ and $g(s_1(n))$ there is no increase in computational power. For instance, with $g(n) = n^2$ one gets $g(s_1(n)) = s_1(n)^2$. The idea is to choose $s_1(n)$ that grows very fast and such that even if $g(s_1(n))$ grows faster, no language can be recognized using a space complexity in between.

> Oded's Note: *The proof can be extended to the case where* $g : \mathsf{N} \times \mathsf{N} \mapsto \mathsf{N}$ *and* $s_2(n) = g(n, s_1(n))$. *Thus, one can have* $s_2(n) = n \cdot s_1(n)$, *answering a question raised in class.*

Theorem 4.10 (Blum's Speed-up Theorem): *For any recursive function* $g : \mathsf{N} \mapsto \mathsf{N}$ *with* $g(n) \geq n$, *there exists a recursive language* L *so that for any machine* M *deciding* L *in space* $s : \mathsf{N} \mapsto \mathsf{N}$ *there exists a machine* M' *deciding* L *in space* $s' : \mathsf{N} \mapsto \mathsf{N}$ *with* $s'(n) = g^{-1}(s(n))$.

So there exist languages for which we can always choose a better machine M recognizing them.

> Oded's Note: *Note that an analogous theorem for time-complexity (which holds too), stands in some contrast to the optimal algorithm for solving NP-search problems presented in the previous lecture.*

Bibliographic Notes

Our presentation is based mostly on the textbook [1]. A proof of the hierarchy theorem can also be found in [4]. The proofs of Theorems 4.9 and 4.10 can be found in [1]. Theorem 4.3 and 4.5 are due to [2] and [3] respectively.

1. J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

2. Lewis, Stearns, Hartmanis, "Memory bounds for recognition of context free and context sentitive languages", in proceedings of *IEEE Switching Circuit Theory and Logical Design* (old FOCS), 1965, pages 191–202.

3. Stearns, Lewis, Hartmanis, "Hierarchies of memory limited computations", in proceedings of *IEEE Switching Circuit Theory and Logical Design* (old FOCS), 1965, pages 179–190.

4. M. Sipser. *Introduction to the Theory of Computation*, PWS Publishing Company, 1997.

Lecture 5

Non-Deterministic Space

Notes taken by Yoad Lustig and Tal Hassner

Summary: We recall two basic facts about deterministic space complexity, and then define non-deterministic space complexity. Three alternative models for measuring non-deterministic space complexity are introduced: the standard non-deterministic model, the online model and the offline model. The equivalence between the non-deterministic and online models and their exponential relation to the offline model are proved. After the relationships between the non-deterministic models are presented we turn to investigate the relation between the non-deterministic and deterministic space complexity. Savitch's Theorem is presented and we conclude with a translation lemma.

5.1 Preliminaries

During the last lectures we have introduced the notion of space complexity, and in order to be able to measure sub-linear space complexity, a variant model of a Turing machine was introduced. In this model in addition to the work tape(s) and the finite state control, the machine contains two special tapes : an input tape and an output tape. These dedicated tapes are restricted each in it's own way. The input tape is read only and the output tape is write only and unidirectional (i.e. the head can only move in one direction).
In order to deal with non-deterministic space complexity we will have to change the model again, but before embarking on that task, two basic facts regarding the relations between time and space complexity classes should be reminded.

To simplify the description of asymptotic behaviour of functions we define :

Definition 5.1 *Given two functions* $f : \mathsf{N} \to \mathsf{N}$ *and* $g : \mathsf{N} \to \mathsf{R}$
f is at least g *if there exists an* $n_0 \in \mathsf{N}$ *s.t. for all* $n \geq n_0$ $f(n) \geq \lceil g(n) \rceil$.
f is at least linear *if there exists a linear function* g *s.t.* f *is at least* g *(there exists a constant* $c > 0$ *s.t.* f *is at least* cn*).*

Fact 5.1.1 *For every function* $S(\cdot)$ *which is at least* $log(\cdot)$ $DSPACE(S) \subseteq DTIME(2^{O(S)})$.

Proof: Given a Turing machine M, a complete description of it's computational state on a fixed input at time t can be given by specifying :

- The contents of the work tape(s).
- The location of the head(s) on work tape(s).
- The location of the head on the input tape.
- The state of the machine.

Denote such a description a *configuration of* M. (Such a configuration may be encoded in many ways, however in the rest of the discussion we will assume a standard encoding was fixed, and would not differentiate between a configuration and it's encoding. For example we might refer to the space needed to hold such a configuration. This is of course the space needed to hold the representation of the configuration and therefore this is a property of the encoding method, however from an asymptotic point of view the minor differences between reasonable encoding methods make little difference). A complete description of an entire computation can be made simply by specifying the configuration at every time t of the computation.

If during a computation at time t, machine M reached a configuration in which it has already been in at time $t_1 < t$, (i.e. the configurations of M at times t_1 and t are identical), then there is a cycle in which the machine moves from one configuration to the next ultimately returning to the original configuration after $t - t_1$ steps. Since M is deterministic such a cycle cannot be broken and therefore M's computation will never end.
The last observation shows that during a computation in which M stops, there are no such cycles and therefore no configuration is ever reached twice. It follows that the running time of such a machine is bounded by the number of possible configurations, so in order to bound the time it is enough to bound the number of possible configurations.

If a machine M never uses more than s cells, then on a given input x, the number of configurations is bounded by the number of possible contents of s cells (i.e. $|\Gamma_M|^s$, where Γ_M is the tape alphabet of machine M),times the number of possible locations of the work head (i.e. s), times the number of possible locations of the input head (i.e. $|x|$), times the number the possible states (i.e. $|S_M|$). If the number of cells used by a machine is a function of the input's length the same analysis holds and gives us a bound on the number of configurations as a function of the input's length.
For a given machine M and input x, denote by $\#conf(M,x)$ the number of possible configurations of machine M on input x. We have seen that for a machine M that works in space $S(\cdot)$ on input x, $\#conf(M,x) = |\Gamma_M|^{S(|x|)} \cdot S(|x|) \cdot |x| \cdot |S_M| = 2^{O(S(|x|))} \cdot |x|$

Therefore in the context of the theorem (i.e. $S(|x|) = \Omega(log(|x|))$) we get that on input x the time of M's computation is bounded by : $\#conf(M,x) = 2^{O(S(|x|))}$ ■

Fact 5.1.2 *For every function* $T(\cdot)$ $DTIME(T) \subseteq DSPACE(T)$.

Proof: Clearly no more then $T(|x|)$ cells can be reached by the machine's head in $T(|x|)$ steps. ■

Note : In the (far) future we will show a better bound (i.e. $DTIME(T) \subseteq DSPACE(\frac{T}{log(T)})$) which is non-trivial.

5.2 Non-Deterministic space complexity

In this section we define and relate three different models of non-deterministic space complexity.

5.2.1 Definition of models (online vs offline)

During our discussion on $\mathcal{NP}$ we noticed that the idea of a non-deterministic Turing machine can be formalized in two approaches, the first approach is that the transition function of the machine is non-deterministic (i.e. the transition function is a multi-valued function), in the second approach the transition function is deterministic but in addition to the input the machine gets an extra string (viewed as a guess): the machine is said to accept input x iff there exists a guess y s.t. the machine's computation on (x,y) ends in an accepting state. (In such a case y is called a witness for x).

In this section we shall try to generalize these approaches and construct a model suitable for measuring non-deterministic space complexity. The first approach can be applied to our standard turing machine model.
Put formally, the definition of a non-deterministic Turing machine under the first approach is as follows :

Definition 5.2 (non-deterministic Turing machine): *A* non deterministic Turing machine *is a Turing machine with a non-deterministic transition function, having a work tape, a read-only input tape, and a unidirectional write-only output tape. The machine is said to accept input x if there exists a computation ending in an accepting state.*

Trying to apply the second approach in the context of space complexity a natural question arises : should the memory used to hold the guess be metered?
It seems reasonable not to meter that memory as the machine does not "really" use it for computation. (Just as the machine does not "really" use the memory that holds the input). Therefore a special kind of memory (another tape) must be dedicated to the guess and that memory would not be metered. However if we do not meter the machine for the guess memory, we must restrict the access to the guess tape, just as we did in the case of the input tape. (surely if we allow the machine to write on the guess tape without being metered and that way get "free" auxiliary memory that would be cheating).
It is clear that the access to the guess tape should be read only.

Definition 5.3 (offline non-deterministic Turing machine): *An* offline non-deterministic Turing machine *is a Turing machine with a work tape, a read-only input tape, a two-way read-only guess tape, and a unidirectional write-only output tape, where the contents of the guess tape is selected non-deterministically. The machine is said to accept input x if there exists contents to the guess tape (a guess string y) s.t. when the machine starts working with x in the input tape and y in the guess tape it eventually enters an accepting state.*

As was made explicit in the definition, there is another natural way in which access to the guess tape can be farther limited: the tape can be made unidirectional (i.e. allow the head to move only in one direction).

Definition 5.4 (online non-deterministic Turing machine): *An* online non-deterministic Turing machine *is a Turing machine with a work tape,a read-only input tape, a unidirectional read-only guess tape (whos contents are selected non-deterministicly), and a unidirectional write-only output tape. Again, the machine is said to accept x if there exists a guess y s.t. the machine working on (x,y)*

will eventually enter an accepting state.

An approach that limits the guess tape to be unidirectional seems to correspond to an online guessing process – a non-deterministic machine works and whenever there are two (or more) possible ways to continue the machine guesses (online) which way to choose. If such a machine "wants" to "know" which way it guessed in the past, it must record it's guesses (use memory). On the other hand, the approach that allows the guess tape to be two-way corresponds to an offline guessing process i.e. all the guesses are given before hand (as a string) and whenever the machine wants to check what was guessed at any stage of the computation, it can look at the guesses list.

It turns out that the first non-deterministic model and the online model are equivalent. (Although the next claim is phrased for language decision problems, it holds with the same proof for other kinds of problems).

Claim 5.2.1 *For every language L there exists a non-deterministic Turing machine M_N that identifies L in time $O(T)$ and space $O(S)$ iff there exists an online Turing machine M_{on} that identifies L in time $O(T)$ and space $O(S)$.*

Proof: Given M_N it can be easily transformed to an online machine M_{on} in the following way: M_{on} simulates M_N and whenever M_N has several options for a next move (it must choose non-deterministicaly which option to take), M_{on} decides which option to take according to the content of the cell scanned at the guess tape, then move the guess tape head one cell to the right.

In some cases we may want to restrict the alphabet of the guess (for example to $\{0,1\}$). In those cases there is a minor flaw in the above construction as the number of options for M_N's next move may be bigger than the guess alphabet thus the decision which option to take cannot be made according to the content of a single guess tape cell. This is only an apparent flaw since we can assume with out loss of generality that M_N has at most two options to choose from. Such an assumption can be made since a choice from any number of options can be transformed to a sequence of choices from two options at a time by building a binary tree with the original options as leaves. This kind of transformation can be easily implemented on M_N by adding states that correspond to the inner nodes of the tree. The time of the transformed machine has increased at most by a factor of the height of the tree which is constant in the input size.

The transformation from an online machine M_{on} to a non-deterministic machine is equally easy: If we would have demanded that the guess head of M_{on} must advance every time, the construction would have been trivial i.e. at every time M_{on} moves according to it's state and the contents of the cells scanned by the input-tape, work-tape and guess-tape heads, if the contents of the guess cell scanned are not known there may be several moves possible (one for each possible guess symbol), M_N could have simply choose non-deterministically between those. However as we defined it, the guess tape head may stay in place, in such a case the non-deterministic moves of the machine are dependendent (are fixed by the same symbol) untill the guess head moves again. This is not a real problem, all we have to do is remember the current guess symbol, i.e. M_N states would be $S_{M_{on}} \times \Sigma$ where $S_{M_{on}}$ is $M'_{on}s$ states and Σ is the guess alphabet, (M_N being in state (s,a) corresponds to M_{on} being in state s while it's guess head scans a). The transition function of M_N is defined in the natural way. Suppose M_N is in state (s,a) and scans symbols b and c in it's work and input tapes, this correspond to M_{on} being in state s while scanning a,b and c. In this case M_{on} transition function is well defined, (denote the new state by s'), M_N will move the work and input heads as M_{on} moves it's heads, if the guess head of M_{on} stays fixed then the new state of M_N is (s',a),

otherwise M_{on} reads a new guess symbol, so M_N chooses non-deterministically a new state of the form (s', a') (i.e. guesses what is read from the new guess tape cell). ■

These models define complexity classes in a natural way. In the following definitions $M(x, y)$ should be read as "the machine M with input x and guess y".

Definition 5.5 ($NSPACE_{on}$): *For any function* $T : \mathsf{N} \to \mathsf{N}$

$$NSPACE_{on}(T) \stackrel{\text{def}}{=} \left\{ L \subseteq \Sigma^* \;\middle|\; \begin{array}{l} \textit{There exists an online Turing machine } M_{on} \textit{ s.t. for any input } x \in \Sigma^* \\ \textit{there exists a witness } y \in \Sigma^* \textit{ for which } M_{on}(x, y) \textit{ accepts iff } x \in L, \\ \textit{and that for any } y \in \Sigma^* \; M_{on} \textit{ uses at most } T(|x|) \textit{ space.} \end{array} \right\}$$

Definition 5.6 ($NSPACE_{off}$): *For any function* $T : \mathsf{N} \to \mathsf{N}$

$$NSPACE_{off}(T) \stackrel{\text{def}}{=} \left\{ L \subseteq \Sigma^* \;\middle|\; \begin{array}{l} \textit{There exists an offline Turing machine } M_{off} \textit{ s.t. for any input } x \in \Sigma^* \\ \textit{there exists a witness } y \in \Sigma^* \textit{ for which } M_{off}(x, y) \textit{ accepts iff } x \in L, \\ \textit{and that for any } y \in \Sigma^* \; M_{off} \textit{ uses at most } T(|x|) \textit{ space.} \end{array} \right\}$$

5.2.2 Relations between $NSPACE_{on}$ and $NSPACE_{off}$

In this section the exponential relation between $NSPACE_{on}$ and $NSPACE_{off}$ will be established.

Theorem 5.7 *For any function* $S : \mathsf{N} \to \mathsf{N}$ *so that* S *is at least logarithmic and* $\log S$ *is space constructible,*
$NSPACE_{on}(S) \subseteq NSPACE_{off}(log(S))$.

Given an online machine M_{on} that works in space bounded by S we shall construct an offline machine M_{off} which recognizes the same language as M_{on} and works in space bounded by $O(log(S))$. We will see later (Theroem 8) the opposite relation i.e. given an offline machine M_{off} that works in space S, one can construct an online machine M_{on} that recognizes the same language and works in space $2^{O(S)}$.

The general idea of the proof is that if we had a full description of the computation of M_{on} on input x, we can just look at the end of the computation and copy the result (many of us are familiar with the general framework from our school days). The problem is that M_{off} does not have a computation of M_{on} however it can use the power of non-determinism to guess it. This is not the same as having a computation, since M_{off} cannot be sure that what was guessed is really a computation of M_{on} on x. This has to be checked before copying the result. (The absence of the last stage caused many of us great troubles in our school days).

To prove the theorem all we have to show is that checking that a guess is indeed a computation of a space $S(\cdot)$-online machine can be done in $log(S(|x|))$ space. To do that we will first need a technical result concerning the length of computations of such a machine M_{on}, this result is obtained using a similar argument to the one used in the proof of Fact 5.1.1 ($DSPACE(S) \subseteq DTIME(2^{O(S)})$).

Proof: *(Theorem 5.7:* $NSPACE_{on}(S) \subseteq NSPACE_{off}(log(S))$*):*
Given an online machine M_{on} that works in space bounded by S we shall construct an offline machine M_{off} which recognize the same language as M_{on} and works in space bounded by $O(log(S))$. Using claim 2.1, there exists a non-deterministic machine M_N equivalent to M_{on}, so it is enough to construct M_{off} to be equivalent to M_N.

As in the proof of Fact 5.1.1 ($DSPACE(S) \subseteq DTIME(2^{O(S)})$) we would like to describe the state of the computation by a configuration. (As M_N uses a different model of computation we

must redefine configuration to capture the full description of the computation at a given moment, however after re-examination we discover that the state of the computation in the non-deterministic model is fully captured by the same components i.e. the contents of the work tape, the location of the work and input tape heads and the state of the machine, so the definition of a configuration can remain the same).

Claim 5.2.2 *If there exists an accepting computation of M_N on input x then there exists such a computation in which no configuration appears more than once.*

Proof: Suppose that $c_0, c_1, \ldots, c_n$ is a description of an accepting computation as a sequence configurations in which some configuration appear more than once. We can assume, without loss of generality that both c_0 and c_n appear only once. Assume for $0 < k < l < n$, $c_k = c_l$. We claim that $c_0, \ldots, c_k, c_{l+1}, \ldots, c_n$ is also a description of an accepting computation. To prove that, one has to understand when is a sequence of configurations a description of an accepting computation, This is the case if the following hold :

1. The first configuration (i.e. c_0) describes a situation in which M_N starts a computation with input x (initial state, the work tape empty).

2. Every configuration c_j is followed by a configuration (i.e. c_{j+1}) that is possible in the sense that, M_N may move in one step from c_j to c_{j+1}.

3. The last configuration (i.e. c_n) describes a situation in which the M_N accepts.

When $c_{k+1}, \ldots, c_l$ (the cycle) is removed properties 1 and 3 do not change as c_0 and c_n remain the same. Property 2 still holds since c_{l+1} is possible after c_l and therefore after c_k.
$c_0, \ldots, c_k, c_{l+1}, \ldots, c_n$ is a computation with a smaller number of identical configurations and clearly one can iterate the process to get a sequence with no identical configurations at all. ■

Remark : The proof of the last claim follows a very similar reasoning to the proof of Fact 5.1.1 ($DSPACE(S) \subseteq DTIME(2^{O(S)})$), but with an important difference. In the context of non-determinism it is possible that a computation of a given machine is arbitrarily long (the machine can enter a loop and leave it non-deterministicaly). The best that can be done is to prove that short computations exist.

We saw that also arbitrarily long computations may happen, these computations do not add power to the model since the same languages can be recognized if we forbid long computations. A similar question may rise regarding infinite computations. A machine may reject either by halting in a rejecting (non-accepting) state, or by entering an infinite computation, it is known that by demanding that all rejecting computations of a turing machine will halt, one reduces the power of the model (the class R as opposed to RE), the question is is the same true for space bounded machines ? It turns out that this is not the case (i.e. we may demand with out loss of generality that every computation of a space bounded machine halts). By Claim 5.2.2 machine that works in space S works in time $2^{O(S)}$, we can transform such a machine to a machine that always halts by adding a time counter that counts untill the time limit has passed and then halts in a rejecting state (time out). Such a counter would only cost $log(2^{O(S)}) = O(S)$ so adding it does not change the space bound significantly.

Now we have all we need to present the idea of the proof.

Given input x machine M_{off} will guess a sequence of at most $\#conf(M, x)$ of configurations of M_N, and then check that it is indeed an accepting computation by verifying properties 1–3 (in

the proof of Claim 5.2.2). If the guess turns out to be an accepting computation, M_{off} will accept otherwise reject.

How much space does M_{off} need to do the task?

The key point is that in order to verify these properties M_{off} need only look at 2 consecutive configurations at a time and even those are already on the guess tape, so the work tape only keeps a fixed number of counters (pointing to the interesting cell numbers on the guess and input tapes).

M_{off} treats it's guess as if it is composed of blocks, each contains a configuration of M_{on}.
To verify property 1, all M_{off} has to do is check that the first block (configuration) describes an initial computational state i.e. check that M_N is in the initial state and that the work tape is empty. That can be done using $O(1)$ memory.
To verify property 2 for a specific couple of consecutive configurations M_{off} has to check that the contents of the work tape in those configurations is the same except perhaps the cell on which M_N's work head was, that the content of the cell the head was on, the state of the machine and the new location of the work head are the result of a possible move of M_N. To do that M_{off} checks that these properties hold for every two consecutive blocks on the guess tape. This can be done using a fixed number of counters (each capable of holding integers upto the length of a single block) + $O(1)$ memory.
To verify property 3 all M has to do is to verify the last block (configuration) describes an accepting configuration. That can be done using $O(1)$ memory.
All that is left is to calculate the space needed to hold a counter. This is the maximum between log the size of a configuration and $log(|x|)$. A configuration is composed of the following parts :

- The contents of the work-tape – $O(S(|x|))$ cells
- The location of the work head – $log(O(S(|x|)))$ cells
- The state of the machine M_N – $O(1)$ cells
- The location of the input head – $O(log(|x|))$ cells

Since S is at least logaithmic, the length of a configuration is $O(S(|x|))$, and the size of a counter which points to location in a configuration is $O(1) + \log(S(|x|)))$.

Comment: Two details which were omitted are (1) the low-level implementation of the verification of property 2, and (2) dealing with the case that the guess is not of the right form (i.e., does not consists of a sequence of configurations of M_{on}). ■

Theorem 5.8 *For any space constractable function $S : \mathsf{N} \to \mathsf{N}$ which is at least logarithmic. $NSPACE_{off}(S) \subseteq NSPACE_{on}(2^{O(S)})$.*

As in the last theorem, given a machine of one model we would like to find a machine of the other model accepting the same language. This time an offline machine M_{off} is given and we would like to construct an online machine M_{on}.

In such a case the naive approach is simulation, i.e. trying to build a machine M_{on} that simulates M_{off}. This approach would not give us the space bound we are looking for, however, trying to follow that approach will be instructive, so that is what we will do.

The basic approach is to try and simulate M_{off} by an online machine M_{on} (in the previous theorem we did even better than that by guessing the computation and only verifying it's correctness

(that way the memory used to hold the computation was free). This kind of trick will not help us here because the process of verification involves comparing two configurations and in an online machine that would force us to copy a configuration to the work tape. Since holding a configuration on the work tape costs $O(S(|x|))$ space we might as well try to simulate M_{off} in a normal way).

Since we only have an online machine which cannot go back and forth on the guess tape, the straightforward approach would seem to be : guess the content of a guess tape for M_{off} then copy it to the work tape of the online machine M_{on}. That gives M_{on} two way access to the guess and now M_{on} can simulate M_{off} in a straight forward way. The only question remains how much space would be needed ? (clearly at least as long as the guess)

The length of the guess can be bounded using a similar analysis to the one we saw at Fact 5.1.1 ($DSPACE(S) \subseteq DTIME(2^{O(S)})$), only this time things are a bit more complicated.

If we look on M_{off}'s guess head during a computation it moves back and forth thus it's movement forms a "snake like path" over the guess tape.

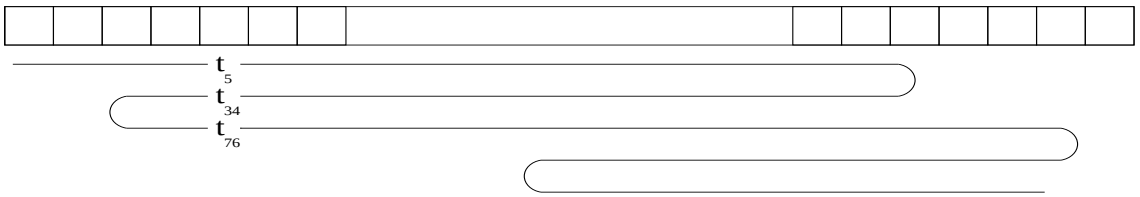


Figure 5.1: The guess head movement

The guess head can visit a cell on the guess tape many times, but we claim the number of times a cell is visited by the head can be bounded. The idea is, as in Fact 5.1.1, that a machine cannot be in the exact same situation twice without entering an infinite loop.

To formalize the last intuition we would need a notion of configuration (a machine's exact situation) this time for an offline machine. To describe in full the computational state of an offline machine one would have to describe all we described in a deterministic model (contents of work tape, location of work and input head and the machine state) and in addition the contents of the guess tape and the location of the guess head. However we intend to use the configuration notion for a very specific purpose, in our case we are dealing with a specific cell on the guess tape while the guess is fixed. Therefore denote by CWG (configuration without guess) of M_{off} its configuration without the the guess tape contents and the guess head location. (exactly the same components as in the non-deterministic configuration). Once again the combinatorial analysis shows us that the number of possible CWGs is $|\Gamma|^{S(|x|)}S(|x|)|S_M|log(|x|)$ which is equal to $\#conf(M,x)$.

Claim 5.2.3 *The number of times during an accepting computation of M_{off} in which the guess tape head visits a specified cell is lesser or equal to $\#conf(M,x)_M = 2^{O(S)}$.*

Proof: If M_{off} visits a single cell twice while all the parameters in the CWG (contents of work tape, location of work and input head and state of the machine) are the same then the entire computation state is the same, because the contents of the guess tape and the input remains fixed throughout the computation. Since M_{off}'s transition function is deterministic this means that M_{off} is in an infinite loop and the computation never stops.

Since M_{off} uses only $S(|x|)$ space there are only $\#conf(M,x)$ possible CWGs and therefore $\#conf(M,x)$ bounds the number of times the guess head may return to a specified cell. ■

Now we can (almost) bound the size of the guess.

Claim 5.2.4 *If for input x there exists a guess y s.t. the machine M_{off} stops on x with guess y, then there exists such a guess y satisfying $|y| < |\Gamma| \cdot \#conf(M,x)^{\#conf(M,x)} = 2^{2^{O(S(|x|))}}$.*

Proof: Denote the guess tape cells $c_0 c_1 \dots c_{|y|}$ and their content $y = g_o \dots g_{|y|}$. Given a computation of M_{off} and a specified cell c_i the guess head may have visited c_i several times during the computation, each time M_{off} was in another CWG. We can associate with every cell c_i the sequence of CWGs M_{off} was in when it visited c_i, denote such a sequence by *visiting sequence of* c_i. (Thus the first CWG in the visiting sequence of c_i is the CWG M_{off} was in the first time the guess head visited c_i, the second CWG in the visiting sequence is the CWG M_{off} was in the second time the guess head visited c_i and so on). By the last claim we get that the length of a visiting sequence is between 0 and $\#conf(M,x)$.

Suppose that for $k < l$, c_k and c_l both have the same visiting sequence and the same content i.e. $g_k = g_l$. Then the guess $g_0, \dots, g_k, g_{l+1}, \dots, g_{|y|}$ is also a guess that will cause M_{off} to accept input x. The idea is the same as we saw in the proof of Claim 5.2.2, i.e. if there are two points in the computation in which the machine is in the exact same situation, then the part of the computation between these two points can be cut off and the result would still be a computation of the machine. To see that this is the case here, we need just follow the computation, when the machine first tries to move from cell c_k to cell c_{k+1} (denote this time t_1^k) it's CWG is the same CWG that describes the machine's state when first moving from cell c_l to c_{l+1} (denote this time t_1^l) therefore we can "skip" the part of the computation between t_1^k and t_1^l and just put the guess head on c_{l+1} and still have a "computation" (the reason for the quatation marks is that normal computations do not have guess head teleportations). By similar reasoning whenever the machine tries to move from c_{l+1} to c_l (or from c_k to c_{k+1}) we can just put the guess head on c_k (respectively c_{l+1}) and "cut off" the part of the computation between the time it moved from c_{l+1} to the correponding time it arrived at c_k (respectively c_k and c_{l+1}). If we would have done exactly that i.e. always "teleporting" the head and cutting the middle part of the computation, we would get a "computation" in which the guess head never entered the part of the guess tape between c_k and c_{l+1} so actually we would have a real computation (this time with out the quotation marks) on the guess $g_0 g_1 \dots g_k g_{l+1} g_{l+2} \dots g_{|y|}$.

Since we can iterate cut and paste process until we get a guess with no two cells with identical visiting sequences and content, we can assume the guess contains no two such cells.

There are $\#conf(M,x)$ possible CWGs therefore $\#conf(M,x)^n$ sequences of n CWGs. Each visiting sequence is a sequence of CWGs of length at most $\#conf(M,x)$ so over all there are $\sum_{i=1}^{\#conf(M,x)} \#conf(M,x)^i \leq \#conf(M,x) \cdot \#conf(M,x)^{\#conf(M,x)} = \#conf(M,x)^{\#conf(M,x)+1} = 2^{2^{O(S(|x|))}}$ possibilities for a visiting sequence. Multiplied by the $|\Gamma|$ possibilities for the guess itself at each guess tape cell, this bounds the length of our short guess. ■

We have succeeded in bounding the length of the guess and therefore the space needed to simulate M_{off} in an online machine using a straightforward approach. Unfortunately the bound is a double exponential bound and we want better. The good news is that during the analysis of the naive approach to the problem we have seen almost all that is necessary to prove Theorem 5.8.

Proof: *(Theorem 5.8: $NSPACE_{off}(S) \subseteq NSPACE_{on}(2^{O(S)})$.):*
Given an offline machine M_{off} we shall construct an online machine M_{on} that accepts the same language.

In the proof of the last claim (bounding the length of the guess) we saw another way to describe the computation. If we knew the guess, instead of a configuration sequence (with time as an index), one can look at a sequence of visiting sequences (with the guess tape cells as index). Therfore if

we add the contents of the guess cell to each visiting sequence, the sequence of the augumented visiting sequences would describe the computation.

Our online machine M_{on} will guess an M_{off} computation described in the visiting sequences form and check whether indeed the guess is an accepting computation of M_{off} (accept if so, reject otherwise). The strategy is very similar to what was done in the proof of Theorem 5.7 (where an offline machine guessed a computation of an online machine and verified it).

To follow this strategy we need to slightly augment the definition of a visiting sequence. Given a computation of M_{off} and a guess tape cell c_i denote by *directed visiting sequence (DVS) of* c_i :

- The content of the guess cell c_i
- The visiting sequence of c_i
- For every CWG in the visiting sequence, the direction from which the guess head arrived to the cell (either R, L or S standing for Right, Left or Stay)

We shall now try to characterize when a string of symbols represents an accepting computation in this representation.

A DVS has the *reasonable returning direction property* if : whenever according to a CWG and cell content the guess head should move right, then the direction associated with the next CWG (returning direction) is left. (respectively the returning direction from a left head movement is right, and from staying is stay).

An ordered pair of DVSs is called *locally consistent* if they appear as if they may be consecutive in a computation i.e. whenever according to the CWG and the guess symbol in one of the DVSs the guess head should move to the cell that the other DVS represents then the CWG in the other DVS that corresponds to the consecutive move of M_{off} is indeed the CWG M_{off} would be in according to the transition function. (The corresponding CWG is well defined because we can count how many times did the head leave the cell of the first DVS in the direction of the cell of other DVS and the corresponding CWG can be found by counting how many time sthe head arrived from that direction). In addition to that, both DVSs must be first entered from the left, and both must have the reasonable returning property.

What must be checked in order to verify a candidate string is indeed an encoded computation of M_{off} on input x ?

1. The CWG in the first DVS is describing an initial configuration of M_{off}.
2. Every two consecutive DVSs are locally consistent.
3. In some DVS the last CWG is describing an accepting configuration.
4. In the last (most right) DVS, there is no CWG that according to it and the symbol on the guess tape the guess head should move to the right.

M_{on} guesses a sequence of DVSs and checks the properties 1–4. To do that, M_{on} never has to hold more then two consecutive DVSs + $O(1)$ memory. Since by Claim 5.2.4 the space needed for a DVS is $log(2^{2^{O(S(|x|))}}) = 2^{O(S(|x|))}$, M_{on} works in space $2^{O(S(|x|))}$. ■

The online model is considered more natural for measuring space complexity (and is equivalent to the first formulation of a non-deterministic Turing machine), therefore it is considered the standard model. In the future when we say "non-deterministic space" we mean as measured in the online model. Thus, we shorthand $NSPACE_{on}$ by $NSPACE$. That is, for any function $S : \mathsf{N} \to \mathsf{N}$, we let $NSPACE(S) \stackrel{\text{def}}{=} NSPACE_{on}(S)$.

5.3 Relations between Deterministic and Non-Deterministic space

The main thing in this section is Savitch's Theorem asserting that non-deterministic space is at most quadratically stronger than deterministic space.

5.3.1 Savitch's Theorem

In this section we present the basic result regarding the relations between deterministic and non deterministic space complexity classes. It is easy to see that for any function $S : \mathsf{N} \to \mathsf{N}$, $DSPACE(S) \subseteq NSPACE(S)$ as deterministic machines are in particular degenerated non-deterministic machines. The question is how much can be "gained" by allowing non-determinism.

Theorem 5.9 (Savitch): *For every space constractable function $S(\cdot)$ which is at least logarithmic $NSPACE(S) \subseteq DSPACE(S^2)$.*

For any non-deterministic machine M_N that accepts L in space S, we will show a deterministic machine M that accepts L in space S^2.

Definition 5.10 (M's configuration graph over x) : *Given a machine M which works in space S and an input string x, M's* configuration graph over x, *G_M^x, is the directed graph in which the set of vertices is all the possible configurations of M (with input x) and there exists a directed edge from s_1 to s_2 iff it is possible for M, being in configuration s_1, to change to configuration s_2.*

Using this terminology, M is deterministic iff the out degree of all the vertices in G_M^x is one.

Since we can assume without loss of generality that M accepts only in one specific configuration (assume M clears the work tape and move the head to the initial position before accepting), denote that configuration by $accept_M$ and the initial configuration by $start_M$. The question whether there exists a computation of M that accepts x can now be phrased in the graph terminology as "is there a directed path from $start_M$ to $accept_M$ in G_M^x".

Another use of this terminology may be in formulating the argument we have repeatedly used during the previous discussions : if there exists a computation that accept x then there exists such a computation in which no configuration appears more than once. Phrased in the configuration graph terminology this reduces to the obvious statement that if there exists a path between two nodes in a graph then there exists a simple path between them. If M works in space $S(|x|)$ then the number of nodes in G_M^x is $|V_M^x| = \#conf(M, x)$ therefore if there exists a path from $start_M$ to $accept_M$ then there is one of length at most $|V_M^x|$.

We reduced the problem of whether M accepts x to a graph problem of the sort "is there a directed path in G from s to t which is at most l long ?". This kind of problem can be solved in $O(log(|l|) \cdot log(|G|))$ space. (The latter is true assuming that the graph is given in a way that enables the machine to find the vertices and the vertices neighbors in a space efficient way, this is the case in G_M^x).

Claim 5.3.1 *Given a graph $G = (V, E)$, two vertices $s, t \in V$ and a number l, in a way that solving the question of whether there exists an edge between two vertices can be done in $O(S)$ space, the question "is there a path of length at most l from s to t" can be answered in space $O(S \cdot log(l))$.*

Proof: If there is a path from s to t of length at most l either there is an edge from s to t or there is a vertex u s.t. there is a path from s to u of length at most $\lceil l/2 \rceil$ and a path from u to t

of length at most $\lfloor l/2 \rfloor$. It is easy to implement a recursive procedure PATH(a, b, l) to answer the question.

```
1  boolean PATH(a, b, l)
2    if there is an edge from a to b then return TRUE
3    (otherwise continue as follows :)
4      for every vertex v
5        if PATH(a, v, ⌈l/2⌉) and PATH(v, b, ⌊l/2⌋)
6          then return TRUE
7      otherwise return FALSE
```

How much space does PATH(a, b, l) use?

When we call PATH with parameter l it uses $O(S)$ space to store a, b and l, check whether there is an edge from s to t, and handle the for-loop control variable (i.e. v). In addition it invokes PATH twice with parameter $l/2$, but the key point is that both invocations use the same space (or in other words, the second invocations re-uses the space used by the first). Letting $W(l)$ denote the space used in invoking PATH with parameter l, we get the recursion $W(l) = O(S) + W(l/2)$, with end-condition $W(1) = O(S)$. The solution of this relation is $W(l) = O(S \cdot log(l))$.

(The solution is obvious because we add $O(S)$, $log(l)$ times (halving l at every iteration, it will take $log(l)$ iterations to get to 1). The solution is also easily verified by induction, denote by c_1 the constant from the $O(S)$ and $c_2 = 2c_1$, the induction step : $W(l) \leq c_1 S + c_2 S \cdot log(l/2) = c_1 S + c_2 S log(l) - c_2 S = c_2 S log(l) + (c_1 - c_2)S$ and for $c_2 > c_1$ we get $W(L) \leq c_2 S log(l/2))$. ■

Now the proof of Savitch's theorem is trivial.

Proof: *(Theorem 5.9 (Savitch's theorem): $NSPACE(S) \subseteq DSPACE(S^2)$) :*
The idea is to apply Claim 5.3.1 by asking "is there a path from $start_M$ to $accept_M$ in G_M^x ?" (we saw that this is equivalent to "does M accept x"). It may seem that we cannot apply Claim 5.3.1 in this case since G_M^x is not given explicitly as an input, however since the deterministic machine M get x as the input, it can build G_M^x so G_M^x is given implicitly. Our troubles are not over since storing all G_M^x is too space consuming, but there is no need for that, our deterministic machine can build G_M^x on the fly i.e. build and keep in memory only the parts it needs for the operation it performs now then reuse the space to hold other parts of the graph that may be needed for the next operations. This can be done since the vertices of G_M^x are configurations of M_N and there is an edge from v to u iff it is possible for M_N being in configuration v to change for configuration u, and that can easily be checked by looking at the transition function of M_N. Therefore If M works in $O(S)$ space then in G_M^x we need $O(S)$ space to store a vertex (i.e. a configuration), and $log(O(S))$ space to check if there is an edge between two stored vertices, all that is left is to apply the Claim 5.3.1. ■

5.3.2 A translation lemma

Definition 5.11 (NL) *The complexity class* Non-Deterministic logarithmic space, *denoted $\mathcal{NL}$, is defined as $NSPACE(O(log(n)))$.*

Sometimes Savitch's theorem can be found phrased as:

$$\mathcal{NL} \subseteq DSPACE(log(n)^2).$$

This looks like a special case of the theorem as we phrased it, but is actually equivalent to it. What we miss in order to see the full equivalence is a proof that containment of complexity classes "translates upwards".

Lemma 5.3.2 (Translation lemma): *Given S_1, S_2, f space constractable functions s.t. $S_2(f)$ is also space constractible and $S_2(n) \geq log(n), f(n) \geq n$ then if $NSPACE(S_1(n)) \subseteq DSPACE(S_2(n))$ then $NSPACE(S_1(f(n))) \subseteq DSPACE(S_2(f(n)))$.*

Using the translation lemma, it is easy to derive the general Savitch's therorem from the restricted case of NL: Given that $\mathcal{NL} \subseteq DSPACE(log(n)^2)$, given a function $S(\cdot)$ choose $S_1(\cdot) = log(\cdot), S_2(\cdot) = log(\cdot)^2$ and $f(\cdot) = 2^{S(\cdot)}$ (f would be constractible if S was) now, applying the translation lemma, we get that $NSPACE(log(2^S)) \subseteq DSPACE(log(2^S)^2)$ which is equivalent to $NSPACE(S) \subseteq DSPACE(S^2)$.

Proof: Given $L \in NSPACE(S_1(f(n)))$ we must prove the existence of a machine M that works in space $S_2(f(n))$ and accepts L.

The idea is simple, transform our language L of non-deterministic space complexity $S_1(f)$ to a language L^{pad} of non-deterministic space complexity S_1 by enlarging the input, this can be done by padding. Now we know that L^{pad} is also of deterministic space complexity S_2. Since the words of L^{pad} are only the words of L padded, we can think of a machine that given an input pads it and then checks if it is in L^{pad}. The rest of the proof is just carrying out this program carefully while checking that we do not step out of the space bounds for any input.

There exists M_1 which works in space $S_1(f(n))$ and accepts L. Denote by L^{pad} the language $L^{pad} \stackrel{\text{def}}{=} \{x\$^i | x \in L$ and M_1 accpets x in $S_1(|x| + i)$ space.$\}$ where \$ is a new symbol.

We claim now that L^{pad} is of non-deterministic space complexity S_1. To check whether a candidate string s is in L^{pad} we have to check that it is of form $x\$^j$ for some j (that can be done using $O(1)$ space). If so (i.e. $s = x\$^j$), we have to check that M_1 accepts x in $S_1(f(|x|+j))$ space and do that without stepping out of the S_1 space bound on the original input (i.e. $S_1(|s|) = S_1(|x|+j)$). This can be done easily by simulating M_1 on x while checking that M_1 does not step over the space bound (the space bound $S_1(|x|+j)$ can be calculated since S_1 is space constructable). (The resulting machine is referred to as M_2.)

Since L^{pad} is in $NSPACE(S_1)$ it is also in $DSPACE(S_2)$; i.e., there exists a deterministic machine M_3 that recognizes L^{pad} in S_2 space.

Given the deterministic machine M_3 we will construct a deterministic machine M_4 that accepts the original L in space $S_2(f)$ in the following way:

On input x, we simulate M_3 on $x\$^j$ for $j = 1, 2, \ldots$ as long as our space permits (i.e., using space at most $S_2(f(|x|))$, including all our overheads). This can be done as follows: If the head of M_3 is within x, M_4's input head will be on the corresponding point in the input tape, whenever the head of M_3 leaves the x part of the input, M_4 keeps a counter of M_3's input head position (and supplies the simualted M_3 with either \$ or black as appropriate). Recall that we also keep track that M_3 does not use more that $S_2(f(|x|))$ (for that reason we need $S_2(f)$ to be constractible), and if M_3 tries to step out of this bound we will treat it as if M_3 rejected. If during our simulations M_3 accept so does M_4 otherwise M_4 rejects.

Basicly M_4 is trying to find a right j that will cause M_3 to accept, if x is not in L then neither is $x\$^j$ in L^{pad} (for any j) and therefore M_3 will not accept any such string untill M_4 will eventually reject x (which will happen when j is sufficiently large so that $\log j$ superseeds $S_2(f(|x|))$ which is our own space bound). If on the other hand x is in L than M_1 accepts it in $S_1(f(|x|))$ space therefore M_3 accepts $x\$^j$ for some $j \leq f(|x|) - |x|$ (since to hold $f(|x|) - |x|$ one needs only a

counter of size $log(f(|x|)$ and S_2 is bigger then log this counter can be kept within the space bound of $S_2(f(|x|))$ and M_4 will get to try the right $x\$^i$ and will eventually accept x). ■

Remark: In the last proof thre was no essential use of the model deterministic or non-deterministic, so by similar argument we can prove analogous results (for example, $DSPACE(S_1) \subseteq DSPACE(S_2)$ implies $DSPACE(S_1(f)) \subseteq DSPACE(S_2(f))$).

By a similar argument we may also prove analogous results regarding time complexity classes. In this case we cannot use our method of searching for the correct padding since this method (while being space efficient) is time consuming. On the other hand, under suitable hypothesis, we can can compute f directly and so do not need to search for the righ padding. We define $L_2^{pad} = \{x\$^{f(|x|)-|x|} : x \in L\}$ and now M_4 can compute $f(|x|)$ and run M_3 on $x\$^{f(|x|)-|x|}$ in one try. There are two minor modifications that have to be done. Firstly, we assume all the functions involved $S_1, S_2, f \geq n$ (this is a reasonable assumption when dealing with time-complexity classes). Secondly, M_2 has to check whether the input $x\$^j$ is indeed $x\$^{f(|x|)-|x|}$; this is easy if it can compute $f(|x|)$ within it's time bounds (i.e., $S_1(|x\$^j|)$), but may not be the case if the input $x\$^j$ is much shorter than $f(|x|)$. To solve that, M_2 only has to time itself while computing $f(|x|)$ and if it fails to compute $f(|x|)$ within the time bound it rejects.

Bibliographic Notes

Oded's Note: *To be done – find the references for the relationship between the two definitions of non-deterministic space.*

Savitch's Theorem is due to [1]; its proof can be found in any standard textbook (e.g., see textbooks referred to in the previous lecture).

1. W.J. Savitch, "Relationships between nondeterministic and deterministic tape complexities", *JCSS*, Vol. 4 (2), pages 177-192, 1970.

Lecture 6

Inside Non-Deterministic Logarithmic Space

Notes taken by Amiel Ferman and Noam Sadot

Summary: We start by considering space complexity of (decision and search) problems solved by using oracles with known space complexities. Then we study the complexity class $\mathcal{NL}$ (the set of languages decidable within Non-Deterministic Logarithmic Space); We show a problem which is complete for $\mathcal{NL}$, namely the Connectivity problem (Deciding for a given directed graph $G = (V, E)$ and two vertices $u, v \in V$ whether there is a directed path from u to v). Then we prove the somewhat surprising result: $\mathcal{NL} = \text{co}\mathcal{NL}$ (i.e., $\mathcal{NL}$ class is closed under complementation).

6.1 The composition lemma

The following lemma was used implicitly in the proof of Savitch's Theorem:

Lemma 6.1.1 (composition lemma – decision version): *Suppose that machine M solves problem Π while using space $s(\cdot)$ and having oracle access to* decision tasks $\Pi_1, ..., \Pi_t$. *Further suppose that for every i, the task Π_i can be solved within space $s_i(\cdot)$. Then, Π can be solved within space $s'(\cdot)$, where $s'(n) \stackrel{\text{def}}{=} s(n) + \max_i\{s_i(\exp(s(n)))\}$.*

Proof: Let us fix a certain input x of length n for the machine M. First, it is clear from the definition of M that a space of length at most $s(n)$ is used on M's work-tape for the computation to be carried out. Next, we must consider all possible invocations of the decision tasks which M has oracle access to. Let M_i be (a determinstic) Turing Machine, computing decision task Π_i. Since at each step of its computation, M may query some oracle M_i, it is clear that the contents of each such query depends on the different configurations that M went through until it reached the configuration in which it invoked the oracle. In this sense, the input to M_i is a query that M "decided on". We may deduce that an input to an oracle is bounded by the size of the set of all configurations of machine M on the (fixed) input x (this is the maximal length of a such a query). Let us bound the maximal size of such a query: It is the number of all different configurations of M on input of size n: $|\Sigma_M|^{s(n)} \times s(n) \times n$, where we multiply the number of all possible contents of the work-tape (whose length is bounded by $s(n)$) with the number of possible positions (of the

head) on the work-tape and with the number of possible positions on the input-tape (whose length is n) respectively (Σ_M is the work alphabet defined for the machine M).

Since the number of configurations of the machine M on input of length n is $exp(s(n))$, it is clear that the simulation of M_i would require no more than $s_i(exp(s(n)))$. Since we do not need to store the contents of the work-tape after each such simulation, but rather invoke each M_i whenever we need it and erase all contents of the work-tape related to that simulation, it is clear that in addition to the space $s(n)$ of work-tape mentioned above, we need to consider the maximum space that a certain M_i would need during its simulation, hence the result. ■

We stress that the above lemma refers to decision problems, where the output is a single bit. Thus, in the simulation of the M_i's the issue of storing parts of the output of M_i does not arise. Things are different if we compose search problems. (Recall that above and below we refer to deterministic space-bounded computations)

Lemma 6.1.2 (composition lemma – search version): *Suppose that machine M solves problem Π while using space $s(\cdot)$ and having oracle access to* search tasks $\Pi_1, ..., \Pi_t$. (As we shall see, it does not matter if machine M has a one-way or two-way access to the oracle-reply tape.) *Further suppose that all queries of machine M have length bounded by exp(s(n)) and that the answers are also so bounded.*[1] *Further suppose that for every i, the task Π_i can be solved within space $s_i(\cdot)$. Then, Π can be solved within space $s'(\cdot)$, where $s'(n) \stackrel{\text{def}}{=} s(n) + \max_i\{s_i(\exp(s(n)))\}$.*

The emulation of the oracles here is more complex than before since these oracles may return strings rather than single bits. Furthermore, the replies to different oracles Π_i's may be read concurrently. In the emulation we cannot afford to run M_i on the required query and store the answer, since storing the answer would use too much space. In order to avoid this, every time we need any bit in the answer to $\Pi_i(q)$, (where q is a query) we need to run M_i again on q and fetch the required bit from the on-line generated output, scanning (and omitting) all other bits; i.e., the answer that would be received from the oracle would be written on the output one bit at a time and by using a counter, the machine could tell when did it reach the desired bit of the answer (this process would halt since the length of the answer is bounded). Note that this procedure is applicable regardless if M has one-way or two-way access to the oracle-reply tape. Note that unlike in Lemma 6.1.1, here we cannot bound the length of the query by the number of possible configurations since this number is too large (as it includes the number of possible oracle answers). Instead, we use the hypothesis in the lemma.

Anallagous, but much simpler, result holds for the time complexity:

Lemma 6.1.3 (composition lemma – time version): *Suppose that machine M solves problem Π while using time $t(\cdot)$ and having oracle access to* decision tasks $\Pi_1, ..., \Pi_k$. *Further suppose that for every i, the task Π_i can be solved within time $t_i(\cdot)$. Then, Π can be solved within time $t'(\cdot)$, where $t'(n) \stackrel{\text{def}}{=} t(n) \times \max_i\{t_i(t(n))\}$.*

Proof: Similarly to the proof regarding Space, we shall fix a certain input x of length n for the machine M. First, it is clear from the definition of M that time $t(n)$ is suffices for the computation of M on x to be carried out. Next, we must consider all possible invocations of the decision tasks which M has oracle access to. Here at each step of the computation M could invoke an oracle M_i and so it is clear that the time complexity of the computation would be $t(n)$ multiplied by the

[1] Without this assumption, we cannot bound the number of configurations of machine M on a fixed input, as the configuration depends on the location of M's head on the oracle-reply tape.

maximal time complexity of an oracle. In order to find the time complexity of some oracle M_i, we have to consider the possible length of a query to M_i; since there are $t(n)$ time units during the computation of M on x, the size of the query to M_i could be at most $t(n)$.

We deduce that the time complexity of some oracle M_i which is invoked at some point of the computation of M on x, could be at most $t_i(t(n))$. According to what was said above, the time complexity of M on x would be the number of time units of its computation - $t(n)$ - multiplied by the maximal time complexity of some oracle M_i $(1 \leq i \leq k)$, hence the result. ■

6.2 A complete problem for $\mathcal{NL}$

The complexity class $\mathcal{NL}$ is defined to be simply $NSPACE(O(log(n)))$. More formally we have:

Definition 6.1 *$\mathcal{NL}$: A language L belongs to $\mathcal{NL}$ if there is a nondeterminstic Turing machine M that accepts L and a function $f(n) = O(log(n))$ such that for every input x and for every computation of M at most $f(|x|)$ different work-tape cells are used.*

Our goal in this section and the following one would be to study some properties of the class $\mathcal{NL}$. To that end we define the following:

Definition 6.2 *A log-space reduction of L_1 to L_2 is a log-space computable function f such that* $\forall x,\ x \in L_1 \ \Leftrightarrow\ f(x) \in L_2$

Note that a log-space reduction is analagous to a Karp-reduction (where space corresponds to time and the logarithmic number of cells correspond to polynomial number of steps). Actually, since each function that can be computed in space $s(\cdot)$, can also be computed in time $exp(s(\cdot))$, we have that a log-space reduction is a special case of a polynomial time reduction. The next definition would define a notion analagous to $\mathcal{NP}$-completeness (as we will see, this would prove useful in proving a proposition about $\mathcal{NL}$ which is analagous to a proposition in $\mathcal{NP}$):

Definition 6.3 *L is $\mathcal{NL}$-Complete if:*

(1) *$L \in \mathcal{NL}$; and*

(2) *$\forall L' \in \mathcal{NL}$, L' is log-space reducible to L.*

6.2.1 Discussion of Reducibility

As implied from the definitions above, our goal would be to find a problem which is complete for the class $\mathcal{NL}$. Prior to that, we must make sure that the concept of completness is indeed meaningful for the class $\mathcal{NL}$. The following propositions ensure exactly that.

