

Chapter 2

P, NP and NP-Completeness

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

Luke, 1:1–4

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

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

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

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

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

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

We stress that NP-complete problems are not the only hard problems in NP (i.e., if P is different than NP then NP contains problems that are neither NP-complete nor in P). We also note that the P-vs-NP Question is not about inventing sophisticated algorithms or ruling out their existence, but rather boils down to the analysis of a single known algorithm; that is, we will present an optimal search algorithm for any problem in NP, while having not clue about its time complexity.

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

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

Organization: In Section 2.1 we present the two formulations of the P-vs-NP Question. The general notion of a reduction is presented in Section 2.2, where we highlight its applicability outside the domain of NP-completeness. Section 2.3 is devoted to the theory of NP-completeness, whereas Section 2.4 treats three relatively advanced topics (i.e., the framework of promise problems, the existence of optimal search algorithms for NP, and the class coNP).

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

2.1 The P versus NP Question

Our daily experience is that it is harder to solve a problem than it is to check the correctness of a solution. Is this experience merely a coincidence or does it represent a fundamental fact of life (or a property of the world)? This is the essence of the P versus NP Question, where *P represents search problems that are efficiently solvable and NP represents search problems for which solutions can be efficiently checked.*

Another natural question captured by the P versus NP Question is whether proving theorems is harder than verifying the validity of these proofs. In other words, the question is whether deciding membership in a set is harder than being convinced of this membership by an adequate proof. In this case, *P represents decision problems that are efficiently solvable, whereas NP represents sets that have efficiently checkable proofs of membership.*

These two meanings of the P versus NP Question are rigorously presented and discussed in Sections 2.1.1 and 2.1.2, respectively. The equivalence of the two formulations is shown in Section 2.1.3, and the common belief that P is different from NP is further discussed in Section 2.1.5. Let us start by recalling the notion of efficient computation.

Teaching note: Most students have heard of P and NP before, but we suspect that many have not obtained a good explanation of what the P vs NP Question actually represents. This unfortunate situation is due to using the standard technical definition of NP (which refers to the fictitious and confusing device called a non-deterministic polynomial-time machine). Instead, we advocate the use of the more cumbersome definitions sketched in the foregoing paragraphs (and elaborated in Sections 2.1.1 and 2.1.2), which clearly capture the fundamental nature of NP.

The notion of efficient computation. Recall that we associate efficient computation with polynomial-time algorithms.¹ This association is justified by the fact that polynomials are a class of moderately growing functions that is closed under operations that correspond to natural composition of algorithms. Furthermore, the class of polynomial-time algorithms is independent of the specific model of computation, as long as the latter is “reasonable” (cf. the Cobham-Edmonds Thesis). Both issues are discussed in §1.2.3.4.

Advanced note on the representation of problem instances. As noted in Section 1.2.2, many natural (search and decision) problems are captured more naturally by the terminology of promise problems (cf. Section 2.4.1), where the domain of possible instances is a subset of $\{0, 1\}^*$ rather than $\{0, 1\}^*$ itself. For example, computational problems in graph theory presume some simple encoding of graphs as strings, but this encoding is typically not onto (i.e., not all strings encode graphs), and thus not all strings are legitimate instances. However, in these cases, the set of legitimate instances (e.g., encodings of graphs) is efficiently recognizable (i.e., membership in it can be decided in polynomial-time). Thus, artificially extending the set of instances to the set of all possible strings (and allowing trivial solutions for the corresponding dummy instances) does not change the complexity of the original problem. We further discuss this issue in Section 2.4.1.

2.1.1 The search version: finding versus checking

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

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

One type of search problems that cannot be solved efficiently consists of search problems for which the solutions are too long in terms of the problem’s instances.

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

In such a case, merely typing the solution amounts to an activity that is deemed inefficient. Thus, we focus our attention on search problems that are not in this class. That is, we consider only search problems in which the length of the solution is bounded by a polynomial in the length of the instance. Recalling that search problems are associated with binary relations (see Definition 1.1), we focus our attention on polynomially bounded relations.

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

Recall that $(x,y) \in R$ means that y is a solution to the problem instance x , where R represents the problem itself. For example, in the case of finding a prime factor of a given integer, we refer to a relation R such that $(x,y) \in R$ if the integer y is a prime factor of the integer x .

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

2.1.1.1 The class P as a natural class of search problems

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

Definition 2.2 (efficiently solvable search problems):

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

Note that $R(x)$ denotes the set of valid solutions for the problem instance x . Thus, the solver A is required to find a valid solution (i.e., satisfy $A(x) \in R(x)$) whenever

such a solution exists (i.e., $R(x)$ is not empty). On the other hand, if the instance x has no solution (i.e., $R(x) = \emptyset$) then clearly $A(x) \notin R(x)$. The extra condition (also made in Definition 1.1) requires that in this case $A(x) = \perp$. Thus, algorithm A always outputs a correct answer, which is a valid solution in the case that such a solution exists and otherwise provides an indication that no solution exists.

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

2.1.1.2 The class NP as another natural class of search problems

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

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

Definition 2.3 (search problems with efficiently checkable solutions):

- *The search problem of a polynomially bounded relation $R \subseteq \{0, 1\}^* \times \{0, 1\}^*$ has efficiently checkable solutions if there exists a polynomial time algorithm A such that, for every x and y , it holds that $A(x, y) = 1$ if and only if $(x, y) \in R$.*
- *We denote by \mathcal{PC} (standing for “Polynomial-time Check”) the class of search problems that correspond to polynomially-bounded binary relations that have efficiently checkable solutions. That is, $R \in \mathcal{PC}$ if the following two conditions hold:*

1. For some polynomial p , if $(x, y) \in R$ then $|y| \leq p(|x|)$.
2. There exists a polynomial-time algorithm that given (x, y) determines whether or not $(x, y) \in R$.

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

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

2.1.1.3 The P versus NP question in terms of search problems

Is it the case that every search problem in \mathcal{PC} is in \mathcal{PF} ? That is, if one can efficiently check the correctness of solutions with respect to some (polynomially-bounded) relation R , then is it necessarily the case that the search problem of R can be solved efficiently? In other words, if it is *easy to check* whether or not a given solution for a given instance is correct, then is it also *easy to find* a solution to a given instance?

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

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

2.1.2 The decision version: proving versus verifying

As we shall see in the sequel, the study of search problems (e.g., the \mathcal{PC} -vs- \mathcal{PF} Question) can be “reduced” to the study of decision problems. Since the latter

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

problems have a less cumbersome terminology, complexity theory tends to focus on them (and maintains its relevance to the study of search problems via the aforementioned reduction). Thus, the study of decision problems provides a convenient way for studying search problems. For example, the study of the complexity of deciding the satisfiability of Boolean formulae provides a convenient way for studying the complexity of finding satisfying assignments for such formulae.

We wish to stress, however, that decision problems are interesting and natural *per se* (i.e., beyond their role in the study of search problems). After all, some people do care about the truth, and so determining whether certain claims are true is a natural computational problem. Specifically, determining whether a given object (e.g., a Boolean formula) has some predetermined property (e.g., is satisfiable) constitutes an appealing computational problem. The P-vs-NP Question refers to the complexity of solving such problems for a wide and natural class of properties associated with the class NP. The latter class refers to properties that have “efficient proof systems” allowing for the verification of the claim that a given object has a predetermined property (i.e., is a member of a predetermined set). Jumping ahead, we mention that the P-vs-NP Question refers to the question of whether properties that have efficient proof systems can also be decided efficiently (without proofs). Let us clarify all these notions.

Properties of objects are modeled as subsets of the set of all possible objects (i.e., a property is associated with the set of objects having this property). For example, the property of being a prime is associated with the set of prime numbers, and the property of being connected (resp., having a Hamiltonian path) is associated with the set of connected (resp., Hamiltonian) graphs. Thus, we focus on deciding membership in sets (as in Definition 1.2). The standard formulation of the P-vs-NP Question refers to the questionable equality of two natural classes of decision problems, denoted P and NP (and defined in §2.1.2.1 and §2.1.2.2, respectively).

2.1.2.1 The class P as a natural class of decision problems

Needless to say, we are interested in the class of decision problems that are efficiently solvable. This class is traditionally denoted \mathcal{P} (standing for Polynomial-time). The following definition refers to the formulation of solving decision problems (provided in Definition 1.2).

Definition 2.4 (efficiently solvable decision problems):

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

As in Definition 2.2, we have defined a fundamental class of problems, which contains many natural problems (e.g., determining whether or not a given graph is connected), but we do not have a good understanding regarding its actual contents (i.e., we are unable to characterize many natural problems with respect to membership in this class). In fact, there are many natural decision problems that are

not known to reside in \mathcal{P} , and a natural class containing a host of such problems is presented next. This class of decision problems is denoted NP (for reasons that will become evident in Section 2.1.4).

2.1.2.2 The class NP and NP-proof systems

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

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

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

Let us consider a couple of examples before turning to the actual definition. For example, the set of Hamiltonian graphs has a verification procedure that, given a pair (G, P) , accepts if and only if P is a Hamiltonian path in the graph G . In this case P serves as a proof that G is Hamiltonian. Note that such proofs are relatively short (i.e., the path is actually shorter than the description of the graph) and are easy to verify. Needless to say, this proof system satisfies the

³**Advanced comment:** In continuation to Footnote 1, we note that in this chapter we consider *deterministic* (polynomial-time) verification procedures, and consequently the completeness and soundness conditions that we state here are error-less. In contrast, in Chapter 9, we will consider various types of probabilistic (polynomial-time) verification procedures as well as probabilistic completeness and soundness conditions. A common theme that underlies both treatments is that efficient verification is interpreted as meaning verification by a process that runs in time that is polynomial in the length of the assertion. In the current chapter, we use the equivalent formulation that considers the running time as a function of the total length of the assertion and the proof, but require that the latter has length that is polynomially bounded by the length of the assertion.

aforementioned completeness and soundness conditions. In the case of satisfiable Boolean formulae, given a formula ϕ and a truth assignment τ , the verification procedure instantiates ϕ (according to τ), and accepts if and only if simplifying the resulting Boolean expression yields the value **true**. In this case τ serves as a proof that ϕ is satisfiable, and the alleged proofs are indeed relatively short and easy to verify.

Definition 2.5 (efficiently verifiable proof systems):

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

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

(Such a string y is called an **NP-witness** for $x \in S$.)

2. **Soundness:** For every $x \notin S$ and every y , it holds that $V(x,y) = 0$.

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

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

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

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

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

Note that for any search problem R in \mathcal{PC} , the set of instances that have a solution with respect to R (i.e., the set $S_R \stackrel{\text{def}}{=} \{x : R(x) \neq \emptyset\}$) is in \mathcal{NP} . Specifically, for any $R \in \mathcal{PC}$, consider the verification procedure V such that $V(x,y) \stackrel{\text{def}}{=} 1$ if and only if $(x,y) \in R$, and note that the latter condition can be decided in $\text{poly}(|x|)$ -time. Thus, *any search problem in \mathcal{PC} can be viewed as a problem of searching*

for (efficiently verifiable) *proofs* (i.e., NP-witnesses for membership in the set of instances having solutions). On the other hand, any NP-proof system gives rise to a natural search problem in \mathcal{PC} ; that is, the problem of searching for a valid proof (i.e., an NP-witness) for the given instance (i.e., the verification procedure V yields the search problem that corresponds to $R = \{(x, y) : V(x, y) = 1\}$). Thus, $S \in \mathcal{NP}$ if and only if there exists $R \in \mathcal{PC}$ such that $S = \{x : R(x) \neq \emptyset\}$.

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

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

2.1.2.3 The P versus NP question in terms of decision problems

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

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

2.1.3 Equivalence of the two formulations

As hinted several times, *the two formulations of the P-vs-NP Questions are equivalent*. That is, every search problem having efficiently checkable solutions is solvable in polynomial time (i.e., $\mathcal{PC} \subseteq \mathcal{PF}$) if and only if membership in any set that has an NP-proof system can be decided in polynomial time (i.e., $\mathcal{NP} \subseteq \mathcal{P}$). Recalling that $\mathcal{P} \subseteq \mathcal{NP}$ (whereas \mathcal{PF} is not contained in \mathcal{PC} (Exercise 2.1)), we prove the following.

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

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

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

Reflection: The first part of the proof of Theorem 2.6 associates with each set S in \mathcal{NP} a natural relation R (in \mathcal{PC}). Specifically, R consists of all pairs (x, y) such that y is an NP-witness for membership of x in S . Thus, the search problem of R consists of finding such an NP-witness, when given x as input. Indeed, R is called the **witness relation** of S , and solving the search problem of R allows to decide membership in S . Thus, $R \in \mathcal{PC} \subseteq \mathcal{PF}$ implies $S \in \mathcal{P}$. In the second part of the proof, we associate with each $R \in \mathcal{PC}$ a set S'_R (in \mathcal{NP}), but S'_R is more

“expressive” than the set $S_R \stackrel{\text{def}}{=} \{x : \exists y \text{ s.t. } (x, y) \in R\}$ (which gives rise to R as its witness relation). Specifically, S'_R consists of strings that encode pairs (x, y') such that y' is a prefix of some string in $R(x) = \{y : (x, y) \in R\}$. The key observation is that deciding membership in S'_R allows to solve the search problem of R ; that is, $S'_R \in \mathcal{P}$ implies $R \in \mathcal{PF}$.

