Texts in Computational Complexity: P, NP and NP-Completeness

Oded Goldreich
Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Rehovot, ISRAEL.

January 27, 2006

Summary: The main focus of these notes is the P-vs-NP Question and the theory of NP-completeness.

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

An alternative formulation, in terms of decision problems, refers to assertions that have efficiently verifiable proofs (of relatively short length). Such sets of assertions correspond to the class NP, and the question is whether or not proofs for such assertions can be found efficiently (i.e., is there an efficient algorithm that an assertion determines its validity and/or finds a proof for its validity). Thus, the P-vs-NP Question can be phrased as asking *whether or not discovering proofs is harder than verifying their correctness*; that is, is proving harder than verifying (i.e., are proofs valuable at all).

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

The theory of NP-completeness is based on the notion of a reduction, which is a relation between computational problems. Loosely speaking, one computational problem is reducible to another problem if it is possible to efficiently solve the former when provided with an (efficient) algorithm for solving the latter. Thus, the first problem

is not harder to solve than the second one. A problem (in NP) is NP-complete if any problem in NP is reducible to it. Thus, the fate of the entire class NP (with respect to inclusion in P) rests with each individual NP-complete problem. In particular, showing that a problem is NP-complete implies that this problem is not in P unless NP equals P. Amazingly enough, NP-complete problems exist, and furthermore hundreds of natural computational problems arising in many different areas of mathematics and science are NP-complete.

Additional topics covered in these notes include the general notion of a polynomial-time reduction and the existence of problems in NP that are neither NP-complete nor in P. The text is prepended by preliminaries that present the relevant background of computability theory.

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

> **Teaching note:** This text contains amy teaching notes, reflecting 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 these notes is often taught in wrong ways, which fail to communicate its fundamental nature.

Contents

1 Computational Tasks and Models **4**
- 1.1 Representation . . . 4
- 1.2 Computational Tasks . . . 5
- 1.3 Uniform Models (Algorithms) . . . 6
  - 1.3.1 Turing machines . . . 8
  - 1.3.2 Uncomputable functions . . . 11
  - 1.3.3 Universal algorithms . . . 14
  - 1.3.4 Time and space complexity . . . 16
  - 1.3.5 Oracle machines . . . 18
  - 1.3.6 Restricted models . . . 19
- 1.4 Non-uniform Models (Circuits and Advice) . . . 19
  - 1.4.1 Boolean Circuits . . . 20
  - 1.4.2 Machines that take advice . . . 23
  - 1.4.3 Restricted models . . . 24
- 1.5 Complexity Classes . . . 25

2 The P versus NP Question **25**
- 2.1 The search version: finding versus checking . . . 26
  - 2.1.1 The class P as a natural class of search problems . . . 27
  - 2.1.2 The class NP as another natural class of search problems . . . 28
  - 2.1.3 The P versus NP question in terms of search problems . . . 29
- 2.2 The decision version: proving versus verifying . . . 29

2.2.1 The class P as a natural class of decision problems . . . 29
2.2.2 The class NP and NP-proof systems . . . 30
2.2.3 The P versus NP question in terms of decision problems . . . 32
2.3 Equivalence of the two formulations . . . 32
2.4 The traditional definition of NP . . . 33
2.5 In support of P different from NP . . . 34
2.6 Two technical comments regarding NP . . . 35

3 Polynomial-time Reductions **36**
3.1 The general notion of a reduction . . . 36
3.2 Reducing optimization problems to search problems . . . 38
3.3 Self-reducibility of search problems . . . 39

4 NP-Completeness **42**
4.1 Definitions . . . 43
4.2 The existence of NP-complete problems . . . 43
4.3 Some natural NP-complete problems . . . 45
4.3.1 Circuit and formula satisfiability: CSAT and SAT . . . 46
4.3.2 Combinatorics and graph theory . . . 51
4.4 NP sets that are neither in P nor NP-complete . . . 55

Notes **57**

Exercises **58**

Bibliography **63**

1 Computational Tasks and Models

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

The contents of Sections 1.1–1.3 corresponds to a traditional *Computability course*, and most of this material is taken for granted in the rest of the current course. In contrast, Section 1.4 presents basic preliminaries regarding non-uniform models of computation (i.e., various types of Boolean circuits), and these are only used lightly in the rest of the current course. Thus, whereas Sections 1.1–1.3 are absolute prerequisites for the rest of this course, Section 1.4 is not.

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

1.1 Representation

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

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

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

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

Numbers. Unless stated differently, natural numbers will be encoded by their binary expansion; that is, the string $b_{n-1} \cdots b_1 b_0 \in \{0,1\}^n$ encodes the number $\sum_{i=0}^{n-1} b_i \cdot 2^i$. Rational numbers will be represented as pairs of natural numbers. In the rare cases in which one considers real numbers as part of the input to a computational problem, one actually mean rational approximations of these real numbers.

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

1.2 Computational Tasks

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

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

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

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

Decision problems. A decision problem consists of a specification of a subset of the possible instances. Given an instance, one is required to determine whether the instance is in the specified set. For example, consider the problem where one is given a natural number, and is asked to determine whether or not the number is a prime. One important case, which corresponds to the aforementioned search problems, is the case of the set of instances having a solution; indeed, being able to determine whether or not a solution exists is a prerequisite to being able to solve the corresponding search problem (as per Definition 1). In general, decision problems refer to the natural task of making binary decision, a task that is not uncommon in daily life. In any case, in the following definition of solving search problems, the potential solver is again a function (i.e., in this case it is a Boolean function that is supposed to indicate membership in the said set).

Definition 2 (solving a decision problem) *Let $S \subseteq \{0,1\}^*$. A function $f : \{0,1\}^* \to \{0,1\}$* solves the decision problem of S (or decides membership in S) *if for every x it holds that $f(x) = 1$ if and only if $x \in S$.*

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

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

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

1.3 Uniform Models (Algorithms)

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

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

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

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

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

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

In order to define computation (and computation time) rigorously, one needs to specify some model of computation; that is, provide a concrete definition of environments and a class of rules that may be applied to them. Such a model corresponds to an abstraction of a real computer (be it a PC, mainframe or network of computers). One simple abstract model that is commonly used is that of *Turing machines* (see, §1.3.1 below). Thus, specific algorithms are typically formalized by corresponding Turing machines (and their time complexity is represented by the time complexity of the corresponding Turing machines). We stress, however, that most results in the Theory of Computation hold regardless of the specific computational model used, as long as it is "reasonable" (i.e., satisfies the aforementioned simplicity condition and can perform some obviously simple computations).

What is being Computed? The above discussion has implicitly referred to computations and algorithms as means of computing functions. Specifically, an algorithm A computes the function $f_A : \{0,1\}^* \to \{0,1\}^*$ defined by $f_A(x) = y$ if, when invoked on input x, algorithm A halts with output y. However, computations can also be viewed as a means of "solving search problems" or "making decisions" (as in Definitions 1 and 2). Specifically, we will say that algorithm A solves the search problem of R (resp., decides membership in S) if f_A solves the search problem of R (resp., decides membership in S). In the rest of this exposition we associate the algorithm A with the function f_A computed by it; that is, we write $A(x)$ instead of $f_A(x)$. For sake of future reference, we summarize the foregoing discussion.

Definition 3 (solution by an algorithm) *We denote by $A(x)$ the output of algorithm A on input x. Algorithm A* solves the search problem *R* (resp., the decision problem *S*) *if A, viewed as a function, solves R* (resp., *S*).

Organization of the rest of this section: In §1.3.1 we provide a sketchy description of the model of Turing machines. This is done merely for sake of providing a concrete model that supports the study of computation and its complexity, whereas most of the material in this course will not depend on the specifics of this model. In §1.3.2 and §1.3.2 we discuss two fundamental properties of any reasonable model of computation: the existence of uncomputable functions and the existence of universal computations. The time (and space) complexity of computation is defined in §1.3.4. We also discuss oracle machines and restricted models of computation (in §1.3.5 and §1.3.6, respectively).

1.3.1 Turing machines

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

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

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

The actual model. Following is a high-level description of the model of Turing machines; the interested reader is referred to standard textbooks (e.g., [16]) for further details. Recall that we need to specify the set of possible environments, the set of machines (or computation rules), and the effect of applying such a rule on an environment.

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

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

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

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

- A single computation step of such a Turing machine depends on its current location on the tape, on the contents of the corresponding cell and on the internal state of the machine. Based on the latter two elements, the transition function determines a new symbol-state pair as well as a movement instruction (i.e., "left" or "right" or "stay"). The machine modifies the contents of the said cell and its internal state accordingly, and moves as directed. That is, suppose that the machine is in state q and resides in a cell containing the symbol σ, and suppose that the transition function maps (σ, q) to (σ', q', D). Then, the machine modifies the contents of the said cell to σ', modifies its internal state to q', and moves one cell in direction D. Figure 1 shows a single step of a Turing machine that, when in state 'b' and seeing a binary symbol σ, replaces σ with the symbol $\sigma + 2$, maintains its internal state, and moves one position to the right.[2]

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

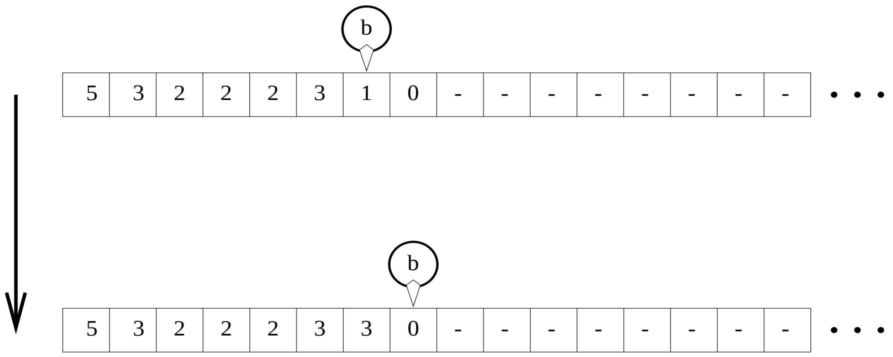


Figure 1: A single step by a Turing machine.

The initial environment (or configuration) of a Turing machine consists of the machine residing in the first (i.e., left-most) cell and being in its initial state. Typically, one also mandates that, in the initial configuration, a prefix of the tape's cells hold bit values, which concatenated together are considered the input, and the rest of the tape's cells hold a special symbol (which in Figure 1 is denoted by '-'). Once the machine halts, the output is defined as the contents of the cells that are to the left of its location (at termination time).[3] Thus, each machine defines a function mapping inputs to outputs, called the function computed by the machine.

Multi-tape Turing machines. We comment that in most expositions, one refers to the location of the "head of the machine" on the tape (rather than to the "location of the machine on the tape"). The standard terminology is more intuitive when extending the basic model, which refers to a single

[2]Figure 1 corresponds to a machine that, when in the initial state (i.e., 'a'), replaces the symbol σ by $\sigma + 4$, modifies its internal state to 'b', and moves one position to the right. Indeed, "marking" the leftmost cell (in order to allow for recognizing it in the future), is a common practice in the design of Turing machines.

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

tape, to a model that supports a constant number of tapes. In the model of multi-tape machines, each step of the machine depends and effects the cells that are at the head location of the machine on each tape. We mention that the extension of the model to multi-tape Turing machines is crucial to the definition of space complexity. A less fundamental advantage of the model of multi-tape Turing machines is that it allows for an easier design of machines that compute desired functions.

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

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

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

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

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

- $\texttt{reset}(r)$, where r is an index of a register, results in setting the value of register r to zero.
- $\texttt{inc}(r)$, where r is an index of a register, results in incrementing the content of register r. Similarly $\texttt{dec}(r)$ causes a decrement.

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

The program counter is incremented after the execution of each instruction, and the next instruction to be executed by the machine is the one to which the program counter points (and the machine halts if the program counter exceeds the program's length). The input to the machine may be defined as the contents of the first n memory cells, where n is placed in a special input register. We note that the RAM model satisfies the Church-Turing Thesis, but in order to make it closer to a real-life computer we augment the model by instructions like add(r_1, r_2) (resp., mult(r_1, r_2)), which results in adding (resp., multiplying) the contents of registers r_1 and r_2 and placing the result in register r_1. We suggest proving that this abstract RAM can be emulated by a Turing machine.[4] (Hint: note that during the emulation, we only need to hold the input, the contents of all registers, and the contents of the memory cells that were accessed during the computation.)[5]

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

1.3.2 Uncomputable functions

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

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

Theorem 4 (on the scarcity of computable functions): *The set of computable functions is countable, whereas the set of all functions* (from strings to string) *has cardinality* $\aleph$.