Proposition 6.2.1 *If L is log-space reducible to L' and L' is solvable in log-space then L is solvable in log-space.*

Proof: Since L' is solvable in logarithmic space, there exists a machine M' which decides L' using logarithmic space. Furthermore, since L is log-space reducible to L', there exists a function $f(\cdot)$ computable in log-space, such that $x \in L \ \Leftrightarrow\ f(x) \in L'$ and so there exists a machine M such that for every input x would first compute $f(x)$ and then would simulate M' in order to decide on $f(x)$, both actions demanding log-space (as $lg(|f(x)|) \leq lg(exp(lg(|x|))) = O(lg(|x|))$) and ensuring that M would accept x iff $x \in L$. ■

Interestingly, such reduction also preserve non-deterministic space:

Proposition 6.2.2 *If L is log-space reducible to L' and $L' \in \mathcal{NL}$ than $L \in \mathcal{NL}$*

Instead of proving the last proposition, we will prove a related proposition regarding Non-Deterministic Time:

Proposition 6.2.3 *If L if Karp-Reducible to L' and $L' \in \mathcal{NP}$ then $L \in \mathcal{NP}$*

Proof: Since L is Karp-Reducible to L', there is a many-to-one function $f(\cdot)$, computable in polynomial time, such that: $x \in L \Leftrightarrow f(x) \in L'$. Furthermore, since L' is in $\mathcal{NP}$, there is a Non-Deterministic Turing machine M' that can guess a witness y for an input z (the length of y is a polynomial in the size of z) in polynomial time such that $R_{L'}(z, y)$ holds (where $R_{L'}$ is the relation that defines the language L' in $\mathcal{NP}$). We will construct a Non-Determinstic machine M for deciding L in the following way: For a given input $x \in L$, M will compute $f(x)$ (deterministically in polynomial time) and then would just simulate M' (mentioned above) on input $f(x)$ to find a witness y (non-deterministically in polynomial time) such that $R_{L'}(f(x), y)$ would hold. Thus, M defines a relation R_L such that for every input $x \in L$, it guesses a witness y (non-determinstically in polynomial time) such that $R_L(x, y)$ holds (i.e., $R_L(x, y) = R_{L'}(f(x), y)$). So by definition, L is in $\mathcal{NP}$. ■

We can use the proof of Proposition 6.2.3 to prove Proposition 6.2.2: Instead of a function $f(\cdot)$ computable in polynomial time, we are guarnteed to have a function $f(\cdot)$ which is computed in logarithmic space. Furthermore we may presume the existence of a machine M' deciding the language L' in logarithmic space (instead of non-determinstic polynomial time). It is now clear, that one may construct a non-deterministic machine M which may decide the language L in logarithmic space (which is analagous to the machine M which decided L in non-deterministic polynomial time).

Note that requiring the existence of a Cook-Reduction instead of a Karp-Reduction in Proposition 6.2.3 would probably make this proposition false: This stems from the fact that if a language L is Cook-Reducible to a language $L' \in \mathcal{NP}$ it does not necessarily mean that $L \in \mathcal{NP}$. In particular, any co$\mathcal{NP}$ language is Cook-Reducible to its complement. Still, if $\mathcal{NP} \neq$ co$\mathcal{NP}$ we have that $\overline{SAT} \notin \mathcal{NP}$ (and yet $\overline{SAT}$ is reducible to SAT). We conclude that if $\mathcal{NP} \neq$ co$\mathcal{NP}$ then Cook reductions are strictly more powerful than Karp reductions (since the class of languages which are Cook-reducible to $\mathcal{NP}$ contains co$\mathcal{NP}$, whereas the languages which are Karp-reducible to $\mathcal{NP}$ are exactly $\mathcal{NP}$). A more trivial example of this difference in power is the fact that any language in $\mathcal{P}$ is Cook-reducible to the empty set, whereas only the empty set is Karp-reducible to the empty set.

However, in the next proposition, as well as in Proposition 6.2.1, a Cook-reduction would do:

Proposition 6.2.4 *If L is polynomial-time reducible to L' and $L' \in P$ then $L \in P$*

In this last proposition, if L would be Cook-Reducible to L' then it is clear that the machine that emulates the oracle machine and answeres the queries by simulating the macine that decides L' (and runs in polynomial time), would be a polynomial-time machine that decides L (here the use of the oracle on L' and its actual simulation didn't make a difference in the running time). An analouge argument applies to Proposition 6.2.1. That is, if there exists a log-space oracle machine which deecides L by making polynomially-bounded queries to L', and L' is solvable in log-space then so is L (actually this follows from Lemma 6.1.1).

6.2.2 The complete problem: directed-graph connectivity

The problem that we will study would be graph connectivity (denoted as CONN) which we define next:

Definition 6.4 (directed connectivity – CONN): *$CONN$ is the is defined as a set of triples, (G, v, u), where $G = (V, E)$ is a directed graph, $v, u \in V$ are two vertices in the graph so that there is is a directed path from v to u in G*

As we shall see, the problem CONN is a natural problem to study in the context of space complexity. Intuitively, a computation of a Turing machine (deterministic or not) on some fixed input, could allways be pictured as a graph with nodes realting to the machine configurations and edges relating to transitions between configurations. Thus the question of whether there exists a certain accepting computation in the machine reduces to the question of the existence of a certain directed path in a graph: that path which connects the node which corresponds to the initial configuration and the node which corresponds to an accepting configuration (on a certain input). We note that in a deterministic machine the out-degree of each node in the graph would be exactly one, while in a non-determinstic machine the out-degree of each node could be any non-negative number (because of the possibilty of the non-determinstic machine to move to any one of a certain configurations), however in both cases the out-degree of each node is constant (depending only on the machine). Continuing this line of thought, it's not hard to see that CONN could be proved to be complete for the class $\mathcal{NL}$, i.e. it is itself in $\mathcal{NL}$ and every machine in $\mathcal{NL}$ could be reduced to it. The details are proved in the following:

Theorem 6.5 *$CONN$ is $\mathcal{NL}$-Complete.*

> Oded's Note: *The following proof is far too detailed to my taste. The basic ideas are very simple. Firstly, it is easy to design a non-deterministic log-space machine which accepts $CONN$ by just guessing an adequate directed path. Secondly, it is easy to reduce any language $L \in \mathcal{NL}$ to $CONN$ by just considering the directed graph of configurations of a log-space machine (accepting L) on the given input, denoted x. Each such configuration consists of a location on the input-tape, a location of the work-tape, the contents of the work-tape and the state of the machine. A directed edge leads from one configurations to another iff they are possible consequetive configuration of a computation on input x. The key point is that the edge relation can be determined easily by examining the two configurations and the relevant bit of x (pointed to in the first configuration).*

Proof: First we show that $CONN \in \mathcal{NL}$ (see Definition 6.3). We will build a machine M that would decide for the input $G = (V, E)$ and $v, u \in V$ whether there exists a path in G from v to u. Of course, M would do so non-determinstically in $O(log(n))$ space where n is the size of the input. The outline of the algorithm is as follows: We start with the node v (given at the input) and a counter which is initialized to the number of nodes in G. At each step we decrement the counter and guess a node which is adjacent to the current node we have (initially v). If the node we have guessed is not adjacent to the node we hold, we just reject. No harm is done since this is a non-deterministic computation: it suffices that there will be some computation that would accept. This procedure concludes when either the counter reaches 0 (the path shold not be longer than the number of nodes) or the last node we have guessed is u (the other node specified in the input). The actual guessing of a node could be done in several ways, one would be to implement a small procedure that would non-deterministically write on the work tape symbols that would

encode some node (by scanning the list of edges or adjacency matrix which are part of the input). Then, it will just check wheather the node it had written is adjacent to the current node we hold. Correctness and complexity analysis follow the formal specification of the algorithm:

Input: $G = (V, E)$, $v, u \in V$

Task: Find whether there exists a directed path from v to u in G.

1. $x \leftarrow v$
2. counter $\leftarrow |V|$
3. repeat
4. decrement counter by 1
5. guess a node $y \in V$ s.t. $(x, y) \in E$
6. if $y \neq u$ then $x \leftarrow y$
7. until $y = u$ or counter $= 0$
8. if $y = u$ then accept, else reject

First we will prove the correctness of this algorithm: On the one hand, suppose that the algorithm accepts the input $G = (V, E), v, u \in V$. This implies that during the repeat-until loop, the algorithm has guessed a sequence of nodes such that each one of them had a directed edge to its successor and that the final node in this sequence is u and the initial node is v (from the first line and the check that is made in the last line). Clearly, this implies a directed path from v to u in G (Note that from the existence of the counter, the number of steps in this computation is bounded by $O(n)$). On the other hand, suppose that there is a directed path in G from v to u. This path is a sequence $\{v, x_1, \ldots, x_k, u\}$ where $k \leq (n-2)$ and there is a directed edge from each node in this sequence to its successor. In this case it is clear that the computation of the algorithm above in which it guesses each one of the nodes in the sequence starting from v (from the first line of the algorithm) and ending with u (from the last line of the computation) is an accepting computation, and thus the algorithm would accept the input $G = (V, E), v, u \in V$.

We conclude that there is an accepting computation of the algorithm above on the input $G = (V, E), v, u \in V$ iff there is a directed path in G from v to u.

All that is left to be shown is that the implementation of such an algorithm in a non-determinstic machine would require no more than a logarithmic space in the size of the input: First, it is clear that each one of the variables required to represent a node in the graph need not be represented in more than logarithmic number of cells in the size of the input (for example, in order to represent a number n in binary notation, we need no more than $lg(n)$ bits). The same argument applies to the counter which has to count a number which is bounded by the size of the input. Secondly, all other data besides the variables may be kept at a constant number of cells of the work-tape (for example a bit that would indicate whether $y = u$ etc.). As was specified above regarding the implementation of step 5 (the guessing of the node), the actuall guessing procedure, which would be done non-determinstically, uses number of cells which is equal exactly to the length required to the representation of a node (which is again logarithmic in the size of the input). We conclude that the implementation of the algorithm above on a non-determinstic machine M requires a logarithmic

space in the size of the input, and so we may conclude that the machine M decides CONN in non-deterministic logrithmic space, i.e., $CONN \in \mathcal{NL}$.

Now we need to show that every language $L \in \mathcal{NL}$ is log-space reducible to CONN. Let L be a language in $\mathcal{NL}$, then there is a non-deterministic logarithmic space machine M that decides L. We will show that for every input x, we can build in nondeterministic logarithmic space an input $(G = (V, E), start \in V, end \in V)$ (which is a function of machine M and input x) such that there is a path in G from *start* to *end* if and only if M accepts input x. The graph G we will construct would simply be the graph of all possible configurations of M given x as an input. That is, the nodes denote different configurations of M while computing on input x, and the arcs denote possible immediate transitions between configurations.

The graph is constructed (deterministic) log-space as follows;

Input: An input string x (the machine M is fixed)

Task: Output a graph $G = (V, E)$ and two nodes $v, u \in V$ such that there is a path from v to u in the graph iff x is accepted by M.

1. compute n, the number of different configurations of M while computing input x
2. for $i = 1$ to n
3. for $j = 1$ to n
4. if there is a transition (by a single step of M) from configuration number i to configuration number j ouput 1 otherwise output 0
5. output 1 and n

First we will show that this procedure indeed outputs the representation of a graph and two nodes in that graph such that there exists a directed path between those two nodes iff the input x is accepted by machine M. In the first line we compute the number of all possible configurations of machine M while computing on input x. Then, we consider every ordered pair of configurations (represented by numbers between 1 and n) and output 1 iff there is indeed a directed transition between those two configurations in the computation of M on x. Our underlying assumption is that 1 represents the initial configuration and n represents the (only) accepting configuration (if there were several accepting configurations we define a new one and draw edges from the previous accepting configurations to the new one). Thus, the output of the above procedure is simply an adjacency matrix of a graph in which each of its nodes correspond to a unique configuration of M while computing on input x , and a directed edge exists between two nodes i and j iff there is a (direct) transition in M between the configuration represented by x and the configuration represented by y. It is now clear that a directed path from the first node (according to our enumeration) to the last node in the graph would correspond to an accepting computation of machine M on input x, and that such a path would not exists should there be no such accepting computation.

Next, we must show that the above procedure could indeed be carried out using no more than a logarithmic space in the length of the input (i.e., the input x). In order to do that, we will show that the number of different configurations of M while computing x is polynomial in the length of x. This would imply that in order to count these configurations we need no more than a logarithmic space in the length of x. So, we will count the number of possible configurations

of M while computing on a given input x. That number would be the number of possible states (a constant determined by M) multiplied by the number of possible contents of the work-tape which is $|\Sigma_M|^{O(log(n))}$ where Σ_M is the alphabet of M and is also a constant determined by M (let us remember that since M is log-space bounded, the number of used work-tape squares could not surpass $O(log(n))$), multiplied by the number of different positions of the reading head on the input tape which is n and finally, mutiplied by the number of different possible positions of the reading head on the work-tape which is $O(log(n))$. All in all, the number of different configurations of M while computing input x is: $|States_M| \times |\Sigma_M|^{O(log(n))} \times n \times O(log(n)) = O(n^k)$ (where k is a constant). That is, the number of different configurations of M while computing input x is polynomial in the length of x.

We may conclude now that the initial action in the procedure above, that of counting the number of different configurations of M while computing on input x could be carried out in logarithmic space in the length of x.

Secondly, we show that the procedure of checking whether there exists a (direct) transition between two configurations represented by two integers can be implemented as well in logarithmic space: We show that there is a machine M'' that receives as inputs two integers and would return a positive answer if and only if there is a (direct) transition of M between the two configurations which are represented by those integers. Note first that integers correspond to strings over some convinient alphabet and that such strings correspond to configurations. (The correspondance is by trivial computations.) Thus, all we need to determine is whether the fixed machine M can pass in one (non-deterministic) step, on input x, between a given pair of configurations. This depends only on the transition function of M, which M'' has hard-wired, and on a single bit in the input x; that is, the bit the location of which is indicated in the first of the two given configurations. That is, suppose that the first configuration is of the form (i, j, w, s), where i is a location on the input-tape, j a location on the work-tape, w the contents of the work-tape and s the machine's state. Same for the second configuration, denoted (i', j', w', s'). Then we check if M when reading symbol x_i (the i^{th} bit of x) from its input tape, and w_j (the j^{th} symbol of w) from its work-tape can make a single transition resulting in the configuration (i', j', w', s'). In particular, it must hold that $i' \in \{i-1, i, i+1\}$, $j' \in \{j-1, j, j+1\}$, and w' differs from w at most on location j. Furthermore, these small changes must depend on the transition function of M. Since there is a constant number of possible transitions (in M's transition function), we may just check all of them.

We have shown that the above procedure outputs a representation of a graph and two nodes in that graph for a machine M and input x, such that there is a directed path between the nodes iff there is an accepting computation of M on x. Furthermore we have shown that this procedure may be implemented requiring no more than a log-space in the size of the input (the input string x) which concludes the proof. ∎

6.3 Complements of complexity classes

Definition 6.6 (complement of a language): *Let $L \subseteq \{0,1\}^*$ be a language. The* complement of a language *L, denoted $\overline{L}$ is the language $\{0,1\}^* \backslash L$.*

To make this definition more accurate, we assume that every word in $\{0,1\}^*$ represents an instance of the problem.

Example 6.7 *: $\overline{CONN}$ is the following set: $\{(G, u, v) : G$ is a directed graph, $u, v \in V(G)$, there is no directed path from u to v in $G\}$.*

Definition 6.8 *(*complement of class*): Let $\mathcal{C}$ be a complexity class. The* complement of the class $\mathcal{C}$ *is denoted* co$\mathcal{C}$ *and is defined to be* $\{\overline{L} : L \in \mathcal{C}\}$

It is immediately obvious that if $\mathcal{C}$ is a deterministic time or space complexity class, then co$\mathcal{C} = \mathcal{C}$, in particular, $\mathcal{P} = \text{co}\mathcal{P}$. This is true, since we can change the result of the derministic machine from 'yes' to 'no' and vice versa.

However, in non-deterministic complexity classes, this method does not work. Let M be a Turing Machine that accepts a language L non-deterministicly. If $x \in L$, then there is at least one successful computation of M on x (i.e., there is a succinct verification that $x \in L$). We denote by $\overline{M}$ the non-deterministic Turing Machine that does the same as M, but replaces the output from "yes" to "no" and vice versa. Hence, if the new machine $\overline{M}$ accepts an input z, there is one accepting computation for $\overline{M}$ on z, i.e. non-accepting computation in M (by definition). In other words, $\overline{M}$ will accept z, if M has some unsuccessful guesses to prove that $z \in L$. This, however, does not mean that $\overline{M}$ accepts $\overline{L}$, since z could possibly be in L by other guesses of the machine M. For example we don't know whether co$\mathcal{NP}$ is equal to $\mathcal{NP}$. The conjecture is that $\mathcal{NP} \neq \text{co}\mathcal{NP}$

Yet, in the particular case of nondeterministic space equality does hold. It can be proven that any non-deterministic space $NSPACE(s(n))$ for $s(n) \geq log(n)$ is closed under complementation. This result which was proven by Neil Immerman in 1988, is going to be proven here for the case $\mathcal{NL}$. By the following proposition, it suffices to show that $CONN \in \text{co}\mathcal{NL}$ (or equivalently $\overline{CONN} \in \mathcal{NL}$.

Proposition 6.3.1 *: If for an $\mathcal{NL}$-complete language L it holds that $L \in \text{co}\mathcal{NL}$ then $\mathcal{NL} = \text{co}\mathcal{NL}$.*

Proof: Let L' be a language in $\mathcal{NL}$. Since $L \in \mathcal{NL}$-complete, we have a log-space reduction f from L' to L (see Definition 6.2). The function f satisfies:

$$x \in L' \Leftrightarrow f(x) \in L$$

Taking the opposite direction, we get:

$$x \in \overline{L'} \Leftrightarrow f(x) \in \overline{L}$$

By the definition of the reduction, f is also a reduction from $\overline{L'}$ to $\overline{L}$. By proposition 6.2.2 we know that since $\overline{L} \in \mathcal{NL}$ (because by hypothesis $L \in \text{co}\mathcal{NL}$) then $\overline{L'} \in \mathcal{NL}$ (i.e. $L' \in \text{co}\mathcal{NL}$). We conclude that for every $L' \in \mathcal{NL}$, $\overline{L'} \in \mathcal{NL}$, thus $\mathcal{NL} = \text{co}\mathcal{NL}$. ■

6.4 Immerman Theorem: $\mathcal{NL} = \text{co}\mathcal{NL}$

In this section, we are going to prove a surprising theorem, which claims that non-deterministic log-space is closed under complementation. Due to the proof in Theorem 6.5 that $CONN \in \mathcal{NL}$-complete, and using proposition 6.3.1, we only need to prove that $\overline{CONN} \in \mathcal{NL}$, where $\overline{CONN}$ is the complementary problem of $CONN$ as defined in Example 6.7. Formally, The decision problem of the language $\overline{CONN}$, is obtained as the following:

Input: a directed graph $G = (V, E)$ and two nodes $u, v \in V(G)$.

Question: Is there no directed path from v to u ?

In order to show that $\overline{CONN} \in \mathcal{NL}$, we use the following theorem, which will be proven later:

Theorem 6.9 *Given a directed graph $G = (V, E)$ and a node $v \in V(G)$, the number of nodes reachable from v in G can be computed by a non-deterministic Turing Machine within log-space.*

A non-deterministic Turing Machine that computes a function f of the input, as in the theorem above, is defined as follows:

Definition 6.10 (non-deterministic computation of functions): *A non-deterministic Turing Machine M is said to compute a function f if on any input x, the following two conditions hold:*

1. *either M halts with the right answer $f(x)$, or M halts with output "failure"; and*
2. *at least one of the machine computations halts with the right answer.*

6.4.1 Theorem 6.9 implies $\mathcal{NL} = \text{co}\mathcal{NL}$

Lemma 6.4.1 *Assuming Theorem 6.9, $\overline{CONN} \in \mathcal{NL}$*

Assuming Theorem 6.9, we have a non-deterministic Turing Machine denoted CR ($CR \stackrel{\text{def}}{=}$ Count Reachable) that counts all the nodes that are reachable in a directed graph G from a single node v in non-deterministic log-space. The idea of the proof is that once we know to compute using CR the number of nodes reachable from v in G, we can also non-deterministically scan all the vertices reachable from v, using this value. This is done non-deterministically by guessing connected paths to each of the reachable nodes from v. Once the machine discovered all reachable nodes from v (i.e. the number of reachable nodes it found equals the output of CR) and the node u isn't reachable, it can decide that there is no connected path between v and u in G.

Proof: Let $x = (G, u, v)$ be an input for the problem $\overline{CONN}$. We fix (G, v) and give it as an input to the machine CR, which is the non-deterministic machine that enumerates all the nodes in the graph G reachable from the node v as was assumed before to work in non-deterministic log-space. In other words, we use CR, as a "black box".

We construct a non-deterministic machine here that uses the following simulation that solves this problem:

- Firstly, it simulates CR on the input (G, v). If the run fails, the machine rejects. Otherwise, we denote the answer by N.
- For each vertex w in the graph, it guesses whether w is reachable from v and if yes, it guesses non-deterministicly a directed path from v to w, by guessing at most $n-1$ vertices (we know by a simple combinatorial fact, that each two connected nodes in a grpah $G = (V, E)$ are connected within path of length less than or equal to $n-1$) and verifies that it is a valid path. For each correct path, it increments a counter.
- If $w = u$, and it founds a valid path then it rejects.
- If $counter \neq N$ then the machine rejects the input, otherwise the machine accepts the input. ($counter \neq N$ means that not all reachable vertices were found and verified, whereas $counter = N$ means that all were examined. If none of these equals u then we should indeed accept).

Formally, we have the following algorithm:

Input: $G = (V, E)$, $v, u \in V(G)$
Task: Find whether there is no connected path between u and v.

1. **Simulating CR. If CR fails, the algorithm rejects, else $N \leftarrow CR((G, v))$.**
2. $counter \leftarrow 0$
3. **for** $w = 1$ **to** n **do** *(w is a candidate reachable vertex)*
4. **guess if w is reachable from v. If not, proceed to next iteration of step 3.**
   (we continue in steps 5-17 only if we guessed that w is reachable from v)
5. $p \leftarrow 0$ *(counter for path length)*
6. $v_1 \leftarrow v$ *(v_1 is initially v)*
7. **repeat** *(guess and verify a path from v to w)*
8. $p \leftarrow p + 1$
9. **guess a node** v_2 *(v_1 and v_2 are the last and the current nodes)*
10. **if** $(v_1, v_2) \notin E$ **then reject**
11. **if** $v_2 \neq w$ **then** $v_1 \leftarrow v_2$
12. **until** $(v_2 = w)$ **or** $(p = n - 1)$
13. **if** $(v_2 = w)$ **then** *(counting all reachable $w \neq u$)*
14. **begin**
15. $counter \leftarrow counter + 1$
16. **if** $w = u$ **then reject**
17. **end**
18. **if** $N \neq counter$ **then reject, else accept.**

We know that CR works in $O(log(|G|))$. In each step of the simulation, our algorithm uses only 6 variables in addition of those of CR, namely the counters $counter, w, p$, the representations of the nodes v_2, v_1 and N. the counters, and N are bounded by the number of vertices, n. Every new change of one of this variables will be written again on the work tape by reusing space. Therefore, they can be implemented in $O(log(n))$ space. The nodes, clearly, are represented in $O(log(|G|))$ space. Thus, we use no more then $O(log(|G|))$ space in the work tape in this machine (where x is the input). The correctness is proved next:

To show correctness we need to show that it has a computation that accepts the input if and only if there is no direct path from v to u in G.

Consider first the case that the machine accepts. A necessary condition (for this event) is that $counter = N$ (line 18); that is, the number of vertices that were found to be reachable is exactly the correct one (i.e., N). This means that every possible vertex that is reachable from v was counted. But, if u was found to be one of them the machine should have rejected before (in line 16). Therefore, u cannot be reachable from v by a directed path.

Suppose, on the other hand, that there is no directed path between u and v in G. Then if all guesses made are correct then the machine will necessarily accept. Specifically, we look at a

computation in which (1) the machine correctly guesses (in line 4) for each vertex w whether it is reachable from v; (2) for each reachable vertex w it guesses well a directed path from v; and (3) machine CR did not fail (and thus N equals the number of vertices reachable from v). In this case $N = counter$, and since u is not connected to v the machine accepts.

Therefore, we proves the lemma. ■

Using this result (under the assumption that Theorem 6.9 is valid), we obtain $\mathcal{NL} = \text{co}\mathcal{NL}$.

Theorem 6.11 (Immerman 88'): $\mathcal{NL} = \text{co}\mathcal{NL}$

Proof: We proved in Theorem 6.5 that $CONN \in \mathcal{NL}$-complete. In Lemma 6.4.1, we proved that $\overline{CONN} \in NL$ (or $CONN \in \text{co}\mathcal{NL}$). Using Proposition 6.3.1, we get, that $\mathcal{NL} = \text{co}\mathcal{NL}$. ■

An extension of this theorem can show that for any $s(n) \geq log(n)$, $NSPACE(s(n)) = coNSPACE(s(n))$.

6.4.2 Proof of Theorem 6.9

To conclude this proof we are only left with the proof of Theorem 6.9, i.e. the existness of a machine CR, that computes the number of nodes reachable from a vertex v in a directed graph $G = (V, E)$.

We use the following notations for a fixed point v in a fixed directed graph $G = (V, E)$:

Definition 6.12 *R_j is the set of vertices which are reachable from v by a path of length less than or equal to j. In addition, N_j is defined to be the number of nodes in R_j, namely $|R_j|$.*

It can be seen that,

$$\{v\} = R_0 \subseteq R_1 \subseteq \cdots \subseteq R_{n-1} = R$$

where n denotes the number of nodes in G, and R denotes the set of vertices reachable from v.

There is a strong connection between R_j and R_{j-1} for $j \geq 1$, since any path of length j is a path of length $j-1$ with an additional edge. The following claim will be used later in the development of the machine CR:

Claim 6.4.2 *The following equation holds:*

$$R_j = \begin{cases} R_{j-1} \cup \{u : w \in R_{j-1}, (w,u) \in E(G)\} & \text{if } j \geq 1 \\ \{v\} & \text{if } j = 0 \end{cases}$$

Proof: For $j = 0$: Clear from definition.
For $j \geq 1$: $R_{j-1} \subseteq R_j$ by definition. $\{u : w \in R_{j-1}, (w,u) \in E(G)\}$ represents all the nodes which are adjacent to R_{j-1}, i.e. have length at most $j-1+1 = j$. This set is also contained in R_j. Thus, $R_{j-1} \cup \{u : w \in R_{j-1}, (w,u) \in E(G)\} \subseteq R_j$.

In the opposite direction, every node $u \in R_j$, which is not v, is reachable from v along a path with length less or equal to j. Thus, its predecessor in this path has length less or equal to $j-1$. Thus, $R_j \subseteq \{u : w \in R_{j-1}, (w,u) \in E(G)\} \cup \{v\} \subseteq R_{j-1} \cup \{u : w \in R_{j-1}, (w,u) \in E(G)\}$ (since $\{v\} \subseteq R_{j-1}$ for any $j \geq 1$).

Therefore, the claim follows. ■

Corollary 6.13 *For any $j \geq 1$, a node $w \in R_j$, if and only if there is a node $r \in R_{j-1}$ such that $r = w$ or $(r, w) \in E(G)$.* ■

We now construct a non-deterministic Turing Machine, CR, that counts the number of nodes in a directed graph G, reachable from a node in the graph v.

Our purpuse in this algorithm is to compute N_{n-1} where n is the number of nodes in G, to find the number of all reachable nodes from v. This recursive idea in Claim 6.4.2 is the main idea behind the following algorithm, which is build iteratively. In each stage, the algorithm computes N_j by using N_{j-1}. It has an initial value N_0, which we know to be $|\{v\}| = 1$. The itrations use the non-deterministic power of the machine.

The high-level description of the algorithm is as follows:

- For each j from 1 to $n-1$, it tries to calculate recursively N_j from N_{j-1}. This is done from N_0 to N_{n-1}, which is the desired output. Here is how N_j is computed.
  - For each node w in the graph,
    - For each node r in the graph, it guesses if $r \in R_{j-1}$ and if the answer is yes, it guesses a path with length less than or equal to $j-1$, from v to r. It verifies that the path is valid. If it is, it knows that r is a node in R_{j-1}. Otherwise, it rejects. It counts each node r, such that $r \in R_{j-1}$, by $counter_{j-1}$. The machine checks whether $w = r$ or $(r, w) \in E(G)$. If it is, (by using Corrolary 6.13), $w \in R_j$, and then it indicates by a flag, $flag_w$ that w is in R_j. ($flag_w$ is initially 0).
  - It counts the number of vertices r in R_{j-1} we found, and verifies that it is equal to N_{j-1}. Otherwise, it rejects. If the machine does not reject, we know that every node $r \in R_{j-1}$ was found. Therfore using Corrolary 6.13, the membership of w in R_j is decided properly (i.e. $flag_w$ has the right value).
  - At the end of this process it sums up the flags, $flag_w$'s into a counter, $counter_j$ (i.e. counts the number of nodes that were found to be in R_j).
- It stores the value of $counter_j$ for N_j to begin a new iteration (or to give the result in case we reach $j = n-1$).

We stress that all counters are implemented using the same space. That is, the only thing which passes from iteration $j-1$ to iteration j is N_{j-1}.

The detailed code follows:

Input: $G = (V, E)$, $v \in V(G)$

Task: Find the number of reachable nodes from v in G.

1. **Computing** $n = |V(G)|$
2. $N_0 \leftarrow 1 = |R_0|$
3. **for** $j = 1$ **to** $n-1$ **do**
4. $\quad counter_j \leftarrow 0$
5. $\quad$ **for** $w = 1$ **to** n **do** *(lines 5-24 compute N_j)*
6. $\quad\quad counter_{j-1} \leftarrow 0$ *(w is a potential member in R_j)*
7. $\quad\quad flag_w = 0$

8. **for** $r = 1$ **to** n **do** *(We try to enumerate R_{j-1} using N_{j-1})*

9. **guess if** $r \in R_{j-1}$**. If not, proceed to next iteration of step 8.**
(we continue in steps 10-21 only if we guessed that $r \in R_{j-1}$)

10. $v_1 \leftarrow v$ *(v_1 is initially v)*

11. $p \leftarrow 0$

12. **repeat** *(guess and verify a path from v to r, such that $r \in R_{j-1}$)*

13. $p \leftarrow p + 1$

14. **guess a node** v_2 *(v_1 and v_2 are the last and current nodes)*

15. **if** $(v_1, v_2) \notin E$ **then halt(failure)**

16. **if** $v_2 \neq r$ **then** $v_1 \leftarrow v_2$

17. **until** $(v_2 = r)$ **or** $(p = j - 1)$

18. **if** $v_2 \neq r$ **then halt(failure)**

19. $counter_{j-1} \leftarrow counter_{j-1} + 1$

20. **if** $(r = w)$ **or** $((r, w) \in E(G))$ **then** *(check that $w \in R_j$)*

21. $flag_w = 1$

22. **if** $counter_{j-1} \neq N_{j-1}$ **then halt(failure)**

23. $counter_j \leftarrow counter_j + flag_w$

24. $N_j = counter_j$

25. **output** N_{n-1}

Lemma 6.4.3 *The machine CR uses $O(log(n))$ space.*

Proof: Computing the number of nodes (line 1) can be made in at most $O(log(n))$, by simply counting the number of nodes with a counter on the input tape. In each other step of the running of the machine, it only needs to know at most ten variables, i.e. the counters for the 'for' loops: j, w, r, p, the value of N_{j-1} for the current j, the two counters for the size of the sets $counter_j, counter_{j-1}$, two nodes of the guessing part v_2, v_1, and the indicating flag $flag_w$.

Oded's Note: *The proof might be more convincing if the code was modified so that N_{PREV} is used instead of N_{j-1}, $counter_{CURR}$ instead of $counter_j$, and $counter_{PREV}$ instead of $counter_{j-1}$. In such a case, line 24 will prepare for the next iteration by setting $N_{PREV} \leftarrow counter_{CURR}$. Such a description will better emphasise the fact that we use only a constant number of variables, and that their storage space is re-used in the iterations.*

Whenever is needed to change information, like increasing a counter, or changing a variable, it reuses the space it needs for this goal. Every counter we use counts no more than the number of nodes in the graph, hence we can implement each one of them in $O(log(n))$ space. Each node is assumed to take $O(log(n))$ to store, i.e. its number in the list of nodes. And the $flag_w$ clearly takes only 1 bit. Therefore, to store these variables, it is enough to use $O(log(n))$ space.

Except for these variables, we don't use any additional space. All that is done is comparing nodes with the input tape, and checking whether two nodes are adjacent in the adjecancy matrix that represents the graph. These operations can be done only by scanning the input tape, and take no more then $O(log(n))$ space, for counters that scan the matrix.

Therefore, this non-deterministic Turing Machine uses only $O(log(n))$ or $O(log(|x|))$ where $x = (G, v)$ is the input to the machine. ■

Lemma 6.4.4 *If the machine CR outputs an integer, then it correctly gives the result of* N_{n-1}.

Proof: We'll prove it by induction on the iteration of computing N_j:
For $j = 0$: It is obviously correct.
If it computes correctly N_{j-1}, and it did not halt while computing N_j, then it computes correctly N_j as well: By the assumption of the induction we have a computation that computes N_{j-1}, that is stored correctly. All we have to prove is that $counter_j$ is incremented if and only if the current w is indeed in R_j (line 23), since then N_j will have correctly the number of nodes in R_j. Since the machine didn't failed till now, $counter_{j-1}$ has to be equal to N_{j-1} (line 22), by the assumption of the induction. This means that the machine indeed found all $r \in R_{j-1}$, since $counter_{j-1}$ is incremented for each node that is found to be in R_{j-1} (line 19). Therefore, using Corrolary 6.13, we know that the machine changes the flag, $flag_w$, of a node w if only if $w \in R_j$. And this flag is the value that is added to $counter_j$ (line 23). Therefore, the counter is incremented if and only if $w \in R_j$. ■

Corollary 6.14 *Machine CR satisfies Theorem 6.9.*

Proof: We have shown in Lemma 6.4.4 that if the machine doesn't fail, it gives the right result. It is left to prove that there exists a computation in which the machine doesn't fail.

The correct computation is done as follows. For each node $r \notin R_{j-1}$, the machine guesses well in line 9 that indeed $r \notin R_{j-1}$ and stops working on this node. For each node $r \in R_{j-1}$, the machine guesses in line 9 so, and in addition it guesses correctly the nodes that form the directed path from v to r in line 14. In this computation, the machine will not fail. In lines 15 and 18, there is no failure, since only $r \in R_{j-1}$ nodes get to these lines, and in these lines it guesses corectly the connected path from v. Therefore, in line 22, all nodes $r \in R_{j-1}$ were counted, since the machine guesses them corectly, and the machine will not halt either. Thus, the machine doesn't fail on the above computation.

Using Lemma 6.4.3, we know that the machine uses $O(log(n))$ space. Therefore, CR is a non-deterministic machine that satisfies Theorem 6.9. ■

Bibliographic Notes

The proofs of both theorems (i.e., NL-completeness of CONN and NL=coNL) can be found in [2]. The latter result was proved independently by Immerman [1] and Szelepcsenyi [3].

1. N. Immerman. Nondeterministic Space is Closed Under Complementation. *SIAM Jour. on Computing*, Vol. 17, pages 760–778, 1988.

2. M. Sipser. *Introduction to the Theory of Computation*, PWS Publishing Company, 1997.

3. R. Szelepcsenyi. A Method of Forced Enumeration for Nondeterministic Automata. *Acta Informatica*, Vol. 26, pages 279–284, 1988.

Lecture 7

Randomized Computations

Notes taken by Erez Waisbard and Gera Weiss

Summary: In this lecture we extend the notion of efficient computation by allowing algorithms (Turing machines) to toss coins. We study the classes of languages that arise from various natural definitions of acceptance by such machines. We will focus on polynomial running time machines of the following types:

1. One-sided error machines ($RP, coRP$).
2. Two-sided error machines (BPP).
3. Zero error machines (ZPP)

We will also consider probabilistic machines that uses logarithmic spaces (RL).

7.1 Probabilistic computations

The basic thought underlying our discussion is the association of efficient computation with probabilistic polynomial time Turing machines. We will consider efficient only algorithms that run in time that is no more than a fixed polynomial in the length of the input.
There are two ways to define randomized computation. One, that we will call *online* is to enter randomized steps, and the second that we will call *offline* is to use an additional randomizing input and evaluate the output on such random input.

In the fictitious model of non-deterministic machines, one accepting computation was enough to include an input in the language accepted by the machine. In the randomized model we will consider the probability of acceptance rather than just asking if the machine has an accepting computation.

> *Then he said, "May the Lord not be angry, but let me speak just once more. What if only ten can be found there?" He answered, "For the sake of ten, I will not destroy it."* [Genesis 18:32].

As God didn't agree to save Sedom for the sake of less then ten peoples, we will not consider an input to be in the accepted language unless it has a noticeable probability to be accepted.

> Oded's Note: *The above illustration is certainly not my initiative. Besides some reservations regarding this specific part of the bible (and more so the interpretations given*

to it during the centuries), I fear that 10 may not strick the reader as "many" but rather as closer to "existence". In fact, standard interpretations of this passage stress the minimalistic nature of the challenge – barely above unique existence...

The online approach: One way to look at a randomized computation is to allow the Turing machine to make random moves. Formally this can be modeled as letting the machine to choose randomly among the possible moves that arise from a nondeterministic transition table. If the transition table maps one ($< state >, < symbol >$) pair to two different ($< state >, < move >, < symbol >$) triples then the machine will choose each transition with equal probabilities.

Syntactically, the online probabilistic Turing machine will look the same as the nondeterministic machine. The difference is at the definition of the accepted language. The criterion of an input to be accepted by a regular nondeterministic machine is that the machine will have at least one accepting computation when it is invoked with this input. In the probabilistic case, we will consider the probability of acceptance. We would be interested in *how many* accepting computation the machine has (or rather what is the probability of such computation). We postulate that the machine choose every step with equal probability, and so get a probability space on possible computations. We look at a computation as a tree, where a node is a configuration and it's children are all the possible configurations that the machine can pass to in a single step. The tree is describing the possible computations of the machine when running on a given input. The output of a probabilistic Turing machine on an input x is not a string but a random variable. Without loss of generality we can consider only binary tree because if the machine has more than two possible steps, it is possible to build another machine that will simulate the given machine with two step transition table. This is possible even if the original machine had steps with probability that has infinite binary expansion. Let say, for example, that the machine has a probability of $\frac{1}{3}$ to get from step A to step B. Then we have a problem when trying to simulate it by unbiased binary coins, because there is the binary expansion of $\frac{1}{3}$ is infinite. But we can still get as close as we want to the original machine, and this is good enough for our purposes.

The offline approach: Another way to consider nondeterministic machines is, as we did before, to use an additional input as a guess. For NP machines we gave an additional input that was used as a witness. The analogous idea is to view the outcome of the internal coin tosses as an auxiliary input. The machine will receive two inputs, the real input, x, and the guess input, r. Imagine that the machine receives this second input from an external 'coin tossing device' rather than toss coins internally.

Notation: We will use the following notation to discuss various properties of probabilistic machines:

$$Prob_r[M(x,r) = z]$$

Sometimes, we will drop the r and keep it implicitly like in the following notation:

$$Prob[M(x) = z]$$

By this notations we mean the probability that the machine M with real input x and guess input r, distributed uniformly, will give an output z. The probability space is that of all possible r taken with uniform distribution. This statement is more confusing than it seems to be because the machine may use different number of guesses for different inputs. It may also use different number of guesses on the same input, if the computation depends on the outcome of previous guesses.

Oded's Note: *Actually, the problem is with the latter case. That is, if on each input all computations use the same number of coin tosses (or "guesses"), denoted l, then each such computation occurs with probability 2^{-l}. However, in the general case, where the number of coin tosses may depend on the outcome of previous tosses, we may just observe that a halting computation with coin outcome sequence r occurs with probability exactly $2^{-|r|}$.*

Oded's Note: *An alternative approach is to modify the randomized machine so that it does use the same number of coin tosses in each computation on the same input.*

7.2 The classes RP and $coRP$ – One-Sided Error

The first two classes of languages that arise from probabilistic computations that we consider are the one-sided error (polynomial running time) computable languages. If there exist a machine that can decide the language with good probability in polynomial time it is reasonable to consider the problem as relatively easy. Good probability here means that the machine will be sure only in one case and will give the right answer in the other case but only with good probability (the cases are when $x \in L$ and when $x \notin L$).
From here on, a polynomial probabilistic Turing machine means a probabilistic machine that always (no matter what coin tosses it gets) halts after a polynomial (in the length of the input) number of steps.

Definition 7.1 (Random Polynomial-time – RP): *The complexity class RP is the class of all languages L for which there exist a probabilistic polynomial-time Turing machine M, such that*

$$x \in L \Rightarrow Prob[M(x) = 1] \geq \frac{1}{2}.$$

$$x \notin L \Rightarrow Prob[M(x) = 1] = 0.$$

Definition 7.2 (Complementary Random Polynomial-time – $coRP$): *The complexity class $coRP$ is the class of all languages L for which there exist a probabilistic polynomial-time Turing machine M, such that*

$$x \in L \Rightarrow Prob[M(x) = 1] = 1.$$

$$x \notin L \Rightarrow Prob[M(x) = 0] \geq \frac{1}{2}.$$

One can see from the definitions that these two classes complement each other. If you have a machine that decides a language L with good probability (in one of the above senses), you can use the same machine to decide the complementary language in the complementary sense.
That is, an alternative (and equivalent) way to define $coRP$ is:

$$coRP = \{\overline{L} : L \in RP\}$$

Comparing NP to RP: It is instructive to compare the definitions of RP and NP. In both classes we had the offline definition that used an external witness (in NP) or randomization (in RP).

Given an RP machine, M, since the machine run in polynomial-time, the size of the guesses that it can use is bounded by a polynomial in the size of x. For every given integer $n \in \mathsf{N}$ we consider the relation:

$$R_n \stackrel{\text{def}}{=} \left\{(x,r) \in \{0,1\}^n \times \{0,1\}^{p(n)} : M(x,r) = 1\right\}$$

which consists of all accepted inputs of length n and their accepting coin tosses (i.e r).

The same is also applicable for NP machines, which run also in polynomial-time and can only use witnesses that are bounded by a polynomial in the length of the input. So, for NP machine M, we consider the relation:

$$R_n \stackrel{\text{def}}{=} \left\{(x,y) \in \{0,1\}^n \times \{0,1\}^{p(n)} : M(x,y) = 1\right\}$$

which consist of all accepted inputs of length n and their witnesses (i.e y).

In both cases we will use the relation:

$$R = \bigcup_{n=1}^{\infty} R_n$$

which consists of all the accepted inputs and their witness/coin-tosses.

Using this relation we can compare Definition 7.1 to the definition of NP in the following table:

NP	RP
$x \in L \Rightarrow \exists y,\ (x,y) \in R$	$x \in L \Rightarrow \text{Prob}_r\ [(x,r) \in R] \geq \frac{1}{2}$
$x \notin L \Rightarrow \forall y,\ (x,y) \notin R$	$x \notin L \Rightarrow \forall r,\ (x,r) \notin R$

From this table, it is seems that these two classes are close. The witness in the nondeterministic model is replaced by the coin-tosses and the criteria for acceptance has changed. The difference is that, in the nondeterministic model, one witness was enough for us to say that an input is accepted, and in the probabilistic model we are asking for many coin-tosses. Clearly,

Proposition 7.2.1 $NP \supseteq RP$

Proof: Let L be an arbitrary language in RP. If $x \in L$ then there exist a Turing machine M and a coin-tosses y such that $M(x,y) = 1$ (more than $\frac{1}{2}$ of the coin-tosses are such). So we can use this y as a witness (considering the same machine as a nondeterministic machine with the coin-tosses as witnesses). If $x \notin L$ then $Prob_r[M(x,r) = 1] = 0$ so there is no witness. ∎

Notice that there is a big difference between nondeterministic Turing machines and probabilistic Turing machines. The first is a fictitious concept that is invented to explore the properties of search problems, while the second is a realistic model that describe machines that one can really build.We use the nondeterministic model to describe problems like a search problem with an efficient verification, while the probabilistic model is used as an efficient computation.

It is fair to ask if a computer can toss-coins as an elementary operation. We answer this question positively based on our notion of randomness and the ability of computers to use random-generating instrumentation like reading unstable electric circuits. The question is whether this random operation gives us more power than we had with the regular deterministic machines.

RP is one-sided error: The definition of RP does not ask for the same behavior on inputs that are in the language as it asks for inputs that are in the language.

- If $x \notin L$ then the answer of the machine must be correct no matter what guesses we make. In this case, the probability to get a wrong answer is zero so the answer of the machine is right for every r.
- But, if $x \in L$, the machine is allowed to make mistakes. In this case, we have a non-zero probability that the answer of the machine will be wrong (still this probability is not "too big").

The definition favors one type of mistake while in practice we don't find very good reason to favor it. We will see later that there are different families of languages that do not favor any type of error. We will call these languages two-sided error languages.

It was reasonable to discuss one-sided errors when we where developing NP, because verification is one-sided by nature, but it is less useful for exploring the notion of efficient computation.

Invariance of the constant and beyond: Recall that for $L \in RP$

$$x \in L \Rightarrow Prob_r[M(x,r) = 1] \geq \frac{1}{2}$$

The constant $\frac{1}{2}$ in the definition of RP is arbitrary. We could choose every constant strictly threshold between zero and one, and get the same complexity class. Our choice of $\frac{1}{2}$ is somewhat appealing because it says that at least half of the witnesses are good.
If you have, for example, a machine that can decide some language L with a greater probability than $\frac{1}{3}$ to say "YES" for an input that is in the language, you can build another machine that will invoke the first machine three times on every input and return the "YES" if one of them answered "YES". Obviously this machine will answer correctly on inputs that are not in the language (because the first machine will always say "NO"), and it will say "YES" on inputs that are in the language with higher probability than before. The original probability of not getting the correct answer when the input is in the language was smaller than $\frac{2}{3}$, when repeating the computation for three time this probability falls down to less than $\left(\frac{2}{3}\right)^3 = \frac{8}{27}$ meaning that we now get the correct answer with probability greater than $\frac{19}{27}$ (which is greater than $\frac{1}{2}$).

So we could use $\frac{1}{3}$ instead of $\frac{1}{2}$ without changing the class of languages. This procedure of amplification can be used to show the same result for every constant, but we will prove further that one can even use thresholds that depend on the length of the input.

We are looking at two probability spaces: one when $x \notin L$ and one when $x \in L$, and defined a random variable (representing the decision of the machine) on each of this spaces. In case $x \notin L$ the latter random variable is identically zero (i.e., "reject"), whereas in case $x \in L$ the random variable may be non-trivial (i.e., is 1 with probability above some given threshold and 0 otherwise).

Moving from one threshold to a higher one amount to the following: In case $x \in L$, the fraction of points in the probability space assigned the value 1 is lower bounded by the first threshold. Our aim is to hit such a point with probability lower bounded by a higher threshold. This is done by merely making repeated independent samples into the space, where the number of the trials is easily determined by the relation between the two thresholds. We stress that in case $x \notin L$ all points in the probability space are identically assigned (the value 0) and so it does not matter how many times we try (we'll always see zeros).

We will show that one can even replace the constant $\frac{1}{2}$ by either $\frac{1}{p(|x|)}$ or $1-2^{-p(|x|)}$, where $p(\cdot)$ is any fixed polynomial, and get the same family of languages. We take these two margins, because once we will show the equivalence of these two thresholds, it will follow that every threshold that one might think of in between will do. Consider the following definitions:

Definition 7.3 ($RP1$): *L is in $RP1$ if there exist a polynomial running-time Turing machine M and a polynomial $p(\cdot)$ such that*
$$\begin{array}{l} x \in L \Rightarrow Prob_r[M(x,r)=1] \geq \frac{1}{p(|x|)} \\ x \notin L \Rightarrow Prob_r[M(x,r)=0] = 1 \end{array}$$

Definition 7.4 ($RP2$): *L is in $RP2$ if there exist a polynomial running-time Turing machine M and a polynomial $p(\cdot)$ such that*
$$\begin{array}{l} x \in L \Rightarrow Prob_r[M(x,r)=1] \geq 1-2^{-p(|x|)} \\ x \notin L \Rightarrow Prob_r[M(x,r)=0] = 1 \end{array}$$

These definitions seems very far from each other, because in $RP1$ we ask for a probabilistic algorithm (Turing machine) that answer correctly with a very small probability (but not negligible), while in $R2$ we ask for an efficient algorithm (Turing machine) that we can almost ignore the probability of it's mistake. However, these two definition actually define the same class (as we will prove in the next paragraph). This implies that having an algorithm with a noticeable probability of success implies existence of and efficient algorithm with negligible probability of error.

Proposition 7.2.2 *RP1=RP2*

Proof:

$RP1 \supseteq RP2$
This direction is trivial because if $|x|$ is big enough then the bound in Definition 7.3 (i.e $\frac{1}{p(|x|)}$) is smaller than the bound in Definition 7.4 (i.e $1-2^{-p(|x|)}$) so being in $RP2$ implies being in $RP1$ for almost all inputs. The finitely many inputs for which this does not hold can be incorporated in the machine of Definition 7.3. Thus $RP1 \supseteq RP2$.

$RP1 \subseteq RP2$
We will use a method known as *amplification*:
We will try the weaker machine (of $RP1$) enough times so that the probability of giving a wrong answer will be small enough. Assume that we have a machine M_1 such that

$$\forall x \in L : Prob_r[M_1(x,r)=1] \geq \frac{1}{p(|x|)}$$

We will define a new machine M_2, up to a function $t(|x|)$ that we will determine later, as follows:

$$M_2(x) \stackrel{\text{def}}{=} \begin{cases} invoke\ M_1(x)\ t(|x|)\ times\ with\ different\ randomly\ selected\ r's \\ if\ some\ of\ these\ invocations\ returned\ 'YES'\ return\ 'YES' \\ else\ return\ 'NO' \end{cases}$$

Let $t = t(|x|)$. Then for $x \in L$

$$Prob[M_2(x)=0] = (Prob[M_1(x)=0])^{t(|x|)} \leq \left(1-\frac{1}{p(|x|)}\right)^{t(|x|)}$$

To find the desired $t(|x|)$ we can solve the equation:

$$\left(1 - \frac{1}{p(|x|)}\right)^{t(|x|)} \leq 2^{-p(|x|)}$$

And obtain

$$t(|x|) \geq p(|x|) \cdot \log_2 \left(1 - \frac{1}{p(|x|)}\right)^{-1} = \frac{p(|x|)^2}{\log_2 e}$$

Where $e \approx 2.7182818...$ is the natural logarithm base.

So by letting $t(|x|) = p(|x|)^2$ in the definition of M_2 we get a machine that run in polynomial time and decides L with probability greater than $1 - 2^{-p(|x|)}$ to give right answer for $x \in L$ (and always correct on $x \notin L$). ■

7.3 The class BPP – Two-Sided Error

One may argue that RP is too strict because it ask that the machine has to give 100% correct answer for inputs that are not in the language.

We derived the definition of RP from the definition of NP, but NP didn't reflect an actual computational model for search problems but rather a model for verification. One may find that looking at a two-sided error is more appealing as a model for search problem computations.

We want a machine that will recognize the language with high probability, where probability refers to the event "The machine answers correctly on an input x regardless if $x \in L$ or $x \notin L$". This will lead us to two-sided error version of the randomized computation. First recall the notation:

$$\chi_L(x) \stackrel{\text{def}}{=} \begin{cases} 1 & x \in L \\ 0 & x \notin L \end{cases}$$

Definition 7.5 (Bounded-Probability Polynomial-time – BPP): *The complexity class BPP is the class of all languages L for which there exist a probabilistic polynomial-time Turing machine M, such that*

$$\forall x : Prob[M(x) = \chi_L(x)] \geq \frac{2}{3}.$$

That means that:

$$If\ x \in L\ then\ Prob[M(x) = 1] \geq \frac{2}{3}.$$

$$If\ x \notin L\ then\ Prob[M(x) = 1] < \frac{1}{3}.$$

The phrase "bounded-probability" means that the success probability is bounded away from failure probability.

The BPP machine is a machine that makes mistakes but returns the correct answer most of the time. By running the machine a large number of times and returning the majority of the answers we are guaranteed by the law of large numbers that our mistake will be very small.
The idea behind the BPP class is that M accept by majority with a noticeable gap between the probability to accept inputs that are in language and the probability to accept inputs that are not in the language, and it's running time is bounded by a polynomial.