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

2.1.4 The traditional definition of NP

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

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

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

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

- A non-deterministic polynomial-time Turing machine is defined as one that makes a number of steps that is polynomial in the length of the input. Traditionally, \mathcal{NP} is defined as the class of sets that are accepted by some non-deterministic polynomial-time Turing machine.

We stress that Definition 2.7 refers to a fictitious model of computation. Specifically, Definition 2.7 makes no reference to the number (or fraction) of possible

computations of the machine (on a specific input) that yield a specific output.⁴ Definition 2.7 only refers to whether or not computations leading to a certain output exist (for a specific input). The question of what does the mere existence of such possible computations mean in terms of real-life is not addressed, because the model of a non-deterministic machine is not meant to provide a reasonable model of a real-life computer. The model is meant to capture something completely different (i.e., it is meant to provide an elegant definition of the class \mathcal{NP} , while relying on the fact that Definition 2.7 is equivalent to Definition 2.5).

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

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

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

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

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

⁴**Advanced comment:** In contrast, the definition of a probabilistic machine refers to this number (or, equivalently, to the probability that the machine produces a specific output, when the probability is essentially taken uniformly over all possible computations). Thus, a probabilistic machine refers to a natural model of computation that can be realized provided we can equip the machine with a source of randomness. For details, see Section 6.1.

2.1.5 In support of P different from NP

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

Immanuel Kant (1724–1804)

Kant talks on the importance of *both* philosophical considerations (referred to as “concepts”) and empirical considerations (referred to as “intuition”) to science (referred to as (sound) “knowledge”).

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

- *Philosophical considerations:* Both formulations of the P-vs-NP Question refer to natural questions about which we have strong intuition. The notion of solving a (search) problem seems to presume that, at least in some cases (if not in general), finding a solution is significantly harder than checking whether a presented solution is correct. This translates to $\mathcal{PC} \setminus \mathcal{PF} \neq \emptyset$. Likewise, the notion of a proof seems to presume that, at least in some cases (if not in general), the proof is useful in determining the validity of the assertion; that is, that deciding the validity of an assertion may be made significantly easier when provided with a proof. This translates to $\mathcal{P} \neq \mathcal{NP}$, which also implies that it is significantly harder to find proofs than to verify their correctness, which again coincides with the daily experience of researchers and students.
- *Empirical considerations:* The class NP (or rather \mathcal{PC}) contains thousands of different problems for which no efficient solving procedure is known. Many of these problems have arisen in vastly different disciplines, and were the subject of extensive research of numerous different communities of scientists and engineers. These essentially independent studies have all failed to provide efficient algorithms for solving these problems, a failure that is extremely hard to attribute to sheer coincidence or a stroke of bad luck.

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

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

2.1.6 Two technical comments regarding NP

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

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

2.2 Polynomial-time Reductions

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

Teaching note: We assume that many students have heard of reductions, but we fear that most have obtained a conceptually poor view of their fundamental nature. In particular, we fear that reductions are identified with the theory of NP-completeness, while reductions have numerous other important applications that have little to do with NP-completeness (or completeness with respect to some other class). Furthermore, we believe that it is important to show that natural search and optimization problems can be reduced to decision problems.

2.2.1 The general notion of a reduction

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

The notion of a general algorithmic reduction was discussed in §1.2.3.2 and §1.2.3.5. These reductions, called Turing-reductions (cf. §1.2.3.2) and modeled by oracle machines (cf. §1.2.3.5), made no reference to the time complexity of the main algorithm (i.e., the oracle machine). Here, we focus on efficient (i.e., polynomial-time) reductions, which are often called *Cook reductions*. That is, we consider oracle machines (as in Definition 1.11) that run in time polynomial in the length of their input. We stress that the running time of an oracle machine is the number of steps made during its computation, and that the oracle's reply on each query is obtained in a single step.

The key property of efficient reductions is that they allow for the transformation of efficient implementations of the subroutine into efficient implementations of the task reduced to it. That is, as we shall see, if one problem is Cook-reducible to another problem and the latter is polynomial-time solvable then so is the former.

The most popular case is that of reducing decision problems to decision problems, but we will also consider reducing search problems to search problems and reducing search problems to decision problems. Note that when reducing to a decision problem, the oracle is determined as the single valid solver of the decision problem (i.e., the function $f : \{0, 1\}^* \rightarrow \{0, 1\}$ solves the decision problem of membership in S if, for every x , it holds that $f(x) = 1$ if $x \in S$ and $f(x) = 0$ otherwise). In contrast, when reducing to a search problem, there may be many different valid solvers (i.e., each function f that satisfies $(x, f(x)) \in R$ for every $(x, y) \in R$ is a valid solver of the search problem of R). We capture both cases in the following definition.

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

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

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

mapping of x to x' ; that is, the standard definition of a Karp-reduction is actually as follows.

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

Thus, syntactically speaking, a Karp-reduction is not a Cook-reduction, but it trivially gives rise to one (i.e., on input x , the oracle machine makes query $f(x)$, and returns the oracle answer). Being slightly inaccurate but essentially correct, we shall say that Karp-reductions are special cases of Cook-reductions. Needless to say, Karp-reductions constitute a very restricted case of Cook-reductions. Still, this restricted case suffices for many applications (e.g., most importantly for the theory of NP-completeness (when developed for decision problems)), but not for reducing a search problem to a decision problem. Furthermore, whereas each decision problem is Cook-reducible to its complement, some decision problems are *not* Karp-reducible to their complement (see Exercises 2.7 and 2.33).

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

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

Indeed, the function f preserves the existence of solutions; that is, for any x , it holds that $R(x) \neq \emptyset$ if and only if $R'(f(x)) \neq \emptyset$. As for the second function (i.e., g), it maps any solution y' for the reduced instance $f(x)$ to a solution for the original instance x (where this mapping may also depend on x). It is natural to consider also a third function, which maps solutions for R to solutions for R' (see Exercise 2.28).

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

- We say that two problems are **computationally equivalent** if they are reducible to one another. This means that the two problems are essentially as hard (or as easy). Note that, for various complexity classes (e.g., \mathcal{NP} and \mathcal{PC}), computationally equivalent problems need not reside in the same class, since the reductions allowed here are Cook-reductions. For example, as we shall see in Section 2.2.3, there exist many natural $R \in \mathcal{PC}$ such that the search problem of R and the decision problem of $S_R = \{x : \exists y \text{ s.t. } (x, y) \in R\}$ are computationally equivalent, although (even syntactically) the two problems do not belong to the same class. Also, each decision problem is computationally equivalent to its complement, although the two problems may not belong to the same class (see Section 2.4.3).
- We say that a *class of problems, \mathcal{C} , is reducible to a problem Π' if every problem in \mathcal{C} , is reducible to Π'* . We say that the class \mathcal{C} is reducible to the class \mathcal{C}' if for every $\Pi \in \mathcal{C}$ there exists $\Pi' \in \mathcal{C}'$ such that Π is reducible to Π' . For example, recall that \mathcal{PC} is reducible to \mathcal{NP} .

The fact that we allow Cook-reductions is essential to various important connections between decision problems and other computational problems. Specifically, as shown in Section 2.2.2, a natural class of optimization problems is reducible to \mathcal{NP} . Recall that \mathcal{PC} is reducible to \mathcal{NP} (as shown implicitly in the proof of Theorem 2.6). Furthermore, as shown in Section 2.2.3, many natural search problems in \mathcal{PC} are reducible to a corresponding *natural* decision problem in \mathcal{NP} (rather than merely to some problem in \mathcal{NP}).

2.2.2 Reducing optimization problems to search problems

Many search problems refer to a set of potential solutions, per each problem instance, such that different solutions are assigned different “values” (resp., “costs”). In such a case, one may be interested in finding a solution that has value exceeding some threshold (resp., cost below some threshold), or (even better) finding a solution of maximum value (resp., minimum cost). For simplicity, let us focus on the case of a value that we wish to maximize. Still, there are two different objectives (i.e., exceeding a threshold and optimizing), giving rise to two different (auxiliary) search problems related to the same relation R . Specifically, for a binary relation R and a *value function* $f : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \mathbb{R}$, we consider two search problems.

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

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

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

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

actually referring to the search problem of the relation

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

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

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

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

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

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

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

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

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

2.2.3 Self-reducibility of search problems

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

Teaching note: Our usage of the term self-reducibility differs from the traditional one. Traditionally, a decision problem is called (downwards) 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). For further details, see Exercise 2.13.

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

Note that the search problem of R and the problem of deciding membership in S_R refer to the same instances: The search problem requires finding an adequate

solution (i.e., given x find $y \in R(x)$), whereas the decision problem refers to the question of whether such solutions exist (i.e., given x determine whether or not $R(x)$ is non-empty). Thus, S_R is really the “decision problem implicit in R ,” because it is a decision problem that one implicitly solves when solving the search problem of R . Indeed, for any R , *the decision problem of S_R is easily reducible to the search problem for R* (and if R is in \mathcal{PC} then S_R is in \mathcal{NP}).⁵ It follows that *if a search problem R is self-reducible then it is computationally equivalent to the decision problem S_R .*

Note that the general notion of a reduction (i.e., Cook-reduction) seems inherent to the notion of self-reducibility. This is the case not only due to syntactic considerations, but rather due to the following inherent reason. An oracle to any decision problem returns a single bit per invocation, while the intractability of a search problem in \mathcal{PC} must be due to lacking more than a “single bit of information” (see Exercise 2.10).

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

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

Proposition 2.14 (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} .

Proof: We present an oracle machine that solves the search problem of R_{SAT} by making oracle calls to SAT . Given a formula ϕ , we find a satisfying assignment to ϕ (in case such an assignment exists) as follows. First, we query SAT on ϕ itself, and return an indication that there is no solution if the oracle answer is 0 (indicating $\phi \notin \text{SAT}$). Otherwise, we let τ , initiated to the empty string, denote a prefix of a satisfying assignment of ϕ . We proceed in iterations, where in each iteration we

⁵For example, the reduction invokes the search oracle and answer 1 if and only if the oracle returns some string (rather than the “no solution” symbol).

extend τ by one bit. This is done as follows: First we derive a formula, denoted ϕ' , by setting the first $|\tau| + 1$ variables of ϕ according to the values $\tau 0$. We then query SAT on ϕ' (which means that we ask whether or not $\tau 0$ is a prefix of a satisfying assignment of ϕ). If the answer is positive then we set $\tau \leftarrow \tau 0$ else we set $\tau \leftarrow \tau 1$. This procedure relies on the fact that if τ is a prefix of a satisfying assignment of ϕ and $\tau 0$ is not a prefix of a satisfying assignment of ϕ then $\tau 1$ must be a prefix of a satisfying assignment of ϕ .

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

Reductions analogous to the one used in the proof of Proposition 2.14 can be presented also for other search problems (and not only for NP-complete ones). Two such examples are searching for a 3-coloring of a given graph and searching for an isomorphism between a given pair of graphs (where the first problem is known to be NP-complete and the second problem is believed not to be NP-complete). In both cases, the reduction of the search problem to a decision problem involves extending a prefix of a valid solution by making suitable queries in order to decide which extension to use. Note, however, that in these cases the process of getting rid of constants (representing partial solutions) is more involved. For example, in the case of Graph 3-Colorability (resp., Graph Isomorphism) we need to enforce a partial coloring of a given graph (resp., a partial isomorphism between a given pair of graphs); see Exercises 2.11 and 2.12, respectively.

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

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

⁶While the problem seems rather technical at the current setting (as it merely amounts to whether or not the definition of SAT allows Boolean constants in its instances), it is far from being so technical in other cases (see Exercises 2.11 and 2.12).

specific structure of SAT was used, allowing for a direct and natural transformation of instances of S'_R to instances of S_R .

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

For an arbitrary $R \in \mathcal{PC}$, deciding membership in S'_R is not necessarily reducible to deciding membership in S_R . Furthermore, deciding membership in S'_R is not necessarily reducible to the search problem of R . (See Exercises 2.14, 2.15, and 2.16.)