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

Proof: Since each computable function is computable by a machine that has a finite description, there is a 1-1 correspondence between the set of computable functions and the set of strings (which

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

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

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

The Halting Problem: In contrast to the preliminary discussion, at this point we consider also machines that may not halt on some inputs. (The functions computed by such machines are partial functions that are defined only on inputs on which the machine halts.) Again, we rely on the postulate that each machine in the model has a finite description, and denote the description of machine M by $\langle M \rangle \in \{0,1\}^*$. The halting function, $\mathtt{h} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$, is defined such that $\mathtt{h}(\langle M \rangle, x) \stackrel{\text{def}}{=} 1$ if and only if M halts on input x. The following result goes beyond Theorem 4 by pointing to an explicit function (of natural interest) that is not computable.

Theorem 5 (undecidability of the halting problem): *The halting function is not computable.*

The term undecidability means that the corresponding decision problem cannot be solved by an algorithm. That is, Theorem 5 asserts that the decision problem associated with the set $\mathtt{h}^{-1}(1) = \{(\langle M \rangle, x) : \mathtt{h}(\langle M \rangle, x) = 1\}$ is not solvable by an algorithm (i.e., there exists no algorithm that, given a pair $(\langle M \rangle, x)$, decides whether or not M halts on input x). Actually, the following proof shows that there exists no algorithm that, given $\langle M \rangle$, decides whether or not M halts on input $\langle M \rangle$.

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

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

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

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

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

Turing-reductions. The core of the second part of the proof of Theorem 5 is an algorithm that solves one problem (i.e., computes $\mathtt{d}'$) by using as a subroutine an algorithm that solves another problem (i.e., computes $\mathtt{h}$). In fact, the first algorithm is actually an algorithmic scheme that refers to a "functionally specified" subroutine rather than to an actual (implementation of such a) subroutine, which may not exist. Such an algorithmic scheme is called a Turing-reduction (i.e., we have Turing-reduced the computation of $\mathtt{d}'$ to the computation of $\mathtt{d}$, which in turn Turing-reduces to $\mathtt{h}$). The "natural" ("positive") meaning of a Turing-reduction of f' to f is that when given an algorithm for computing f we obtain an algorithm for computing f'. In contrast, the proof of Theorem 5 uses the "unnatural" ("negative") counter-positive: if (as we know) there exists no algorithm for computing $f' = \mathtt{d}'$ then there exists no algorithm for computing $f = \mathtt{h}$ (which is what we wanted to prove). Jumping ahead, we mention that resource-bounded Turing-reductions (e.g., polynomial-time reductions) play a central role in complexity theory itself, and again they are used mostly in a "negative" way. We will define such reductions and extensively use them throughout the course.

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

Theorem 6 (Rice's Theorem): *Let $\mathcal{F}$ be a non-trivial subset[6] of the set of all computable partial functions, and let $S_{\mathcal{F}}$ be the set of strings that describe machines that compute functions in $\mathcal{F}$. Then deciding membership in $S_{\mathcal{F}}$ cannot be solved by an algorithm.*

Theorem 6 can be proved by a Turing-reduction from $\mathtt{d}$. We do not provide a proof because this is too remote from the main subject matter of the course. We stress that Theorems 5 and 6 hold for any reasonable model of computation (referring both to the potential solvers and to the machines the description of which is given as input to these solvers). Thus, Theorem 6 means that *no algorithm can determine any non-trivial property of the function computed by a given computer program* (written in any programming language). For example, *no algorithm can determine whether or not a given computer program halt on each possible input.* The relevance of this assertion to the project of program verification is obvious.

The Post Correspondence Problem. We mention that undecidability arises also outside of the domain of questions regarding computing devices (given as input). Specifically, we consider the Post Correspondence Problem in which the input consists of two (equal length) sequences of strings, $(\alpha_1, ..., \alpha_k)$ and $(\beta_1, ..., \beta_k)$, and the question is whether or not there exists a sequence of indices

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

$i_1, ..., i_\ell \in \{1, ..., k\}$ such that $\alpha_{i_1} \cdots \alpha_{i_\ell} = \beta_{i_1} \cdots \beta_{i_\ell}$. (We stress that the length of this sequence is not bounded.)[7]

Theorem 7 *The Post Correspondence Problem is undecidable.*

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

1.3.3 Universal algorithms

So far we have used the postulate that, in any reasonable model of computation, each machine (or computation rule) has a finite description. Furthermore, we also used the fact that such model should allow for the easy modification of such descriptions such that the resulting machine computes an easily related function (see the proof of Theorem 5). Here we go one step further and postulate that the description of machines (in this model) is "effective" in the following natural sense: there exists an algorithm that, given a description of a machine (resp., computation rule) and a corresponding environment, determines the environment that results from performing a single step of this machine on this environment (resp. the effect of a single application of the computation rule). This algorithm can, in turn, be implemented in the said model of computation (assuming this model is general; see the Church-Turing Thesis). Successive applications of this algorithm leads to the notion of a universal machine, which (for concreteness) is formulated below in terms of Turing machines.

Definition 8 (universal machines): *A* universal Turing machine *is a Turing machine that on input a description of a machine M and an input x returns the value of $M(x)$ if M halts on x and otherwise does not halt.*

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

Theorem 9 *There exists a universal Turing machine.*

Theorem 9 asserts that the partial function $\mathtt{u}$ is computable. In contrast, it can be shown that any extension of $\mathtt{u}$ to a total function is uncomputable. That is, for any total function $\hat{\mathtt{u}}$ that agrees with the partial function $\mathtt{u}$ on all the inputs on which the latter is defined, it holds that $\hat{\mathtt{u}}$ is uncomputable.[8]

Proof: Given a pair $(\langle M \rangle, x)$, we just emulate the computation of machine M on input x. This emulation is straightforward, because by the effectiveness of the description of M, we can iteratively determine the next instantaneous configuration of the computation of M on input x. If the said

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

[8]The claim is easy to prove for the total function $\hat{\mathtt{u}}$ that extends $\mathtt{u}$ and assigns the special symbol $\perp$ to inputs on which $\mathtt{u}$ is undefined (i.e., $\hat{\mathtt{u}}(\langle M \rangle, x) \stackrel{\rm def}{=} \perp$ if $\mathtt{u}$ is not defined on $(\langle M \rangle, x)$ and $\hat{\mathtt{u}}(\langle M \rangle, x) \stackrel{\rm def}{=} \mathtt{u}(\langle M \rangle, x)$ otherwise). In this case $\mathtt{h}(\langle M \rangle, x) = 1$ if and only if $\hat{\mathtt{u}}(\langle M \rangle, x) \neq \perp$, and so the halting function $\mathtt{h}$ is Turing-reducible to $\hat{\mathtt{u}}$. In the general case, we may adapt the proof of Theorem 5 by observing that, for a machine M that halts on every input, it holds that $\hat{\mathtt{u}}(\langle M \rangle, x) = \mathtt{u}(\langle M \rangle, x)$ for every x (and in particular for $x = \langle M \rangle$).

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

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

In addition to their practical importance, the existence of universal machines (and their variants) has important consequences in the theories of computability and computational complexity. Here we merely note that Theorem 9 implies that many questions about the behavior of a universal machine on certain input types are undecidable. In particular, there is no algorithm that, given $X \stackrel{\rm def}{=} (\langle M\rangle, x)$, can tell whether or not a (fixed) universal machine halts on input X. Revisiting the proof of Theorem 7, it follows that the Post Correspondence Problem remains undecidable even if the sequences are restricted to have a specific length (i.e., k is fixed).

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

Theorem 10 (complexity w.r.t a universal machine): *Let U be a universal machine. Then, for every machine M', there exists a constant c such that $K_U(s) \leq K_{M'}(s) + c$ for every string s.*

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

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

   (Hint: apply Theorem 10 to the machine that computes the identity mapping.)

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

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

3. Some strings of length n have complexity at least n. Furthermore, for every n and i,

$$|\{s \in \{0,1\}^n : K(s) \leq n - i\}| < 2^{n-i+1}$$

   (Hint: different strings must have different descriptions with respect to U.)

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

1.3.4 Time and space complexity

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

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

The time complexity of a problem. As stated in the preface and in the introduction, complexity theory is typically unconcerned with the (time) complexity of a specific algorithm. It is rather concerned with the (time) complexity of a problem, assuming that this problem is solvable by an algorithm. Intuitively, the time complexity of such a problem is defined as the time complexity of the fastest algorithm that solves this problem (assuming that the latter term is well-defined).[9] More generally, we will be interested in upper and lower bounds on the (time) complexity of algorithms that solve the problem. However, the complexity of a problem may depend on the specific model of computation in which algorithms that solve it are implemented. The following Cobham-Edmonds Thesis asserts that the variation (in the time complexity) is not too big, and in particular is irrelevant for much of the current focus of complexity theory (e.g., for the P-vs-NP Question).

[9] **Advanced comment:** As we shall see in a future lecture (cf. Blum's Speed-up Theorem), the naive assumption that a "fastest algorithm" for solving a problem exists is not always justified. On the other hand, the assumption is justified in some important cases (see, e.g., the Optimal Algorithm for NP).

The Cobham-Edmonds Thesis. As stated above, the time complexity of a problem may depend on the model of computation. For example, deciding membership in the set $\{xx : x \in \{0,1\}^*\}$ can be done in linear-time on a two-tape Turing machine, but requires quadratic-time on a single-tape Turing machine.[10] On the other hand, any problem that has time complexity t in the model of multi-tape Turing machines, has complexity $O(t^2)$ in the model of single-tape Turing machines. The Cobham-Edmonds Thesis asserts that the time complexities in any two "reasonable and general" models of computation are polynomially related. That is, *a problem has time complexity t in some* "reasonable and general" *model of computation if and only if it has time complexity* $\text{poly}(t)$ *in the model of* (single-tape) *Turing machines.*

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

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

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

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

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

[10] Proving the latter fact is quite non-trivial. One proof is by a "reduction" from a communication complexity problem [10, Sec. 12.2]. Intuitively, a single-tape Turing machine that decides membership in the aforementioned set can be viewed as a channel of communication between the two parts of the input. Focusing our attention on inputs of the form $y0^n z0^n$, for $y, z \in \{0,1\}^n$, each time the machine passes from the first part to the second part it carries $O(1)$ bits of information (in its internal state) while making at least n steps. The proof is completed by invoking the linear lower bound on the communication complexity of the (two-argument) identity function (i.e, $\mathtt{id}(y,z) = 1$ if $y = z$ and $\mathtt{id}(y,z) = 0$ otherwise, cf. [10, Chap. 1]).

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

Universal machines, revisited. Time complexity yields an important variant of the universal function $\mathtt{u}$ (presented in §1.3.3). Define $\mathtt{u}'(\langle M\rangle, x, t) \stackrel{\text{def}}{=} y$ if on input x machine M halts within t steps and outputs the string y, and $\mathtt{u}'(\langle M\rangle, x, t) \stackrel{\text{def}}{=} \perp$ if on input x machine M makes more than t steps. Unlike $\mathtt{u}$, the function $\mathtt{u}'$ is computable. Furthermore, $\mathtt{u}'$ is computable by a machine U' that on input $X = (\langle M\rangle, x, t)$ halts after $\text{poly}(t)$ steps. Indeed, machine U' is a variant of a universal machine (i.e., on input X, machine U' merely emulates M for t steps rather than forever). Note that the number of steps taken by U' depends on the specific model of computation (and that some overhead is unavoidable because emulating each step of M requires reading the relevant portion of the description of M).

