On Teaching the Basics of Complexity Theory

(In Memory of Shimon Even [1935–2004])*

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

August 14, 2005

Abstract

We outline a conceptual framework for teaching the basic notions and results of complexity theory. Our focus is on using definitions and on organizing the presentation in a way that reflects the fundamental nature of the material. We do not attempt to provide a self-contained presentation of the material itself, but rather outline our suggestions regarding how this material should be presented in class. In addition, we express our opinions on numerous related issues.

We focus on the P-vs-NP Question, the general notion of a reduction, and the theory of NP-completeness. In particular, we suggest presenting the P-vs-NP Question both in terms of search problems and in terms of decision problems (where NP is viewed as a class of proof systems). As for the theory of NP-completeness, we suggest highlighting the mere existence of NP-complete sets.

*This essay was written towards publication in a forthcoming book in memory of Shimon Even.

Contents

1 Introduction

This is a highly opinionated essay that advocates a concept-oriented approach towards teaching technical material such as the basics of complexity theory. In addition to making various suggestions, I express my opinion on a variety of related issues. I do hope that this essay will stir discussion and maybe even affect the way some courses are being taught.

1.1 Teaching and current student perception of Complexity Theory

Shimon Even had a passion for good teaching, and so writing this essay in his memory seems most appropriate. In my opinion, good teaching is an art (and, needless to say, Shimon was one of its top masters). It is hard (if at all possible) to cultivate artistic talents, but there are certain basic principles that underly each art form, and these can be discussed.

One central aspect of good teaching is putting things in the right perspective; that is, a perspective that clarifies the motivation for the various definitions and results. Nothing should be easier when it comes to complexity theory: It is easy to provide a good perspective on the basic notions and results of complexity theory, because these are of fundamental nature and of great intuitive appeal. Unfortunately, often this is not the way this material is taught. The annoying (and quite amazing) consequences are students that have only a vague understanding of the *conceptual meaning* of these fundamental notions and results.

1.2 The source of trouble and eliminating it

In my opinion, it all boils down to taking the time to explicitly discuss the conceptual meaning of definitions and results. After all, the most important aspects of a scientific discovery are the intuitive question that it addresses, the reason that it addresses this question, the way it phrases the question, the approach that underlies its answer, and the ideas that are embedded in the answer. All these have to be reflected in the way the discovery is presented. In particular, one should use the "right" definitions (i.e., those that reflect better the fundamental nature of the notion being defined), and proceed in the (conceptually) "right" order. Two concrete examples follow.

Typically[1], NP is defined as the class of languages recognized by nondeterministic polynomial-time machines. Even bright students may have a hard time figuring out (by themselves) why one should care about such a class. On the other hand, when defining NP as the class of assertions that have easily verifiable proofs, each student is likely to understand its fundamental nature. Furthermore, the message becomes even more clear when discussing the search version analogue.

Similarly, one typically[1] takes the students throughout the detailed proof of Cook's Theorem before communicating to them the striking message (i.e., that "universal" problems exist at all, let alone that many natural problems like SAT are universal). Furthermore, in some cases, this message is not communicated explicitly at all.

1.3 Concrete suggestions

The rest of this essay provides concrete suggestions for teaching the basics of complexity theory, where by the basics I mean the P-vs-NP Question and the theory of NP-completeness. This material is typically taught as part of an undergraduate course on computability and complexity theory, and

[1]However, exceptions do exists: There are teachers and even textbooks that deviate from the standard practice being bashed here.

my suggestions are targeted primarily at computer scientists teaching such a course. However, I believe that my suggestions are valid regardless of the context in which this material is being taught.

I assume that the basic material itself is well-known to the reader. Thus, *my focus is not on the material itself, but rather on how it should be presented in class.* The two most important suggestions were already mentioned above:

1. The teacher should communicate the fundamental nature of the P-vs-NP Question while referring to definitions that (clearly) reflect this nature. In particular, I suggest explicitly presenting the implication of the P-vs-NP Question on the complexity of search problems, in addition to presenting the implication to decision problems.

2. The teacher should communicate the striking significance of the mere existence of NP-complete problems (let alone natural ones), before exhausting the students with complicated reductions.

Additional suggestions include providing a general perspective on the concept of a reduction, establishing tight relations between the complexity of search and decision problems, decoupling the proof of NP-hardness of SAT by using Circuit-SAT as an intermediate problem, and mentioning some additional topics (e.g., NP-sets that are neither in P nor NP-complete) rather than a host of NP-completeness results.

I advocate a model-independent presentation of the questions and results of complexity theory. I claim that most questions and results in complexity theory (like all results of computability theory) hold for any reasonable model of computation and can be presented with minimal reference to the specifics of the model.[2] In fact, in most cases, the specific model of computation is irrelevant. Typically, the presentation needs to refer to the specifics of the model of computation only when encoding the relation between consecutive instantaneous configurations of computation (see Section 4.3). However, such an encoding is possible for any reasonable model of computation, and this fact should be stressed.

It is also important to start a course (or series of lectures) by providing a wide perspective on its subject matter, which in this case is complexity theory. I would say that complexity theory is a central field of (Theoretical) Computer Science, concerned with the study of the *intrinsic* complexity of computational tasks, where this study tend to aim at *generality*: The field focuses on natural computational resources (most notably time), and the effect of limiting these resources on the *class of problems* that can be solved. Put in other words, complexity theory aims at understanding the *nature of efficient computation*. I suggest re-iterating the wider goals of complexity theory at the end of the course (or series of lectures), and illustrating them at that point by sketching a few of the active research directions and the results obtained in them. My own suggestion for such a brief overview is presented in Section 6.

Finally, until we reach the day in which every student can be assumed to have understood the meaning of the P-vs-NP Question and of NP-completeness, I suggest not to assume such an understanding when teaching an advanced complexity theory course. Instead, I suggest starting such a course with a fast discussion of the P-vs-NP Question and NP-completeness, making sure

[2]The specifics of the (reasonable) model are irrelevant for all questions and results mentioned in this essay, except for Theorem 6 where the model is important only for the exact bound on the slow-down of the optimal algorithm. Similarly, the specifics of the model effect the exact quantitative form of hierarchy theorems, but not their mere existence. Finally, in contrary to some beliefs, the specifics of the model are irrelevant also for most results regarding space complexity, provided that reasonable accounting of work-space is applied.

that the students understand the conceptual meaning of these basics.[3] (Needless to say, the rest of the course should also clarify the conceptual meaning of the material being taught.)

1.4 A parenthetical comment on computability versus complexity

This essay refers to the current situation in many schools, where the basics of complexity theory are taught within a course in which material entitled "computability" plays at least an equal role. The essay is confined to the "complexity" part of such a course, and takes the "computability" part for granted.