In general, self-reducibility is a property of the search problem and not of the decision problem implicit in it. Furthermore, under plausible assumptions (e.g., the intractability of factoring), there exists relations $R_1, R_2 \in \mathcal{PC}$ having the same implicit-decision problem (i.e., $\{x : R_1(x) \neq \emptyset\} = \{x : R_2(x) \neq \emptyset\}$) such that R_1 is self-reducible but R_2 is not (see Exercise 2.17). However, for many natural decision problems this phenomenon does not arise; that is, *for many natural NP-decision problems S , any NP-witness relation associated with S (i.e., $R \in \mathcal{PC}$ such that $\{x : R(x) \neq \emptyset\} = S$) is self-reducible.* Indeed, see the discussion following the proof of Proposition 2.14 as well as the subsequent comment.

Teaching note: Towards the following advanced comment, we assume that the students have heard of NP-completeness. Actually, we only need the students to know the definition of NP-completeness (i.e., a set S is \mathcal{NP} -complete if $S \in \mathcal{NP}$ and every set in \mathcal{NP} is reducible to S). Yet, the teacher may prefer postponing the presentation of the following advanced comment to Section 2.3.1 (or even to a later stage).

Advanced comment: Recall that, in general, self-reducibility is a property of the search problem R and not of the decision problem implicit in it (i.e., $S_R = \{x : R(x) \neq \emptyset\}$). In contrast, in the special case of NP-complete problems, self-reducibility holds for any witness relation associated with the (NP-complete) decision problem. That is, *all search problems that refer to finding NP-witnesses for any NP-complete decision problem are self-reducible.*

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

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

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

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

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

2. A reduction of S'_R to S_R .

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

The theorem follows. ■

2.3 NP-Completeness

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

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

2.3.1 Definitions

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

1. Π is in the class (i.e., Π being in \mathcal{NP} or \mathcal{PC} , depending on whether Π is a decision or a search problem).
2. Each problem in the class is reducible to Π . This condition is called NP-hardness.

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

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

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

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

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

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

2.3.2 The existence of NP-complete problems

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

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

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

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

Definition: The relation $R_{\mathbf{u}}$ consists of pairs $\langle \langle M, x, 1^t \rangle, y \rangle$ such that M accepts the input pair (x, y) within t steps, where $|y| \leq t$.⁷ The corresponding set $S_{\mathbf{u}} \stackrel{\text{def}}{=} \{ \bar{x} : \exists y \text{ s.t. } (\bar{x}, y) \in R_{\mathbf{u}} \}$ consists of triples $\langle M, x, 1^t \rangle$ such that machine M accepts some input of the form (x, \cdot) within t steps.

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

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

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

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

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

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

⁷Instead of requiring that $|y| \leq t$, one may require that M is “canonical” in the sense that it reads its entire input before halting.

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

Bounded Halting and Non-Halting

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

The decision version of **Bounded Non-Halting** refers to a fundamental computational problem in the area of program verification; specifically, the question of whether a given program halts within a given time-bound on all inputs of a given length.⁸ We mention the **Bounded Halting** problem because it is often referred to in the literature, but we believe that **Bounded Non-Halting** is more relevant to the project of program verification (because one seeks programs that halt on all inputs rather than programs that halt on some input).

It is easy to prove that both problems are NP-complete (see Exercise 2.19). Note that the two (decision) problems are not complementary (i.e., $(M, 1^t)$ may be a yes-instance of both decision problems).⁹

⁸The length parameter need not equal the time-bound. Indeed, a more general version of the problem refers to two bounds, ℓ and t , and to whether the given program halts within t steps on each possible ℓ -bit input. It is easy to prove that the problem remains NP-complete also in the case that the instances are restricted to have parameters ℓ and t such that $t = p(\ell)$, for any fixed polynomial p (e.g., $p(n) = n^2$, rather than $p(n) = n$ as used in the main text).

⁹Indeed, $(M, 1^t)$ can not be a no-instance of both decision problems, but this does not make the problems complementary. In fact, the two decision problems yield a three-way partition of the instances $(M, 1^t)$: (1) pairs $(M, 1^t)$ such that for *every input* x (of length at most t) the computation of $M(x)$ halts within t steps, (2) pairs $(M, 1^t)$ for which such halting occurs on *some inputs but not on all inputs*, and (3) pairs $(M, 1^t)$ such that there *exists no input* (of length at most t) on which M halts in t steps. Note that instances of type (1) are no-instances of **Bounded Non-Halting**, whereas instances of type (3) are no-instances of **Bounded Halting**. It follows that recognizing each of these three sets of instances is NP-hard under Cook-reductions.

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

In light of the above, the common practice of bashing Bounded (Non-)Halting as an “unnatural” problem seems very odd at an age in which computer programs plays such a central role. (Nevertheless, we will use the term “natural” in this traditionally and odd sense in the next title, which refers to natural computational problems that seem unrelated to computation.)

2.3.3 Some natural NP-complete problems

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

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

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

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

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

2.3.3.1 Circuit and formula satisfiability: CSAT and SAT

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

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

CSAT. Recall that Boolean circuits are directed acyclic graphs with internal vertices, called **gates**, labeled by Boolean operations (of arity either 2 or 1), and external vertices called **terminals** that are associated with either inputs or outputs. When setting the inputs of such a circuit, all internal nodes are assigned values in the natural way, and this yields a value to the output(s), called an evaluation of the circuit on the given input. The evaluation of circuit C on input z is denoted $C(z)$. We focus on circuits with a single output, and let CSAT denote the set of satisfiable Boolean circuits (i.e., a circuit C is in CSAT if there exists an input z such that $C(z) = 1$). We also consider the related relation $R_{\text{CSAT}} = \{(C, z) : C(z) = 1\}$.

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

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

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

that corresponds to any $R \in \mathcal{PC}$ (as in the proof of Theorem 2.18), we establish the fact that CSAT is NP-complete. Details follow.

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

Before describing the circuit C_x , let us consider a possible computation of M_R on input (x, y) , where x is fixed and y represents a generic string of length at most $p_R(|x|)$. Such a computation proceeds for $t = t_R(|x| + p_R(|x|))$ steps, and corresponds to a sequence of $t + 1$ instantaneous configurations, each of length t . Each such configuration can be encoded by t pairs of symbols, where the first symbol in each pair indicates the contents of a cell and the second symbol indicates either a state of the machine or the fact that the machine is not located in this cell. Thus, each pair is a member of $\Sigma \times (Q \cup \{\perp\})$, where Σ is the finite “work alphabet” of M_R , Q is its finite set of internal states, and \perp is an indication that the machine is not present at a cell. The initial configuration includes xy as input, and the decision of $M_R(x, y)$ can be read from (the leftmost cell of) the last configuration.¹⁰ With the exception of the first row, the values of the entries in each row are determined by the entries of the row just above it, where this determination reflects the transition function of M_R . Furthermore, the value of each entry in the said array is determined by the values of (up to) three entries that reside in the row above it (see Exercise 2.20). Thus, the aforementioned computation is represented by a $(t + 1) \times t$ array, where each entry encodes one out of a constant number of possibilities, which in turn can be encoded by a constant-length bit string. See Figure 2.1.

The circuit C_x has a structure that corresponds to the aforementioned array. Each entry in the array is represented by a *constant* number of gates such that when C_x is evaluated at y these gates will be assigned values that encode the contents of the said entry. In particular, the entries of the first row of the array are “encoded” by hard-wiring the reduction’s input (i.e., x), and feeding the circuit’s input (i.e., y) to the adequate input terminals. That is, the circuit has $p_R(|x|)$ (“real”) input terminals, and the hard-wiring of constants to the other $O(t - p_R(|x|))$ gates that represent the first row is done by simple gadgets (as in Figure 1.3). Indeed, additional hard-wiring in the first row corresponds to the other fixed elements of the initial configuration (i.e., the blank symbols, and the encoding of the initial state and of the initial location; cf. Figure 2.1). The entries of subsequent rows

¹⁰We refer to the output convention presented in §1.2.3.1, by which the output is written in the leftmost cells and the machine halts at the cell to its right.

(1,a)	(1,-)	(0,-)	(y _{1,-})	(y _{2,-})	(-, -)	(-, -)	(-, -)	(-, -)	(-, -)	initial configuration (with input 110y ₁ y ₂)
(3,-)	(1,b)	(0,-)	(y _{1,-})	(y _{2,-})	(-, -)	(-, -)	(-, -)	(-, -)	(-, -)	
(3,-)	(1,-)	(0,b)	(y _{1,-})	(y _{2,-})	(-, -)	(-, -)	(-, -)	(-, -)	(-, -)	
(3,-)	(1,c)	(0,-)								
(3,c)	(1,-)	(0,-)								
(1,-)	(1,f)	(0,-)								

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

Figure 2.1: An array representing ten computation steps on input 110y₁y₂.

will be “encoded” (or rather computed at evaluation time) by using *constant-size* circuits that determine the value of an entry based on the three relevant entries in the row above it. Recall that each entry is encoded by a constant number of gates, and thus these constant-size circuits merely compute the constant-size function described in Exercise 2.20. In addition, the circuit will have a few extra gates that check the values of the entries of the last row in order to determine whether or not it encodes an accepting configuration.¹¹ Note that the circuit C_x can be constructed in polynomial time from the string x , because we just need to encode x in an appropriate manner as well as generate a “highly uniform” grid-like circuit of size $O(t_R(|x| + p_R(|x|))^2)$.¹²

Although the foregoing construction of C_x capitalizes on various specific details of the (one-tape) Turing machine model, it can be adapted to any other “rea-

¹¹In continuation to Footnote 10, we note that it suffices to check the values of the two leftmost entries of the last row. We assumed here that the circuit propagates a halting configuration to the last row. Alternatively, we may check for the existence of an accepting/halting configuration in the entire array, since this condition is quite simple.

¹²**Advanced comment:** A more efficient construction, which generate almost-linear sized circuits (i.e., circuits of size $\tilde{O}(t_R(|x| + p_R(|x|)))$) is known; see [173].

sonable” model of efficient computation.¹³ Alternatively, we recall the Cobham-Edmonds Thesis asserting that any problem that is solvable in polynomial-time (on some “reasonable” model) can be solved in polynomial-time by a (one-tape) Turing machine.

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

SAT. Recall that Boolean formulae are special types of Boolean circuits (i.e., circuits having a tree structure).¹⁴ We further restrict our attention to formulae given in conjunctive normal form (CNF). We denote by SAT the set of satisfiable CNF formulae (i.e., a CNF formula ϕ is in SAT if there exists a truth assignment τ such that $\phi(\tau) = 1$). We also consider the related relation $R_{\text{SAT}} = \{(\phi, \tau) : \phi(\tau) = 1\}$.

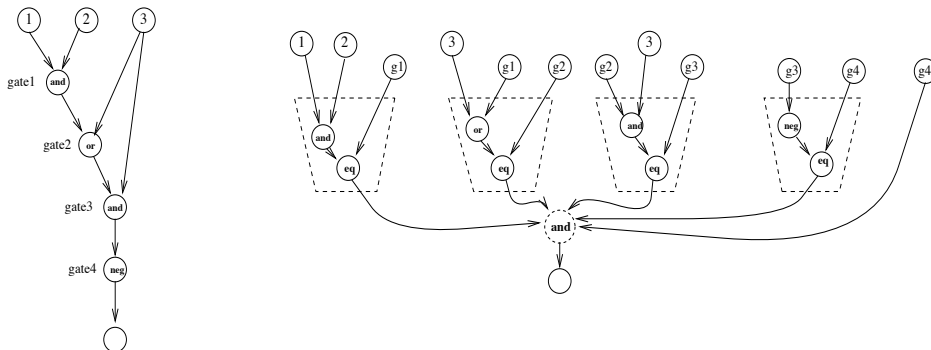
Theorem 2.21 (NP-completeness of SAT): *The set (resp., relation) SAT (resp., R_{SAT}) is \mathcal{NP} -complete (resp., \mathcal{PC} -complete).*

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

We start by Karp-reducing CSAT to SAT . Given a Boolean circuit C , with n input terminals and m gates, we first construct m *constant-size* formulae on $n + m$ variables, where the first n variables correspond to the input terminals of the circuit, and the other m variables correspond to its gates. The i^{th} formula will depend on the variable that correspond to the i^{th} gate and the 1-2 variables that correspond to

¹³**Advanced comment:** Note that it is actually inessential that each entry in each configuration is determined by a constant number of entries in the previous configuration. Any polynomial-time computable transformation of configurations will do, since we can emulate such a transformation by a polynomial-size circuit. Indeed, this emulation will be based on presenting the said transformation in some concrete model of computation, which brings us to the next comment (invoking the Cobham-Edmonds Thesis).

¹⁴For an alternative definition, see Section G.2.



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

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

the vertices that feed into this gate (i.e., 2 vertices in case of \wedge -gate or \vee -gate and a single vertex in case of a \neg -gate, where these vertices may be either input terminals or other gates). This (constant-size) formula will be satisfied by a truth assignment if and only if this assignment matches the gate’s functionality (i.e., feeding this gate with the corresponding values result in the corresponding output value). Note that these *constant-size* formulae can be written as constant-size CNF formulae (in fact, as 3CNF formulae).¹⁵ Taking the conjunction of these m formulae as well as the variable associated with the gate that feeds into the output terminal, we obtain a formula ϕ in CNF (see Figure 2.2, where $n = 3$ and $m = 4$).