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

1.3.5 Oracle machines

The notion of Turing-reductions, which was discussed in §1.3.2, is captured by the definition of oracle machines. Loosely speaking, an oracle machine is a machine that is augmented such that it may pose questions to the outside. (A rigorous formulation of this notion is provided below.) We consider the case in which these questions, called queries, are answered consistently by some function $f : \{0,1\}^* \to \{0,1\}^*$, called the oracle. That is, if the machine makes a query q then the answer it obtains is $f(q)$. In such a case, we say that the oracle machine is given access to the oracle f. For an oracle machine M, a string x and a function f, we denote by $M^f(x)$ the output of M on input x when given access to the oracle f. (Re-examining the second part of the proof of Theorem 5, observe that we have actually described an oracle machine that computes $\mathtt{h}$ when given access to the oracle $\mathtt{d}'$.)

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

Definition 11 (using an oracle): *An* oracle machine *is a Turing machine with an additional tape, called the* oracle tape*, and two special states, called* oracle invocation *and* oracle spoke*. The* computation of the oracle machine M on input x and access to the oracle $f : \{0,1\}^* \to \{0,1\}^*$ *is defined based on the successive configuration function. For configurations with state different from* oracle invocation *the next configuration is defined as usual. Let γ be a configuration in which the machine's state is* oracle invocation *and suppose that the actual contents of the oracle tape is q (i.e., q is the contents of the maximal prefix of the tape that holds bit values).*[11] *Then, the configuration following γ is identical to γ, except that the state is* oracle spoke*, and the actual contents of the oracle tape is $f(q)$. The string q is called M's* query *and $f(q)$ is called the* oracle's reply*.*

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

1.3.6 Restricted models

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

In our opinion, the most important motivation for the study of these restricted models of computation is that they provide simple models for some natural (or artificial) phenomena. This motivation, however, seems only remotely related to the study of the complexity of various computational tasks. Thus, in our opinion, the study of these restricted models (e.g., any of the lower levels of Chomsky's Hierarchy [7, Chap. 9]) should be decoupled from the study of computability theory (let alone the study of complexity theory).

1.4 Non-uniform Models (Circuits and Advice)

By a non-uniform model of computation we mean a model in which for each possible input length one considers a different computing device. That is, there is no "uniformity" requirement relating devices that correspond to different input lengths. Furthermore, this collection of devices is infinite by nature, and (in absence of a uniformity requirement) this collection may not even have a finite description. Nevertheless, each device in the collection has a finite description. In fact, the relationship between the size of the device (resp., the length of its description) and the length of the input that it handles will be of major concern. The hope is that the finiteness of all parameters (which refer to a single device in such a collection) will allow for the application of combinatorial techniques to analyze the limitations of certain settings of parameters.

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

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

We will focus on two related models of non-uniform computing devices: Boolean circuits (§1.4.1) and "machines that take advice" (§1.4.2). The former model is more adequate for the study of the evolution of computation (i.e., development of lower-bound techniques), whereas the latter is more adequate for modeling purposes (e.g., upper-bounding the ability of efficient algorithms).

1.4.1 Boolean Circuits

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

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

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

2. The graph sources (i.e., vertices with no in-going edges) are called input terminals. Each input terminal is labeled by a natural number (which is to be thought of the index of an input variable). (For sake of defining formulae, we allow different input terminals to be labeled by the same number.)[12]

3. The graph sinks (i.e., vertices with no out-going edges) are called output terminals, and we require that they have in-degree 1. Each output terminal is labeled by a natural number such that if the circuit has m output terminals then they are labeled $1, 2, ..., m$. That is, we disallow different output terminals to be labeled by the same number, and insist that the labels of the output terminals are consecutive numbers. (Indeed, the labels of the output terminals will correspond to the indices of locations in the circuit's output.)

For sake of simplicity, we also mandate that the labels of the input terminals are consecutive numbers.[13]

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

[13]This convention slightly complicates the construction of circuits that ignore some of the input values. Specifically, we use artificial gadgets that have in-coming edges from the corresponding input terminals, and compute an adequate constant. To avoid having this constant as an output terminal, we feed it into an auxiliary gate such that the value of the latter is determined by the other in-going edge (e.g., a constant 1 fed into an $\vee$-gate). See example of dealing with x_3 in Figure 2.

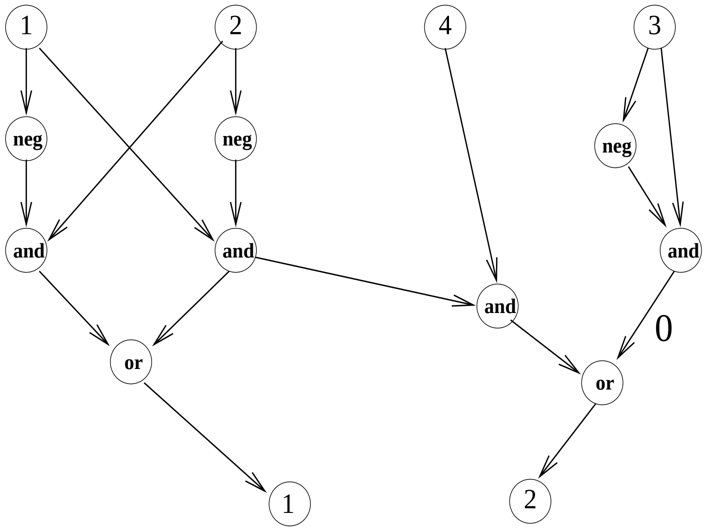


Figure 2: A circuit computing $f(x_1, x_2, x_3, x_4) = (x_1 \oplus x_2, x_1 \wedge \neg x_2 \wedge x_4)$.

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

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

  Indeed, the hypothesis that the circuit is acyclic implies that the process of determining values for the circuit's vertices is well-defined: As long as the value of some vertex is undetermined, there exists a vertex such that its value is undetermined but the values of all its children are determined. Thus, the process can make progress, and terminates when the values of all vertices (including the output terminals) are determined.

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

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

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

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

The circuit complexity of functions. We highlight some simple facts about the circuit complexity of functions. (These facts are in clear correspondence to facts regarding Kolmogorov Complexity mentioned in §1.3.3.)

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

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

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

   (Hint: consider a Turing machine that computes the function, and consider its computation on a generic n-bit long input. The corresponding computation can be emulated by a circuit that consists of $t(n)$ layers such that each layer represents an instantaneous configuration of the machine, and the relation between consecutive configurations is captured by ("uniform") local gadgets in the circuit. For further details see the proof of Theorem 33, which presents a similar emulation.)

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

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

Note that the first fact implies that families of circuits can compute functions that are uncomputable by algorithms. Furthermore, this phenomenon occurs also when restricting attention to families of polynomial-size circuits. See further discussion in §1.4.2.

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

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

1.4.2 Machines that take advice

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

Definition 12 (taking advice): *We say that* algorithm A computes the function f using advice of length $\ell : \mathbb{N} \to \mathbb{N}$ *if there exists an infinite sequence $(a_n)_{n\in\mathbb{N}}$ such that*

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

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

Note that any function having circuit complexity s can be computed using advice of length $O(s \log s)$, where the log factor is due to the fact that a graph with v vertices and e edges can be described by a string of length $2e \log_2 v$. Note that the model of machines that use advice allows for some sharper bounds than the ones stated in §1.4.1: every function can be computed using advice of length ℓ such that $\ell(n) = 2^n$, and some uncomputable functions can be computed using advice of length 1.

Theorem 13 (the power of advice): *There exist functions that can be computed using one-bit advice but cannot be computed without advice.*

Proof: Taking any uncomputable Boolean function $f : \mathbb{N} \to \{0,1\}$, consider the function f' defined as $f'(x) = f(|x|)$. Note that f is Turing-reducible to f' (e.g., on input n make any n-bit query to f', and return the answer).[14] Thus, f' cannot be computed without advice. On the other hand, f' can be easily computed by using the advice sequence $(a_n)_{n\in\mathbb{N}}$ such that $a_n = f(n)$; that is, the algorithm merely outputs the advice bit (and indeed $a_{|x|} = f(|x|) = f'(x)$, for every $x \in \{0,1\}^*$). ■

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

1.4.3 Restricted models

As noted in §1.4.1, the model of Boolean circuits allows for the introduction of many natural subclasses of computing devices. Following is a laconic review of a few of these subclasses. We will refer to various types of Boolean formulae in the rest of this course, and thus suggest not to skip the following two paragraphs.

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

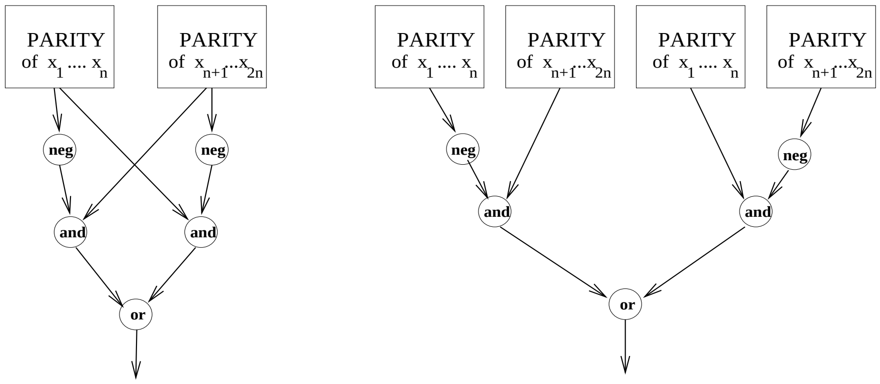


Figure 3: Recursive construction of parity circuits and formulae.

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

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

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

1.5 Complexity Classes

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

1. A type of computational problems (see Section 1.2). Indeed, most classes refer to decision problems, but classes of search problems, promise problems, and other types of problems will also be considered.

2. A model of computation, which may be either uniform (see Section 1.3) or non-uniform (see Section 1.4).

3. A complexity measure and a function (or a set of functions), which put together limit the class of computations of the previous item; that is, we refer to the class of computations that have complexity not exceeding the specified function (or set of functions). For example, in §1.3.4, we mentioned time complexity and space complexity, which apply to any uniform model of computation. We also mentioned polynomial-time computations, which are computations in which the time complexity (as a function) does not exceed some polynomial (i.e., a member of the set of polynomial functions).

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

2 The P versus NP Question

Our daily experience is that it is harder to solve a problem than it is to check the correctness of a solution. Is this experience merely a coincidence or does it represent a fundamental fact of life (or property of the world)? This is the essence of the P versus NP Question, where *P represents search problems that are efficiently solvable and NP represents search problems for which solutions can be efficiently checked.* Another natural question captured by the P versus NP Question is whether proving theorems is harder that verifying the validity of these proofs. In other words, the question is whether deciding membership in a set is harder than being convinced of this membership by an adequate proof. In this case, *P represents decision problems that are efficiently solvable, whereas NP represents sets that have efficiently checkable proofs of membership.* These two meanings of the P versus NP Question are rigorously presented and discussed in Sections 2.1 and 2.2, respectively. The equivalence of the two formulations is shown in Section 2.3, and the common belief that P is different from NP is further discussed in Section 2.5.

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

The notion of efficient computation. Recall that we associate efficient computation with polynomial-time algorithms.[15] Furthermore, the latter class is independent of the specific model of computation, as long as the latter is "reasonable" (cf. the Cobham-Edmonds Thesis). Both issues are discussed in §1.3.4.

A note on the representation of problem instances. As noted in Section 1.2, many natural (search and decision) problems are captured more naturally by the terminology of promise problems, where the domain of possible instances is a subset of $\{0,1\}^*$ rather than $\{0,1\}^*$ itself. 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 are possible instances). Typically, in these cases, the set of possible instances (e.g., encodings of graphs) is efficiently recognizable (i.e., membership in it can be decided in polynomial-time). Thus, artificially extending the set of instances to the set of all possible strings (and allowing trivial solutions for the corresponding dummy instances) does not change the complexity of the original problem. We further discuss this issue in a future lecture.