Let me seize the opportunity and express my opinion on this combined course on computability and complexity theory. In my opinion, complexity theory should play the main role in this course, whereas the basic concepts and results of computability theory should be regarded as an important preliminary material. That is, I view computability theory as setting the stage for the study of the complexity of the computational tasks that can be automated at all. Thus, the computability aspects of such a course should be confined to establishing that the intuitive notion of an algorithm can be rigorously defined, and to emphasizing the uncomputability of most functions and of some natural functions (e.g., the Halting predicate). This includes introducing the idea of a universal algorithm, but does not included extensive programming with Turing machines or extensive study of (complexity-free) Turing reductions. Needless to say, I oppose the teaching of finite automata (let alone context-free grammars) within such a course.

Expanding upon the opinions expressed in the last paragraph is beyond the scope of the current essay. On the other hand, the rest of this essay is independent of the foregoing remarks. That is, it refers to the basic material of complexity theory, regardless of the question within which course this material is taught and what role does it play in such a course.

1.5 Organization

Section 2 contains a presentation of the P-vs-NP Question both in terms of search problems and in terms of decision problems. Section 3 contains a general treatment of reductions as well as a subsection on "self-reducibility" (of search problems). Section 4 contains a presentation of the basic definitions and results of the theory of NP-completeness (as well as a mention of the existence of NP-sets that are neither in P nor NP-complete). Section 5 mentions three additional topics that are typically not taught in a basic course on computability and complexity theory. These topics include the conjectured non-existence of coNP-sets that are NP-complete, the existence of optimal search algorithms for NP-relations, and the notion of promise problems.

As a general rule, the more standard the material is, the less detail we provide about is actual technical contents. Our focus is on the conceptual contents of the material, and technical details are given merely for illustration. We stress again that this essay is not supposed to serve as a textbook, but rather as a conceptual framework.

The essay is augmented by a brief overview of complexity theory. Unlike the rest of this essay, which assumes familiarity with the material, this overview (Section 6) is supposed to be accessible to the novice (or an "outsider"), and may be used accordingly. One possible use is as a base for introductory comments on complexity theory to be made either at the beginning of a graduate course on the topic or at the end of the (currently prevailing) undergraduate course on computability and complexity theory.

[3] In fact, this essay is based on my notes for three lectures (covering the basic material), which were given in a graduate course on complexity theory (see [4]).

2 P versus NP

Most students have heard of P and NP before, but we suspect that many have not obtained a good explanation of what the P-vs-NP Question actually represents. This unfortunate situation is due to using the standard technical definition of NP (which refers to nondeterministic polynomial-time) rather than using (somehat more cumbersome) definitions that clearly capture the fundamental nature of NP. Below, we take the alternative approach. In fact, we present two fundamental formulations of the P-vs-NP Question, one in terms of search problems and the other in terms of decision problems.

Efficient computation. The teacher should discuss the association of efficient computation with polynomial-time algorithms, stressing that this association merely provides a convenient way of addressing fundamental issues.[4] In particular, polynomials are merely a "closed" set of moderately growing functions, where "closure" means closure under addition, multiplication and functional composition. These closure properties guarantee the closure of the class of efficient algorithms under natural algorithmic composition operations such as sequential execution and subroutine calls. (The specifics of the model of computation are also immaterial, as long as the model is "reasonable"; this strengthening of the Church–Turing Thesis is called the Cobham–Edmonds Thesis.)

2.1 The search version: finding versus checking

In the eyes of non-experts, search problems are more natural than decision problems: typically, people seeks solutions more than they stop to wonder whether or not solutions exist. Thus, we recommend starting with a formulation of the P-vs-NP Question in terms of search problems. Admittingly, the cost is more cumbersome formulations (presented in Figure 1), but it is more than worthwhile. Furthermore, the equivalence to the decision problem formulation gives rise to conceptually appealing exercises.

We focus on polynomially-bounded relations, where a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is polynomially-bounded if there exists a polynomial p such that for every $(x,y) \in R$ it holds that $|y| \leq p(|x|)$. For such a relation it makes sense to ask whether, given an "instance" x, one can efficiently find a "solution" y such that $(x,y) \in R$. The polynomial bound on the length of the solution (i.e., y) guarantees that the intrinsic complexity of outputting a solution may not be due to the length (or mere typing) of the required solution.

The class P as a natural class of search problems. With each polynomially-bounded relation R, we associate the following search problem: *given x find y such that $(x,y) \in R$ or state that no such y exists.* The class $\mathcal{P}$ *corresponds*[5] to the class of search problems that are solvable in polynomial-time; that is, a relation R (or rather the search problem of R) is polynomial-time solvable if there exists a polynomial-time algorithm that given x find y such that $(x,y) \in R$ or state that no such y exists.

[4] Indeed, we claim that these fundamental issues are actually independent of the aforementioned association. For example, the question of whether finding a solution is harder than verifying its validity makes sense under any reasonable notion of "hardness". Similarly, the claim that factoring (or any other "NP problem") is "easily reducible" to SAT holds for many reasonable notions of "easy to compute" mappings.

[5] We leave it to the teacher whether to actually define $\mathcal{P}$ (resp., $\mathcal{NP}$) as a class of search problems or to reserve this notion for the relevant class of decision problems (and merely talk about a "correspondence" between the search and decision problem classes). Our own preference is to introduce different notations for the search problem classes (see Figure 1).

The class NP as another natural class of search problems. A polynomially-bounded relation R is called an NP-relation if, given an alleged instance-solution pair, one can efficiently check whether or not the pair is valid; that is, there exists a polynomial-time algorithm that given x and y determines whether or not $(x, y) \in R$. The class $\mathcal{NP}$ *corresponds*[5] to the class of search problems for NP-relations (and contains a host of natural search problems). It is reasonable to focus on search problems for NP-relations, because the ability to efficiently recognize a valid solution seems to be a natural prerequisite for a discussion regarding the complexity of finding such solutions. (Indeed, one can introduce (unnatural) non-NP-relations for which the search problem is solvable in polynomial-time; still the restriction to NP-relations is very natural.)

The P versus NP question in terms of search problems: *Is it the case that the search problem of any NP-relation can be solved in polynomial-time?* In other words, if it is easy to check whether or not a given solution for a given instance is correct, then is it also easy to find a solution to a given instance?

If $\mathcal{P} = \mathcal{NP}$ (in terms of search problems) then this would mean that whenever solutions to given instances can be efficiently verified for correctness it is also the case that these solutions can be efficiently found (when given only the instance). This would mean that all reasonable search problems (i.e., all NP-relations) are easy to solve. Needless to say, such a situation would contradict the intuitive feeling (and daily experience) that some reasonable search problems are hard to solve. On the other hand, if $\mathcal{P} \neq \mathcal{NP}$ then there exist reasonable search problems (i.e., some NP-relations) that are hard to solve. This conforms with our daily experience by which some reasonable problems are easy to solve whereas others are hard to solve.