Invariance of constant and beyond: The $\frac{2}{3}$ is, again, an arbitrary constant. Replacing the $\frac{2}{3}$ in the definition by any other constant greater than $\frac{1}{2}$ does not change the class defined. If, for example we had a machine, M that recognize some language L with probability $p > \frac{1}{2}$, meaning that $Prob[M(x) = \chi_L(x)] \geq p$, we could easily build a machine that will recognize L with any given probability $q > p$ by invoking this machine sufficiently many times and returning the majority of the answers. This will clearly increase the probability of giving correct answer to the wanted threshold, and run in polynomial time.

In the RP case we had two probability spaces that we could distinguish easily because we had a guarantee that if $x \notin L$ then the probability to get one is zero, hence if you get $M(x) = 1$ for some input x, you could say *for sure* that $x \in L$.
In the BPP case, the amplification is less trivial because we have zeroes and ones in both probability spaces (the probability space is not constant when $x \in L$ nor when $x \notin L$).
The reason that we can apply amplification in the BPP case (despite the above difference) is that invoking the machine many times and counting how many times it returns one gives us an estimation on the fraction of ones in the whole probability space. It is useful to get an estimator for the fraction of the ones in the probability space because when this fraction is greater than $\frac{2}{3}$ we have that $x \in L$, and when this fraction is less than $\frac{1}{3}$ we have that $x \notin L$ (this fraction tells us in which probability space we are in).

If we rewrite the condition in Definition 7.5 as:

$$If\ x \in L\ then\ Prob[M(x) = 1] \geq \frac{1}{2} + \frac{1}{6}.$$

$$If\ x \notin L\ then\ Prob[M(x) = 1] < \frac{1}{2} - \frac{1}{6}.$$

We could consider the following change of constants:

$$If\ x \in L\ then\ Prob[M(x) = 1] \geq p + \epsilon.$$

$$If\ x \notin L\ then\ Prob[M(x) = 1] < p - \epsilon.$$

for any given $p \in (0, 1)$ and $0 < \epsilon < \min\{p, 1 - p\}$.

If we had such a machine, we could invoke the machine many times and get increasing probability to have the fraction of ones in our innovations to be in an ϵ neighborhood of the real fraction of ones in the whole space (by the law of large numbers). After some fixed number of iterations (that does not depend on x), we can get that probability to be larger than $\frac{2}{3}$.
This means that if we had such a machine (with p and ϵ instead of $\frac{1}{2}$ and $\frac{1}{6}$), we could build another machine that will invoke it some fixed number of times and will decide the same language with probability greater than $\frac{2}{3}$.

The conclusion is that the $\frac{1}{2} \pm \frac{1}{6}$ is arbitrary in Definition 7.5, and can be replaced by any $p \pm \epsilon$ such that $p \in (0, 1)$ and $0 < \epsilon < \min\{p, 1 - p\}$. But we can do more than that and use threshold that depend on the length of the input as we will prove in the following claims:

The weakest possible BPP definition: Using the above framework, we'll show that for every polynomial-time computable threshold, denoted f below, and any "noticeable" margin (represented by 1/poly), we can recover the "standard" threshold (of 1/2) and the "safe" margin of 1/6.

Claim 7.3.1 *$L \in BPP$ if and only if there exist a polynomial-time computable function $f : \mathsf{N} \mapsto [0,1]$, a positive polynomial $p(\cdot)$ and a probabilistic polynomial-time Turing machine M, such that:*

$$\forall x \in L : Prob[M(x) = 1] \geq f(|x|) + \frac{1}{p(|x|)}$$

$$\forall x \notin L : Prob[M(x) = 1] < f(|x|) - \frac{1}{p(|x|)}$$

Proof:
It is easy to see that by choosing $f(|x|) \equiv \frac{1}{2}$ and $p(|x|) \equiv 6$ we get the original definition of BPP (see Definition 7.5), hence every BPP language satisfies the above condition.

Assume that we have a probabilistic Turing machine, M, with these bounds on the probability to get 1. Then, for any given input x, we look at the random variable $M(x)$, which is a Bernoulli random variable with unknown parameter $p = Exp[M(x)]$. Using a well known fact that the expectation of a Bernoulli random variable is exactly the probability to get one we get that $p = Prob[M(x) = 1]$.
So by estimating p we can say something about whether $x \in L$ or $x \notin L$. The most natural estimator is to take the mean of n samples of the random variable (i.e the answers of n independent invocations of $M(x)$).
Then we will use the known statistical method of confidence intervals on the parameter p. The confidence interval method gives a bound within which a parameter is expected to lie with a certain probability. Interval estimation of a parameter is often useful in observing the accuracy of an estimator as well as in making statistical inferences about the parameter in question.
In our case we want to know with probability higher than $\frac{2}{3}$ if $p \in \left[0, f(|x|) - \frac{1}{p(|x|)}\right]$ or $p \in \left[f(|x|) + \frac{1}{p(|x|)}, 1\right]$. This is enough because $p \in \left[0, f(|x|) - \frac{1}{p(|x|)}\right] \Rightarrow x \notin L$ and $p \in \left[f(|x|) + \frac{1}{p(|x|)}, 1\right] \Rightarrow x \in L$ (note that $p \in \left(f(|x|) - \frac{1}{p(|x|)}, f(|x|) + \frac{1}{p(|x|)}\right)$ is impossible). So if we can get a bound of size $\frac{1}{p(|x|)}$ within which p is expected to lie within a probability greater than $\frac{2}{3}$, we can decide $L(M)$ with this probability (and hence $L(M) \in BPP$ by Definition 7.5).
We define the following Turing machine (up to an unknown number n that we will compute later) :

$$M'(x) \stackrel{\text{def}}{=} \begin{cases} \textit{Invoke } M(x) \textit{ } n \textit{ times (call the result of the } i\textit{'th invocation } t_i). \\ \textit{Compute } \hat{p} \leftarrow \frac{1}{n} \cdot \Sigma_{i=1}^{n} t_i \\ \textit{if } \hat{p} > f(|x|) \textit{ say } 'YES' \textit{ else say } 'NO' \end{cases}$$

Note that $\hat{p}$ is exactly the mean of a sample of size n taken from the random variable $M(x)$. This machine do the normal statistical process of estimating a random variable by taking samples and using the mean as an estimator for the expectation. If we will be able to show that with an appropriate n the estimator will not fall too far from the real value with a good probability, it will follow that this machine answers correctly with the same probability.
To resolve n we will use Chernoff's inequality which states that for any set of n independent Bernoulli variables $\{X_1, X_2, ..., X_n\}$ with the same expectations $p \leq \frac{1}{2}$ and for every $\delta, 0 < \delta \leq p(p-1)$, we have

$$Prob\left[\left|\frac{\Sigma_{i=1}^{n} X_i}{n} - p\right| > \delta\right] < 2 \cdot e^{-\frac{\delta^2}{2 \cdot p(1-p)} \cdot n} \leq 2 \cdot e^{-\frac{\delta^2}{2 \cdot \frac{1}{4}} \cdot n} = 2 \cdot e^{-2n \cdot \delta^2}$$

So by taking $\delta = \frac{1}{p(|x|)}$ and $n = -\frac{\ln \frac{1}{6}}{2 \cdot \delta^2}$ we get that our Turing machine M' will decide $L(M)$ with probability greater than $\frac{2}{3}$ suggesting that $L(M) \in BPP$. ■

The strongest possible BPP definition: On the other hand, one can reduce the error probability of BPP machines to an exponentially vanishing amount.

Claim 7.3.2 *For every $L \in BPP$ and every positive polynomial $p(\cdot)$ there exist a probabilistic polynomial-time Turing machine M, such that:*

$$\forall x : Prob[M(x) = \chi_L(x)] \geq 1 - 2^{-p(|x|)}$$

Proof:
If this condition is true for every polynomial, we can choose $p(|x|) \equiv 2$ and get M such that:

$$\begin{aligned} &\forall x : Prob[M(x) = \chi_L(x)] \geq 1 - 2^{-2} = \tfrac{3}{4} \\ &\Rightarrow \forall x : Prob[M(x) = \chi_L(x)] \geq \tfrac{2}{3} \\ &\Rightarrow L \in BPP \end{aligned}$$

Let L be a language in BPP and let M be the machine guaranteed in Definition 7.5. We can amplify the probability of right answer by invoking M many times and taking the majority of it's answers. Define the following machine (again up to the number n that we will find later):

$$M'(x) \stackrel{\text{def}}{=} \begin{cases} \textit{Invoke } M(x) \textit{ } n \textit{ times (call the result of the } i\textit{'th invocation } t_i). \\ \textit{Compute } \hat{p} \leftarrow \frac{1}{n} \cdot \Sigma_{i=1}^{n} t_i \\ \textit{if } \hat{p} > \frac{1}{2} \textit{ say } 'YES' \textit{ else say } 'NO' \end{cases}$$

From Definition 7.5, we get that if we know that $Exp[M(x)]$ is greater than half it follows that $x \in L$ and if we know that $Exp[M(x)]$ is smaller than half it follows that $x \notin L$ (because $Exp[M(x)] = Prob[M(x) = 1]$)
But Definition 7.5 gives us more. It says that the expectation of $M(x)$ is bounded away from $\frac{1}{2}$ so we can use the confidence interval method.

From Chernoff's inequality we get that

$$Prob\left[\left|M'(x) - Exp[M(x)]\right| \leq \frac{1}{6}\right] \geq 1 - 2 \cdot e^{-\frac{n}{18}}$$

But if $|M'(x) - Exp[M(x)]|$ is smaller than $\frac{1}{6}$ we get from Definition 7.5 that the answer of M' is correct, because it is close enough to the the expectation of $M(x)$ which is guaranteed to be above $\frac{2}{3}$ when $x \in L$ and bellow $\frac{1}{3}$ when $x \notin L$. So we get that:

$$Prob\left[M'(x) = \chi_L(x)\right] \geq 1 - 2 \cdot e^{-\frac{n}{18}}$$

Thus, for every polynomial $p(\cdot)$, we can choose n, such that

$$2^{p(|x|)} \geq 2 \cdot e^{-\frac{n}{18}}$$

and get that:

$$Prob\left[M'(x) = \chi_L(x)\right] \geq 1 - 2^{p(|x|)}$$

So M' satisfies the claimed condition. ■

Conclusion: We see that a gap of $\frac{1}{p(|x|)}$ and a gap of $1 - 2^{-p(|x|)}$ which look like "weak" and "strong" versions of BPP are the same. As shown above the "weak" version is actually equivalent to the "strong" version, and both are equivalent to the original definition of BPP.

Some comments about BPP:

1. $RP \subseteq BPP$
   It is obvious that one-sided error is a special case of two-sided error.

2. We don't know if $BPP \subseteq NP$. It might be so but we don't get it from the definition like we did in RP.

3. If we define $coBPP \stackrel{\text{def}}{=} \{\overline{L} : L \in BPP\}$ we get, from the symmetry of the definition of BPP, that $coBPP = BPP$.

7.4 The class PP

The class PP is wider than what we have seen so far. In the BPP case we had a gap between the number of accepting computations and non-accepting computations. This gap enabled us to determine with good probability (using confidence intervals) if $x \in L$ or $x \notin L$.
The gap was wide enough so we could invoke the machine polynomially many times and notice the difference between inputs that are in the language and inputs that are not in the language. The PP class don't put the gap restriction, hence the gap may be very small (even one guess can make a difference).
Running the machine polynomially many times may not help. If we have a machine that answers correctly with probability more than $\frac{1}{2}$, and we want to get another machine that answers correctly with probability greater than $\frac{1}{2} + \epsilon$ (for a given $0 < \epsilon < \frac{1}{2}$) we can't always do it in polynomial time because we might not have the gap that we had in Definition 7.5.

Definition 7.6 $$PP \stackrel{\text{def}}{=} \left\{ L \subseteq \{0,1\}^* \;\middle|\; \begin{array}{l} \textit{There exist a polynomial time} \\ \textit{Turing machine } M \textit{ s.t} \\ \forall x, Prob[M(x) = \chi_L(x)] > \frac{1}{2} \end{array} \right\}$$

Note that it is important that we define $>$ and not $\geq$, since otherwise we can simply "flip a coin" and completely ignore the input (we can decide to say $'YES'$ if we get head and $'NO'$ if we get tail and this will satisfy the definition of the machine) and there is no use for a machine that runs a lot of time and gives no more knowledge than what we already have (assuming one knows how to flip a coin). However the actual definition of PP gives very little as well: The difference between what happens in case $x \in L$ and in case $x \notin L$ is negligible (rather than "noticeable" as in the definition of BPP). We abuse this weak requirement in the proof of Item 3 (below).

From the definition of PP we get a few interesting facts:

1. $PP \subseteq PSPACE$
   Let L be a language in PP, let M be the probabilistic Turing machine that exists according to Definition 7.6. Let $p(\cdot)$ be the polynomial bounding it's running time. We will build a new machine M' that decides L in a polynomial space. Given an input x, the new machine will run M on x using all possible coin tosses with length $p(|x|)$ and decides by majority (i.e if M

accepted the majority of it's invocations then M' accepts x, else it rejects x).
Every invocation of M on x requires a polynomial space. And, because we can use the same space for all invocations, we see that M' uses polynomial space (the fact that we run it exponentially many times does not matter). The answer of M' is correct because M is a PP machine that answers correctly for more than half of the guesses.

2. Small Variants
We mentioned that, in Definition 7.6, we can't take $\geq$ instead of of $>$ because this will give us no information. But what about asking for $\geq$ when $x \notin L$ and $>$ when $x \in L$ (or the other way around) ? We will show, in the next claim, that this will not change the class of languages. A language has such a machine if and only if it has a PP machine.

Consider the following definition:

Definition 7.7 $$PP1 \stackrel{\text{def}}{=} \left\{ L \subseteq \{0,1\}^* \;\middle|\; \begin{array}{l} \textit{There exist a polynomial time} \\ \textit{Turing machine } M \textit{ s.t} \\ x \in L \Rightarrow Prob[M(x)=1] > \frac{1}{2} \\ x \notin L \Rightarrow Prob[M(x)=0] \geq \frac{1}{2} \end{array} \right\}$$

The next claim will show that this relaxation will not change the class defined:

Claim 7.4.1 $PP1 = PP$

Proof:
$PP \subseteq PP1$:
If we have a machine that satisfies Definition 7.6 it also satisfies Definition 7.7, so clearly $L \in PP \Rightarrow L \in PP1$.

$PP \supseteq PP1$:
Let L be any language in $PP1$. If M is the machine guaranteed by Definition 7.7, and $p(\cdot)$ is the polynomial bounding it's running time (and thus the number of coins that it uses), we can define another machine M' as follows:

$$M'\left(x, \left(a_1, a_2, ..., a_{p(|x|)+1}, b_1, b_2, ..., b_{p(|x|)}\right)\right) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \textit{if } a_1 = a_2 = ... = a_{p(|x|)+1} = 0 \textit{ then return } 'NO' \\ \textit{else return } M\left(x, \left(b_1, b_2, ..., b_{p(|x|)}\right)\right) \end{array} \right.$$

M' chooses one of two moves. One move, which happens with probability $2^{-(p(|x|)+1)}$, will return $'NO'$. The second move, which happens with probability $1 - 2^{-(p(|x|)+1)}$ will invoke M with independent coin tosses.
This gives us that

$$Prob[M'(x)=1] = Prob[M(x)=1] \cdot \left(1 - 2^{-(p(|x|)+1)}\right)$$

and

$$Prob[M'(x)=0] = Prob[M(x)=0] \cdot \left(1 - 2^{-(p(|x|)+1)}\right) + 2^{-(p(|x|)+1)}$$

The trick is to shift the answer of M towards the $'NO'$ direction with a very small probability. This shift is smaller than the smallest probability difference that M could have. So if $M(x)$ is biased towards the $'YES'$, our shift will keep the direction of the bias (it will only lower it). But if there is no bias (or bias towards NO), our shift will give us a bias towards the $'NO'$ answer.

If $x \in L$ then $Prob[M(x) = 1] > \frac{1}{2}$, hence $Prob[M(x) = 1] \geq \frac{1}{2} + 2^{-p(|x|)}$ (because the difference is at least one computation which happens with probability $2^{-p(|x|)}$), so:

$$\begin{aligned} Prob[M'(x) = 1] &\geq \left(\frac{1}{2} + 2^{-p(|x|)}\right) \cdot \left(1 - 2^{-(p(|x|)+1)}\right) \\ &= \frac{1}{2} + 2^{-p(|x|)} - 2^{-(p(|x|)+2)} - 2^{-(2p(|x|)+1)} > \frac{1}{2} \end{aligned}$$

If $x \notin L$ then $Prob[M(x) = 0] \geq \frac{1}{2}$, hence

$$\begin{aligned} Prob[M'(x) = 0] &\geq \frac{1}{2} \cdot \left(1 - 2^{-(p(|x|)+1)}\right) + 2^{-(p(|x|)+1)} \\ &= \frac{1}{2} - 2^{-(p(|x|)+2)} + 2^{-(p(|x|)+1)} > \frac{1}{2} \end{aligned}$$

And, as a conclusion, we get that in any case

$$Prob[M'(x) = \chi_L(x)] > \frac{1}{2}$$

So M' satisfies Definition 7.6, and thus $L \in PP$. ■

3. $NP \subseteq PP$
Suppose that $L \in NP$ is decided by a nondeterministic machine M with a running-time that is bounded by the polynomial $p(|x|)$. The following machine M' then will decide L by means of Definition 7.6:

$$M'\left(x, \left(b_1, b_2, ..., b_{p(|x|)+1}\right)\right) \stackrel{\text{def}}{=} \begin{cases} if\ b_1 = 1\ then\ return\ M\left(x, \left(b_2, b_3, ..., b_{p(|x|)+1}\right)\right) \\ else\ return\ 'YES' \end{cases}$$

M' uses it's random coin-tosses as a witness to M with only one toss that it does not pass to M'. This toss is used to choose it's move. One of the two possible moves gets it to the ordinary computation of M with the same input (and the witness is the random input). The other choice gets it to a computation that always accepts.
Consider a string x.

If M doesn't have an accepting computation then the probability that M' will answer 1 is exactly $\frac{1}{2}$ (it is the probability that the first coin will fall on one). On the other hand, if M has at least one accepting computation then the probability that M' will answer correctly is greater than $\frac{1}{2}$.

So we get that:

$x \in L \Rightarrow \quad Prob[M'(x) = 1] > \frac{1}{2}$
$x \notin L \Rightarrow \quad Prob[M'(x) = 0] \geq \frac{1}{2}$

By Definition 7.7, we conclude that $L \in PP1$, and by the previous claim ($PP = PP1$), we get that $L \in PP$.

4. $coNP \subseteq PP$
   Easily seen from the symmetry in the definition of PP.

7.5 The class ZPP – Zero error probability.

RP definition is asymmetric and we can't say whether $RP = coRP$. It would be interesting to examine the properties of $RP \bigcap coRP$ which is clearly symmetric. It seems that problems which are in $RP \bigcap coRP$ can benefit from the accurate result of RP deciding Turing machine (if $x \notin L$) and of $coRP$ deciding Turing machine (if $x \in L$).
Another interesting thing to consider is to let the machine say "I don't know" for some inputs. We will discuss machines that can return this answer but answer correctly otherwise.
We will prove that these two ideas give rise to the same class of languages.

Definition 7.8 (ZPP): *$L \in ZPP$ if there exist a probabilistic polynomial-time Turing machine M, such that:*

$$\begin{aligned} &\forall x, && Prob[M(x) = \perp] \leq \tfrac{1}{2} \\ &\forall x, && Prob[M(x) = \chi_L(x) \text{ or } M(x) = \perp] = 1 \end{aligned}$$

Where we denote the unknown answer sign as $\perp$.

Again the value $\frac{1}{2}$ is arbitrary and can be replaced like we did before to be anything between $2^{-p(|x|)}$ to $1 - \frac{1}{p(|x|)}$. If we have a ZPP machine that doesn't know the answer with probability half, we can run it $p(|x|)$ times and get a machine that doesn't know the answer with probability $2^{-p(|x|)}$ because this is the probability that none of our invocation know the answer (the other way is obvious because $2^{-p(|x|)}$ is smaller than $\frac{1}{2}$ for all but final inputs). If we have a machine that know the answer with probability $\frac{1}{p(|x|)}$, we can use it to build a machine that know the answer with probability $\frac{1}{2}$ by invoking it $p(|x|)$ times (the other way is, again, trivial).

Proposition 7.5.1 $ZPP = RP \bigcap coRP$

Proof: Take $L \in ZPP$. Let M be the machine guaranteed in Definition 7.8. We will show how to build a new machine M' which decides L according to Definition 7.1 (this will imply that $ZPP \subseteq RP$).

$$M'(x) \stackrel{\text{def}}{=} \begin{cases} b \leftarrow M(x) \\ \textit{if } b = \perp \textit{ then output } 0 \\ \textit{else output } b \textit{ itself} \end{cases}$$

By doing so, if $x \notin L$ then by returning 0 when $M(x) = \perp$ we will always answer correctly (because in this case $M(x) \neq \perp \Rightarrow M'(x) = \chi_L(x) \Rightarrow M'(x) = 0$).
If $x \in L$, the probability of getting the right answer with M' is greater than $\frac{1}{2}$ because M will return a definite answer ($M(x) \neq \perp$) with probability greater than $\frac{1}{2}$ and M's definite answers are

always correct (it never return a wrong answer because it returns $\perp$ when it is uncertain).
In the same way it can be seen that $ZPP \subseteq coRP$ (the machine that we will build will return 1 when M is uncertain), hence we get that $ZPP \subseteq RP \bigcap coRP$.

Assume now that $L \in RP \bigcap coRP$. Let M_{RP} be the RP machine and M_{coRP} the $coRP$ machine that decides L (according to Definition 7.1 and Definition 7.2). We define $M'(x)$ using M_{RP} and M_{coRP} as follows:

$$M'(x) \stackrel{\text{def}}{=} \begin{cases} run\ M_{RP}(x),\ if\ it\ says\ 'YES'\ then\ return\ 1 \\ run\ M_{coRP}(x),\ if\ it\ says\ 'NO'\ then\ return\ 0 \\ otherwise\ return\ \perp \end{cases}$$

If M_{RP} says $'YES'$ then, by Definition 7.1, we are guaranteed that $x \in L$. Notice that it can happen that $x \in L$ and $M_{RP}(x) = 0$ but not the other way around (There are 1's in the probability space $M(x)$ when $x \in L$, but the probability space $M(x)$ when $x \notin L$ is all zeroes. So if $M(x)$ returns $'YES'$, we know that the first probability space is the case).
In a similar way, if M_{coRP} says $'NO'$ then, by Definition 7.2, we are guaranteed that $x \notin L$. Thus we never get a wrong answer.

If $x \in L$ then, by Definition 7.1, we will get a $'YES'$ answer form M_{RP} and hence from M' with probability greater than $\frac{1}{2}$. If $x \notin L$ than, by Definition 7.2, we will get a $'NO'$ answer form M_{coRP} and hence from M' with probability greater than $\frac{1}{2}$.
So in either cases we can be sure that M' returns a definite (not $\perp$) and correct answer with probability greater than $\frac{1}{2}$.

The conclusion is that M' is indeed a ZPP machine so $RP \bigcap coRP \subseteq ZPP$ and, together with the previous part, we conclude that $RP \bigcap coRP = ZPP$. ■

Summing what we have seen so far we can write the following relations

$$P \subseteq ZPP \subseteq RP \subseteq BPP$$

It is believed that $BPP = P$ so there is no real help that randomized computations can contribute when trying to solve search problems. Also if the belief is true then all the distinctions between the above classes are of no use.

7.6 Randomized space complexity

Like we did with NL, we also define randomized space classes. Here also, it is possible to consider both the online and off-line models and we will work with the online model.

7.6.1 The definition

Definition 7.9 *For any function* $S : \mathsf{N} \to \mathsf{N}$

$$RSPACE(S) \stackrel{\text{def}}{=} \left\{ L \subseteq \{0,1\}^* \;\middle|\; \begin{array}{l} There\ exists\ a\ randomized\ Turing\ machine\ M \\ s.t.\ for\ any\ input\ x \in \{0,1\}^* \\ x \in L \Rightarrow Prob[M(x) = 1] \geq \frac{1}{2} \\ x \notin L \Rightarrow Prob[M(x) = 0] = 0 \\ and\ M\ uses\ at\ most\ S(|x|)\ space \\ and\ exp(S(|x|))\ time. \end{array} \right\}$$

We are interested in the case where the space is logarithmic. The class which put the logarithmic space restriction is RL.

Definition 7.10 $RL \stackrel{\text{def}}{=} RSPACE(log)$

The time restriction is very important. Let us see what happens if we don't put the time restriction in Definition 7.9.

Definition 7.11 *For any function* $S : \mathsf{N} \to \mathsf{N}$

$$badRSPACE(S) \stackrel{\text{def}}{=} \left\{ L \subseteq \{0,1\}^* \;\middle|\; \begin{array}{l} \textit{There exists a randomized Turing machine } M \\ \textit{s.t. for any input } x \in \{0,1\}^* \\ x \in L \Rightarrow Prob[M(x) = 1] \geq \frac{1}{2} \\ x \notin L \Rightarrow Prob[M(x) = 0] = 0 \\ \textit{and } M \textit{ uses at most } S(|x|) \textit{ space} \\ \text{(no time restrictions!)} \end{array} \right\}$$

Proposition 7.6.1 badRSPACE(S) = NSPACE(S)

Proof: We start with the easy direction. Let $L \in badRSPACE(S)$. If $x \in L$ then there are many witnesses but one is enough. On the other hand for $x \notin L$ there are no witness.

The other direction is the interesting one. Suppose $L \in NSPACE(S)$. Let M be the Non-deterministic Turing machine which decides L in space $S(|x|)$. Recall that for every $x \in L$ there exists an accepting computation of M on input x which halts within $exp(S(|x|))$ steps (see previous lectures!). Then if $x \in L$ there exist r of length $exp(S(|x|))$, so that $M(x, r) = 1$ (here r denotes the offline non-deterministic guesses used by M). Thus, selecting r uniformly among the strings of length $exp(S(|x|))$, the probability that $M(x, r) = 1$ is at least $2^{-exp(S(|x|))}$. So if we repeatedly invoke $M(x, .)$ on random r's, we can expect that after $2^{exp(S(|x|))}$ tries we will see an accepting computation (assuming all the time that $x \in L$).

> Oded's Note: *Note that the above intuitive suggestion already abuses the fact that badRSPACE has no time bounds. We plan to run in expected time which is* double exponential *in the space bound; whereas the good definition of RSPACE allows only time exponential in the space bound.*

So we want to run M on x and a newly randomly selected r (of length $exp(S(|x|))$) for about $2^{exp(S(|x|))}$ times and accept iff M accepts in one of these tries. A naive implementation is just to do so. But this requires holding a counter capable of counting upto $t \stackrel{\text{def}}{=} 2^{exp(S(|x|))}$, which means using space $exp(S(|x|))$ (which is much more than we are allowed). So we have the basic idea which is good but still have a problem how to count. The solution will be to use a "randomized counter" that will only use $S(|x|)$ space.

The randomized counter is implemented as follows. We "flip" $k = \log_2 t$ coins. If all are heads then we will stop otherwise we go on. The expected number of tries is $2^{-k} = t$, exactly the number of tries we wanted to have. But this randomized counter requires only a real counter capable of counting upto k, and so can be implemented in space $\log_2 k = \log_2 \log_2 t = S(|x|)$. ■

Clearly,

Claim 7.6.2 $L \subseteq RL \subseteq NL$

7.6.2 Undirected Graph Connectivity is in RL

In the previous lecture we saw that directed connectivity is NL-Complete. We will now show in brief that undirected connectivity is in RL. The problem is defined as follows.

Input: An undirected graph G and two vertices s and t.

Task: Find if there is a path between s and t in G.

Claim 7.6.3 *Let n denote the number of vertices in the graph. Then, with probability at least $\frac{1}{2}$, a random walk of length $8n^3$ starting from s visits all vertices in the connected component of s.*

By a random walk, we mean a walk which iteratively selects at random a neighbour of the current vertex and moves to it.

Proof sketch: In the following, we consider the connected component of vertex s, denoted $G' = (V', E')$. For any edge, (u, v) (in E'), we let $T_{u,v}$ be a random variable representing the number of steps taken in a random walk starting at u until v is first encountered. It is easy to see that $\mathrm{E}[T_{u,v}] \leq 2|E'|$. Also, letting $\mathrm{cover}(G')$ be the expected number of steps in a random walk starting at s and ending when the last of the vertices of V' is encountered, and C be any directed cycle which visits all vertices in G', we have

$$\begin{aligned} \mathrm{cover}(G') &\leq \sum_{(u,v)\in C} \mathrm{E}[T_{u,v}] \\ &\leq |C| \cdot 2|E'| \end{aligned}$$

Letting C be a traversal of some spanning tree of G', we conclude that $\mathrm{cover}(G') < 4 \cdot |E'| \cdot |V'|$. Thus, with probability at least $1/2$, a random walk of length $8 \cdot |E'| \cdot |V'|$ starting at s visits all vertices of G'. ∎

The algorithm for deciding undirected connectivity is now obvious: Just take a "random walk" of length $8n^3$ starting from vertex s and see if t is encountered. The space requirement is merely a register to hold the current vertex (i.e., $\log n$ space) and a counter to count upto $8n^3$ (again $(\log n)$ space). Furthermore, the use of a counter guarantees that the running time of the algorithm is exponential in its (logarithmic) space bound. The implementation is straightforward

1. Set $counter = 0$ and $v = s$. Compute n (the number of vertices in the graph).
2. Uniformly select a neighbour u of v.
3. If $u = t$ then halt and accept, else set $v = u$ and $counter = counter + 1$.
4. If $counter = 8n^3$ then halt and reject, else goto Step (2).

Cleraly, if s is connected to t then, by the above claim, the algorithm accepts with probability at least $1/2$. On the other hand, the algorithm alwas rejects if s is not connected to t. Thus, UNdirected graph CONNectivity (UNCONN) is in RL.

Note that the straightforward adaptation of the above algorithm to the directed case (i.e., directed graph connectivity considered in previous lecture) fails: Consider, for example, a directed graph consisting of a directed path $1 \to 2 \to \cdots \to n$ augmented by directed edges going from every vertex $i > 1$ to vertex 1. An algorithm which tries to take a directed random walk starting from

vertex 1 is highly unlikely to reach vertex n in poly(n) many steps. Loosely speaking, this is the case since in each step from a vertex $i > 1$, we move towards vertex n with probability 1/2, but otherwise return to vertex 1. The fact that the above algorithm fails should not come as a great surprise, as the directed connectivity problem is NL-complete and so placing it in RL will imply $NL = RL$.

> Oded's Note: *$NL = RL$ is not considred as unlikely as $NP = RP$, but even if $NL = RL$ proving this seems very hard.*

Bibliographic Notes

Probabilistic Turing Machines and corresponding complexity classes (including $\mathcal{BPP}, \mathcal{RP}, \mathcal{ZPP}$ and $\mathcal{PP}$) were first defined by Gill [2]. The proof that NSPACE equals badRSPACE (called RSPACE in [2]), as well as the techinque of a randomized counter is from [2].

The robusteness of the various classes under various thresholds was established above using straightforward amplifications (i.e., running the algorithm several times with independent random choices). Randomness-efficient amplification methods have been extensively studied since the mid 1980's. See Section 3.6 in [3].

The random-walk algorithm for deciding undirected connectivity is due to Aleliunas *et. al.* [1]. Other examples of randomized algorithms can be found in Appendix B.1 of [3]. We specifically recommend the following examples

- Testing primality (B.1.5): This BPP algorithm is different from the famous $coRP$ algorithm for recognizing the set of primes.
- Finding a perfect matching (B.1.2): This algorithm is arguably simpler than known deterministic polynomial-time algorithms.
- Finding minimum cuts in graphs (B.1.7): This algorithm is arguably simpler than known deterministic polynomial-time algorithms.

A much more extensive treatment of randomized algorithm is given in [4].

1. 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 FOCS*, pages 218–223, 1979.
2. J. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, Vol. 6(4), pages 675–695, 1977.
3. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. Algorithms and Combinatorics series (Vol. 17), *Springer*, 1998. Copies have been placed in the faculty's library.
4. R. Motwani and P. Raghavan. *Randomized Algorithms*, Cambridge University Press, 1995.

Lecture 8

Non-Uniform Polynomial Time ($\mathcal{P}$/Poly)

Notes taken by Moshe Lewenstein, Yehuda Lindell and Tamar Seeman

Summary: In this lecture we introduce the notion of non-uniform polynomial time and the corresponding complexity class $\mathcal{P}$/poly. In this computational model, Turing machines are provided an external advice string to aid them in their computation. The *non-uniformity* is expressed in the fact that a different advice string may be defined for every different length of input. We show that $\mathcal{P}$/poly upper bounds efficient computation (as $\mathcal{BPP} \subseteq \mathcal{P}$/poly), yet even contains some non-recursive languages. The effect of introducing *uniformity* is discussed (as an attempt to rid $\mathcal{P}$/poly of its absurd intractable languages) and shown to reduce the class to be exactly $\mathcal{P}$. Finally, we show that, among other things, $\mathcal{P}$/poly may help us separate $\mathcal{P}$ from $\mathcal{NP}$. We do this by showing that trivially $\mathcal{P} \subset \mathcal{P}$/poly, and that under a reasonable conjecture $\mathcal{NP} \not\subseteq \mathcal{P}$/poly.

8.1 Introduction

The class of $\mathcal{P}$/poly, or non-uniform polynomial time, is the class of Turing machines which receive external advice to aid computation. More specifically for all inputs of length n a Turing machine is supplemented with a single advice string a_n of polynomial length . Alternatively we may view a non-uniform machine as an infinite series of Turing machines $\{M_n\}$, where M_i computes for inputs of length i. In this case the advice is "hardwired" into the machine.

The class of $\mathcal{P}$/poly provides an upper bound on what is considered to be efficient computation. This upper bound is not tight; for example, as we shall show later, $\mathcal{P}$/poly contains non-recursive languages. However, the upper bound ensures that every efficiently computable language is contained in $\mathcal{P}$/poly.

An additional motivation in creating the class of $\mathcal{P}$/poly is to help separate the classes of $\mathcal{P}$ and $\mathcal{NP}$. This idea is explained in further detail below.

8.1.1 The Actual Definition

We now define the class of $\mathcal{P}$/poly according to two different definitions, and then show that these two definitions are in fact equivalent. Recall that:

$$\chi_L(x) = \begin{cases} 1, & \text{if } x \in L; \\ 0, & \text{otherwise.} \end{cases}$$

Definition 8.1 (standard): *$L \in \mathcal{P}$/poly if there exists a sequence of circuits $\{C_n\}$, where for each n, C_n has n inputs and one output, and there exists a polynomial $p(.)$ such that for all n, size(C_n) $\leq p(n)$ and $C_n(x) = \chi_L(x)$ for all $x \in \{0,1\}^n$.*

A series of polynomial circuits $\{C_n\}$ as defined above is called a *non-uniform family of circuits.* The non-uniformity is expressed in the fact that there is not necessarily any connection between a circuit of size n and $n+1$. In fact for every n we may define a completely different "algorithm".

Note that the circuits in the above definition can be simulated in time linear to their size. Thus although time is not explicitly mentioned in the definition, it is implicit.

Definition 8.2 (alternative): *$L \in \mathcal{P}$/poly if there exists a polynomial-time two-input machine M, a polynomial $p(.)$, and a sequence $\{a_n\}$ of advice strings, where length(a_n) $\leq p(n)$, such that for all n and for all $x \in \{0,1\}^n$, $M(a_n, x) = \chi_L(x)$.*

If exponentially long advice were allowed in the above definition, then a_n could be a look-up table containing $\chi_L(x)$ for any language L and every input x of length n. Thus every language would trivially be in such a class. However, this is not the case as a_n is polynomially bounded. Restricting the length of the advice defines a more meaningful class, but as we have mentioned, some intractable problems still remain "solvable".

Proposition 8.1.1 *The two definitions of $\mathcal{P}$/poly are equivalent.*

Proof:
($\Rightarrow$): Assume $L \in \mathcal{P}$/poly by Definition 1, i.e. there exists a family $\{C_n\}$ of circuits deciding L, such that size(C_n) is polynomial in n. Let desc(C_n) be the description of C_n according to a standard encoding of circuits. Consider the universal Turing machine M such that for all n, and all x of length n, M(desc(C_n),x) simulates $C_n(x)$. Then define the sequence $\{a_n\}$ of advice strings such that for every n, a_n = desc(C_n). Thus $L \in \mathcal{P}$/poly by Definition 2.

($\Leftarrow$): Assume L is in $\mathcal{P}$/poly by Definition 2, i.e. there exist a Turing machine M and a sequence of advice $\{a_n\}$ deciding L. We look at all possible computations of $M(a_n, \cdot)$ for n-bit inputs. $M(a_n, \cdot)$ is a polynomial time-bounded deterministic Turing machine working on n-length inputs. In the proof of Cook's Theorem, in Lecture 2, we showed that Bounded Halting is Levin-reducible to Circuit Satisfiability. Given an instance of Bounded Halting $(< M(\cdot,\cdot) >, x, 1^t)$ the reduction is comprised of constructing a circuit C which on input y outputs $M(x, y)$. The situation here is identical since for $M(a_n, \cdot)$ a circuit may be constructed which on input x outputs $M(a_n, x)$. In other words we build a sequence $\{C_n\}$ of circuits, where for each n, C_n is an encoding of $M(a_n, \cdot)$. Thus L is in $\mathcal{P}$/poly by Definition 1. ■

It should be noted that in Definition 2, M is a finite object, whereas $\{a_n\}$ may be an infinite sequence (as is the sequence $\{C_n\}$ of circuits according to Definition 1). Thus $\mathcal{P}$/poly is an unrealistic mode of computation, since such machines cannot actually be constructed.

8.1.2 $\mathcal{P}$/poly and the $\mathcal{P}$ versus $\mathcal{NP}$ Question

As mentioned above, one of the motivations in defining the class of $\mathcal{P}$/poly is to separate $\mathcal{P}$ from $\mathcal{NP}$. The idea is to show that there is a language which is in $\mathcal{NP}$ but is not in $\mathcal{P}$/poly, and thus not in $\mathcal{P}$. In this way, we would like to show that $\mathcal{P} \neq \mathcal{NP}$. To do so, though, we must first understand the relationship of $\mathcal{P}$/poly to the classes $\mathcal{P}$ and $\mathcal{NP}$. Trivially, $\mathcal{P} \subseteq \mathcal{P}$/poly because the class $\mathcal{P}$ may be viewed as the set of $\mathcal{P}$/poly machines with empty advice, i.e. $a_n = \lambda$ for all n.

At first glance, Definition 2 of $\mathcal{P}$/poly appears to resemble that of $\mathcal{NP}$. In $\mathcal{NP}$, $x \in L$ iff there exists a witness w_x such that $M(x, w_x) = 1$. The witness is somewhat analogous to the advice in $\mathcal{P}$/poly. However, the definition of $\mathcal{P}$/poly differs from that of $\mathcal{NP}$ in two ways:

1. For a given n, $\mathcal{P}$/poly has a universal witness a_n as opposed to $\mathcal{NP}$ where every x of length n may have a different witness.

2. In the definition of $\mathcal{NP}$, for every $x \notin L$, for every witness w, $M(x, w) = 0$. In other words, there do not exist false witnesses. However, this is not true for $\mathcal{P}$/poly. We do not claim that there are no bad advice strings for Definition 2 of $\mathcal{P}$/poly; we merely claim that there exists a good advice string.

We therefore see that the definitions of $\mathcal{NP}$ and $\mathcal{P}$/poly differ from each other; this raises the possibility that there may be a language which is in $\mathcal{NP}$ but not in $\mathcal{P}$/poly. As we shall show later this seems to be likely since a sufficient condition for the existence of such a language is based upon a reasonable conjecture. Since $\mathcal{P}$ is contained in $\mathcal{P}$/poly, finding such a language is sufficient to fulfill our goal. In fact, the original motivation for $\mathcal{P}$/poly was the belief that one may be able to prove lower bounds on sizes of circuits computing certain functions (e.g., the characteriztic function of an NP-complete language). So far, no such bounds are known (except if one restricts the circuits in various ways; as we'll discuss in next semester).

8.2 The Power of $\mathcal{P}$/poly

As we have mentioned, $\mathcal{P}$/poly is not a realistic mode of computation. Rather, it provides an *upper bound* on what we consider efficient computation (that is, any language not in $\mathcal{P}$/poly should definitely not be efficiently computable). In the last lecture we defined probabilistic computation and reevaluated our view of efficient computation to be $\mathcal{BPP}$, rather than $\mathcal{P}$. We now show that $\mathcal{BPP} \subseteq \mathcal{P}$/poly and therefore that $\mathcal{P}$/poly also upper bounds our "new" view of efficient computation.

However, we will also show that $\mathcal{P}$/poly contains far more than $\mathcal{BPP}$. This actually yields a very high upper bound. In fact $\mathcal{P}$/poly even contains non-recursive languages. This containment should convince anyone that $\mathcal{P}$/poly does not reflect any level of realistic computation.

Theorem 8.3 : *$\mathcal{P}$/poly contains non-recursive languages.*

Proof: This theorem is clearly implied from the following two facts:

1. There exist unary languages which are non-recursive, and

2. For every unary language L, $L \in \mathcal{P}$/poly.

We remind the reader that L is a unary language if $L \subseteq \{1\}^*$.

Proof of Claim 1:

Let L be any non-recursive language. Define $L' = \{1^{index(x)} \mid x \in L\}$ where index(x) is the position of x in the standard enumeration of binary strings (i.e. we view the string as a binary number). Clearly L' is unary and non-recursive (any Turing machine recognizing L' can trivially be used to recognize L).

Proof of Claim 2:

For every unary language L, define

$$a_n = \begin{cases} 1, & \text{if } 1^n \in L; \\ 0, & \text{otherwise.} \end{cases}$$

A Turing machine can trivially decide L in polynomial (even linear) time given x and $a_{|x|}$, by simply accepting iff x is unary and $a_{|x|} = 1$. Therefore, $L \in \mathcal{P}$/poly. ■

The ability to decide intractable languages is a result of the non-uniformity inherent in $\mathcal{P}$/poly. There is no requirement that the series $\{a_n\}$ is even computable.

Note that this method of reducing a language to its unary equivalent cannot help us with polynomial classes as the reduction itself is exponential. However, for recursive languages we are interested in computability only.

Theorem 8.4 : *$\mathcal{BPP} \subseteq \mathcal{P}$/poly.*

Proof: Let $L \in \mathcal{BPP}$. By means of amplification, there exists a probabilistic Turing machine M such that for every $x \in \{0,1\}^n$: $Prob_{r \in \{0,1\}^{poly(n)}}[M(x,r) = \chi_L(x)] > 1 - 2^{-n}$, (the probabilities are taken over all possible choices of random strings).
Equivalently, M is such that $Prob_r[M(x,r) \neq \chi_L(x)] < 2^{-n}$. We therefore have:

$$Prob_r[\exists x \in \{0,1\}^n : M(x,r) \neq \chi_L(x)] \leq \sum_{x \in \{0,1\}^n} Prob_r[M(x,r) \neq \chi_L(x)] < 2^n \cdot 2^{-n} = 1.$$

The first inequality comes from the Union Bound, that is, for every series of sets $\{A_i\}$ and every random variable X:

$$Prob(X \in \bigcup_{i=1}^{n} A_i) \leq \sum_{i=1}^{n} Prob(X \in A_i).$$

and the second inequality is based on the error probability of the machine.

Note that if for every random string r, there is at least one x such that $M(x,r) \neq \chi_L(x)$, then the above probability would equal 1. We can therefore conclude that there is *at least* one string r such that for every x, $M(x,r) = \chi_L(x)$. We therefore set $a_n = r$ (note that r is different for different lengths of input n, but this is fine according to the definition of $\mathcal{P}$/poly). Our $\mathcal{P}$/poly machine simulates M, using a_n as its random choices. ■

This method of proof is called *the probabilistic method*. We do not know how to find these advice strings and the proof of their existence is implicit. We merely argue that the probability that a random string is not an adequate advice is strictly smaller than 1. This is enough to obtain the theorem.

8.3 Uniform Families of Circuits

As we have mentioned earlier, circuits of different sizes belonging to a non-uniform family may have no relation to each other. This results in the absurd situation of having families of circuits deciding non-recursive languages.

This leads us to the following definition which attempts to define families of circuits which *do* match our expectations of realistic computation.

Definition 8.5 (uniform circuits): *A family of circuits $\{C_n\}$ is called* uniform *if there exists a deterministic polynomial time Turing machine M such that for every n, $M(1^n) = desc(C_n)$, where $desc(C_n)$ is a standard encoding of circuits.*

Thus a uniform family of circuits has a succinct (finite) description (or equivalently for a series of advice strings). Clearly, a uniform family of circuits cannot recognize non-recursive languages. Actually, the restriction of uniformity is far greater than just this.

Theorem 8.6 : *A language L has a uniform family of circuits $\{C_n\}$ such that for all n and for all $x \in \{0,1\}^n$ $C_n(x) = \chi_L(x)$ if and only if $L \in \mathcal{P}$.*

Proof:
($\Rightarrow$) Let $\{C_n\}$ be a uniform family of circuits deciding L, and M the polynomial time Turing machine which generates the family. The following is a polynomial time algorithm for deciding L:

On input x:

- $C_{|x|} \leftarrow M(1^{|x|})$
- Simulate $C_{|x|}(x)$ and return the result.

Since M is polynomial-time bounded and the circuits are of polynomial size, the algorithm clearly runs in polynomial time. Therefore $L \in \mathcal{P}$.

($\Leftarrow$) $L \in \mathcal{P}$. Therefore, there exists a polynomial time Turing machine M deciding L. As in the proof of Cook's Theorem, a polynomial size circuit deciding L on strings of length n may be built from M in time polynomial in n. The Turing machine M' that constructs the circuits may then be taken as M in the definition of uniform circuits. That is, given x, M' calculates $|x|$ and builds the appropriate circuit.

Alternatively, by Definition 2, no advice is necessary here and we may therefore take $a_n = \lambda$ for every n. ■

8.4 Sparse Languages and the $\mathcal{P}$ versus $\mathcal{NP}$ Question

In this section we will see why $\mathcal{P}$/poly may help us separate between $\mathcal{P}$ and $\mathcal{NP}$. We will first define sparse languages.

Definition 8.7 (sparse languages): *A language S is* sparse *if there exists a polynomial $p(\cdot)$ such that for every n $|S \cap \{0,1\}^n| \leq p(n)$.*

Example: Trivially, every unary language is sparse (take $p(n) = 1$).

Theorem 8.8 : *$\mathcal{NP} \subseteq \mathcal{P}/poly$ if and only if for every $L \in \mathcal{NP}$, the language L is Cook-reducible to a sparse language.*

As we conjecture that no $\mathcal{NP}$-Complete language can be sparse, we have that $\mathcal{NP}$ contains languages not found in $\mathcal{P}/\text{poly}$.

Proof: It is enough for us to prove that SAT $\in \mathcal{P}/\text{poly}$ if and only if SAT is Cook-reducible to some sparse language.

($\Rightarrow$) Suppose that SAT $\in \mathcal{P}/\text{poly}$. Therefore there exists a series of advice strings $\{a_n\}$ and a Turing machine M as in Definition 2, where $\forall n\ |a_n| \leq q(n)$ for some polynomial $q(\cdot)$.
Define $s_i^n = 0^{i-1}10^{q(n)-i}$ and define $S = \{1^n 0 s_i^n : \text{ for } n \geq 0 \text{ where bit } i \text{ of } a_n \text{ is } 1\ \}$.
Clearly S is sparse since for every n $|S \cap \{0,1\}^{n+q(n)+1}| \leq |a_n| \leq q(n)$.

We now show a Cook-reduction of SAT to S:
Input: φ of length n

1. Reconstruct a_n by $q(n)$ queries to S. Specifically, the queries are: $1^n 0 s_1^n, 1^n 0 s_2^n, ..., 1^n 0 s_{q(n)}^n$.
2. Run $M(a_n, \varphi)$ thereby solving SAT in (standard) polynomial time.

We therefore solve SAT with a polynomial number of queries to an S-oracle, i.e. SAT Cook-reduces to S.

($\Leftarrow$) Suppose that SAT Cook-reduces to some sparse language S. Therefore, there exists a polynomial time bounded oracle machine M^S which solves SAT. Let $t(\cdot)$ be M's (polynomial) time-bound. Then, on input x, machine M makes queries of length at most $t(|x|)$.
Construct a_n in the following way: concatenate all strings of length *at most* $t(n)$ in S. Since S is sparse, there exists some polynomial $p(\cdot)$ such that $\forall n\ |S \cap \{0,1\}^n| \leq p(n)$. The length of the list of strings of lengths exactly i in a_n is then less than or equal to $i \cdot p(i)$ (i.e. at most $p(i)$ different strings of length i each). Therefore:

$$|a_n| \leq \sum_{i=1}^{t(n)} i \cdot p(i) < t(n)^2 \cdot p(t(n))$$

So, a_n is polynomial in length. Now, given a_n, every oracle query to S can be "answered" in polynomial time. For a given string x, we check if $x \in S$ by simply scanning a_n and seeing if x appears or not. Therefore, M^S may by simulated by a deterministic machine with access to a_n. This machine takes at most $t(n) \cdot |a_n|$ time (each lookup may take as long as scanning the advice). Therefore SAT $\in \mathcal{P}/\text{poly}$. ■

As we have mentioned, we conjecture that there are no sparse $\mathcal{NP}$-Complete languages. This conjecture holds for both Karp and Cook reductions. However for Karp-reductions, the ramifications of the existence of a sparse $\mathcal{NP}$-Complete language would be extreme, and would show that $\mathcal{P} = \mathcal{NP}$. This is formally stated and proved in the next theorem. It is interesting to note that our belief that $\mathcal{NP} \not\subseteq \mathcal{P}/\text{poly}$ is somewhat parallel to our belief that $\mathcal{P} \neq \mathcal{NP}$ when looked at in the context of sparse languages.

Theorem 8.9 $\mathcal{P} = \mathcal{NP}$ *if and only if for every language $L \in \mathcal{NP}$, the language L is Karp-reducible to a sparse language.*

Proof:
($\Rightarrow$): Let $L \in \mathcal{NP}$. We define the following trivial function as a Karp-reduction of L to $\{1\}$:

$$f(x) = \begin{cases} 1, & \text{if } x \in L; \\ 0, & \text{otherwise.} \end{cases}$$

If $\mathcal{P} = \mathcal{NP}$ then L is polynomial-time decidable and it follows that f is polynomial-time computable. Therefore, L Karp-reduces to the language $\{1\}$, which is obviously sparse.

($\Leftarrow$): For sake of simplicity we prove a weaker result for this direction. However the claim is true as stated. Beforehand we need the following definition:

Definition 8.10 (guarded sparse languages): *A sparse language S is called* guarded *if there exists a sparse language G in $\mathcal{P}$ such that $S \subseteq G$.*

The language that we considered in the proof of theorem 8: $S = \{1^n 0 s_i^n : \text{for } n \geq 0, \text{ where bit } i \text{ of } a_n \text{ is } 1\}$ is an example of a sparse guarded language. It is obviously sparse and it is guarded by $G = \{1^n 0 s_i^n : \forall n \geq 0 \text{ and } 1 \leq i \leq q(n)\}$. Note that any unary language is also a guarded sparse language since $\{1^n : n \geq 0\}$ is sparse and trivially in $\mathcal{P}$.

The slightly weaker result that we prove for this direction is as follows.