2.1 The search version: finding versus checking

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

Much of computer science is concerned with solving various search problems (as in Definition 1). Examples of such problems include finding solutions to systems of linear (or polynomial) equations, finding a spanning tree in a graph, finding a short traveling salesman tour in a metric space,

[15] **Advanced comment:** Here, 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 more advanced lectures. We stress that the most important facts and questions that are addressed in the current lecture hold also with respect to probabilistic polynomial-time algorithms.

and finding a scheduling of jobs to machines such that various given constraints are satisfied. Furthermore, search problems correspond to the daily notion of "solving problems" and thus are of natural general interest. In the current section, we will consider the question of *which search problems can be solved efficiently.*

One type of search problems that cannot be solved efficiently consists of search problems for which the solutions are too long in terms of the problem's instances. In such a case, merely typing the solution amounts to an activity that is deemed inefficient. Thus, we focus our attention on search problems that are not in this class. That is, we consider only search problems in which the length of the solution is bounded by a polynomial in the length of the instance. Recalling that search problems are associated with binary relations (see Definition 1), we focus our attention on polynomially bounded relations.

Definition 14 (polynomially bounded relations): *We say that $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is* polynomially-bounded *if there exists a polynomial p such that for every $(x,y) \in R$ it holds that $|y| \leq p(|x|)$.*

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

2.1.1 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 2.3 and 3.3.) The following definition refers to the formulation of solving search problems provided in Definition 1.

Definition 15 (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) requires that in this case $A(x) = \perp$. Thus, algorithm A always outputs a correct answer, which is a valid solution in the case that such a solution exists or an indication that no solution exists.

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

2.1.2 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 2.3–2.4 and 3.3.)

Definition 16 (search problems with efficiently checkable solutions):

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

The class $\mathcal{PC}$ contains thousands of natural problems (e.g., finding a traveling salesman tour of length that does not exceed a given threshold, finding the prime factorization of a given composite, etc). In each of these natural problems, it is easy to see that the correctness of solutions can be checked efficiently (e.g., given a traveling salesman tour it is easy to compute its length and check whether or not it exceeds the given threshold).[16]

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

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

complexity of finding such solutions. We warn, however, that $\mathcal{PF}$ contains (unnatural) problems that are not in $\mathcal{PC}$ (see Exercise 41).

2.1.3 The P versus NP question in terms of search problems

Is it the case that every search problem in $\mathcal{PC}$ is in $\mathcal{PF}$? That is, if R has efficiently checkable solutions then is it necessarily the case that the search problem of R can be solved efficiently? In other words, if it is *easy to check* whether or not a given solution for a given instance is correct then is it also *easy to find* a solution to a given instance?

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

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

2.2 The decision version: proving versus verifying

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

2.2.1 The class P as a natural class of decision problems

Properties of objects are modeled as subsets of the set of all possible objects (i.e., a property is associated with the set of objects having this property). Thus, we focus on deciding membership in sets (as in Definition 2). We consider the class of decision problems that are efficiently solvable, a class that is traditionally denoted $\mathcal{P}$ (standing for Polynomial-time). The following definition refers to the formulation of solving decision problems provided in Definition 2.

Definition 17 (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 15, we have defined a fundamental class of problems, which contains many natural problems (e.g., determining whether or not a given graph is connected), but we do not have a good understanding regarding its scope (i.e., which problems this class contains). Specifically, there are many natural problems that are not known to reside in $\mathcal{P}$, and a natural class containing a host of such problems is presented next.

2.2.2 The class NP and NP-proof systems

(The origin of the notation NP will be discussed in Section 2.4.) We view NP as the class of decision problems that have efficiently verifiable proof systems. Loosely speaking, we say that a set S has a proof system if instances in S have valid proofs of membership (i.e., proofs accepted as valid by the system), whereas instances not in S have no valid proofs. Indeed, proofs are defined as strings that (when accompanying the instance) are accepted by the (efficient) verification procedure. We say that V is a verification procedure for membership in S if it satisfies the following two conditions:

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

We note that the soundness condition captures the "security" of the verification procedure; that is, its ability not to be fooled by anything into proclaiming a wrong assertion. The completeness condition captures the "viability" of the verification procedure; that is, its ability to be convinced of any valid assertion, when presented with an adequate proof. (We stress that, in general, proof systems are defined in terms of their verification procedures, which must satisfy adequate completeness and soundness conditions.) Our focus here is on efficient verification procedures that utilize relatively short proofs (i.e., proofs that are of length that is polynomially bounded by the length of the corresponding assertion).[17]

Definition 18 (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:*

[17] **Advanced comment:** In continuation to Footnote 15, we note that in this lecture we consider *deterministic* (polynomial-time) verification procedures, and consequently the completeness and soundness conditions that we state here are error-less. In contrast, in a future lecture, 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 lecture, 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.

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 42), and that in some cases the difference is not artificial (see Exercise 43).

> **Teaching note:** Using Definition 18, it is typically easy to show that natural decision problems are in $\mathcal{NP}$. All that is needed is designing adequate NP-proofs of membership, which is typically quite straightforward and natural, because such decision problems are typically phrased as asking about the existence of a structure (or object) that can be easily verified as valid. For example, SAT is defined as the set of satisfiable Boolean formulae, which means asking about the existence of satisfying assignments. Indeed, we can efficiently check whether a given assignment satisfies a given formula, which means that we have (a verification procedure for) an NP-proof system for SAT.

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

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

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

2.2.3 The P versus NP question in terms of decision problems

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

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

2.3 Equivalence of the two formulations

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

Theorem 19 *$\mathcal{PC} \subseteq \mathcal{PF}$ 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 18). Then $R \stackrel{\text{def}}{=} \{(x,y) : V(x,y) = 1\}$ is a polynomially bounded relation in $\mathcal{PC}$, and by the hypothesis its search problem is solvable in polynomial time (i.e., $R \in \mathcal{PC} \subseteq \mathcal{PF}$). Denoting by A the polynomial-time algorithm solving the search problem of R, we decide membership in S in the obvious way. That is, on input x, we output 1 if and only if $A(x) \neq \perp$, where the latter event holds if and only if $A(x) \in R(x)$, which in turn occurs if and only if $R(x) \neq \emptyset$ (equiv., $x \in S$). Thus, $\mathcal{NP} \subseteq \mathcal{P}$ (and $\mathcal{NP} = \mathcal{P}$) follows.

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

Reflection: The first part of the proof of Theorem 19 associates with each set S in $\mathcal{NP}$ a natural relation R (in $\mathcal{PC}$). Specifically, R consists of all pairs (x, y) such that y is an NP-witness for membership of x in S. Thus, the search problem of R consists of finding such an NP-witness, when given x as input. Indeed, R is called the witness relation of S, and solving the search problem of R allows to decide membership in S. In the second part of the proof, we associate with each $R \in \mathcal{PC}$ a set S'_R (in $\mathcal{NP}$), but S'_R is not the natural set $S_R \stackrel{\text{def}}{=} \{x : \exists y \text{ s.t. } (x, y) \in R\}$ (which gives rise to R as its witness relation). Specifically, S'_R consists of strings that encode pairs (x, y') such that y' is a prefix of some string in $R(x) = \{y : (x, y) \in R\}$. Thus, deciding membership in S'_R allows to solve the search problem of R.

Conclusion: Theorem 19 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.

2.4 The traditional definition of NP

Unfortunately, Definition 18 is not the commonly used definition of $\mathcal{NP}$. Traditionally, $\mathcal{NP}$ is defined as the class of sets that can be decided by a *fictitious* device called a non-deterministic polynomial-time machine (which explains the source of the notation NP). The reason that this class of fictitious devices is important is because it captures (indirectly) the definition of NP-proofs. Since the reader may come across the traditional definition of $\mathcal{NP}$ when studying different works, the author feels obliged to provide the traditional definition as well as a proof of its equivalence to Definition 18.

Definition 20 (non-deterministic polynomial-time Turing machines):

- *A* non-deterministic Turing machine *is define as in §1.3.1, except that the transition function maps symbol-state pairs to subsets of triples* (rather than to a single triple) *in* $\Sigma \times Q \times \{-1, 0, +1\}$. *Accordingly, the configuration following a specific instantaneous configuration may be one of several possibilities, each determine by a different possible triple. Thus, the* computations of a non-deterministic machine *on a* (fixed) *given input may result in different outputs.*

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

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

We stress that Definition 20 makes no reference to the number (or fraction) of possible computations of the machine that yield a specific output (on a specific input).[18] Definition 20 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 20 is equivalent to Definition 18).

> **Teaching note:** Whether or not Definition 20 is elegant is a matter of taste. For sure, many students find Definition 20 quite confusing, possibly because they assume that it represents some natural model of computation and allow themselves to be fooled by their intuition regarding such models. (Needless to say, the students' intuition regarding computation is irrelevant when applied to a fictitious model.)

Note that, unlike other definitions in this lecture, Definition 20 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 44).

Theorem 21 *Definition 18 is equivalent to Definition 20. That is, a set S has an NP-proof system if and only if there exists a non-deterministic polynomial-time machine that accepts S.*

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

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

2.5 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.*

[18] **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.

Immanuel Kant (1724–1804)

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

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

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

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

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

2.6 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 mean that the validity of y can be determined without reading all of it (which means that some substring of y can be used as the effective y in the original definitions).

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

3 Polynomial-time Reductions

We present the general notion of (polynomial-time) reductions among computational problems, and view the notion of a "Karp-reduction" as an important special case that suffices (and is more convenient) in many cases. Such reductions play a key role in the theory of NP-completeness, which is the topic of Section 4, but we stress that the fundamental nature of the notion of a reduction and highlight two specific applications (i.e., reducing search and optimization problems to decision problems). Furthermore, in the latter applications, it will be important to use the general notion of a reduction (i.e., "Cook-reduction" rather than "Karp-reduction").

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

3.1 The general notion of a reduction

Reductions are procedures that use "functionally specified" subroutines. That is, the functionality of the subroutine is specified, but its operation remains unspecified and its running-time is counted at unit cost. Analogously to algorithms, which are modeled by Turing machines, reductions can be modeled as *oracle* (Turing) machines. A reduction solves one computational problem (which may be either a search or 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 §1.3.1 and §1.3.5. These reductions, called Turing-reductions (cf. §1.3.1) and modeled by oracle machines (cf. §1.3.5), 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 11) that run in time polynomial in the length of their input. We stress that the running time of an oracle machine is the number of steps made during its computation, and that the oracle's reply on each query is obtained in a single step.

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

Definition 22 (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 45 for other properties of Cook-reductions.) Also observe that the second part of the proof of Theorem 19 is actually a Cook-reduction of the search problem of any R in $\mathcal{PC}$ to a decision problem regarding a related set S'_R in $\mathcal{NP}$. Thus, that proof establishes that *any search problem in $\mathcal{PC}$ is Cook-reducible to some decision problem in $\mathcal{NP}$*. We shall see a tighter relation between search and decision problems in Section 3.3 (i.e., R will be reduced to $S_R = \{x : \exists y \text{ s.t. } (x,y)\in R\}$ rather than to $S'_R = \{(x,y') : \exists y'' \text{ s.t. } (x,y'y'')\in R\}$).

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

Definition 23 (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 47 and 66).

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

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

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

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

3.2 Reducing optimization problems to search problems

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

1. *Exceeding a threshold*: Given a pair (x, v) the task is to find $y \in R(x)$ such that $f(x, y) \geq v$, where $R(x) = \{y : (x, y) \in R\}$. That is, we are actually referring to the search problem of the relation
$$R_f \stackrel{\text{def}}{=} \{(\langle x, v\rangle, y) : (x, y) \in R \land f(x, y) \geq v\}, \tag{1}$$
where $\langle x, v\rangle$ denotes a string that encodes the pair (x, v).

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

Examples of value functions include the size of a clique in a graph, the amount of flow in a network (with link capacities), etc. The task may be to find a clique of size exceeding a given threshold in a given graph or to find a maximum-size clique in a given graph. Note that, in these examples, the "base" search problem (i.e., the relation R) is quite easy to solve, and the difficulty arises from the auxiliary condition on the value of a solution (presented in R_f and R'_f). Indeed, one may trivialize R (i.e., let $R(x) = \{0,1\}^{\mathrm{poly}(|x|)}$ for every x), and impose all necessary structure by the function f (see Exercise 48).