Recall that search problems refer to binary relations. For such a relation R, the corresponding search problem is given x to find y such that $(x, y) \in R$ (or assert that no such y exists). We suggest defining two classes of search problems.

- $\mathcal{PF}$ (standing for "Poly-Find") denotes the class of search problems that are solvable in polynomial-time. That is, $R \in \mathcal{PF}$ if there exists a polynomial time algorithm that given x finds y such that $(x, y) \in R$ (or assert that no such y exists).
- $\mathcal{PC}$ (standing for "Poly-Check") denotes the class of search problems that correspond to polynomially-bounded binary relations that are "checkable" in polynomial-time. That is, $R \in \mathcal{PC}$ if the following two conditions hold
  1. For some polynomial p, if $(x, y) \in R$ then $|y| \leq p(|x|)$.
  2. There exists a polynomial-time algorithm that given (x, y) determines whether or not $(x, y) \in R$.

In terms of search problems the P-vs-NP Question consists of asking whether or not $\mathcal{PC}$ is contained in $\mathcal{PF}$. The conjectured inequality $\mathcal{P} \neq \mathcal{NP}$ implies that $\mathcal{PC} \setminus \mathcal{PF} \neq \emptyset$.

Figure 1: P-vs-NP in terms of search problems: notational suggestions.

2.2 The decision version: proving versus verifying

We suggest starting by asserting the natural stature of decision problems (beyond their role in the study of search problems). After all, some people do care about the truth, and so determining whether a given object has some claimed property is an appealing problem. The P-vs-NP Question refers to the complexity of answering such questions for a wide and natural class of properties associated with the class $\mathcal{NP}$. The latter class refers to properties that have efficient proof systems allowing for the verification of the claim that a given object has a predetermined property (i.e., is a member of a predetermined set).

For an NP-relation R, we denote the set of instances having a solution by L_R; that is, $L_R = \{x : \exists y\ (x,y) \in R\}$. Such a set is called an NP-set, and $\mathcal{NP}$ denotes the class of all NP-sets. Intuitively, an NP-set is a set of valid statements (i.e., statements of membership of a given x in L_R) that can be efficiently verified when given adequate proofs (i.e., a corresponding NP-witness y such that $(x,y) \in R$). This leads to viewing NP-sets as proof systems.

NP-proof systems. Proof systems are defined in terms of their verification procedures. Here we focus on the natural class of efficient verification procedures, where efficiency is represented by polynomial-time computations. (We should either require that the time is polynomial in terms of the statement or confine ourselves to "short proofs" – that is, proofs of length that is bounded by a polynomial in the length of the statement.) NP-relations correspond to proof systems with efficient verification procedures. Specifically, the NP-relation R corresponds to the (proof system with a) verification procedure that checks whether or not the alleged statement-proof pair is in R. This proof system satisfies the natural completeness and soundness conditions: every true statement (i.e., $x \in L_R$) has a valid proof (i.e., an NP-witness y such that $(x,y) \in R$), whereas false statements (i.e., $x \notin L_R$) have no valid proofs (i.e., $(x,y) \notin R$ for all y's).

Recall that decision problems refer to membership in sets. We suggest defining two classes of decision problems, which indeed coincide with the standard definitions of $\mathcal{P}$ and $\mathcal{NP}$.

- $\mathcal{P}$ denotes the class of decision problems that are solvable in polynomial-time. That is, $S \in \mathcal{P}$ if there exists a polynomial time algorithm that given x determines whether or not $x \in S$.
- $\mathcal{NP}$ denotes the class of decision problems that have NP-proof systems. The latter are defined in terms of a (deterministic) polynomial-time verification algorithm. That is, $S \in \mathcal{NP}$ if there exists a polynomial p and a polynomial-time algorithm V that satisfy the following completeness and soundness conditions:

    1. Completeness: if $x \in S$ then there exists y of length at most $p(|x|)$ such that $V(x,y) = 1$. (Such a string y is called an NP-witness for $x \in S$.)
    2. Soundness: if $x \notin S$ then for every y it holds that $V(x,y) = 0$.

    Indeed, the point is defining $\mathcal{NP}$ as a class of sets of assertions having efficient verification procedures.

In terms of decision problems the P-vs-NP Question consists of asking whether or not $\mathcal{NP}$ is contained in $\mathcal{P}$. Since $\mathcal{P} \subseteq \mathcal{NP}$, the question is phrased as whether or not $\mathcal{NP}$ equals $\mathcal{P}$.

Figure 2: P-vs-NP in terms of decision problems: notational suggestions.

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

2.3 Equivalence of the two formulations

We strongly recommend proving that *the two formulations of the P-vs-NP Questions are equivalent.* That is, the search problem of every NP-relation is solvable in polynomial time if and only if membership in any NP-set can be decided in polynomial time (see Figure 3). This justifies the focus on the latter (simpler) formulation.

Referring the notations of Figures 1 and 2, we prove that $\mathcal{PC} \subseteq \mathcal{PF}$ if and only if $\mathcal{NP} = \mathcal{P}$.

- Suppose that the inclusion holds for the search version (i.e., $\mathcal{PC} \subseteq \mathcal{PF}$). Let L be an arbitrary NP-set and R_L be the corresponding witness relation. Then R_L is a NP-relation, and by the hypothesis its search problem is solvable in polynomial time (i.e., $R_L \in \mathcal{PC} \subseteq \mathcal{PF}$). This yields a polynomial-time decision procedure for L; i.e., given x try to find y such that $(x, y) \in R_L$ (and output "yes" iff such a y was found). Thus, $\mathcal{NP} = \mathcal{P}$ follows.
- Suppose that $\mathcal{NP} = \mathcal{P}$ (as classes of sets), and let R be an arbitrary NP-relation. Then the set $S_R \stackrel{\text{def}}{=} \{(x, y') : \exists y'' \text{ s.t. } (x, y'y'') \in R\}$ (where $y'y''$ denotes the concatenation of y' and y'') is in $\mathcal{NP}$ and hence in $\mathcal{P}$. This yields a polynomial-time algorithm for solving the search problem of R, by extending a prefix of a potential solution bit by bit (while using the decision procedure to determine whether or not the current prefix is valid). Thus, $\mathcal{PC} \subseteq \mathcal{PF}$ follows.

Figure 3: A proof that $\mathcal{PC} \subseteq \mathcal{PF}$ if and only if $\mathcal{NP} = \mathcal{P}$.

We also suggest mentioning that $\mathcal{NP}$ is sometimes defined as the class of sets that can be decided by a *fictitious* device called a nondeterministic polynomial-time machine (and that this is the source of the notation NP). The reason that this class of fictitious devices is important is because it captures (indirectly) the definition of NP-proof systems. We suggest proving that indeed the definition of $\mathcal{NP}$ in terms of nondeterministic polynomial-time machine equals our definition of $\mathcal{NP}$ (in terms of the class of sets having NP-proof systems).