Note that ϕ can be constructed in polynomial-time from the circuit C ; that is, the mapping of C to $\phi = f(C)$ is polynomial-time computable. We claim that C is in CSAT if and only if ϕ is in SAT.

1. Suppose that for some string s it holds that $C(s) = 1$. Then, assigning the i^{th} auxiliary variable the value that is assigned to the i^{th} gate of C when evaluated on s , we obtain (together with s) a truth assignment that satisfies ϕ . This is the case because such an assignment satisfies all m constant-size CNFs as well as the variable associated with the output of C .
2. On the other hand, if τ satisfies ϕ then the first n bits in τ correspond to an input on which C evaluates to 1. This is the case because the m constant-size CNFs guarantee that the variables of ϕ are assigned values that correspond

¹⁵Recall that any Boolean function can be written as a CNF formula having size that is exponential in the length of its input, which in this case is a constant (i.e., either 2 or 3). Furthermore, note that the Boolean functions that we refer to here depends on 2-3 Boolean variables (since they indicate whether or not the corresponding values respect the gate’s functionality).

to the evaluation of C on the first n bits of τ , while the fact that the variable associated with the output of C has value `true` guarantees that this evaluation of C yields the value 1.

Note that the latter mapping (of τ to its n -bit prefix) is the second mapping required by the definition of a Levin-reduction.

Thus, we have established that f is a Karp-reduction of CSAT to SAT, and that augmenting f with the aforementioned second mapping yields a Levin-reduction of R_{CSAT} to R_{SAT} . ■

Comment. The fact that the second mapping required by the definition of a Levin-reduction is explicit in the proof of the validity of the corresponding Karp-reduction is a fairly common phenomenon. Actually (see Exercise 2.28), typical presentations of Karp-reductions provide two auxiliary polynomial-time computable mappings (in addition to the main mapping of instances from one problem (e.g., CSAT) to instances of another problem (e.g., SAT)): The first auxiliary mapping is of solutions for the preimage instance (e.g., of CSAT) to solutions for the image instance of the reduction (e.g., of SAT), whereas the second mapping goes the other way around. (Note that only the main mapping and the second auxiliary mapping are required in the definition of a Levin-reduction.) For example, the proof of the validity of the Karp-reduction of CSAT to SAT, denoted f , specified two additional mappings h and g such that $(C, s) \in R_{\text{CSAT}}$ implies $(f(C), h(C, s)) \in R_{\text{SAT}}$ and $(f(C), \tau) \in R_{\text{SAT}}$ implies $(C, g(C, \tau)) \in R_{\text{CSAT}}$. Specifically, in the proof of Theorem 2.21, we used $h(C, s) = (s, a_1, \dots, a_m)$ where a_i is the value assigned to the i^{th} gate in the evaluation of $C(s)$, and $g(C, \tau)$ being the n -bit prefix of τ .

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

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

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

Related problems. Note that instances of SAT can be viewed as systems of Boolean conditions over Boolean variables. Such systems can be emulated by various types of systems of arithmetic conditions, implying the NP-hardness of solving the latter types of systems. Examples include systems of *integer* linear inequalities (see Exercise 2.23), and systems of quadratic equalities (see Exercise 2.25).

2.3.3.2 Combinatorics and graph theory

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

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

We start with the **set cover** problem, in which an instance consists of a collection of finite sets S_1, \dots, S_m and an integer K and the question (for decision) is whether or not there exist (at most)¹⁶ K sets that cover $\bigcup_{i=1}^m S_i$ (i.e., indices i_1, \dots, i_K such that $\bigcup_{j=1}^K S_{i_j} = \bigcup_{i=1}^m S_i$).

Proposition 2.23 *Set Cover is NP-complete.*

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

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

¹⁶Clearly, in case of **Set Cover**, the two formulations (i.e., asking for exactly K sets or at most K sets) are computationally equivalent.

which in turn implies that $\tau_1 \cdots \tau_n$ satisfies ϕ . Thus, if $f(\phi)$ is a yes-instance of **Set Cover** then $\phi \in \text{SAT}$. \square

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

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

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

Proof Sketch: The reduction is obtained by composing three reductions. We first reduce a *restricted case* of **3SAT** to a restricted version of **Set Cover**, denoted **3SC**, in which each set has at most three elements (and an instance consists, as in the general case, of a sequence of finite sets as well as an integer K). Specifically, we refer to **3SAT** instances that are restricted such that each *variable* appears in at most three clauses, and recall that this restricted problem is NP-complete (see Proposition 2.22). Actually, we further reduce this special case of **3SAT** to one in which each *literal* appears in at most two clauses.¹⁷ Now, we reduce the new version of **3SAT** to **3SC** by using the (very same) reduction presented in the proof of Proposition 2.23, and observing that the size of each set in the reduced instance is at most three (i.e., one more than the number of occurrences of the corresponding literal).

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

¹⁷This can be done by observing that if all three occurrences of a variable are of the same type (i.e., they are all negated or all non-negated) then this variable can be assigned a value that satisfies all clauses in which it appears, and so the variable and the clauses in which it appear can be omitted from the instance. This yields a reduction of **3SAT** instances in which each variable appears in at most three clauses to **3SAT** instances in which each literal appears in at most two clauses. Actually, a closer look at the proof of Proposition 2.22 reveals the fact that the reduced instances satisfy the latter property anyhow.

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

Vertex Cover, Independent Set, and Clique. Turning to graph theoretic problems (see Section G.1), we start with the **Vertex Cover** problem, which is a special case of the **Set Cover** problem. The instances consists of pairs (G, K) , where $G = (V, E)$ is a simple graph and K is an integer, and the problem is whether or not there exists a set of (at most) K vertices that is incident to all graph edges (i.e., each edge in G has at least one endpoint in this set). Indeed, this instance of **Vertex Cover** can be viewed as an instance of **Set Cover** by considering the collection of sets $(S_v)_{v \in V}$, where S_v denotes the set of edges incident at vertex v (i.e., $S_v = \{e \in E : v \in e\}$). Thus, the NP-hardness of **Set Cover** follows from the NP-hardness of **Vertex Cover** (but this implication is unhelpful for us here: we already know that **Set Cover** is NP-hard and we wish to prove that **Vertex Cover** is NP-hard). We also note that the **Vertex Cover** problem is computationally equivalent to the **Independent Set** and **Clique** problems (see Exercise 2.26), and thus it suffices to establish the NP-hardness of one of these problems.

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

Teaching note: The following reduction is not the “standard” one (see Exercise 2.27). It is rather adapted from the FGLSS-reduction (see Exercise 9.14), and is used here in anticipation of the latter. Furthermore, although the following reduction tends to create a larger graph, the author finds it more clear than the “standard” reduction.

Proof Sketch: We show a reduction from 3SAT to **Independent Set**. On input a 3CNF formula ϕ with m clauses and n variables, we construct a graph with $7m$ vertices, denoted G_ϕ . The vertices are grouped in m cliques, each corresponding to one of the clauses and containing 7 vertices that correspond to the 7 truth assignments (to the 3 variables in the clause) that *satisfy the clause*. In addition to the internal edges of these m cliques, we add an edge between each pair of vertices that correspond to partial assignments that are *mutually inconsistent*. That is, if a specific (satisfying) assignment to the variables of the i^{th} clause is inconsistent with some (satisfying) assignment to the variables of the j^{th} clause then we connect the

corresponding vertices by an edge. (Note that the internal edges of the m cliques may be viewed as a special case of the edges connecting mutually inconsistent partial assignments.) Thus, on input ϕ , the reduction outputs the pair (G_ϕ, m) .

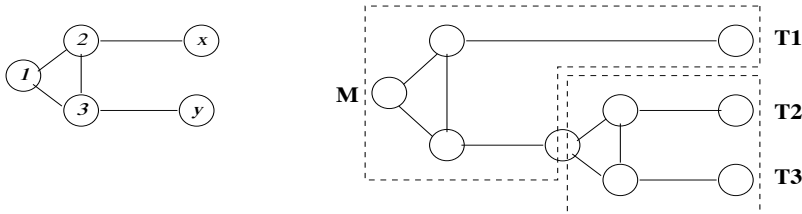
Note that if ϕ is satisfiable by a truth assignment τ then there are no edges between the m vertices that correspond to the partial satisfying assignment derived from τ . (We stress that any truth assignment to ϕ yields an independent set, but only a satisfying assignment guarantees that this independent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

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

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

Proof Sketch: We reduce 3SAT to G3C by mapping a 3CNF formula ϕ to the graph G_ϕ , which consists of two special (“designated”) vertices, a gadget per each variable of ϕ , a gadget per each clause of ϕ , and edges connecting some of these components.

- The two designated vertices are called **ground** and **false**, and are connected by an edge that ensures that they must be given different colors in any 3-coloring of G_ϕ . We will refer to the color assigned to the vertex **ground** (resp., **false**) by the name **ground** (resp., **false**). The third color will be called **true**.
- The gadget associated with variable x is a pair of vertices, associated with the two literals of x (i.e., x and $\neg x$). These vertices are connected by an edge, and each of them is also connected to the vertex **ground**. Thus, in a 3-coloring of G_ϕ one of the vertices associated with the variable is colored **true** and the other is colored **false**.
- The gadget associated with a clause C is depicted in Figure 2.3. It contains a **master vertex**, denoted **M**, and three **terminal vertices**, denoted **T1**, **T2** and **T3**. The master vertex is connected by edges to the vertices **ground** and **false**, and thus in a 3-coloring of G_ϕ the master vertex must be colored **true**. The gadget has the property that it is possible to color the terminals with any combination of the colors **true** and **false**, except for coloring all terminals with **false**. The terminals of the gadget associated with clause C will be *identified* with the vertices that are associated with the corresponding



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

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

literals appearing in C . This means that the various clause-gadgets are not vertex-disjoint but may rather share some terminals (with the vertex-gadgets as well as among themselves).¹⁸ See Figure 2.4.

Verifying the validity of the reduction is left as an exercise. \square

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

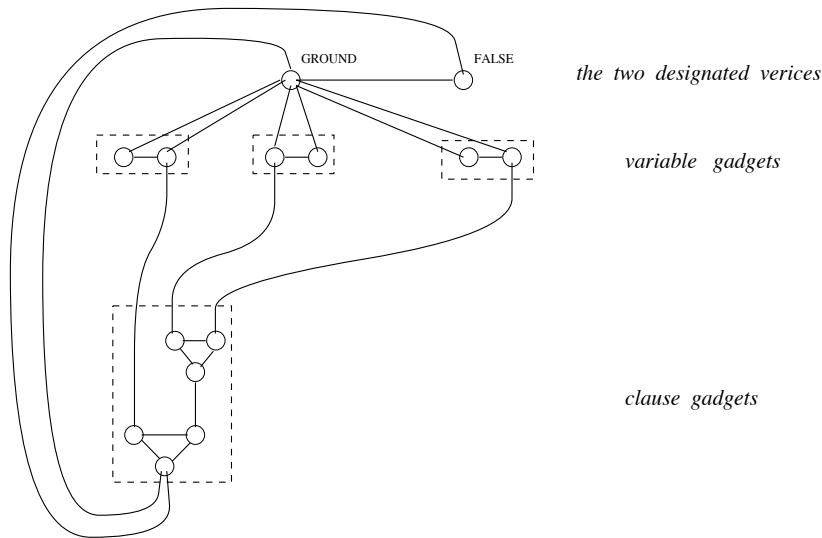
As stated in Section 2.3.3, thousands of problems have been shown to be NP-complete (cf., [81, Apdx.], which contains a list of more than three hundreds main entries). Things reached a situation in which people seem to expect any NP-set to be either NP-complete or in \mathcal{P} . This naive view is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

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

Theorem 2.27 asserts that if $\mathcal{NP} \neq \mathcal{P}$ then \mathcal{NP} is partitioned into three non-empty classes: the class \mathcal{P} , the class of problems to which \mathcal{NP} is Cook-reducible, and the rest, denote $\mathcal{NP}\mathcal{I}$. We already know that the first two classes are not empty, and Theorem 2.27 establishes the non-emptiness of $\mathcal{NP}\mathcal{I}$ under the condition that $\mathcal{NP} \neq \mathcal{P}$, which is actually a necessary condition (because if $\mathcal{NP} = \mathcal{P}$ then every set in \mathcal{NP} is Cook-reducible to any other set in \mathcal{NP}).