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 25 *For any polynomial-time computable $f : \{0,1\}^* \times \{0,1\}^* \to \mathbb{R}$ and a polynomially bounded binary relation R, let R_f and R'_f be as in Eq.* (1) *and Eq.* (2)*, respectively. Then the search problems of R_f and R'_f are computationally equivalent.*

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

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

Turning to the opposite direction, we reduce the search problem of R_f to the search problem of R'_f by first finding the optimal value $v_x = \max_{y \in R(x)}\{f(x, y)\}$ (by binary search on its possible values), and next finding a solution of value (at least) v_x. In both steps, we use oracle calls to R_f. For simplicity, we assume that f assigns positive integer values, and let $\ell = \text{poly}(|x|)$ 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 any only if $R_f(\langle x, v\rangle) = \emptyset$, which in turn is indicated by the oracle answer $\perp$ (to the query $\langle x, v\rangle$). Making ℓ queries, we determine v_x (see Exercise 49). Note that in case $R(x) = \emptyset$, all answers will indicate that $R_f(\langle x, v\rangle) = \emptyset$, which we treat as if $v_x = 0$. Finally, we make the query (x, v_x), and halt returning the oracle's answer. ■

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

On the complexity of R_f and R'_f. We focus on the natural case in which $R \in \mathcal{PC}$ and f is polynomial-time computable. In this case, Theorem 25 implies that R_f and R'_f are computationally equivalent. A closer look reveals, however, that $R_f \in \mathcal{PC}$ always holds, whereas $R'_f \in \mathcal{PC}$ does *not* necessarily hold. That is, the problem of finding a solution (for a given instance) that exceeds a given threshold is in the class $\mathcal{PC}$, whereas the problem of finding an optimal solution is not necessarily in the class $\mathcal{PC}$. For example, the problem of finding a clique of a given size K in a given graph G is in $\mathcal{PC}$, whereas the problem of finding a maximum size clique in a given graph G is not known (and quite unlikely) to be in $\mathcal{PC}$ (although it is Cook-reducible to $\mathcal{PC}$). Indeed, the class of problems that are reducible to $\mathcal{PC}$ is a natural and interesting class (which is contained in Σ_2); indeed, for every $R \in \mathcal{PC}$ and polynomial-time computable f, the former class contains R'_f.

3.3 Self-reducibility of search problems

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

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

Definition 26 (the decision implicit in a search and self-reducibility): *The* decision problem implicit the search problem of R *is deciding membership in the set* $S_R = \{x : R(x) \neq \emptyset\}$*, where* $R(x) = \{y : (x,y) \in R\}$*. The search problem of* R *is called* self-reducible *if it can be reduced to the decision problem of* S_R*.*

Note that the search problem of R and the problem of deciding membership in S_R refer to the same instances: The search problem requires finding an adequate solution (i.e., given x find $y \in R(x)$), whereas the decision problem refers to the question of whether such solutions exist (i.e., given x answer whether or not $R(x)$ is non-empty). Thus, S_R is really the "decision problem implicit in R," because it is a decision problem that one implicitly solves when solving the search problem of R. Indeed, *the decision problem of* S_R *is always reducible to the search problem for* R[19] (and if R is in $\mathcal{PC}$ then S_R is in $\mathcal{NP}$). It follows that *if a search problem* R *is self-reducible then it is computationally equivalent to the decision problem* S_R.

Self-reducibility means a reduction of the search problem to the decision problem implicit in it. We shall see that self-reducibility is a property of many natural search problems (including all NP-complete search problems). This justifies the relevance of decision problems to search problems in a stronger sense than established in Section 2.3: Recall that in Section 2.3 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, the set of satisfiable Boolean formulae (in CNF), and consider the search problem in which given a formula one should provide a truth assignment that satisfies it. The corresponding relation is denoted $R_{\mathtt{SAT}}$; that is, $(\phi, \tau) \in R_{\mathtt{SAT}}$ if τ is a satisfying assignment to the formulae ϕ. The decision problem implicit in $R_{\mathtt{SAT}}$ is indeed SAT. Note that $R_{\mathtt{SAT}}$ is in $\mathcal{PC}$ (i.e., it is polynomially-bounded and membership of (ϕ, τ) in $R_{\mathtt{SAT}}$ is easy to decide (by evaluating a Boolean expression)).

Proposition 27 ($R_{\mathtt{SAT}}$ is self-reducible): *The search problem of* $R_{\mathtt{SAT}}$ *is reducible to* SAT.

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

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

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

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

We wish to highlight a key point that has been blurred in the foregoing description. Recall that the formula ϕ' is obtained by replacing some variables by constants, which means that ϕ' *per se* contains Boolean variables as well as Boolean constants. However, the standard definition of SAT disallows Boolean constants in its instances.[20] Nevertheless, ϕ' can be simplified such that the resulting form contains no Boolean constants. This simplification is according to the straightforward boolean rules: That is, the constant `false` can be omitted from any clause, but if a clause contains only occurrences of the constant `false` then the entire formula simplifies to `false`. Likewise, if the constant `true` appears in a clause then the entire clause can be omitted, but if all clauses are omitted then the entire formula simplifies to `true`.) Needless to say, if the simplification process yields a Boolean constant then we may skip the query, and otherwise we just use the simplified form of ϕ' as our query. ■

Reductions analogous to the one used in the proof of Proposition 27 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 50 and 51, respectively.

Reflection: The proof of Proposition 27 (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 19 and is implicit in the proof of Proposition 27.

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

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

For an arbitrary $R \in \mathcal{PC}$, deciding membership in S'_R is not necessarily reducible to deciding membership in S_R. Furthermore, deciding membership in S'_R is not necessarily reducible to the search problem of R. (See Exercises 52 and 53.)

[20] While the problem seems rather technical at the current setting (as it merely amounts to whether or not the definition of SAT allows Boolean constants in its instances), it is far from being so technical in other cases (see Exercises 50 and 51).

Teaching note: In the rest of this section, we assume that the students have heard of NP-completeness. Specifically, all we need the students to know is that a set S is $\mathcal{NP}$-complete if $S \in \mathcal{NP}$ and every set in $\mathcal{NP}$ is reducible to S. Yet, the teacher may prefer postponing the presentation of the following material to Section 4.1 (or even to a later stage).

Advanced comment: In general, self-reducibility is a property of the search problem and not of the decision problem implicit in it. Assuming that $\mathcal{P} \neq \mathcal{NP}$, there exists relations $R_1, R_2 \in \mathcal{PC}$ having the same implicit-decision problem (i.e., $\{x : R_1(x) \neq \emptyset\} = \{x : R_2(x) \neq \emptyset\}$) such that R_1 is self-reducible but R_2 is not (see Exercise 54). However, this phenomenon does not arise when NP-complete problems are involved; that is, *all search problems that refer to finding NP-witnesses for any NP-complete decision problem are self-reducible.*

Theorem 28 *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 27, 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 27 (as well as being explicit in the proof of Theorem 19).

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. ■

4 NP-Completeness

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

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

4.1 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 29 (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 30 (NP-completeness of search problems, restricted notion): *A binary relation R is $\mathcal{PC}$-complete if it is in $\mathcal{PC}$ and every relation in $\mathcal{PC}$ is Levin-reducible to R.*

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

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

4.2 The existence of NP-complete problems

We suggest not to confuse the mere existence of NP-complete problems, which is remarkable by itself, with the even more remarkable existence of "natural" NP-complete problems. 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 31 *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 $S_{\mathtt{u}}$ and $R_{\mathtt{u}}$ (and defined next), which are used to derive the simplest proof of the current theorem.

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

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

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

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

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

$$x \mapsto f(x) \stackrel{\text{def}}{=} \langle M_R, x, 1^{t_R(|x|+p_R(|x|))} \rangle. \tag{3}$$

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

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

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

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

Bounded Halting and Non-Halting

We note that the problem shown to be NP-complete in the proof of Theorem 31 is related to the following two problems, called `Bounded Halting` and `Bounded Non-Halting`. Fixing any programming language, the instance to each of these problems consists of a program π and a time bound t (presented in unary). The decision version of `Bounded Halting` (resp., `Bounded Non-Halting`) consists of determining whether or not there exists an input (of length at most t) on which the program π halts (resp., does *not* halt) in t steps, whereas the search problem consists of finding such an input.

Thus, the decision version of `Bounded Non-Halting` refers to a fundamental computational problem in the area of program verification; specifically, the question of whether a given program halts within a given time-bound on all inputs of a given length.[21] 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 58). The fact that `Bounded Non-Halting` is probably intractable (i.e., is intractable provided that $\mathcal{P}\neq\mathcal{NP}$) is even more relevant to the project of program verification than the fact that the Halting Problem is undecidable. The reason being that the latter problem (as well as other related undecidable problems) refers to arbitrarily long computations, whereas the former problem refers to computations of explicitly bounded number of steps. Specifically, `Bounded Non-Halting` is concerned with the existence of an input that causes the program to violate a certain condition (i.e., halting) within a given time-bound.

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

4.3 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 [6]).

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

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

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

Proposition 32 *If an NP-complete problem* Π *is reducible to some problem* Π' *in NP then* Π' *is NP-complete. Furthermore, reducibility via Karp-reductions* (resp., Levin-reductions) *is preserved.*

Proof: The proof boils down to asserting the transitivity of reductions. That is, every problem in NP is reducible to Π, which in turn is reducible to Π'. Thus, by transitivity of reduction (see Exercise 46), every problem in NP is reducible to Π', which means that Π' is NP-hard and the proposition follows. ■

4.3.1 Circuit and formula satisfiability: CSAT and SAT

We consider two related computational problems, CSAT and SAT, which refer (in the decision version) to the satisfiability of circuits and formulae, respectively. (We refer the reader to the definition of Boolean circuits, formulae and CNF formulae that appear in §1.4.1.)

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

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

Theorem 33 (NP-completeness of CSAT): *The set* (resp., relation) $\mathtt{CSAT}$ (resp., $R_{\mathtt{CSAT}}$) *is* $\mathcal{NP}$*-complete* (resp., $\mathcal{PC}$-complete).

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

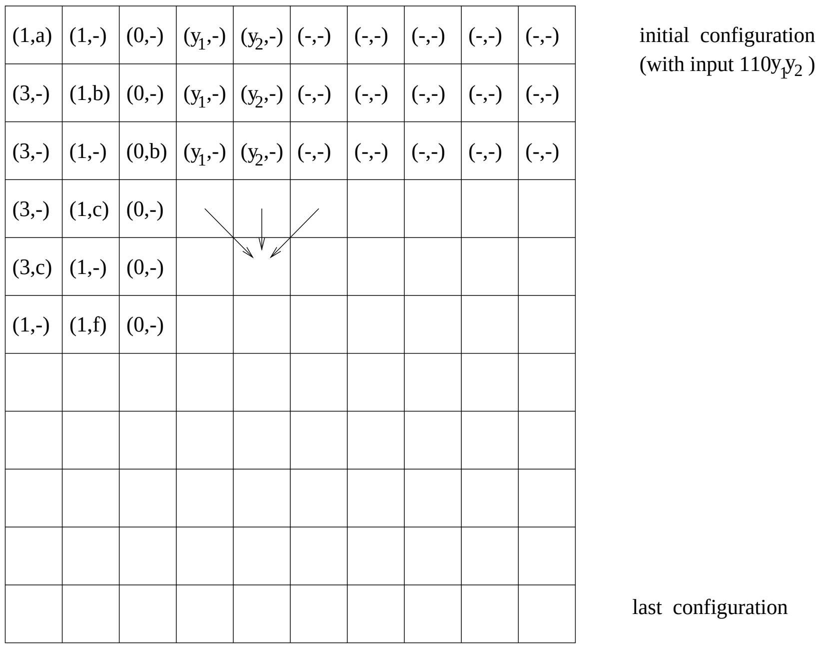
We will present (again, but for the last time in this text) a generic reduction, this time of any NP-problem to CSAT. The reduction is based on the observation, mentioned in §1.4.1, that the computation of polynomial-time algorithms can be emulated by polynomial-size circuits. In the current context, we wish to emulate the computation of a fixed machine M on input (x,y), *where x is fixed and y varies* (but $|y| = \mathrm{poly}(|x|)$ and the total number of steps of $M(x,y)$ is polynomial in $|x|+|y|$). Thus, x will be "hard-wired" into the circuit, whereas y will serve as the input to the circuit. The circuit itself, denoted C_x, will consists of "layers" such that each layer represents an instantaneous configuration of the machine M, and the relation between consecutive configurations in a computation of this machine is captured by ("uniform") local gadgets in the circuit. The number of layers will depend on the polynomial that upper-bounds the running-time of M, and an additional gadget will be used to detect whether the last configuration is accepting. Thus, only the first layer of the circuit C_x will depend on x. The punch-line is that determining whether, for a given x, there exists a y ($|y| = \mathrm{poly}(|x|)$) 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 31), 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 31 (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 $f(x) \stackrel{\mathrm{def}}{=} C_x$, such that $C_x(y) = 1$ if and only if M_R accepts the input (x,y) within $t_R(|x|+p_R(|x|))$ steps. Thus, it will follow that $x \in S$ if and only if there exists $y \in \{0,1\}^{p_R(|x|)}$ such that $C_x(y) = 1$ (i.e., if and only if $C_x \in$ CSAT). The circuit C_x will depend on x as well as on M_R, p_R and t_R. (We stress that M_R, p_R and t_R are fixed, whereas x is varies and thus explicit in our notation.)