3 Reductions and Self-reducibility

We assume that many students have heard of reductions, but again we fear that most of them have obtained a conceptually poor view of their nature. We believe that this is due to expositions that start with a technical definition of many-to-one (polynomial-time) reductions (i.e., Karp-reductions), rather than with a motivational discussion. Below, we take an the alternative approach, presenting first the general notion of (polynomial-time) reductions among computational problems, and viewing the notion of a Karp-reduction as an important special case that suffices (and is more convenient) in many cases.

3.1 The general notion of a reduction

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

The key property of reductions is that they translate efficient procedures for the subroutine into efficient procedures for the invoking machine. That is, if one problem is Cook-reducible to another problem and the latter is polynomial-time solvable then so is the former.

The most popular case is of reducing decision problems to decision problems, but we will also consider reducing search problems to search problems or reducing search problems to decision problems. Indeed, a good exercise is showing that the search problem of any NP-relation can be reduced to deciding membership in some NP-set (which is the actual contents of the second item of Figure 3).

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

Indeed, a Karp-reduction is a syntactically restricted notion of a reduction. This restricted case suffices for many cases (e.g., most importantly for the theory of NP-completeness (when developed for decision problems)), but not in case we want to reduce a search problem to a decision problem. Furthermore, whereas each decision problem is reducible to its complement, some decision problems are not Karp-reducible to their complement (e.g., the trivial decision problem).[6] Likewise, each decision problem in $\mathcal{P}$ is reducible to any computational problem by a reduction that does not use the subroutine at all, whereas such a trivial reduction is disallowed by the syntax of Karp-reductions. (Nevertheless, a popular exercise of dubious nature is to show that any decision problem in $\mathcal{P}$ is Karp-reducible to any *non-trivial* decision problem.)

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

We say that two problems are computationally equivalent if they are reducible to one another. This means that the two problems are essentially equally hard (or equally easy).

[6] We call a decision problem trivial if it refers to either the empty set or the set of all strings.

3.2 Self-reducibility of search problems

We suggest introducing the notion of self-reducibility[7] for several reasons. Most importantly, it further justifies the focus on decision problems (see discussion following Proposition 1). In addition, it illustrates the general notion of a reduction, and asserts its relevance beyond the theory of NP-completeness.

The search problem of R is called self-reducible if it can be reduced to the decision problem of $L_R = \{x : \exists y\ (x, y) \in R\}$ (rather than to the set S_R as in Figure 3). Note that the decision problem of L_R is always reducible to the search problem for R (e.g., invoke the search subroutine and answer "yes" if and only if it returns some string (rather than the "no solution" symbol)).

We will see that all NP-relations that correspond to NP-complete sets are self-reducible, mostly via "natural reductions". We start with SAT, the set of satisfiable Boolean formulae (in CNF). Let R_{SAT} be the set of pairs (ϕ, τ) such that τ is a satisfying assignment to the formulae ϕ. Note that R_{SAT} is an NP-relation (i.e., it is polynomially-bounded and easy to decide (by evaluating a Boolean expression)).

Proposition 1 (R_{SAT} is self-reducible): *The search problem of R_{SAT} is reducible to SAT.*

Thus, the search problem of R_{SAT} is computationally equivalent to deciding membership in SAT. Hence, in studying the complexity of SAT, we also address the complexity of the search problem of R_{SAT}. This justifies the relevance of decision problems to search problems in a stronger sense than established in Section 2.3: The study of decision problems determines not only the complexity of the class of "NP-search" problems but rather determines the complexity of each individual search problem that is self-reducible.

Proof: Given a formula ϕ, we use a subroutine for SAT in order to find a satisfying assignment to ϕ (in case such an assignment exists). First, we query SAT on ϕ itself, and return "no solution" if the answer we get is 'false'. Otherwise, we let τ, initiated to the empty string, denote a prefix of a satisfying assignment of ϕ. We proceed in iterations, where in each iteration we extend τ by one bit. This is done as follows: First we derive a formula, denoted ϕ', by setting the first $|\tau| + 1$ variables of ϕ according to the values $\tau 0$. Next we query SAT on ϕ' (which means that we ask whether or not $\tau 0$ is a prefix of a satisfying assignment of ϕ). If the answer is positive then we set $\tau \leftarrow \tau 0$ else we set $\tau \leftarrow \tau 1$ (because if τ is a prefix of a satisfying assignment of ϕ and $\tau 0$ is not a prefix of a satisfying assignment of ϕ then $\tau 1$ must be a prefix of a satisfying assignment of ϕ).

A key point is that each formula ϕ' (which contains Boolean variables as well as constants) can be simplified to contain no constants (in order to fit the canonical definition of SAT, which disallows Boolean constants). That is, after replacing some variables by constants, we should simplify clauses according to the straightforward boolean rules (e.g., a false literal can be omitted from a clause and a true literal appearing in a clause allows omitting the entire clause). ■

Advanced comment: A reduction analogous to the one used in the proof of Proposition 1 can be presented also for other NP-search problems (and not only for NP-complete ones).[8] Consider, for

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

[8] We assume that the students have heard of NP-completeness. If this assumption does not hold for your class, then the presentation of the following material should be postponed (to Section 4.1 or to an even later stage).

example, the problem Graph 3-Colorability and prefixes of a 3-coloring of the input graph. Note, however, that in this case the process of getting rid of constants (representing partial solutions) is more involved.[9] In general, if you don't see a "natural" self-reducibility process for some NP-complete relation, you should know that a self-reduction process does exist (alas it maybe not be a natural one).

Theorem 2 *The search problem of the NP-relation of any NP-complete set is self-reducible.*

Proof: Let R be an NP-relation of the NP-complete set L_R. In order to reduce the search problem of R to deciding L_R, we compose the three reductions mentioned next:

1. The search problem of R is reducible to the search problem of R_{SAT} (by the NP-completeness of the latter).
2. The search problem of R_{SAT} is reducible to SAT (by Proposition 1).
3. The decision problem SAT is reducible to the decision problem L_R (by the NP-completeness of the latter).

The theorem follows. ■

4 NP-completeness

Some (or most) students have heard of NP-completeness before, but we suspect that many have missed important conceptual points. Specifically, we stress that the mere existence of NP-complete sets (regardless of whether this is SAT or some other set) is amazing.

4.1 Definitions

The standard definition is that a set is NP-complete if it is in $\mathcal{NP}$ and every set in $\mathcal{NP}$ is reducible to it via a Karp-reduction. Indeed, there is no reason to insist on Karp-reductions (rather than using arbitrary reductions), except that the restricted notion suffices for all positive results and is easier to work with.