The following proof of Theorem 2.27 presents an unnatural decision problem in $\mathcal{NP}\mathcal{I}$. We mention that some natural problems (e.g., factoring) are conjectured to be neither solvable in polynomial-time nor NP-hard. In particular, assuming that factoring is intractable, there exist rather natural decision problems in $\mathcal{NP}\mathcal{I}$. Furthermore, if $\mathcal{NP} \neq \text{co}\mathcal{NP}$, where $\text{co}\mathcal{NP} = \{\{0, 1\}^* \setminus S : S \in \mathcal{NP}\}$, then $\Delta \stackrel{\text{def}}{=} \underline{\hspace{10em}}$

¹⁸Alternatively, we may use disjoint gadgets and “connect” each terminal with the corresponding literal (in the corresponding vertex gadget). Such a connection (i.e., an auxiliary gadget) should force the two end-points to have the same color in any 3-coloring of the graph.



A single clause gadget and the relevant variables gadgets.

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

$\mathcal{NP} \cap \text{co}\mathcal{NP} \subseteq \mathcal{P} \cup \mathcal{NPI}$ holds (as a corollary to Theorem 2.33). In other words, if $\mathcal{NP} \neq \text{co}\mathcal{NP}$ then $\Delta \setminus \mathcal{P}$ is a (natural) subset of \mathcal{NPI} , and the non-emptiness of \mathcal{NPI} follows provided that $\Delta \neq \mathcal{P}$. Recall that Theorem 2.27 establishes the non-emptiness of \mathcal{NPI} under the seemingly weaker assumption that $\mathcal{NP} \neq \mathcal{P}$.

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

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

1. Any polynomial-time reduction (of any set not in \mathcal{P}) to any finite set (e.g., a finite subset of S) must fail, because only sets in \mathcal{P} are Cook-reducible

to a finite set. Thus, for any finite set F and any potential reduction (i.e., a polynomial-time oracle machine), there exists an input x on which this reduction to F fails.

We stress that the aforementioned reduction fails while the only queries that are answered positively are those residing in F . Furthermore, the aforementioned failure is due to a finite set of queries (i.e., the set of all queries made by the reduction when invoked on an input that is smaller or equal to x). Thus, for every finite set $F \subset S' \subseteq S$, any reduction of S to S' can be failed by dropping a finite number of elements from S' and without dropping elements of F .

2. For every finite set F , any polynomial-time decision procedure for $S \setminus F$ must fail, because S is (trivially) Cook-reducible to $S \setminus F$. Thus, for any potential decision procedure (i.e., a polynomial-time algorithm), there exists an input x on which this procedure fails.

We stress that this failure is due to a finite “prefix” of $S \setminus F$ (i.e., the set $\{z \in S \setminus F : z \leq x\}$). Thus, for every finite set F , any polynomial-time decision procedure for $S \setminus F$ can be failed by keeping a finite subset of $S \setminus F$.

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

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

Next, let us clarify the process in which reductions and decision procedures are ruled out. We present a construction of a “filter” set F in \mathcal{P} such that the final set S' will equal $S \cap F$. Recall that we need to select F such that each polynomial-time reduction of S to $S \cap F$ fails, and each polynomial-time procedure for deciding $S \cap F$ fails. The key observation is that for every finite F each polynomial-time reduction of S to $S \cap F$ fails, whereas for every co-finite F (i.e., finite $\{0, 1\}^* \setminus F$) each polynomial-time procedure for deciding $S \cap F$ fails. Furthermore, each of these failures occur on some input, and this input is determined by finite portions

of S and F . Thus, we alternate between failing possible reductions and decision procedures, while not trying to determine the “optimal” points of alternation but rather determining points of alternation in a way that allows for efficiently deciding membership in F . Specifically, we let $F = \{x : f(|x|) \equiv 1 \pmod{2}\}$, where $f : \mathbb{N} \rightarrow \{0\} \cup \mathbb{N}$ is defined next such that $f(n)$ can be computed in time $\text{poly}(n)$.

The value of $f(n)$ is defined by the the following experiment that consists of exactly n^3 computation steps (where cubic time is selected to allow for some non-trivial manipulations of data as conducted next). For $i = 0, 1, \dots$, we scan all inputs in lexicographic order trying to find an input on which the $i+1^{\text{st}}$ (modified) machine fails, where this machine is an oracle machine if $i+1$ is odd and a standard machine otherwise. In order to determine whether or not a failure occurs on a particular input x , we may need to know whether or not x is in the set $S' = S \cap F$ as well as whether some other strings (which may appear as queries) are in S' . Deciding membership in $S \in \mathcal{NP}$ can be done in exponential-time (by using the exhaustive search algorithm that tries all possible NP-witnesses). Indeed, this means that when computing $f(n)$ we may only complete the treatment of inputs that are of logarithmic (in n) length, but anyhow in n^3 steps we can not hope to reach (in our lexicographic scanning) strings of length $3 \log_2 n$. As for deciding membership in F , this requires ability to compute f on adequate integers. That is, we may need to compute the value of $f(n')$ for various integers n' , but as noted n' will be much smaller than n . Thus, the value of $f(n')$ is just computed recursively (while counting the recursive steps in our total number of steps).¹⁹ The point is that, when considering an input x , we may need the values of f only on $\{1, \dots, p_{i+1}(|x|)\}$, where p_{i+1} is the polynomial bounding the running-time of the $i+1^{\text{st}}$ (modified) machine, and obtaining such a value takes at most $p_{i+1}(|x|)^3$ steps. Finally, if we detect a failure of the $i+1^{\text{st}}$ machine, then we increase i and proceed to the next iteration. When we reach the allowed number of steps (i.e., n^3 steps), we halt outputting the current value of i (i.e., the current i is output as the value of $f(n)$).

As hinted in the foregoing, it is most likely that we will complete n^3 steps much before examining inputs of length $3 \log_2 n$, but this does not matter. What matters is that f is *monotonically non-decreasing* (because more steps allow to fail at least as many machines) and that f is *unbounded* (see Exercise 2.34). Furthermore, by construction, $f(n)$ is computed in $\text{poly}(n)$ time. \square

Comment: The proof of Theorem 2.27 actually establishes that *for every $S \notin \mathcal{P}$ there exists $S' \notin \mathcal{P}$ such that S' is Karp-reducible to S but S is not Cook-reducible to S'* .²⁰ Thus, if $\mathcal{P} \neq \mathcal{NP}$ then there exists an infinite sequence of sets S_1, S_2, \dots in $\mathcal{NP} \setminus \mathcal{P}$ such that S_{i+1} is Karp-reducible to S_i but S_i is not Cook-reducible to S_{i+1} . That is, there exists an infinite hierarchy of problems (albeit unnatural ones), all in \mathcal{NP} , such that each problem is “easier” than the previous ones (in the sense that it can be reduced to the previous problems while these problems cannot

¹⁹We do not bother to present a more efficient implementation of this process. That is, we may afford to recompute $f(n')$ every time we need it (rather than store it for later use).

²⁰The said Karp-reduction (of S' to S) maps x to itself if $x \in F$ and otherwise maps x to a fixed no-instance of S .

be reduced to it).

2.4 Three relatively advanced topics

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

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

2.4.1 Promise Problems

Promise problems are a natural generalization of search and decision problems, where one explicitly considers a set of legitimate instances (rather than considering any string as a legitimate instance). As noted before, this provides a more adequate formulation of natural computational problems (and indeed this formulation is used in all informal discussions). For example, in §2.3.3.2 we presented such problems using phrases like “given a graph and an integer...” (or “given a collection of sets...”). In other words, we assumed that the input instance has a certain format (or rather we “promised the solver” that this is the case). Indeed, we claimed that in these cases the assumption can be removed without affecting the complexity of the problem, but we avoided providing a formal treatment of this issue, which is done next.

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

2.4.1.1 Definitions

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

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

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

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

- The search problem R with promise P is in the **promise problem extension** of \mathcal{PF} if there exists a polynomial-time algorithm that solves this problem.²¹
- The search problem R with promise P is in the **promise problem extension** of \mathcal{PC} if there exists a polynomial T and an algorithm A such that, for every $x \in P$ and $y \in \{0,1\}^*$, algorithm A makes at most $T(|x|)$ steps and it holds that $A(x,y) = 1$ if and only if $(x,y) \in R$.

We stress that nothing is required of the solver in the case that the input violates the promise (i.e., $x \notin P$); in particular, in such a case the algorithm may halt with a wrong output. (Indeed, the standard formulation of search problems is obtained by considering the trivial promise $P = \{0,1\}^*$.)²² In addition to the foregoing motivation for promise problems, we mention one natural class of search problems with a promise. These are search problem in which the promise is that the instance has a solution (i.e., in terms of the foregoing notation $P = S_R$). We refer to such search problems by the name candid search problems.

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

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

Decision problems with a promise. In the context of decision problems, a promise problem is a relaxation in which one is only required to determine the status of instances that belong to a predetermined set, called the **promise**. The requirement of efficient verification is adapted in an analogous manner. In view of the standard usage of the term, we refer to *decision problems with a promise* by the name *promise problems*. Formally, promise problems refer to a three-way partition of the set of all strings into yes-instances, no-instances and instances that

²¹In this case it does not matter whether the time complexity of A is defined on inputs in P or on all possible strings. Suppose that A has (polynomial) time complexity T on inputs in P , then we can modify A to halt on any input x after at most $T(|x|)$ steps. This modification may only effects the output of A on inputs not in P (which is OK by us). The modification can be implemented in polynomial-time by computing $t = T(|x|)$ and emulating the execution of $A(x)$ for t steps. A similar comment applies to the definition of \mathcal{PC} , \mathcal{P} and \mathcal{NP} .

²²Here we refer to the formulation presented in Section 2.1.6.

violate the promise. Standard decision problems are obtained as a special case by insisting that all inputs are allowed (i.e., the promise is trivial).

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

- *The promise problem $(S_{\text{yes}}, S_{\text{no}})$ is solved by algorithm A if for every $x \in S_{\text{yes}}$ it holds that $A(x) = 1$ and for every $x \in S_{\text{no}}$ it holds that $A(x) = 0$. The promise problem is in the promise problem extension of \mathcal{P} if there exists a polynomial-time algorithm that solves it.*
- *The promise problem $(S_{\text{yes}}, S_{\text{no}})$ is in the promise problem extension of \mathcal{NP} if there exists a polynomial p and a polynomial-time algorithm V such that the following two conditions hold:*
 1. *Completeness: For every $x \in S_{\text{yes}}$, there exists y of length at most $p(|x|)$ such that $V(x, y) = 1$.*
 2. *Soundness: For every $x \in S_{\text{no}}$ and every y , it holds that $V(x, y) = 0$.*

We stress that for algorithms of polynomial-time complexity, it does not matter whether the time complexity is defined only on inputs that satisfy the promise or on all strings (see Footnote 21). Thus, the extended classes \mathcal{P} and \mathcal{NP} (like \mathcal{PF} and \mathcal{PC}) are invariant under this choice.

Reducibility among promise problems. The notion of a Cook-reduction extend naturally to promise problems, when postulating that a query that violates the promise (of the problem at the target of the reduction) may be answered arbitrarily.²³ That is, the oracle machine should solve the original problem no matter how queries that violate the promise are answered. The latter requirement is consistent with the conceptual meaning of reductions and promise problems. Recall that reductions captures procedures that make subroutine calls to an arbitrary procedure that solves the reduced problem. But, in the case of promise problems, such a solver may behave arbitrarily on instances that violate the promise. We stress that the main property of a reduction is preserved (see Exercise 2.35): *if the promise problem Π is Cook-reducible to a promise problem that is solvable in polynomial-time, then Π is solvable in polynomial-time.*

We warn that the extension of a complexity class to promise problems does not necessarily inherit the “structural” properties of the standard class. For example, in contrast to Theorem 2.33, there exists promise problems in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ such that every set in \mathcal{NP} can be Cook-reduced to them: see Exercise 2.36. Needless to say, $\mathcal{NP} = \text{co}\mathcal{NP}$ does not seem to follow from Exercise 2.36. See further discussion at the end of §2.4.1.2.

²³It follows that Karp-reductions among promise problems are not allowed to make queries that violate the promise. Specifically, we say that the promise problem $\Pi = (\Pi_{\text{yes}}, \Pi_{\text{no}})$ is Karp-reducible to the promise problem $\Pi' = (\Pi'_{\text{yes}}, \Pi'_{\text{no}})$ if there exists a polynomial-time mapping f such that for every $x \in \Pi_{\text{yes}}$ (resp., $x \in \Pi_{\text{no}}$) it holds that $f(x) \in \Pi'_{\text{yes}}$ (resp., $f(x) \in \Pi'_{\text{no}}$).

2.4.1.2 Discussion

The following discussion refers both to the decision and search versions of promise problems. Recall that promise problems offer the most direct way of capturing natural computational problems (e.g., when referring to computational problems regarding graphs, the promise mandates that the input is a graph).

