Chapter

1

The Complexity of Optimization Problems

IN INTRODUCTORY computer programming courses we learn that computers are used to execute algorithms for the solution of *problems*. Actually, the problems we want to solve by computer may have quite varying characteristics. In general, we are able to express our problem in terms of some *relation* $P \subseteq I \times \mathcal{S}$, where I is the set of *problem instances* and $\mathcal{S}$ is the set of *problem solutions*. As an alternative view, we can also consider a predicate $p(x,y)$ which is true if and only if $(x,y) \in P$. If we want to analyze the properties of the computations to be performed, it is necessary to consider the characteristics of the sets I, $\mathcal{S}$ and of the relation P (or of the predicate p) more closely.

In some cases, we just want to determine if an instance $x \in I$ satisfies a given condition, i.e., whether $\pi(x)$ is verified, where π is a specified (unary) predicate. This happens, for example, when we want to check if a program is syntactically correct, or a certain number is prime, or when we use a theorem prover to decide if a logical formula is a theorem in a given theory. In all these cases, relation P reduces to a function $f : I \mapsto \mathcal{S}$, where $\mathcal{S}$ is the binary set $\mathcal{S} = \{\text{YES}, \text{NO}\}$ (or $\mathcal{S} = \{0,1\}$), and we denote our problem as a *decision* (or *recognition*) *problem*. We may also consider *search* problems, where, for any instance $x \in I$, a solution $y \in \mathcal{S}$ has to be returned such that $(x,y) \in P$ is verified. This includes such problems as, for example, finding a path in a graph between two given nodes or determining the (unique) factorization of an integer. In other cases, given an instance $x \in I$, we are interested in finding the "best" solution y^* (according to some measure) among all solutions $y \in \mathcal{S}$ such that $(x,y) \in P$ is verified. This is the case when, given a point q and a set of points Q in the plane, we want to determine the point $q' \in Q$ which is nearest to q or when, given a weighted graph, we want to find a Hamiltonian cycle, if any, of minimum cost. Problems of this kind are called *optimization problems* and have occurred frequently in most human activities, since the beginning of the history of mathematics.

The aim of this book is to discuss how and under what conditions optimization problems which are computationally hard to solve can be efficiently approached by means of algorithms which only return "good" (and possibly not the best) solutions. In order to approach this topic, we first have to recall how the efficiency of the algorithms and the computational complexity of problems are measured. The goal of this chapter is, indeed, to introduce the reader to the basic concepts related to the complexity of optimization problems. Since the whole setting of complexity theory is built up in terms of decision problems, we will first discuss these concepts by showing the main results concerning decision problems. In particular, we will introduce the two complexity classes P and NP, whose properties have been deeply investigated in the last decades, and we will briefly discuss their relationship. Subsequently, we will shift our attention to the optimization problem context and we will show how the previously introduced concepts can be adapted to deal with this new framework.

1.1 Analysis of algorithms and complexity of problems

THE MOST direct way to define the efficiency of an algorithm would be to consider how much time (or memory), for any instance, the algorithm takes to run and output a result on a given machine. Such a point of view depends on the structural and technological characteristics of the machines and of their system software. It is possible to see that, as a matter of fact, the cost of the same algorithm on two different machines will differ only by no more than a multiplicative constant, thus making cost evaluations for a certain machine significant also in different computing environments.

As a consequence, the algorithms presented throughout the book will be usually written in a Pascal-like language and will be hypothetically run on an abstract Pascal machine, composed by a control and processing unit, able to execute Pascal statements, and a set of memory locations identified by all variable and constant identifiers defined in the algorithm. Unless otherwise specified, all statements concerning the efficiency of algorithms and the complexity of problem solutions will refer to such a computation model.

1.1.1 Complexity analysis of computer programs

The simplest way to measure the running time of a program in the model chosen above is the *uniform cost* measure and consists in determining the overall number of instructions executed by the algorithm before halting.

This approach to computing the execution cost of a program is natural, but, since it assumes that any operation can be executed in constant time on operands of any size (even arbitrarily large), it may lead to serious anomalies if we consider the possibility that arbitrarily large values can be represented in memory locations. Thus, in practice, the uniform cost model may be applied in all cases where it is implicitly assumed that all memory locations have the same given size and the values involved in any execution of the algorithm are not greater than that size, i.e., any value to be represented during an execution of the algorithm can be stored in a memory location. In all other cases (for example, where a bound on the size of the values involved cannot be assumed) a different cost model, known as *logarithmic cost* model should be used.

The logarithmic cost model is obtained by assigning to all instructions a cost which is a function of the number of bits (or, equivalently for positive integers, of the logarithm) of all values involved. In particular, basic instructions such as additions, comparisons, and assignments are assumed to have cost proportional to the number of bits of the operands, while a cost $O(n\log n)$ for multiplying or dividing two n-bit integers may be adopted.

For example, in the execution of an assignment instruction such as $a := b+5$, we consider the execution cost equal to $\log|b| + \log 5$. Notice that we did not specify the base of the logarithm, since logarithms in different bases differ only by a multiplicative constant.

Such a cost model avoids the anomalies mentioned above and corresponds to determining the number of operations required to perform arithmetical operations with arbitrary precision on a real computer.

Both approaches can also be applied to evaluate the execution cost in terms of the amount of memory (space) used by the algorithm. In this framework, the uniform cost model will take into account the overall number of distinct memory locations accessed by the algorithm during the computation. On the other hand, according to the logarithmic cost model it should be necessary to consider the number of bits of the maximum value contained in such locations during the computation.

◀ Example 1.1

Consider Program 1.1 for computing $r = x^y$, where $x, y \in \mathbb{N}$. In the uniform cost model, this algorithm has time cost $2+3y$, while in the logarithmic cost model, it presents a time cost bounded by the expression $ay\log y + by^2\log x(\log y + \log\log x) + c$, where a, b, c are suitable constants (see Exercise 1.1). Concern-

Program 1.1: Exponentiation

```
input Nonnegative integers $x, y$;
output Nonnegative integer $r = x^y$;
begin
   $r := 1$;
   while $y \neq 0$ do
   begin
      $r := r * x$;
      $y := y - 1$
   end;
   return $r$
end.
```

ing the amount of memory used, the resulting uniform cost is 3 (the variables x, y, r), while the logarithmic cost is $\log y + (y+1)\log x$.

In the following, we will refer to the uniform cost model in all (time and space) complexity evaluations performed. This is justified by the fact that the numbers used by the algorithms described in this book will be sufficiently small with respect to the length of their inputs. Before going into greater detail, we need to make clear some aspects of cost analysis that may help us to make the ~~evaluation~~ easier and more expressive and to introduce a ~~suitable~~ notation.