We will also refer to the search version of NP-completeness. We say that a binary relation is NP-complete if it is an NP-relation and every NP-relation is reducible to it.

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

4.2 The existence of NP-complete problems

We suggest not to confuse the mere existence of NP-complete problems, which is remarkable by itself, with the even more remarkable existence of "natural" NP-complete problems. We believe that the following proof facilitates the delivery of this message as well as focusing on the essence of NP-completeness, rather than on more complicated technical details.

[9] Details can left as an exercise to the student. You may hint that a partial 3-coloring can be hard-wired into the graph by augmenting the graph with adequate gadgets that force equality (or inequality) between the colors of two vertices (of our choice).

Theorem 3 *There exist NP-complete relations and sets.*

Proof: The proof (as well as any other NP-completeness proof) is based on the observation that some NP-relations (resp., NP-sets) are "rich enough" to encode all NP-relations (resp., NP-sets). This is most obvious for the "generic" NP-relation, denoted R_U (and defined below), which is used to derive the simplest proof of the current theorem.

The relation R_U consists of pairs $(\langle M, x, 1^t \rangle, y)$ such that M is a description of a (deterministic) Turing machine that accepts the pair (x, y) within t steps, where $|y| \leq t$. (Instead of requiring that $|y| \leq t$, one may require that M is canonical in the sense that it reads its entire input before halting.) It is easy to see that R_U is an NP-relation, and thus $L_U \stackrel{\text{def}}{=} \{\overline{x} : \exists y\ (\overline{x}, y) \in R_U\}$ is an NP-set. Indeed, R_U is recognizable by a universal Turing machine, which on input $(\langle M, x, 1^t \rangle, y)$ emulates (t steps of) the computation of M on (x, y), and U indeed stands for universal (machine). (Thus, the proof extends to any reasonable model of computation, which has adequate universal machines.)

We now turn to showing that any NP-relation is reducible to R_U. As a warm-up, let us first show that any NP-set is Karp-reducible to L_U. Let R be an NP-relation, and $L_R = \{x : \exists y\ (x, y) \in R\}$ be the corresponding NP-set. Let p_R be a polynomial bounding the length of solutions in R (i.e., $|y| \leq p_R(|x|)$ for every $(x, y) \in R$), let M_R be a polynomial-time machine deciding membership (of alleged (x, y) pairs) in R, and let t_R be a polynomial bounding its running-time. Then, the Karp-reduction maps an instance x (for L) to the instance $\langle M_R, x, 1^{t_R(|x|+p_R(|y|))} \rangle$.

Note that this mapping can be computed in polynomial-time, and that $x \in L$ if and only if $\langle M_R, x, 1^{t_R(|x|+p_R(|y|))} \rangle \in L_U$.

To reduce the search problem of R to the search problem of R_U, we use essentially the same reduction. On input an instance x (for R), we make the query $\langle M_R, x, 1^{t_R(|x|+p_R(|y|))} \rangle$ to the search problem of R_U and return whatever the latter returns. Note that if $x \notin L_R$ then the answer will be "no solution", whereas for every x and y it holds that $(x, y) \in R$ if and only if $(\langle M_R, x, 1^{t_R(|x|+p_R(|y|))} \rangle, y) \in R_U$. ■

Advanced comment. Note that the role of 1^t in the definition of R_U is to make R_U an NP-relation. In contrast, consider the relation $R_H \stackrel{\text{def}}{=} \{(\langle M, x \rangle, y) : M(xy) = 1\}$ (which corresponds to the halting problem). Indeed, the search problem of any relation (an in particular of any NP-relation) is Karp-reducible to the search problem of R_H, but the latter is not solvable at all (i.e., there exists no algorithm that halts on every input and on input $\overline{x} = \langle M, x \rangle$ outputs y such that $(\overline{x}, y) \in R_H$ iff such a y exists).

4.3 CSAT, SAT, and other NP-complete problems

Once the mere existence of NP-complete problems has been established, we suggest informing the students of the fact that many natural problems are NP-complete, and demonstrating this fact with a few examples. Indeed, SAT is a good first example, both because the reduction to it is instructive and because it is a convenient starting point to further reductions. As a second example, we suggest various variants of the Set Cover problem. Additional reductions may be deferred to homework assignments, and presenting them in class seems inadequate in the context of a course on complexity theory.

We suggest establishing the NP-completeness of SAT by a reduction from the circuit satisfaction problem (CSAT), after establishing the NP-completeness of the latter. Doing so allows decoupling two important issues in the proof of the NP-completeness of SAT: (1) the emulation of Turing machines by circuits, and (2) the encoding of circuits by formulae with auxiliary variables. Following is a rough outline, which focuses on the decision version.

CSAT. Define Boolean circuits as directed acyclic graphs with internal vertices, called gates, labeled by Boolean operations (of arity either 2 or 1), and external vertices called terminals that are associated with either inputs or outputs. When setting the inputs of such a circuit, all internal nodes are assigned values in the natural way, and this yields a value to the output(s), called an evaluation of the circuit on the given input. Define the satisfiability problem of such circuits as determining, for a given circuit, whether there exists a setting to its inputs that makes its (first) output evaluate to 1. Prove the NP-completeness of the circuit satisfaction problem (CSAT), by reducing any NP-set to it (where the set is represented by the machine that recognizes the corresponding NP-relation). The reduction boils down to encoding possible computations of a Turing machine by a corresponding layered circuit, where each layer represents an instantaneous configuration of the machine, and the relation between consecutive configurations is captured by ("uniform") local gadgets in the circuit. For further details, see Figure 4. (The proof extends to any other "reasonable" model of efficient computation.)

Following are some additional comments on the proof of the NP-completeness of CSAT. These comments refer to the high-level structure of the reduction, and do not provide a full (low-level) description of it.

For a machine M_R (as in the proof of Theorem 3), we will represent the computation of M_R on input (x, y), where x is the input to the reduction and y is undetermined, by a circuit C_x that takes such a string y as input. Thus, $C_x(y) = 1$ if and only if M_R accepts (x, y), and so C_x is satisfiable if and only if $x \in L_R$. The reduction maps x to a circuit C_x as follows.

The circuit C_x consists of layers such that the i^{th} layers of wires (connecting the $i-1^{\text{st}}$ and i^{th} layers of vertices) represents the instantaneous configuration of $M_R(x, y)$ just before the i^{th} step. In particular, the gates of the $i+1^{\text{st}}$ layer are designed to guaranteed that the instantaneous configuration of $M_R(x, y)$ just before the i^{th} step is transformed to the instantaneous configuration of $M_R(x, y)$ just before the $i+1^{\text{st}}$ step. Only the first layer of C_x depends on x itself (which is "hard-wired" into the circuit). The rest of the construction depends only on $|x|$ and M_R.