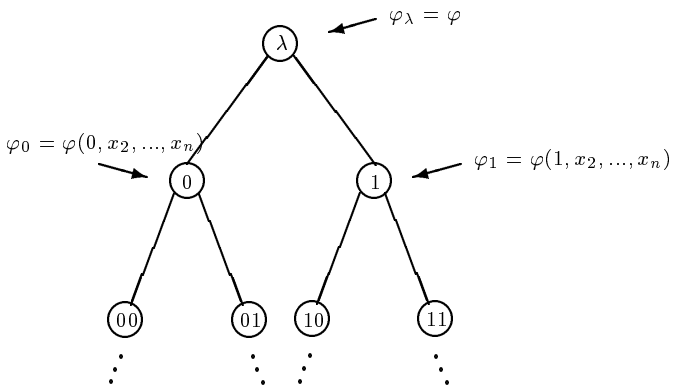
Proposition 8.4.1 *If SAT is Karp-reducible to a guarded sparse language then SAT $\in \mathcal{P}$.*

Proof: Assume that SAT is Karp-reducible to a sparse language S that is guarded by G. Let f be the Karp-reduction of SAT to S. We will show a polynomial-time algorithm for SAT.

Input: A Boolean formula $\varphi = \varphi(x_1, ..., x_n)$.

Envision the binary tree of all possible assignments. Each node is labelled $\alpha = \alpha_1\alpha_2...\alpha_i \in \{0,1\}^i$ which corresponds to an assignment of φ's first i variables. Let $\varphi_\alpha(x_{i+1}, ..., x_n) = \varphi(\alpha_1, ..., \alpha_i, x_{i+1}, ..., x_n)$ be the CNF formula corresponding to α. We denote $x_\alpha = f(\varphi_\alpha)$ (recall that $\varphi_\alpha \in \text{SAT} \Leftrightarrow x_\alpha \in S$).

The root is labelled λ, the empty string, where $\varphi_\lambda = \varphi$. Each node labelled α has two sons, one labelled $\alpha 0$ and the other labelled $\alpha 1$ (note that the sons have one variable less in their corresponding formulae). The leaves are labelled with n-bit strings corresponding to full assignments, and therefore to a Boolean constant.



The tree of assignments.

The strategy we will employ to compute φ will be a DFS search on this tree from root to leaves using a branch and bound technique. We backtrack from a node *only* if there is no satisfying assignment in its entire subtree. As soon as we find a leaf satisfying φ, we halt returning the assignment.

At a node α we consider x_α. If $x_\alpha \notin G$ (implying that $x_\alpha \notin S$), then φ_α is not satisfiable. This implies that the subtree of α contains no satisfying assignments and we can stop the search on this subtree. If $x_\alpha \in G$, then we continue searching in α's subtree.

At a leaf α we check if the assignment α is satisfiable (note that it is not sufficient to check that $x_\alpha \in G$ since f reduces to S and not to G). This is easy as we merely need to evaluate a Boolean expression in given values.

The key to the polynomial time-bound of the algorithm lies in the sparseness of G. If we visit a number of nodes equal to the number of strings in G of appropriate length, then the algorithm will clearly be polynomial. However, for two different nodes α and β, it may be that $x_\alpha = x_\beta \in G$ and we search both their subtrees resulting in visiting too many nodes. We therefore maintain a set B such that $B \subseteq G - S$ remains invariant throughout. Upon backtracking from a node α (where $x_\alpha \in G$), we place x_α in B. We then check for every node α, that $x_\alpha \notin B$ before searching its subtree, thus preventing a multiple search.

Algorithm: On input $\varphi = \varphi(x_1, ..., x_n)$.

1. $B \leftarrow \emptyset$
2. Tree-Search(λ)
3. In case the above call was not halted, reject φ as non-satisfiable.

In the following procedure, returning from a recursive call on α indicates that the subtree rooted in α contains no satisfying assignment (or, in other words, φ_α is not satisfiable). In case we reach a leaf associated with a satisfying assignment, the procedure halts outputting this assignment.

Procedure Tree-Search(α)

1. determine $\varphi_\alpha(x_{i+1}, ..., x_n) = \varphi(\alpha_1, ..., \alpha_i, x_{i+1}, ..., x_n)$
2. if $|\alpha| = n$: /* at a leaf - φ_α is a constant */
   if $\varphi_\alpha \equiv T$ then output the assignment α and halt
   else return
3. if $|\alpha| < n$:
   (a) compute $x_\alpha = f(\varphi_\alpha)$
   (b) if $x_\alpha \notin G$ /* checkable in poly-time, because $G \in \mathcal{P}$ */
   then return /* $x_\alpha \notin G \Rightarrow x_\alpha \notin S \Rightarrow \varphi_\alpha \notin$ SAT */
   (c) if $x_\alpha \in B$ then return
   (d) Tree-Search($\alpha 0$)
   Tree-Search($\alpha 1$)
   (e) /* We reach here only if both calls in the previous step fail. */
   if $x_\alpha \in G$ then add x_α to B
   (f) return

End Algorithm.

Correctness: During the algorithm B maintains the invariant $B \subseteq G - S$. To see this note that x_α is added to B only if $x_\alpha \in G$ and we are backtracking. Since we are backtracking there are no satisfying assignments in α's subtree, so $x_\alpha \notin S$.

Note that if $x_\alpha \in S$ then $x_\alpha \in G$ ($S \subseteq G$) and $x_\alpha \notin B$ (because B maintains $B \subseteq G - S$). Therefore, if φ is satisfiable then we will find some satisfying assignment since for all nodes α on the path from the root to the appropriate leaf, $x_\alpha \in S$, and its sons are developed.

Complexity: To show that the time complexity is polynomial it is sufficient to show that only a polynomial portion of the tree is "developed". The following claim will yield the desired result.

Claim 8.4.2 *Let α and β be two nodes in the tree such that (1) neither is a prefix/ancestor of the other and (2) $x_\alpha = x_\beta$. Then it is not possible that the sons of both nodes were developed (in Step 3d).*

Proof: Assume we arrived at α first. Since α is not an ancestor of β we arrive at β after backtracking from α. If $x_\alpha \notin G$ then $x_\beta \notin G$ since $x_\beta = x_\alpha$ and we will not develop either. Otherwise, it must be that $x_\alpha \in B$ after backtracking from α. Therefore $x_\beta \in B$ and its sons will not be developed (see Step 3c). ■

Corollary 8.4.3 *Only a polynomial portion of the tree is "developed".*

Proof: There exists a polynomial $q(.)$ that time-bounds the Karp-reduction f. Since every x_α is obtained by an application of f, $x_\alpha \in \cup_{i \leq q(n)}\{0,1\}^i$. Yet G is sparse so $|G \cap (\cup_{i \leq q(n)}\{0,1\}^i)| \leq p(n)$ for some polynomial $p(\cdot)$.

Consider a certain level of the tree. Every two nodes α and β on this level are not ancestors of each other. Moreover on this level of the tree there are at most $p(n)$ different α's such that $x_\alpha \in G$. Therefore by the previous claim the number of x_α's developed forward on this level is bounded by $p(n)$. Therefore the overall number of nodes developed is bounded by $n \cdot p(n)$. ■

SAT $\in \mathcal{P}$ and the proof is complete. ■

Bibliographic Notes

The class P/poly was defined by Karp and Lipton as part of a general formulation of "machines which take advise" [3]. They have noted the equivalence to the traditional formulation of polynomial-size circuits, the effect of uniformity, as well as the relation to Cook-reducibility to sparse sets (i.e., Theorem 8.8).

Theorem 8.4 is atrriburted to Adleman [1], who actually proved $\mathcal{RP} \subseteq \mathcal{P}$/poly using a more involved argument. Theorem 8.9 is due to Fortune [2].

1. L. Adleman, "Two theorems on random polynomial-time", in *19th FOCS*, pages 75–83, 1978.

2. S. Fortune, "A Note on Sparse Complete Sets", *SIAM J. on Computing*, Vol. 8, pages 431–433, 1979.

3. R.M. Karp and R.J. Lipton. "Some connections between nonuniform and uniform complexity classes", in *12th STOC*, pages 302-309, 1980.

Lecture 9

The Polynomial Hierarchy (PH)

Notes taken by Ronen Mizrahi

Summary: In this lecture we define a hierarchy of complexity classes starting from $\mathcal{NP}$ and yet contained in PSPACE. This is done in two ways, the first by generalizing the notion of Cook reductions, and the second by generalizing the definition of $\mathcal{NP}$. We show that the two are equivalent. We then try to make some observations regarding the hierarchy, our main concern will be to learn when does this hierarchy collapse, and how can we relate it to complexity classes that we know already such as BPP and P/Poly.

9.1 The Definition of the class PH

In the literature you may find three common ways to define this class, two of those ways will be presented here. (The third, via "alternating" machines is omitted here.)

9.1.1 First definition for PH: via oracle machines

Intuition

Recall the defintion of a Cook reduction, the reduction is done using a polynomial time machine that has access to some oracle. Requiring that the oracle will belong to a given complexity class C, will raise the question:

> What is the complexity class of all those languages that are Cook reducable to some language from C?

For example:

> Let us set the complexity class of the oracle to be $\mathcal{NP}$, then for Karp reduction we know that every language L, that is Karp reducable to some language in $\mathcal{NP}$ (say SAT), will also be in $\mathcal{NP}$. However it is not clear what complexity class will a Cook reduction (to $\mathcal{NP}$) yield.

Perliminary definitions

Definition 9.1 (the language $L(M^A)$): *The language $L(M^A)$ is the set of inputs accepted by machine M given access to oracle A.*

Notations:

- M^A : The orcale machine M with access to oracle A.
- $M^A(x)$: The output of the orcale machine M^A on input x.

We note the following interesting cases for the above definition:

1. M is a deterministic polynomial time machine. Then M is a Cook reduction of $L(M^A)$ to A.
2. M is a probabilistic polynomial time machine. Then M is a randomized Cook reduction of $L(M^A)$ to A.
3. M is a non-deteministic polynomial time machine (note that the non determinism is related only to M, A is an oracle and as such it always gives the right answer). When we define the polynomial hierarchy we will use this case.

Observe that given one of the above cases, knowing the complexity class of the oracle, will define another complexity class which is the set of languages $L(M^A)$, where A is an oracle from the given comlexity class. The resulting complexity class may be one that is known to us (such as $\mathcal{P}$ or $\mathcal{NP}$), or a new class.

Definition 9.2 (the class M^C): *Let M be an oracle machine. Then M^C is the set of languages obtained from the machine M given access to an oracle from the class of languages C. That is,*

$$M^C \stackrel{\text{def}}{=} \{L(M^A) : A \in C\}$$

For example:

- $M^{\mathcal{NP}} = \{L(M^A) : A \in \mathcal{NP}\}$

  Note: we do not gain much by using $\mathcal{NP}$, rather than any $\mathcal{NP}$-complete language (such as SAT). That is, we know that any language, A, in $\mathcal{NP}$ is Karp reducable to SAT, by using this reduction we can alter M, and obtain a new machine $\tilde{M}$, such that $L(M^A) = L(\tilde{M}^{SAT})$.

In the following definition we abuse notation a little. We write ${C_1}^{C_2}$ but refer to machines natually associated with the class C_1, and to their natural extension to orale machines. We note that not every class has a natural enumeration of machines associated with it, let allow a natural extension of such machines to oracle machines. However, such associations and extensions do hold for the main classes we are interested in such as $\mathcal{P}$, $\mathcal{NP}$ and $\mathcal{BPP}$.

Definition 9.3 (the class ${C_1}^{C_2}$ – a fuzzy framework): *Assume that C_1 and C_2 are classes of languages, and also that for each language L in C_1, there exists a machine M_L, such that $L = L(M_L)$. Furthermore, consider the extension of M_L into an oracle machine M so that given access to the empty oracle M behaves as M_L (i.e., $L(M_L) = L(M^{\emptyset})$). Then ${C_1}^{C_2}$ is the set of languages obtained from such machines M_L, where $L \in C_1$, given access to an oracle for a language from the class of languages C_2. That is,*

$$ {C_1}^{C_2} = \{L(M^A) : L(M^{\emptyset}) \in C_1 \ \& \ A \in C_2\} $$

The above framework can be properly instantiated in some important cases. For example:

- $\mathcal{P}^C = \{L(M^A) : M \text{ is deterministic polynomial-time oracle machine } \& \ A \in C\}$
- $\mathcal{NP}^C = \{L(M^A) : \text{same as above but } M \text{ is non-deterministic}\}$
- $\mathcal{BPP}^C = \{L(M^A) : \text{same as above but } M \text{ is probabilistic}\}$

  Here we mean that with probability at least $2/3$, machine M on input x and oracle aceess to $A \in C$ correctly decides whether $x \in L(M^A)$.

Back to the motivating question: Observe that saying that L is Cook-reducible to SAT (i.e., $L \propto_C SAT$) is equivalent to writing $L \in \mathcal{P}^{\mathcal{NP}}$. We may now re-address the question regarding the power of Cook reductions. Observe that $\mathcal{NP} \cup \text{co}\mathcal{NP} \subseteq \mathcal{P}^{\mathcal{NP}}$, this is because:

- $\mathcal{NP} \subseteq \mathcal{P}^{\mathcal{NP}}$ holds, because for $L \in \mathcal{NP}$ we can take the oracle A to be an oracle for the language L, and the machine $M \in \mathcal{P}$ to be a trivial machine that takes its input asks the oracle about it, and outputs the oracle's answer.
- $\text{co}\mathcal{NP} \subseteq \mathcal{P}^{\mathcal{NP}}$ holds, because we can take the same oracle as above, and a different (yet still trivial) machine $M \in \mathcal{P}$ that asks the oracle about its input, and outputs the boolean complement of the oracle's answer.

We conclude that under the assumption that $\mathcal{NP} \neq \text{co}\mathcal{NP}$, Cook-reductions to $\mathcal{NP}$ give us more power than Karp-reductions to the same class.

> Oded's Note: *We saw such a result already, but it was quite artificial. I refer to that fact that $\mathcal{P}$ is Cook-reducible to the class of trivial langugaes (i.e., the class $\{\emptyset, \{0,1\}^*\}$), whereas non-trivial languages can not be Karp-reduced to trivial ones.*

Actual definition

Definition 9.4 (the class Σ_i): *Σ_i is a sequence of sets and will be defined inductively:*

- $\Sigma_1 \stackrel{\text{def}}{=} \mathcal{NP}$
- $\Sigma_{i+1} \stackrel{\text{def}}{=} \mathcal{NP}^{\Sigma_i}$

Notations:

- $\Pi_i \stackrel{\text{def}}{=} \text{co}\Sigma_i$
- $\Delta_{i+1} \stackrel{\text{def}}{=} \mathcal{P}^{\Sigma_i}$

Definition 9.5 (The hierarchy – PH): $\mathsf{PH} \stackrel{\text{def}}{=} \cup_{i=1}^{\infty} \Sigma_i$

The arbitrary choice to use the Σ_i's (rather than the Π_i's or Δ_i's) is justified by the following observations.

Almost syntaxtic observations

Proposition 9.1.1 $\Sigma_i \cup \Pi_i \subseteq \Delta_{i+1} \subseteq \Sigma_{i+1} \cap \Pi_{i+1}$.

Proof: We prove each of the two containments:

1. $\Sigma_i \cup \Pi_i \subseteq \Delta_{i+1} = \mathcal{P}^{\Sigma_i}$.
   The reason for that is the same as for $\mathcal{NP} \cup \text{co}\mathcal{NP} \subseteq \mathcal{P}^{\mathcal{NP}} = \Delta_2$ (see above)
2. $\mathcal{P}^{\Sigma_i} \subseteq \Sigma_{i+1} \cap \Pi_{i+1}$.
   $\mathcal{P}^{\Sigma_i} \subseteq \mathcal{NP}^{\Sigma_i} = \Sigma_{i+1}$ is obvious. Since $\mathcal{P}^{\Sigma_i}$ is closed under complementation, $L \in \mathcal{P}^{\Sigma_i}$ implies that $\overline{L} \in \mathcal{P}^{\Sigma_i} \subseteq \Sigma_{i+1}$ and so $L \in \Pi_{i+1}$.

■

Proposition 9.1.2 $\mathcal{P}^{\Sigma_i} = \mathcal{P}^{\Pi_i}$ *and* $\mathcal{NP}^{\Sigma_i} = \mathcal{NP}^{\Pi_i}$.

Proof: Given a machine M and an oracle A, it is easy to modify M to $\tilde{M}$ such that: $L(M^{\overline{A}}) = L(\tilde{M}^A)$. The way we build $\tilde{M}$ is by taking M and flipping every answer obtained from the oracle. In particular, if M is deterministic (resp. non-deterministic) polynomail-time then so is $\tilde{M}$. Thus, for such M and any class $\mathcal{C}$ the classes $M^{\text{co}\mathcal{C}}$ and $\tilde{M}^{\mathcal{C}}$ are identical. ■

9.1.2 Second definition for PH: via quantifiers

Intuition

The approach taken here is to recall one of the definitions of $\mathcal{NP}$ and try to generalize it.

Definition 9.6 (polynomially-bounded relation): *a k-ary relation R is called* polynomially bounded *if there exists a polynomial $p(.)$ such that:*

$$\forall (x_1, \ldots, x_k)\,,\ [(x_1, \ldots, x_k) \in R \Longrightarrow (\forall i)\,|x_i| \leq p(|x_1|)]$$

Note: our definition requires that all the elements of the relation are not too long with regard to the first element, but the first element may be very long. We could even require a stronger condition: $\forall i \forall j |x_i| \leq p(|x_j|)$, this will promise that every element of the relation is not too long with regard to every one of the others. We do not make this requirement because the above definition will turn out to be satisfactory for our needs, this is because in our relations the first element is the input word, and we need the rest of the elements in the relation to be bounded in the length of the input. Also the complexity classes, that we will define using the notion of a polynomially bounded k-ary relation, will turn out the same for both the weak and the strong definition of the relation.

We now state again the definition of the complexity class $\mathcal{NP}$:

Definition 9.7 ($\mathcal{NP}$): *$L \in \mathcal{NP}$ if there exists a polynomially bounded and polynomial time recognizable binary relation R_L such that:*

$$x \in L \ \ \textit{iff}\ \ \exists y \ s.t.\ (x, y) \in R_L$$

The way to generalize this definition will be to use a k-ary relation instead of just a binary one.

Actual definition

What we redefine is the sequence of sets Σ_i such that Σ_1 will remain $\mathcal{NP}$. The definition for PH remains the union of all the Σ_i's.

Definition 9.8 (Σ_i): *$L \in \Sigma_i$ if there exists a polynomially bounded and polynomial time recognizable (i+1)-ary relation R_L such that:*

$$x \in L \text{ iff } \exists y_1 \forall y_2 \exists y_3 \ldots Q_i y_i, \text{ s.t. } (x, y_1, \ldots, y_i) \in R_L$$

- $Q_i = \forall$ *if i is even*
- $Q_i = \exists$ *otherwise*

9.1.3 Equivalence of definitions

We have done something that might seem a mistaken; that is, we have given the same name for an object defined by two different definitions. However, we now intend to prove that the classes produced by the two definitions are equal. A more conventional way to present those two definitions is to state one of them as the definition for PH, and then prove an "if and only if" theorem that characterizes PH according to the other definition.

Theorem 9.9 : *The above two definitions of PH are equivalent. Furthermore, for every i, the class Σ_i as in Definition 9.4 is identical to the one in Definition 9.8.*

Proof: We will show that for every i, the class Σ_i by the two definitions is equal. In order to distinguish between the clases produced by the two definitions we will introduce the following notation:

- Σ_i^1 is the set Σ_i produced by the first definition.
- Σ_i^2 is the set Σ_i produced by the second definition.
- Π_i^1 is the set Π_i produced by the first definition.
- Π_i^2 is the set Π_i produced by the second definition.

Part 1: We prove by induction on i that $\forall i$, $\Sigma_i^2 \subseteq \Sigma_i^1$:

- Base of induction: Σ_1 was defined to be $\mathcal{NP}$ in both cases so there is nothing to prove.
- We assume that the claim holds for i and prove for $i+1$: suppose $L \in \Sigma_{i+1}^2$ then by definition it follows that there exists a relation R_L such that:

  $$x \in L \text{ iff } \exists y_1 \forall y_2 \exists y_3 \ldots Q_i y_i Q_{i+1} y_{i+1}, \text{ s.t. } (x, y_1, \ldots, y_i, y_{i+1}) \in R_L$$

  In other words this means that:

  $$x \in L \text{ iff } \exists y_1, \text{ s.t. } (x, y_1) \in L_i$$

  where L_i is defined as follows:

  $$L_i \stackrel{\text{def}}{=} \{(x', y') : \forall y_2 \exists y_3 \ldots Q_i y_i Q_{i+1} y_{i+1}, \text{ s.t. } (x', y', \ldots, y_i, y_{i+1}) \in R_L\}$$

We claim that $L_i \in \Pi_i^2$, this is by complementing the definition of Σ_i^2. If we do this complementation for $L \in \Sigma_i^2$ we get:

$$x \in L \text{ iff } \exists y_1 \forall y_2 \ldots Q_i y_i, \text{ s.t. } (x, y_1, \ldots, y_i) \in R_L$$

$$x \in \overline{L} \text{ iff } \forall y_1 \exists y_2 \ldots \overline{Q}_i y_i, \text{ s.t. } (x, y_1, \ldots, y_i) \notin R_L$$

This is almost what we had in the definition of L_i except for the "$\notin R_L$" as opposed to "$\in R_L$". Remember that deciding membership in R_L is polynomial time recognizable, and therefore its complement is also so. Now that we have that $L_i \in \Pi_i^2$, we can use the inductive hypothesis $\Pi_i^2 \subseteq \Pi_i^1$. So far we have managed to show that:

$$x \in L \text{ iff } \exists y_1, \text{ s.t. } (x, y_1) \in L_i$$

Where L_i belongs to Π_i^1. We now claim that $L \in \mathcal{NP}^{\Pi_i^1}$, this is true because we can write a non-deterministic, polynomial-time machine, that decides membership in L, by guessing y_1, and using an oracle for L_i. Therefore we can further conclude that:

$$L \in \mathcal{NP}^{\Pi_i^1} \equiv \mathcal{NP}^{\Sigma_i^1} \equiv \Sigma_{i+1}^1.$$

Part 2: We prove by induction on i that $\forall i$, $\Sigma_i^1 \subseteq \Sigma_i^2$:

- Base of induction: as before.
- Induction step: suppose $L \in \Sigma_{i+1}^1$ then there exists a non-deterministic polynomial time machine M such that $L \in L(M^{\Sigma_i^1})$ which means that:

$$\exists L' \in \Sigma_i^1, \text{ s.t. }, L = L(M^{L'})$$

From the definition of $M^{L'}$ it follows that:

$$x \in L \text{ iff } \exists y, q_1, a_1, \ldots, q_t, a_t \text{ s.t. } :$$

1. Machine M, with non-determinstic choices y, interacts with its oracle in the following way:
   - 1^{st} query $= q_1$ and 1^{st} answer $= a_1$
   - .
   - .
   - .
   - t^{th} query $= q_t$ and t^{th} answer $= a_t$
2. for every $1 \leq j \leq t$:
   - $(a_j = 1) \Longrightarrow q_j \in L'$
   - $(a_j = 0) \Longrightarrow q_j \notin L'$

where y is a description of the non-determinstic choices of M.

Let us view the above according to the second definition, that is, according to the definition with quantifiers, then the first item is a polynomial time predicate and therefore this potentially puts L in $\mathcal{NP}$. The second item involves L'. Recall that $L' \in \Sigma_i^1$ and that by the inductive hypothesis $\Sigma_i^1 \subseteq \Sigma_i^2$, and therefore we can view membership in L' according to the second definition, and embed this result within what we have above. This will yield that for every $1 \leq j \leq t$:

- $(a_j = 1) \Longrightarrow \exists y_1^{(j,1)} \forall y_2^{(j,1)} \ldots Q_i y_i^{(j,1)}$, s.t. $(q_j, y_1^{(j,1)}, \ldots, y_i^{(j,1)}) \in R_{L'}$
- $(a_j = 0) \Longrightarrow \forall y_1^{(j,2)} \exists y_2^{(j,2)} \ldots Q_i y_i^{(j,2)}$, s.t. $(q_j, y_1^{(j,2)}, \ldots, y_i^{(j,2)}) \in \overline{R}_{L'}$

Let us define:

- w_1 is the concatenation of:
  $y, q_1, a_1, \ldots, q_t, a_t$, and $y_1^{(j,1)}$ for all j s.t. $(a_j = 1)$.
- w_2 is the concatenation of:
  $y_1^{(j,2)}$ for all j s.t. $(a_j = 0)$, and $y_2^{(j,1)}$ for all j s.t. $(a_j = 1)$.
  .
  .
  .
- w_i is the concatenation of:
  $y_{i-1}^{(j,2)}$ for all j s.t. $(a_j = 0)$, and $y_i^{(j,1)}$ for all j s.t. $(a_j = 1)$.
- w_{i+1} is the concatenation of:
  $y_i^{(j,2)}$ for all j s.t. $(a_j = 0)$.

R_L will be the $(i+1)$-ary relation defined in the following way: $(w_1, \ldots, w_{i+1}) \in R_L$ iff for every $1 \leq j \leq t$:

- $(a_j = 1) \Longrightarrow (q_j, y_1^{(j,1)}, \ldots, y_i^{(j,1)}) \in R_{L'}$
- $(a_j = 0) \Longrightarrow (q_j, y_1^{(j,2)}, \ldots, y_i^{(j,2)}) \in \overline{R}_{L'}$

where the w_i's are parsed analogously to the above.

Since $R_{L'}$ and $\overline{R}_{L'}$ where polynomially bounded, and polynomial time recognizable, so is R_L.

Altogether we have:

$$x \in L \text{ iff } \exists w_1, \forall w_2, \ldots Q_{i+1} w_{i+1}, \text{ s.t. } (w_1, \ldots, w_{i+1}) \in R_L$$

It now follows from the definition of Σ_{i+1}^2 that $L \in \Sigma_{i+1}^2$ as needed.

■

9.2 Easy Computational Observations

Proposition 9.2.1 $PH \subseteq PSPACE$

Proof: We will show that $\Sigma_i \subseteq PSPACE$ for all i. Let $L \in \Sigma_i$, then we know by the definition with quantifiers that:

$$x \in L \text{ iff } \exists y_1 \forall y_2 \exists y_3 \ldots Q_i y_i, \text{ s.t. } (x, y_1, \ldots, y_i) \in R_L$$

Given x we can use i variables to try all the possibilities for $y_1, \ldots, y_i$ and make sure that they meet the above requirement. Since the relation R_L is polynomially bounded, we have a polynomial bound on the length of each of the y_i's that we are checking. Thus we have constructed a deterministic machine that decides L.

This machine uses i variables, the length of each of them is polynomially bounded in the length of the input. Since i is a constant, the overall space used by this machine is polynomial. ■

Proposition 9.2.2 *$\mathcal{NP} = \text{co}\mathcal{NP}$ implies $PH \subseteq \mathcal{NP}$ (which implies $PH = \mathcal{NP}$).*

Intuitively the extra power that non-deterministic Cook reductions have over non-deterministic Karp reductions, comes from giving us the ability to complement the oracle's answers for free. What we claim here is that if this power is meaningless then the whole hierarchy collapses.

Proof: We will show by induction on i that $\forall i$, $\Sigma_i = \mathcal{NP}$:

1. $i = 1$: by definition $\Sigma_1 = \mathcal{NP}$.

2. Induction step: by the inductive hypothesis it follows that $\Sigma_i = \mathcal{NP}$ so what remains to be shown is that $\mathcal{NP}^{\mathcal{NP}} = \mathcal{NP}$. Containment in one direction is obvious so we focus on proving that $\mathcal{NP}^{\mathcal{NP}} \subseteq \mathcal{NP}$. Let $L \in \mathcal{NP}^{\mathcal{NP}}$ then there exist a non-deterministic, polynomial-time machine M, and an oracle $A \in \mathcal{NP}$, such that $L = L(M^A)$. Since $\mathcal{NP} = \text{co}\mathcal{NP}$ it follows that $\overline{A} \in \mathcal{NP}$ too. Therefore, there exist relations R_A and $R_{\overline{A}}$ ($\mathcal{NP}$ relations for A and $\overline{A}$ respectively) such that:

   - $q \in A$ iff $\exists w$, s.t. $(q, w) \in R_A$.
   - $q \in \overline{A}$ iff $\exists w$, s.t. $(q, w) \in R_{\overline{A}}$

   Using these relations, and the definition of $\mathcal{NP}^{\mathcal{NP}}$ we get:
   $x \in L$ iff $\exists y, q_1, a_1, \ldots, q_t, a_t$, such that, for all $1 \leq j \leq t$:

   - $a_j = 1 \iff q_j \in A \iff \exists w_j, (q_j, w_j) \in R_A$
   - $a_j = 0 \iff q_j \in \overline{A} \iff \exists w_j, (q_j, w_j) \in R_{\overline{A}}$.

   Define:

   - w is the concatenation of: $y, q_1, a_1, \ldots, q_t, a_t, w_1, \ldots, w_t$
   - R_L is a binary relation such that:
     $(x, w) \in R_L$ iff for all $1 \leq j \leq t$:
     - $a_j = 1 \Longrightarrow (q_j, w_j) \in R_A$
     - $a_j = 0 \Longrightarrow (q_j, w_j) \in R_{\overline{A}}$.

     Since M is a polynomial-time machine, t is polynomial in the length of x. Combining this fact with the fact that both R_A and $R_{\overline{A}}$ are polynomial-time recognizable, and polynomially bounded, we conclude that so is R_L.

   All together we get that there exists an $\mathcal{NP}$ relation R_L such that :

   $$x \in L \text{ iff } \exists w, \text{ s.t. } (x, w) \in R_L$$

   Thus, $L \in \mathcal{NP}$.

■

Generalizing Proposition 9.2.2, we have

Proposition 9.2.3 *For every $k \geq 1$, if $\Pi_k = \Sigma_k$ then $PH = \Sigma_k$.*

A proof is presented in the appendix to this lecture.

9.3 BPP is contained in PH

Not knowing whether $\mathcal{BPP}$ is contained in $\mathcal{NP}$, it is of some confort to know that it is contained in the Polynomial-Hierarchy (which extends $\mathcal{NP}$).

Theorem 9.10 (Sipser and Lautemann): $BPP \subseteq \Sigma_2$.

Proof: Let $L \in BPP$ then there exists a probabilistic polynomial time machine $A(x,r)$ where x is the input and r is the random guess. By the definition of BPP, with some amplification we get, for some polynomial $p(.)$:

$$\forall x \in \{0,1\}^n, \text{ s.t. } \Pr_{r \in_R \{0,1\}^{p(n)}}[A(x,r) \neq \chi(x)] < \frac{1}{3p(n)}$$

where $\chi(x) = 1$ if $x \in L$ and $\chi(x) = 0$ otherwise.

> Oded's Note: *A word about the above is in place. Note that we do* **not** *assert that the error decreases as a fast fixed function of n, where the function is fixed before we determine the randomness complexity of the new algorithm. We saw result of that kind in Lecture 7; but here we claim something different. That is, that the error probability may depend on the randomness complexity of the new algorithm. Still, the dependency required here is easy to achieve. Specifically, suppose that the original algorithm uses $m = \text{poly}(n)$ coins. Then by running it t times and ruling by majority we decrease the error probability to $\exp(-\Omega(t))$. The randomness complexity of the new algorithm is tm. So we need to set t such that $\exp(-\Omega(t)) < 1/3mt$, which can be satisfied with $t = O(\log m) = O(\log n)$.*

The key observation is captured by the following claim

Claim 9.3.1 *Denote $m = p(n)$ then, for every $x \in L \cap \{0,1\}^n$, there exist $s_1, \ldots, s_m \in \{0,1\}^m$ such that*

$$\forall r \in \{0,1\}^m, \bigvee_{i=1}^{m} A(x, r \oplus s_i) = 1 \tag{9.1}$$

Actually, the same sequence of s_i's may be used for all $x \in L \cap \{0,1\}^n$ (provided that $m \geq n$ which holds without loss of generality). However, we don't need this extra property.

Proof: We will show existence of such s_i's by the Probabilistic Method: That is, instead of showing that an object with some property exists we will show that a random object has the property with positive probability. Actually, we will upper bound the probability that a random object does not have the desired property. In our case we look for existence of s_i's satisfying Eq. (9.1), and so we will upper bound the probability, denoted P, that randomly chosen s_i's do not satisfy Eq. (9.1):

$$\begin{aligned}
P &\stackrel{\text{def}}{=} \Pr_{s_1,\ldots,s_m \in_R \{0,1\}^m}[\neg \forall r \in \{0,1\}^m, \bigvee_{i=1}^{m} (A(x, r \oplus s_i) = 1)] \\
&= \Pr_{s_1,\ldots,s_m \in_R \{0,1\}^m}[\exists r \in \{0,1\}^m, \bigwedge_{i=1}^{m} (A(x, r \oplus s_i) = 0)] \\
&\leq \sum_{r \in \{0,1\}^m} \Pr_{s_1,\ldots,s_m \in_R \{0,1\}^m}[\bigwedge_{i=1}^{m} (A(x, r \oplus s_i) = 0)]
\end{aligned}$$

where the inequality is due to the union bound. Using the fact that the events of choosing s_i's uniformly are independent, we get that the probability of all the events happening at once equals to the multiplication of the probabilities. Therefore:

$$P \leq \sum_{r\in\{0,1\}^m} \prod_{i=1}^{m} \Pr_{s_i\in_R\{0,1\}^m}[A(x, r \oplus s_i) = 0]$$

Since in the above probability r is fixed, and the s_i's are uniformly distributed then (by a property of the $\oplus$ operator), the $s_i \oplus r$'s are also uniformly distributed. Recall that we consider an arbitrary fixed $x \in L \cap \{0,1\}^n$. Thus,

$$\begin{aligned} P &\leq 2^m \cdot \Pr_{s\in_R\{0,1\}^m}[A(x, s) = 0]^m \\ &\leq 2^m \cdot \left(\frac{1}{3m}\right)^m \ll 1 \end{aligned}$$

The claim holds. ■

Claim 9.3.2 *For any $x \in \{0,1\}^n \setminus L$ and for all $s_1, \ldots, s_m \in \{0,1\}^m$, there exists $r \in \{0,1\}^m$ so that $\bigvee_{i=1}^m A(x, r \oplus s_i) = 0$.*

Proof: We will actually show that for all $s_1, \ldots, s_m$ there are many such r's. Let $s_1, \ldots, s_m \in \{0,1\}^m$ be arbitrary.

$$\Pr_{r\in\{0,1\}^m}[\bigvee_{i=1}^{m} A(x, r \oplus s_i) = 0] = 1 - \Pr_{r\in\{0,1\}^m}[\bigvee_{i=1}^{m} A(x, r \oplus s_i) = 1]$$

However, since $x \notin L$ and $\Pr_{r\in\{0,1\}^m}[A(x, r) = 1] < 1/3m$, we get

$$\begin{aligned} \Pr_{r\in\{0,1\}^m}[\bigvee_{i=1}^{m} A(x, r \oplus s_i) = 1] &\leq \sum_{i=1}^{m} \Pr_{r\in\{0,1\}^m}[A(x, r \oplus s_i) = 1] \\ &\leq m \cdot \frac{1}{3m} = \frac{1}{3} \end{aligned}$$

and so,

$$\Pr_{r\in\{0,1\}^m}[\bigvee_{i=1}^{m} A(x, r \oplus s_i) = 0] \geq \frac{2}{3}$$

Therefore there exist (many) such r's and the claim holds. ■

Combining the results of the two claims together we get:

$$x \in L \text{ iff } \exists s_1, \ldots s_m \in \{0,1\}^m, \forall r \bigvee_{i=1}^{m} A(x, r \oplus s_i) = 1$$

This assertion corresponds to the definition of Σ_2, and therefore $L \in \Sigma_2$ as needed. ■

Comment: The reason we used the $\oplus$ operator is because it has the property that given an arbitrary fixed r, if s is uniformly distributed then $r \oplus s$ is also uniformly distributed. Same for fixed s and random r. Any other efficient binary operation with this property may be used as well.

9.4 If NP has small circuits then PH collpases

The following result shows that an unlikely event regarding non-uniform complexity (i.e., the class $\mathcal{P}$/poly) implies an unlikely event regarding uniform complexity (i.e., PH).

Theorem 9.11 (Karp & Lipton): *If* $\mathcal{NP} \subseteq \mathcal{P}/\text{poly}$ *then* $\Pi_2 = \Sigma_2$, *and so* $PH = \Sigma_2$.

Proof: We will only prove the first implication in the theorem. The second follows by Proposition 9.2.3. Showing that Σ_2 is closed under complementation, gives us that $\Pi_2 = \Sigma_2$. So what we will actually prove is that $\Pi_2 \subseteq \Sigma_2$.

Let L be an arbitrary language in Π_2, then there exists a trinary polynomially bounded, and polynomial time recognizable relation R_L such that:

$$x \in L \text{ iff } \forall y \exists z \text{ s.t. }, (x, y, z) \in R_L$$

Let us define:

$$L' \stackrel{\text{def}}{=} \{(x', y') : \exists z, \text{ s.t. } (x', y', z) \in R_L\}$$

Then we get that:

- $x \in L$ iff $\forall y, (x, y) \in L'$
- $L' \in \mathcal{NP}$

Consider a Karp reduction of L' to 3SAT, call it f:

$$x \in L \text{ iff } \forall y, f(x, y) \in 3SAT$$

Let us now use the assumption that $\mathcal{NP} \subseteq P/Poly$ for 3SAT, then it follows that 3SAT has small circuits $\{C_m\}_m$, where m is the length of the input. We claim that also 3SAT has small circuits $\{C'_n\}_n$ where n is the number of variables in the formula. This claim holds since the length of a 3SAT formula is of $O(n^3)$ and therefore $\{C'_n\}$ can use the larger sets of circuits $C_1, \ldots, C_{O(n^3)}$. Let us embed the circuits in our statement regarding membership in L, this will yield:

$x \in L$ iff $\exists (C'_1, \ldots, C'_n)$ $(n \stackrel{\text{def}}{=} max_y\{\#var(f(x, y))\})$ s.t.:

- $C'_1, \ldots, C'_n$ correctly computes 3SAT, for formulas with a corresponding number of variables.
- $\forall y,\ C'_{\#var(f(x,y))}(f(x, y)) = 1$

The second item above gives us that $L \in \Sigma_2$, since the quantifiers are as needed. However it is not clear that the first item is also bahaving as needed. We will restate the first item as follows:

$$\forall \phi_1, \ldots, \phi_n, [\bigwedge_{i=2}^{n} C'_i(\phi_i) = (C'_{i-1}(\phi'_i) \vee C'_{i-1}(\phi''_i)) \bigwedge C'_1 \textit{ operates correctly}] \tag{9.2}$$

Where:

$\phi_i(x_1, \ldots, x_i)$ is any formula over i variables.

$\phi'_i(x_1, \ldots, x_{i-1}) \stackrel{\text{def}}{=} \phi_i(x_1, \ldots, x_{i-1}, 0)$

$\phi''_i(x_1, \ldots, x_{i-1}) \stackrel{\text{def}}{=} \phi_i(x_1, \ldots, x_{i-1}, 1)$

A stupid technicality: Note that assigning a value to one of the variables, gives us a formula that is not in CNF as required by 3SAT (as its clauses may contain constants). However this can easily be achieved, by iterating the following process, where in each iteration one of the following rules is applied:

- $x \vee 0$ should be changed to x.
- $x \vee 1$ should be changed to 1.
- $x \wedge 0$ should be changed to 0.
- $x \wedge 1$ can be changed to x.
- $\neg 1$ can be changed to 0.
- $\neg 0$ can be changed to 1.

If we end-up with a formula in which some variables do not appear, we can augment it by adding clauses of the form $x \wedge \neg x$.

> Oded's Note: *An alternative resolution of the above technicality is to extend the definition of CNF so to allow constnats to appear (in clauses).*

Getting back to the main thing: We have given a recursive definition for a correct computation of the circuits (on 3SAT). The base of the recursion is checking that a single variable fromula is handled correctly by C'_1, which is very simple (just check if the single variable formula is satisfiable or not, and compare it to the output of the circuit). In order to validate the $(i+1)^{\text{th}}$ circuit, we wish to use the i^{th} circuit, which has already been validated. Doing so requires us to reduce the number of variables in the formula by one. This is done by assigning to one of the variables both possible values (0 or 1), and obtaining two formulas upon which the i^{th} circuit can be applied. The full formula is satisfiable iff at least one of the reduced formulas is satisfiable. Therefore we combine the results of applying the i^{th} circuit on the reduced formulas, with the $\vee$ operation. It now remains to compare it to the value computed by the $(i+1)^{\text{th}}$ circuit on the full formula. This is done for all formula over $i+1$ variables (by the quantification $\forall \phi_{i+1}$).

So all together we get that:

$$x \in L \text{ iff } \exists (C'_1, \ldots, C'_n), \text{ s.t. } \forall y, (\phi_1, \ldots, \phi_n), \ (x, (C'_1, \ldots, C'_n), (y, \phi_1, \ldots, \phi_n)) \in R_L$$

where R_L is a polynomially-bounded 3-ary relation defined using the Karp reduction f, Eq. (9.2) and the simplifying process above. Specifically, the algorithm recognizing R_L computes the formula $f(x,y)$, determines the formulas ϕ'_i and ϕ''_i (for each i), and evaluates circuits (the description of which is given) on inputs which are also given. Cleraly, this algorithm can be implemented in polynomial-time, and so it follows that $L \in \Sigma_2$ as needed. ■

Bibliographic Notes

The Polynomial-Time Hierarchy was introduced by Stockmeyer [6]. The third equivalent formulation via "alternating machines" can be found in [1].

The fact that $\mathcal{BPP}$ is in the Polynomial-time hierarchy was proven independently by Lautemann [4] and Sipser [5]. We have followed Lautemann's proof. The ideas underlying Sipser's proof

found many applications in complexity theory, and will be presented in the next lecture (in the approximation procedure for $\#\mathcal{P}$). Among these applications, we mention Stockmeyer's approximation procedure for $\#\mathcal{P}$ (cf., citel9:S83), the reduction of SAT to uniqueSAT (cf. [8] and next lecture), and the equivalence between public-coin interactive proofs and general interactive proofs (cf. [2] and Lecture 11).

The fact that $\mathcal{NP} \subseteq \mathcal{P}/\text{poly}$ implies a collapse of the Polynomial-time hierarchy was proven by Karp and Lipton [3].

1. A.K. Chandra, D.C. Kozen and L.J. Stockmeyer. Alternation. *JACM*, Vol. 28, pages 114–133, 1981.

2. 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*, pages 59–68, 1986.

3. R.M. Karp and R.J. Lipton. "Some connections between nonuniform and uniform complexity classes", in *12th STOC*, pages 302-309, 1980.

4. C. Lautemann. BPP and the Polynomial Hierarchy. *IPL*, 17, pages 215–217, 1983.

5. M. Sipser. A Complexity Theoretic Approach to Randomness. In *15th STOC*, pages 330–335, 1983.

6. L.J. Stockmeyer. The Polynomial-Time Hierarchy. *Theoretical Computer Science*, Vol. 3, pages 1–22, 1977.

7. L. Stockmeyer. The Complexity of Approximate Counting. In *15th STOC*, pages 118–126, 1983.

8. 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.

Appendix: Proof of Proposition 9.2.3

Recall that our aim is to prove the claim:

For every $k \geq 1$, if $\Pi_k = \Sigma_k$ then $PH = \Sigma_k$.

Proof: For an arbitrary fixed k, we will show by induction on i that $\forall i \geq k, \Sigma_i = \Sigma_k$:

1. Base of induction: when $i = k$, there is nothing to show.

2. Induction step: by the inductive hypothesis it follows that $\Sigma_i = \Sigma_k$, so what remains to be shown is that $\mathcal{NP}^{\Sigma_k} = \Sigma_k$. Containment in one direction is obvious so we focus on proving that $\mathcal{NP}^{\Sigma_k} \subseteq \Sigma_k$.

   Let $L \in \mathcal{NP}^{\Sigma_k}$, then there exist a non-deterministic, polynomial-time machine M, and an oracle $A \in \Sigma_k$, such that $L = L(M^A)$. Since $\Pi_k = \Sigma_k$ it follows that $\overline{A} \in \Sigma_k$ too. Therefore, there exist relations R_A and $R_{\overline{A}}$ ($k+1$-ary relations, polynomially bounded, and polynomial time recognizable, for A and $\overline{A}$ respectively) such that :

   - $q \in A$ iff $\exists w_1, \forall w_2, \ldots, Q_k w_k$ s.t. $(q, w_1, \ldots w_k) \in R_A$.

- $q \in \overline{A}$ iff $\exists w_1, \forall w_2, \ldots, Q_k w_k$ s.t. $(q, w_1, \ldots w_k) \in R_{\overline{A}}$.

Using those relations, and the definition of $\mathcal{NP}^{\Sigma_k}$ we get:
$x \in L$ iff $\exists y, q_1, a_1, \ldots, q_t, a_t$ s.t. for all $1 \leq j \leq t$:

- $a_j = 1 \Longleftrightarrow q_j \in A \Longleftrightarrow \exists w_1^{(j,1)}, \forall w_2^{(j,1)}, \ldots, Q_k w_k^{(j,1)}$ s.t. $(q_j, w_1^{(j,1)}, \ldots w_k^{(j,1)}) \in R_A$.
- $a_j = 0 \Longleftrightarrow q_j \in \overline{A} \Longleftrightarrow \exists w_1^{(j,0)}, \forall w_2^{(j,0)}, \ldots, Q_k w_k^{(j,0)}$ s.t. $(q_j, w_1^{(j,0)}, \ldots w_k^{(j,0)}) \in R_{\overline{A}}$.

Define:

- w_1 is the concatenation of: $y, q_1, a_1, \ldots, q_t, a_t, w_1^{(1,0)}, \ldots, w_1^{(t,0)}, w_1^{(1,1)}, \ldots, w_1^{(t,1)}$.
  .
  .
- w_k is the concatenation of: $w_k^{(1,0)}, \ldots, w_k^{(t,0)}, w_k^{(1,1)}, \ldots, w_k^{(t,1)}$
- R_L is a $k+1$-ary relation such that:
  $(x, w_1, \ldots, w_k) \in R_L$ iff for all $1 \leq j \leq t$:
  - $a_j = 1 \implies (q_j, w_1^{(j,1)}, \ldots w_k^{(j,1)}) \in R_A$.
  - $a_j = 0 \implies (q_j, w_1^{(j,0)}, \ldots w_k^{(j,0)}) \in R_{\overline{A}}$.

  Since M is a polynomial machine, then t is polynomial in the length of x. R_A and $R_{\overline{A}}$ are polynomial time recognizable, and polynomially bounded relations.
  Therefore R_L is also so.

All together we get that there exists a polynomially bounded, and polynomial time recognizable relation R_L such that :

$$x \in L \text{ iff } \exists w_1, \forall w_2, \ldots, Q_k w_k \text{ s.t. } (x, w_1, \ldots, w_k) \in R_L$$

By the definition of Σ_k, $L \in \Sigma_k$.

■

Lecture 10

The counting class $\#\mathcal{P}$

Notes taken by Oded Lachish, Yoav Rodeh and Yael Tauman

Summary: Up to this point in the course, we've focused on decision problems where the questions are YES/NO questions. Now we are interested in counting problems. In $\mathcal{NP}$ an element was in the language if it had a short checkable witness. In $\#\mathcal{P}$ we wish to count the number of witnesses to a specific element. We first define the complexity class $\#\mathcal{P}$, and classify it with respect to other complexity classes. We then prove the existence of $\#\mathcal{P}$-complete problems, and mention some natural ones. Then we try to study the relation between $\#\mathcal{P}$ and $\mathcal{NP}$ more exactly, by showing we can probabilistically approximate $\#\mathcal{P}$ using an oracle in $\mathcal{NP}$. Finally, we refine this result by restricting the oracle to a weak form of SAT (called $uniqueSAT$).

10.1 Defining $\#\mathcal{P}$

We used the notion of an $\mathcal{NP}$-relation when defining $\mathcal{NP}$. Recall:

Definition 10.1 *($\mathcal{NP}$ relation) : An $\mathcal{NP}$ relation is a relation $R \subseteq \Sigma^\star \times \Sigma^\star$ such that:*

- *R is polynomial time decidable.*
- *There exists a polynomial $p(\cdot)$ such that for every $(x,y) \in R$, it holds that $|y| \leq p(|x|)$.*

Given an $\mathcal{NP}$-relation R we defined:

Definition 10.2 $L_R \stackrel{\text{def}}{=} \{x \in \Sigma^\star \mid \exists y \text{ s.t. } (x,y) \in R\}$

We regard the y's that satisfy $(x,y) \in R$ as witnesses to the membership of x in the language L_R. The decision problem associated with R, is the question: Does there exist a witness to a given x? This is our definition of the class $\mathcal{NP}$. Another natural question we can ask is: How many witnesses are there for a given x? This is exactly the question we capture by defining the complexity class $\#\mathcal{P}$. We first define:

Definition 10.3 *For every binary relation $R \subseteq \Sigma^\star \times \Sigma^\star$, the counting function $f_R : \Sigma^\star \to \mathsf{N}$, is defined by:*

$$f_R(x) \stackrel{\text{def}}{=} |\{y \mid (x,y) \in R\}|$$

The function f_R captures our notion of counting witnesses in the most natural way. So, we define $\#\mathcal{P}$ as a class of functions. Specifically, functions that count the number of witnesses in an $\mathcal{NP}$-relation.

Definition 10.4 $\#\mathcal{P} = \{f_R : R \text{ is an } \mathcal{NP} \text{ relation}\}$

We encounter some problems when trying to relate $\#\mathcal{P}$ to other complexity classes, since it is a class of functions while all classes we discussed so far are classes of languages. To solve this, we are forced to give a less natural definition of $\#\mathcal{P}$, using languages. For each $\mathcal{NP}$-relation R, we associate a language $\#_R$. How do we define $\#_R$? Our first attempt would be:

Definition 10.5 (Counting language — first attempt) *:* $\#_R = \{(x,k) : |\{y : (x,y) \in R\}| = k\}$

First observe that given an oracle to f_R, it is easy to decide $\#_R$. This is a nice property of $\#_R$, since we would like it to closely represent our other formalism using functions. For the same reason we also want the other direction: Given an oracle to $\#_R$, we would like to be able to calculate f_R efficiently (in polynomial time). This is not as trivial as the other direction, and in fact, is not even necessarily true. So instead of tackling this problem, we alter our definition:

Definition 10.6 (Counting language — actual definition) *:* $\#_R = \{(x,k) : |\{y : (x,y) \in R\}| \geq k\}$. *In other words,* $(x,k) \in \#_R$ *iff* $k \leq f_R(x)$.

We choose the last definition, because now we can prove the following:

Proposition 10.1.1 *For each $\mathcal{NP}$-relation R :*

1. *$\#_R$ is Cook reducible to f_R*
2. *f_R is Cook reducible to $\#_R$*

We denote the fact that problem P Cook reduces to problem Q by $P \ \alpha_c \ Q$.
Proof:

1. *($\#_R$ is Cook reducible to f_R)* : Given (x,k), we want to decide whether $(x,k) \in \#_R$. We use our oracle for f_R, by calling it with parameter x. As an answer we get : $l = |\{y \ : \ (x,y) \in R\}|$. If $l \geq k$ then we accept, otherwise reject.

2. *(f_R is Cook reducible to $\#_R$)* : Given x, we want to find $f_R(x) = |\{y : (x,y) \in R\}|$ using our oracle. We know $f_R(x)$ is in the range $\{0, \ldots, 2^{|p(x)|}\}$, where $p(\cdot)$ is the polynomial bounding the size of the witnesses in the definition of an NP-relation. The oracle given is exactly what we need to implement binary search.

   $BINARY(x, Lower, Upper)$:

   - if $(Lower = Upper)$ output $Lower$.
   - $Middle = \frac{Lower+Upper}{2}$
   - if $(x, Middle) \in \#_R$ output $BINARY(x, Middle, Upper)$
   - else output $BINARY(x, Lower, Middle)$

   Where the branching in the third line is because if $(x, Middle) \in \#_R$, then $f_R(x) \geq Middle$, so we need only search for the result in the range $[Middle, Upper]$. A symmetric argument explains the *else* clause.

