CAMBRIDGE

Cambridge University Press
978-0-521-19248-4 - P, NP, and NP-Completeness: The Basics of Computational Complexity
Oded Goldreich
Excerpt
More information

1

Computational Tasks and Models

Overview: We assume that the reader is familiar with computing devices but may associate the notion of computation with specific incarnations of it. Our first goal is to promote viewing computation as a general phenomenon, which may capture both artificial and natural processes. Loosely speaking, a computation is a process that modifies a relatively large environment via repeated applications of a simple and predetermined rule. Although each application of the rule has a very limited effect, the effect of many applications of the rule may be very complex.

We are interested in the transformation of the environment effected by the computational process (or computation), where the computation rule is designed to achieve a desired effect. 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. Thus, the computation defines a mapping from inputs to outputs, and such a mapping can be viewed as solving a search problem (i.e., given an instance x find a solution y that relates to x in some predetermined way) or a decision problem (i.e., given an instance x determine whether or not x has some predetermined property).

Indeed, our focus will be on solving computational tasks (mostly search and decision problems), where a computational task refers to an infinite set of instances such that each instance is associated with a set of valid solutions. In the case of search problem this set may contain several different solutions (per each instance), but in the case of a decision problem the set of solutions is a singleton that consists of a binary value (per each instance).

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-19248-4 - P, NP, and NP-Completeness: The Basics of Computational Complexity
Oded Goldreich
Excerpt
More information

In order to provide a basis for a rigorous study of the complexity of computational tasks, we need to define computation (and its complexity) rigorously. This, in turn, requires specifying a concrete model of computation, which corresponds to an abstraction of a real computer (be it a PC, mainframe, or network of computers) and yet is simpler (and thus facilitates further study). We will refer to the model of Turing machines, but any reasonable alternative model will do.

We also discuss two fundamental features of any reasonable model of computation: the existence of problems that cannot be solved by any computing device (in this model) and the existence of universal computing devices (in this model).

Organization. We start by introducing the general framework for our discussion of computational tasks (or problems). This framework refers to the *representation of instances* as binary sequences (see Section 1.1) and focuses on *two types of tasks*: searching for solutions and making decisions (see Section 1.2). Once computational tasks are defined, we turn to methods for solving such tasks, which are described in terms of some *model of computation*. The description of such models is the main contents of this chapter.

Specifically, we consider two types of models of computation: uniform models and non-uniform models (see Sections 1.3 and 1.4, respectively). The *uniform models correspond to the intuitive notion of an algorithm*, and will provide the stage for the rest of the book (which focuses on efficient algorithms). In contrast, non-uniform models (e.g., Boolean circuits) facilitate a closer look at the way a computation progresses, and will be used only sporadically in this book. Thus, *whereas Sections 1.1–1.3 are absolute prerequisites for the rest of this book, Section 1.4 is not.*

Teaching Notes

This chapter provides the necessary preliminaries for the rest of the book; that is, we discuss the notion of a computational task and present computational models for describing methods for solving such tasks.

Sections 1.1–1.3 correspond to the contents of a traditional *Computability* course, except that our presentation emphasizes some aspects and deemphasizes others. In particular, the presentation highlights the notion of a universal machine (see Section 1.3.4), explicitly discusses the complexity of computation

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-19248-4 - P, NP, and NP-Completeness: The Basics of Computational Complexity
Oded Goldreich
Excerpt
More information

(Section 1.3.5), and provides a definition of oracle machines (Section 1.3.6). This material (with the exception of Kolmogorov Complexity) is taken for granted in the rest of the current book. In contrast, Section 1.4 presents basic preliminaries regarding non-uniform models of computation (e.g., various types of Boolean circuits), and these are used only lightly in the rest of the book.

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 the Turing machine model is exactly as powerful as a model that is closer to a real-life computer (see the "sanity check" in §1.3.2.2); that is, a function can be computed by a Turing machine if and only if it is computable by a machine of the latter model. For starters, one may 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.