Figure 4: Encoding computations of a Turing machine in a Boolean circuit.

The above reduction is called "generic" because it (explicitly) refers to any (generic) NP-set. However, the common practice is to establish NP-completeness by a reduction from some NP-complete set (i.e., a set already shown to be NP-complete). This practice is based on the fact that if an NP-complete problem Π is reducible to some problem Π' in NP then Π' is NP-complete. The proof of this fact boils down to asserting the transitivity of reductions.

SAT. Define Boolean formulae, which may be viewed as Boolean circuits with a tree structure. Prove the NP-completeness of the formula satisfaction problem (SAT), even when the formula is given in a nice form (i.e., CNF). The proof is by a reduction from CSAT, which in turn boils down to introducing auxiliary variables in order to cut the computation of a deep circuit into a conjunction of related computations of shallow (i.e., depth-2) circuits (which may be presented as CNF formulae). The aforementioned auxiliary variables hold the possible values of the internal wires of the circuit.

3SAT. Note that the formulae resulting from the latter reduction are in conjunctive normal form (CNF) with each clause referring to three variables (i.e., two corresponding to the input wires of a node/gate and one to its output wire). Thus, the above reduction actually establishes the NP-completeness of 3SAT (i.e., SAT restricted to CNF formula with up to three variables per clause). Alternatively, reduce SAT (for CNF formula) to 3SAT (i.e., satisfiability of 3CNF formula) by replacing long clauses with conjunctions of three-variable clauses using auxiliary variables.

In order to establish the NP-completeness of the search version of the aforementioned problems we need to present a polynomial-time mapping of solutions for the target problem (e.g., SAT) to solutions for the origin problem (e.g., CSAT). Note that such a mapping is typically explicit in the argument establishing the validity of the Karp-reduction.

Set Cover and other problems. If time permits, one may want to present another class of NP-complete problems, and our choice is of Set Cover. There is a simple reduction from SAT to Set Cover (with the sets corresponding to the sets of clauses that are satisfied when assigning a specific Boolean variable a specific Boolean value). When applied to a restricted version of SAT in which each variable appears in at most three clauses, the same reduction implies the NP-completeness of a version of Set Cover in which each set contains at most three elements. (Indeed, one should first establish the NP-completeness of the aforementioned restricted version of SAT.) Using the restricted version of Set Cover one may establish the NP-completeness of Exact Cover (even when restricted to 3-element sets). The latter problem is a convenient starting point for further reductions.

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

Many (to say the least) other NP-sets have been shown to be NP-complete. A very partial list includes Graph 3-Colorability, Subset Sum, (Exact) Set Cover, and the Traveling Salesman Problem. (Hundreds of other natural problems can be found in [3].) Things reach a situation in which some computer scientists seem to expect any NP-set to be either NP-complete or in $\mathcal{P}$. This naive view is wrong:

Theorem 4 *Assuming $\mathcal{NP} \neq \mathcal{P}$, there exist NP-sets that are neither NP-complete nor in $\mathcal{P}$.*

We mention that some natural problems (e.g., factoring) are conjectured to be neither solvable in polynomial-time nor NP-hard, where a problem Π is NP-hard if any NP-set is reducible to solving Π. See discussion following Theorem 5. We recommend to either state Theorem 4 without a proof or merely provide the proof idea (which is sketched next).

Proof idea. The proof is by modifying a set in $\mathcal{NP} \setminus \mathcal{P}$ such that to fail all possible reductions (to this set) and all possible polynomial-time decision procedures (for this set). Specifically, we start with some $L \in \mathcal{NP} \setminus \mathcal{P}$ and derive $L' \subset L$ (which is also in $\mathcal{NP} \setminus \mathcal{P}$) by making each reduction (say of L) to L' fail by dropping finitely many elements from L (until the reduction fails), whereas all possible polynomial-time fail to decide L' (which differ from L only on a finite number of inputs). We use the fact that any reduction (of some set in $\mathcal{NP} \setminus \mathcal{P}$) to a finite set (i.e., a finite subset of L) must fail (and this failure is due to a finite set of queries), whereas any efficient decision procedure for L (or L modified on finitely many inputs) must fail on some finite portion of all possible inputs (of L). The process of modifying L into L' proceeds in iterations, alternatively failing a potential reduction (by dropping sufficiently many strings from the rest of L) and failing a potential decision procedure (by including sufficiently many strings from the rest of L). This can be done efficiently because it is inessential to determine the optimal points of alternation (where sufficiently many strings were dropped (resp., included) to fail a potential reduction (resp., decision procedure)). Thus, L' is the intersection of L and some set in $\mathcal{P}$, which implies that $L' \in \mathcal{NP} \setminus \mathcal{P}$.

5 Three additional topics

The following topics are typically not mentioned in a basic course on complexity. Still, pending on time constraints, we suggest covering them at some minimal level.

5.1 The class coNP and NP-completeness

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

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

Specifically, $\text{co}\mathcal{NP} = \{\{0,1\}^* \setminus L : L \in \mathcal{NP}\}$ is the class of sets that are complements of NP-sets. That is, if R is an NP-relation and $L_R = \{x : \exists y\ (x,y) \in R\}$ is the associated NP-set then $\{0,1\}^* \setminus L_R = \{x : \forall y\ (x,y) \notin R\}$ is the corresponding coNP-set.

It is widely believed that $\mathcal{NP}$ is not closed under complementation (i.e., $\mathcal{NP} \neq \text{co}\mathcal{NP}$). Indeed, this conjecture implies $\mathcal{P} \neq \mathcal{NP}$ (because $\mathcal{P}$ is closed under complementation). The conjecture $\mathcal{NP} \neq \text{co}\mathcal{NP}$ means that some coNP-sets (e.g., the complements of NP-complete sets) do not have NP-proof systems; that is, there is no NP-proof system for proving that a given formula is not satisfiable.

If indeed $\mathcal{NP} \neq \text{co}\mathcal{NP}$ then some NP-sets cannot be Karp-reducible to any coNP-set.[10] However, each NP-set is reducible to some coNP-set (by a straightforward Cook-reduction that just flips the answer), and so the non-existence of such Karp-reduction does not seem to represent anything really fundamental. In contrast, we mention that $\mathcal{NP} \neq \text{co}\mathcal{NP}$ implies that some NP-sets cannot be reduced to sets in the intersection $\mathcal{NP} \cap \text{co}\mathcal{NP}$ (even under general (i.e., Cook) reductions). Specifically,

Theorem 5 *If $\mathcal{NP} \cap \text{co}\mathcal{NP}$ contains an NP-hard set then $\mathcal{NP} = \text{co}\mathcal{NP}$.*

Recall that a set is NP-hard if every NP-set is reducible to it (possibly via a general reduction). Since $\mathcal{NP} \cap \text{co}\mathcal{NP}$ is conjectured to be a proper superset of $\mathcal{P}$, it follows (using the conjecture $\mathcal{NP} \neq \text{co}\mathcal{NP}$) that there are NP-sets that are neither in $\mathcal{P}$ nor NP-hard (specifically, the sets in $(\mathcal{NP} \cap \text{co}\mathcal{NP}) \setminus \mathcal{P}$ are neither in $\mathcal{P}$ nor NP-hard). Notable candidates are sets related to the integer factorization problem (e.g., the set of pairs (N, s) such that s has a square root modulo N that is a quadratic residue modulo N and the least significant bit of s equals 1).