   The output is: $f_R(x) = BINARY(x, 0, 2^{p(|x|)})$. Binary search in general, runs in time logarithmic in interval it searches in. In our case : $O(\log(2^{p(|x|)})) = O(p(|x|))$. We conclude, that the algorithm runs in polynomial time in $|x|$.

■

Notice that we could have changed our definition of $\#_R$, to be:

$$\#_R = \{(x,k) : |\{y : (x,y) \in R)\}| \leq k\}$$

The proposition would still hold. We could have also changed it to a strict inequality, and gotten the same result.

¿From now on we will use the more natural definition of $\#\mathcal{P}$: as a class of functions. This doesn't really matter, since we showed that in terms of cook-reducibility, the two definitions are equivalent.

It seems that the counting problem related to a relation R should be harder than the corresponding decision problem. It is unknown whether it is strictly harder, but it is certainly not weaker. That is,

Proposition 10.1.2 *For every $\mathcal{NP}$-relation R, the corresponding language L_R Cook reduces to f_R.*

Proof: Given $x \in \Sigma^\star$, use the oracle to calculate $f_R(x)$. Now, $x \in L_R$ if and only if $f_R(x) \geq 1$. ■

Corollary 10.7 *$\mathcal{NP}$ Cook reduces to $\#\mathcal{P}$*

On the other hand we can bound the complexity of $\#\mathcal{P}$ from above:

Claim 10.1.3 *$\#\mathcal{P}$ Cook reduces to* $\mathcal{P}$SPACE

Proof: Given x, we want to calculate $f_R(x)$ using polynomial space. Let $p(\cdot)$ be the polynomial bounding the length of the witnesses of R. We run over all possible witnesses of length $\leq p(|x|)$. For each one, we check in polynomial time whether it is a witness for x, and sum the number of witnesses. All this can be done in space $O(p(|x|) + q(|x|))$, where $q(\cdot)$ is the polynomial bounding the running time (and therefore space) of the witness checking algorithm. Such a polynomial exists since R is an $\mathcal{NP}$-relation. ■

10.2 Completeness in #P

When one talks about complexity classes, proving the existence, and finding complete problems in the complexity class, is of great importance. It helps reason about the whole class using only one specific problem. Therefore, we are looking for an $\mathcal{NP}$-relation R, s.t. for every other $\mathcal{NP}$-relation Q, there is a Cook reduction from f_Q to f_R. Formally:

Definition 10.8 *($\#\mathcal{P}$-complete) : f is $\#\mathcal{P}$-complete if*

1. *f is in $\#\mathcal{P}$.*
2. *For every g in $\#\mathcal{P}$, g Cook reduces to f.*

With Occam's Razor in mind, we'll try to find a complete problem, such that all other problems are reducible to it using a very simple form of reduction. Note that by restricting the kind of reductions we allow, we may rule out candidates for $\#\mathcal{P}$-complete problems. We take a restricted form of a Levin reduction ϕ from f_Q to f_R:

$$\forall x \in \Sigma^\star : f_Q(x) = f_R(\phi(x))$$

By allowing only this kind of reduction, we can find out several things about our candidates for #$\mathcal{P}$-complete problems. For example:

$$f_Q(x) \geq 1 \Leftrightarrow f_R(\phi(x)) \geq 1$$

In other words :

$$x \in L_Q \Leftrightarrow \phi(x) \in L_R$$

Which means that ϕ is a Karp reduction from L_Q to L_R. This implies that the decision problem related to R must be $\mathcal{NP}$-complete. Moreover, we require that the reduction preserves the number of witnesses for every input x. We capture this notion in the following definition:

Definition 10.9 *(*Parsimonious*) : A reduction $\phi : \Sigma^\star \to \Sigma^\star$, is* Parsimonious *w.r.t. $\mathcal{NP}$-relations Q and R if for every x : $|\{y : (x, y) \in Q\}| = |\{y : (\phi(x), y) \in R\}|$.*

Corollary 10.10 *if R is an NP-relation, and for every $\mathcal{NP}$-relation Q there exists $\phi_Q : \Sigma^\star \to \Sigma^\star$ s.t. ϕ_Q is parsimonious w.r.t. Q and R then f_R is #$\mathcal{P}$-complete.*

As we've said, a parsimonious reduction from f_Q to f_R must be a Karp reduction from L_Q to L_R. Therefore, we'll try to prove that the Karp reductions we used to prove SAT is $\mathcal{NP}$-complete, are also parsimonious, and thereby $\#SAT$ is #$\mathcal{P}$-complete.

Definition 10.11

$$R_{SAT} = \left\{ (\psi, \tau) \;\middle|\; \begin{array}{l} \psi \text{ is a boolean formula on variables } V(\psi) \\ \tau \text{ is a truth assignment for } V(\psi) : \psi(\tau) = 1 \end{array} \right\}$$

We have proved $SAT \stackrel{\text{def}}{=} L_{R_{SAT}}$ is $\mathcal{NP}$-complete by a series of Karp reductions. All we need to show is that each step is in fact a parsimonious reduction.

Theorem 10.12 *$\#SAT \stackrel{\text{def}}{=} f_{R_{SAT}}$ is #$\mathcal{P}$-complete.*

Proof: (outline)

1. Obviously $\#SAT$ is in #$\mathcal{P}$, since R_{SAT} is an NP-relation.

2. - The reduction from a generic $\mathcal{NP}$-relation R to Bounded-Halting, is parsimonious because the correspondence between the witnesses is not only one-to-one, it is in fact the identity.
   - The reduction from Bounded-Halting to Circuit-SAT consists of creating for each time unit a set of variables that can describe each possible configuration uniquely. Since a successful run is a specific list of configurations, and corresponds to one witness of Bounded-Halting, we get the same witness translated into one unique representation in binary variables.
   - In the reduction from Circuit-SAT to SAT we add extra variables for each internal gate in the circuit. Each satisfying assignment to the original circuit uniquely determines all the values in the internal gates, and therefore gives us exactly one satisfying assignment to the formula.

■

Notice that we actually proved that the counting problems associated with Bounded-Halting, Circuit-SAT, and SAT are $\#\mathcal{P}$-complete. Not only did we prove $\#SAT$ to be $\#\mathcal{P}$-complete, we also showed that for every f in $\#\mathcal{P}$, there exists a parsimonious reduction from f to $\#SAT$.

The reader might have gotten the impression that every NP-relation R, such that f_R is $\#\mathcal{P}$-complete implies L_R is $\mathcal{NP}$-complete. But the following theorem shows the contrary:

Theorem 10.13 *There exists an $\mathcal{NP}$-relation R s.t. f_R is $\#\mathcal{P}$-complete, and L_R is polynomial time decidable.*

Notice that such a $\#\mathcal{P}$-complete function, does not have the property that we showed $\#SAT$ has: Not all other functions in $\#\mathcal{P}$ have a parsimonious reduction to it. In fact it cannot be that every $\#\mathcal{P}$ problem has a Karp reduction to f_R, since otherwise L_R would be $\mathcal{NP}$-complete.

The idea in the proof is to modify a hard to calculate relation by adding easy to recognize witnesses to every input, so that the question of existence of a witness becomes trivial, yet the counting problem remains just as hard. Clearly, the $\#\mathcal{P}$-hardness will have to be proven by a non-parsimonious reduction (actually even a non-Karp reduction).

Proof: We define :

$$R'_{SAT} = \left\{ (\phi,(\tau,\sigma)) \;\middle|\; \begin{array}{c} (\phi(\tau)=1) \wedge (\sigma=1) \\ \vee \\ \sigma = 0 \end{array} \right\}$$

Obviously $L_{R'_{SAT}} = \Sigma^\star$, so it is in $\mathcal{P}$. But $f_{R'_{SAT}}$ is $\#\mathcal{P}$-complete, since for every ϕ : ϕ's witnesses in R'_{SAT} are:

$$\{(\tau,1) \;:\; \phi(\tau)=1\} \cup \{(\tau,0)\}$$

Which means:

$$\#SAT(\phi) + 2^{|\text{Variables}(\phi)|} = f_{R'_{SAT}}(\phi)$$

So given an oracle to $f_{R'_{SAT}}$ we can easily calculate $\#SAT$, meaning that $f_{R'_{SAT}}$ is $\#\mathcal{P}$-complete. ■

We proved the above theorem by constructing a somewhat unnatural NP-relation. We will now find a more natural problem that gives the same result (i.e., which is also $\#\mathcal{P}$-complete).

Definition 10.14 *(*Bipartite Graph*) : $G = (V_1 \cup V_2, E)$ is a* Bipartite Graph *if*

- $V_1 \cap V_2 = \emptyset$
- $E \subseteq V_1 \times V_2$

Definition 10.15 *(*Perfect Matching*) : Let $G = (V_1 \cup V_2, E)$ be a bipartite graph. A* Perfect Matching *is a set of edges $M \subseteq E$, that satisfies:*

1. *every v_1 in V_1 appears in exactly one edge of M.*
2. *every v_2 in V_2 appears in exactly one edge of M.*

Definition 10.16 *(*Perfect Matching — equivalent definition*) : Let $G = (V_1 \cup V_2, E)$ be a bipartite graph. A* Perfect Matching *is a one-to-one and onto function $f : V_1 \to V_2$ s.t. for every v in V_1, $(v, f(v)) \in E$.*

Proof: *(equivalence of definitions) :*

- Assume we have a subset of edges $M \subseteq E$ that satisfies the first definition. Define a function $f : V_1 \to V2$:

  $$f(v_1) = v_2 \iff (v_1, v_2) \in M$$

  f is well defined, because each v_1 in V_1 appears in exactly one edge of M. It is one-to-one and onto because each v_2 in V_2 appears in exactly one edge of M. Since $M \subseteq E$, f satisfies the condition that for all v_1 in V_1 : $(v_1, f(v_1))$ is in E.

- Assume we have a one-to-one and onto function $f : V_1 \to V_2$ that satisfies the above condition. We construct a set $M \subseteq E$:

  $$M = \{(v_1, f(v_1)) : v_1 \in V_1\}$$

  $M \subseteq E$ because for every v_1 in V_1 we know that $(v_1, f(v_1))$ is in E. The two conditions are also satisfied:

  1. Since f is a function, every v_1 in V_1 appears in exactly one edge of M.
  2. Since f is one-to-one and onto, every v_2 in V_2 appears in exactly one edge of M.

■

Definition 10.17 *$R_{PM} = \{(G, f) : G$ is a bipartite graph and f is a perfect matching of $G\}$*

Fact 10.2.1 *$L_{R_{PM}}$ is polynomial time decidable.*

The idea of the algorithm is to reduce the problem to a network-flow problem which is known to have a polynomial time algorithm. Given a bipartite graph $G = (V_1 \cup V_2, E)$, we construct a directed graph $G' = (V_1 \cup V_2 \cup \{s, t\}, E')$, so that:

$$E' = E \cup \{(s, v_1) : v_1 \in V_1\} \cup \{(v_2, t) : v_2 \in V_2\}$$

where E is viewed as directed edges from V_1 to V_2. What we did, is add a source s and connect it to one side of the graph, and a sink t connected to the other side. We transform it to a flow problem by setting a weight of 1 to every edge in the graph. There is a one to one correspondence between partial matchings and integer flows in the graph: Edges in the matching correspond to edges in E having a flow of 1. Therefore, there exists a perfect matching iff there is a flow of size $|V_1| = |V_2|$.

Theorem 10.18 *$f_{R_{PM}}$ is #$\mathcal{P}$-complete.*

This result is proved by showing that the problem of computing the permanent of a $\{0, 1\}$ matrix is #$\mathcal{P}$-complete. We will show the reduction from counting the number of perfect matchings to computing the permanent of such matrices. In fact, the two problems are computationally equivalent.

Definition 10.19 *(*Permanent*) : The* permanent *of an $n \times n$ matrix $A = (a_{i,j})_{i,j=1}^{n}$ is:*

$$Perm(A) = \sum_{\pi \in S_n} \prod_{i=1}^{n} a_{i,\pi(i)}$$

Where $S_n = \{\pi : \pi$ is a permutation of $\{1, \ldots, n\}\}$.

Note that the definition of the permanent of a matrix closely resembles that of the determinant of a matrix. In the definition of determinant, we have the same sum and product, except that each element in the sum is multiplied by the $sign \in \{-1, 1\}$ of the permutation π. Yet, computing the determinant is in $\mathcal{P}$, while computing the permanent is $\#\mathcal{P}$-complete, and therefore is believed not to be in $\mathcal{P}$. The main result in this section is the (unproven here) theorem:

Theorem 10.20 *$Perm$ is $\#\mathcal{P}$-complete.*

To show the equivalence of computing $f_{R_{PM}}$ and $Perm$, we use:

Definition 10.21 *(*Bipartite Adjacency Matrix*) : Given a bipartite graph $G = (V_1 \cup V_2, E)$, where $V_1 = \{1, \ldots, n\}$, and $V_2 = \{1, \ldots, m\}$, we define the* Bipartite Adjacency Matrix *of the graph G, as an $n \times m$ matrix $B(G)$, where :*

$$B(G)_{i,j} = \begin{cases} 1 & (i,j) \in E \\ 0 & otherwise \end{cases}$$

Proposition 10.2.2 *Given a bipartite graph $G = (V_1 \cup V_2, E)$ where $|V_1| = |V_2|$,*

$$f_{R_{PM}}(G) = Perm(B(G))$$

Proof:

$$\begin{aligned} Perm(B(G)) &= |\{\pi \in S_n : \textstyle\prod_{i=1}^{n} b_{i,\pi(i)} = 1\}| \\ &= |\{\pi \in S_n : \forall i \in \{1, \ldots, n\}, b_{i,\pi(i)} = 1\}| \\ &= |\{\pi \in S_n : \forall i \in \{1, \ldots, n\}, (i, \pi(i)) \in E\}| \\ &= |\{\pi \in S_n : \pi \text{ is a perfect matching in } G\}| \\ &= f_{R_{PM}}(G) \end{aligned}$$

■

We just showed that the problem of counting the number of perfect matchings in a bipartite graph Cook reduces to the problem of calculating the permanent of a $\{0, 1\}$ matrix. Notice that the other direction is also true by the same proof: Given a $\{0, 1\}$ matrix, create the bipartite graph that corresponds to it.

Now we will show another graph counting problem, that is equivalent to both of these:

Definition 10.22 *(*Cycle Cover*) : A* Cycle Cover *of a directed graph G, is a set of vertex disjoint simple cycles that cover all the vertices of G. More formally: $C \subseteq E$ is a* cycle cover *of G if for every connected component V_1 of $G' = (V, C)$, there is an ordering $V_1 = \{v_0, \ldots v_{d-1}\}$ s.t. $(v_i, v_j) \in C \Leftrightarrow j = i + 1 (mod\ d)$*

Notice that there is no problem with connected components of size 1, because we allow self loops.

Definition 10.23 *$\#Cycle(G)$ = number of cycle covers of G.*

Definition 10.24 *(*Adjacency Matrix*) : The* Adjacency Matrix *of a directed graph $G = (\{1, \ldots, n\}, E)$ is an $n \times n$ matrix $A(G)$:*

$$A(G)_{i,j} = \begin{cases} 1 & (i,j) \in E \\ 0 & otherwise \end{cases}$$

Proposition 10.2.3 *For every directed graph G, $Perm(A(G)) = \#Cycle(G)$*

In proving this proposition we use the following:

Claim 10.2.4 *C is a cycle cover of G if and only if every $v \in V$ has an out-degree and in-degree of 1 in $G' = (V, C)$.*

Proof: (Claim)

- ($\Longrightarrow$) Every vertex appears in exactly one cycle of C, because the cycles are disjoint. Also, since the cycles are simple, every vertex has an in-degree and out-degree of 1.
- ($\Longleftarrow$) For every connected component V_0 of G', take a vertex $v_0 \in V_0$, and create the directed path : $v_0, v_1, \ldots,$, where for every i, $(v_i, v_{i+1}) \in C$. Since the out-degree of every vertex is 1 in G', this path is uniquely determined, and:

  1. There must exist a vertex v_i that appears twice : $v_i = v_j$. Because V is finite.
  2. We claim that the least such i is 0. Otherwise, the in-degree of v_i is greater than 1.
  3. All the vertices of V_0 appear in our path because it is a connected component of G'.

  Thus each V_0 induces a directed cycle, and so G' is a collection of disjoint directed cycles which cover all V.

■

Proof: (Proposition) We'll define:

$$\Omega \stackrel{\text{def}}{=} \{\pi \in S_n : \forall i \in \{1, \ldots, n\}, (i, \pi(i)) \in E\}$$

It is easy to see that $Perm(A(G)) = |\Omega|$. Every $\pi \in \Omega$ defines

$$C_\pi = \{(i, \pi(i)) : i \in \{1, \ldots, n\}\} \subseteq E$$

and since π is a 1-1 and onto $\{1, \ldots, n\}$, the out-degree and in-degree of each vertex in C_π is 1. So C_π is a cycle cover of G. On the other hand, every cycle cover $C \subseteq G$ defines a mapping: $\pi_C(i) = j$ s.t. $(i, j) \in C$, and by the above claim, this is a permutation. ■

10.3 How close is $\#\mathcal{P}$ to $\mathcal{NP}$?

The main purpose of this lecture, is to study the class $\#\mathcal{P}$, and classify it as best as we can among other complexity classes we've studied. We've seen some examples of $\#\mathcal{P}$ complete problems. We also gave upper and lower complexity bounds on $\#\mathcal{P}$:

$$\mathcal{NP} \; \alpha_c \; \#\mathcal{P} \; \alpha_c \; \mathcal{P}\text{SPACE}$$

We will now try to refine these bounds by showing that $\#\mathcal{P}$ is not as far from $\mathcal{NP}$ as one might suspect. In fact, a counting problem in $\#\mathcal{P}$ can be probabilistically approximated in polynomial time using an NP oracle.

10.3.1 Various Levels of Approximation

We will start by introducing the notion of a range problem. A range problem is a relaxation of the problem of calculating a function. Instead of requiring one value for each input, we allow a full range of answers for each input.

Definition 10.25 *(*Range Problem*) : A* Range Problem Π *is defined by two functions* $\Pi = (\Pi_l, \Pi_u)$. $\Pi_l, \Pi_u : \Sigma^\star \to \mathsf{N}$. *s.t. on input* $x \in \Sigma^\star$, *the problem is to find* $t \in (\Pi_l(x), \Pi_u(x))$, *or in other words, return an integer* t, *s.t.* $\Pi_l(x) < t < \Pi_u(x)$.

Note that there is no restriction on the functions Π_l and Π_u, they can even be non-recursive. Since we are going to use range problems to denote an approximation to a function, we define a specific kind of range problems that are based on a function:

Definition 10.26 *(*Strong Range*) : For* $f : \Sigma^\star \to \mathsf{N}$, *and a polynomial* $p(\cdot)$, *we define the range problem* $StrongRange_p(f) = (l, u)$ *where:*

$$l(x) = f(x) \cdot (1 - \tfrac{1}{p(|x|)})$$
$$u(x) = f(x) \cdot (1 + \tfrac{1}{p(|x|)})$$

Strong range captures our notion of a good approximation. We will proceed in a series of reductions that will eventually give us the desired result. The first result we prove, is that it is enough to strongly approximate $\#SAT$.

Proposition 10.3.1 *If we can approximate* $\#SAT$ *strongly we can just as strongly approximate any* f *in* $\#\mathcal{P}$. *In other words : For every* f *in* $\#\mathcal{P}$, *and every polynomial* $p(\cdot)$,

$$StrongRange_p(f) \ \alpha_c \ StrongRange_p(\#SAT)$$

Proof: As we've seen, for every f in $\#\mathcal{P}$, there is a parsimonious reduction ϕ_f w.r.t. f and $\#SAT$. Meaning, for all x : $f(x) = \#SAT(\phi_f(x))$. We may assume that $|\phi_f(x)| > |x|$, because we can always pad $\phi_f(x)$ with something that will not change the number of witnesses:

$$\phi_f(x) \wedge z_1 \wedge z_2 \wedge \ldots \wedge z_{|x|}$$

We now use our oracle to $StrongRange_p(\#SAT)$ on $\phi_f(x)$, and get a result t, that satisfies :

$$\begin{array}{rcl} t & \in & (1 \pm \frac{1}{p(|\phi(x)|)}) \cdot \#SAT(\phi_f(x)) \\ & \subseteq & (1 \pm \frac{1}{p(|x|)}) \cdot f(x) \end{array}$$

■

We now wish to define a weaker form of approximation:

Definition 10.27 *(*Constant Range*) : For* $f : \Sigma^\star \to \mathsf{N}$, *and a constant* $c > 0$, *we define the range problem* $ConstantRange_c(f) = (l, u)$ *where:*

$$l(x) = \tfrac{1}{c} \cdot f(x)$$
$$u(x) = c \cdot f(x)$$

We want to show that approximating $\#SAT$ up to a constant suffices to approximate $\#SAT$ strongly. We'll in fact prove a stronger result: that an even weaker form of approximation is enough to approximate $\#SAT$ strongly.

Definition 10.28 *(*Weak Range*) : For $f : \Sigma^\star \to \mathsf{N}$, and a constant $\epsilon > 0$, we define the range problem $WeakRange_\epsilon(f) = (l, u)$ where:*

$$l(x) = (\tfrac{1}{2})^{|x|^{1-\epsilon}} \cdot f(x)$$
$$u(x) = 2^{|x|^{1-\epsilon}} \cdot f(x)$$

It is quite clear that $ConstantRange$ is a stronger form of approximation than $WeakRange$:

Claim 10.3.2 *For every $0 < \epsilon < 1$ and $c > 0$:*

$$WeakRange_\epsilon(\#SAT) \ \alpha_c \ ConstantRange_c(\#SAT)$$

Proof: Simply because for large enough n:

$$(\frac{1}{2}, 2)^{n^{1-\epsilon}} \subseteq (\frac{1}{c}, c)$$

where we use $(\frac{1}{2}, 2)^{n^{1-\epsilon}}$ to denote the range $((\frac{1}{2})^{n^{1-\epsilon}}, 2^{n^{1-\epsilon}})$. ■

Now we prove the main result:

Proposition 10.3.3 *For every polynomial $p(\cdot)$, and constant $0 < \epsilon < 1$,*

$$StrongRange_p(\#SAT) \ \alpha_c \ WeakRange_\epsilon(\#SAT)$$

Proof: Given ϕ, a boolean formula on variables $\vec{x}$. Define a polynomial $q(n) \geq (2n \cdot p(n))^{\frac{1}{\epsilon}}$, and build ϕ' :

$$\phi' = \bigwedge_{i=1}^{q(|\phi|)} \phi(\vec{x^i})$$

Where each $\vec{x^i}$ is a distinct copy of the variables $\vec{x}$. Obviously $\#SAT(\phi') = (\#SAT(\phi))^{q(|\phi|)}$. Notice $|\phi'| \leq 2|\phi| \cdot q(|\phi|)$. Now, assuming we have an oracle to $WeakRange_\epsilon(\#SAT)$, we call it on ϕ' to get:

$$\begin{array}{rcl} t & \in & (\frac{1}{2}, 2)^{|\phi'|^{1-\epsilon}} \cdot \#SAT(\phi') \\ & \subseteq & (\frac{1}{2}, 2)^{2|\phi| \cdot q(|\phi|)^{1-\epsilon}} \cdot (\#SAT(\phi))^{q(|\phi|)} \end{array}$$

Our result would be $s = t^{\frac{1}{q(|\phi|)}}$. And we have :

$$\begin{array}{rcl} s & \in & (\frac{1}{2}, 2)^{\frac{2|\phi|}{q(|\phi|)^\epsilon}} \cdot \#SAT(\phi) \\ & \subseteq & (\frac{1}{2}, 2)^{\frac{2|\phi|}{2|\phi|p(|\phi|)}} \cdot \#SAT(\phi) \\ & = & (\frac{1}{2}, 2)^{\frac{1}{p(|\phi|)}} \cdot \#SAT(\phi) \\ & \subseteq & (1 \pm \frac{1}{p(|\phi|)}) \cdot \#SAT(\phi) \end{array}$$

where the last containment follows from :

$$\forall x \geq 1 : (1 - \frac{1}{x})^x \leq \frac{1}{2} \ \text{ and } \ 2 \leq (1 + \frac{1}{x})^x$$

■

After a small diversion into proving a stronger result than needed, we conclude that all we have to do is to find a constant $c > 0$, such that we can solve $ConstantRange_c(\#SAT)$.

We still have a hard time solving the problem directly, so we'll do yet another reduction into a relaxed form of decision problems, called promise problems. While machines that solve decision problems are required to give an exact answer for every input, promise problems are only required to do so on a predefined 'promise' set.

Definition 10.29 *(*Promise Problem*) : A* Promise Problem $\Pi = (\Pi_Y, \Pi_N)$*, where* $\Pi_Y, \Pi_N \subseteq \Sigma^\star$*, and* $\Pi_Y \cap \Pi_N = \emptyset$*, is the question: Given* $x \in Promise(\Pi) \stackrel{\text{def}}{=} \Pi_Y \cup \Pi_N$*, decide whether* $x \in \Pi_Y$*.*

Notice that if $x \notin Promise(\Pi)$, there is no requirement. Also, promise problems are a generalization of decision problems, where in decision problems $Promise(\Pi) = \Sigma^\star$, so no promise is made.

Definition 10.30 *(*$Gap_8\#SAT$*) The promise problem* $Gap_8\#SAT = (Gap_8\#SAT_Y, Gap_8\#SAT_N)$*, where:*

$$\begin{aligned} Gap_8\#SAT_Y &= \{(\phi, K) : \#SAT(\phi) > 8K\} \\ Gap_8\#SAT_N &= \{(\phi, K) : \#SAT(\phi) < \tfrac{1}{8}K\} \end{aligned}$$

We now continue in our reductions. For this we choose $c = 64$, and show we can solve $ConstantRange_{64}(\#SAT)$ using an oracle to $Gap_8\#SAT$.

Proposition 10.3.4 *$ConstantRange_{64}(\#SAT)$ Cook reduces to $Gap_8\#SAT$*

Proof: We run the following algorithm on input ϕ:

- $i = 0$
- While ($Gap_8\#SAT$ answers YES on $(\phi, 8^i)$) do $i = i + 1$
- return $8^{i-\frac{1}{2}}$

Denote $\alpha = \log_8(\#SAT(\phi))$. The result $8^{k-\frac{1}{2}}$, satisfies : $\alpha - 2 < k - \frac{1}{2} < \alpha + 2$, because :

- For all $i < \alpha - 1$, $\#SAT(\phi) > 8 \cdot 8^i$, so $(\phi, 8^i) \in Gap_8\#SAT_Y$. Therefore, we are promised that in such a case the algorithm will increment such an i, and not stop. So, $k \geq \alpha - 1$ follows.
- For all $i > \alpha + 1$, $\#SAT(\phi) < \frac{1}{8} \cdot 8^i$, so $(\phi, 8^i) \in Gap_8\#SAT_N$. Meaning that the algorithm must stop at the first such i or before. The first such i that satisfies $i > \alpha + 1$ also satisfies $i \leq \alpha + 2$. Therefore $k \leq \alpha + 2$.

Now :

$$\begin{array}{ccccc} \alpha - 1 & \leq & k & \leq & \alpha + 2 \\ & & \Downarrow & & \\ \alpha - 2 & < & k - \frac{1}{2} & < & \alpha + 2 \end{array}$$

We conclude:

$$8^{k-\frac{1}{2}} \in (\frac{1}{64}, 64) \cdot \#SAT(\phi)$$

■

So far we've shown the following reductions:

$$\begin{gathered} StrongRange_{\text{poly}}(\#\mathcal{P}) \ \alpha_c \ StrongRange_{\text{poly}}(\#SAT) \ \alpha_c \ WeakRange_{\epsilon}(\#SAT) \\ \alpha_c \ ConstantRange_{64}(\#SAT) \ \alpha_c \ Gap_8\#SAT \end{gathered}$$

Since Cook reductions are transitive, we get :

$$StrongRange_{\text{poly}}(\#\mathcal{P}) \ \alpha_c \ Gap_8\#SAT$$

We will show how to solve $Gap_8\#SAT$ using an oracle to SAT, but with a small probability of error. So we will show, that in general, if we can solve a problem P using an oracle to a promise

problem Q, then if we have an oracle to Q that makes little mistakes, we can solve P with high probability.

Comment : *(Amplification)* : For every promise problem P, and machine M that satisfies:

$$\text{for every } x \in Promise(P) : Prob[M(x) = P(x)] > \frac{2}{3}$$

If on input x that is in $Promise(P)$, we run M on x, $O(n)$ times, then the majority of the results will equal $P(x)$ with probability greater than $1 - 2^{-n}$.

This we proved when we talked about $\mathcal{BPP}$, and the proof stays exactly the same, using Chernoff's bound. Note that we do not care if machine M has an oracle or not, and if so how this oracle operates, as long as different runs of M are independent.

Proposition 10.3.5 *Given a problem P and a promise problem Q, such that P Cook reduces to Q, if we have a probabilistic machine Q' that satisfies:*

$$\textit{for every } x \in Promise(Q) : Prob[Q'(x) = Q(x)] > \frac{2}{3}$$

then for every polynomial $p(\cdot)$, we have a probabilistic polynomial time machine M that uses an oracle to Q', and satisfies:

$$Prob[M^{Q'}(y) \textit{ is a solution of } P \textit{ on input } y] > 1 - 2^{-p(|y|)}$$

Proof: We start by noticing that since the reduction from P to Q is polynomial, there exists a polynomial $q(\cdot)$, such that the oracle Q is called less than $q(|y|)$ times. Since we use Q' and not Q as an oracle, we have a probability of error. If each one of these calls had a probability of error less than : $\frac{1}{q(|y|)} \cdot 2^{-p(|y|)}$, then by using the union bound we would get that the probability that at least one of the oracle calls was incorrect is less than $2^{-p(|y|)}$. The probability of M being correct, is at least the probability that all oracle calls are correct, therefore in this case it is greater than $1 - 2^{-p(|y|)}$.

Using the comment about amplification, we can amplify the probability of success of each oracle call to $1 - \frac{1}{q(|y|)} 2^{-p(|y|)}$, by calling it $O(p(|y|) \cdot \log q(|y|))$ number of times, which is polynomial in the size of the input. ■

In conclusion, all we have to do is show that we can solve $Gap_8 \#SAT$ with a probability of error $< \frac{1}{3}$. Then, we showed that we can find a solution to $\#SAT$, that is very close to the real solution $(StrongRange_p(\#SAT))$, with a very high probability of success.

10.3.2 Probabilistic Cook Reduction

In the next sections, we extensively use the notion of probabilistic reduction. Therefore, we'll define it formally, and prove some of it's properties.

Definition 10.31 *(*Probabilistic Cook Reduction*) : Given promise problems P and Q, we say that there is a* Probabilistic Cook Reduction *from P to Q denoted $P \alpha_R Q$, if there is a probabilistic polynomial time oracle machine M that uses Q as an oracle, and satisfies:*

$$\textit{for every } x \in Promise(P) : Prob[M^Q(x) = P(x)] > \frac{2}{3}$$

where $M^Q(x)$ denotes then operation of machine M on input x when given oracle access to Q. Whenever a query to Q satisfies the promise of Q, the answer is correct, but when the query violates the promise the answer may be arbitrary.

Notice that in the definition, the oracle has no probability of error. We now show that this restriction does not matter, and we can do the same even if the oracle is implemented with bounded probability of error.

Proposition 10.3.6 *If P probabilistically Cook reduces to Q, and we have a probabilistic machine Q' that satisfies*

$$\text{for every } x \in Promise(Q) : Prob[Q'(x) = Q(x)] > \frac{2}{3}$$

then we have a probabilistic polynomial time oracle machine M that uses Q' as an oracle, and satisfies :

$$\text{for every } y \in Promise(P) : Prob[M^{Q'}(y) = P(y)] > \frac{2}{3}$$

Proof: By the definition of a probabilistic Cook reduction, we have a probabilistic polynomial time oracle machine N that satisfies:

$$\text{for every } y \in Promise(P) : Prob[N^{Q}(y) = P(y)] > \frac{3}{4}$$

Where we changed $\frac{2}{3}$ to $\frac{3}{4}$, using the comment about amplification. Machine N runs in polynomial time, therefore it calls the oracle a polynomial $p(|y|)$ number of times. We can assume Q' to be correct with a probability $> \frac{1}{9} \cdot \frac{1}{p(|y|)}$, by calling it each time instead of just once, $O(\log(p(|y|)))$ times, and taking the majority. Using the union bound, the probability that all oracle calls (to this modified Q') are correct is greater than $\frac{8}{9}$.

When all oracle calls are correct, machine N returns the correct result. Therefore with probability greater than $\frac{3}{4} \cdot \frac{8}{9} = \frac{2}{3}$ we get the correct result. ■

We list some properties of probabilistic cook reductions:

- Deterministic Cook reduction is a special case (i.e., $P\ \alpha_c\ Q \implies P\ \alpha_R\ Q$).
- *Transitivity* : $P\ \alpha_R\ Q\ \alpha_R\ R \implies P\ \alpha_R\ R$

10.3.3 $Gap_8\#SAT$ Reduces to SAT

Our goal, is to show that we can approximate any problem in $\#\mathcal{P}$ using an oracle to SAT. So far we've reduced the problem several times, and got:

$$StrongRange_{\text{poly}}(\#\mathcal{P})\ \alpha_c\ Gap_8\#SAT$$

Now we'll show:

$$Gap_8\#SAT\ \alpha_R\ SAT$$

And using the above properties of probabilistic Cook reductions, this will mean that we can approximate $\#\mathcal{P}$ very closely, with an exponentially small probability of error.

Reminder: $Gap_8\#SAT$ is the promise problem on input pairs (ϕ, k), where ϕ is a boolean formula, and k is a natural number. $Gap_8\#SAT = (Gap_8\#SAT_Y, Gap_8\#SAT_N)$, where:

$$\begin{aligned} Gap_8\#SAT_Y &= \{(\phi, k) : \#SAT(\phi) > 8k\} \\ Gap_8\#SAT_N &= \{(\phi, k) : \#SAT(\phi) < \tfrac{1}{8}k\} \end{aligned}$$

How do we approach the problem? We know, that there is either a very large or a very small number of truth assignment in comparison to the input parameter k. So if we take a random $\frac{1}{k}$

fraction of the assignments, with high probability in the first case at least one of them is satisfying, and in the second, none are. Assume that we have a way of restricting our formula to a random fraction of the assignments S that satisfies : each assignment τ is in the set with probability $\frac{1}{k}$ independently of all other assignments. We set $\phi'(\tau) = \phi(\tau) \wedge (\tau \in S)$. Then we simply check satisfiability of ϕ'. First notice:

$$Prob_S[\phi' \in SAT] \;=\; 1 - Prob_S[\forall \tau \text{ s.t. } \phi(\tau) = 1 \;:\; \tau \notin S] \;=\; 1 - (\tfrac{k-1}{k})^{\#SAT(\phi)}$$

Therefore:

$$\begin{array}{lllllll} \text{If } \#SAT(\phi) > 8k \text{ then} & Prob_S[\phi' \in SAT] & > & 1 - (\frac{k-1}{k})^{8k} & \approx & 1 - \frac{1}{e^8} & > \frac{2}{3} \\ \text{If } \#SAT(\phi) < \frac{1}{8}k \text{ then} & Prob_S[\phi' \in SAT] & < & 1 - (\frac{k-1}{k})^{\frac{1}{8}k} & \approx & 1 - \frac{1}{e^{\frac{1}{8}}} & < \frac{1}{3} \end{array}$$

The problem is, we don't have an efficient procedure to choose such a random S. So we weaken our requirements, instead of total independence, we require only pairwise independence. Specifically, we use the following tool:

Definition 10.32 (Universal$_2$ Hashing) *: A family of functions, $H_{n,m}$, mapping $\{0,1\}^n$ to $\{0,1\}^m$ is called* Universal$_2$ *if for a uniformly selected h in $H_{n,m}$, the random variables $\{h(e)\}_{e\in\{0,1\}^n}$ are pairwise independent and uniformly distributed over $\{0,1\}^m$. That is, for every $x \neq y \in \{0,1\}^n$, and $a, b \in \{0,1\}^m$,*

$$Prob_{h\in H_{n,m}}[h(x) = a \;\&\; h(y) = b] \;=\; (2^{-m})^2$$

An efficient construction of such families is required to have algorithms for selecting and evaluating functions in the family. That is,

1. *selecting:* There exists a probabilistic polynomial-time algorithm that on input $(1^n, 1^m)$, outputs a description of a uniformly selected function in $H_{n,m}$.

2. *evaluating:* There exists a polynomial-time algorithm that on input: a description of a function $h \in H_{n,m}$ and a domain element $x \in \{0,1\}^m$ outputs the value $h(x)$.

A popular example is the family of all affine transformations from $\{0,1\}^n$ to $\{0,1\}^m$. That is, all functions of the form $h_{A,b}(x) = Ax + b$, where A is an m-by-n 0-1 matrix, b is an m-dimensional 0-1 vector, and arithmetic is modulo 2. Clearly, this family has an efficient construction. In Appendix A, we will show that this family is Universal$_2$.

Lemma 10.3.7 (Leftover Hash Lemma): *Let $H_{n,m}$ be a family of Universal$_2$ Hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$, and let $\epsilon > 0$. Let $S \subseteq \{0,1\}^n$ be arbitrary provided that $|S| \geq \epsilon^{-3} \cdot 2^m$. Then:*

$$Prob_h[|\{e \in S : h(e) = 0^m\}| \in (1 \pm \epsilon) \cdot \frac{|S|}{2^m}] > 1 - \epsilon$$

The proof of this lemma appears in Appendix B.

We are now ready to construct a probabilistic Cook reduction from $Gap_8\#SAT$ to SAT, using a Universal$_2$ family of functions. Specifically we will use the family of affine transformations.

Theorem 10.33 *$Gap_8\#SAT \;\alpha_R\; SAT$*

Proof: We construct a probabilistic polynomial time machine M which is given oracle access to SAT. On input $(\phi, 2^m)$, where ϕ has n variables, M operates as follows:

1. Select uniformly $h \in H_{n,m} = \{\text{Affine transformations from } \{0,1\}^n \text{ to } \{0,1\}^m\}$. The function h is represented by a $\{0,1\}$ matrix $A_{m\times n} = (a_{i,j})_{\substack{i=1,...,m \\ j=1,...,n}}$ and a $\{0,1\}$ vector $b = (b_i)_{i=1,...,m}$.

2. We construct a formula ψ_h, on variables $x_1, ..., x_n, y_1, ..., y_t$, so that for every $x \in \{0,1\}^n$ $h(x) = 0^m$ iff there exists an assignment to the y_i's so that $\psi_h(x_1, ..., x_n, y_1, ..., y_t)$ is true. Furthermore, in case $h(x) = 0^m$, there is a unique assignment to the y_i's so that $\psi_h(x_1, ..., x_n, y_1, ..., y_t)$ is true.

   The construction of ψ_h can be presented in two ways. In the abstract way, we just observe that applying the standard Cook-reduction to the assertion $h(x) = 0^m$, results in the desired formula. (The claimed properties have to be verified indeed.) A more concrete way is to start by the following observations

$$\begin{array}{c} h(x_1, \ldots x_n) = 0^m \\ \Updownarrow \\ \bigwedge_{i=1}^m \left(\sum_{j=1}^n a_{i,j} x_j \equiv b_i \pmod 2 \right) \\ \Updownarrow \\ \bigwedge_{i=1}^m \left((b_i \oplus 1) \oplus \bigoplus_{j=1}^n (a_{i,j} \wedge x_j) \right) \end{array}$$

   Introducing auxiliary variables, as in the construction of the standard reduction from Circuit–Satisfiability to 3SAT, we obtain the desired formula ψ_h. For example, introducing variables $y_1, ..., y_n, y_{1,1}, ..., y_{m,n}$, the above formula is satisfied for a particular setting of the x_i's iff the following formula is satisfyiable for these x_i's (and furthermore for a unique setting of the y_i's):

$$\bigwedge_{i=1}^m (b_i \oplus 1 \oplus y_i) \ \wedge \ \bigwedge_{i=1}^m \left(y_i = \bigoplus_{j=1}^n y_{i,j} \right) \ \wedge \ \bigwedge_{i=1}^m \bigwedge_{j=1}^n (y_{i,j} = a_{i,j} \wedge x_j)$$

   So all that is left is to write a CNF for $\bigoplus_{j=1}^n y_{i,j}$, by using additional auxiliary variables. To write a CNF for $\bigoplus_{j=1}^n z_j$, we look at a binary tree of depth $\ell \stackrel{\text{def}}{=} \log_2 n$ which computes the XOR in the natrual way. We introduce an auxiliary variable for each internal node, and obtain

$$w_{0,1} \ \wedge \ \bigwedge_{i=0}^{\ell-1} \bigwedge_{j=1}^{2^i} (w_{i,j} = w_{i+1,2j-1} \oplus w_{i+1,2j}) \ \wedge \ \bigwedge_{j=1}^n (w_{\ell,j} = z_j)$$

3. Define $\phi' = \phi \wedge \psi_h$. Use our oracle to SAT on ϕ', and return the result.

The validity of the reduction is established via the following two claims.

Claim 1: If $(\phi, 2^m) \in Gap_8\#SAT_Y$ then $\phi' \in SAT$ with probability $> \frac{1}{2}$.

Claim 2: If $(\phi, 2^m) \in Gap_8\#SAT_N$ then $\phi' \in SAT$ with probability $< \frac{1}{8}$.

Before proving these claims, we note that the gap in the probabilities in the two cases (i.e., $(\phi, 2^m) \in Gap_8\#SAT_Y$ and $(\phi, 2^m) \in Gap_8\#SAT_N$) can be "amplified" to obtain the desired probabilities (i.e., $\phi' \in SAT$ with probability at least $2/3$ in the first case and at most $1/3$ in the second).

Proof Claim 1: We define $S_\phi \stackrel{\text{def}}{=} \{x : \phi(x) = 1\}$. Because $(\phi, 2^m) \in Gap_8\#SAT_Y$, we know that $|S_\phi| > 8 \cdot 2^m$ Now:

$$\begin{array}{rcl} Prob_h[\phi' \in SAT] & = & Prob_h[\{x : \phi(x) = 1 \ \& \ h(x) = 0^m\} \neq \emptyset] \\ & = & Prob_h[\{x \in S_\phi : h(x) = 0^m\} \neq \emptyset] \\ & \geq & Prob_h[|\{x \in S_\phi : h(x) = 0^m\}| \in (1 \pm \frac{1}{2}) \frac{|S_\phi|}{2^m}] \ > \ \frac{1}{2} \end{array}$$

The last inequality is an application of the Leftover Hash lemma, setting $\epsilon = \frac{1}{2}$, and the claim follows. □

Proof Claim 2: As $(\phi, 2^m) \in Gap_8\#SAT_N$, we have $|S_\phi| < \frac{1}{8} \cdot 2^m$.

$$\begin{array}{rcl} Prob_h[\phi' \in SAT] & = & Prob_h[\{x \in S_\phi : h(x) = 0^m\} \neq \emptyset] \\ & = & Prob_h[(\bigcup_{x \in S_\phi} \{x : h(x) = 0^m\}) \neq \emptyset] \\ & \leq & \sum_{x \in S_\phi} Prob_h[h(x) = 0^m] \\ & < & \frac{1}{8} \cdot 2^m \cdot 2^{-m} \quad = \quad \frac{1}{8} \end{array}$$

The last inequality uses the union bound, and the claim follows. □

Combining the two claims (and using amplification), the theorem follows. ■

In conclusion, we have shown:

$$StrongApprox_{\text{poly}}(\#\mathcal{P}) \ \alpha_c \ Gap_8\#SAT \ \alpha_R \ SAT$$

Which is what we wanted.

10.4 Reducing to *uniqueSAT*

We've introduced the notion of *promise problems* as a means to prove that we can approximate $\#SAT$ using SAT. But promise problems are interesting by their own right, so we will try to investigate them a bit more. We've shown that using an oracle to SAT we can solve $Gap_8\#SAT$. The converse is also true, because we've shown we can approximate (deterministically) $\#SAT$ using $Gap_8\#SAT$, so all we have to do is approximate well enough, to differentiate 0 from positive results, and thus, solve SAT. We will try to refine this result, by showing that a more restricted version of $Gap_8\#SAT$ is enough to solve SAT (and even approximate $\#SAT$).

Definition 10.34 *$Gap_8\#SAT'$ is the promise problem on input pairs (ϕ, k) defined by:*

$$\begin{array}{rcl} Gap_8\#SAT'_Y & = & \{(\phi, k) : 8k < \#SAT(\phi) < 32k\} \\ Gap_8\#SAT'_N & = & \{(\phi, k) : \#SAT(\phi) < \frac{1}{8}k\} \end{array}$$

Claim 10.4.1 *SAT Cook reduces to $Gap_8\#SAT'$*

Proof: Given ϕ, first we will create formula ϕ', s.t. $\#SAT(\phi') = 15 \cdot \#SAT(\phi)$. Take 4 variables $\{x_1, x_2, x_3, x_4\}$ not appearing in ϕ. and define:

$$\begin{array}{rcl} \psi & = & (x_1 \vee x_2 \vee x_3 \vee x_4) \\ \phi' & = & \phi \wedge \psi \end{array}$$

Observe that $\#SAT(\psi) = 15$, and since the variables of ψ do not appear in ϕ, the above equality holds. So we know that :

$$\begin{array}{rcl} \#SAT(\phi') \geq 15 & \iff & \phi \in SAT \\ \#SAT(\phi') = 0 & \iff & \phi \notin SAT \end{array}$$

For every $0 \leq i \leq |Variables(\phi')|$, we call our oracle: $Gap_8\#SAT'(\phi', 2^i)$. We claim : One of the answers is YES iff $\phi \in SAT$.

- Suppose that $\phi \notin SAT$. Then $\#SAT(\phi') = 0 < \frac{1}{8}k$ for all $k > 0$, therefore for all i, $(\phi', 2^i) \in Gap_8\#SAT'_N$, so we are promised to always get a NO answer.

- Suppose $\phi \in SAT$, so as we showed, $\#SAT(\phi') \geq 15$. Therefore, $\log_2(\#SAT(\phi')) \geq \log_2(15) > 3$. There exists an integer $i \geq 0$ s.t.

$$\begin{array}{c} i < \log_2(\#SAT(\phi')) - 3 < i + 2 \\ \Downarrow \\ 2^{i+3} < \#SAT(\phi') < 2^{i+5} \\ \Downarrow \\ 8 \cdot 2^i < \#SAT(\phi') < 32 \cdot 2^i \end{array}$$

  And for that i, we are guaranteed to get a YES answer.

■

The reader may wonder why we imposed this extra restriction on $Gap_8\#SAT$. We want to show that we can solve SAT using weak oracles. For example $Gap_8\#SAT'$ is a weak oracle. But we wish to continue in our reductions, and our next step is:

Definition 10.35 *$fewSAT$ is the promise problem defined by:*

$$\begin{array}{lcll} fewSAT_Y & = & \{\phi : 1 \leq \#SAT(\phi) < 100\} & \subset \quad \overline{SAT} \\ fewSAT_N & = & \{\phi : \#SAT(\phi) = 0\} & = \quad \overline{SAT} \end{array}$$

Proposition 10.4.2 *$Gap_8\#SAT'$ probabilistically Cook reduces to $fewSAT$*

Proof: We will use the same reduction we used when proving $Gap_8\#SAT \; \alpha_R \; SAT$, except we now have $Gap_8\#SAT'$. Recall, we uniformly select $h \in H_{n,m}$, and construct $\phi'(x) = \phi(x) \wedge (h(x) = 0^m)$.

We make analogous claims to the ones stated in the former proof:

- *claim 1* : If $(\phi, 2^m) \in Gap_8\#SAT'_Y$ then $\phi' \in fewSAT$ with probability $> \frac{1}{2}$.

- *claim 2* : If $(\phi, 2^m) \in Gap_8\#SAT'_N$ then $\phi' \in fewSAT$ with probability $< \frac{1}{8}$.

1. Since $(\phi, 2^m) \in Gap_8\#SAT'_Y$, we have:

$$8 \cdot 2^m < |S_\phi \stackrel{\text{def}}{=} \{x : \phi'(x) = 1\}| < 32 \cdot 2^m$$

   So now:

$$\begin{array}{rcl} Prob_h[\phi' \in fewSAT] & = & Prob_h[0 < |\{x : \phi(x) = 1 \;\&\; h(x) = 0^m\}| < 100] \\ & = & Prob_h[0 < |\{x \in S_\phi : h(x) = 0^m\}| < 100] \\ & \geq & Prob_h[(1 - \frac{1}{2}) \cdot 8 < |\{x \in S_\phi : h(x) = 0^m\}| < (1 + \frac{1}{2}) \cdot 32] \\ & \geq & Prob_h[|\{x \in S_\phi : h(x) = 0^m\}| \in (1 \pm \frac{1}{2})\frac{|S_\phi|}{2^m}] > \frac{1}{2} \end{array}$$

2. In the original proof we showed: if $(\phi, 2^m) \in Gap_8\#SAT_N$ then ϕ' is not satisfiable with probability greater than $\frac{7}{8}$. Notice :$Gap_8\#SAT_N = Gap_8\#SAT'_N$, so if $(\phi, 2^m) \in Gap_8\#SAT'_N$ then ϕ' is not satisfiable with probability greater than $\frac{7}{8}$, and in that case, it's in $fewSAT_N$, so we are guaranteed to get a NO answer.

■

As a last step in this endless crusade to understand the complexity of SAT promise problems, we will show that the weakest SAT related promise problem, is in fact as strong as the others.

Definition 10.36 *uniqueSAT is the promise problem on input ϕ defined by:*

$$\begin{array}{rcccc} uniqueSAT_Y & = & \{\phi : \#SAT(\phi) = 1\} & \subset & SAT \\ uniqueSAT_N & = & \{\phi : \#SAT(\phi) = 0\} & = & \overline{SAT} \end{array}$$

Proposition 10.4.3 *$fewSAT$ Cook reduces to $uniqueSAT$*

Proof: Given a formula ϕ, we want to solve $fewSAT$. For each $1 \leq i < 100$ we construct a formula ϕ_i, s.t. :

- $\phi \notin SAT \Rightarrow \phi_i \notin SAT$.
- ϕ_i has a unique satisfying assignment if ϕ has exactly i satisfying assignments.

If we can do this, we can check all these ϕ_i's, with our oracle to $uniqueSAT$. If all of them are NO, then we return NO, otherwise we answer YES. This is correct because if $0 < k \stackrel{\text{def}}{=} \#SAT(\phi) < 100$, then ϕ_k has exactly one satisfying assignment, and therefore $uniqueSAT$ returns YES on ϕ_k. Also, if $\phi \notin SAT$, then all for all i : $\phi_i \in uniqueSAT_N$, so all the answers must be NO.

All that is left is to construct ϕ_i: We create i copies of ϕ, each on a separate set of variables:

$$\psi_i = \wedge_{j=1}^{i} \phi(x_1^j, \ldots, x_n^j)$$

First notice, that if $\phi \notin SAT$, then so is ψ_i. Now assume $\#SAT(\phi) = i$. Every satisfying assignment of ψ_i, corresponds to i satisfying assignments of ϕ. But we want to force them to be different, so we would require that the assignments are different and add this requirement to ψ_i. But then, we will have exactly $i!$ satisfying assignments to the new ψ_i. To solve this, instead of just requiring that they are different, we will impose a lexicographical ordering of the solutions, which will fix one satisfying assignment from the $i!$ possible.

$$\phi_i = \psi_i \wedge \bigwedge_{j=1}^{i-1} (\vec{x^j} <_{lex} \vec{x^{j+1}})$$

■

Just for the heck of it, we'll list all the reductions in order:

$$\begin{array}{c} StrongRange_{\text{poly}}(\#\mathcal{P}) \ \alpha_c \\ StrongRange_{\text{poly}}(\#SAT) \ \alpha_c \\ WeakRange_{\epsilon}(\#SAT) \ \alpha_c \\ ConstantRange_{64}(\#SAT) \ \alpha_c \\ Gap_8\#SAT \ \alpha_R \\ SAT \ \alpha_c \\ Gap_8\#SAT' \ \alpha_R \\ fewSAT \ \alpha_c \\ uniqueSAT \end{array}$$

Some collapsing gives us:

$$StrongRange_{\text{poly}}(\#\mathcal{P}) \ \alpha_c \ Gap_8\#SAT \ \alpha_R \ uniqueSAT$$

Bibliographic Notes

The counting class $\#\mathcal{P}$ was introduced by Valiant [4], who proved that computing the permanent of 0-1 matrices is $\#\mathcal{P}$-complete. Valiant's proof first establishes the $\#\mathcal{P}$-hardness of computing the permanent of integer matrices (the entries are actually restricted to $\{-1, 0, 1, 2, 3\}$), and next reduces the computation of the permanent of integer matrices to the the permanent of 0-1 matrices. A de-constructed version of Valinat's proof can be found in [1].