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

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

1. *Extending the set of instances to the set of all possible strings* (and allowing trivial solutions for the corresponding dummy instances). For example, in the case of a search problem, we may either define all instance that violate the promise to have no solution or define them to have a trivial solution (e.g., be a solution for themselves); that is, for a search problem R with promise P , we may consider the (standard) search problem of R where R is modified such that $R(x) = \emptyset$ for every $x \notin P$ (or, say, $R(x) = \{x\}$ for every $x \notin P$). In the case of a promise (decision) problem $(S_{\text{yes}}, S_{\text{no}})$, we may consider the problem of deciding membership in S_{yes} , which means that instances that violate the promise are considered as no-instances.
2. *Considering every string as a valid encoding of an object that satisfies the promise.* That is, fixing any string x_0 that satisfies the promise, we consider

every string that violates the promise as if it were x_0 . In the case of a search problem R with promise P , this means considering the (standard) search problem of R where R is modified such that $R(x) = R(x_0)$ for every $x \notin P$. Similarly, in the case of a promise (decision) problem $(S_{\text{yes}}, S_{\text{no}})$, we consider the problem of deciding membership in S_{yes} (provided $x_0 \in S_{\text{no}}$ and otherwise we consider the problem of deciding membership in $\{0, 1\}^* \setminus S_{\text{no}}$).

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

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

Other uses and some reservations. As mentioned in the foregoing, the notion of a promise problem provides good formulations of many natural computational problems. In addition, this notion provides adequate formulations for a variety of computational complexity notions and results. Examples include the notion of “unique solutions” (see Section 6.2.3) and the formulation of “gap problems” as capturing various approximation tasks (see Section 10.1). In all these cases, promise problems allow to discuss natural computational problems and make statements about their inherent complexity. Thus, the complexity of promise problems (and classes of such problems) addresses natural questions and concerns. Consequently, demonstrating the intractability of a promise problem in some class (e.g., saying that some promise problem in \mathcal{NP} cannot be solved by a polynomial-time algorithm) carries the same conceptual message as demonstrating the intractability of a standard problem in the corresponding class. In contrast, as indicated at the end of §2.4.1.1, structural properties of promise problems may not hold for the corresponding classes of standard problems (e.g., see Exercise 2.36). Indeed, we do distinguish here between the inherent (or absolute) properties such as intractability and structural (or relative) properties such as reducibility.

2.4.1.3 The common convention

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

2.4.2 Optimal search algorithms for NP

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

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

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

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

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

We stress that the hidden constant in the O-notation depends only on A' , but in the following proof the dependence is exponential in the length of the description of algorithm A' (and it is not known whether a better dependence can be achieved). Indeed, this dependence as well as the idea underlying it constitute one negative aspect of this otherwise amazing result. Another negative aspect is that the optimality of algorithm A refers only to inputs that have a solution (i.e., inputs in S_R).

Finally, we note that the theorem as stated refers only to models of computation that have machines that can emulate a given number of steps of other machines with a constant overhead. We mention that in most natural models the overhead of such emulation is at most poly-logarithmic in the number of steps, in which case it holds that $t_A(x) = \tilde{O}(t_{A'}(x) + p(|x|))$.

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 (as a function of the length of the first element in the input pair). We present the following algorithm A that merely emulates all possible search algorithms “in parallel” and checks the result provided by each of them (using M), halting whenever it obtains a correct solution.

Since there are infinitely many possible algorithms, it may not be clear what we mean by the expression “emulating all possible algorithms in parallel.” What we mean is emulating them at different “rates” such that the infinite sum of these rates converges to 1 (or to any other constant). Specifically, we will emulate the i^{th} possible algorithm at rate $1/(i+1)^2$, which means emulating a single step of this algorithm per $(i+1)^2$ emulation steps (performed for all algorithms). Note that a straightforward implementation of this idea may create a significant overhead, involved in switching frequently from the emulation of one machine to the emulation of another. Instead, we present an alternative implementation that proceeds in iterations.

In the j^{th} iteration, for $i = 1, \dots, 2^{j/2} - 1$, algorithm A emulates $2^j/(i+1)^2$ steps of the i^{th} machine (where the machines are ordered according to the lexicographic order of their descriptions). Each of these emulations is conducted in one chunk, and thus the overhead of switching between the various emulations is insignificant (in comparison to the total number of steps being emulated). In the case that some of these emulations halts with output y , algorithm A invokes M on input (x, y) and output y if and only if $M(x, y) = 1$. Furthermore, the verification of a solution provided by a candidate algorithm is also emulated at the expense of its step-count. (Put in other words, we augment each algorithm with a canonical procedure (i.e., M) that checks the validity of the solution offered by the algorithm.)

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

For every x , let us denote by $t'(x)$ the number of steps taken by A' on input x , where $t'(x)$ also accounts for the running time of $M(x, \cdot)$; that is, $t'(x) = t_{A'}(x) + p(|x|)$, where $t_{A'}(x)$ is the number of steps taken by $A'(x)$. Then, the emulation of $t'(x)$ steps of A' on input x is “covered” by the j^{th} iteration of A , provided that $2^j/(2^{|A'|+1})^2 \geq t'(x)$ where $|A'|$ denotes the length of the description of A' . (Indeed,

we use the fact that the algorithms are emulated in lexicographic order, and note that there are at most $2^{|A'|+1} - 2$ algorithms that precede A' in lexicographic order.) Thus, on input x , algorithm A halts after at most $j_{A'}(x)$ iterations, where $j_{A'}(x) = 2(|A'| + 1) + \log_2(t_{A'}(x) + p(|x|))$, after emulating a total number of steps that is at most

$$t(x) \stackrel{\text{def}}{=} \sum_{j=1}^{j_{A'}(x)} \sum_{i=1}^{2^{j/2}-1} \frac{2^j}{(i+1)^2} < 2^{j_{A'}(x)+1} = 2^{2|A'|+3} \cdot (t_{A'}(x) + p(|x|)).$$

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

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

2.4.3 The class coNP and its intersection with NP

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

$$\text{co}\mathcal{C} \stackrel{\text{def}}{=} \{\{0, 1\}^* \setminus S : S \in \mathcal{C}\}. \quad (2.4)$$

Specifically, $\text{co}\mathcal{NP} = \{\{0, 1\}^* \setminus S : S \in \mathcal{NP}\}$ is the class of sets that are complements of sets in \mathcal{NP} .

Recalling that sets in \mathcal{NP} are characterized by their witness relations such that $x \in S$ if and only if there exists an adequate NP-witness, it follows that their complement sets consists of all instances for which there are no NP-witness (i.e., $x \in \{0, 1\}^* \setminus S$ if there is no NP-witness for x being in S). For example, $\text{SAT} \in \mathcal{NP}$ implies that the set of unsatisfiable CNF formulae is in $\text{co}\mathcal{NP}$. Likewise, the set of graphs that are not 3-colorable is in $\text{co}\mathcal{NP}$. (Jumping ahead, we mention that it is widely believed that these sets are not in \mathcal{NP} .)

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

It is widely believed that $\mathcal{NP} \neq \text{co}\mathcal{NP}$ (which means that \mathcal{NP} is not closed under complementation). Indeed, this conjecture implies $\mathcal{P} \neq \mathcal{NP}$ (because \mathcal{P} is closed under complementation). The conjecture $\mathcal{NP} \neq \text{co}\mathcal{NP}$ means that some

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

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

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

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

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

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

In particular, assuming $\mathcal{NP} \neq \text{co}\mathcal{NP}$, no set in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ can be NP-complete, even when NP-completeness is defined with respect to Cook-reductions. Since $\mathcal{NP} \cap \text{co}\mathcal{NP}$ is conjectured to be a proper superset of \mathcal{P} , it follows (assuming $\mathcal{NP} \neq \text{co}\mathcal{NP}$) that there are decision problems in \mathcal{NP} that are neither in \mathcal{P} nor NP-hard (i.e., specifically, the decision problems in $(\mathcal{NP} \cap \text{co}\mathcal{NP}) \setminus \mathcal{P}$). We stress that Theorem 2.33 refers to standard decision problems and not to promise problems (see Section 2.4.1 and Exercise 2.36).

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

Fixing any S and $S' \in \mathcal{NP} \cap \text{co}\mathcal{NP}$ such that S is Cook-reducible to S' , we prove that $S \in \mathcal{NP}$ (and the proof that $S \in \text{co}\mathcal{NP}$ is similar).²⁴ Let us denote by M the oracle machine reducing S to S' . That is, on input x , machine M makes queries and decides whether or not to accept x , and its decision is correct provided all queries are answered according to S' . To show that $S \in \mathcal{NP}$, we will present an NP-proof system for S . This proof system (or rather its verification procedure), denoted V , accepts a pair of the form $(x, ((z_1, \sigma_1, w_1), \dots, (z_t, \sigma_t, w_t)))$ if the following two conditions hold:

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

That is, V check that, on input x , after obtaining the answers $\sigma_1, \dots, \sigma_{i-1}$ to the first $i - 1$ queries, the i^{th} query made by M equals z_i . In addition, V checks that M outputs 1 (indicating acceptance), while making the queries z_1, \dots, z_t and receiving the answers $\sigma_1, \dots, \sigma_t$, respectively.

Note that V does not have oracle access to S' . The procedure V rather emulates the computation of $M(x)$ by using, for each i , the bit σ_i (provided to V as part of its input) as an answer to the i^{th} query of $M(x)$. The correctness of these answers will be verified (by V) separately (i.e., see the next item).

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

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

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

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

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

²⁴Alternatively, we show that $S \in \text{co}\mathcal{NP}$ by applying the following argument to $\overline{S} \stackrel{\text{def}}{=} \{0, 1\}^* \setminus S$ and noting that \overline{S} is Cook-reducible to S' (via S , or alternatively that \overline{S} is Cook-reducible to $\{0, 1\}^* \setminus S' \in \mathcal{NP} \cap \text{co}\mathcal{NP}$).

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

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

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

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

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

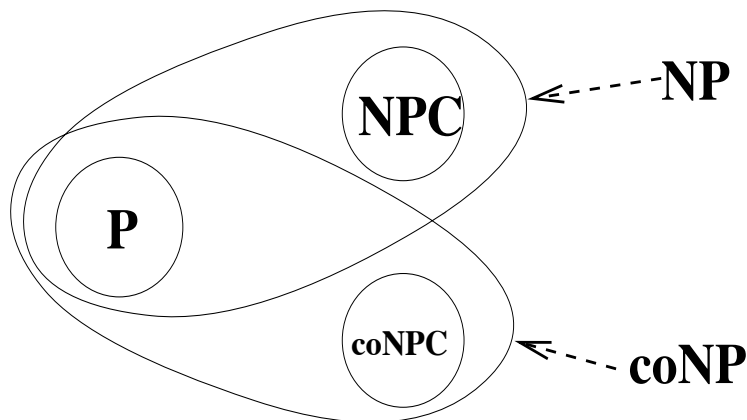


Figure 2.5: The world view under $\mathcal{P} \neq \text{co}\mathcal{NP} \cap \mathcal{NP} \neq \mathcal{NP}$.

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

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

Chapter Notes

Many sources provide historical accounts of the developments that led to the formulation of the *P vs NP Problem* and to the discovery of the theory of NP-completeness (see, e.g., [81, Sec. 1.5] and [213]). Still, we feel that we should not refrain from offering our own impressions, which are based on the texts of the original papers.

Nowadays, the theory of NP-completeness is commonly attributed to Cook [55], Karp [131], and Levin [146]. It seems that Cook's starting point was his interest

in theorem proving procedures for propositional calculus [55, P. 151]. Trying to provide evidence to the difficulty of deciding whether or not a given formula is a tautology, he identified \mathcal{NP} as a class containing “many apparently difficult problems” (cf, e.g., [55, P. 151]), and showed that any problem in \mathcal{NP} is reducible to deciding membership in the set of 3DNF tautologies. In particular, Cook emphasized the importance of the concept of polynomial-time reductions and the complexity class \mathcal{NP} (both explicitly defined for the first time in his paper). He also showed that CLIQUE is computationally equivalent to SAT, and envisioned a class of problems of the same nature.

Karp’s paper [131] can be viewed as fulfilling Cook’s prophecy: Stimulated by Cook’s work, Karp demonstrated that a “large number of classic difficult computational problems, arising in fields such as mathematical programming, graph theory, combinatorics, computational logic and switching theory, are [NP-]complete (and thus equivalent)” [131, P. 86]. Specifically, his list of twenty-one NP-complete problems includes Integer Linear Programming, Hamilton Circuit, Chromatic Number, Exact Set Cover, Steiner Tree, Knapsack, Job Scheduling, and Max Cut. Interestingly, Karp defined \mathcal{NP} in terms of verification procedures (i.e., Definition 2.5), pointed to its relation to “backtrack search of polynomial bounded depth” [131, P. 86], and viewed \mathcal{NP} as the residence of a “wide range of important computational problems” (which are not in \mathcal{P}).