Proof: Suppose that $L \in \mathcal{NP} \cap \text{co}\mathcal{NP}$ is NP-hard. Given any $L' \in \text{co}\mathcal{NP}$, we will show that $L' \in \mathcal{NP}$. We will merely use the fact that L' reduces to L (which is in $\mathcal{NP} \cap \text{co}\mathcal{NP}$). Such a reduction exists because L' is reducible $\overline{L}' \stackrel{\text{def}}{=} \{0,1\}^* \setminus L'$ (via a general reduction), whereas $\overline{L}' \in \mathcal{NP}$ and thus is reducible to L (which is NP-hard).

[10] Specifically, we claim that sets in $\mathcal{NP} \setminus \text{co}\mathcal{NP}$ cannot be Karp-reducible to sets in $\text{co}\mathcal{NP}$. In fact, we prove that only sets in $\text{co}\mathcal{NP}$ are Karp-reducible to sets in $\text{co}\mathcal{NP}$. Equivalently, let us prove that only sets in $\mathcal{NP}$ are Karp-reducible to sets in $\mathcal{NP}$, where the equivalence follows by noting that a reduction of L to L' is also a reduction of $\{0,1\}^* \setminus L$ to $\{0,1\}^* \setminus L'$. Indeed, suppose that L Karp-reduces to $L' \in \mathcal{NP}$. Then $L \in \mathcal{NP}$ by virtue of the NP-relation $\{(x,y) : (f(x), y) \in R'\}$, where R' is the witness relation of L'.

To show that $L' \in \mathcal{NP}$, we will present an NP-relation, R', that characterizes L' (i.e., $L' = \{x : \exists y\ (x,y) \in R'\}$). The relation R' consists of pairs of the form $(x, ((z_1, \sigma_1, w_1), ..., (z_t, \sigma_t, w_t)))$, where on input x the reduction of L' to L accepts after making the queries $z_1, ..., z_t$, obtaining the corresponding answers $\sigma_1, ..., \sigma_t$, and for every i it holds that if $\sigma_i = 1$ then w_i is an NP-witness for $z_i \in L$, whereas if $\sigma_i = 0$ then w_i is an NP-witness for $z_i \in \{0,1\}^* \setminus L$.

We stress that we use the fact that both L and $\overline{L} \stackrel{\text{def}}{=} \{0,1\}^* \setminus L$ are NP-sets, and refer to the corresponding NP-witnesses. Note that R' is indeed an NP-relation: The length of solutions is bounded by the running-time of the reduction (and the corresponding NP-witnesses). Membership in R' is decided by checking that the sequence of (z_i, σ_i)'s matches a possible query-answer sequence in an accepting execution of the reduction[11] (ignoring the correctness of the answers), and that all answers (i.e., σ_i's) are correct. The latter condition is easily verified by use of the corresponding NP-witnesses. ■

5.2 Optimal search algorithms for NP-relations

The title of this section sounds very promising, but our guess is that the students will be less excited once they see the proof. We claim the existence of an *optimal search algorithm for any NP-relation.* Furthermore, we will explicitly present such an algorithm, and prove that it is optimal (without knowing its running time).

Theorem 6 *For every NP-relation R there exists an algorithm A that satisfies the following:*

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

We stress that the hidden constant in the O-notation depends only on A', but in the following proof the dependence is exponential in the length of the description of algorithm A' (and it is not known whether a better dependence can be achieved). Optimality holds in a "point-wise" manner (i.e., for every input), and the additive polynomial term (i.e., $p(|x|)$) is insignificant in case the NP-problem is not solvable in polynomial-time. On the other hand, the optimality of algorithm A refers only to inputs that have a solution (i.e., $x \in L_R$). Interestingly, we establish the optimality of A without knowing what its (optimal) running-time is. Furthermore, *the P-sv-NP Question boils down to determining the running time of a single explicitly presented algorithm* (i.e., the optimal algorithm A). Finally, we note that the theorem as stated refers only to models of computation that have machines that can emulate a given number of steps of other machines with a constant overhead. We mention that in most natural models the overhead of such emulation is at most poly-logarithmic in the number of steps, in which case it holds that $t_A(x) = \widetilde{O}(t_{A'}(x) + p(|x|))$.

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

Since there are infinitely many possible algorithms, we should clarify what we mean by "running them all in parallel". What we mean is to run them at different rates such that the infinite

[11] That is, we need to verify that on input x, after obtaining the answers $\sigma_1, ..., \sigma_{i-1}$ to the first $i-1$ queries, the i^{th} query made by the reduction equals z_i.

sum of rates converges to 1 (or any other constant). Specifically, we will run the i^{th} possible algorithm at rate $1/(i+1)^2$. Note that a straightforward implementation of this idea may create a significant overhead, involved in switching frequently from the computation of one machine to another. Instead we present an alternative implementation that proceeds in iterations. In the j^{th} iteration, for $i = 1, ..., 2^{j/2}$, we emulate $2^j/(i+1)^2$ steps of the i^{th} machine. Each of these emulations is conducted in one chunk, and thus the overhead of switching between the various emulations is insignificant (in comparison to the total number of steps being emulated). We stress that in case some of these emulations halts with output y, algorithm A invokes M on input (x, y) and output y if and only if $M(x, y) = 1$. Furthermore, the verification of a solution provided by a candidate algorithm is also emulated at the expense of its step-count. (Put in other words, we augment each algorithm with a canonical procedure (i.e., M) that checks the validity of the solution offered by the algorithm.)

In order to guarantee that A also halts on $x \notin L_R$, we let it run an exhaustive search for a solution, in parallel to all searches, and halt with output $\perp$ in case this exhaustive search fails.

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

5.3 Promise Problems

Promise problems are a natural generalization of decision problems (and search problems can be generalized in a similar manner). In fact, in many cases, promise problems provide the more natural formulation of a decision problem. Formally, promise problems refer to a three-way partition of the set of all strings into yes-instances, no-instances, and instances that violate the promise. A potential decider is only required to distinguish yes-instances from no-instances, and is allowed arbitrary behavior on inputs that violate the promise. Standard decision problems are obtained as a special case by postulating that all inputs are allowed (i.e., the promise is trivial).