The approximation procedure for $\#\mathcal{P}$ is due to Stockmeyer [3], following an idea of Sipser [2]. Our exposition follows further developments in the area. The randomized reduction of SAT to uniqueSAT is due to Valiant and Vazirani [5]. Again, our exposition is a bit different.

1. A. Ben-Dor and S. Halevi. Zeo-One Permanent is #P-Complete, A Simpler Proof. In *2nd Israel Symp. on Theory of Computing and Systems* (*ISTCS93*), IEEE Computer Society Press, pages 108–117, 1993.

2. M. Sipser. A Complexity Theoretic Approach to Randomness. In *15th STOC*, pages 330–335, 1983.

3. L. Stockmeyer. The Complexity of Approximate Counting. In *15th STOC*, pages 118–126, 1983.

4. L.G. Valiant. The Complexity of Computing the Permanent. *Theoretical Computer Science*, Vol. 8, pp. 189–201, 1979.

5. 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.

Appendix A: A Family of Universal$_2$ Hash Functions

In this appendix we show that the family of affine transformations from $\{0,1\}^n$ to $\{0,1\}^m$ is efficiently constructible and is Universal$_2$.

1. *selecting:* Simply selecting uniformly and independently each bit of A and b, will output a uniformly selected affine transformation. This runs in $O(nm + m)$ time, which is polynomial in the length of the input.

2. *evaluating:* Calculating Ax takes $O(mn)$ time, and the addition of b adds $O(m)$ time. All in all, polynomial in the size of the input.

Proposition: *The family of affine transformations from* $\{0,1\}^n$ *to* $\{0,1\}^m$ *is Universal*$_2$.

Proof: Given $x_1 \neq x_2 \in \{0,1\}^n$, and $y_1, y_2 \in \{0,1\}^m$. If $x_1 = 0^n$, then

$$\begin{aligned} Prob_{A,b}[h(x_1) = y_1 \ \& \ h(x_2) = y_2] &= Prob_{A,b}[b = y_1 \ \& \ Ax_2 + b = y_2] \\ &= Prob_{A,b}[b = y_1 \ \& \ Ax_2 = y_2 - y_1] \\ &= Prob_A[Ax_2 = y_2 - y_1] \cdot Prob_b[b = y_1] \\ &= 2^{-m} \cdot 2^{-m} = (2^{-m})^2 \end{aligned}$$

Where $Prob[Ax_2 = y_2 - y_1] = 2^{-m}$, because for a given vector $x_2 \neq 0^n$, a uniformly chosen linear transformation A, maps x_2 uniformly into $\{0,1\}^m$. If $x_2 = 0$ the same argument holds. Assume

both are different than 0^m. Since we choose among the linear transformations uniformly, it does not matter in what base we represent them. Since $x_1, x_2 \neq 0$, and they are both in $\{0,1\}^n$, they must be linearly independent. So we may assume they are both base vectors in the representation of A, meaning one column in A : column a_1 in A represents the image of x_1, and a different column a_2 represents the image of x_2.

$$\begin{aligned} Prob_{A,b}[h(x_1) = y_1 \ \& \ h(x_2) = y_2] &= Prob_{A,b}[Ax_1 + b = y_1 \ \& \ Ax_2 + b = y_2] \\ &= Prob_{a_1,a_2,b}[a_1 + b = y_1 \ \& \ a_2 + b = y_2] \\ &= Prob_{a_1,a_2}[a_1 = y_1 - b \ \& \ a_2 = y_2 - b] \qquad \text{(for every } b\text{)} \\ &= Prob_{a_1}[a_1 = y_1 - b] \cdot Prob_{a_2}[a_2 = y_2 - b] \quad \text{(for every } b\text{)} \\ &= 2^{-m} \cdot 2^{-m} = (2^{-m})^2 \end{aligned}$$

■

Appendix B: Proof of Leftover Hash Lemma

In this appendix, we prove the Leftover Hash Lemma (Lemma 10.3.7). We first restate the lemma.

The Leftover Hash Lemma: Let $H_{n,m}$ be a family of Universal$_2$ Hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$, and let $\epsilon > 0$. Let $S \subseteq \{0,1\}^n$ be arbitrary provided that $|S| \geq \epsilon^{-3} \cdot 2^m$. Then:

$$Prob_h[|\{e \in S : h(e) = 0^m\}| \in (1 \pm \epsilon) \cdot \frac{|S|}{2^m}] > 1 - \epsilon$$

Proof: We define for each $e \in \{0,1\}^n$ a random variable X_e :

$$X_e = \begin{cases} 1 & h(e) = 0^m \\ 0 & \text{otherwise} \end{cases}$$

For each $e_1 \neq e_2 \in \{0,1\}^n$, we claim that X_{e_1}, X_{e_2} are stochastically independent, because they are functions of the independent random variables $h(e_1)$ and $h(e_2)$ respectively. That is, we use the known fact by which if X and Y are independent random variables then, for every function f, $f(X)$ and $f(Y)$ are also independent random variables.

We compute :

$$\begin{aligned} E(x_e) &= Prob[X_e = 1] = \tfrac{1}{2^m} \\ VAR(X_e) &= Prob[X_e = 1] \cdot (1 - Prob[X_e = 1]) = \tfrac{1}{2^m}(1 - \tfrac{1}{2^m}) \end{aligned}$$

We define a new random variable $Y = \sum_{e \in S} X_e$. In other words : $Y = |\{e \in S : h(e) = 0^m\}|$. Since the X_e's are pairwise independent we get:

$$\begin{aligned} E(Y) &= \textstyle\sum_{e \in S} E(X_e) = \frac{|S|}{2^m} \\ VAR(Y) &= \textstyle\sum_{e \in S} VAR(X_e) = \frac{|S|}{2^m}(1 - \frac{1}{2^m}) = (1 - \frac{1}{2^m}) \cdot E(Y) \end{aligned}$$

We will now use the Chebychev inequality to prove:

$$\begin{aligned} Prob[|\{e \in S : h(e) = 0^m\}| \in (1 \pm \epsilon) \cdot \tfrac{|S|}{2^m}] &= Prob[Y \in (1 \pm \epsilon) \cdot E(Y)] \\ &= Prob[|Y - EY| \leq \epsilon \cdot E(Y)] \\ &\geq 1 - \tfrac{VAR(Y)}{(\epsilon \cdot E(Y))^2} = 1 - \tfrac{(1 - \frac{1}{2^m}) \cdot E(Y)}{\epsilon^2 (E(Y))^2} \\ &= 1 - \tfrac{(1 - \frac{1}{2^m}) 2^m}{\epsilon^2 \cdot |S|} \geq 1 - \epsilon \cdot (1 - \tfrac{1}{2^m}) > 1 - \epsilon \end{aligned}$$

■

Lecture 11

Interactive Proof Systems

Notes taken by Danny Harnik, Tzvika Hartman and Hillel Kugler

Summary: We introduce the notion of interactive proof systems and the complexity class IP, emphasizing the role of randomness and interaction in this model. The concept is demonstrated by giving an interactive proof system for the graph non-isomorphism language. We discuss the power of the class IP and prove that co$\mathcal{NP} \subseteq \mathcal{IP}$. We discuss issues regarding the number of rounds allowed in a proof system and introduce the class AM capturing languages recognized by Arthur-Merlin games.

11.1 Introduction

A proof is a way of convincing a party of a certain claim. When talking about proofs, we consider two parties: the *prover* and the *verifier*. Given an assertion, the prover's goal is to convince the verifier of it's validity, whereas the verifier's objective is to accept only a correct assertion. In mathematics, for instance, the prover provides a fixed sequence of claims and the verifier checks that they are truthful and that they imply the theorem. In real life, however, the notion of a proof has a much wider interpretation. A proof is a process rather than a fixed object, by which the validity of the assertion is established. For instance, a job interview is a process in which the candidate tries to convince the employer that she should hire him. In order to make the right decision, the employer carries out an interactive process. Unlike a fixed set of questions, in an interview the employer can adapt her questions according to the answers of the candidate, and therefore extract more information, and lead to a better decision. This example exhibits the power of a proof process rather than a fixed proof. In particular it shows the benefits of interaction between the parties.

In many contexts, finding a proof requires creativity and originality, and therefore attracts most of the attention. However, in our discussion of proof systems, we will focus on the task of the verifier – the verification process. Typically the verification procedure is considered to be relatively easy while finding the proof is considered a harder task. The asymmetry between the complexity of verification and finding proofs is captured by the complexity class NP.

We can view NP as a proof system, where the only restriction is on the complexity of the verification procedure (the verification procedure must take at most polynomial-time). For each language $L \in$NP there exists a polynomial-time recognizable relation R_L such that:

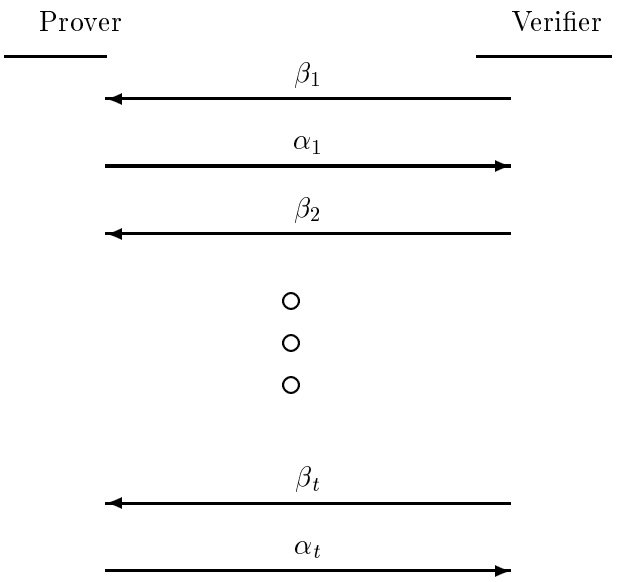
$$L = \{x : \exists y \text{ s.t. } (x, y) \in R_L\}$$

and $(x, y) \in R_L$ only if $|y| \leq poly(|x|)$. In a proof system for an NP language L, a proof for the claim "$x \in L$" consists of the prover sending a witness y, and the verifier checking in polynomial-time whether $(x, y) \in R_L$. Such a witness exists only if the claim is true, hence, only true assertions can be proved by this system. Note that there is no restriction on the time complexity of finding the proof (witness). A good proof system must have the following properties:

1. The verifier strategy is efficient (polynomial-time in the NP case).
2. Correctness requirements:
   - Completeness : For a true assertion, there is a convincing proof strategy (in the case of NP, if $x \in L$ then a witness y exists).
   - Soundness : For a false assertion, no convincing proof strategy exists (in the case of NP, if $x \notin L$ then no witness y exists).

In the following discussion we introduce the notion of *interactive proofs*. To do so, we generalize the requirements from a proof system, adding interaction and randomness.

Roughly speaking, an interactive proof is a sequence of questions and answers between the parties. The verifier asks the prover a question β_i and the prover answers with message α_i. At the end of the interaction, the verifier decides based the knowledge he acquired in the process whether the claim is true or false.



11.2 The Definition of IP

Following the above discussion we define

Definition 11.1 (interactive proof systems): *An* interactive proof system *for a language L is a two-party game between a verifier and a prover that interact on a common input in a way satisfying the following properties:*

1. *The verifier strategy is a probabilistic polynomial-time procedure* (where time is measured in terms of the length of the common input).

2. *Correctness requirements:*

   - Completeness *: There exists a prover strategy P, such that for every $x \in L$, when interacting on the common input x, the prover P convinces the verifier with probability at least $\frac{2}{3}$.*
   - Soundness *: For every $x \notin L$, when interacting on the common input x, any prover strategy P^* convinces the verifier with probability at most $\frac{1}{3}$.*

Note that the prover strategy is computationally unbounded.

Definition 11.2 (The IP Hierarchy): *The complexity class IP consists of all the languages having an interactive proof system.*

We call the number of messages exchanged during the protocol between the two parties, the number of rounds *in the system.*

For every integer function $r(\cdot)$, the complexity class IP($r(\cdot)$) consists of all the languages that have an interactive proof system in which, on common input x, at most $r(|x|)$ rounds are used.

For a set of integer functions R, we denote

$$\mathcal{IP}(R) = \bigcup_{r \in R} \mathcal{IP}(r(\cdot))$$

11.2.1 Comments

- Clearly, $\mathcal{NP} \subseteq \mathcal{IP}$ (actually, $\mathcal{NP} \subseteq \mathcal{IP}(1)$).

  Also, $\mathcal{BPP} = \mathcal{IP}(0)$.

- The number of rounds in IP cannot be more than a polynomial in the length of the common input, since the verifier strategy must run in polynomial-time. Therefore, if we denote by `poly` the set of all integer polynomial functions, then $\mathcal{IP} = \mathcal{IP}(\texttt{poly})$.

- The requirement for completeness, can be modified to require perfect completeness (acceptance probability 1). In other words, if $x \in L$, the prover can always convince the verifier. These two definitions are equivalent. Unlike this, if we require perfect soundness, interactive proof systems collapse to NP-proof systems. These results will be shown in Section 11.5.

- Much like in the definition of the complexity class BPP, the probabilities $\frac{2}{3}$ and $\frac{1}{3}$ in the completeness and soundness requirements can be replaced with probabilities as extreme as $1 - 2^{-p(\cdot)}$ and $2^{-p(\cdot)}$, for any polynomial $p(\cdot)$. In other words the following claim holds:

  Claim 11.2.1 *Any language that has an interactive proof system, has one that achieves error probability of at most $2^{-p(\cdot)}$ for any polynomial $p(\cdot)$.*

  Proof: We repeat the proof system sequentially for k times, and take a majority vote. Denote by z the number of accepting votes. If the assertion holds, then z is the sum of k independent Bernoulli trials with probability of success at least $\frac{2}{3}$. An error in the new protocol happens if $z < \frac{1}{2}k$.

Using Chernoff's Bound :

$$\Pr[z < (1-\delta)E(z)] < e^{-\frac{\delta^2 E(z)}{2}}$$

We choose $k = O(p(\cdot))$ and $\delta = \frac{1}{4}$ and note that $E(z) = \frac{2}{3}k$ (so that $\frac{3}{4} \cdot \frac{2}{3} = \frac{1}{2}$) to get:

$$\Pr\left[z < \frac{1}{2} \cdot k\right] < 2^{-p(\cdot)}$$

The same argument holds for the soundness error (as due to the sequential nature of the interaction we can assert that in each of the k iterations, for any history of prior interactions, the success probability of any cheating strategy is bounded by 1/3). ■

The proof above uses sequential repetition of the protocol to amplify the probabilities. This suffices for showing that the class IP is invariant under the various definitions discussed. However, this method increases the number of rounds used in the proof system. In order to show the invariance of the class IP$(r(\cdot))$, an analysis of the parallel repetition version should be given. (Such an argument is given in Appendix C.1 of [3].)

- Introducing both interaction and randomness in the IP class is essential.
  - By adding interaction only, the interactive proof systems collapse to NP-proof systems. Given an interactive proof system for a prover and a deterministic verifier, we construct an NP- proof system. The prover can predict the verifier's part of the interaction and send the full transcript as an NP witness. The verifier checks that the witness is a valid and accepting transcript of the original proof system. An alternative argument uses the fact that interactive proof systems with perfect soundness are equivalent to NP-proof systems (and the fact that a deterministic verifier necessarily yields perfect soundness).
  - By adding randomness only, we get a proof system in which the prover sends a witness and the verifier can run a BPP algorithm for checking its validity. We obtain a class IP(1) (also denoted MA) which seems to be a randomized (and perhaps stronger) version of NP.

11.2.2 Example – Graph Non-Isomorphism (GNI)

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are called *isomorphic* (denoted $G_1 \cong G_2$) if there exists a 1-1 and onto mapping $\pi : V_1 \to V_2$ such that $(u, v) \in E_1 \Leftrightarrow (\pi(u), \pi(v)) \in E_2$. The mapping π, if existing, is called an *isomorphism* between the graphs. If no such mapping exists then the graphs are *non-isomorphic* (denoted $G_1 \ncong G_2$).

GNI is the language containing all pairs of non-isomorphic graphs. Formally :

$$GNI = \{(G_1, G_2) : G_1 \ncong G_2\}$$

An interactive proof system for GNI:

- G_1 and G_2 are given as input to the verifier and the prover. Assume without loss of generality that $V_1 = V_2 = \{1, 2, ..., n\}$
- The verifier chooses $i \in_R \{1, 2\}$ and $\pi \in_R S_n$ (S_n is the group of all permutations on $\{1, 2, ..., n\}$).

He applies the mapping π on the graph G_i to obtain a graph H

$$H = (\{1, 2, ..., n\}, E_H) \text{ where } E_H = \{(\pi(u), \pi(v)) : (u, v) \in E_i\}$$

and sends the graph H to the prover.

- The prover sends $j \in \{1, 2\}$ to the verifier.
- The verifier accepts iff $j = i$.

Motivation : 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 and therefore, the best choice the prover could make is a random one. This fact enables the verifier to distinguish between the two cases. Formally:

Claim 11.2.2 *The above protocol is an interactive proof system for GNI.*

Comment: We show that the above protocol is an interactive proof system with soundness probability at most $\frac{1}{2}$ rather than $\frac{1}{3}$ as in the formal definition. However, this is equivalent by an amplification argument (see Claim 11.2.1).

Proof: We have to show that the above system satisfies the two properties in the definition of interactive proof systems:

- The verifier's strategy can be easily implemented in probabilistic polynomial time. (The prover's complexity is unbounded and indeed, he has to check isomorphism between two graphs, a problem not known to be solved in probabilistic polynomial time.)
- – Completeness : In case $G_1 \ncong G_2$, every graph can be isomorphic to at most one of G_1 or G_2 (otherwise, the existence of a graph isomorphic to both G_1 and G_2 implies $G_1 \cong G_2$). It follows that the prover can always send the correct j (i.e. a j such that $j = i$), since $H \cong G_i$ and $H \ncong G_{3-i}$.
  – Soundness : In case $G_1 \cong G_2$ we show that the prover convinces the verifier with probability at most $\frac{1}{2}$ (the probability ranges over all the possible coin tosses of the verifier, i.e. the choice of i and π). Denote by H the graph sent by the verifier. $G_1 \cong G_2$ implies that H is isomorphic to both G_1 and G_2. For $k = 1,2$ let

  $$S_{G_k} = \{\sigma \in S_n \mid \sigma G_k = H\}$$

  This means that when choosing $i = k$, the verifier can obtain H only by choosing $\pi \in S_{G_k}$.
  Assume $\tau \in S_n$ is an isomorphism between G_2 and G_1, i.e. $G_1 = \tau G_2$. For every $\sigma \in S_{G_1}$ it follows that $\sigma\tau \in S_{G_2}$ (because $\sigma\tau G_2 = \sigma G_1 = H$). Therefore, τ is a 1-1 mapping from S_{G_1} to S_{G_2} (since S_n is a group). Similarly, τ^{-1} is a 1-1 mapping from S_{G_2} to S_{G_1}. Combining the two arguments we get that $|S_{G_1}| = |S_{G_2}|$. Therefore, given that H was sent, the probability that the verifier chose $i = 1$ is equal to the probability of the choice $i = 2$. It follows that for every decision the prover makes he has success probability $\frac{1}{2}$ and therefore, his total probability of success is $\frac{1}{2}$.

■

The above interactive proof system is implemented with only 2 rounds. Therefore,

Corollary 11.3 *GNI $\in$ IP(2).*

11.3 The Power of IP

We have already seen that NP$\subseteq$ IP. The above example suggests that the power of IP is even greater. Since GNI is not known to be in NP we conjecture that NP$\subset$ IP (strict inclusion). Furthermore, the class of languages having interactive proof systems is shown to be equivalent to the powerful complexity class PSPACE. Formally,

Theorem 11.4 $\mathcal{IP} = \mathcal{PSPACE}$.

We will only give a partial proof of the theorem. We'll only show that co$\mathcal{NP} \subseteq \mathcal{IP} \subseteq \mathcal{PSPACE}$.

11.3.1 IP is contained in PSPACE

We start by proving the less interesting direction of the theorem (i.e., IP $\subseteq$ PSPACE). This is proven by showing that (for every fixed verifier), an optimal prover strategy exists and can be implemented in polynomial-space.

The Optimal Prover: Given a fixed verifier strategy, there exists an optimal prover strategy; that is, for every common input x, the optimal strategy has the highest possible probability of convincing the verifier. Note that an optimal prover strategy is well-defined, as for every input x and fixed prover strategy, the probability that the prescribed verifier accepts is well-defined (and the number of prover's strategies for input x is finite). A more explicit way of arguing the existence of an optimal prover strategy yields an algorithm for computing it. We first observe that given the verifier strategy and the verifier's coin tosses, we can simulate the whole interaction and it's outcome for any prover strategy. Now, the optimal prover strategy may enumerate all possible outcomes of the verifier's coin tosses, and count how many times each strategy succeeds. The optimal strategy for each input, is one that yields the highest number of successes. Furthermore, this can be done in polynomial-space:

Claim 11.3.1 *The optimal prover strategy can be computed in polynomial-space.*

Proof: We assume without loss of generality that the verifier tosses all his coins before the interaction begins. We also assume that the verifier plays first. Let β_i be the i^{th} message sent by the verifier and α_i be the i^{th} message sent by the prover. Let r be the outcome of all the verifier's coin tosses. Let $R_{\beta_1,\alpha_1,...,\alpha_{i-1},\beta_i}$ be the set of all r's (outcome of coin tosses) that are consistent with the interaction $\beta_1, \alpha_1, ..., \alpha_{i-1}, \beta_i$.

Let $F(\beta_1, \alpha_1, ..., \alpha_{i-1}, \beta_i)$ be the probability that an interaction (between the optimal prover and the fixed verifier) beginning with $\beta_1, \alpha_1, ..., \alpha_{i-1}, \beta_i$ will result in acceptance. The probability is taken uniformly over the verifier's relevant coin tosses (only r such that $r \in R_{\beta_1,\alpha_1,...,\alpha_{i-1},\beta_i}$).

Suppose an interaction between the two parties consists of $\beta_1, \alpha_1, ..., \alpha_{i-1}, \beta_i$ and it is now the prover's turn to play. Using the function F, the prover can find the optimal move. We show that a polynomial-space prover can recursively compute $F(\beta_1, \alpha_1, ..., \alpha_{i-1}, \beta_i)$. Furthermore, in the process, the prover finds an α_i that yields this probability and hence, an α_i that is an optimal move for the prover.

The best choice for α_i is one that gives the highest expected value of $F(\beta_1, \alpha_1, ..., \alpha_i, \beta_{i+1})$ over all of the possiblities of verifier's next message (β_{i+1}). Formally :

$$(1) \qquad F(\beta_1, \alpha_1, ..., \alpha_{i-1}, \beta_i) = \max_{\alpha_i} E_{\beta_{i+1}}[F(\beta_1, \alpha_1, ..., \alpha_i, \beta_{i+1})]$$

Let $V(r, \alpha_1, ..., \alpha_i)$ be the message β_{i+1} that the verifier sends after tossing coins r and receiving messages $\alpha_1, ..., \alpha_i$ from the prover.

The probability for each possible message β_{i+1} to be sent by after $\beta_1, \alpha_1, ..., \alpha_i$ is the portion of possible coins $r \in R_{\beta_1,\alpha_1,...,\alpha_{i-1},\beta_i}$ that yield the message β_{i+1} (i.e. $\beta_{i+1} = V(r, \alpha_1, ..., \alpha_i)$). This yields the following equation for the expected probability :

$$(2) \qquad E_{\beta_{i+1}}[F(\beta_1, \alpha_1, ..., \alpha_i, \beta_{i+1})] = \frac{1}{|R_{\beta_1,\alpha_1,...,\beta_i}|} \sum_{r \in R_{\beta_1,\alpha_1,...,\beta_i}} F(\beta_1, \alpha_1, ..., \alpha_i, V(r, \alpha_1, ..., \alpha_i))$$

Combining (1) and (2) we get the recursion formula

$$F(\beta_1, \alpha_1, ..., \alpha_{i-1}, \beta_i) = \max_{\alpha_i} \frac{1}{|R_{\beta_1,\alpha_1,...,\beta_i}|} \sum_{r \in R_{\beta_1,\alpha_1,...,\beta_i}} F(\beta_1, \alpha_1, ..., \alpha_i, V(r, \alpha_1, ..., \alpha_i))$$

We now show how to compute the function F in polynomial-space:
For each potential α_i, we enumerate all possible values of r. For each r, all of the following can be done in polynomial-space:

- Checking if $r \in R_{\beta_1,\alpha_1,...,\beta_i}$ by simulating the verifier in the first i interactions (when given r the verifier strategy is polynomial).
- Calculating $\beta_{i+1} = V(r, \alpha_1, ..., \alpha_i)$ again by simulating the verifier.
- Recursively computing $F(\beta_1, \alpha_1, ..., \alpha_i, \beta_{i+1})$.

  In order for the recursion to be polynomial-space computable, we need to show that the recursion stops after polynomially many stages, and that the last stage can be computed in polynomial-space. The recursion stops when reaching a full transcript of the proof system. In such a case the prover can enumerate r and find the probability of acceptance among all consistent r by simulating the verifier. Clearly, this can be done in polynomial-space. Also the depth of the recursion must be at most polynomial, which is obviously the case here, since it is bounded by the number of rounds.

Using polynomial-size counters, we can sum the probabilities for all consistent r, and find the expected probability for each α_i. By repeating this for all possible α_i we can find one that maximizes the expectation. Altogether, the prover's optimal strategy can be calculated in polynomial-space.

Note: All the probabilities are taken over the verifier's coin tosses (no more than a polynomial number of coins). This enables us to use polynomial-size memory for calculating all probabilities with exact resolution (by representing them as rational numbers – storing the numerator and denominator separately). ■

Corollary 11.5 *IP* $\subseteq$ *PSPACE*

Proof: If $L \in$ IP then there exists an interactive proof system for L and hence there exists a polynomial-space optimal prover strategy. Given input x and the verifier's coin tosses, we can simulate (in polynomial-space) the interaction between the optimal prover and the verifier and determine this interaction's outcome. We enumerate over all the possible verifier's coin tosses and accept only if more than $\frac{2}{3}$ of the outcomes are accepting. Clearly, we accept if and only if $x \in L$ and this can be implemented in polynomial-space. ■

11.3.2 coNP is contained in IP

As mentioned above, we will not prove that $\mathcal{PSPACE} \subseteq \mathcal{IP}$. Instead, we prove a weaker theorem (i.e., co$\mathcal{NP} \subseteq \mathcal{IP}$), which by itself is already very interesting. The proof of the weaker theorem presents all but one ingrediant of the proof $\mathcal{PSPACE} \subseteq \mathcal{IP}$ (and the missing ingrediant is less interesting).

Theorem 11.6 co$\mathcal{NP} \subseteq \mathcal{IP}$

Proof: We prove the theorem by presenting an interactive proof system for the coNP-complete problem $\overline{3SAT}$ (the same method can work for the problem $\overline{SAT}$ as well). $\overline{3SAT}$ is the set of non-satisfiable $3CNF$ formulae: Given a $3CNF$ formula ϕ, it is in the set if no truth assignment to it's variables satisfies the formula.

The proof uses an arithmetic generalization of the boolean problem, which allows us to apply algebraic methods in the proof system.

The Arithmetization of a Boolean CNF formula: Given the formula ϕ with variables $x_1, ..., x_n$ we perform the following replacements:

Boolean		Arithmetic
T	$\longrightarrow$	positive integers
F	$\longrightarrow$	0
x_i	$\longrightarrow$	x_i
$\overline{x_i}$	$\longrightarrow$	$(1 - x_i)$
$\vee$	$\longrightarrow$	$+$
$\wedge$	$\longrightarrow$	$\cdot$
$\phi(x_1, ..., x_n)$	$\longrightarrow$	$\Phi(x_1, ..., x_n)$

Every boolean $3CNF$ formula ϕ is transformed into a multi-variable polynomial Φ. It is easy to see that for every assignment $x_1, ..., x_n$, we have

$$\phi(x_1, ..., x_n) = F \iff \Phi(x_1, ..., x_n) = 0$$

Summing over all possible assignments, we obtain an equation for the non-satisfiability of ϕ:

$$\phi \text{ is unsatisfiable } \iff \sum_{x_1=0,1} ... \sum_{x_n=0,1} \Phi(x_1, ..., x_n) = 0$$

Suppose ϕ has m clauses of length three each, thus any 0-1 assignment to $x_1, ..., x_n$ gives $\Phi(x_1, ..., x_n) \leq 3^m$. Since there are 2^n different assignments, the sum above is bounded by $2^n \cdot 3^m$. This fact allows us to move our calculations to a finite field, by choosing a prime q such that $q > 2^n \cdot 3^m$, and working modulo this prime. Thus proving that ϕ is unsatisfiable reduces to proving that

$$\sum_{x_1=0,1} ... \sum_{x_n=0,1} \Phi(x_1, ..., x_n) \equiv 0 \pmod{q}$$

We choose q to be not much larger than $2^n \cdot 3^m$ (this is possible due to the density of the prime numbers). Thus, we obtain that all calculations over the field $GF(q)$ can be done in polynomial-time (in the input length). Working over a finite field will later help us in the task of uniformly selecting an element in the field.

The interactive proof system for $\overline{3SAT}$:

- Both sides receive the common boolean formula ϕ. They perform the arithmetization procedure and obtain Φ.
- The prover picks a prime q such that $q > 2^n \cdot 3^m$, and sends q to the verifier. The verifier checks that q is indeed a prime. If not he rejects.
- The verifier initializes $v_0 = 0$.
- The following is performed n times (i runs from 1 to n):
  - The prover sends a polynomial $P_i(\cdot)$ of degree at most m to the verifier.
  - The verifier checks whether $P_i(0) + P_i(1) \equiv v_{i-1} \pmod{q}$ and that the polynomial's degree is at most m.
    If not, the verifier rejects.
    Otherwise, he uniformly picks $r_i \in_R GF(q)$, calculates $v_i = P_i(r_i)$ and sends r_i to the prover.
- The verifier accepts if $\Phi(r_1, ...r_n) \equiv v_n \pmod{q}$ and rejects otherwise.

Motivation: The prover has to find a sequence of polynomials that satisfies a number of restrictions. The restrictions are imposed by the verifier in the following interactive manner: after receiving a polynomial from the prover, the verifier sets a new restriction for the next polynomial in the sequence. These restrictions guarantee that if the claim holds (ϕ is unsatisfiable), such a sequence can always be found (we call it the "Honest prover strategy"). However, if the claim is false, any prover strategy has only a small probability of finding such a sequence (the probability is taken over the verifier's coin tosses). This yields the completeness and soundness of the suggested proof system. The nature of these restrictions is fully clarified in the proof of soundness, but we will first show that the verifier strategy is efficient.

The verifier strategy is efficient: Most steps in the protocol are calculations of polynomials of degree m in n variables, these are easily calculated in polynomial-time. The transformation to an arithmetic field is linear in the formula's length.

Checking primality is known to be in BPP and therefore can be done by the verifier. Furthermore, it can be shown that primality testing is in NP, so the prover can send the verifier an NP-witness to the fact that q is a prime, and the verifier checks this witness in polynomial-time.

Finally, picking an element $r \in_R GF(q)$ can be done in $O(\log q)$ coin tosses, that is polynomial in the formula's length.

The honest prover strategy: For every i define the polynomial:

$$P_i^*(z) = \sum_{x_{i+1}=0,1} ... \sum_{x_n=0,1} \Phi(r_1, ..., r_{i-1}, z, x_{i+1}, ..., x_n)$$

Note that $r_1, ..., r_{i-1}$ are constants set by the verifier in the previous stages and known to the prover at the i^{th} stage, and z is the polynomial's variable.
The following facts are evident about P_i^*:

- Calculating P_i^* may take exponential-time, but this is no obstacle for a computationally unbounded prover.
- The degree of P_i^* is at most m. Since there are at most m clauses in ϕ, the highest degree of any one variable is m (if it appears in all clauses).

Completeness of the proof system: When the claim holds, the honest prover always succeeds in convincing the verifier. For $i > 1$:

$$(3.1) \quad P_i^*(0) + P_i^*(1) = \sum_{x_i=0,1} P_i^*(x_i) =_{(1)} \sum_{x_i=0,1} ... \sum_{x_n=0,1} \Phi(r_1, ..., r_{i-1}, x_i, ..., x_n)$$

$$=_{(2)} P_{i-1}^*(r_{i-1}) \equiv_{(3)} v_{i-1} \pmod{q}$$

Equality (1) is due to the definition of P_i^*. Equality (2) is due to the definition of P_{i-1}^*. Equality (3) is the definition of v_{i-1}.
Also for $i = 1$, since the claim holds we have:

$$P_1^*(0) + P_1^*(1) = \sum_{x_1=0,1} P_1^*(x_1) = \sum_{x_1=0,1} ... \sum_{x_n=0,1} \Phi(x_1, ..., x_n) \equiv v_0 \pmod{q}$$

And finally: $v_n = P_n^*(r_n) = \Phi(r_1, ..., r_n)$.

We showed that the polynomials of the honest prover pass all of the verifier's tests, obtaining perfect completeness of the proof system.

Soundness of the proof system: If the claim is false, an honest prover will definitely fail after sending P_1^*, thus a prover must be dishonest.

Roughly speaking, we will show that if a prover is dishonest in one round, then with high probability he must be dishonest in the next round as well. In the last round, his dishonesty is revealed. Formally:

Lemma 11.3.2 *If $P_i^*(0) + P_i^*(1) \not\equiv v_{i-1} \pmod{q}$ then either the verifier rejects in the i^{th} round, or $P_i^*(r_i) \not\equiv v_i \pmod{q}$ with probability at least $1 - \frac{m}{q}$, where the probability is taken over the verifier's choices of r_i.*

We stress that P_i^* is the polynomial of the honest prover strategy (as defined above), while P_i is the polynomial actually sent by the prover (v_i is set using P_i).
Proof: (of lemma) If the prover sends $P_i = P_i^*$, we get:

$$P_i(0) + P_i(1) \equiv P_i^*(0) + P_i^*(1) \not\equiv v_{i-1} \pmod{q}$$

and the verifier rejects immeadiately.

Otherwise the prover sends $P_i \neq P_i^*$. We assume P_i passed the verifier's test (if not the verifier rejects and we are done). Since P_i and P_i^* are of degree at most m, there are at most m choices of $r_i \in GF(q)$ such that

$$P_i^*(r_i) \equiv P_i(r_i) \pmod{q}$$

For all other choices:

$$P_i^*(r_i) \not\equiv P_i(r_i) \equiv v_i \pmod{q}$$

Since the verifier picks $r_i \in_R GF(q)$, we get $P_i^*(r_i) \equiv v_i \pmod{q}$ with probability at most $\frac{m}{q}$, ■

Suppose the verifier does not reject in any of the n rounds. Since the claim is false (ϕ is satisfiable), we have $P_1^*(0) + P_1^*(1) \not\equiv v_0 \pmod{q}$. Applying alternately the lemma and the following equality: for every $i \geq 2$ $P_{i-1}^*(r_{i-1}) = P_i^*(0) + P_i^*(1)$ (due to equation 3.1) , we get that $P_n^*(r_n) \not\equiv v_n \pmod{q}$ with probability at least $(1 - \frac{m}{q})^n$. But $P_n^*(r_n) = \Phi(r_1, ..., r_n)$ so the verifier's last test fails and he rejects. Altogether the verifier fails with probability $(1 - \frac{m}{q})^n > 1 - \frac{nm}{q} > \frac{2}{3}$ (by the choice of q). ■

11.4 Public-Coin Systems and the Number of Rounds

An interesting question is how the power of interactive proof systems is affected by the number of rounds allowed. We have already seen that GNI can be proved by an interactive proof with 2 rounds. Despite this example of a coNP language, we conjecture that coNP $\not\subseteq$ IP($O(1)$). Together with the previous theorem we get:

Conjecture 11.7

$$\mathcal{IP}(O(1)) \subset \mathcal{IP}(poly) \quad \textit{(strict containment)}$$

A useful tool in the study of interactive proofs, is the public coin variant, in which the verifier can only ask random questions.

Definition 11.8 (public-coin interactive proofs – $\mathcal{AM}$): *Public coin proof systems (known also as Arthur-Merlin games) are a special case of interactive proof systems, in which, at each round, the verifier can only toss coins, and send their outcome to the prover. In the last round, the verifier decides whether to accept or reject.*

For every integer function $r(\cdot)$, the complexity class $\mathcal{AM}(r(\cdot))$ consists of all the languages that have an Arthur-Merlin proof system in which, on common input x, at most $r(|x|)$ rounds are used.

Denote $\mathcal{AM} = \mathcal{AM}(2)$.

We note that the definition of $\mathcal{AM}$ as Arthur-Merlin games with two rounds is inconsistent with the notation IP= IP($poly$)). (Unfortunately, that's what is found in the literature.)

The difference between the Arthur-Merlin games and the general interactive proof systems can be viewed as the difference between asking tricky questions, versus asking random questions. Surprisingly it was shown that these two versions are essentially equivalent:

Theorem 11.9 (Relating $\mathcal{IP}(\cdot)$ to $\mathcal{AM}(\cdot)$):

$$\forall r(\cdot) \quad \mathcal{IP}(r(\cdot)) \subseteq \mathcal{AM}(r(\cdot)+2)$$

The following theorem shows that power of $\mathcal{AM}(r(\cdot))$ is invariant under a linear change in the number of rounds:

Theorem 11.10 (Linear Speed-up Theorem):

$$\forall r(\cdot) \geq 2 \quad \mathcal{AM}(2r(\cdot)) = \mathcal{AM}(r(\cdot))$$

The above two theorems are quoted without proof. Combining them we get:

Corollary 11.11 $\forall r(\cdot) \geq 2 \quad \mathcal{IP}(2r(\cdot)) = \mathcal{IP}(r(\cdot))$.

Corollary 11.12 (Collapse of constant-round IP to two-round AM):

$$\mathcal{IP}(O(1)) = \mathcal{AM}(2)$$

11.5 Perfect Completeness and Soundness

In the definition of interactive proof systems we require the existence of a prover strategy that for $x \in L$ convinces the verifier with probability at least $\frac{2}{3}$ (analogous to the definition of the complexity class BPP). One can consider a definition requiring *perfect completeness*; i.e., convincing the verifier with probability 1 (analogous to coRP). We will now show that the definitions are equivalent.

Theorem 11.13 *If a language L has an interactive proof system then it has one with perfect completeness.*

We will show that given a public coin proof system we can construct a perfect completeness public coin proof system.

We use the fact that public coin proof systems and interactive proof systems are equivalent (see Theorem 11.9), so if L has an interactive proof system it also has a public coin proof system. We define:

$$AM^0(r(\cdot)) \quad = \quad \{L \mid \text{ L has perfect completeness } r(\cdot) \text{ round public coin proof system}\}$$

So given an interactive proof system we create a public coin proof system and using the following lemma convert it to one with perfect completeness. Thus, the above theorem which refers to arbitrary interactive proofs follows from the following lemma which refers only to public-coin interactive proofs.

Lemma 11.5.1 *If L has a public coin proof system then it has one with perfect completeness*

$$AM(r(\cdot)) \subseteq AM^0(r(\cdot)+1)$$

Proof: Given an Arthur-Merlin proof system, we construct an Arthur-Merlin proof system with perfect completeness and one more round. We use the same idea as in the proof of BPP $\subseteq$ PH.

Assume, without loss of generality, that the Arthur-Merlin proof system consists of $2t$ rounds, and that Arthur sends the same number of coins m in each round (otherwise, ignore the redundant coins). Also assume that the completeness and soundness error probabilities of the proof system are at most $\frac{1}{3tm}$. This is obtained using amplification (see Section 11.2.1).

We denote the messages sent by Arthur (the verifier) $r_1, ..., r_t$ and the messages sent by Merlin (the prover) $\alpha_1, ..., \alpha_t$. Denote by $\langle P, V\rangle_x(r_1, ..., r_t)$ the outcome of the game on common input x between the optimal prover P and the verifier V in which the verifier uses coins $r_1, ..., r_t$: $\langle P, V\rangle_x(r_1, ..., r_t) = 0$ if the verifier rejects and $\langle P, V\rangle_x(r_1, ..., r_t) = 1$ otherwise.

We construct a new proof system with perfect completeness, in which Arthur and Merlin play tm games simultaneously. Each game is like the original game except that the random coins are shifted by a fixed amount. The tm shifts (one for each game) are sent by Merlin in an additional round at the beginning. Arthur accepts if at least one of the games is accepting. Formally, we add an additional round at the beginning in which Merlin sends the shifts $S^1, ..., S^{tm}$ where $S^i = (S^i_1, ..., S^i_t), S^i_j \in \{0,1\}^m$ for every i between 1 and tm. Like in the original proof system Arthur sends messages $r_1, ..., r_t$, where $r_i \in_R \{0,1\}^m$. For game i and round j, Merlin considers the random coins to be $r_j \oplus S^i_j$ and sends as a message α^i_j where α^i_j is computed according to $(r_1 \oplus S^i_1, ..., r_j \oplus S^i_j)$. The entire message in round j is $\alpha^1_j, ..., \alpha^{tm}_j$. At the end of the protocol Arthur accepts if at least one out of the tm games is accepting.

In order to show perfect completeness we will show that for every $x \in L$ there exist $S^1, ..., S^{tm}$ such that for all $r_1, ...r_t$ at least one of the games is accepting. We use a probabilistic argument to show that the complementary event occurs with probability strictly smaller than 1.

$$\begin{aligned}
&\Pr_{S^1,...S^{tm}}\left[\exists r_1, ...r_t \bigwedge_{i=1}^{tm}(\langle P,V\rangle_x(r_1 \oplus S_1^i, ..., r_t \oplus S_t^i) = 0)\right] \\
&\quad \leq_{(1)} \sum_{r_1,...r_t \in \{0,1\}^m} \Pr_{S^1,...S^{tm}}\left[\bigwedge_{i=1}^{tm}(\langle P,V\rangle_x(r_1 \oplus S_1^i, ..., r_t \oplus S_t^i) = 0)\right] \\
&\quad \leq_{(2)} 2^{tm} \cdot \left(\frac{1}{3tm}\right)^{tm} \quad < \quad 1
\end{aligned}$$

Inequality (1) is obtained using the union bound. Inequality (2) is due to the fact that the $r_j \oplus S_j^i$ are independent random variables so the results of the games are independent, and that the optimal prover fails to convince the verifier on a true assertion with probability at most $\frac{1}{3tm}$.

We still have to show that the proof system suggested satisfies the soundness requirement. We show that for every $x \notin L$ and for any prover strategy P^* and choices of shifts $S^1, ..., S^{tm}$ the probability that one or more of the tm games is accepting is at most $\frac{1}{3}$.

$$\begin{aligned}
&\Pr_{r_1,...r_t}\left[\bigvee_{i=1}^{tm}(\langle P^*,V\rangle_x(r_1 \oplus S_1^i, ..., r_t \oplus S_t^i) = 1)\right] \\
&\quad \leq_{(1)} \sum_{i=1}^{tm} \Pr_{r_1,...,r_t}\left[\langle P^*,V\rangle_x(r_1 \oplus S_1^i, ..., r_t \oplus S_t^i) = 1\right] \\
&\quad \leq_{(2)} \sum_{i=1}^{tm} \frac{1}{3tm} \quad = \quad \frac{1}{3}
\end{aligned}$$

Inequality (1) is obtained using the union bound. Inequality (2) is due to the fact that any prover has probability of at most $\frac{1}{3tm}$ of success for a single game (because any strategy that the prover can play in a copy of the parallel game can be played in a single game as well). ■

Unlike the last theorem, requiring *perfect soundness* (i.e. for every $x \notin L$ and every prover strategy P^*, the verifier always rejects after interacting with P^* on common input x) reduces the model to an NP-proof system , as seen in the following proposition:

Proposition 11.5.2 *If a language L has an interactive proof system with perfect soundness then $L \in \mathcal{NP}$.*

Proof: Given an interactive proof system with perfect soundness we construct an NP proof system. In case $x \in L$, by the completeness requirement, there exists an accepting transcript. The prover finds an outcome of the verifier's coin tosses that gives such a transcript and sends the full transcript along with the coin tosses. The verifier checks in polynomial time that the transcript is valid and accepting and if so - accepts. This serves as an NP-witness to the fact that $x \in L$. If $x \notin L$ then due to the perfect soundness requirement, no outcome of verifier's coin tosses yields an accepting transcript and therefore there are no NP-witnesses. ■

Bibliographic Notes

Interactive proof systems were introduced by Goldwasser, Micali and Rackoff [5], with the explicit objective of capturing the most general notion of efficiently verifiable proof systems. The original motivation was the introduction of zero-knowledge proof systems, which in turn were supposed to provide (and indeed do provide) a powerful tool for the design of complex cryptographic schemes.

First evidence that interactive proofs may be more powerful than NP-proofs was given by Goldreich, Micali and Wigderson [4], in the form of the interactive proof for Graph Non-Isomorphism presented above. The full power of interactive proof systems was discoved by Lund, Fortnow, Karloff, Nisan, and Shamir (in [7] and [8]). The basic technique was presented in [7] (where it was shown that co$\mathcal{NP} \subseteq \mathcal{IP}$) and the final result ($\mathcal{PSPACE} = \mathcal{IP}$) in [8]. Our presentation follows [8]. For further discussion of credits, see [3].

Public-coin interactive proofs (also known as Arthur-Merlin proofs) were introduced by Babai [1]. The fact that these restricted interactive proofs are as powerful as general ones was proved by Goldwasser and Sipser [6]. The linear speed-up (in number of rounds) of public-coin interactive proofs was shown by Babai and Moran [2].

1. L. Babai. Trading Group Theory for Randomness. In *17th STOC*, pages 421–429, 1985.

2. L. Babai and S. Moran. Arthur-Merlin Games: A Randomized Proof System and a Hierarchy of Complexity Classes. *JCSS*, Vol. 36, pp. 254–276, 1988.

3. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. Algorithms and Combinatorics series (Vol. 17), *Springer*, 1998.

4. O. Goldreich, S. Micali and A. Wigderson. Proofs that Yield Nothing but their Validity or All Languages in NP Have Zero-Knowledge Proof Systems. *JACM*, Vol. 38, No. 1, pages 691–729, 1991. Preliminary version in *27th FOCS*, 1986.

5. S. Goldwasser, S. Micali and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SICOMP*, Vol. 18, pages 186–208, 1989. Preliminary version in *17th STOC*, 1985. Earlier versions date to 1982.

6. 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*, pages 59–68, 1986.

7. C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic Methods for Interactive Proof Systems. *JACM*, Vol. 39, No. 4, pages 859–868, 1992. Preliminary version in *31st FOCS*, 1990.

8. A. Shamir. IP = PSPACE. *JACM*, Vol. 39, No. 4, pages 869–877, 1992. Preliminary version in *31st FOCS*, 1990.

Lecture 12

Probabilistically Checkable Proof Systems

Notes taken by Alon Rosen and Vered Rosen

Summary: In this lecture we introduce the notion of Probabilistically Checkable Proof (PCP) systems. We discuss some complexity measures involved, and describe the class of languages captured by corresponding PCP systems. We then demonstrate the alternative view of $\mathcal{NP}$ emerging from the PCP theorem, and use it in order to prove two non-approximability results for the problems $max3SAT$ and $maxCLIQUE$.

12.1 Introduction

Loosely speaking, a probabilistically checkable proof system (PCP) for a language consists of a probabilistic polynomial-time verifier having direct access to individual bits of a binary string. This string (called oracle) represents a proof, and typically will be accessed only partially by the verifier. Queries to the oracle are positions on the bit string and will be determined by the verifier's input and coin tosses (potentially, they might be determined by answers to previous queries as well). The verifier is supposed to decide whether a given input belongs to the language.

If the input belongs to the language, the requirement is that the verifier will always accept (i.e. given access to an adequate oracle). On the other hand, if the input does not belong to the language then the verifier will reject with probability at least $\frac{1}{2}$, no matter which oracle is used.

One can view PCP systems in terms of interactive proof systems. That is, one can think of the oracle string as being the prover and of the queries as being the messages sent to him by the verifier. In the PCP setting however, the prover is considered to be memoryless and thus cannot adjust his answers based on previous queries posed to him.

A more appealing interpretation is to view PCP systems as a possible way of generalizing $\mathcal{NP}$. Instead of conducting a polynomial-time computation upon receiving the entire proof (as in the case of $\mathcal{NP}$), the verifier is allowed to toss coins and query the proof only at locations of his choice. This either allows him to inspect very long proofs (looking at no more than polynomially many locations), or alternatively, look at very few bits of a possible proof.

Most surprisingly, PCP systems have been used to fully characterize the languages in $\mathcal{NP}$. This characterization has been found to be useful in connecting the hardness involved in the approximation of some *NP-hard* problems with the $\mathcal{P} \neq \mathcal{NP}$ question. In other words, very strong

non-approximability results for various classical optimization problems have been established using PCP systems for $\mathcal{NP}$ languages.

12.2 The Definition

12.2.1 The basic model

In the definition of PCP systems we make use of the notion of a probabilistic oracle machine. In our setting, this will be a probabilistic Turing machine which, in addition to the usual features, will have direct access (counted as a single step) to individual bits of a binary string (the oracle). ¿From now on, we denote by $M^{\pi}(x)$ the output of machine M on input x, when given such oracle access to the binary string π.

Definition 12.1 (Probabilistically Checkable Proofs - PCP) *A* probabilistic checkable proof system for a language L *is a probabilistic polynomial-time oracle machine* (called verifier)*, denoted M, satisfying*

- *Completeness: For every $x \in L$ there exists an oracle π_x such that:*

$$\Pr[M^{\pi_x}(x) = 1] = 1$$

- *Soundness: For every $x \notin L$ and every oracle π:*

$$\Pr[M^{\pi}(x) = 1] \leq \frac{1}{2}$$

where the probability is taken over M's internal coin tosses.

12.2.2 Complexity Measures

When considering a randomized oracle machine, some complexity measures other than time may come into concern. A natural thing would be to count the number of queries made by the verifier. This number determines what is the portion of the proof being read by the verifier. Another concern would be to count the number of coins tossed by the randomized oracle machine. This in turn determines what is the total number of possible executions of the verifier (once an oracle is fixed).

It turns out that the class of languages captured by PCP systems varies greatly as the above mentioned resources of the verifier are changed. This motivates a quantitative refinement of the definition of PCP systems which captures the above discussed concept.

Definition 12.2 (Complexity Measures for PCP) *Let $r, q : N \to N$ be integer functions (in particular constant). The complexity class $\mathcal{PCP}(r(\cdot), q(\cdot))$ consists of languages having a probabilistic checkable proof system in which it holds that:*

- Randomness Complexity*: On input $x \in \{0,1\}^*$, the verifier makes at most $r(|x|)$ coin tosses.*
- Query Complexity*: On input $x \in \{0,1\}^*$, the verifier makes at most $q(|x|)$ queries.*

For sets of integer functions R and Q, we let

$$\mathcal{PCP}(R, Q) \stackrel{\text{def}}{=} \bigcup_{r \in R, q \in Q} \mathcal{PCP}(r(\cdot), q(\cdot))$$

In particular, we 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 (e.g. $f \in \log$ iff $f(n) = O(\log n)$). From now on, whenever referring to a PCP system, we will also specify its corresponding complexity measures.