Independently of these developments, while being in the USSR, Levin proved the existence of “universal search problems” (where universality meant NP-completeness). The starting point of Levin’s work [146] was his interest in the “perebor” conjecture asserting the inherent need for brute-force in some search problems that have efficiently checkable solutions (i.e., problems in \mathcal{PC}). Levin emphasized the implication of polynomial-time reductions on the relation between the time complexity of the related problem (for any growth rate of the time complexity), asserted the NP-completeness of six “classical search problems”, and claimed that the underlying method “provides a mean for readily obtaining” similar results for “many other important search problems.”

It is interesting to note that although the works of Cook [55], Karp [131], and Levin [146] were received with different levels of enthusiasm, none of the contemporaries realized the depth of the discovery and the difficulty of the question posed (i.e., the P-vs-NP Question). This fact is evident in every account from the early 1970’s, and may explain the frustration of the corresponding generation of researchers, which expected the P-vs-NP Question to be resolved in their life-time (if not in a matter of years). Needless to say, the author’s opinion is that there was absolutely no justification for these expectations, and that one should have actually expected quite the opposite.

We mention that the three “founding papers” of the theory of NP-completeness (i.e., Cook [55], Karp [131], and Levin [146]) use the three different types of reductions used in this chapter. Specifically, Cook uses the general notion of polynomial-time reduction [55], often referred to as Cook-reductions (Definition 2.9). The notion of Karp-reductions (Definition 2.10) originates from Karp’s paper [131], whereas its augmentation to search problems (i.e., Definition 2.11) originates from

Levin's paper [146]. It is worth noting that unlike Cook and Karp's works, which treat decision problems, Levin's work is stated in terms of search problems.

The reductions presented in §2.3.3.2 are not necessarily the original ones. Most notably, the reduction establishing the NP-hardness of the Independent Set problem (i.e., Proposition 2.25) is adapted from [70] (see also Exercise 9.14). In contrast, the reductions presented in §2.3.3.1 are merely a re-interpretation of the original reduction as presented in [55]. The equivalence of the two definitions of \mathcal{NP} (i.e., Theorem 2.8) was proved in [131].

The existence of NP-sets that are neither in P nor NP-complete (i.e., Theorem 2.27) was proven by Ladner [143], Theorem 2.33 was proven by Selman [191], and the existence of optimal search algorithms for NP-relations (i.e., Theorem 2.31) was proven by Levin [146]. (Interestingly, the latter result was proved in the same paper in which Levin presented the discovery of NP-completeness, independently of Cook and Karp.) Promise problems were explicitly introduced by Even, Selman and Yacobi [68]; see [90] for a survey of their numerous applications.

We mention that the standard reductions used to establish natural NP-completeness results have several additional properties or can be modified to have such properties. These properties include an efficient transformation of solutions in the direction of the reduction (see Exercise 2.28), the preservation of the number of solutions (see Exercise 2.29), being computable by a log-space algorithm (see Section 5.2.2), and being invertible in polynomial-time (see Exercise 2.30). We also mention the fact that all known NP-complete sets are (effectively) isomorphic (see Exercise 2.31).

Exercises

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

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

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

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

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

Guideline: Consider the following verification procedures V_1 and V_2 for the set of composite numbers. Let $V_1(n, y) = 1$ if and only if $y = n$ and n is not a prime, and

$V_2(n, m) = 1$ if and only if m is a non-trivial divisor of n . Show that valid proofs with respect to V_1 are easy to find, whereas valid proofs with respect to V_2 are hard to find.

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

Exercise 2.5 Verify the following properties of Cook-reductions:

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

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

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

Exercise 2.7 Show that some decision problems are not Karp-reducible to their complement (e.g., the empty set is not Karp-reducible to $\{0, 1\}^*$).

A popular exercise of dubious nature is showing that any decision problem in \mathcal{P} is Karp-reducible to any *non-trivial* decision problem, where the decision problem regarding a set S is called non-trivial if $S \neq \emptyset$ and $S \neq \{0, 1\}^*$. It follows that every non-trivial set in \mathcal{P} is Karp-reducible to its complement.

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

(Hint: use a Boolean function.)

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

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

Exercise 2.10 Show that if $R \in \mathcal{PC} \setminus \mathcal{PF}$ is self-reducible then the relevant Cook-reduction makes more than a logarithmic number of queries to S_R . More generally, show that if $R \in \mathcal{PC} \setminus \mathcal{PF}$ is Cook-reducible to any decision problem, then this reduction makes more than a logarithmic number of queries.

Guideline: Note that the oracle answers can be emulated by trying all possibilities, and that the correctness of the output of the oracle machine can be efficiently tested.

Exercise 2.11 Show that the standard search problem of Graph 3-Colorability is self-reducible, where this search problem consists of finding a 3-coloring for a given input graph.

(Hint: Iteratively extend the current prefix of a 3-coloring of the graph by making adequate oracle calls to the decision problem of Graph 3-Colorability. Specifically, encode the question of whether or not $(\chi_1, \dots, \chi_t) \in \{1, 2, 3\}^t$ is a prefix of a 3-coloring of the graph G as a query regarding the 3-colorability of an auxiliary graph G' .)²⁵

Exercise 2.12 Show that the standard search problem of Graph Isomorphism is self-reducible, where this search problem consists of finding an isomorphism between a given pair of graphs.

(Hint: Iteratively extend the current prefix of an isomorphism between the two N -vertex graphs by making adequate oracle calls to the decision problem of Graph Isomorphism. Specifically, encode the question of whether or not $(\pi_1, \dots, \pi_t) \in [N]^t$ is a prefix of an isomorphism between $G_1 = ([N], E_1)$ and $G_2 = ([N], E_2)$ as a query regarding isomorphism between two auxiliary graphs G'_1 and G'_2 .)²⁶

Exercise 2.13 (downwards self-reducibility) We say that S is downwards self-reducible if there exists a Cook-reduction of S to itself that only makes queries that are each shorter than the reduction's input (i.e., if on input x the reduction makes the query q then $|q| < |x|$).²⁷

1. Show that SAT is downwards self-reducible with respect to a natural encoding of CNF formulae. Note that this encoding should have the property that instantiating a variable in a formula results in a shorter formula.

A harder exercise consists of showing that Graph 3-Colorability is downwards self-reducible with respect to some reasonable encoding of graphs. Note that this encoding has to be selected carefully (if it is to work for a process analogous to the one used in Exercise 2.11).

2. Suppose that S is downwards self-reducible by a reduction that outputs the disjunction of the oracle answers. (Note that this is the case for SAT.) Show that in this case, S is characterized by a witness relation $R \in \mathcal{PC}$ (i.e., $S = \{x : R(x) \neq \emptyset\}$) that is self-reducible (i.e., the search problem of R is Cook-reducible to S). Needless to say, it follows that $S \in \mathcal{NP}$.

Guideline: Include $(x_0, \langle x_1, \dots, x_t \rangle)$ in R if $x_t \in S \cap \{0, 1\}^{O(1)}$ and, for every $i \in \{0, 1, \dots, t-1\}$, on input x_i the self-reduction makes a set of queries that contains x_{i+1} . Prove that, indeed, $R \in \mathcal{PC}$ and $S = \{x : R(x) \neq \emptyset\}$.

²⁵Note that we merely need to check whether G has a 3-coloring in which the equalities and inequalities induced by (χ_1, \dots, χ_t) hold. This can be done by adequate gadgets (e.g., inequality is enforced by an edge between the corresponding vertices, whereas equality is enforced by an adequate subgraph that includes the relevant vertices as well as auxiliary vertices). For Part 1 of Exercise 2.13, equality is better enforced by combining the two vertices.

²⁶This can be done by attaching adequate gadgets to pairs of vertices that we wish to be mapped to one another (by the isomorphism). For example, we may connect the vertices in the i^{th} pair to an auxiliary star consisting of $(N + i)$ vertices.

²⁷Note that on some instances the reduction may make no queries at all. (This prevents a possible non-viability of the definition due to very short instances.)

Note that the notion of downwards self-reducibility may be generalized in some natural ways. For example, we may say that S is downwards self-reducible also in case it is computationally equivalent to some set that is downwards self-reducible (in the foregoing strict sense). Note that Part 2 still holds.

Exercise 2.14 (NP problems that are not self-reducible) Assuming that $\mathcal{P} \neq \mathcal{NP} \cap \text{co}\mathcal{NP}$, show that there exists a search problem R in \mathcal{PC} that is not self-reducible (i.e., the search problem of R is not Cook-reducible to the decision problem S_R implicit in R). Prove that it follows that $S'_R = \{(x, y') : \exists y'' \text{ s.t. } (x, y'y'') \in R\}$ is not Cook-reducible to $S_R = \{x : \exists y \text{ s.t. } (x, y) \in R\}$.

Guideline: Given $S \in \mathcal{NP} \cap \text{co}\mathcal{NP} \setminus \mathcal{P}$, present relations $R_1, R_2 \in \mathcal{PC}$ such that $S = \{x : R_1(x) \neq \emptyset\} = \{x : R_2(x) = \emptyset\}$. Then, consider the relation $R = \{(x, 1y) : (x, y) \in R_1\} \cup \{(x, 0y) : (x, y) \in R_2\}$, and prove that $R \notin \mathcal{PF}$ but $S_R = \{0, 1\}^*$.

Exercise 2.15 In continuation to Exercise 2.14 and assuming that $\mathcal{P} \neq \mathcal{NP}$, present a search problem R in \mathcal{PC} such that deciding S'_R is not reducible to the search problem of R .

Guideline: Consider the relation $R = \{(x, 0x) : x \in \{0, 1\}^*\} \cup \{(x, 1y) : (x, y) \in R'\}$, where R' is an arbitrary relation in $\mathcal{PC} \setminus \mathcal{PF}$, and prove that $R \in \mathcal{PF}$ but $S'_R \notin \mathcal{P}$.

Exercise 2.16 In continuation to Exercise 2.14, present a natural search problem R in \mathcal{PC} such that if factoring integers is intractable then the search problem R (and so also S'_R) is not reducible to S_R .

Guideline: Consider the relation R such that $(N, Q) \in R$ if the integer Q is a non-trivial divisor of the integer N . Use the fact that the set of prime numbers is in \mathcal{P} .

Exercise 2.17 In continuation to Exercises 2.14 and 2.16, show that under suitable assumptions there exists relations $R_1, R_2 \in \mathcal{PC}$ having the same implicit-decision problem (i.e., $\{x : R_1(x) \neq \emptyset\} = \{x : R_2(x) \neq \emptyset\}$) such that R_1 is self-reducible but R_2 is not.

Exercise 2.18 Provide an alternative proof of Theorem 2.15 without referring to the set $S'_R = \{(x, y') : \exists y'' \text{ s.t. } (x, y'y'') \in R\}$. Hint: use Proposition 2.14.

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

Exercise 2.19 Prove that **Bounded Halting** and **Bounded Non-Halting** are NP-complete, where the problems are defined as follows. The instance consists of a pair $(M, 1^t)$, where M is a Turing machine and t is an integer. The decision version of **Bounded Halting** (resp., **Bounded Non-Halting**) consists of determining whether or not there exists an input (of length at most t) on which M halts (resp., does *not* halt) in t steps, whereas the search problem consists of finding such an input. (Hint: Either modify the proof of Theorem 2.18 or present a reduction of (say) the search problem of R_{u} to the search problem of Bounded (Non-)Halting. Indeed, the exercise is more straightforward in the case of Bounded Halting.)

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

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

Exercise 2.21 Present and analyze a reduction of SAT to 3SAT.

Guideline: For a clause C , consider auxiliary variables such that the i^{th} variable indicates whether one of the first i literals is satisfied, and replace C by a 3CNF that uses the original variables of C as well as the auxiliary variables. For example, the clause $\bigvee_{i=1}^t x_i$ is replaced by the conjunction of 3CNFs that are logically equivalent to the formulae $(y_2 \equiv (x_1 \vee x_2))$, $(y_i \equiv (y_{i-1} \vee x_i))$ for $i = 3, \dots, t$, and y_t . We comment that this is not the standard reduction, but we find it more appealing conceptually.²⁸

Exercise 2.22 (efficient solveability of 2SAT) In contrast to Exercise 2.21, prove that 2SAT (i.e., the satisfiability of 2CNF formulae) is in \mathcal{P} .

Guideline: Consider the following “forcing process” for CNF formulae. If the formula contains a singleton clause (i.e., a clause having a single literal), then the corresponding variable is assigned the only value that satisfies the clause, and the formula is simplified accordingly (possibly yielding a constant, which is either **true** or **false**). The process is repeated until the formula is either a constant or contains only 2-literal clauses. Note that a formula ϕ is satisfiable if and only if the formula obtained from ϕ by the forcing process is satisfiable.