1 **Worst case analysis.** The first aspect we have to make clear is that, for most applications, at least as a first step of the analysis, we are not interested in determining the precise running time or space of an algorithm for all particular values of the input. Our concern is rather more in determining the behavior of these execution costs as the input size grows. However, instances of the same size may anyway result in extremely different execution costs. It is well known, for example, that sorting algorithms may have a different behavior, from the point of view of the running time, if they have to sort an array whose elements are randomly chosen or an array of partially sorted items. For such reasons, in order to specify the performance of an algorithm on inputs of size n, we determine the cost of applying the algorithm on the *worst case* instance of that size, that is on the instance with highest execution cost.

In contrast with worst case analysis, in several situations a different approach may be taken (known as *average case* analysis), consisting of determining the average cost of running the algorithm on all instances of size n, assuming that such instances occur with a specified (usually uniform) probability distribution.

Worst case analysis is widely used since it provides us with the certainty that, in any case, the given algorithm will perform its task within the established time bound. Besides, it is often easier to obtain, while the average case analysis may require complex mathematical calculations and, moreover, has validity limited by the probabilistic assumptions that have been made on the input distribution.

2 **Input size.** As stated above, we are interested in expressing the execution costs as a (growing) function of the size of the instance of the problem to be solved. But, how to measure that size? In Example 1.1 the running time of the algorithm was provided as a function of the input values x, y. Even if this seemed a reasonable choice in the case of the computation of an integer function, conventionally, a different *input size measure* is adopted: the *length* or *size* of the input, that is the number of digits (possibly bits) needed to present the specific instance of the problem. This conventional choice is motivated by the fact that problems may be defined over quite heterogeneous data: integers or sequences of integers, sets, graphs, geometric objects, arrays and matrices, etc. In all cases, in order to be submitted as input to a computer algorithm, the data will have to be presented in the form of a character string over a suitable finite alphabet (e.g., the binary alphabet $\{0,1\}$). It is therefore natural to adopt the length of such a string as a universal input size measure good for all kinds of problems.

Thus we assume there exists an *encoding scheme* which is used to describe any problem instance in a string of characters over some alphabet. Even if different encoding schemes usually result in different strings (and input sizes) for the same instance, it is important to observe that for a wide class of *natural* encoding schemes (i.e., encoding schemes that do not introduce an artificially redundant description) the input sizes of the same instance are not too different from each other. This is expressed by saying that for any pair of natural encoding schemes e_1, e_2 and for any problem instance x, the resulting strings are polynomially related, that is there exist two polynomials p_1, p_2 such that $|e_1(x)| \leq p_1(|e_2(x)|)$ and $|e_2(x)| \leq p_2(|e_1(x)|)$, where $|e_i(x)|$ denotes the length of string $e_i(x)$, for $i = 1,2$.

In the following, for any problem instance x, we will denote with the same symbol x the string resulting by any natural encoding of the instance itself. In general, we will denote with the same symbol both an object and its encoding under any encoding scheme.

◄ Example 1.2

Let us consider the problem of determining if an integer $x \in Z^+$ is a prime number. A trivial algorithm which tries all possible divisors d, $1 \leq d \leq \sqrt{x}$, can perform (if x is prime) a number of steps proportional to $\sqrt{x}$. If we use the natural encoding scheme which represents integer x as a binary string of length $n = |x| \approx \log x$, the execution cost is instead proportional to $2^{n/2}$. Even if the two values are equal, the second one gives greater evidence of the complexity of the algorithm, which grows exponentially with the size of the input. Actually, at the current state of ~~knowledge, it is not yet known whether any algorithm for primality testing with~~ a polynomial number of steps exists. Notice that using a unary base encoding (an unnatural scheme) would result in an evaluation of the execution cost as the square root of the instance size.

3 **Asymptotic analysis of algorithms.** If we go back to the analysis of the running time of Program 1.1, we realize the following two facts. First, the expressions contain constants, which represent both the cost of some instructions (such as during the initialization phase of an algorithm) whose execution does not depend on the particular instance and the fact that single instructions, on real machines, present execution costs which are not unitary but depend on technological characteristics of the machine (and system software) and would have been extremely difficult to determine precisely. Second, the expressions themselves are upper bounds and not precise evaluations of the computation costs.

For the above two reasons, by making use of the standard *O-notation* (see Appendix A), we may as well describe the execution costs of the algorithm by saying that its execution time *asymptotically* grows no more than $O(y)$ in the uniform cost model and $O(y^2 \log x(\log y + \log\log x))$ in the logarithmic cost model.

More generally, let us see how the notation used to express the asymptotic behavior of functions can be used to describe the running time (or space) of an algorithm.

Let us denote as $\hat{t}_{\mathcal{A}}(x)$ the running time of algorithm $\mathcal{A}$ on input x. The *worst case running time* of $\mathcal{A}$ is then given by

$$t_{\mathcal{A}}(n) = \max\{\hat{t}_{\mathcal{A}}(x) \mid \forall x : |x| \leq n\}.$$

Similarly, let us denote as $\hat{s}_{\mathcal{A}}(x)$ the running space of algorithm $\mathcal{A}$ on input x. The *worst case running space* of $\mathcal{A}$ is then given by

$$s_{\mathcal{A}}(n) = \max\{\hat{s}_{\mathcal{A}}(x) \mid \forall x : |x| \leq n\}.$$

Definition 1.1 ► *Algorithm complexity bounds*

We say that algorithm $\mathcal{A}$

1. *has complexity (upper bound) $O(g(n))$ if $t_{\mathcal{A}}(n)$ is $O(g(n))$;*
2. *has complexity (lower bound) $\Omega(g(n))$ if $t_{\mathcal{A}}(n)$ is $\Omega(g(n))$;*
3. *has complexity (exactly) $\Theta(g(n))$ if $t_{\mathcal{A}}(n)$ is $\Theta(g(n))$.*

Similar definitions can be introduced for the space complexity.

By using the preceding notation, if an algorithm performs a number of steps bounded by $an^2 + b\log n$ to process an input of size n, we say that its complexity is $O(n^2)$. If, moreover, we are able to prove that for any n sufficiently large there exists an input instance of size n on which the algorithm performs at least cn^2 steps, we say that its complexity is also $\Omega(n^2)$. In such a case we also say that the complexity of the algorithm is $\Theta(n^2)$.

◀ Example 1.3

Let us again consider the analysis of Program 1.1 in Example 1.1. As we have already seen, the running time of this algorithm under logarithmic cost measure is $ay\log y + by^2\log x(\log y + \log\log x) + c$. As noticed above, the term $by^2\log x(\log y + \log\log x)$ dominates all others. Let $n = |x| + |y|$ be the overall input size: a proper expression for the asymptotic cost analysis as a function of the input size would be $O(n^2 2^{2n})$.

The typical procedure to perform a complexity analysis of an algorithm requires deciding, case by case, on the following issues:

1. How to measure the execution cost. Usually, in the case of evaluating the running time, this is done by determining a *dominant* operation, that is an operation which is executed at least as often as any other operation in the algorithm, thus characterizing the asymptotic complexity of the algorithm. More formally, if the asymptotic complexity of the algorithm is $\Theta(g(n))$, a dominant operation is any operation whose contribution to the cost is $\Theta(g(n))$.