As noted in Section 1.3.7, we reject the common coupling of computability theory with the theory of automata and formal languages. Although the historical links between these two theories (at least in the West) cannot be denied, this fact cannot justify coupling two fundamentally different theories (especially when such a coupling promotes a wrong perspective on computability theory). Thus, in our opinion, the study of any of the lower levels of Chomsky's Hierarchy [16, Chap. 9] should be decoupled from the study of computability theory (let alone the study of Complexity Theory). Indeed, this is related to the discussion of the "revision of the CS curriculum" in the preliminary section "To the Teacher."

The perspective on non-uniform models of computation provided by Section 1.4 is more than the very minimum that is required for the rest of this book. If pressed for time, then the teacher may want to skip all of Section 1.4.2 as well as some of the material in Section 1.4.1 and Section 1.4.3 (i.e., avoid §1.4.1.2 as well as §1.4.3.2). Furthermore, for a minimal presentation of Boolean formulae, one may use Appendix A.2 instead of §1.4.3.1.

1.1 Representation

In mathematics and most other 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,

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-19248-4 - P, NP, and NP-Completeness: The Basics of Computational Complexity
Oded Goldreich
Excerpt
More information

where the latter can be thought of as a (possibly partial) representation of the instance. Indeed, the answer to any fully specified question is implicit in the question itself, and computation is employed to make this answer explicit.

Computational tasks refer to objects that are represented in some canonical way, where such canonical representation provides an "explicit" and "full" (but not "overly redundant") description of the corresponding object. Furthermore, when we discuss natural computational problems, we always use a natural representation of the corresponding objects. We will only consider *finite* objects like numbers, sets, graphs, and functions (and keep distinguishing these types of objects although, actually, they are all equivalent). While the representation of numbers, sets, and functions is quite straightforward (see the following), we refer the reader to Appendix A.1 for a discussion of the representation of graphs.

In order to facilitate a study of methods for solving computational tasks, these tasks are defined with respect to infinitely many possible instances (each being a finite object). Indeed, the comparison of different methods seems to require the consideration of infinitely many possible instances; otherwise, the choice of the language in which the methods are described may totally dominate and even distort the discussion (cf., e.g., the discussion of Kolmogorov Complexity in §1.3.4.2).

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 (long) strings. The set of all strings is denoted $\{0,1\}^*$; that is, $\{0,1\}^* = \cup_{n\in\mathbb{N}}\{0,1\}^n$, where $0 \in \mathbb{N}$. 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., $x = x_1 x_2 \cdots x_{|x|}$). 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 $x_1 x_1 \cdots x_m x_m 01 y_1 \cdots y_n \in \{0,1\}^*$). 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 $\sum_{i=0}^{n-1} b_i \cdot 2^i$, where typically we assume that this representation has no leading zeros (i.e., $b_{n-1} = 1$), except when the number itself is zero. Rational numbers

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-19248-4 - P, NP, and NP-Completeness: The Basics of Computational Complexity
Oded Goldreich
Excerpt
More information

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 means rational approximations of these real numbers.

Sets are usually represented as lists, which means that the representation introduces an order that is not specified by the set itself. Indeed, in general, the representation may have features that are not present in the represented object. Functions are usually represented as sets of argument–value pairs (i.e., functions are represented as binary relations, which in turn are sets of ordered pairs).

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 (e.g., produced by some algorithm) that something is wrong.

1.2 Computational Tasks

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

1.2.1 Search Problems

A search problem consists of a specification of a (possibly empty) set of valid solutions for each possible instance. 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 (e.g., finding shortest paths in a graph, finding an occurrence of a given pattern in a given string, finding the median value in a given list of numbers, etc). Furthermore, search problems correspond to the daily notion of "solving a problem" (e.g., finding one's way between two locations), 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.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-19248-4 - P, NP, and NP-Completeness: The Basics of Computational Complexity
Oded Goldreich
Excerpt
More information

Definition 1.1 (solving a search problem)**:** *Let* $R \subseteq \{0,1\}^* \times \{0,1\}^*$ *and* $R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\}$ *denote the set of solutions for the instance* x*. A function* $f : \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$ solves the search problem of R *if for every* x *the following holds: if* $R(x) \neq \emptyset$ *then* $f(x) \in R(x)$ *and otherwise* $f(x) = \perp$.