In contrary to the common perception, promise problems are no offshoot for *abnormal* situations, but *are rather the norm*: Indeed, the standard and natural presentation of natural decision problems is actually in terms of promise problems, although the presentation rarely refers explicitly to the terminology of promise problems. Consider a standard entry in [3] (or any similar compendium) reading something like "given a planar graph, determine whether or not ...". A more formal statement will refer to strings that represent planar graphs. Either way, the natural formulation actually refers to a promise problem (where the promise in this case is that the input is a planar graph).

We comment that the discrepancy between the intuitive promise problem formulation and the standard formulation in terms of decision problems can be easily bridged in the case that there exists an efficient algorithm for determining membership in the "promise set" (i.e., the set of instances that satisfy the promise). In this case, the promise problem is computationally equivalent to deciding membership in the set of yes-instances. However, in case the promise set is not tractable, the terminology of promise problems is unavoidable. Examples include the notion of "unique solutions", the formulation of "gap problems" that capture various approximation tasks, and complete problems for various probabilistic complexity classes. For a recent survey on promise problems and their applications, the interested reader is referred to [5].

6 A brief overview of Complexity Theory

(*The following text was originally written as a brief overview of complexity theory, intended for the novice. It can also be used as a basis for communicating the essence of complexity theory to the outside (i.e., to scientists in other disciplines and even to the general interested public). Thus, unlike the rest of this essay, which is intended for the teacher, this section is intended for the student (or for other "outsiders" that the teacher may wish to address). The text starts with an overview of the P-vs-NP Question and the theory of NP-completeness, repeating themes that were expressed in the previous sections. Still, in light of the different potential uses of this text, I preferred not to eliminate this part of the overview.*)

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

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

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

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

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

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

The foregoing discussion of the P-versus-NP Question also hints to *the importance of representation*, a phenomenon that is central to complexity theory. In general, complexity theory is concerned with problems the solutions of which are implicit in the problem's statement. That is, the problem contains all necessary information, and one merely needs to process this information in order to supply the answer.[12] Thus, complexity theory is concerned with manipulation of information, and its transformation from one representation (in which the information is given) to another representation (which is the one desired). Indeed, a solution to a computational problem is merely a different representation of the information given; that is, a representation in which the answer is explicit rather than implicit. For example, the answer to the question of whether or not a given Boolean formula is satisfiable is implicit in the formula itself (but the task is to make the answer explicit). Thus, complexity theory clarifies a central issue regarding representation; that is, the distinction between what is explicit and what is implicit in a representation. Furthermore, it even suggests a quantification of the level of non-explicitness.

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

The concept of *randomness* has puzzled thinkers for ages. Their perspective can be described as ontological: they asked "what is randomness" and wondered whether it exist at all (or is the world deterministic). The perspective of complexity theory is behavioristic: it is based on defining objects as equivalent if they cannot be told apart by any efficient procedure. That is, a coin toss is (defined to be) "random" (even if one believes that the universe is deterministic) if it is infeasible to predict the coin's outcome. Likewise, a string (or a distribution of strings) is "random" if it

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

is infeasible to distinguish it from the uniform distribution (regardless of whether or not one can generate the latter). Interestingly, randomness (or rather pseudorandomness) defined this way is efficiently expandable; that is, under a reasonable complexity assumption (to be discussed next), short pseudorandom strings can be deterministically expanded into long pseudorandom strings. Indeed, it turns out that randomness is intimately related to intractability. Firstly, note that the very definition of pseudorandomness refers to intractability (i.e., the infeasibility of distinguishing a pseudorandomness object from a uniformly distributed object). Secondly, as hinted above, a complexity assumption that refers to the existence of functions that are easy to evaluate but hard to invert (called *one-way functions*) imply the existence of deterministic programs (called *pseudorandom generators*) that stretch short random seeds into long pseudorandom sequences. In fact, it turns out that the existence of pseudorandom generators is equivalent to the existence of one-way functions.

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

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

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

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

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

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

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

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

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

In case the reader feels dizzy, it is no wonder. We took an ultra-fast air-tour of some mountain tops, and dizziness is to be expected. Needless to say, a good graduate course in complexity theory should consist of climbing some of these mountains by foot, step by step, and stopping to look around and reflect.

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

Historical Notes

Many sources provide historical accounts of the developments that led to the formulation of the *P vs NP Problem* and the development of the theory of NP-completeness (see, e.g., [3]). We thus refrain from attempting to provide such an account.

One technical point that we mention is that the three "founding papers" of the theory of NP-completeness (i.e., [1, 6, 8]) refer to the three different terms of reductions used above. Specifically, Cook used the general notion of polynomial-time reduction [1], often referred to as Cook-reductions. The notion of Karp-reductions originates from Karp's paper [6], whereas its augmentation to search problems originates from Levin's paper [8]. It is worth noting that unlike Cook and Karp's works, which treat decision problems, Levin's work is stated in terms of search problems.

The existence of NP-sets that are neither in P nor NP-complete (i.e., Theorem 4) was proven by Ladner [7], Theorem 5 was proven by Selman [9], and the existence of optimal search algorithms for NP-relations (i.e., Theorem 6) was proven by Levin [8]. (Interestingly, the latter result was proved in the same paper in which Levin presented the discovery of NP-completeness, independently of Cook and Karp.) Promise problems were explicitly introduced by Even, Selman and Yacobi [2].

Acknowledgments

I am grateful to Arny Rosenberg, Alan Selman, Salil Vadhan, and an anonymous referee for their useful comments and suggestions.

References

[1] S.A. Cook. The Complexity of Theorem Proving Procedures. In *3rd STOC*, pages 151–158, 1971.

[2] S. Even, A.L. Selman, and Y. Yacobi. The Complexity of Promise Problems with Applications to Public-Key Cryptography. *Inform. and Control*, Vol. 61, pages 159–173, 1984.

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

[4] O. Goldreich. *Introduction to Complexity Theory – notes for a one-semester course*. Weizmann Institute of Science, Spring 2002. Available from `http://www.wisdom.weizmann.ac.il/~oded/cc.html`

[5] O. Goldreich. On Promise Problems: In memory of Shimon Even (1935–2004). *ECCC*, TR05-018, January 2005.

[6] R.M. Karp. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, R.E. Miller and J.W. Thatcher (eds.), Plenum Press, pages 85–103, 1972.

[7] R.E. Ladner. On the Structure of Polynomial Time Reducibility. *Jour. of the ACM*, 22, 1975, pages 155–171.

[8] L.A. Levin. Universal Search Problems. *Problemy Peredaci Informacii 9*, pages 115–116, 1973. Translated in *problems of Information Transmission 9*, pages 265–266.

[9] A. Selman. On the structure of NP. *Notices Amer. Math. Soc.*, Vol 21 (6), page 310, 1974.