   Typically, in a sorting algorithm a dominant operation is the comparison of two elements, in matrix multiplication a dominant operation may be assumed to be the multiplication of elements, etc.

2. Which cost measure we want to adopt. As said above, the logarithmic cost model is more precise, but the simpler uniform cost model can be applied if we make the assumption that all values involved in the algorithm execution are upper bounded by some value (usually some function of the input size).

3. How to measure the input size, that is what characteristic parameter of the input instance is the one whose growth towards infinity determines the asymptotic growth of the computation cost. In this case, typical examples are the number of graph nodes and edges in graph algorithms, the number of rows and columns of matrices in algebraic algorithms, etc.

◀ Example 1.4

Let us consider a simple version of the well known *Insertion Sort* algorithm (see Program 1.2). Suppose that an array of size n is given and that we want to sort

Program 1.2: Insertion Sort

```
input Array $A[1,\ldots,n]$ of integers;
output Array $A$ sorted in increasing order;
begin
   for $i := 1$ to $n-1$ do
   begin
      $j := i+1$;
      while $j \geq 2$ and $A[j] < A[j-1]$ do
      begin
         swap $A[j]$ and $A[j-1]$;
         $j := j-1$
      end
   end;
   return $A$
end.
```

it in increasing order. Without loss of generality let us assume that the range of the elements is the finite interval $\{1,\ldots,M\}$, where M is a constant. Here the input size is characterized by the number n of elements in the array. In the uniform model we may simply take into consideration comparison operations. Since the algorithm contains two loops each of which is repeated at most n times, the running time is clearly bounded by $O(n^2)$. On the other side, it is not difficult to observe that the number of comparisons performed by the algorithm is indeed $n(n-1)/2$ in the worst case, that is when the array is ordered in decreasing order. Hence, the algorithm also has a lower bound $\Omega(n^2)$ on the running time.

Regarding the running space of the algorithm, it is immediate to see that an $\Omega(n)$ lower bound holds, due to the need for accessing all values in the input instance. The algorithm accesses such values plus a constant number of additional locations, thus resulting in a $O(n)$ upper bound.

1.1.2 Upper and lower bounds on the complexity of problems

Suppose we have to solve a problem and we have an algorithm whose time complexity is $O(g(n))$. According to the following definition, we say that $O(g(n))$ is an upper bound to the time complexity of the given problem.

Definition 1.2 ► *Problem complexity upper bound*

Given a problem $\mathcal{P}$ we say that the time complexity upper bound *of $\mathcal{P}$ is $O(g(n))$ if there exists an algorithm for $\mathcal{P}$ whose running time is $O(g(n))$.*

In other words, the complexity upper bound for a problem provides information on the amount of time which is asymptotically sufficient to solve the problem. Knowing the upper bound does not mean that a precise knowledge of the complexity of the problem is available. In fact, other more efficient algorithms with smaller running times may exist, and we may just be unaware of their existence.

A more precise characterization of the time complexity of a problem is achieved when we are also able to find a time complexity lower bound, that is we are able to establish how much time is anyhow needed (asymptotically) to solve the problem, no matter what algorithms are used. Knowing a lower bound for a problem provides important information on the intrinsic difficulty of solving the problem, especially in the case that the lower bound is sufficiently close to (or even coincides with) the upper bound.

◀ Definition 1.3
Problem complexity bounds

Given a problem $\mathcal{P}$ we say that the time complexity lower bound *of $\mathcal{P}$ is $\Omega(g(n))$ if any algorithm for $\mathcal{P}$ has a running time $\Omega(g(n))$, and that the* time complexity *of $\mathcal{P}$ is $\Theta(g(n))$ if its upper bound is $O(g(n))$ and its lower bound is $\Omega(g(n))$.*

Establishing a complexity lower bound for a problem is a hard task, since it requires stating (and proving) a property that has to hold for all (known and unknown) algorithms that solve the problem.

In the case of sorting, by means of an information theoretic argument, it can be proved that, in terms of comparisons, the time complexity lower bound is $\Omega(n \log n)$, that is any algorithm for sorting that uses only comparisons requires $\Omega(n \log n)$ comparisons. Since the time complexity upper bound for sorting is also $O(n \log n)$, this means that, in terms of comparisons, the time complexity of sorting is exactly determined to be $\Theta(n \log n)$.

Unfortunately, such a precise characterization is not achieved in all cases. For most problems of great practical interest (the satisfiability of a Boolean formula in the propositional calculus, the multiplication of two matrices, the decomposition of a number into prime factors, etc.) the time complexity has not presently been precisely determined. In particular, this is the case for all optimization problems which form the subject of this book. Despite this difficulty, a classification of (both decision and optimization) problems according to their complexity can be established in formal terms. Before examining how the complexity of optimization problems can be defined, let us recall some basic notions about the complexity of decision problems.

1.2 Complexity classes of decision problems

A PROBLEM $\mathcal{P}$ is called a *decision problem* if the set $I_{\mathcal{P}}$ of all instances of $\mathcal{P}$ is partitioned into a set $Y_{\mathcal{P}}$ of *positive instances* and a set $N_{\mathcal{P}}$ of *negative instances* and the problem asks, for any instance $x \in I_{\mathcal{P}}$, to verify whether $x \in Y_{\mathcal{P}}$.

Actually, since any algorithm for $\mathcal{P}$ also can receive inputs that do not correspond to legal instances of $\mathcal{P}$, the set of inputs to such an algorithm is partitioned in $Y_{\mathcal{P}}$, $N_{\mathcal{P}}$ and $D_{\mathcal{P}}$, the set of inputs not corresponding to instances of $\mathcal{P}$.

We assume that any algorithm for a decision problem is able to return a YES or NO answer, for example by printing it (see Fig. 1.1).

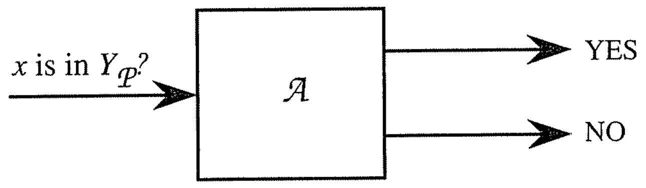


Figure 1.1
Solving a decision problem

Definition 1.4 ► *Problem solution*

A decision problem $\mathcal{P}$ is solved *by an algorithm $\mathcal{A}$ if the algorithm halts for every instance $x \in I_{\mathcal{P}}$, and returns* YES *if and only if $x \in Y_{\mathcal{P}}$. We also say that set $Y_{\mathcal{P}}$ is* recognized *by $\mathcal{A}$. Moreover, we say that $\mathcal{P}$ is solved in time $t(n)$ (space $s(n)$) if the time (space) complexity of $\mathcal{A}$ is $t(n)$ ($s(n)$).*

Notice that, if $x \in N_{\mathcal{P}} \cup D_{\mathcal{P}}$, then $\mathcal{A}$ may either halt returning NO, halt without returning anything, or never halt.