Indeed, $R = \{(x,y) \in \{0,1\}^* \times \{0,1\}^* : y \in R(x)\}$. The solver f is required to find a solution to the given instance x whenever such a solution exists; that is, given x, the solver is required to output some $y \in R(x)$ whenever the set $R(x)$ is not empty. It is also required that the solver f never outputs a wrong solution; that is, if $R(x) \neq \emptyset$ then $f(x) \in R(x)$, and if $R(x) = \emptyset$ then $f(x) = \perp$. This means that f indicates whether or not x has any solution (since $f(x) \in \{0,1\}^*$ if x has a solution, whereas $f(x) = \perp \notin \{0,1\}^*$ otherwise). Note that the solver is not necessarily determined by the search problem (i.e., the solver is uniquely determined if and only if $|R(x)| \leq 1$ holds for every x).

Of special interest is the case of search problems having a unique solution (for each possible instance); that is, the case that $|R(x)| = 1$ for every x. In this case, 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 by $R'(x) \stackrel{\text{def}}{=} y$ if and only if $R(x) = \{y\}$). Popular examples include sorting a sequence of numbers, multiplying integers, finding the prime factorization of a composite number, and so on.[1]

1.2.2 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 (i.e., whether or not the given number is in the set of prime numbers). Note that one typically presents decision problems in terms of deciding whether a given object has some predetermined property, but this can always be viewed as deciding membership in some predetermined set (i.e., the set of objects having this property). For example, when talking about determining whether or not a given graph is connected, we refer to deciding membership in the set of connected graphs.

[1] For example, sorting is represented as a binary relation that contains all pairs of sequences such that the second sequence is a sorted version of the first sequence. That is, the pair $((x_1,\ldots,x_n),(y_1,\ldots,y_n))$ is in the relation if and only if there exists a permutation π over $[n]$ such that $y_i = x_{\pi(i)}$ and $y_i < y_{i+1}$ for every relevant i.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-19248-4 - P, NP, and NP-Completeness: The Basics of Computational Complexity
Oded Goldreich
Excerpt
More information

One important type of decision problems concerns those derived from search problems by considering the set of instances having a solution (with respect to some fixed search problem); that is, for any binary relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ we consider the set $\{x : R(x) \neq \emptyset\}$. Indeed, being able to determine whether or not a solution exists is a prerequisite to being able to solve the corresponding search problem (as per Definition 1.1).

In general, decision problems refer to the natural task of making binary decisions, a task that is not uncommon in daily life (e.g., determining whether a traffic light is red). In any case, in the following definition of solving decision problems, the potential solver is again a function; specifically, in this case the solver is a Boolean function, which is supposed to indicate membership in a predetermined set.

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

That is, the solver f is required to indicate whether or not the instance x resides in the predetermined set S. This indication is modeled by a binary value, where 1 corresponds to a positive answer and 0 corresponds to a negative answer. Thus, given x, the solver is required to output 1 if $x \in S$, and output 0 otherwise (i.e., if $x \notin S$).

Note that the function that solves a decision problem is uniquely determined by the decision problem; that is, if f solves (the decision problem of) S, then f equals the characteristic function of S (i.e., the function $\chi_S : \{0,1\}^* \to \{0,1\}$ defined such that $\chi_S(x) = 1$ if and only if $x \in S$).

As hinted already in Section 1.2.1, the solver of a search problem implicitly determines membership in the set of instances that have solutions. That is, *if f solves the search problem of R, then the Boolean function $f' : \{0,1\}^* \to \{0,1\}$ defined by $f'(x) \stackrel{\text{def}}{=} 1$ if and only if $f(x) \neq \perp$ solves the decision problem of $\{x : R(x) \neq \emptyset\}$.*

Terminology. We often identify the decision problem of a set S with S itself, and also identify S with its characteristic function. Likewise, we often identify the search problem of a relation R with R itself.