Before describing the circuit C_x, let us consider a possible computation of M_R on input (x,y), where x is fixed and y represents a generic string of length at most $p_R(|x|)$. Such a computation proceeds for $t = t_R(|x| + p_R(|x|))$ steps, and corresponds to a sequence of $t+1$ instantaneous configurations, each of length t. Each such configuration can be encoded by t pairs of symbols, where the first symbol in each pair indicates the contents of a cell and the second symbol indicates either a state of the machine or that the machine is not located in this cell. Thus, each pair is a member of $\Sigma \times (Q \cup \{\perp\})$, where Σ is the finite "work alphabet" of M_R, Q is its finite set of internal states, and $\perp$ is an indication that the machine is not present at a cell. The initial configuration includes xy as input, and the decision of $M_R(x,y)$ can be read from (the leftmost cell of) the last configuration.[22] 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 59). 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 4.

The circuit C_x has a structure that corresponds to the aforementioned array. Each entry in the array is represented by a *constant* number of gates such that when C_x is evaluated at y these gates will be assigned values that encode the contents of the said entry. In particular, the entries of the first row of the array are "encoded" by hard-wiring the reduction's input (i.e., x), and feeding the circuit's input (i.e., y) to the adequate input terminals. That is, the circuit has $p_R(|x|)$ ("real") input terminals, and the hard-wiring of constants to other gates that represent the first row is done by a simple gadget (as in Figure 2). Indeed, additional hard-wiring in the first row corresponds to the other fixed elements of the initial configuration (i.e., the blank symbols and the encoding

[22] We refer to the output convention presented in §1.3.1, by which the output is written in the leftmost cells and the machine halts at the cell to its right.



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

Figure 4: An array representing ten computation steps on input $110y_1y_2$.

of the initial state and the initial location; cf. Figure 4). The entries of subsequent rows will be "encoded" (or rather computed at evaluation time) by using *constant-size* circuits that determine the value of an entry based on the three relevant entries in the row above it. Recall that each entry is encoded by a constant number of gates, and thus these constant-size circuits merely compute the constant-size function described in Exercise 59. 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.[23] 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 $O(t_R(|x|+p_R(|x|))^2)$.[24]

Although the foregoing construction of C_x capitalizes on various specific details of the (one-tape) Turing machine model, it can be adapted to any other "reasonable" model of efficient computation.[25] Alternatively, we recall the Cobham-Edmonds Thesis asserting that any deci-

[23]In continuation to Footnote 22, 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.

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

[25]**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).

sion 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 $f(x) = C_x \in \texttt{CSAT}$. Furthermore, $(x, y) \in R$ if and only if $(f(x), y) \in R_{\texttt{CSAT}}$. It follows that f is a Karp-reduction of S to CSAT, and, for $g(x, y) \stackrel{\text{def}}{=} y$ it holds that (f, g) is a Levin-reduction of R to $R_{\texttt{CSAT}}$. The theorem follows. ■

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

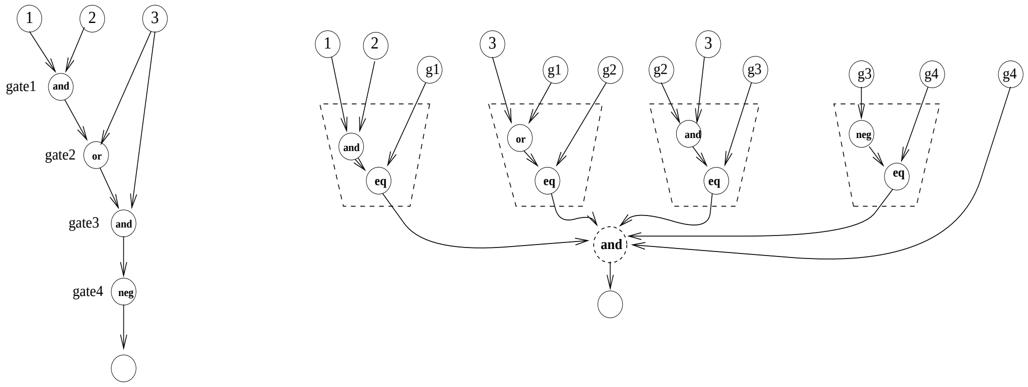
Theorem 34 (NP-completeness of SAT): *The set* (resp., relation) SAT (resp., $R_{\texttt{SAT}}$) *is $\mathcal{NP}$-complete* (resp., $\mathcal{PC}$-complete).

Proof: Since the set of possible instances of SAT is a subset of the set of instances of CSAT, it is clear that $\texttt{SAT} \in \mathcal{NP}$ (resp., $R_{\texttt{SAT}} \in \mathcal{PC}$). To prove that SAT is NP-hard, we reduce CSAT to SAT (and use Proposition 32). The reduction boils down to introducing auxiliary variables in order to "cut" the computation of an arbitrary ("deep") circuit into a conjunction of related computations of "shallow" circuits (i.e., depth-2 circuits) of unbounded fan-in, which in turn may be presented as a CNF formula. The aforementioned auxiliary variables hold the possible values of the internal gates of the input circuit, and the clauses of the CNF formula enforce the consistency of these values with the corresponding gate operation. For example, if $\texttt{gate}_i$ and $\texttt{gate}_j$ feed into $\texttt{gate}_k$, which is a $\wedge$-gate, then the corresponding variables g_i, g_j, g_k should satisfy $g_k \equiv (g_i \wedge g_j)$ (which can be written as a 3CNF with four clauses). Details follow.

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 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).[26] 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 5, 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. Suppose that for some string s it holds that $C(s) = 1$. Then, assigning the i^{th} auxiliary variable

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



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

Figure 5: The idea underlying the reduction of CSAT to SAT.

the value that corresponds to the one assigned to the i^{th} gate of C when evaluated on s, we obtain (together with s) a truth assignment that satisfies ϕ (because such an assignment satisfies all m constant-size CNFs as well as the variable associated with the output of C). On the other hand, if τ satisfies ϕ then the first n bits in τ correspond to an input on which C evaluates to 1 (because the m constant-size CNFs guarantee that the variables of ϕ are assigned values that correspond to the evaluation of C on the first n bits of τ). Note that the latter mapping (of τ to its n-bit prefix) is the second mapping required by the definition of a Levin-reduction. Thus, we have established that f is a Karp-reduction of `CSAT` to `SAT`, and that augmented with the second mapping it yields a Levin-reduction of $R_{\texttt{CSAT}}$ to $R_{\texttt{SAT}}$. ■

Comment. The fact that the second mapping required by the definition of a Levin-reduction is explicit in the proof of the validity of the corresponding Karp-reduction is a fairly common phenomenon. Actually, typical presentations of Karp-reductions provide two auxiliary polynomial-time computable mappings (in addition to the main mapping for instances from one problem (e.g., CSAT) to another (e.g., SAT)): The first auxiliary mapping is of solutions for the preimage instance (e.g., of CSAT) to solutions for the image instance of the reduction (e.g., of SAT), whereas the second mapping goes the other way around. (Note that only the main mapping and the second auxiliary mapping are required in the definition of a Levin-reduction.) For example, the proof of the validity of the Karp-reduction of `CSAT` to `SAT`, denoted f, specified two mappings h and g such that $(C,s) \in R_{\texttt{CSAT}}$ implies $(f(C), h(C,s)) \in R_{\texttt{SAT}}$ and $(f(C),\tau) \in R_{\texttt{SAT}}$ implies $(C, g(C,\tau)) \in R_{\texttt{CSAT}}$. Specifically, in the proof of Theorem 34, we used $h(C,s) = (s, a_1, ..., a_m)$ where a_i is the value assigned to the i^{th} gate in the evaluation of $C(s)$, and $g(C,\tau)$ being the n-bit prefix of τ. See Exercise 56.

3SAT. Note that the formulae resulting from the Karp-reduction presented in the proof of Theorem 34 are in conjunctive normal form (CNF) with each clause referring to at most three variables. Thus, the above reduction actually establishes the NP-completeness of 3SAT (i.e., SAT restricted to CNF formula with up to three variables per clause). Alternatively, one may Karp-reduce SAT (for CNF formula) to 3SAT (i.e., satisfiability of 3CNF formula), by replacing long clauses with conjunctions of three-variable clauses using auxiliary variables (see Exercise 60). Either way, we get the following result, where the furthermore part is proved by an additional reduction.

Proposition 35 *3SAT is NP-complete. Furthermore, the problem remains NP-complete also if we restrict the instances such that each variable appears in at most three clauses.*

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

4.3.2 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 a discussion of promise problems. We will also neglect to show that these decision problems are in NP.

We start with the `set cover` problem, in which an instance consists of a collection of finite sets $S_1, ..., S_m$ and an integer K and the question (for decision) is whether or not there exist (at most)[27] K sets that cover $\bigcup_{i=1}^m S_i$ (i.e., indices $i_1, ..., i_K$ such that $\bigcup_{j=1}^K S_{i_j} = \bigcup_{i=1}^m S_i$).

Proposition 36 `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 $S_{1,\mathtt{t}}, S_{1,\mathtt{f}}, .., S_{n,\mathtt{t}}, S_{n,\mathtt{f}} \subseteq \{1, ..., m\}$ such that $S_{i,\mathtt{t}}$ (resp., $S_{i,\mathtt{f}}$) is the set of the indices of the clauses (of ϕ) that are satisfied by setting the i^{th} variable to `true` (resp., `false`). That is, if the i^{th} variable appears unnegated (resp., negated) in the j^{th} clause then $j \in S_{i,\mathtt{t}}$ (resp., $j \in S_{i,\mathtt{f}}$). Note that the union of these $2n$ sets equals $\{1, ..., m\}$. Now, on input ϕ, the reduction outputs the Set Cover instance $f(\phi) \stackrel{\text{def}}{=} ((S_1, .., S_{2n}), n)$, where $S_{2i-1} = S_{i,\mathtt{t}} \cup \{m+i\}$ and $S_{2i} = S_{i,\mathtt{f}} \cup \{m+i\}$ for $i = 1, ..., n$.

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

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

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

Proposition 37 `3-Exact Cover` *is NP-complete.*

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

Proof Sketch: The reduction is obtained by composing three reductions. We first reduce a restricted case of `3SAT` (proved to be NP-complete in Proposition 35) to a restricted version of `Set Cover`, denoted `3SC`, in which each set has at most three elements (and an instance consists, as in the general case, of a sequence of finite sets as well as an integer K). Consider an instance ϕ of the aforementioned restricted case of `3SAT` (in which each *variable* appears in at most three clauses). If all three occurrences of a variable are of the same type (i.e., they are all negated or all non-negated) then this variable can be assigned a value that satisfies all clauses in which it appears, and so the variable and the clauses in which it appear can be omitted from the instance. In other words, we can reduce this restriction of `3SAT` to one in which each *literal* appears in at most two clauses.[28] Now, we reduce the new version of `3SAT` to `3SC` by using exactly the reduction presented in the proof of Proposition 36, and observing that the size of the generated sets is at most 3 (i.e., one more than the number of occurrences of the corresponding literal).