There are several reasons why the main concepts of computational complexity have been stated in terms of decision problems instead of computation of functions.

First of all, given an encoding scheme, any decision problem can be seen, independently from its specific kind of instances (graphs, numbers, strings, etc.), as the problem of discriminating among two sets of strings: encodings of instances in $Y_{\mathcal{P}}$ and strings in $N_{\mathcal{P}} \cup D_{\mathcal{P}}$. This allows a more formal and unifying treatment of all of them as language recognition problems. In general, for any decision problem $\mathcal{P}$, we will denote the corresponding language as $L_{\mathcal{P}}$.

Secondly, decision problems have a YES or NO answer. This means that in the complexity analysis we do not have to pay attention to the cost of producing the result. All the costs are strictly of a computational nature and they have nothing to do with the size of the output or the time needed to return it (for example by printing it).

The main aim of the theory of computational complexity is the characterization of collections of problems with respect to the computing resources (time, space) needed to solve them: such collections are denoted as *complexity classes*.

Let us now define some important complexity classes (it is intended that Defs. 1.2 and 1.3 can be applied *–mutatis mutandis–* to space complexity).

Problem 1.1: Satisfying truth assignment

INSTANCE: CNF formula $\mathcal{F}$ on a set V of Boolean variables, truth assignment on V, $f : V \mapsto \{\text{TRUE}, \text{FALSE}\}$.

QUESTION: Does f satisfy $\mathcal{F}$?

◀ Definition 1.5 *Complexity classes*

For any function $f(n)$, let TIME$(f(n))$ (SPACE$(f(n))$) *be the collection of decision problems which can be solved with a time (space) complexity* $O(f(n))$.

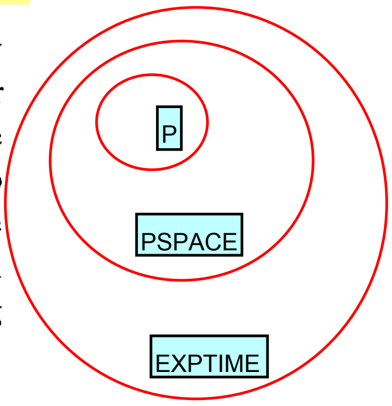
Often, we are more interested in classifying problems according to less refined classes. Thus, we may define the following complexity classes:

1. the class of all problems solvable in time proportional to a polynomial of the input size: $\text{P} = \cup_{k=0}^{\infty} \text{TIME}(n^k)$;

2. the class of all problems solvable in space proportional to a polynomial of the input size: $\text{PSPACE} = \cup_{k=0}^{\infty} \text{SPACE}(n^k)$;

3. the class of all problems solvable in time proportional to an exponential of the input size: $\text{EXPTIME} = \cup_{k=0}^{\infty} \text{TIME}(2^{n^k})$.

Clearly, P $\subseteq$ PSPACE: indeed, any algorithm cannot access more memory locations than the number of computing steps performed. Moreover, under suitable assumptions on the computation model, it is also easy to prove that PSPACE $\subseteq$ EXPTIME (see, for example, Exercise 6.11 which refers to the Turing machine model of computation). Whether these inclusions are strict (that is, P $\subset$ PSPACE and PSPACE $\subset$ EXPTIME), are open problems in complexity theory. The only separation result already known concerning the above defined classes is P $\subset$ EXPTIME.



The class P is traditionally considered as a reasonable threshold between tractable and intractable problems.

◀ Example 1.5

Let us consider SATISFYING TRUTH ASSIGNMENT, that is, Problem 1.1. This problem is in P. In fact, there exists a trivial decision algorithm which first plugs in the truth values given by f and then checks whether each clause is satisfied. The time complexity of this algorithm is clearly linear in the input size.

◀ Example 1.6

Let us consider SATISFIABILITY, that is, Problem 1.2. It is easy to show that this problem belongs to PSPACE. Consider the algorithm that tries every possible truth assignment on V and for each assignment computes the value of $\mathcal{F}$. As soon as the value of $\mathcal{F}$ is TRUE, the algorithm returns YES. If $\mathcal{F}$ is false for all truth assignments, the algorithm returns NO. The space needed by the algorithm to check one truth assignment is clearly polynomial in the size of the input. Since

Problem 1.2: Satisfiability

INSTANCE: CNF formula $\mathcal{F}$ on a set V of Boolean variables.

QUESTION: Is $\mathcal{F}$ satisfiable, i.e., does there exist a truth assignment $f : V \mapsto \{\text{TRUE}, \text{FALSE}\}$ which satisfies $\mathcal{F}$?

this space can be recycled and at most polynomial space is required for the overall control, this algorithm has indeed polynomial-space complexity.

Example 1.7 ▶ Given a sequence of Boolean variables $V = \{v_1, v_2, \ldots, v_n\}$ and a CNF Boolean formula $\mathcal{F}$ on V, let the corresponding quantified Boolean formula be defined as

$$Q_1 v_1 Q_2 v_2 Q_3 v_3 \ldots Q_n v_n \mathcal{F}(v_1, v_2, \ldots, v_n),$$

where, for any $i = 1, \ldots, n$, $Q_i = \exists$ if i is odd and $Q_i = \forall$ if i is even. We can see that QUANTIFIED BOOLEAN FORMULAS, that is Problem 1.3, belongs to PSPACE by modifying the PSPACE algorithm for SATISFIABILITY above. We try all truth assignments on V by assigning values to the variables from left to right, and after each assignment computing the value of the expression to the right. For example, assume that $v_1 \ldots v_{i-1}$ have been assigned values $z_1, \ldots, z_{i-1}$, respectively. Besides, let r_1 be the value of the expression

$$Q_{i+1} v_{i+1} \ldots Q_n v_n \mathcal{F}(z_1, z_2, \ldots, z_{i-1}, \text{TRUE}, v_{i+1}, \ldots, v_n)$$

and r_0 the value of the expression

$$Q_{i+1} v_{i+1} \ldots Q_n v_n \mathcal{F}(z_1, z_2, \ldots, z_{i-1}, \text{FALSE}, v_{i+1}, \ldots, v_n).$$

Then the value of expression

$$Q_i v_i Q_{i+1} v_{i+1} \ldots Q_n v_n \mathcal{F}(z_1, z_2, \ldots, z_{i-1}, v_i, v_{i+1}, \ldots, v_n)$$

is $r_0 \wedge r_1$ if i is even (i.e., $Q_i = \forall$) and $r_0 \vee r_1$ if i is odd (i.e., $Q_i = \exists$). This algorithm can easily be implemented in order to compute the value of the whole expression using space polynomial in the input size (see Exercise 1.6).