Reflection. Most people would consider search problems to be more natural than decision problems: Typically, people seeks solutions more often 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

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-19248-4 - P, NP, and NP-Completeness: The Basics of Computational Complexity
Oded Goldreich
Excerpt
More information

to require more cumbersome formulations. This is the main reason that most expositions choose to focus on decision problems. The current book attempts to devote at least a significant amount of attention to search problems, too.

1.2.3 Promise Problems (an Advanced Comment)

Many natural search and decision problems are captured more naturally by the terminology of promise problems, in which 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 type (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 $\{0, 1\}^*$. For the time being, we ignore this issue, but we shall revisit it in Section 5.1. Here we just note that in typical cases, the issue can be ignored by postulating that every string represents some legitimate object; for example, each string that is not used in the natural representation of these objects is postulated to be a representation of some fixed object (e.g., when representing graphs, we may postulate that each string that is not used in the natural representation of graphs is in fact a representation of the 1-vertex graph).

1.3 Uniform Models (Algorithms)

We finally reach the heart of the current chapter, which is the definition of (uniform) models of computation. Before presenting these models, let us briefly explain the need for their formal definitions.

Indeed, we are all familiar with computers and with the ability of computer programs to manipulate data. But this familiarity is rooted in positive experience; that is, we have some experience regarding some things that computers can do. In contrast, Complexity Theory is focused at what computers cannot do, or rather with drawing the line between what can be done and what cannot be done. Drawing such a line requires a precise formulation of *all* possible computational processes; that is, we should have a clear definition of *all* possible computational processes (rather than some familiarity with some computational processes).

We note that while our main motivation for defining formal models of computation is to capture the intuitive notion of an algorithm, such models also provide a useful perspective on a wide variety of processes that take place in the world.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-19248-4 - P, NP, and NP-Completeness: The Basics of Computational Complexity
Oded Goldreich
Excerpt
More information

Organization of Section 1.3. We start, in Section 1.3.1, with a general and abstract discussion of the notion of computation. Next, in Section 1.3.2, we provide a high-level description of the model of Turing machines. This is done merely for the sake of providing a concrete model that supports the study of computation and its complexity, whereas the material in this book will not depend on the specifics of this model. In Section 1.3.3 and Section 1.3.4 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 Section 1.3.5. We also discuss oracle machines and restricted models of computation (in Section 1.3.6 and Section 1.3.7, respectively).

1.3.1 Overview and General Principles

Before being formal, let us offer a general and abstract description of the notion of computation. This description applies both to artificial processes (taking place in computers) and to processes that are aimed at modeling the evolution 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 the "mechanical" aspects of the natural reality, that is, the rules that determine the evolution of the reality (rather than the specific state of the reality at a specific time). In this case, the starting point of the study is the actual evolution process that takes place in the natural reality, and the goal of the study is finding the (computation) rule that underlies this natural process. In a sense, the goal of science at large can be phrased as finding (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 a corresponding artificial environment. Thus, our starting point is a desired functionality, and our aim is to design

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-19248-4 - P, NP, and NP-Completeness: The Basics of Computational Complexity
Oded Goldreich
Excerpt
More information

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

In the rest of this book (i.e., outside the current chapter), we will also consider the number of steps (i.e., applications of the rule) taken by the computation on 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 Section 1.3.2). 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 almost all 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 apparently simple computations).

What is being Computed? The foregoing discussion has implicitly referred to algorithms (i.e., computational processes) as means of computing functions. Specifically, an algorithm A computes the function $f_A : \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$ defined by $f_A(x) = y$ if, when invoked on input x, algorithm A halts with output y. However, algorithms can also serve as means of "solving search problems" or "making decisions" (as in Definitions 1.1 and 1.2). Specifically, we will say

© in this web service Cambridge University Press www.cambridge.org