12.2.3 Some Observations

- The definition of PCP involves binary queries to the oracle (which is itself a binary string). These queries specify locations on the string whose binary values are the answers to the corresponding queries. From now on, when given a query q to an oracle π the corresponding binary answer will be denoted π_q. Note that an oracle string can possibly be of exponential length (since one can specify an exponentially far location on the string using polynomially many bits).

- 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 answers to previously received oracle answers). From now on, whenever referring to a PCP verifier it will be assumed to be adaptive (unless otherwise specified).

- A possible motivation for the introduction of PCP systems would be to provide an alternative view of $\mathcal{NP}$, one that will rid us of the "rigidity" of the conventional view. In this regard randomness seems to be a most important ingredient, it provides us the possibility to be "imprecise" in the acceptance of false instances. This is best seen when taking the probability bound in the soundness condition to be zero. This will cause that no probability is involved in the definition and will make it collapse into $\mathcal{NP}$. To see this, notice that in the above case, the output of the verifier does not vary with the outcome of its coin tosses. This means that in order to determine the verifier's decision on some input, it suffices to examine only one of its possible executions (say, when using the all zero coin sequence). In such an execution only a polynomial portion of the PCP proof is being read by the verifier. It is easy to see, that in this case, the PCP and $\mathcal{NP}$ definitions coincide (just treat the relevant portion of the PCP proof as an $\mathcal{NP}$-*witness*).

  Note that in order to be consistent with the $\mathcal{NP}$ definition we require perfect completeness (i.e. a true instance is always accepted).

- The definition of PCP requires that for every x in L there exists a proof π_x for which it holds that $\Pr[M^{\pi_x}(x) = 1] = 1$. This means that π_x is potentially different for every x. However, we can assume w.l.o.g., that there exists a proof π which is common to all x's in L. This π will simply be the concatenation of all π_x's (according to some ordering of the x's in L). Since the verifier is polynomial we can assume that all π_x's are at most exponentially long (the verifier cannot access more than an exponentially long prefix of his proof). Therefore, the location of π_x within π will not be more than exponential in $|x|$ away, and so can be accessed in poly($|x|$) time.

- The oracle in a PCP system is viewed in a somewhat different manner than previously. We demonstrate this by comparing a PCP system to the mechanism of a *Cook-reduction*. Recall that a language L_1 is *Cook-reducible* to L_2 if there exists an oracle machine M such that for **all** $x \in \{0,1\}^*$ it holds that $M^{L_2}(x) = \chi_{L_1}(x)$. Note that the oracle in the *Cook-reduction* mechanism is the language L_2, and is supposed to exist for **all** $x \in \{0,1\}^*$ (regardless of the question whether x is in L or not). In contrast, in the case of PCP systems the oracle

π is supposed **not** to exist whenever x is not in L. That is, every oracle would cause the verifer to reject x with probability at least $\frac{1}{2}$. Therefore, in the PCP case (as opposed to the *Cook-reduction* case) there is a lack of "symmetry" between the positive instances of L and the negative ones.

12.3 The PCP characterization of NP

12.3.1 Importance of Complexity Parameters in PCP Systems

As was already mentioned in subsection 12.2.2, the class of languages captured by PCP systems varies greatly as the appropriate parameters $r(\cdot)$ and $q(\cdot)$ are modified. This fact is demonstrated by the following assertions:

- If $\mathcal{NP} \subseteq \mathcal{PCP}(o(\log), o(\log))$ then $\mathcal{NP} = \mathcal{P}$
- $\mathcal{PCP}(\text{poly}, \text{poly}) = \mathcal{NEXP}$ $(= \mathcal{NTIME}(2^{\text{poly}}))$

By taking either one of the complexity measures to zero the definition of PCP collapses into one of the following degenerate cases:

- $\mathcal{PCP}(\text{poly}, 0) = \text{co}\mathcal{RP}$
- $\mathcal{PCP}(0, \text{poly}) = \mathcal{NP}$

When looking at the above degenerate cases of the PCP definition we do not really gain any novel view on the complexity classes involved (in this case, co$\mathcal{RP}$ and $\mathcal{NP}$). Thus, the whole point of introducing the PCP definition may be missed. What we would like to see are more delicate assertions involving both non-zero randomness and query complexity. In the following subsection we demonstrate how PCP systems can be used in order to characterize the complexity class $\mathcal{NP}$ in such a non-degenerate way. This characterization will lead to a new perspective on $\mathcal{NP}$ and enable us to further investigate the languages in it.

12.3.2 The PCP Theorem

As already stated, the languages in the complexity class $\mathcal{NP}$ are trivially captured by PCP systems using zero randomness and a polynomial number of queries. A natural question arises: can the two complexity measures be traded off, in a way that still captures the class $\mathcal{NP}$? Most surprisingly, not only the answer to the above question is positive, but also a most powerful result emerges. The number of queries made by the verifier can be brought down to a constant while using only a logarithmic number of coin tosses. This result is known as the PCP theorem (it will be cited here without a proof).

Our goal is to characterize $\mathcal{NP}$ in terms of PCP systems. We start by demonstrating how $\mathcal{NP}$ upper bounds a fairly large class in the PCP hierarchy. This is the class of languages having a PCP system whose verifier makes a polynomial number of queries while using a logarithmic number of coin tosses.

Proposition 12.3.1 $\mathcal{PCP}(\log, \text{poly}) \subseteq \mathcal{NP}$

Proof: Let L be a language in $PCP(\log, \text{poly})$. We will show how to use its PCP system in order to construct a non-deterministic machine M which decides L in polynomial-time. This will imply that L is in $\mathcal{NP}$.

Let M' be the probabilistic-polynomial time oracle machine in the above $\mathcal{PCP}(\log, \text{poly})$ system for L. We are guaranteed that on input $x \in \{0,1\}^*$, M' makes $\text{poly}(|x|)$ queries using $O(\log(|x|))$ coin tosses. For the sake of simplicity, we prove the claim for a non-adaptive M' (in order to adjust the proof to the adaptive case, some minor modifications are required).

Denote by $\langle r_1, \ldots, r_m \rangle$ the sequence of all m possible outcomes of the coin tosses made by M' (note that $|r_i| = O(\log(|x|))$ and $m = 2^{O(\log(|x|))} = \text{poly}(|x|)$). Denote by $\langle q_1^i, \ldots, q_{n_i}^i \rangle$ the sequence of n_i queries made by M when using the coin sequence r_i (note that n_i is potentially different for each i, and is polynomial in $|x|$). Since M' is non-adaptive, its queries are determined as a function of the input x and the coin sequence r_i, and do not depend on answers to previous queries.

By the completeness condition we are guaranteed that for every x in L there exists a PCP proof π_x, such that the verifier M' always accepts x when given access to π_x. A natural candidate for an *$\mathcal{NP}$-witness* for x would be π_x. However, as already stated in subsection 12.2.3, π_x might be of exponential size in $|x|$, and therefore unsuitable to be used as an *$\mathcal{NP}$-witness*. We will therefore use a "compressed" version of π_x, this version corresponds to the portion of the proof which is actually being read by the verifier M'.

We now turn to the construction of a witness w, given $x \in L$ and a corresponding oracle π_x (for the sake of simplicity we denote it by π). Consider all possible executions of M' on input x given access to the oracle string π (each execution depends on the coin sequence r_i). Take the substring of π containing all the bits examined by M' during these executions (i.e. $\left\{ \langle \pi_{q_1^i}, \ldots, \pi_{q_{n_i}^i} \rangle \right\}_{i=1}^{m}$). Encode each entry in this substring as $\langle index, \pi_{index} \rangle$ (that is, $\langle query, answer \rangle$), denote the resulting encoded string by w_x^π (note that now $|w_x^\pi|$ is polynomial in $|x|$).

We now describe the non-deterministic machine M which decides L in polynomial time. Given input x, and w on the guess tape, M will simulate the execution of M' on input x for all possible r_i's. Every query made by M' will be answered by M according to the corresponding answers appearing in w (by performing binary search on the indices in w). The machine M will accept *if and only if* M' would have accepted x for **all** possible r_i's.

Since M simulates the execution of M' exactly m times (which is polynomial in $|x|$), and since M' is a polynomial time machine, then M is itself a polynomial-time machine, as required. It remains to be seen that $L(M)$ indeed equals L:

- $\forall x \in L$, we show that there exists w such that $M(x,w) = 1$. By the perfect completeness condition of the PCP system for L, there exists an oracle π such that $\Pr[M'^{\pi}(x) = 1] = 1$. Therefore, it holds that for **all** coin sequences r_i, the machine M' accepts x while accessing π. It immediately follows by definition that $M(x, w_x^\pi) = 1$, where w_x^π is as described above.

- $\forall x \notin L$, we show that for **all** w's it holds that $M(x,w) = 0$. By the soundness condition of the PCP system for L, for all oracles π it holds that $\Pr[M'^{\pi}(x) = 1] \leq \frac{1}{2}$. Therefore, for at least $\frac{1}{2}$ of the possible coin sequences r_i, M does not accept x while accessing π. Assume, for the sake of contradiction, that there exists a witness w for which it holds that $M(x,w) = 1$. By the definition of M this means that for all possible coin tosses M' accepts x when given answers from w. We can therefore use w in order to construct an oracle, π^w, for which it holds that $\Pr\left[M'^{\pi^w}(x) = 1\right] = 1$, in contradiction to the soundness condition. (the oracle π^w can be constructed as follows: for every index q that appears in w, define π_q^w to be the binary answer corresponding to q. Define the rest of π^w arbitrarily.)

Consider now the case of an adaptive M'. In this case, we can construct w_x^π adaptively. Given an input $x \in L$ and a corresponding oracle π, run M'^{π} on x for every random string r_i, and see what

are the queries made by M' (which depend on x, r_i and answers to previous queries). Then take w_x^π to be the substring of π that is defined by all these queries, as before. ■

The essence of the above proof, is that given a PCP proof (of logarithmic randomness) for some x in L we can efficiently "pack" it (compress it into polynomial size) and transform it into an $\mathcal{NP}$-witness for x. This is due to the fact that the **total** portion of the proof used by the verifier (in all possible runs, i.e. over all possible coin sequences) is bounded by a polynomial. In light of the above, any result of the type

$$\mathcal{NP} \subseteq \mathcal{PCP}(\log, q(\cdot))$$

would be interesting, since it implies that for every $x \in L$, we can construct a witness with the additional property, that enables a "lazy" verifier to toss coins, and decide membership in L, based only on a tiny portion of the $\mathcal{NP}$-witness (as will be further discussed in subsection 12.3.3).

It turns out that the polynomial $q(\cdot)$ bounding the number of queries in a result of the above kind can be taken to be a constant. This surprising result is what we refer to as the PCP theorem.

Theorem 12.3 (The PCP Theorem)

$$\mathcal{NP} \subseteq \mathcal{PCP}(\log, O(1))$$

The PCP theorem is a culmination of a sequence of works, each establishing a meaningful and increasingly stronger statement. The proof of the PCP theorem is one of the most complicated proofs in the theory of computation and it is beyond our scope to prove it here. We state as a side remark, that the smallest possible number of queries for which the PCP theorem has been proven is currently 5 (whereas with 3 queries one can get arbitrarily close to soundness error 1/2).

The conclusion is that $\mathcal{NP}$ is *exactly* the set of languages which have a PCP verifier that asks a constant number of queries using a logarithmic number of coin tosses.

Corollary 12.4 (The PCP Characterization of $\mathcal{NP}$)

$$\mathcal{NP} = \mathcal{PCP}(\log, O(1))$$

Proof: Combining Theorem 12.3 with Proposition 12.3.1, we obtain the desired result. ■

12.3.3 The PCP Theorem gives rise to "robust" $\mathcal{NP}$-relations

Recall that every language L in $\mathcal{NP}$ can be associated with an $\mathcal{NP}$-relation R_L (in case the language is natural, so is the relation). This relation consists of all pairs (x, y) where x is a positive instance of L and y is a corresponding $\mathcal{NP}$-*witness*. The PCP theorem gives rise to another (unnatural) relation R'_L with some extra properties. In the following subsection we briefly discuss some of the issues regarding the relation R'_L.

Since every $L \in \mathcal{NP}$ has a $\mathcal{PCP}(\log, O(1))$ system we are guaranteed that for every x in L there exists a PCP proof π_x, such that the corresponding verifier machine M always accepts x when given access to π_x. In order to define our relation we would like to consider pairs of the form (x, π_x). However, in general, π_x might be of exponential size in $|x|$, and therefore unsuitable to be used in an $\mathcal{NP}$-relation. In order to "compress" it into polynomial size we can use the construction introduced in the proof of Proposition 12.3.1 (i.e. of a witness w for the non-deterministic machine M). Denote by π'_x the resulting "compressed" version of π_x. We are now ready to define the relation:

$$R'_L \stackrel{\text{def}}{=} \{(x, \pi'_x) \mid \Pr[M^{\pi_x}(x) = 1] = 1\}$$

By the definition of PCP it is obvious that $x \in L$ *if and only if* there exists π'_x such that $(x, \pi'_x) \in R'_L$. It follows from the details in the proof of proposition 12.3.1 that R'_L is indeed recognizable in polynomial-time.

Although not stated in the theorem, the proof of the PCP theorem actually demonstrates how to efficiently transform an $\mathcal{NP}$-witness y (for an instance x of $L \in \mathcal{NP}$) into an oracle proof $\pi_{x,y}$ for which the PCP verifier always accepts x. Thus, there is a Levin-reduction between the natural $\mathcal{NP}$-relation for L and R'_L.

We conclude that any $\mathcal{NP}$-witness of R_L can be efficiently transformed into an $\mathcal{NP}$-witness of R'_L (i.e. an oracle proof) which offers a trade-off between the portion of the $\mathcal{NP}$-witness being read by the verifier and the amount of certainty it has in its answer. That is, if the verifier is willing to tolerate an error probability of 2^{-k}, it needs to inspect $O(k)$ bits of the proof (the verifier chooses k random strings $r_1, \ldots, r_k$ uniformly among $\{0,1\}^{O(\log)}$. It will be convinced with probability 2^{-k} that the input x is in L, if for every i, M accepts x using randomness r_i and given oracle access to the appropriate $O(1)$ queries).

12.3.4 Simplifying assumptions about $\mathcal{PCP}(\log, O(1))$ verifiers

When considering a $\mathcal{PCP}(\log, O(1))$ system, some simplifying assumptions about the corresponding verifier machine can be made. We now turn to introduce two of them:

1. Any verifier in a $\mathcal{PCP}(\log, O(1))$ system can be assumed to be non-adaptive (i.e. its queries are determined as a function of the input and the random tape only, and do not depend on answers to previous queries). This is due to the fact that any adaptive $\mathcal{PCP}(\log, O(1))$ verifier can be converted into a non-adaptive one by modifying it in such a way that it will consider **all** possible sequences of $\{0,1\}$ answers given to its queries by the oracle. This certainly costs us in an exponential blowup in the query complexity, but, since the number of queries made by the original (adaptive) verifier is constant, so will be the query complexity of the modified (non-adaptive) verifier after the blowup. Note that in general, adaptive verifiers *are* more powerful than non-adaptive ones (in terms of quantitative results). There are constructions in which adaptive verifiers make less queries than non-adaptive ones while achieving the same results.

2. Any verifier in a $\mathcal{PCP}(\log, O(1))$ system can be assumed to always make the same (constant) number of queries (regardless of the outcome of its coin tosses). Take any verifier in a $\mathcal{PCP}(\log, O(1))$ system not satisfying the above property. Let t be the maximal number of queries made in some execution of the above verifier (over all possible outcomes of the coin tosses). For every possible outcome of the coin tosses, modify the verifier in such a way that it will ask a total number of t queries, make him ignore answers to the newly added queries. Clearly, such a verifier will be consistent with the original one, and will still make only a constant number of queries (which is t).

From now on, whenever referring to $\mathcal{PCP}(\log, O(1))$ systems, free use of the above assumptions will be made (without any loss of generality).

12.4 PCP and non-approximability

Many natural optimization problems are known to be $\mathcal{NP}$-*hard*. However, many times an approximation to the exact value of the solution could be sufficient for our needs. In this section we will investigate the existence (or rather, the inexistence) of efficient approximation algorithms for two $\mathcal{NP}$-*complete* problems, namely, $max3SAT$ and $maxCLIQUE$.

An algorithm for a given problem is considered a C-*approximation* algorithm if for every instance it generates an answer that is off the correct answer by a factor of at most C. The question of interest, is given an $\mathcal{NP}$-*complete* problem Π, what is the best C for which there is a C-approximation algorithm for Π.

The PCP characterization of $\mathcal{NP}$ provides us an alternative view of languages in $\mathcal{NP}$. This view is not as rigid as the original one, and thus creates a framework which is apparently more insightful for the study of approximability.

We start by rephrasing the PCP theorem in an alternative way. This in turn will be used in order to derive an immediate non-approximability result for $max3SAT$. While rephrasing the PCP theorem, a new type of polynomial-time reductions, which we call *amplifying*, emerges.

12.4.1 Amplifying Reductions

Consider an unsatisfiable $3CNF$ formula[1]. It may be the case that the formula is very "close" to being satisfiable. For example, there exist unsatisfiable formulae such that by removing only one of their clauses, they suddenly become satisfiable.

In contrast, there exist unsatisfiable $3CNF$ formulae which are much "farther" from being satisfiable than the above mentioned formulae. These formulae may always have a constant fraction of unsatisfied clauses (for all possible truth assignments). As a consequence, they offer us the (most attractive) feature of being able to probabilistically check whether a certain truth assignment satisfies them or not (by randomly sampling their clauses and picking with constant probability a clause which is unsatisfied by this assignment). Not surprisingly, this resembles the features of a PCP system.

Loosely speaking, amplifying reductions of 3SAT [2] to itself are *Karp-reductions*, which, in addition to the conventional properties, have the property that they map unsatisfiable $3CNF$ formulae into unsatisfiable $3CNF$ formulae which are "far" from being satisfiable (in the above sense).

Definition 12.5 (amplifying reduction) *An* amplifying reduction *of* 3SAT *to itself is a polynomial-time computable function f mapping the set of $3CNF$ formulae to itself such that for some constant $\epsilon > 0$ it holds that:*

- *f maps satisfiable $3CNF$ formulae to satisfiable $3CNF$ formulae.*
- *f maps non-satisfiable $3CNF$ formulae to (non-satisfiable) $3CNF$ formulae for which every truth assignment satisfies at most an $1 - \epsilon$ fraction of the clauses.*

An amplifying reduction of a language L in $\mathcal{NP}$ to 3SAT, can be defined analogously.

[1] Recall that a t-CNF formula is a boolean formula consisting of a conjunction of clauses, where each clause is a disjunction of up to t literals (a literal is a variable or its negation).

[2] 3SAT is the problem of deciding whether a given $3CNF$ formula has a satisfying truth assignment.

12.4.2 PCP Theorem Rephrased

Amplifying reductions seem like a suitable tool to be used in order to construct a PCP system for every language in $\mathcal{NP}$. Not only they are efficiently computable, but they enable us to map negative instances of any language in $\mathcal{NP}$ into negative instances of 3SAT which we may be able to reject on a probabilistic basis (analogously to the soundness condition in the PCP definition).

It turns out that the converse is also true, given a PCP system for a language in $\mathcal{NP}$ we are also able to construct an amplifying reduction of 3SAT to itself.

Theorem 12.6 (PCP theorem rephrased) *The following are equivalent:*

1. $\mathcal{NP} \subseteq \mathcal{PCP}(\log, O(1))$. (The PCP Theorem).
2. *There exists an amplifying reduction of* 3SAT *to itself.*

Proof: We start with the ((1) $\Rightarrow$ (2)) direction. Consider any language $L \in \mathcal{NP}$. By the PCP theorem L has a $\mathcal{PCP}(\log, O(1))$ system, we will now show how to use this system in order to construct an amplifying reduction from L to 3SAT. This will in particular hold for $L =$ 3SAT (which is itself in $\mathcal{NP}$), and the claim will follow.

Let M be the probabilistic polynomial-time oracle machine in the above $\mathcal{PCP}(\log, O(1))$ system for L. We are guaranteed that on input $x \in \{0,1\}^*$, M makes $t = O(1)$ queries using $O(\log(|x|))$ coin tosses.

Denote by $\langle r_1, \ldots, r_m \rangle$ the sequence of all m possible outcomes of the coin tosses made by M (note that $|r_i| = O(\log(|x|))$ and $m = 2^{O(\log(|x|))} = \text{poly}(|x|)$).

Denote by $\langle q_1^i, \ldots, q_t^i \rangle$ the sequence of t queries made by M when using the coin sequence r_i. As mentioned in subsection 12.3.4, we can assume that M is non-adaptive, therefore its queries are determined as a function of the input x and the coin sequence r_i, and do not depend on answers to previous queries (although not evident from the notation q_j^i, the queries do not depend only on r_i, but on x as well).

We now turn to the construction of the amplifying reduction. Given $x \in \{0,1\}^*$, we construct for each r_i a (constant size) $3CNF$ boolean formula, φ_i^x, describing whether M would have accepted the input x (i.e. describing all possible outputs of M on input x, using the coin sequence r_i). We associate to each query q_j^i a boolean variable $z_{q_j^i}$ whose value should be the answer M gets to the corresponding query. Again, since M is assumed to be non-adaptive, when given its input and coin tosses, M's decision is completely determined by the answers it gets to its queries. In other words, M's decision depends only on the values of $\langle z_{q_1^i}, \ldots, z_{q_t^i} \rangle$.

In order to construct φ_i^x, begin by computing the following truth table: to every possible sequence $\langle z_{q_1^i}, \ldots, z_{q_t^i} \rangle$ assign the corresponding boolean decision of M (i.e. the output of M on input x, using the coin sequence r_i, and given answers $z_{q_j^i}$ to queries q_j^i). Clearly, this can be computed in polynomial-time (by simulating M's execution). Therefore, the whole table can be computed in polynomial-time (since the number of possible assignments to $\langle z_{q_1^i}, \ldots, z_{q_t^i} \rangle$ is 2^t, which is a constant). We can now build a $3CNF$ boolean formula, φ_i^x, which is consistent with the above truth table, this is done in the following way:

1. Construct a t-CNF formula $\psi_i^x = \psi_i^x(z_{q_1^i}, \ldots, z_{q_t^i})$ which is consistent with the truth table.
2. Using a constant number of auxiliary variables, transform it to $3CNF$ (denoted φ_i^x).

Since the table size is constant, the above procedure can be executed in constant time. Note that in the transformation of t-CNF formulae into $3CNF$ formuae, each clause with t literals is substituted by at most t clauses of 3 literals. Since ψ_i^x consists of exactly 2^t clauses we conclude that the number of clauses in φ_i^x is bounded by $t \cdot 2^t$.

Finally, given φ_i^x for $i = 1, \ldots, m$, we let our amplifying reduction f map $x \in \{0,1\}^*$ into the $3CNF$ boolean formula:

$$\varphi^x \stackrel{\text{def}}{=} \bigwedge_{i=1}^{m} \varphi_i^x$$

Since for every $i = 1, \ldots, m$ the (constant size) formula φ_i^x can be computed in polynomial-time (in $|x|$), and since $m = \text{poly}(|x|)$, it follows that the mapping $f : x \mapsto \varphi^x$ is polynomial-time computable, and $|\varphi^x|$ is polynomial in $|x|$ (note also that the number of clauses in φ^x is bounded by $m \cdot t \cdot 2^t$). It remains to be verified that f is indeed an amplifying reduction:

- $\forall x \in L$, we now show that φ^x is in 3SAT, this happens if and only if the corresponding t-CNF formula $\psi^x \stackrel{\text{def}}{=} \bigwedge_{i=1}^{m} \psi_i^x(z_{q_1^i}, \ldots, z_{q_t^i})$ is in t-SAT (recall that ψ_i^x was introduced in the construction of φ_i^x). Since $L \in \text{PCP}$, then there exists an oracle π such that $\Pr[M^\pi(x) = 1] = 1$. Therefore, it holds that for every coin sequence r_i, the machine M accepts x while accessing π. Since ψ_i^x is consistent with the above mentioned truth table it follows that for **all** $i = 1, \ldots, m$, it holds that $\psi_i^x(\pi_{q_1^i}, \ldots, \pi_{q_t^i}) = 1$, and thus ψ^x is in t-SAT. We conclude that φ^x is in 3SAT, as required [3].

- $\forall x \notin L$, we now show that every truth assignment satisfies at most an $1 - \epsilon$ fraction of φ^x's clauses. Since $L \in \text{PCP}$, then for all oracles π it holds that $\Pr[M^\pi(x) = 1] \leq \frac{1}{2}$. Therefore, for at least $\frac{1}{2}$ of the possible coin sequences r_i, machine M does not accept x while accessing π. Put in other words, for each truth assignment (which corresponds to some π) at least $\frac{1}{2}$ of the φ_i^x's are unsatisfiable. Since every unsatisfiable boolean formula always has at least one unsatisfied clause, it follows that for every truth assignment φ^x has at least $\frac{m}{2}$ unsatisfied clauses. Since the number of clauses in φ^x is bounded by $m \cdot t \cdot 2^t$, by taking ϵ to be the constant $\frac{1}{2 \cdot t \cdot 2^t}$ we are guaranteed that every truth assignment satisfies at most an $1 - \epsilon$ fraction of φ^x's clauses.

We now turn to the $((2) \Rightarrow (1))$ direction. Under the assumption that there exists an amplifying reduction of 3SAT to itself we will show that the PCP theorem holds. Consider any language $L \in \mathcal{NP}$. Since L is *Karp-reducible* to 3SAT, it is sufficient to show that $\text{3SAT} \in \mathcal{PCP}(\log, O(1))$.

Let $f : 3CNF \to 3CNF$ be an amplifying reduction of 3SAT to itself. And let ϵ be the constant guaranteed by Definition 12.5. We now show how to use f in order to construct a $\mathcal{PCP}(\log, O(1))$ system for 3SAT. We start by giving an informal description of the verifier machine M. Given a certain $3CNF$ formula φ, M computes $\varphi' = f(\varphi)$. It then tosses coins in order to uniformly choose one of the clauses of φ'. By querying the oracle string (which should be a possible truth assignment for φ') M will assign truth values to the chosen clause's variables. M will accept if and only if the clause is satisfied. The fact that f is an amplifying reduction implies that whenever M gets a negative instance of 3SAT, with constant probability the chosen clause will not be satisfied. In contrast, this will never happen when looking at a positive instance.

We now turn to a more formal definition of the PCP verifier machine M. On input $\varphi \in 3CNF$ and given access to an oracle string π', M is defined in the following way:

[3] Note that all φ_i^x's have disjoint sets of auxiliary variables, hence transforming a satisfying assignment of ψ^x into a satisfying assignment of φ^x causes no inconsistencies.

1. Find the $3CNF$ formula $\varphi' = \varphi'(x_1, \ldots, x_{n'}) \stackrel{\text{def}}{=} f(\varphi)$.
   $\varphi' = \bigwedge_{i=1}^{m'} c_i$ where c_i denotes a clause with 3 literals.
2. Select a clause c_i of φ' uniformly.
   Denote by $\langle x_{i_1}, x_{i_2}, x_{i_3} \rangle$ the three variables whose literals appear in c_i.
3. Query the values of $\langle \pi'_{i_1}, \pi'_{i_2}, \pi'_{i_3} \rangle$ separately, and assign them to $\langle x_{i_1}, x_{i_2}, x_{i_3} \rangle$ accordingly. Verify the truth value of $c_i = c_i(x_{i_1}, x_{i_2}, x_{i_3})$.
4. Repeat stages 2,3 for $\lceil \frac{1}{\epsilon} \rceil$ times independently (note that $\lceil \frac{1}{\epsilon} \rceil$ is constant).
5. Output 1 if and only if in **all** iterations the truth value of c_i was 1.

Clearly, M is a polynomial-time machine. Note that f is computable in polynomial-time (this also implies that $n', m' = \text{poly}(|\varphi|)$). In addition, the number of iterations executed by M is constant, and in each iteration a polynomial amount of work is executed (depending on n', m' which are, as already mentioned, polynomial in $|\varphi|$).

We turn to evaluate the additional complexity measures involved. In terms of randomness, M needs to uniformly choose $\lceil \frac{1}{\epsilon} \rceil$ numbers in the set $\{1, ..., m'\}$. This involves $O(\log(m')) = O(\log(|\varphi|))$ coin tosses, as required. In terms of queries, the number of queries asked by M is exactly $\frac{3}{\epsilon}$ which is constant, again as required. It remains to examine the completeness and soundness of the above PCP system:

- *completeness*: If $\varphi \in$ 3SAT, then $\varphi' \in$ 3SAT (since f is an amplifying reduction). Therefore there exists a truth assignment, π', such that $\varphi'(\pi') = 1$. Now, since every clause of φ' is satisfied by π', it immediately follows that:

$$\Pr\left[M^{\pi'}(\varphi) = 1\right] = 1$$

- *soundness*: If $\varphi \notin$ 3SAT then any truth assignment for φ' satisfies at most an $1 - \epsilon$ fraction of the clauses. Therefore for any possible truth assignment (oracle) π' it holds that

$$\Pr\left[M^{\pi'}(\varphi) = 1\right] = \Pr\left[\bigwedge_{j=1}^{\lceil \frac{1}{\epsilon} \rceil} \left\{ \begin{array}{c} c_{i_j} \text{ is satisfied by} \\ \text{the assignment } \pi' \end{array} \right\}\right] \leq (1-\epsilon)^{\lceil \frac{1}{\epsilon} \rceil} \leq \frac{1}{e} < \frac{1}{2}$$

where the probability is taken over M's internal coin tosses (i.e. over the choice of $i_1, \ldots, i_{\lceil \frac{1}{\epsilon} \rceil}$). ■

Corollary 12.7 *There exists an amplifying reduction of* 3SAT *to itself.*

Proof: Combining the PCP Theorem with Theorem 12.6, we obtain the desired result. ■

12.4.3 Connecting PCP and non-approximability

The characterization of $\mathcal{NP}$ using probabilistic checkable proof systems enabled the area of approximability to make a significant progress.

In general, PCP systems for $\mathcal{NP}$ yield strong non-approximability results for various classical optimization problems. The hardness of approximation is typically established using the notion

of gap problems, which are a particular case of promise problems. Recall that a promise problem consists of two sets (A, B), where A is the set of YES instances and B is the set of NO instances. A and B need not be complementary, that is, an instance $x \in \{0,1\}^*$ is not necessarily in either A or B. We demonstrate the notion of a gap problem using the promise problem $gapCLIQUE$ as an example.

Denote by $maxCLIQUE(G)$ the size of the maximal clique in a graph G. Let $gapCLIQUE_{\alpha,\beta}$ be the promise problem (A, B) where A is the set of all graphs G with $maxCLIQUE(G) \geq \alpha$, and B is the set of all graphs G with $maxCLIQUE(G) \leq \beta$. The gap is defined as α/β. Typically, a hardness result will specify a value C of the gap for which the problem is $\mathcal{NP}$-*hard*. This means that there is no efficient algorithm that approximates the $maxCLIQUE$ size of a graph G within a factor of C (unless $\mathcal{NP} = \mathcal{P}$).

The gap versions of various other optimization problems are defined in an analogous way. In this subsection we bring two non-approximability results concerning the problems $max3SAT$ and $maxCLIQUE$ that will be defined in the sequel.

An immediate non-approximability result for max3SAT

Definition 12.8 *(max3SAT): Define max3SAT to be the following problem: Given a $3CNF$ boolean formula φ find the maximal number of clauses that can be simultaneously satisfied by any truth assignment to the variables of φ.*

$max3SAT$ is known to be $\mathcal{NP}$-*hard*. Therefore, approximating it would be desirable. This motivates the definition of the corresponding gap problem:

Definition 12.9 $(gap3SAT_{\alpha,\beta})$: *Let $\alpha, \beta \in [0,1]$ such that, $\alpha \geq \beta$.*
Define $gap3SAT_{\alpha,\beta}$ to be the following promise problem:

- *The YES instances are all $3CNF$ formulae φ, such that there exists a truth assignment which satisfies at least an α-fraction of the clauses of φ.*
- *The NO instances are all $3CNF$ formulae φ, such that every truth assignment satisfies less than a β-fraction of the clauses of φ.*

Note that $gap3SAT_{1,1}$ is an alternative formulation of 3SAT (the decision problem).

The following claim states that, for some $\beta < 1$, it is $\mathcal{NP}$-*hard* to distinguish between satisfiable $3CNF$ formulae and $3CNF$ formulae for which no truth assignment satisfies more than a β-fraction of its clauses. This result implies that there is some constant $C > 1$ such that $max3SAT$ could not be approximated within C (unless $\mathcal{NP} = \mathcal{P}$). The claim is an immediate result of Corollary 12.7.

Claim 12.4.1 *There exists a constant $\beta < 1$, such that the promise problem $gap3SAT_{1,\beta}$ is $\mathcal{NP}$-hard.*

Proof: Let $L \in \mathcal{NP}$. We want to manifest that L is *Karp-reducible* to $gap3SAT_{1,\beta}$.

$3SAT$ is $\mathcal{NP}$-complete, therefore there exists a *Karp-reduction* f_1 from L to $3SAT$. By Corollary 12.7 there exists an amplifying reduction f_2 (and a constant $\epsilon > 0$) from $3SAT$ to itself. Now, take any $1 - \epsilon < \beta < 1$:

- For $x \in L$, $\varphi = f_2(f_1(x))$ is satisfiable, and is therefore a YES instances of $gap3SAT_{1,\beta}$.
- For $x \notin L$, $\varphi = f_2(f_1(x))$ is not satisfiable. Furthermore, for every truth assignment, the fraction of satisfied clauses in φ is at most $1 - \epsilon$. Therefore, φ is a NO instance of $gap3SAT_{1,\beta}$.

■

Recently, stronger results were proven. These results show that for every $\beta > 7/8$, the problem $gap3SAT_{1,\beta}$ is $\mathcal{NP}$-*hard*. This means that it is infeasible to come with an efficient algorithm that approximates $max3SAT$ within a factor strictly smaller than $8/7$. On the other hand, $gap3SAT_{1,7/8}$ is known to be polynomially solvable, and therefore the $8/7$-approximation ratio is tight.

MaxCLIQUE is non-approximable within a factor of two

We briefly review the definitions of the problems $maxCLIQUE$ and $gapCLIQUE_{\alpha,\beta}$:

Definition 12.10 *(maxCLIQUE): Define maxCLIQUE to be the following problem: Given a graph G, find the size of the maximal clique of G (a clique is a set of vertices such that every pair of vertices share an edge).*

$maxCLIQUE$ is known to be $\mathcal{NP}$-*hard*. Therefore, approximating it would be desirable. This motivates the definition of the corresponding gap problem:

Definition 12.11 ($gapCLIQUE_{\alpha,\beta}$) : *Let $\alpha, \beta : N \to N$ be two functions, satisfying $\alpha(n) \geq \beta(n)$ for every n. For a graph G, denote $|G|$ to be the number of vertices in G.*

Define $gapCLIQUE_{\alpha,\beta}$ to be the following promise problem:

- *The YES instances are all the graphs G, with max clique greater than or equal to $\alpha(|G|)$.*
- *The NO instances are all the graphs G, with max clique smaller than or equal to $\beta(|G|)$.*

We conclude our discussion on PCP systems by presenting a nice theorem which demonstrates the hardness of approximating $maxCLIQUE$. The theorem implies that it is an infeasible task to approximate $maxCLIQUE$ within a constant smaller than two (unless $\mathcal{NP} = \mathcal{P}$).

Note, however, that this is not the strongest result known. It has been shown recently that given a graph G of size N, the value of $maxCLIQUE$ is non-approximable within a factor of $N^{1-\epsilon}$ (for every $\epsilon > 0$). This result is tight, since an $N^{1-o(1)}$-approximation algorithm is known to exist (the latter is scarcely better than the trivial approximation factor of N).

Theorem 12.12 *There exists a function $\alpha : N \to N$, such that the promise problem $gapCLIQUE_{\alpha,\alpha/2}$ is $\mathcal{NP}$-hard.*

Proof: Let $L \in \mathcal{NP}$ be some language. We want to show that L is *Karp-reducible* to the language $gapCLIQUE_{\alpha,\alpha/2}$ (for some function $\alpha : N \to N$ which is not dependent on L, rather it is common to all L's).

Loosely speaking, given input $x \in \{0,1\}^*$ we construct in an efficient way a graph G_x having the following property: If x is in L then G_x is a YES instance of $gapCLIQUE$, whereas, if x is not in L then G_x is a NO instance of $gapCLIQUE$.

We now turn to a formal definition of the above mentioned reduction. By the PCP-theorem, L has a $PCP(O(\log), O(1))$ system. Therefore, there exists a probabilistic polynomial-time oracle machine M, that on input $x \in \{0,1\}^*$ makes $t = O(1)$ queries using $O(\log(|x|))$ random coin tosses.

Again, we let $\langle r_1, \ldots, r_m \rangle$ be the sequence of all m possible outcomes of the coin tosses made by M (note that $m = \text{poly}(|x|)$).

Let $\langle q_1^i, \ldots, q_t^i \rangle$ denote the t queries made by M when using the coin tosses r_i, and let $\langle a_1^i, \ldots, a_t^i \rangle$ be a possible sequence of answers to the corresponding queries. We now turn to define a graph G'_x that corresponds to machine M and input x:

vertices: For every possible r_i, the tuple

$$(\mathbf{r_i}, (\mathbf{q_1^i}, a_1^i), \ldots, (\mathbf{q_t^i}, a_t^i))$$

is a vertex in G'_x *if and only if* when using r_i, and given answers a_j^i to queries q_j^i, M accepts x. Note that since M is non-adaptive, once r_i is fixed then so are the queries $\langle q_1^i, \ldots, q_t^i \rangle$. This implies that two vertices having the same r_i, also have the same q_j^i's. Therefore, the number of vertices corresponding to a certain r_i is smaller or equal to 2^t, and the total number of vertices in G'_x is smaller or equal to $m \cdot 2^t$.

edges: Two vertices $v = (\mathbf{r_i}, (\mathbf{q_1^i}, a_1^i), \ldots, (\mathbf{q_t^i}, a_t^i))$ and $u = (\mathbf{r_j}, (\mathbf{q_1^j}, a_1^j), \ldots, (\mathbf{q_t^j}, a_t^j))$ will **not** have an edge between them *if and only if* they are not *consistent*, that is, v and u contain the same query and each one of them has a different answer to this query.

Note that if u and v contain the same randomness (i.e. r_i is equal to r_j) they do not share an edge, since they cannot be consistent (as mentioned earlier, vertices having the same randomness have also the same queries, so u and v must differ in the answers to the queries).

Finally, modify G'_x by adding to it $(m \cdot 2^t - |G'_x|)$ isolated vertices. The resulting graph will have exactly $m \cdot 2^t$ vertices and will be denoted G_x. Note that since the above modification does not add any edges to G'_x, it does not change the size of any clique in G'_x (in particular it holds that $maxCLIQUE(G_x) = maxCLIQUE(G'_x)$).

The above reduction is efficient, since the graph G_x can be constructed in polynomial time: There are at most $m \cdot 2^t$ vertices in G'_x(which is polynomial in $|x|$ since t is a constant), and to decide whether $(\mathbf{r_i}, (\mathbf{q_1^i}, a_1^i), \ldots, (\mathbf{q_t^i}, a_t^i))$ is a vertex in G'_x, one has to simulate machine M on input x, randomness r_i, queries $\{q_j^i\}_{j=1}^t$ and answers $\{a_j^i\}_{j=1}^t$ and see whether it accepts or not. This is, of course, polynomial, since M is polynomial. Finally, deciding whether two vertices share an edge can be done in polynomial time.

Let $\alpha(n) \stackrel{\text{def}}{=} n/2^t$. Since $|G_x| = m \cdot 2^t$, it holds that $\alpha(|G_x|) = m$. It is therefore sufficient to show a reduction from the language L to $gapCLIQUE_{m,m/2}$, this will imply that the promise problem $gapCLIQUE_{\alpha,\alpha/2}$ is $\mathcal{NP}$-*hard*.

- For $x \in L$, we show that G_x contains a clique of size m. By the PCP definition there exists a proof π such that for every random string r, machine M accepts x using randomness r and given oracle access to π. Look at the following set of m vertices in the graph G_x: $S = \{(\mathbf{r_i}, (\mathbf{q_1^i}, \pi_{q_1^i}), \ldots, (\mathbf{q_t^i}, \pi_{q_t^i})) \text{ for } 1 \leq i \leq m\}$. It is easy to see that all the vertices in S are indeed legal vertices, because π is a proof for x. Also, all the vertices in S must be consistent, because all their answers are given according to π, and therefore, every two vertices in S share an edge. This entails that S is an m-clique in G_x, and therefore G_x is a YES instance of $gapCLIQUE_{m,m/2}$.

- For $x \notin L$, we show that G_x does not contain a clique of size greater than $m/2$. Suppose, in contradiction, that S is a clique in G_x of size greater than $m/2$. Define now the following proof π: For every query and answer (q, a) in one of the vertices of S define $\pi_q = a$. For every other query (which is not included in either of the vertices of S) define π_q to be an arbitrary value in {0,1}. Since S is a clique, all its vertices share an edge and are therefore consistent. Note that π is well defined, the consistency requirement implies that same queries have same answers (for all queries and answers appearing in some vertex in S). Therefore, it cannot be the case that we give two incosistent values to the same entry in π during its construction. Now, since all the vertices of S have different r_i's and $|S|$ is greater than $m/2$, it holds that

for more than $\frac{1}{2}$ of the possible coin sequences r_i, machine M accepts x while accessing π. In other words, $\Pr[M^{\pi}(x) = 1] > 1/2$, in contradiction to the soundness condition. We conclude that indeed G_x does not have a clique of size greater than $m/2$, and is therefore a NO instance of $gapCLIQUE_{m,m/2}$.

■

Bibliographic Notes

The PCP Characterization Theorem is attributed to Arora, Lund, Motwani, Safra, Sudan and Szegedy (see [2] and [1]). These papers, in turn, built on numerous previous works; for details see the papers themselves or [4]. In general, our presentation of PCP follows follows Section 2.4 of [4], and the interested reader is referred to [4] for a survey of further developments and more refined considerations.

The first connection between PCP and hardness of approximation was made by Feige, Goldwasser, Lovasz, Safra, and Szegedy [3]: They showed the connection to maxClique (presented above). The connection to max3SAT and other "MaxSNP approximation" problems was made later in [1].

We did not present the strongest known non-approximability results for max3SAT and maxClique. These can be found in Hastad's papers, [6] and [5], respectively.

1. S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szegedy. Proof Verification and Intractability of Approximation Problems. *JACM*, Vol. 45, pages 501–555, 1998.

2. S. Arora and S. Safra. Probabilistic Checkable Proofs: A New Characterization of NP. *JACM*, Vol. 45, pages 70–122, 1998.

3. U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating Clique is almost NP-complete. *JACM*, Vol. 43, pages 268–292, 1996.

4. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. Algorithms and Combinatorics series (Vol. 17), *Springer*, 1998.

5. J. Hastad. Clique is hard to approximate within $n^{1-\epsilon}$. To appear in *ACTA Mathematica*. Preliminary versions in *28th STOC* (1996) and *37th FOCS* (1996).

6. J. Hastad. Getting optimal in-approximability results. In *29th STOC*, pages 1–10, 1997.

Lecture 13

Pseudorandom Generators

Notes taken by Sergey Benditkis, Boris Temkin and Il'ya Safro

Summary: Pseudorandom generators are defined as efficient deterministic algorithms which stretch short random seeds into longer pseudorandom sequences. The latter are indistiguishable from truely random sequences by any efficient observer. We show that, for efficiently sampleable distributions, computational indistiguishability is preserved under multiple samples. We related pseudorandom generators and one-way functions, and show how to increase the stretching of pseudorandom generators.

13.1 Instead of an introduction

Oded's Note: *See introduction and motivation in the Appendix. Actually, it is recommended to read the appendix before reading the following notes, and to refer to the notes only for full details of some statements made in Sections 13.6.2 and 13.6.3 of the appendix.*

Oded's Note: *Loosely speaking, pseudorandom generators are defined as efficient deterministic algorithms which stretch short random seeds into longer pseudorandom sequences. We stress three aspects: (1) the efficiency of the generator; (2) the stretching of seeds to longer strings; (3) the pseudorandomness of output sequences. The third aspect refers to the key notion of computational indistinguishability. We start with a definition and discussion of the latter.*

13.2 Computational Indistinguishability

We have two things which are called "probability ensembles", and denoted by $\{X_n\}_{n\in\mathsf{N}}$ and $\{Y_n\}_{n\in\mathsf{N}}$. We are talking about infinite sequences of distributions like we talk about the language and each distribution seats on some final domain. Typicaly the distribution X_n will have as a support strings of length polynomial of n, not more and not much less.

Definition 13.1 (probability ensembles): *A* probability ensemble *X is a family $X = \{X_n\}_{n\geq 1}$ such that X_n is a probability distribution on some finite domain.*

What is to say that these ensembles are computational indistinguishable? We want to look at the particular algorithm A and want to ask what is the probability of the event: when you give to A

an input X_n then it says 1 (or 1 is just arbitrary) and look at the difference of the probabilities for answers of execution this algorithm A for two inputs $\{X_n\}_{n\in\mathsf{N}}$ and $\{Y_n\}_{n\in\mathsf{N}}$. And if this difference is negligible, when you look at n as the parameter then we will say that we can not distinguish the first ensamble from the second one.

Definition 13.2 (negligible functions): *The function $f : \mathsf{N} \mapsto [0,1]$ is* negligible *if for all polynomials p, and for all sufficiently large n's, $f(n) < 1/p(n)$.*

Suppose we have two *probability ensambles* $\{X_n\}_{n\in\mathsf{N}}$ and $\{Y_n\}_{n\in\mathsf{N}}$, where X_n and Y_n are distributions over some finite domain.

Definition 13.3 (indistinguishability by a specific algorithm): *Consider some probabilistic algorithm A. We will say that $\{X_n\}$ and $\{Y_n\}$ are* indistinguishable by A *if*

$$|\Pr(A(X_n) = 1) - \Pr(A(Y_n) = 1)| < \frac{1}{p(n)}$$

for every polinomial $p()$ and for every sufficiently large n.

13.2.1 Two variants

Our main focus will be on indistinguishability by any probabilistic polynomial-time algorithm. That is,

Definition 13.4 (canonic notion of computational indistinguishability): *Two probability ansambles $\{X_n\}_{n\in\mathsf{N}}$ and $\{Y_n\}_{n\in\mathsf{N}}$ are* computationally indistinguishable *if they are indistinguishable by any probabilistic polynomial-time algorithm. That is, for every probabilistic polynomial-time algorithm A, and every polinomial $p()$ there exists N s.t. for all $n > N$*

$$|\Pr(A(X_n) = 1) - \Pr(A(Y_n) = 1)| < \frac{1}{p(n)}$$

Another notion that we talk about is indistinguhability by circuits.

Definition 13.5 *Two probability ensembles $\{X_n\}_{n\in\mathsf{N}}$ and $\{Y_n\}_{n\in\mathsf{N}}$ are* indistinguishable by small circuits *if for all families of polynomial-size circuits $\{C_n\}$*

$$|\,[\Pr(C_n(X_n) = 1) - \Pr(C_n(Y_n) = 1)]\,|$$

is neglibible.

13.2.2 Relation to Statistical Closeness

Oded's Note: *This subsection was rewritten by me.*

The notion of Computational Indistinguishability is a relaxation of the notion of *statistical closeness* (or *statistical indistinguishability*).

Definition 13.6 (statistical closeness): *The* statistical difference (or variation distance) *between two distributions, X and Y, is defined by*

$$\Delta(X,Y) \stackrel{\text{def}}{=} \frac{1}{2} \cdot \sum_{\alpha} |\Pr[X=\alpha] - \Pr[Y=\alpha]|$$

Two probability ensembles $\{X_n\}_{n\in\mathsf{N}}$ and $\{Y_n\}_{n\in\mathsf{N}}$ are statistical close *if $\Delta(X_n,Y_n)$ is a negligible function of n. That is, for all polynomial $p()$ there exists N s.t. for all $n > N$ $\Delta(X_n,Y_n) < 1/p(n)$.*

An equivalent definition of $\Delta(X_n,Y_n)$, is the maximum over all subsets, S, of $\Pr[X_n \in S] - \Pr[Y_n \in S]$. (A set S which obtains the maximum is the set of all z's satisfying $\Pr[X_n = z] > \Pr[Y_n = z]$, which proves the equivalence.) Yet another equivalent definition of $\Delta(X_n,Y_n)$ is the maximum over all Boolean f's of $\Pr[f(X_n)=1] - \Pr[f(Y_n)=1]$. Thus,

Proposition 13.2.1 *If two probability ensembles are statistical close then they are computationally indistinguishable.*

We note that there are computationally indistinguishable probability ensembles which are not statistical close.

13.2.3 Computational indistinguishability and multiple samples

Oded's Note: *We show that under certain conditions, computational indistinguishability is preserved under multiple samples.*

Definition 13.7 (constructability of ensembles): *The ensemble $\{Z_n\}_{n\in\mathsf{N}}$ is* probabilistic polynomial-time constructable *if there exists a probabilistic polynomial time algorithm S such that for every n, $S(1_n) \equiv Z_n$.*

Theorem 13.8 *Let $\{X_n\}$ and $\{Y_n\}$ computationally indistinguishable (i.e., indistinguishable by any probabilistic polynomial time algorithm). Suppose they are both probabilistic polynomial time constructable. Let $t()$ be a positive polinomial. Define $\{\overline{X_n}\}_{n\in N}$ and $\{\overline{Y_n}\}_{n\in N}$ in the following way:*

$$\overline{X_n} = X_n^1 \circ X_n^2 \circ \ ... \ \circ X_n^{t(n)}, \quad \overline{Y_n} = Y_n^1 \circ Y_n^2 \circ \ ... \ \circ Y_n^{t(n)}$$

The X_n^i's (resp. Y_n^i's) are independent *copies of X_n (Y_n). Then $\{\overline{X_n}\}$ and $\{\overline{Y_n}\}$ are Probabilistic Polynomial Time indistinguishable.*

Proof: Suppose, there exists a distinguisher D, between $\{\overline{X_n}\}$ and $\{\overline{Y_n}\}$.

Oded's Note: *We use the "hybrid technique": We define hybrid distributions so that the extreme hybrids coincide with $\{\overline{X_n}\}$ and $\{\overline{Y_n}\}$, and link distinguishability of neighboring hybrids to distinguishability of $\{X_n\}$ and $\{Y_n\}$.*

Then define

$$H_n^{(i)} \ = \ \left(X_n^{(1)} \circ X_n^{(2)} \circ \ ... \ X_n^{(i)} \circ Y_n^{(i+1)} \circ \ ... \ Y_n^{(t(n))}\right)$$

It is easy to see that $H_n^{(0)} = \overline{Y_n}$, $\quad H_n^{(t(n))} = \overline{X_n}$.