1. Prove that a 2CNF formula is unsatisfiable if and only if there exists a variable such that any truth assignment to this variable yields a formula that the forcing process maps to the constant **false**.
(Extra hint: Applying the forcing process to a 2CNF formula we obtain a sub-formula of it; that is, each clause of the resulting formula is a clause (rather than a sub-clause) of the original formula.)
2. Using Part 1, present a polynomial-time algorithm for solving the search problem of 2SAT.

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

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

²⁸The standard reduction replaces the clause $\bigvee_{i=1}^t x_i$ by the conjunction of the 3CNFs $(x_1 \vee x_2 \vee z_2)$, $((\neg z_{i-1}) \vee x_i \vee z_i)$ for $i = 3, \dots, t$, and $\neg z_t$.

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

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

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

Exercise 2.25 (Satisfiability of Quadratic Systems over GF(2)) Prove that the following problem is NP-complete. An instance of the problem consists of a system of quadratic equations over GF(2), and the problem is to determine whether there exists an assignment that satisfies all the equations. Note that the result holds also for systems of quadratic equations over the reals (by adding conditions that enforce a value in $\{0, 1\}$).

Guideline: Start by showing that the corresponding problem for cubic equations is NP-complete, by a reduction from 3SAT that maps the clause $x \vee \neg y \vee z$ to the equation $(1 - x) \cdot y \cdot (1 - z) = 0$. Reduce the problem for cubic equations to the problem for quadratic equations by introducing auxiliary variables; that is, given an instance with variables x_1, \dots, x_n , introduce the auxiliary variables $x_{i,j}$'s and add equations of the form $x_{i,j} = x_i \cdot x_j$.

Exercise 2.26 (Clique and Independent Set) The instance of the Independent Set problem consists of a pair (G, K) , where G is a graph and K is an integer, and the question is whether or not the graph G contains an independent set (i.e., a set with no edges between its members) of size (at least) K . The Clique problem is analogous. Prove that both problems are computationally equivalent to the Vertex Cover problem.

Exercise 2.27 (an alternative proof of Proposition 2.25) Consider the following sketch of a reduction of 3SAT to Independent Set. On input a 3CNF formula ϕ with m clauses and n variables, we construct a graph G_ϕ consisting of m triangles (corresponding to the m clauses) augmented with edges that link conflicting literals. That is, if x appears as the i_1^{th} literal of the j_1^{th} clause and $\neg x$ appears

as the i_2^{th} literal of the j_2^{th} clause, then we draw an edge between the i_1^{th} vertex of the j_1^{th} triangle and the i_2^{th} vertex of the j_2^{th} triangle. Prove that $\phi \in 3\text{SAT}$ if and only if G_ϕ has an independent set of size m .

Exercise 2.28 (additional properties of standard reductions) In continuation to the discussion in the main text, consider the following augmented form of Karp-reductions. Such a reduction of R to R' consists of three polynomial-time mappings (f, h, g) such that f is a Karp-reduction of S_R to $S_{R'}$ and the following two conditions hold:

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

(We note that this definition is actually the one used by Levin in [146], except that he restricted h and g to only depend on their second argument.)

Prove that such a reduction implies both a Karp-reduction and a Levin-Reduction, and show that all reductions presented in this chapter satisfy this augmented requirement. Furthermore, prove that in all these cases the main mapping (i.e., f) is 1-1 and polynomial-time invertible.

Exercise 2.29 (parsimonious reductions) Let $R, R' \in \mathcal{PC}$ and let f be a Karp-reduction of $S_R = \{x : R(x) \neq \emptyset\}$ to $S_{R'} = \{x : R'(x) \neq \emptyset\}$. We say that f is *parsimonious* (with respect to R and R') if for every x it holds that $|R(x)| = |R'(f(x))|$. For each of the reductions presented in this chapter, checked whether or not it is parsimonious. For the reductions that are not parsimonious, find alternative reductions that are parsimonious (cf. [81, Sec. 7.3]).

Exercise 2.30 (on polynomial-time invertible reductions (following [35]))

We say that a set S is *markable* if there exists a polynomial-time (marking) algorithm M such that

1. For every $x, \alpha \in \{0, 1\}^*$ it holds that
 - (a) $M(x, \alpha) \in S$ if and only if $x \in S$.
 - (b) $|M(x, \alpha)| > |x|$.
2. There exists a polynomial-time (de-marking) algorithm D such that, for every $x, \alpha \in \{0, 1\}^*$, it holds that $D(M(x, \alpha)) = \alpha$.

Note that all natural NP-sets (e.g., those considered in this chapter) are markable (e.g., for SAT, one may mark a formula by augmenting it with additional satisfiable clauses that use specially designated auxiliary variables). Prove that *if S' is Karp-reducible to S and S is markable then S' is Karp-reducible to S by a length-increasing, one-to-one, and polynomial-time invertible mapping.*²⁹ Infer that for any natural NP-complete problem S , any set in \mathcal{NP} is Karp-reducible to S by a length-increasing, one-to-one, and polynomial-time invertible mapping.

Guideline: Let f be a Karp-reduction of S' to S , and let M be the guaranteed marking algorithm. Consider the reduction that maps x to $M(f(x), x)$.

²⁹When given a string that is not in the image of the mapping, the inverting algorithm returns a special symbol.

Exercise 2.31 (on the isomorphism of NP-complete sets (following [35]))

Suppose that S and T are Karp-reducible to one another by length-increasing, one-to-one, and polynomial-time invertible mappings, denoted f and g respectively. Using the following guidelines, prove that S and T are “effectively” *isomorphic*; that is, present a polynomial-time computable and invertible one-to-one mapping ϕ such that $T = \phi(S) \stackrel{\text{def}}{=} \{\phi(x) : x \in S\}$.

1. Let $F \stackrel{\text{def}}{=} \{f(x) : x \in \{0, 1\}^*\}$ and $G \stackrel{\text{def}}{=} \{g(x) : x \in \{0, 1\}^*\}$. Using the length-preserving condition of f (resp., g), prove that F (resp., G) is a proper subset of $\{0, 1\}^*$. Prove that for every $y \in \{0, 1\}^*$ there exists a unique triple $(j, x, i) \in \{1, 2\} \times \{0, 1\}^* \times \mathbb{N}$ that satisfies one of the following two conditions:

- (a) $j = 1$, $x \in \overline{G} \stackrel{\text{def}}{=} \{0, 1\}^* \setminus G$, and $y = (g \circ f)^i(x)$;
- (b) $j = 2$, $x \in \overline{F} \stackrel{\text{def}}{=} \{0, 1\}^* \setminus F$, and $y = (g \circ f)^i(g(x))$.

(In both cases $i = 0$ is allowed, $h^0(z) = z$, $h^i(z) = h(h^{i-1}(z))$, and $(g \circ f)(z) = g(f(z))$. Hint: consider the maximal sequence of inverse operations $g^{-1}, f^{-1}, g^{-1}, \dots$ that can be applied to y , and note that each inverse shrinks the current string.)

2. Let $U_1 \stackrel{\text{def}}{=} \{(g \circ f)^i(x) : x \in \overline{G} \wedge i \geq 0\}$ and $U_2 \stackrel{\text{def}}{=} \{(g \circ f)^i(g(x)) : x \in \overline{F} \wedge i \geq 0\}$. Prove that (U_1, U_2) is a partition of $\{0, 1\}^*$. Using the fact that f and g are length increasing and polynomial-time invertible, present a polynomial-time procedure for deciding membership in the set U_1 .

Prove the same for the sets $V_1 = \{(f \circ g)^i(x) : x \in \overline{F} \wedge i \geq 0\}$ and $V_2 = \{(f \circ g)^i(f(x)) : x \in \overline{G} \wedge i \geq 0\}$.

3. Note that $U_2 \subseteq G$, and define $\phi(x) \stackrel{\text{def}}{=} f(x)$ if $x \in U_1$ and $\phi(x) \stackrel{\text{def}}{=} g^{-1}(x)$ otherwise.

- (a) Prove that ϕ is a Karp-reduction of S to T .
- (b) Note that ϕ maps U_1 to $f(U_1) = \{f(x) : x \in U_1\} = V_2$ and U_2 to $g^{-1}(U_2) = \{g^{-1}(x) : x \in U_2\} = V_1$. Prove that ϕ is one-to-one and onto.

Observe that $\phi^{-1}(x) = f^{-1}(x)$ if $x \in f(U_1)$ and $\phi^{-1}(x) = g(x)$ otherwise. Prove that ϕ^{-1} is a Karp-reduction of T to S . Infer that $\phi(S) = T$.

Using Exercise 2.30, infer that all natural NP-complete sets are isomorphic.

Exercise 2.32 Prove that a set S is Karp-reducible to some set in \mathcal{NP} if and only if S is in \mathcal{NP} .

(Hint: For the non-trivial direction, see the proof of Proposition 2.32.)

Exercise 2.33 Recall that the empty set is not Karp-reducible to $\{0, 1\}^*$, whereas any set is Cook-reducible to its complement. Thus our focus here is on the *Karp-reducibility of non-trivial sets to their complements*, where a set is non-trivial if it is neither empty nor contains all strings. Furthermore, since any non-trivial set in \mathcal{P} is Karp-reducible to its complement (see Exercise 2.7), we assume that $\mathcal{P} \neq \mathcal{NP}$ and focus on sets in $\mathcal{NP} \setminus \mathcal{P}$.

1. Prove that $\mathcal{NP} = \text{co}\mathcal{NP}$ implies that some sets in $\mathcal{NP} \setminus \mathcal{P}$ are Karp-reducible to their complements.
2. Prove that $\mathcal{NP} \neq \text{co}\mathcal{NP}$ implies that some sets in $\mathcal{NP} \setminus \mathcal{P}$ are not Karp-reducible to their complements.

(Hint: Use NP-complete sets in both parts, and Exercise 2.32 in the second part.)

Exercise 2.34 Referring to the proof of Theorem 2.27, prove that the function f is unbounded (i.e., for every i there exists an n such that n^3 steps of the process defined in the proof allow for failing the $i + 1^{\text{st}}$ machine).

Guideline: Assume, towards the contradiction that f is bounded. Let $i = \sup_{n \in \mathbb{N}} \{f(n)\}$ and n' be the smallest integer such that $f(n') = i$. If i is odd then the set F determined by f is co-finite (because $F = \{x : f(|x|) \equiv 0 \pmod{2}\} \supseteq \{x : |x| \geq n'\}$). In this case, the $i + 1^{\text{st}}$ machine tries to decide $S \cap F$ (which differs from S on finitely many strings), and must fail on some x . Derive a contradiction by showing that the number of steps taken till reaching and considering this x is at most $\exp(\text{poly}(|x|))$, which is smaller than n^3 for some sufficiently large n . A similar argument applies to the case that i is even, where we use the fact that $F \subseteq \{x : |x| < n'\}$ is finite and so the relevant reduction of S to $S \cap F$ must fail on some input x .

Exercise 2.35 Prove that if the promise problem Π is Cook-reducible to a promise problem that is solvable in polynomial-time, then Π is solvable in polynomial-time. Note that the solver may not halt on inputs that violate the promise.

Guideline: Any polynomial-time algorithm solving any promise problem can be modified such that it halts on all inputs.

Exercise 2.36 (NP-complete promise problems in coNP (following [68])) Consider the promise problem **xSAT**, having instances that are pairs of CNF formulae. The yes-instances consists of pairs (ϕ_1, ϕ_2) such that ϕ_1 is satisfiable and ϕ_2 is unsatisfiable, whereas the no-instances consists of pairs such that ϕ_1 is unsatisfiable and ϕ_2 is satisfiable.

1. Show that **xSAT** is in the intersection of (the promise problem classes that are analogous to) \mathcal{NP} and $\text{co}\mathcal{NP}$.
2. Prove that any promise problem in \mathcal{NP} is Cook-reducible to **xSAT**. In designing the reduction, recall that queries that violate the promise may be answered arbitrarily.

Guideline: Show a reduction of **SAT** to **xSAT**. Specifically, show that the search problem associated with **SAT** is Cook-reducible to **xSAT**, by following the ideas of the proof of Proposition 2.14. Actually, we need a more careful implementation of the search process. Suppose that we know (or assume) that τ is a prefix of a satisfying assignment to ϕ , and we wish to extend τ by one bit. Then, for each $\sigma \in \{0, 1\}$, we construct a formula, denoted ϕ'_σ , by setting the first $|\tau| + 1$ variables of ϕ according to the values $\tau\sigma$. We query the oracle about the pair (ϕ'_1, ϕ'_0) , and extend τ accordingly (i.e., we extend τ by the value 1 if and only if the answer is

positive). Note that if both ϕ'_1 and ϕ'_0 are satisfiable then it does not matter which bit we use in the extension, whereas if exactly one formula is satisfiable then the oracle answer is reliable.

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