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

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

Vertex Cover, Independent Set, and Clique. Turning to graph theoretic problems, we start with the `Vertex Cover` problem, which is a special case of the `Set Cover` problem. The instances

[28] Actually, a closer look at the proof of Proposition 35 reveals the fact that the reduced instances satisfy this property anyhow.

consists of pairs (G,K), where $G=(V,E)$ is a simple graph and K is an integer, and the problem is whether or not there exists a set of (at most) K vertices that is incident to all graph edges (i.e., each edge in G has at least one endpoint in this set). Indeed, this instance of `Vertex Cover` can be viewed as an instance of `Set Cover` by considering the collection of sets $(S_v)_{v\in V}$, where S_v denotes the set of edges incident at vertex v (i.e., $S_v=\{e\in E: v\in e\}$). Thus, the NP-hardness of `Set Cover` follows from the NP-hardness of `Vertex Cover`. On the other hand, the `Vertex Cover` problem is computationally equivalent to the `Independent Set` and `Clique` problems (see Exercise 62), and thus it suffices to establish the NP-hardness of one of these problems.

Proposition 38 *The problems* `Vertex Cover`, `Independent Set` *and* `Clique` *are NP-complete.*

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

Note that if ϕ is satisfiable by a truth assignment τ then there are no edges between the m vertices that correspond to the partial satisfying assignment (derived from τ). (We stress that any truth assignment to ϕ yields an independent set, but only a satisfying assignment guarantees that this independent set contains a vertex from each of the m cliques.) Thus, $\phi\in$ `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 $\phi\in$ `SAT`. □

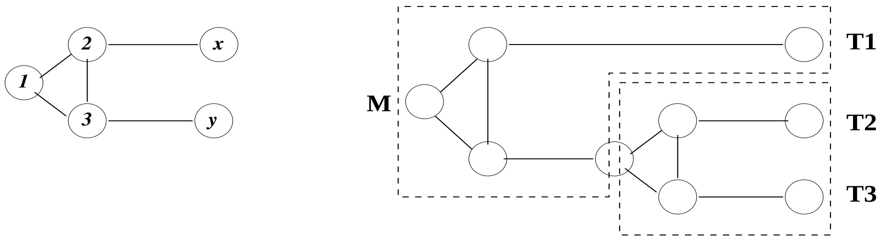
Graph 3-Colorability (`G3C`). In this problem the instances are graphs and the question is whether or not the graph can be colored using three colors (such that neighboring vertices are not assigned the same color).

Proposition 39 `Graph 3-Colorability` *is NP-complete.*

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

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

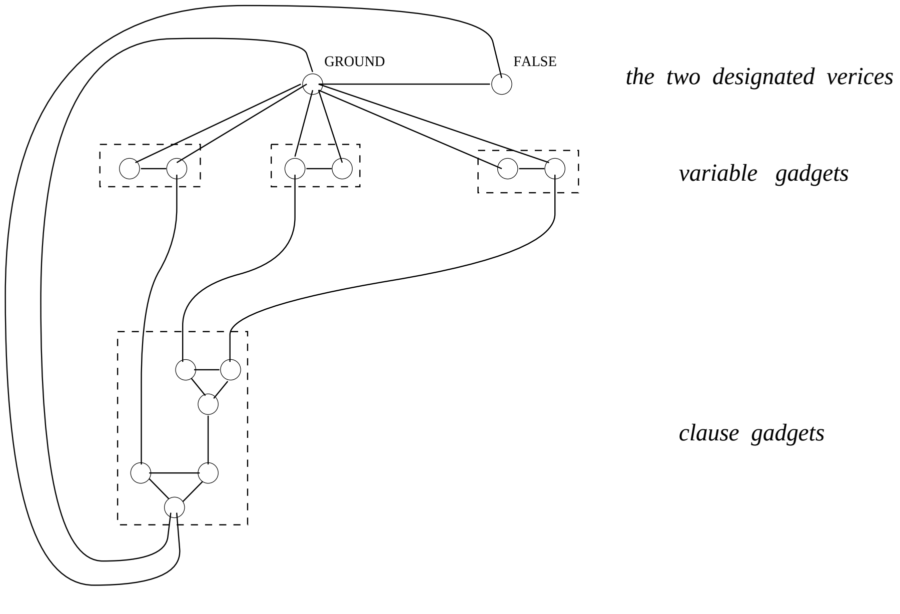
- The gadget associated with variable x is a pair of vertices, associated with the two literals of x (i.e., x and $\neg x$). These vertices are connected by an edge, and each of them is also connected to the vertex `ground`. Thus, in a 3-coloring of G_ϕ one of the vertices associated with the variable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



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

- The gadget associated with a clause C is depicted in Figure 6. 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 literals appearing in C. See Figure 7.



A single clause gadget and the relevant variables gadgets.

Figure 7: The reduction to G3C – connecting the gadgets.

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

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

As stated in Section 4.3, thousands of problems have been shown to be NP-complete (cf., [6, 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 40 *Assuming $\mathcal{NP} \neq \mathcal{P}$, there exist sets in $\mathcal{NP} \setminus \mathcal{P}$ such that some set in $\mathcal{NP}$ is not Cook-reducible to them.*

We mention that some natural problems (e.g., factoring) are conjectured to be neither solvable in polynomial-time nor NP-hard. One candidate class of such problems is $\mathcal{NP} \cap \text{co}\mathcal{NP}$, where $\text{co}\mathcal{NP} = \{\{0,1\}^* \setminus S : S \in \mathcal{NP}\}$. Specifically, if $\Delta \stackrel{\text{def}}{=} \mathcal{NP} \cap \text{co}\mathcal{NP} \neq \mathcal{P}$ and $\mathcal{NP} \neq \text{co}\mathcal{NP}$ then Δ is a class of sets that satisfy the conclusion of Theorem 40.[29] Below, the existence of sets satisfying this conclusion is proved based on the weaker assumption that $\mathcal{NP} \neq \mathcal{P}$ (which is actually a necessary condition for this conclusion).

> **Teaching note:** We recommend either stating Theorem 40 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 a finite set (i.e., a finite subset of S) must fail, because only sets in $\mathcal{P}$ are Cook-reducible to a finite set. Thus, for any potential reduction (i.e., a polynomial-time oracle machine), there exists an input x on which this reduction fails. It follows that this failure is due to a finite set of queries (i.e., the set of all queries made by the reduction when invoked on an input that is smaller or equal to x). Thus, for every finite set $F \subset S$, any reduction of S to S' can be failed by dropping a finite number of elements from S' and without dropping elements of F.

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

[29] This implication is based on the fact that $\mathcal{NP} \neq \text{co}\mathcal{NP}$ implies that sets in $\text{co}\mathcal{NP}$ are not NP-complete.

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

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

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

The value of $f(n)$ is defined by the the following experiment that consists of exactly n^3 computation steps (where cubic time is selected to allow for some non-trivial manipulations of data as conducted next). For $i = 0, 1, ...,$ we scan all inputs in lexicographic order trying to find an input on which the $i+1^{\text{st}}$ (modified) machine fails (where this machine is an oracle machine if i is even and a standard machine otherwise). In order to determine whether or not a failure occurs on a particular input, we may need to know the value of $f(n')$ for some n', which we just compute recursively (while counting the recursive steps in our total number of steps).[30] The point is that, when considering an input x, we may need the values of f only on $\{1, ..., p_{i+1}(|x|)\}$, where p_{i+1} is the polynomial bounding the running-time of the $i+1^{\text{st}}$ (modified) machine, and obtaining such a value takes at most $p_{i+1}(|x|)^3$ steps Also note that we may need to decide membership in $S \in \mathcal{NP}$, which we do by running the straightforward exponential-time algorithm (which tries all possible NP-witnesses). If we detect a failure of the $i+1^{\text{st}}$ machine, we increase i and proceed to the next iteration. When we reach the allowed number of steps (i.e., n^3 steps), we halt outputting the current value of i (i.e., the current i is output as the value of $f(n)$). Indeed, it is most likely that we will reach n^3 steps before examining inputs of length $3\log_2 n$, but this does not matter. What matters is that f is monotonically non-decreasing (because more steps allow to fail at least as many machines) as well as unbounded (see Exercise 67). Furthermore, by construction, $f(n)$ is computed

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

in poly(n) time. □

Comment: The proof of Theorem 40 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'*.[31] Thus, if $\mathcal{P} \neq \mathcal{NP}$ then there exists an infinite sequence of sets $S_1, S_2, \ldots$ in $\mathcal{NP} \setminus \mathcal{P}$ such that S_{i+1} is Karp-reducible to S_i but S_i is not Cook-reducible to S_{i+1}. That is, there exists an infinite hierarchy of problems (albeit unnatural ones), all in $\mathcal{NP}$, such that each problem is "easier" than the previous ones (in the sense that it can be reduced to the previous problems while these problems cannot be reduced to it).

Notes

It is quite remarkable that the theories of uniform and non-uniform computational devices have emerged in two single papers. We refer to Turing's paper [18], which introduced the model of Turing machines, and to Shannon's paper [15], which introduced Boolean circuits.

In addition to introducing the Turing machine model and arguing that it corresponds to the intuitive notion of computability, Turing's paper [18] introduces universal machines and contains proofs of undecidability (e.g., of the Halting Problem). Rice's Theorem is proven in [14], and the undecidability of the Post Correspondence Problem is proven in [13].

The formulation of machines that take advice (as well as the equivalence to the circuit model) originates in [9].

The association of efficient computation with polynomial-time algorithms is attributed to the papers of Cobham [2] and Edmonds [4]. It is interesting to note that Cobham's starting point was his desire to present a philosophically sound concept of efficient algorithms, whereas Edmonds's starting point was his desire to articulate why certain algorithms are "good" in practice.

Many sources provide historical accounts of the developments that led to the formulation of the *P vs NP Problem* and to the development of the theory of NP-completeness (see, e.g., [6, Sec. 1.5] and [17]). Still, we feel that we should not refrain from offering our own impressions, which are based on the text of the original papers.

Nowadays, the theory of NP-completeness is commonly attributed to Cook [3], Karp [8], and Levin [11]. It seems that Cook's starting point was his interest in theorem proving procedures for propositional calculus [3, 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., [3, P. 151]), and showed that any problem in $\mathcal{NP}$ is reducible to deciding membership in the set of 3DNF tautologies. In particular, Cook emphasizes the importance of the concept of polynomial-time reductions and of the complexity class $\mathcal{NP}$ (both explicitly defined for the first time in his paper). He also showed that `CLIQUE` is computationally equivalent to `SAT`, and envisioned a class of problems of the same nature. Karp's paper [8] can be viewed as fulfilling Cook's prophecy: Stimulated by Cook's work, Karp demonstrates that a "large number of classic difficult computational problems, arising in fields such as mathematical programming, graph theory, combinatorics, computational logic and switching theory, are [NP-]complete (and thus equivalent)" [8, P. 86]. Specifically, his list of twenty-one NP-complete problems includes Integer Linear Programming, Hamilton Circuit, Chromatic Number, Exact Set Cover, Steiner Tree, Knapsack, Job Scheduling, and Max Cut. Karp defines $\mathcal{NP}$ in terms of Definition 18, points to its relation to "backtrack search of polynomial bounded depth" and views it as the residence of a "wide range of important computational problems" (which are not in $\mathcal{P}$). Independently

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

of these developments, while being in the USSR, Levin proved the existence of "universal search problems" (where universality meant NP-completeness [11]). The starting point of Levin's work was his interest in the "perebor" conjecture asserting the inherent need for brute-force in some search problems that have efficiently checkable solutions (i.e., problems in $\mathcal{PC}$). He emphasizes the implication of polynomial-time reductions on the relation between the time complexity of the related problem (for any growth rate of the time complexity), asserts the NP-completeness of six "classical search problems" and claims that the underlying method "provides a mean for readily obtaining" similar results for "many other important search problems."

It is interesting to note that although the works of Cook [3], Karp [8], and Levin [11] were received with different levels of enthusiasm, none of the contemporaries realized the depth of the discovery and the difficulty of the question posed (i.e., the P-vs-NP Question). This fact is evident in every account from the early 1970's, and may explain the frustration of the relevant generation of researchers, which expected the P-vs-NP Question to be resolved in their life-time (if not in a matter of years). Needless to say, the author's opinion is that there was absolutely no justification for these expectations, and that one should have actually expected quite the opposite.