Note that no polynomial-time algorithm is known for the problems considered in the last two examples; indeed, it is widely believed that no such algorithm can exist. However, as we will see in the next subsection, these problems present rather different complexity properties, that make them paradigmatic problems in different complexity classes.

1.2.1 The class NP

The classification of problems that we have provided so far is indeed too coarse with respect to the need for characterizing the complexity of several problems of great practical interest. For example, even though we have

Problem 1.3: Quantified Boolean formulas

INSTANCE: CNF Boolean formula $\mathcal{F}$ on a set $V = \{v_1, v_2, \ldots, v_n\}$ of Boolean variables.

QUESTION: Is the quantified Boolean formula

$$\exists v_1 \forall v_2 \exists v_3 \ldots Q v_n \mathcal{F}(v_1, v_2, \ldots, v_n)$$

true, where $Q = \exists$ if n is odd, otherwise $Q = \forall$?

shown that both QUANTIFIED BOOLEAN FORMULAS and SATISFIABILITY belong to PSPACE, it is possible to see that these two problems present quite different characteristics if we consider their behavior when we analyze, instead of the cost of *solving* a problem, the cost of *verifying* that a given mathematical object is indeed the solution of a problem instance.

According to the definition of decision problem given in the preceding section, given an instance x of a decision problem $\mathcal{P}$ (encoded as a string of symbols), the corresponding solution consists of a single binary value 1 or 0 (YES or NO, TRUE or FALSE), expressing the fact that x belongs to the set $Y_{\mathcal{P}}$ of positive instances or not. Actually, in most cases, determining that a problem instance x belongs to $Y_{\mathcal{P}}$ should also be supported by deriving some object $y(x)$ (string, set, graph, array, etc.) which depends on x, whose characteristics are stated in the problem instance and whose existence is what is asked for in the decision problem. In this case we would call $y(x)$ a *constructive solution* of $\mathcal{P}$.

◀ Example 1.8

Let us consider again SATISFIABILITY. Any positive instance of this problem has (at least) one associated solution $y(x)$ represented by a truth assignment f which satisfies the Boolean formula $\mathcal{F}$.

Given a decision problem $\mathcal{P}$ and an instance $x \in Y_{\mathcal{P}}$, verifying whether a string of characters σ is the description of a constructive solution $y(x)$ of $\mathcal{P}$ is often a much easier task than solving the decision problem itself. In the case of SATISFIABILITY, whereas deciding whether there exists a truth assignment which satisfies a CNF Boolean formula is a complex task (no algorithm is known which solves this problem in time polynomial in the length of the input), verifying whether a string represents a satisfying truth assignment can be done in time linear in the size of the instance (see Example 1.5).

The cost of verifying problem solutions can be used as an additional measure to characterize problem complexity. In order to do this, the concept of a *nondeterministic algorithm* has been introduced to provide a com-

Program 1.3: Nondeterministic SAT

```
input CNF Boolean formula $\mathcal{F}$ over a set $V$ of Boolean variables;
output YES if $\mathcal{F}$ is satisfiable;
begin
   for each $v$ in $V$ do
   begin
      guess $y \in \{0,1\}$;
      if $y = 0$ then $f(v)$ := FALSE else $f(v)$ := TRUE
   end;
   if $f$ satisfies $\mathcal{F}$ then return YES else return NO
end.
```

putational framework coherent to the one assumed when problem solving is considered.

A nondeterministic algorithm is an algorithm which, apart from all usual constructs, can execute commands of the type "**guess** $y \in \{0,1\}$". Such an instruction means that y can take as its value either 0 or 1. Essentially, a nondeterministic algorithm has the additional option of "guessing" a value, in particular of guessing a continuation (in a finite set of possible continuations) of the computation performed so far. Thus, while the computations performed by the (deterministic) algorithms considered so far have a linear structure (since at any time there is only one possible next step of the computation), nondeterministic computations can be described by a tree-like structure (called *computation tree*), where **guess** instructions correspond to branching points.

Notice also that a deterministic algorithm presents one outcome for each input instance, while a nondeterministic one will have many different outcomes, in correspondence to different sequences of guesses. If, in particular, we consider nondeterministic algorithms for acceptance or rejection of strings, some sequences of guesses will lead the algorithm to return YES, some to return NO.

Definition 1.6 ► *Nondeterministic problem solution*

Given a decision problem $\mathcal{P}$, a nondeterministic algorithm $\mathcal{A}$ solves $\mathcal{P}$ if, for any instance $x \in I_{\mathcal{P}}$, $\mathcal{A}$ halts for any possible guess sequence and $x \in Y_{\mathcal{P}}$ if and only if there exists at least one sequence of guesses which leads the algorithm to return the value YES.

Example 1.9 ► Program 1.3 is a nondeterministic algorithm for the SATISFIABILITY problem. As one can see, the algorithm essentially guesses a candidate truth assignment among the $2^{|V|}$ possible ones and then verifies whether the guess has been successful. The behavior of the algorithm with input the Boolean formula containing the two clauses $v_1 \vee v_2 \vee \bar{v}_3$ and $\bar{v}_1 \vee \bar{v}_2 \vee v_3$ is shown in Fig. 1.2 where the computation tree is represented. Each path in this tree denotes a possible sequence of guesses of the algorithm: the input is accepted if and only if at least one path produces a truth assignment that satisfies the formula. Notice that, since the verification phase can be performed in polynomial time (see Example 1.5) and the sequence of guesses is polynomially long, the nondeterministic algorithm itself takes polynomial time.