Oded's Note: *Also note that $H_n^{(i)}$ and $H_n^{(i+1)}$ differ only in the distribution of the $i+1^{\text{st}}$ component, which is identical to Y_n in the first hybrid and to X_n in the second. The idea is to distinguish Y_n and X_n by pluging them in the $i+1^{\text{st}}$ component of a distribution. The new distribution will be distributed identically to either $H_n^{(i)}$ or $H_n^{(i+1)}$, respectively.*

Define algorithm D' as follows:

Begin Algorithm Distinguisher
Input α , (taken from X_n or Y_n)
(1) Choose $i \in_R \{1 \,..\, t(n)\}$ (i.e., uniformly in $\{1, ..., t(n)\}$)
(2) Construct $Z = \left(X_n^{(1)} \circ X_n^{(2)} \circ \ ... \ X_n^{(i-1)} \circ \alpha \circ Y_n^{(i+1)} \circ \ ... \ Y_n^{(t(n))}\right)$
Return $D(Z)$
end.

$$\begin{aligned}
\Pr\left[D'(X_n)=1\right] &= \frac{1}{t(n)} \sum_{i=1}^{t(n)} \Pr\left[D(X_n^{(1)} \circ X_n^{(2)} \circ \ ... \ X_n^{(i-1)} \circ X_n \circ Y_n^{(i+1)} \circ \ ... \ Y_n^{(t(n))}) = 1\right] \\
&= \frac{1}{t(n)} \sum_{i=1}^{t(n)} \Pr\left[D(H_n^{(i)}) = 1\right]
\end{aligned}$$

whereas

$$\begin{aligned}
\Pr\left[D'(Y_n)=1\right] &= \frac{1}{t(n)} \sum_{i=1}^{t(n)} \Pr\left[D(X_n^{(1)} \circ X_n^{(2)} \circ \ ... \ X_n^{(i-1)} \circ Y_n \circ Y_n^{(i+1)} \circ \ ... \ Y_n^{(t(n))}) = 1\right] \\
&= \frac{1}{t(n)} \sum_{i=1}^{t(n)} \Pr\left[D(H_n^{(i-1)}) = 1\right].
\end{aligned}$$

Thus,

$$\begin{aligned}
&\left|\Pr\left[D'(X_n)=1\right] \ - \ \Pr\left[D'(Y_n)=1\right]\right| \\
&= \frac{1}{t(n)} \cdot \left| \sum_{i=1}^{t(n)} \Pr[D(H_n^{(i)})=1] \ - \ \sum_{i=1}^{t(n)} \Pr[D(H_n^{(i-1)})=1] \right| \\
&= \frac{1}{t(n)} \cdot \left| \Pr[D(H_n^{(t(n))})=1] \ - \ \Pr[D(H_n^{(0)})=1] \right| \\
&= \frac{1}{t(n)} \cdot \left|\Pr[D(X_n)=1] \ - \ \Pr[D(Y_n)=1]\right| \ \geq \ \frac{1}{t(n) \cdot p(n)}
\end{aligned}$$

for some $p()$ and for infinitely many n's ■

> Oded's Note: *One can easily show that computational indistinguishability by small circuits is preserved under multiple samples. Here we don't need to assume probabilistic polynomial-time constructability of the ensembles.*

13.3 PRG: Definition and amplification of the stretch function

Intuitively, a pseudo-random generator takes a short, truly-random string and stretches it into a long, pseudorandom one. The pseudorandom string should look "random enough" to use it in place of a truly random string.

Definition 13.9 (PseudoRandom Generator – PRG): *The function* $G : \{0,1\}^* \to \{0,1\}^*$ *with stretch function* $l(n)$ *is a* pseudo-random generator *if:*

- G *is a polynomial time algorithm*
- *for every* x, $|G(x)| = l(|x|) > |x|$
- $\{G(U_n)\}$ *and* $\{U_{l(n)}\}$ *are computational indistinguishable, where* U_m *denotes the uniform distribution over* $\{0,1\}^m$.

Oded's Note: *The above definition is minimalistic regarding its stretch requirement. A generator stretching* n *bits into* $n+1$ *bits seems to be of little use. However, as shown next, such minimal stretch generators can be used to construct generators of arbitrary stretch.*

Theorem 13.10 (amplification of stretch function): *Suppose we have a Pseudo-Random Generator* G_1 *with a stretch function* $n+1$. *Then for all polynome* $l(n)$ *there exists a Pseudo-Random Generator with stretch function* $l(n)$.

Proof:

Construct G as follows: We take the input seed x ($|x| = n$) and feed it through G_1. Then we save the first bit of the output of G_1 (denote it by y_1), and feed the rest of the bits as input to a new invocation of G_1. We repeat this operation $l(n)$ times, in the i-th step we invoke G_1 on input determined in the previous step, save the first output bit as y_i and use the rest n bits as an input to step $i+1$. The output of G is $y = y_1 y_2 \ldots y_{l(n)}$. See Figure 1.

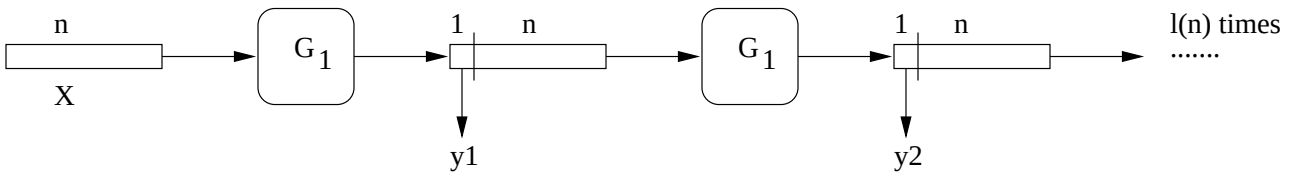


Figure 1.

We claim that G is a Pseudo-Random Generator. The first two requirements for Pseudo-Random Generator are trivial (by construction/definition of G). We will prove the 3rd one. The proof is by contradiction, again using the hybrid method.

Suppose there exists a distinguisher $A : \{0,1\}^* \to \{0,1\}^{l(n)}$ such that exists polynomial $p()$ and for infinitely namy n's

$$\left| \Pr\left[A(G(U_n)) = 1 \right] - \Pr\left[A(U_{l(n)}) = 1 \right] \right| \geq \frac{1}{p(n)}$$

Let us make the following construction . Define sequence of functions $g^{(i)}$:

$g^{(0)}$ is empty

$g^{(i)} = [G_1(x)]_1 \circ g^{(i-1)}([G_1(x)]_{2\ldots(n+1)})$

Where $[y]_i$ is the notation of i-th bit of y and $[y]_{2\ldots(n+1)}$ denotes substring of y from the second bit up to $n+1$-th bit. It is easy to see that $g^{l(n)} = G(x)$.

Construct the class of hybrid distributions $\{H^i\}_{i=1}^{l(n)}$:

$$H^i = U_{l(n)-i} \circ g^i(U_n)$$

One can obeserve that $H^0 = G(U_n)$, and $H^{l(n)} = U_{l(n)}$.
Now we construct the distinguisher D as follows:

Begin Algorithm Distinguisher
Input α , $|\alpha| = n+1$ (taken from $G_1(U_n)$ on U_{n+1})
(1) Choose $i \in_R \{1 \,..\, l(n)\}$
(2) Choose $Z \sim U_{l(n)-i}$
(3) Construct $y = Z \circ \sigma \circ g^{(i-1)}(S)$, where σ is first bit of α and S its n-bit suffix
Return $A(y)$
end.

We denote by $\Pr[A|i]$ the *conitional probability* of event A if particular i was choosen in step (1) of algorithm D. We see that

$$\begin{aligned} &\Pr[\ D(G_1(U_n)) = 1\ ] - \Pr[\ D(U_{n+1}) = 1\ ] \\ &= \frac{1}{l(n)} \sum_{i=1}^{l(n)} (\ \Pr[\ D(G_1(U_n)) = 1\ |i] - \Pr[\ D(U_{n+1}) = 1\ |i]\ ). \qquad (**) \end{aligned}$$

Note that

$$\begin{aligned} \Pr[\ D(G_1(U_n)) = 1|i] &= \Pr\left[\ A(Z_{1..(l(n)-i)} \circ [G_1(U_n)]_1 \circ g^{(i-1)}([G_1(U_n)]_{(2..n+1)}) = 1\ \right] \\ &= \Pr\left[\ A(H^i) = 1\ \right] \end{aligned}$$

and

$$\begin{aligned} \Pr[\ D(U_{n+1}) = 1|i] &= \Pr\left[\ A(Z_{1..(l(n)-i)} \circ [U_{n+1}]_1 \circ g^{(i-1)}([U_{n+1}]_{2..n+1})) = 1\ \right] \\ &= \Pr\left[\ A(H^{i-1}) = 1\ \right] \end{aligned}$$

So equation $(**)$ is

$$\begin{aligned} &\frac{1}{l(n)} \sum_{i=1}^{l(n)} \left(\Pr\left[\ A(H^i) = 1\ \right] - \Pr\left[\ A(H^{i-1}) = 1\ \right]\right) \\ &= \frac{1}{l(n)} \left(\Pr\left[\ A(H^{l(n)}) = 1\ \right] - \Pr\left[\ A(H^0) = 1\ \right]\ \right) \\ &= \frac{1}{l(n)} \left(\Pr[\ A(G(U_n)) = 1\ ] - \Pr\left[\ A(U_{l(n)}) = 1\ \right]\ \right) \end{aligned}$$

so

$$\Pr[\ D(G_1(U_n)) = 1\ ] - \Pr[\ D(U_{n+1}) = 1\ ] \geq \frac{1}{l(n)p(n)}$$

■

13.4 On Using Pseudo-Random Generators

Suppose we have a probabilistic polynomial time algorithm A, which on input of length n uses $m(n)$, random bits. Algorithm A may solve either search problem for some relation or decision problem for some language L. Our claim will be that for all $\varepsilon > 0$ there exists a probabilistic polynomial time algorithm A' that uses only n^{ε} random bits and "behaves" in the same way that A does.

The construction of A' bases on assumption that we are given pseudo-random generator $G : \{0,1\}^{n^{\varepsilon}} \to \{0,1\}^{m(n)}$. Recall that $A(x,R)$ that A is running on input x with coins R.

Algorithm A'
Input $x \in \{0,1\}^n$
Choose $S \in_R \{0,1\}^{n^{\varepsilon}}$
$R \leftarrow G(s)$ (generate the coin tosses)
Return $A(x,R)$ (run A on input x using coins R)
end.

Proposition 13.4.1 (imformal): *It is infeasible given* 1^n *to find* $x \in \{0,1\}^n$*, such that the "behaviour" of* $A'(x)$ *is substantially diferent from* $A(x)$*.*

The meaning of this proposition depends on the computational problem solved by A. In case A solves some NP-search problem, the proposition asserts that it is hard (i.e., feasible only with negligable probability) to find large x's such that A can find the solution for x, and $A'(x)$ will fail to do so. In case A computes some function the proposition applies too.

> Oded's Note: *But the proposition may not hold if A solves a search problem in which instances have many possible solutions and are not efficiently verifiable (as in NP-search problems).*

Below we prove the proposition for the case of decision problems (and the proof actually extends to any function computation). We assume that A gives the correct answer with probability bounded away from $1/2$.

Proof: Suppose we have a finder F, which works in polynomial time , $F(1^n) = x \in \{0,1\}^n$, such that

$$\Pr\left[A'(x) = X_L(x)\right] \ \leq \ \frac{1}{2}$$

where X_L is the characteristic function of a language decideable by A (i.e., $\Pr[A(x) = X_L(x)] \geq 2/3$ for all x's). Construct a distinguisher D as follows:

Begin Algorithm D
Input $\alpha \in \{0,1\}^{m(n)}$
$x \leftarrow F(1^n)$
$v \leftarrow X_L(x)$ with overwhelmingly high probability (i.e., invoke $A(x)$ polynomially many times and take a majority vote).
$w \leftarrow A(x,\alpha)$
If $v = w$ **Then Return** 1
Else Return 0

end.

D is contradicts the pseudorandomness of G because

$$A(x,\alpha) = \begin{cases} X_L(x), \ w.p. \ \geq \frac{2}{3}, & \alpha \sim U_{m(n)} \\ X_L(x) = A'(x), \ w.p. \ \leq \frac{1}{2}, & \alpha \sim G(U_{n^\varepsilon}) \end{cases} \tag{13.1}$$

Furthermore, with probability at least 0.99, the value v found by $D(x)$ equals $X_L(x)$. Thus,

$$\begin{aligned} \Pr[D(U_{m(n)}) = 1] &> 0.66 - 0.01 = 0.65 \\ \Pr[D(G(U_{n^\epsilon})) = 1] &\leq 0.5 + 0.01 < 0.55 \end{aligned}$$

which provides the required gap. ■

> Oded's Note: *Note that for a strong notion of pseudorandom generators, where the output is indistinguishable from random by small circuits we can prove a stronger result; that is, that there are only finitely many x's on which A' behaves differently than A. Thus, in case of decision algorithms, by minor modification to A', we can make A' accept the same language as A.*

13.5 Relation to one-way functions

An important property of a pseudo-random generator $G(S)$ that it turns the seed into the sequence $x = G(S)$ in polynomial time. But the inverse operation of finding the seed S from $G(S)$ would be hard (or else pseudorandomness is violated as shown below). A pseudo-random generator is however, not just a function that hard to invert it also stretches the input into the larger sequence that look random. Still pseudo-random generators can be related to functions which are "only" easy to compute and hard to invert, as defined next.

Definition 13.11 (One-way functions – OWF): *A function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ such that $\forall x \ |f(x)| = |x|$ is* one-way *if :*

- *there is exists polynomial time algorithm A, such that $\forall x \ A(x) = f(x)$*
- *for all probabilistic polynomial time A' and for all polynome $p()$ and for all sufficiently large n's :*

$$\Pr\left[A'(f(U_n)) = f^{-1} \circ f(U_n)\right] < \frac{1}{p(n)}$$

In other words this function must be easy computed and hard inverted. Note an important feauture: the inversion algotihm must fail almost always. But the probability distribution used here is not uniform over all $f(x)$; rather, it is the distribution $f(x)$ when x is choosen uniformly.

> Oded's Note: *The requirement that the function be length preserving (i.e., $|f(x)| = |x|$ for all x's) may be relaxed as long as the length of $f(x)$ is polynomially related to $|x|$. In contrast a function like $f(x) \stackrel{\text{def}}{=} |x|$ would be "one-way" for a trivial and useless reason (on input n in binary one cannot print an n-bit string in polynomial (in $\log n$) time).*

Comment: A popular candidate to be one-way function is based on the conjectured intractability of the integer factorization problem. The length of input and output to the function will not be exactly n, only polynomial in n:
The factoring problem. Let $x, y > 1$ be n-bit integers. Define

$$f(x, y) = x * y$$

When x, y are n-bit primes, it is believed that finding x, y from $x * y$ is computationaly difficult.

> Oded's Note: *So the above should be hard to invert in these cases which occur at density $\approx 1/n^2$. This does not satisfy the definition of one-wayness which requires hardness of inversion almost everywhere, but suitable amplification can get us there. Alternatively, we can redefine the function f so that $f(x, y) = prime(x) * prime(y)$, where $prime()$ is a procedure which uses the input string to generate a large prime so that when the input is a random n-bit string the output is a random $n/O(1)$-bit prime. Such efficient procedures are known to exist. Using less sophisticated methods one can easily construct a procedure which uses n-bits to produce a prime of length $\sqrt[3]{n}/O(1)$.*

Theorem 13.12 *Pseudo-Random Generators exist if and only if One-Way Functions exist.*

So the computational hardness and pseudorandomness are strongly connect each other. If we have the created randomness we can create the hardness, and vice versa. Let us prove one part of the theorem and give hints to special case of other.

PRG $\Longrightarrow$ OWF: Consider pseudo-random generator $G : \{0,1\}^n \to \{0,1\}^{2n}$. Let us define function $f : \{0,1\}^{2n} \to \{0,1\}^{2n}$ as follows:

$$f(xy) = G(x) \qquad (|x| = |y| = n).$$

We claim, that f is one-way function, and the proof is by contradiction :

Suppose probabilistic polynomial time algorithm A' inverts f with success probability greater than $\frac{1}{p(n)}$, where $p(n)$ is polynom.

Consider a distinguisher D :
input: α, $\alpha \in \{0,1\}^{2n}$
$xy \leftarrow A'(\alpha)$
if $f(xy) = \alpha$ return 1
otherwise return 0.

$$\begin{aligned} \Pr[D(G(U_n)) = 1] &= \Pr[D(f(U_n)) = 1] \\ &= \Pr[f(A'(f(U_n))) = f(U_n)] \\ &= \Pr\left[A'(f(U_n)) \in f^{-1}f(U_n)\right] \\ &> \frac{1}{p(n)} \end{aligned}$$

where the last inequality is due to the contradiction hypothesis. On the other hand, there are at most 2^n strings of length $2n$ which have a preimage under G (and so under f). Thus, a uniformly

selected random string of length $2n$ has a preimage under f with probability at most $2^n/2^{2n}$. It follows that

$$\begin{aligned}\Pr[D(U_{2n})=1] &= \Pr[f(A'(U_{2n}))=U_{2n}] \\ &\leq \Pr[U_{2n} \text{ is in the image of } f] \\ &\leq \frac{2^n}{2^{2n}} = 2^{-n}\end{aligned}$$

Thus,

$$\Pr[D(G(U_n))=1] - \Pr[D(U_{2n})=1] > \frac{1}{p(n)} - \frac{1}{2^n} > \frac{1}{q(n)}$$

For some polynome $q()$

OWF $\Longrightarrow$ PRG:

Oded's Note: *The rest of this section is an overview of what is shown in detail in the next lecture (i.e., Lecture 14).*

Let us demonstrate the other direction and build an Pseudo-Random Generator if we have OWF of special form. Suppose the function $f : \{0,1\}^n \to \{0,1\}^n$ is not only OWF but it is also $1-1$. So it is a permutation of strings of length n. Assume that we can get a random bit b from the input, such that b will be hard to "predict" from the output of f. In this case we can construct a Pseudo-Random Generator as a concatenation of $f(x)$ and b.

Definition 13.13 (Hardcore):
Let f be one-way function, $b : \{0,1\}^* \longrightarrow \{0,1\}$ is a hardcore of f if:

- $\exists$ *polinomial time algorithm A, such that* $\forall t A(t) = b(t)$
- $\forall$ *probabilistic polynomial time algorithm A'* $\forall$ *polynom $p(.)$* $\forall$ *sufficiently large $n's$*

$$\Pr[A'(f(U_n)) = b(U_n)] < \frac{1}{2} + \frac{1}{p(n)}$$

In other words this function must be easy to compute and hard to predict out of $f(x)$.

The following theorem can be proven:

Theorem 13.14 *If f is OW, $f'(x,y) = f(x) \circ y$, $(|x| = |y|)$*
then $b(x,y) = \sum_{i=1}^{n} x_i y_i \pmod 2$ is a hardcore of f.

This theorem would be proven in next lecture. Now we can construct a Pseudo-Random Generator G as follows:

$$G(s) = f'(s) \circ b(s)$$

The two first properties of G (poly-time and stretching) are trivial. The pseudorandomness of G follows from the fact that its first n output bits are uniformly distributed and the last bit is unpredictable. Unpredictability translates to indistinguishability, as will be shown in the next lecture.

Bibliographic Notes

The notion of computational indistinguishability was introduced by Goldwasser and Micali [4] (within the context of defining secure encryptions), and given general formulation by Yao [6]. Our definition of pseudorandom generators follows the one of Yao, which is equivalent to a prior formulation of Blum and Micali [1]. For more details regarding this equivalence, as well as many other issues, see [2]. The latter source presents the notion of pseudorandomness discussed here as a special case (or archetypical case) of a general paradigm.

The discovery that computational hardness (in form of one-wayness) can be turned into a pseudorandomness was made in [1]. Theorem 13.12 (asserting that psedorandom generators can be constructed based on any one-way function) is due to [5]. It uses Theorem 13.14 which is due to [3].

1. M. Blum and S. Micali. How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. *SICOMP*, Vol. 13, pages 850–864, 1984. Preliminary version in *23rd FOCS*, 1982.

2. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. Algorithms and Combinatorics series (Vol. 17), *Springer*, 1998.

3. O. Goldreich and L.A. Levin. Hard-core Predicates for any One-Way Function. In *21st STOC*, pages 25–32, 1989.

4. S. Goldwasser and S. Micali. Probabilistic Encryption. *JCSS*, Vol. 28, No. 2, pages 270–299, 1984. Preliminary version in *14th STOC*, 1982.

5. J. Håstad, R. Impagliazzo, L.A. Levin and M. Luby. Construction of Pseudorandom Generator from any One-Way Function. To appear in *SICOMP*. Preliminary versions by Impagliazzo et. al. in *21st STOC* (1989) and Håstad in *22nd STOC* (1990).

6. A.C. Yao. Theory and Application of Trapdoor Functions. In *23rd FOCS*, pages 80–91, 1982.

Oded's Note: *Being in the process of writing an essay on pseudorandomness, it feels a good idea to augment the notes of the current lecture by a draft of this essay. The lecture notes actually expand on the presentation in Sections 13.6.2 and 13.6.3. The other sections in this essay go beyond the lecture notes.*

Appendix: An essay by O.G.

Summary: We postulate that a distribution is pseudorandom if it cannot be told apart from the uniform distribution by an efficient procedure. This yields a robust definition of pseudorandom generators as efficient deterministic programs stretching short random seeds into longer pseudorandom sequences. Thus, pseudorandom generators can be used to reduce the randomness-complexity in any efficient procedure. Pseudorandom generators and computational difficulty are strongly related: loosely speaking, each can be efficiently transformed into the other.

13.6.1 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., [4]), initiated by Shannon [21], is rooted in probability theory and is focused at distributions which are not perfectly random. Shannon's Information Theory characterizes perfect randomness as the extreme case in which the *information content* is maximized (and there is no redundancy at all). Thus, perfect randomness is associated with a unique distribution – the uniform one. In particular, by definition, one cannot generate such perfect random strings from shorter random seeds.

The second theory (cf., [16, 17]), due to Solomonov [22], Kolmogorov [15] and Chaitin [3], is rooted in computability theory and specifically in the notion of a universal language (equiv., universal machine or computing device). It measures the complexity of objects in terms of the shortest program (for a fixed universal machine) which generates the object. Like Shannon's theory, Kolmogorov Complexity is quantitative and perfect random objects appear as an extreme case. Interestingly, 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 generate strings of high Kolmogorov Complexity from short random seeds.

The third theory, initiated by Blum, Goldwasser, Micali and Yao [12, 1, 24], is rooted in complexity theory and is the focus of this essay. This approach is explicitly aimed at providing a notion of perfect randomness which nevertheless allows to efficiently generate perfect 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 which 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 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 all of them Alice flips a coin high in the air, 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.

13.6.2 The Definition of Pseudorandom Generators

Loosely speaking, a pseudorandom generator is an *efficient* program (or algorithm) which *stretches* short random seeds into long *pseudorandom* sequences. The above emphasizes three fundamental aspects in the notion of a pseudorandom generator:

1. *Efficiency*: The generator has to be efficient. We associate efficient computations with those conducted within time which is polynomial in the length of the input. Consequently, we postulate that the generator has to be implementable by a deterministic polynomial-time algorithm.

   This algorithm takes as input a *seed*, as mentioned above. The seed captures a bounded amount of randomness used by a device which "generates pseudorandom sequences." The formulation views any such device as consisting of a deterministic procedure applied to a random seed.

2. *Stretching*: The generator is required to stretch its input seed to a longer output sequence. Specifically, it stretches n-bit long seeds into $\ell(n)$-bit long outputs, where $\ell(n) > n$. The function ℓ is called the *stretching measure* (or *stretching function*) of the generator.

3. *Pseudorandomness*: The generator's output has to look random to any efficient observer. That is, any efficient procedure should fail to distinguish the output of a generator (on a random seed) from a truly random sequence of the same length. The formulation of the last sentence refers to a general notion of *computational indistinguishability* which is the heart of the entire approach.

Computational Indistinguishability: Intuitively, two objects are called computationally indistinguishable if no efficient procedure can tell them apart. As usual in complexity theory, an elegant formulation requires asymptotic analysis (or rather a functional treatment of the running time of algorithms in terms of the length of their input).[1] Thus, the objects in question are infinite

[1] We stress that the asymptotic (or functional) treatment is not essential to this approach. One may develop the entire approach in terms of inputs of fixed lengths and an adequate notion of complexity of algorithms. However, such an alternative treatment is more cumbersome.

sequences of distributions, where each distribution has a finite support. Such a sequence will be called a *distribution ensemble*. Typically, we consider distribution ensembles of the form $\{D_n\}_{n\in\mathsf{N}}$, where for some function $\ell : \mathsf{N}\mapsto\mathsf{N}$, the support of each D_n is a subset of $\{0,1\}^{\ell(n)}$. Furthermore, typically ℓ will be a positive polynomial.

> Oded's Note: *In this essay, I've preferred the traditional mathematical notations. Specifically, I have used distributions (over strings) rather than our non-standard "random variables" (which range over strings). For a distribution D, the traditional notation $x\sim D$ means x selected according to distribution D.*

Definition 13.6.1 (Computational Indistinguishability [12, 24]): *Two probability ensembles, $\{X_n\}_{n\in\mathsf{N}}$ and $\{Y_n\}_{n\in\mathsf{N}}$, are called* computationally indistinguishable *if for any probabilistic polynomial-time algorithm A, for any positive polynomial p, and for all sufficiently large n's*

$$|\Pr_{x\sim X_n}[A(x)=1] - \Pr_{y\sim Y_n}[A(y)=1]| < \frac{1}{p(n)}$$

The probability is taken over X_n (resp., Y_n) as well as over the coin tosses of algorithm A.

A couple of comments are in place. Firstly, we have allowed algorithm A (called a distinguisher) to be probabilistic. This makes the requirement only stronger, and seems essential to several important aspects of our approach. Secondly, we view events occuring with probability which is upper bounded by the reciprocal of polynomials as *negligible*. This is well-coupled with our notion of efficiency (i.e., polynomial-time computations): An event which occurs with negligible probability (as a function of a parameter n), will occur with negligible probability also if the experiment is repeated for poly(n)-many times.

We note that computational indistinguishability is a strictly more liberal notion than statistical indistinguishability (cf., [24, 10]). An important case is the one of distributions generated by a pseudorandom generator as defined next.

Definition 13.6.2 (Pseudorandom Generators [1, 24]): *A deterministic polynomial-time algorithm G is called a* pseudorandom generator *if there exists a* stretching function, *$\ell : \mathsf{N}\mapsto\mathsf{N}$, so that the following two probability ensembles, denoted $\{G_n\}_{n\in\mathsf{N}}$ and $\{R_n\}_{n\in\mathsf{N}}$, are computationally indistinguishable*

1. *Distribution G_n is defined as the output of G on a uniformly selected seed in $\{0,1\}^n$.*
2. *Distribution R_n is defined as the uniform distribution on $\{0,1\}^{\ell(n)}$.*

That is, letting U_m denote the uniform distribution over $\{0,1\}^m$, we require that for any probabilistic polynomial-time algorithm A, for any positive polynomial p, and for all sufficiently large n's

$$|\Pr_{s\sim U_n}[A(G(s))=1] - \Pr_{r\sim U_{\ell(n)}}[A(r)=1]| < \frac{1}{p(n)}$$

Thus, pseudorandom generators are efficient (i.e., polynomial-time) deterministic programs which expand short randomly selected seeds into longer pseudorandom bit sequences, where the latter are defined as computationally indistinguishable from truly random sequences by efficient (i.e., polynomial-time) algorithms. It follows that any efficient randomized algorithm maintains its performance when its internal coin tosses are substituted by a sequence generated by a pseudorandom generator. That is,

Construction 13.6.3 (typical application of pseudorandom generators): *Let A be a probabilistic algorithm, and $\rho(n)$ denote a* (polynomial) *upper bound on its randomness complexity. Let $A(x,r)$ denote the output of A on input x and coin tosses sequence $r \in \{0,1\}^{\rho(|x|)}$. Let G be a pseudorandom generator with stretching function $\ell : \mathsf{N} \mapsto \mathsf{N}$. Then A_G is a randomized algorithm which on input x, proceeds as follows. It sets $k = k(|x|)$ to be the smallest integer such that $\ell(k) \geq \rho(|x|)$, uniformly selects $s \in \{0,1\}^k$, and outputs $A(x,r)$, where r is the $\rho(|x|)$-bit long prefix of $G(s)$.*

It can be shown that it is infeasible to find long x's on which the *input-output behavior* of A_G is *noticeablly different* from the one of A, although A_G may use much fewer coin tosses than A. That is

Theorem 13.6.4 *Let A and G be as above. 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*

$$\sum_{x \in \{0,1\}^n} \Pr[F(1^n) = x] \cdot \Delta_{A,D}(x) \;<\; \frac{1}{p(n)}$$

$$\text{where } \Delta_{A,D}(x) \stackrel{\text{def}}{=} |\Pr_{r \sim U_{\rho(n)}}[D(x, A(x,r)) = 1] - \Pr_{s \sim U_{k(n)}}[D(x, A_G(x,s)) = 1]|$$

and the probabilities are taken over the U_m's as well as over the coin tosses of F and D.

The theorem is proven by showing that a triplet (A, F, D) violating the claim can be converted into an algorithm D' which distinguishes the output of G from the uniform distribution, in contradiction to the hypothesis. 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 pseudorandom generators with large stretching function, *one can considerably reduce the randomness complexity in any efficient application.*

Amplifying the stretch function. Pseudorandom generators as defined above are only required to stretch their input a bit; for example, stretching n-bit long inputs to $(n+1)$-bit long outputs will do. Clearly generator of such moderate stretch function are of little use in practice. In contrast, we want to have pseudorandom generators with an arbitrary long stretch function. By the efficiency requirement, the stretch function can be at most polynomial. It turns out that pseudorandom generators with the smallest possible stretch function can be used to construct pseudorandom generators with any desirable polynomial stretch function. (Thus, when talking about the existence of pseudorandom generators, we may ignore the stretch function.)

Theorem 13.6.5 [9]: *Let G be a pseudorandom generator with stretch function $\ell(n) = n+1$, and ℓ' be any polynomially bounded stretch function, which is polynomial-time computable. Let $G_1(x)$ denote the $|x|$-bit long prefix of $G(x)$, and $G_2(x)$ denote the last bit of $G(x)$ (i.e., $G(x) = G_1(x)\,G_2(x)$). Then*

$$G'(s) \stackrel{\text{def}}{=} \sigma_1\sigma_2\cdots\sigma_{\ell'(|s|)}\,,$$
$$\text{where } x_0 = s,\ \sigma_i = G_2(x_{i-1}) \text{ and } x_i = G_1(x_{i-1}), \text{ for } i = 1, ..., \ell'(|s|)$$

is a pseudorandom generator with stretch function ℓ'.

Proof Sketch: The theorem is proven using the *hybrid technique* (cf., Sec. 3.2.3 in [5]): One considers distributions H_n^i (for $i = 0, ..., \ell(n)$) defined by $U_i^{(1)} P_{\ell(n)-i}(U_n^{(2)})$, where $U_i^{(1)}$ and $U_n^{(2)}$ are independent uniform distributions (over $\{0,1\}^i$ and $\{0,1\}^n$, respectively), and $P_j(x)$ denotes the j-bit long prefix of $G'(x)$. The extreme hybrids correspond to $G'(U_n)$ and $U_{\ell(n)}$, whereas distinguishability of neighboring hybrids can be worked into distinguishability of $G(U_n)$ and U_{n+1}. Loosely speaking, suppose one could distinguish H_n^i from H_n^{i+1}. Then, using $P_j(s) = G_2(s)P_{j-1}(G_1(s))$ (for $j \geq 1$), this means that one can distinguish $H_n^i \equiv (U_i^{(1)}, G_2(U_n^{(2)}), P_{(\ell(n)-i)-1}(G_1(U_n^{(2)})))$ from $H_n^{i+1} \equiv (U_i^{(1)}, U_1^{(1')}, P_{\ell(n)-(i+1)}(U_n^{(2')}))$. Incorporating the generation of $U_i^{(1)}$ and the evaluation of $P_{\ell(n)-i-1}$ into the distinguisher, one could distinguish $(f(U_n^{(2)}), b(U_n^{(2)})) \equiv G_1(U_n)$ from $(U_n^{(2')}, U_1^{(1')}) \equiv U_{n+1}$, in contradiction to the pseudorandomness of G_1. (For details see Sec. 3.3.3 in [5].) ■

13.6.3 How to Construct Pseudorandom Generators

The known constructions transform computation difficulty, in the form of one-way functions (defined below), into pseudorandomness generators. Loosely speaking, a *polynomial-time computable* function is called one-way if any efficient algorithm can invert it only with negligible success probability. For simplicity, we consider only length-preserving one-way functions.

Definition 13.6.6 (one-way function): *A* one-way function, *f, is a polynomial-time computable function such that for every probabilistic polynomial-time algorithm A', every positive polynomial $p(\cdot)$, and all sufficiently large n's*

$$\Pr_{x \sim U_n}\left[A'(f(x)) \in f^{-1}(f(x))\right] < \frac{1}{p(n)}$$

where U_n is the uniform distribution over $\{0,1\}^n$.

Popular candidates for one-way functions are based on the conjectured intractability of Integer Factorization (cf., [18] for state of the art), the Discrete Logarithm Problem (cf., [19] analogously), and decoding of random linear code [11]. The infeasibility of inverting f yields a weak notion of unpredictability: Let $b_i(x)$ denotes the i^{th} bit of x. Then, for every probabilistic polynomial-time algorithm A (and sufficiently large n), it must be the case that $\Pr_{i,x}[A(i, f(x)) \neq b_i(x)] > 1/2n$, where the probability is taken uniformly over $i \in \{1, ..., n\}$ and $x \in \{0,1\}^n$. A stronger (and in fact strongest possible) notion of unpredictability is that of a hard-core predicate. Loosely speaking, a *polynomial-time computable* predicate b is called a hard-core of a function f if all efficient algorithm, given $f(x)$, can guess $b(x)$ only with success probability which is negligible better than half.

Definition 13.6.7 (hard-core predicate [1]): *A polynomial-time computable predicate $b : \{0,1\}^* \mapsto \{0,1\}$ 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*

$$\Pr_{x \sim U_n}(A'(f(x)) = b(x)) < \frac{1}{2} + \frac{1}{p(n)}$$

Clearly, if b is a hard-core of a 1-1 polynomial-time computable function f then f must be one-way.[2] It turns out that any one-way function can be slightly modified so that it has a hard-core predicate.

[2] Functions which are not 1-1 may have hard-core predicates of information theoretic nature; but these are of no use to us here. For example, for $\sigma \in \{0,1\}$, $f(\sigma, x) = 0f'(x)$ has an "information theoretic" hard-core predicate $b(\sigma, x) = \sigma$.

Theorem 13.6.8 (A generic hard-core [8]): *Let f be an arbitrary one-way function, and let g be defined by $g(x,r) \stackrel{\text{def}}{=} (f(x),r)$, where $|x| = |r|$. Let $b(x,r)$ denote the inner-product mod 2 of the binary vectors x and r. Then the predicate b is a hard-core of the function g.*

See proof in Apdx C.2 in [6]. Finally, we get to the construction of pseudorandom generators:

Theorem 13.6.9 (A simple construction of pseudorandom generators): *Let b be a hard-core predicate of a polynomial-time computable 1-1 function f. Then, $G(s) \stackrel{\text{def}}{=} f(s)\,b(s)$ is a pseudorandom generator.*

Proof Sketch: Clearly the $|s|$-bit long prefix of $G(s)$ is uniformly distributed (since f is 1-1 and onto $\{0,1\}^{|s|}$). Hence, the proof boils down to showing that distinguishing $f(s)b(s)$ from $f(s)\sigma$, 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 $f(s)\overline{b(s)}$, where $\overline{\sigma} = 1 - \sigma$, and so yields an algorithm for predicting $b(s)$ based on $f(s)$. ■

In a sense, the key point in the proof of the above theorem is showing that the (obvious by definition) 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 presentation below.

An alternative presentation. Our presentation of the construction of pseudorandom generators, via Construction 13.6.5 and Proposition 13.6.9, is analogous to the original construction of pseudorandom generators suggested by by Blum and Micali [1]: Given an arbitrary stretch function $\ell : \mathsf{N} \mapsto \mathsf{N}$, a 1-1 one-way function f with a hard-core b, one defines

$$G(s) \stackrel{\text{def}}{=} b(x_1)b(x_2)\cdots b(x_{\ell(|s|)})\,,$$

where $x_0 = s$ and $x_i = f(x_{i-1})$ for $i = 1, ..., \ell(|s|)$. The pseudorandomness of G is established in two steps, using the notion of (next bit) unpredictability. An ensemble $\{Z_n\}_{n\in\mathsf{N}}$ is called unpredictable if any probabilistic polynomial-time machine obtaining a prefix of Z_n fails to predict the next bit of Z_n with probability non-negligiblly higher than $1/2$.

Step 1: One first proves that the ensemble $\{G(U_n)\}_{n\in\mathsf{N}}$, where U_n is uniform over $\{0,1\}^n$, is (next-bit) unpredictable (from right to left) [1].

Loosely speaking, if one can predict $b(x_i)$ from $b(x_{i+1})\cdots b(x_{\ell(|s|)})$ then one can predict $b(x_i)$ given $f(x_i)$ (i.e., by computing $x_{i+1}, ..., x_{\ell(|s|)}$ and so obtaining $b(x_{i+1})\cdots b(x_{\ell(|s|)})$), in contradiction to the hard-core hypothesis.

Step 2: Next, one uses Yao's observation by which a (polynomial-time constructible) ensemble is *pseudorandom if and only if it is* (next-bit) *unpredictable* (cf., Sec. 3.3.4 in [5]).

Clearly, if one can predict the next bit in an ensemble then one can distinguish this ensemble from the uniform ensemble (which in unpredictable regardless of computing power). However, here we need the other direction which is less obvious. Still, one can show that (next bit) unpredictability implies indistinguishability from the uniform ensemble. Specifically, consider the following "hybrid" distributions, where the i^{th} hybrid takes the first i bits from the questionable ensemble and the rest from the uniform one. Thus, distinguishing the extreme hybrids implies distinguishing some neighboring hybrids, which in turn implies next-bit predictability (of the questionable ensemble).

A general condition for the existence of pseudorandom generators. Recall that given any one-way 1-1 function, we can easily construct a pseudorandom generator. Actually, the 1-1 requirement may be dropped, but the currently known construction – for the general case – is quite complex. Still we do have.

Theorem 13.6.10 (On the existence of pseudorandom generators [13]): *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 $\ell(n) = 2n$. For $x, y \in \{0,1\}^n$, 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 $G(U_n)$ from U_{2n} by trying to invert and checking the result: Inverting f on its range distribution refers to the distribution $G(U_n)$, whereas the probability that U_{2n} has inverse under f is negligible.

The interesting direction is the construction of pseudorandom generators based on any one-way function. In general (when f may not be 1-1) the ensemble $f(U_n)$ may not be pseudorandom, and so Construction 13.6.9 (i.e., $G(s) = f(s)b(s)$, where b is a hard-core of f) cannot be used *directly*. One idea of [13] is to hash $f(U_n)$ to an almost uniform string of length related to its entropy, using Universal Hash Functions [2]. (This is done after guaranteeing, that the logarithm of the probability mass of a value of $f(U_n)$ is typically close to the entropy of $f(U_n)$.)[3] But "hashing $f(U_n)$ down to length comparable to the entropy" means shrinking the length of the output to, say, $n' < n$. This foils the entire point of stretching the n-bit seed. Thus, a second idea of [13] is to compensate for the $n - n'$ loss by extracting these many bits from the seed U_n itself. This is done by hashing U_n, and the point is that the $(n - n' + 1)$-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.

13.6.4 Pseudorandom Functions

Pseudorandom generators allow to efficiently generate long pseudorandom sequences from short random seeds. Pseudorandom functions (defined below) are even more powerful: They allow efficient direct access to a huge pseudorandom sequence (which is infeasible 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). That is, pseudorandom functions are indistinguishable from random functions by efficient machines which may obtain the function values at arguments of their choice. (Such machines are called oracle machines, and if M is such machine and f is a function, then $M^f(x)$ denotes the computation of M on input x when M's queries are answered by the function f.)

Definition 13.6.11 (pseudorandom functions [7]): *A* pseudorandom function (ensemble)*, with length parameters $\ell_D, \ell_R : \mathsf{N} \mapsto \mathsf{N}$, is a collection of functions $F \stackrel{\text{def}}{=} \{f_s : \{0,1\}^{\ell_D(|s|)} \mapsto \{0,1\}^{\ell_R(|s|)}\}_{s \in \{0,1\}^*}$ satisfying*

- (efficient evaluation): *There exists an efficient* (deterministic) *algorithm which given a* seed*, s, and an $\ell_D(|s|)$-bit* argument*, x, returns the $\ell_R(|s|)$-bit long value $f_s(x)$.*

[3] 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 $x_1, ..., x_{n^2} \in \{0,1\}^n$, we let $f(x_1, ..., x_{n^2}) \stackrel{\text{def}}{=} f'(x_1), ..., f'(x_{n^2})$.

- (pseudorandomness): *For every probabilistic polynomial-time oracle machine, M, for every positive polynomial p and all sufficiently large n's*

$$\left| \Pr_{f\sim F_n}[M^f(1^n)=1] - \Pr_{\rho\sim R_n}[M^\rho(1^n)=1] \right| < \frac{1}{p(n)}$$

where F_n denotes the distribution on F obtained by selecting s uniformly in $\{0,1\}^n$, and R_n denotes the uniform distribution over all functions mapping $\{0,1\}^{\ell_{\rm D}(n)}$ to $\{0,1\}^{\ell_{\rm R}(n)}$.

Suppose, for simplicity, that $\ell_{\rm D}(n)=n$ and $\ell_{\rm R}(n)=1$. Then a function uniformly selected among 2^n functions (of a pseudorandom ensemble) presents an input-output behavior which is indistinguishable in poly(n)-time from the one of a function selected at random among all the 2^{2^n} Boolean functions. Contrast this with the 2^n pseudorandom sequences, produced by a pseudorandom generator, which are computationally indistinguishable from a sequence selected uniformly among all the $2^{{\rm poly}(n)}$ many sequences. Still pseudorandom functions can be constructed from any pseudorandom generator.

Theorem 13.6.12 (How to construct pseudorandom functions [7]): *Let G be a pseudorandom generator with stretching function $\ell(n)=2n$. Let $G_0(s)$ (resp., $G_1(s)$) denote the first (resp., last) $|s|$ bits in $G(s)$, and*

$$G_{\sigma_{|s|}\cdots\sigma_2\sigma_1}(s) \stackrel{\rm def}{=} G_{\sigma_{|s|}}(\cdots G_{\sigma_2}(G_{\sigma_1}(s))\cdots)$$

Then, the function ensemble $\{f_s:\{0,1\}^{|s|}\mapsto\{0,1\}^{|s|}\}_{s\in\{0,1\}^*}$, where $f_s(x)\stackrel{\rm def}{=}G_x(s)$, is pseudorandom with length parameters $\ell_{\rm D}(n)=\ell_{\rm R}(n)=n$.

The above construction can be easily adapted to any (polynomially-bounded) length parameters $\ell_{\rm D},\ell_{\rm R}:\mathsf{N}\mapsto\mathsf{N}$.

Proof Sketch: The proof uses the hybrid technique: The $i^{\rm th}$ hybrid, H_n^i, is a function ensemble consisting of $2^{2^i\cdot n}$ functions $\{0,1\}^n\mapsto\{0,1\}^n$, each defined by 2^i random n-bit strings, denoted $\langle s_\alpha\rangle_{\alpha\in\{0,1\}^i}$. The value of such function at $x=\beta\alpha$, with $|\alpha|=i$, is $G_\beta(s_\alpha)$. The extreme hybrids correspond to our indistinguishability claim (i.e., $H_n^0\equiv f_{U_n}$ and $H_n^n\equiv F_n$), and neighboring hybrids correspond to our indistinguishability hypothesis (specifically, to the indistinguishability of $G(U_n)$ and U_{2n} under multiple samples). ■

We mention that pseudorandom functions have been used to derive negative results in computational learning theory [23] and in complexity theory (cf., Natural Proofs [20]).

13.6.5 The Applicability of 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 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, pseudorandom generators (as defined above) are a key ingredient in an "algorithmic tool-box" – they provide an automatic compiler of programs written with free usage of randomness into programs which make an economical use of randomness.

Indeed, "pseudo-random number generators" have appeared with the first computers. However, typical implementations use generators which are not pseudorandom according to the above definition. Instead, at best, these generators are shown to pass SOME ad-hoc statistical test (cf., [14]).

However, the fact that a "pseudo-random number generator" passes some statistical tests, does not mean that it will pass a new test and that it is 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 type stated above (i.e., 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 13.6.2 aims at such generality, and in fact is tailored to obtain it: The notion of computational indistinguishability, which underlines Definition 13.6.2, covers all possible efficient applications postulating that for all of them pseudorandom sequences are as good as truly random ones.

13.6.6 The Intelectual Contents of Pseudorandom Generators

We shortly discuss some intelectual aspects of pseudorandom generators as defined above.

Behavioristic versus Ontological. Our definition of pseudorandom generators is based on the notion of computational indistinguishability. The behavioristic nature of the latter notion is best demonstrated by confronting it 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 which 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 an observer), as underlying the definition of pseudorandomness, is a behavioristic approach. Furthermore, there exist probability distributions which are not uniform (and are not even statistically close to a uniform distribution) that nevertheless are indistinguishable from a uniform distribution by any efficient method [24, 10]. Thus, distributions which are ontologically very different, are considered equivalent by the behavioristic point of view taken in the definitions above.

A relativistic view of randomness. Pseudorandomness is defined above in terms of its observer. It is a distribution which cannot be told apart from a uniform distribution by any efficient (i.e. polynomial-time) observer. However, pseudorandom sequences may be distinguished from random ones by infinitely powerful powerful (not at our disposal!). Specifically, an exponential-time machine can easily distinguish the output of a pseudorandom generator from a uniformly selected string of the same length (e.g., just by trying all possible seeds). Thus, pseudorandomness is subjective to the abilities of the observer.

Randomness and Computational Difficulty. Pseudorandomness and computational difficulty play dual roles: The definition of pseudorandomness relies on the fact that putting computational restrictions on the observer gives rise to distributions which are not uniform and still cannot be distinguished from uniform. Furthermore, the construction of pseudorandom generators rely on conjectures regarding computational difficulty (i.e., the existence of one-way functions), and this is inevitable: given a pseudorandom generator, we can construct one-way functions. Thus, (non-trivial) pseudorandomness and computational hardness can be converted back and forth.

13.6.7 A General Paradigm

Pseudorandomness as surveyed above can be viewed as an important special case of a general paradigm. A general treatment is provided in [6].

A generic formulation of pseudorandom generators consists of specifying three fundamental aspects – the *stretching 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*). In the above presentation we focused on polynomial-time generators (thus having polynomial stretching measure) which fool any probabilistic polynomial-time observers. A variety of other cases are of interest too, and we briefly discuss some of them.

Weaker notions of computational indistinguishability. Whenever the aim is to replace random sequences utilized by an algorithm with pseudorandom ones, one may try to capitalize on knowledge of the target algorithm. Above we have merely used the fact that the target algorithm runs in polynomial-time. However, for example, if we know that the algorithm uses very little workspace then we may able to do better. Similarly, if we know that the analysis of the algorithm only depends on some specific properties of the random sequence it uses (e.g., pairwise independence of its elements). In general, weaker notions of computational indistinguishability such as fooling space-bounded algorithms, constant-depth circuits, and even specific tests (e.g., testing pairwise independence of the sequence), arise naturally: Generators producing sequences which fool such tests are useful in a variety of applications – if the application utilizes randomness in a restricted way then feeding it with sequences of low randomness-quality may do. Needless to say that we advocate a rigorous formulation of the characteristics of such applications and rigorous construction of generators which fool the type of tests which emerge.

Alternative notions of generator efficiency. The above discussion has focused on one aspect of the pseudorandomness question – the resources or type of the observer (or potential distinguisher). Another important question is whether such pseudorandom sequences can be generated from much shorter ones, and at what cost (or complexity). Throughout this essay we've required the generation process to be at least as efficient as the efficiency limitations of the distinguisher.[4] This seems indeed "fair" and natural. Allowing the generator to be more complex (i.e., use more time or space resources) than the distinguisher seems unfair, but still yields interesting consequences in the context of trying to "de-randomize" randomized complexity classe. For example, one may consider generators working in time exponential in the length of the seed. In some cases we loose nothing by being more liberal (i.e., allowing exponential-time generators). To see why, we consider a typical derandomization argument, proceediing in two steps: First one replaces the true randomness of the algorithm by pseudorandom sequences generated from much shorter seeds, and next one goes deterministically over all possible seeds and looks for the most frequent behavior of the modified algorithm. Thus, in such a case the deterministic complexity is anyhow exponential in the seed length. However, constructing exponential time generators may be easier than constructing polynomial-time ones.

References

[4] If fact, we have require the generator to be more efficient than the distinguisher: The former was required to be a fixed polynomial-time algorithm, whereas the latter was allowed to be any algorithm with polynomial running time.

1. 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 IEEE Symposium on Foundations of Computer Science*, 1982.

2. L. Carter and M. Wegman. Universal Hash Functions. *Journal of Computer and System Science*, Vol. 18, 1979, pages 143–154.

3. G.J. Chaitin. On the Length of Programs for Computing Finite Binary Sequences. *Journal of the ACM*, Vol. 13, pages 547–570, 1966.

4. T.M. Cover and G.A. Thomas. *Elements of Information Theory*. John Wiley & Sons, Inc., New-York, 1991.

5. O. Goldreich. *Foundation of Cryptography – Fragments of a Book*. February 1995. Available from `http://theory.lcs.mit.edu/~oded/frag.html`.

6. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness.* Algorithms and Combinatorics series (Vol. 17), *Springer*, 1998.

7. O. Goldreich, S. Goldwasser, and S. Micali. How to Construct Random Functions. *Journal of the ACM*, Vol. 33, No. 4, pages 792–807, 1986.

8. 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.

9. O. Goldreich and S. Micali. Increasing the Expansion of Pseudorandom Generators. Manuscript, 1984. Available from `http://theory.lcs.mit.edu/~oded/papers.html`

10. O. Goldreich, and H. Krawczyk, On Sparse Pseudorandom Ensembles. *Random Structures and Algorithms*, Vol. 3, No. 2, (1992), pages 163–174.

11. O. Goldreich, H. Krawcyzk and M. Luby. On the Existence of Pseudorandom Generators. *SIAM Journal on Computing*, Vol. 22-6, pages 1163–1175, 1993.

12. 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 ACM Symposium on the Theory of Computing*, 1982.

13. J. Håstad, R. Impagliazzo, L.A. Levin and M. Luby. Construction of Pseudorandom Generator from any One-Way Function. To appear in *SIAM Journal on Computing*. Preliminary versions by Impagliazzo et. al. in *21st ACM Symposium on the Theory of Computing* (1989) and Håstad in *22nd ACM Symposium on the Theory of Computing* (1990).

14. D.E. Knuth. *The Art of Computer Programming*, Vol. 2 (*Seminumerical Algorithms*). Addison-Wesley Publishing Company, Inc., 1969 (first edition) and 1981 (second edition).

15. A. Kolmogorov. Three Approaches to the Concept of "The Amount Of Information". *Probl. of Inform. Transm.*, Vol. 1/1, 1965.

16. L.A. Levin. Randomness Conservation Inequalities: Information and Independence in Mathematical Theories. *Inform. and Control*, Vol. 61, pages 15–37, 1984.

17. M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and its Applications.* Springer Verlag, August 1993.

18. A.M. Odlyzko. The future of integer factorization. *CryptoBytes* (The technical newsletter of RSA Laboratories), Vol. 1 (No. 2), pages 5-12, 1995.

19. A.M. Odlyzko. Discrete logarithms and smooth polynomials. In *Finite Fields: Theory, Applications and Algorithms*, G. L. Mullen and P. Shiue, eds., Amer. Math. Soc., Contemporary Math. Vol. 168, pages 269–278, 1994.

20. A.R. Razborov and S. Rudich. Natural proofs. *Journal of Computer and System Science*, Vol. 55 (1), pages 24–35, 1997.

21. C.E. Shannon. A mathematical theory of communication. *Bell Sys. Tech. Jour.*, Vol. 27, pages 623–656, 1948.

22. R.J. Solomonoff. A Formal Theory of Inductive Inference. *Inform. and Control*, Vol. 7/1, pages 1–22, 1964.

23. L. Valiant. A theory of the learnable. *Communications of the ACM*, Vol. 27/11, pages 1134–1142, 1984.

24. A.C. Yao. Theory and Application of Trapdoor Functions. In *23rd IEEE Symposium on Foundations of Computer Science*, pages 80–91, 1982.