We mention that the three "founding papers" of the theory of NP-completeness (i.e., Cook [3], Karp [8], and Levin [11]) use the three different types of reductions used in these notes. Specifically, Cook uses the general notion of polynomial-time reduction [3], often referred to as Cook-reductions (Definition 22). The notion of Karp-reductions (Definition 23) originates from Karp's paper [8], whereas its augmentation to search problems (i.e., Definition 24) originates from Levin's paper [11]. 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 §4.3.2 are not necessarily the original ones. Most notably, the reduction establishing the NP-hardness of the Independent Set problem (i.e., Proposition 38) is adapted from [5]. In contrast, the reductions presented in §4.3.1 are merely a re-interpretation of the original reduction as presented in [3]. The equivalence of the two definitions of $\mathcal{NP}$ (i.e., Theorem 21) was proved in [8].

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 56), the preservation of the number of solutions (see Exercise 57), being computable by a log-space algorithm, and being invertible in polynomial-time (see [1], which actually refers to a stronger notion).

Exercises

Exercise 41 ($\mathcal{PF}$ 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 42 Show that any $S \in \mathcal{NP}$ has many different NP-proof systems (i.e., verification procedures $V_1, V_2, ...$ such that $V_i(x,y) = 1$ does not imply $V_j(x,y) = 1$ for $i \neq j$).

Guideline: For V and p be as in Definition 18, define $V_i(x,y) = 1$ if $|y| = p(|x|) + 1$ and there exists a prefix y' of y such that $V(x,y) = 1$.

Exercise 43 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 44 Regarding Definition 20, 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 45 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 46 Show that Karp-reductions (and Levin-reductions) are transitive.

Exercise 47 Show that some decision problems are not Karp-reducible to their complement (e.g., the empty set is not Karp-reducible to $\{0,1\}^*$).
A popular exercise of dubious nature is to show that any decision problem in $\mathcal{P}$ is Karp-reducible to any *non-trivial* decision problem, where the decision problem regarding a set S is called non-trivial if $S \neq \emptyset$ and $S \neq \{0,1\}^*$. It follows that every non-trivial set in $\mathcal{P}$ is Karp-reducible to its complement.

Exercise 48 (reducing search problems to optimization problems) For every polynomially bounded relation R, present a polynomial-time computable function f such that the search problem of R is computationally equivalent to the search problem in which given (x,v) one has to find a $y \in \{0,1\}^{\mathrm{poly}(|x|)}$ such that $f(x,y) \geq v$.
(Hint: use a Boolean function.)

Exercise 49 (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 50 Show that the standard search problem of Graph 3-Colorability is self-reducible, where this search problem consists of finding a 3-coloring for a given input graph.
(Hint: Iteratively extend the current prefix of a 3-coloring of the graph by making adequate oracle calls to the decision problem of Graph 3-Colorability. Specifically, encode the question of whether or not $(\chi_1, ..., \chi_t) \in \{1,2,3\}^t$ is a prefix of a 3-coloring of the graph G as a query regarding the 3-colorability of an auxiliary graph G'.)[32]

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

Exercise 51 Show that the standard search problem of Graph Isomorphism is self-reducible, where this search problem consists of finding an isomorphism between a given pair of graphs.
(Hint: Iteratively extend the current prefix of an isomorphism between the two N-vertex graphs by making adequate oracle calls to the decision problem of Graph Isomorphism. Specifically, encode the question of whether or not $(\pi_1, ..., \pi_t) \in [N]^t$ is a prefix of an isomorphism between $G_1 = ([N], E_1)$ and $G_2 = ([N], E_2)$ as a query regarding isomorphism between two auxiliary graphs G_1' and G_2'.)[33]

Exercise 52 (NP problems that are not self-reducible) Assuming that $\mathcal{P} \neq \mathcal{NP}$, show that there exists a search problem R in $\mathcal{PC}$ that is not self-reducible (i.e., the search problem of R is not Cook-reducible to the decision problem S_R implicit in R). Prove that it follows that $S_R' = \{(x, y') : \exists y'' \text{ s.t. } (x, y'y'') \in R\}$ is not Cook-reducible to $S_R = \{x : \exists y \text{ s.t. } (x, y) \in R\}$. Furthermore, prove that deciding S_R' is not reducible to the search problem of R.
(Hint: Consider the relation $R = \{(x, 0x) : x \in \{0,1\}^*\} \cup \{(x, 1y) : (x, y) \in R'\}$, where R' is an arbitrary relation in $\mathcal{PC}$, and note that $S_R = \{0,1\}^*$.)

Exercise 53 In continuation to Exercise 52, present a natural search problem R in $\mathcal{PC}$ such that if factoring integers is intractable then S_R' is not reducible to S_R.

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

Exercise 54 In continuation to Exercises 52 and 53, 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 55 Provide an alternative proof of Theorem 28 without referring to the set $S_R' = \{(x, y') : \exists y'' \text{ s.t. } (x, y'y'') \in R\}$. Hint: use Proposition 27.

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

Exercise 56 (additional properties of standard reductions) In continuation to the discussion in the main text, consider the following augmented form of Karp-reductions. Such a reduction of R to R' consists of three polynomial-time mappings (f, h, g) such that the following two conditions hold:

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

(We note that this definition is actually the one used by Levin in [11], 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 these notes satisfy this augmented requirement. Furthermore, prove that in all these cases the main mapping (i.e., f) is 1-1 and polynomial-time invertible.

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

Exercise 57 (parsimonious reductions) Let $R, R' \in \mathcal{PC}$ and let f be a Karp-reduction of $S_R = \{x : R(x) \neq \emptyset\}$ to $S_{R'} = \{x : R'(x) \neq \emptyset\}$. We say that g is parsimonious (with respect to R and R') if for every x it holds that $|R(x)| = |R'(g(x))|$. For each of the reductions, presented in these notes, checked whether or not it is parsimonious. For the others, alternative reductions that are parsimonious can be found (cf. [6, Sec. 7.3]).

Exercise 58 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 31 or present a reduction of (say) the search problem of $R_{\mathtt{u}}$ to the search problem of Bounded (Non-)Halting. Indeed, the exercise is more straightforward in the case of Bounded Halting.)

Exercise 59 In the proof of Theorem 33, 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 4). Present a function $f_M : \Gamma^3 \to \Gamma$, where $\Gamma = \Sigma \times (Q \cup \{\perp\})$, that substantiates this claim.

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

Exercise 60 Present and analyze a reduction of `SAT` to `3SAT`.

Guideline: For a clause C, consider auxiliary variables such that the i^{th} variable indicates whether one of the first i literals is satisfied, and replace C by a 3CNF that uses the original variables of C as well as the auxiliary variables. For example, the clause $\vee_{i=1}^t x_i$ is replaced by the conjunction of 3CNFs that are logically equivalent to the formulae $(y_2 \equiv (x_1 \vee x_2))$, $(y_i \equiv (y_{i-1} \vee x_i))$ for $i = 3, ..., t$, and y_t. We comment that this is not the standard reduction, but we find it more appealing conceptually.[34]

Exercise 61 (efficient solvability of 2SAT) In contrast to Exercise 60, prove that `2SAT` (i.e., the satisfiability of 2CNF formulae) is in $\mathcal{P}$.

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

Exercise 62 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.

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

Exercise 63 (Integer Linear Programming) Prove that the following problem is NP-complete. An instance of the problem is a systems of linear inequalities (say with integer constants), and the problem is to determine whether the system has an integer solution. For example, is there an integer solution to the system

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

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

Exercise 64 (Maximum Satisfiability over $\mathrm{GF}(2)$**)** 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 equations is in $\mathcal{P}$.)

Guideline: Reduce from 3SAT, using an arithetization similar to the one in Exercise 63. Specifically, replace each clause that contains $t \leq 3$ literals by 2^t linear GF(2) equations that correspond to the different subsets of these literals and assert that their sum (modulo 2) equals one; for example, the clause $x \vee \neg y$ is replaced by the equations $x + (1 - y) = 1$, $x = 1$, $1 - y = 1$, and $0 = 1$. Note that if the original clause was satisfied by a value assignment $\overline{v}$ then exactly 2^{t-1} of the corresponding equations are satisfied by $\overline{v}$, whereas if the original clause was unsatisfied by $\overline{v}$ then none of the corresponding equations is satisfied by $\overline{v}$.

Exercise 65 Prove that a set S is Karp-reducible to some set in $\mathcal{NP}$ if and only if S is in $\mathcal{NP}$.

Exercise 66 Recall that the empty set is not Karp-reducible to $\{0,1\}^*$, whereas any set is Cook-reducible to its complement. Thus our focus is on the Karp-reducibility of non-trivial sets to their complements, where a set is non-trivial if it is neither empty nor contains all strings.

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

   (Recall that any non-trivial set in $\mathcal{P}$ is Karp-reducible to its complement; see Exercise 47.)
2. Show that $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$ implies that some (non-trivial) sets in $\mathcal{NP}$ cannot be Karp-reducible to their complements.

(Hint: Use NP-completeness in both parts, and Exercise 65 in the second part.)

Exercise 67 Referring to the proof of Theorem 40, prove that the function f is unbounded (i.e., for every i there exists an n such that n^3 steps of the process defined in the proof allow for failing the $i + 1^{\text{st}}$ machine).

Guideline: Consider n' such that $f(n') = i$. Assuming, towards the contradiction that $f(n) = i$ for every $n > n'$, it follows that $F = F'' \cup F''$, where $F' = \{x : |x| \leq n' \wedge f(|x|) \equiv 0 \pmod 2\}$ is a finite set and $F'' = \{x : |x| > n'\}$ if i is odd and $F'' = \emptyset$ otherwise. In case i is odd, the $i + 1^{\text{st}}$ machine tries to decide

$S \cap F$ (which differs from S on finitely many strings), and must fail on some x. Derive a contradiction by showing that the number of steps that the process makes till reaching and considering the input x is at most $\exp(\text{poly}(|x|))$. A similar argument applies to the case that i is even, where we use the fact that F is finite and so the relevant reduction of S to $S \cap F$ must fail on some input x.

References

[1] L. Berman and J. Hartmanis. On isomorphisms and density of NP and other complete sets. *SIAM Journal on Computing*, Vol. 6 (2), 1977, pages 305–322. Extended abstract in *8th STOC*, 1976.

[2] A. Cobham. The Intristic Computational Difficulty of Functions. In *Proc. 1964 Iternational Congress for Logic Methodology and Philosophy of Science*, pages 24–30, 1964.

[3] S.A. Cook. The Complexity of Theorem Proving Procedures. In *3rd ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[4] J. Edmonds. Paths, Trees, and Flowers. *Canad. J. Math.*, Vol. 17, pages 449–467, 1965.

[5] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating Clique is almost NP-complete. *Journal of the ACM*, Vol. 43, pages 268–292, 1996. Preliminary version in *32nd FOCS*, 1991.

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

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

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

[9] R.M. Karp and R.J. Lipton. Some connections between nonuniform and uniform complexity classes. In *12th ACM Symposium on the Theory of Computing*, pages 302-309, 1980.

[10] E. Kushilevitz and N. Nisan. *Communication Complexity.* Cambridge University Press, 1996.

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

[12] N. Pippenger and M.J. Fischer. Relations among complexity measures. *Journal of the ACM*, Vol. 26 (2), pages 361–381, 1979.

[13] E. Post. A Variant of a Recursively Unsolvable Problem. *Bull. AMS*, Vol. 52, pages 264–268, 1946.

[14] H.G. Rice. Classes of Recursively Enumerable Sets and their Decision Problems. *Trans. AMS*, Vol. 89, pages 25–59, 1953.

[15] C.E. Shannon. A Symbolic Analysis of Relay and Switching Circuits. *Trans. American Institute of Electrical Engineers*, Vol. 57, pages 713–723, 1938.

[16] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[17] B.A. Trakhtenbrot. A Survey of Russian Approaches to *Perebor* (Brute Force Search) Algorithms. *Annals of the History of Computing*, Vol. 6 (4), pages 384–398, 1984.

[18] C.E. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. Londom Mathematical Soceity*, Ser. 2, Vol. 42, pages 230–265, 1936. A Correction, *ibid.*, Vol. 43, pages 544–546.