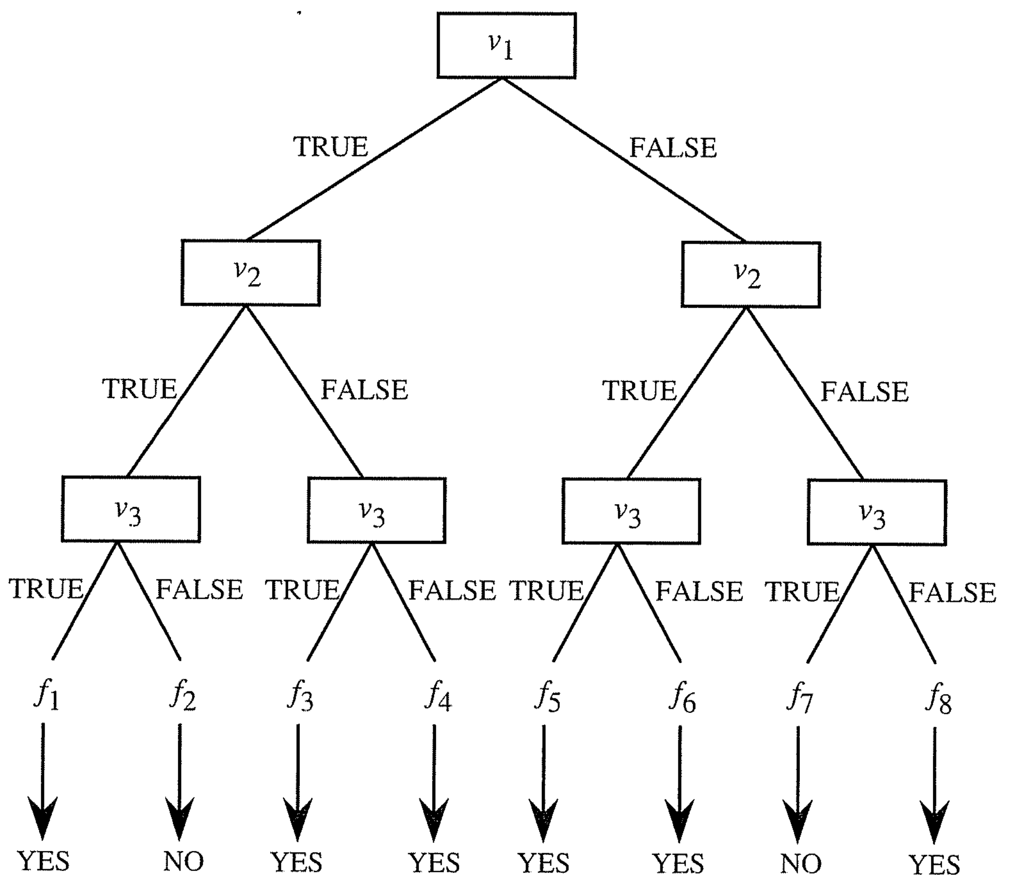


Figure 1.2
A nondeterministic algorithm for SATISFIABILITY with input $v_1 \vee v_2 \vee \bar{v}_3, \bar{v}_1 \vee \bar{v}_2 \vee v_3$

◀ Definition 1.7
Complexity of nondeterministic algorithm

A nondeterministic algorithm $\mathcal{A}$ solves a decision problem $\mathcal{P}$ in time complexity $t(n)$ if, for any instance $x \in I_{\mathcal{P}}$ with $|x| = n$, $\mathcal{A}$ halts for any possible guess sequence and $x \in Y_{\mathcal{P}}$ if and only if there exists at least one sequence of guesses which leads the algorithm to return the value YES *in time at most $t(n)$.*

Without loss of generality, we may assume that all (accepting and non-accepting) computations performed by a nondeterministic algorithm that solves a decision problem in time $t(n)$ halt: we may in fact assume that the algorithm is equipped with a step counter which halts the execution when more than $t(n)$ steps have been performed.

Moreover, without loss of generality, we may also consider a simplified model of a polynomial-time nondeterministic algorithm which performs a polynomially long sequence of **guess** operations at the beginning of its execution. Such a sequence can be seen as a unique overall guess of all the outcomes of the at most polynomial number of guesses in any computation, that is, we assume that all nondeterministic choices to be performed during any computation are guessed at the beginning of the computation.

Definition 1.8 ► *Nondeterministic complexity classes*

For any function $f(n)$, let NTIME$(f(n))$ *be the collection of decision problems which can be solved by a nondeterministic algorithm in time $O(f(n))$.*

We may then define the class NP as the class of all decision problems which can be solved in time proportional to a polynomial of the input size by a nondeterministic algorithm, i.e., NP $= \bigcup_{k=0}^{\infty}$ NTIME(n^k). By the above considerations, this is equivalent to the class of all decision problems whose constructive solutions can be verified in time polynomial in the input size.

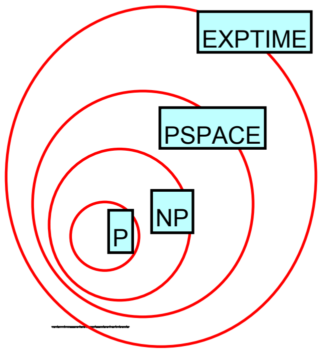
It is easy to see that P $\subseteq$ NP, since a conventional (deterministic) algorithm is just a special case of a nondeterministic one, in which no guess is performed. Moreover, several problems in EXPTIME (and probably not belonging to P) are also in NP. For all such problems a solution to the corresponding constructive problem, if any, has to be found in an exponentially large search space. Therefore, their complexity arises from the need to generate and search such an exponentially large space, whereas, given any element of the search space, checking whether it is a solution or not is a relatively simpler task. A paradigmatic example of such problems is SATISFIABILITY (see Example 1.5).

Verifying a positive solution reduces to the SATISFYING TRUTH ASSIGNMENT, or:
A nondeterministic algorithm is given in Figure 1.2

On the other side, in the case of QUANTIFIED BOOLEAN FORMULAS, there is no known method of exploiting nondeterminism for obtaining a solution in polynomial time. In fact, this problem is believed not to belong to NP, and that, therefore, NP $\subset$ PSPACE. Observe that in the case of QUANTIFIED BOOLEAN FORMULAS the natural constructive solution for a positive instance would be a subtree of the tree of all truth assignments: this subtree alternatively contains nodes with branching factor 1 and nodes with branching factor 2. Then, this solution has size exponential in the input length and cannot be guessed in polynomial time.

This inclusion is true, but not obvious at this moment.

Before closing this section we point out that nondeterministic algorithms present an asymmetry between instances in $Y_{\mathcal{P}}$ and instances in $N_{\mathcal{P}} \cup D_{\mathcal{P}}$. In fact, while in order for an instance to be recognized as belonging to $Y_{\mathcal{P}}$ there must exist *at least one* accepting computation, the same instance is recognized to be in $N_{\mathcal{P}} \cup D_{\mathcal{P}}$ if and only if *all* computations are non-accepting. Even in the case that all computations halt, this makes the conditions to be verified inherently different, since they correspond to checking an existential quantifier in one case and a universal quantifier in the other case.



Problem 1.4: Falsity

INSTANCE: A CNF formula $\mathcal{F}$ on a set V of Boolean variables.

QUESTION: Is it true that there does not exist any truth assignment $f : V \mapsto \{\text{TRUE}, \text{FALSE}\}$ which satisfies $\mathcal{F}$?

Let us consider the FALSITY problem, that is, Problem 1.4. Clearly, this problem has the same set of instances of SATISFIABILITY, but its set of YES instances corresponds to the set of NO instances of SATISFIABILITY, and vice versa. Checking whether an instance is a YES instance is a process that, in the case of SATISFIABILITY, can stop as soon as a satisfying truth assignment has been found, while, in the case of FALSITY, it has to analyze every truth assignment in order to be sure that no satisfying one exists. The relation between SATISFIABILITY and FALSITY is an example of complementarity between decision problems.

◂ Definition 1.9 *Complementary problem*

For any decision problem $\mathcal{P}$ the complementary problem $\mathcal{P}^c$ is the decision problem with $I_{\mathcal{P}^c} = I_{\mathcal{P}}$, $Y_{\mathcal{P}^c} = N_{\mathcal{P}}$, and $N_{\mathcal{P}^c} = Y_{\mathcal{P}}$.

The class of all decision problems which are complementary to decision problems in NP is called co–NP (e.g., FALSITY belongs to co–NP).

1.3 Reducibility among problems

REDUCTIONS ARE a basic tool for establishing relations among the complexities of different problems. Basically, a reduction from a problem $\mathcal{P}_1$ to a problem $\mathcal{P}_2$ presents a method for solving $\mathcal{P}_1$ using an algorithm for $\mathcal{P}_2$. Notice that, broadly speaking, this means that $\mathcal{P}_2$ is at least as difficult as $\mathcal{P}_1$ (since we can solve the latter if we are able to solve the former), provided the reduction itself only involves "simple enough" calculations.

1.3.1 Karp and Turing reducibility

Different types of reducibilities can be defined, as a consequence of the assumption on how a solution to problem $\mathcal{P}_2$ can be used to solve $\mathcal{P}_1$.

◂ Definition 1.10 *Karp reducibility*

A decision problem $\mathcal{P}_1$ is said to be Karp-reducible *(or many-to-one reducible) to a decision problem $\mathcal{P}_2$ if there exists an algorithm $\mathcal{R}$ which* *given any instance $x \in I_{\mathcal{P}_1}$ of $\mathcal{P}_1$, transforms it into an instance $y \in I_{\mathcal{P}_2}$ of $\mathcal{P}_2$ in such a way that $x \in Y_{\mathcal{P}_1}$ if and only if $y \in Y_{\mathcal{P}_2}$. In such a case, $\mathcal{R}$ is said to be a* Karp-reduction *from $\mathcal{P}_1$ to $\mathcal{P}_2$ and we write $\mathcal{P}_1 \leq_m \mathcal{P}_2$. If both $\mathcal{P}_1 \leq_m \mathcal{P}_2$ and $\mathcal{P}_2 \leq_m \mathcal{P}_1$ we say that $\mathcal{P}_1$ and $\mathcal{P}_2$ are* Karp-equivalent *(in symbols $\mathcal{P}_1 \equiv_m \mathcal{P}_2$).*

As a consequence of this definition, if a decision problem $\mathcal{P}_1$ is Karp-reducible to a decision problem $\mathcal{P}_2$, then for any instance x of $\mathcal{P}_1$, x is a positive instance if and only if the transformed instance y is a positive instance for $\mathcal{P}_2$.

It should then be clear that, if a reduction $\mathcal{R}$ from $\mathcal{P}_1$ to $\mathcal{P}_2$ exists and if an algorithm $\mathcal{A}_2$ is known for $\mathcal{P}_2$, an algorithm $\mathcal{A}_1$ for $\mathcal{P}_1$ can be obtained as follows (see Fig. 1.3):

1. given $x \in I_{\mathcal{P}_1}$, apply $\mathcal{R}$ to x and obtain $y \in I_{\mathcal{P}_2}$;
2. apply $\mathcal{A}_2$ to y: if $\mathcal{A}_2$ returns YES then return YES, otherwise ($\mathcal{A}_2$ returns NO) return NO.

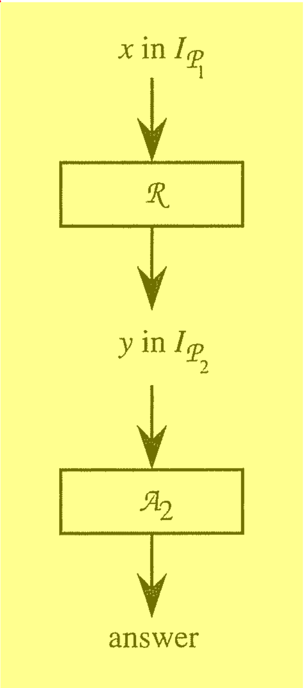


Figure 1.3
How to use a Karp reduction

As a particular case, we define the *polynomial-time Karp-reducibility* by saying that $\mathcal{P}_1$ is polynomially Karp-reducible to $\mathcal{P}_2$ if and only if $\mathcal{P}_1$ is Karp-reducible to $\mathcal{P}_2$ and the corresponding reduction $\mathcal{R}$ is a polynomial-time algorithm. In such a case we write $\mathcal{P}_1 \leq_m^p \mathcal{P}_2$ (again, if both $\mathcal{P}_1 \leq_m^p \mathcal{P}_2$ and $\mathcal{P}_2 \leq_m^p \mathcal{P}_1$ then $\mathcal{P}_1 \equiv_m^p \mathcal{P}_2$).

In the case $\mathcal{P}_1 \leq_m^p \mathcal{P}_2$, efficiently solving $\mathcal{P}_2$, i.e. in polynomial time, implies that also $\mathcal{P}_1$ can be solved efficiently, that is, $\mathcal{P}_2 \in \mathrm{P}$ implies $\mathcal{P}_1 \in \mathrm{P}$. On the contrary, if it is proved that $\mathcal{P}_1$ cannot be solved in polynomial time, then also $\mathcal{P}_2$ cannot be solved efficiently, that is, $\mathcal{P}_1 \notin \mathrm{P}$ implies $\mathcal{P}_2 \notin \mathrm{P}$.

Do Exercise 1.7 (transitivity of polynomial Karp reducibility). Beware the "output size".

Problem 1.5: $\{0,1\}$-Linear Programming

INSTANCE: Set of variables $Z = \{z_1, \ldots, z_n\}$ with domain $\{0,1\}$, set I of linear inequalities on Z.

QUESTION: Does there exist any solution of I, that is, any assignment of values to variables in Z such that all inequalities are verified?

◀ Example 1.10

Let us consider $\{0,1\}$-LINEAR PROGRAMMING, that is, Problem 1.5. It is not hard to see that SATISFIABILITY$\leq^p_m$$\{0,1\}$-LINEAR PROGRAMMING. In fact, any instance $x_{SAT} = (V, \mathcal{F})$ of SATISFIABILITY can be reduced to some instance $x_{LP} = (Z, I)$ of $\{0,1\}$-LINEAR PROGRAMMING as follows. Let $l_{j_1} \vee l_{j_2} \vee \cdots \vee l_{j_{n_j}}$ be the j-th clause in $\mathcal{F}$: a corresponding inequality $\zeta_{j_1} + \zeta_{j_2} + \cdots + \zeta_{j_{n_j}} \geq 1$ is derived for x_{LP}, where $\zeta_{j_k} = z_i$ if $t_{j_k} = v_i$ (for some $v_i \in V$) and $\zeta_{j_k} = (1 - z_i)$ if $t_{j_k} = \bar{v}_i$ (for some $v_i \in V$). It is easy to see that any truth assignment $f : V \mapsto \{\text{TRUE}, \text{FALSE}\}$ satisfies $\mathcal{F}$ if and only if all inequalities in I are verified by the value assignment $f' : Z \mapsto \{0,1\}$ such that $f'(z_i) = 1$ if and only if $f(v_i) = \text{TRUE}$. Moreover, the reduction is clearly polynomial-time computable.

l instead of t in this line

Let us now introduce a different, more general, type of reducibility, which can be applied also to problems which are not decision problems and that, basically, models the possibility, in writing a program to solve a problem, to use subprograms for solving another problem as many times as it is required. Such a situation is formalized in complexity theory by the use of *oracles*.

◀ Definition 1.11 *Oracle*

Let $\mathcal{P}$ be the problem of computing a (possibly multivalued) function $f : I_{\mathcal{P}} \mapsto S_{\mathcal{P}}$. An oracle for problem $\mathcal{P}$ is an abstract device which, for any $x \in I_{\mathcal{P}}$, returns a value $f(x) \in S_{\mathcal{P}}$. It is assumed that the oracle may return the value in just one computation step.

◀ Definition 1.12 *Turing reducibility*

Let $\mathcal{P}_1$ be the problem of computing a (possibly multivalued) function $g : I_{\mathcal{P}_1} \mapsto S_{\mathcal{P}_1}$. Problem $\mathcal{P}_1$ is said to be Turing-reducible (see Fig. 1.4) to a problem $\mathcal{P}_2$ if there exists an algorithm $\mathcal{R}$ which solves $\mathcal{P}_1$ by querying an oracle for $\mathcal{P}_2$. In such a case, $\mathcal{R}$ is said to be a Turing-reduction *from $\mathcal{P}_1$ to $\mathcal{P}_2$ and we write $\mathcal{P}_1 \leq_T \mathcal{P}_2$.*

Again, if $\mathcal{P}_1 \leq_T \mathcal{P}_2$ and $\mathcal{P}_2 \leq_T \mathcal{P}_1$ we have that $\mathcal{P}_1$ and $\mathcal{P}_2$ are Turing-equivalent ($\mathcal{P}_1 \equiv_T \mathcal{P}_2$).

Clearly, Karp-reducibility is just a particular case of Turing-reducibility, corresponding to the case when $\mathcal{P}_1$ and $\mathcal{P}_2$ are both decision problems, the oracle for $\mathcal{P}_2$ is queried just once, and $\mathcal{R}$ returns the same value answered by the oracle. In general, Karp-reducibility is weaker than Turing-reducibility. For example, for any decision problem $\mathcal{P}$, $\mathcal{P}$ is always Turing-reducible to $\mathcal{P}^c$ (and vice versa), while the same does not hold for Karp-reducibility (see Bibliographical notes).

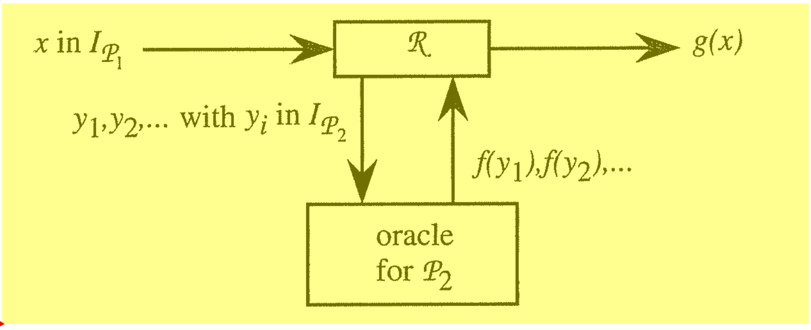

x in $I_{\mathcal{P}_1}$
$\mathcal{R}$
$g(x)$
$y_1, y_2, \ldots$ with y_i in $I_{\mathcal{P}_2}$
$f(y_1), f(y_2), \ldots$
oracle for $\mathcal{P}_2$

Figure 1.4
Turing-reduction from $\mathcal{P}_1$ to $\mathcal{P}_2$

As in the case of Karp-reducibility, we may also introduce a polynomial Turing-reducibility $\leq_T^p$, by saying that $\mathcal{P}_1 \leq_T^p \mathcal{P}_2$ if and only if $\mathcal{P}_1 \leq_T \mathcal{P}_2$ and the Turing-reduction $\mathcal{R}$ is polynomial-time computable with respect to the input size.

Again, in the case $\mathcal{P}_1 \leq_T^p \mathcal{P}_2$, efficiently solving $\mathcal{P}_2$ (in polynomial time) implies that also $\mathcal{P}_1$ can be solved efficiently, that is, $\mathcal{P}_2 \in \mathrm{P}$ implies $\mathcal{P}_1 \in \mathrm{P}$. On the contrary, if it is proved that $\mathcal{P}_1$ cannot be solved in polynomial time, then also $\mathcal{P}_2$ cannot be solved efficiently, that is, $\mathcal{P}_1 \notin \mathrm{P}$ implies $\mathcal{P}_2 \notin \mathrm{P}$.

Example 1.11 ▶ Given two CNF Boolean formulas $\mathcal{F}_1$ and $\mathcal{F}_2$, the EQUIVALENT FORMULAS problem consists of deciding whether $\mathcal{F}_1$ is *equivalent* to $\mathcal{F}_2$, that is, if, for any assignment of values, $\mathcal{F}_1$ is satisfied if and only if $\mathcal{F}_2$ is satisfied.

It is easy to verify that SATISFIABILITY can be solved in polynomial time by a deterministic Turing machine with oracle EQUIVALENT FORMULAS.

Indeed, let $\mathcal{F}$ be a Boolean formula in conjunctive normal form. To decide whether $\mathcal{F}$ is satisfiable it is sufficient to check whether $\mathcal{F}$ is equivalent to $\mathcal{F}_f = x \wedge \bar{x}$ (observe that $\mathcal{F}_f$ cannot be satisfied by any assignment of values). If this is the case, $\mathcal{F}$ is not satisfiable, otherwise it is satisfiable. Clearly, this check can be done by querying the oracle EQUIVALENT FORMULAS about the instance formed by $\mathcal{F}$ and $\mathcal{F}_f$. If the oracle answers positively, $\mathcal{F}$ is not satisfiable, otherwise it is satisfiable.

Let us now introduce two definitions valid for any complexity class and type of reducibility.

Definition 1.13 ▶ *Complexity class closure*
A complexity class C is said to be closed *with respect to a reducibility $\leq_r$ if, for any pair of decision problems $\mathcal{P}_1$, $\mathcal{P}_2$ such that $\mathcal{P}_1 \leq_r \mathcal{P}_2$, $\mathcal{P}_2 \in C$ implies $\mathcal{P}_1 \in C$.*

Definition 1.14 ▶ *Complete problem*
For any complexity class C, a decision problem $\mathcal{P} \in C$ is said to be complete *in C (equivalently, C-complete) with respect to a reducibility $\leq_r$ if, for any other decision problem $\mathcal{P}_1 \in C$, $\mathcal{P}_1 \leq_r \mathcal{P}$.*

The above definition immediately implies that, for any two problems $\mathcal{P}_1$ and $\mathcal{P}_2$ which are C-complete with respect to a reducibility $\leq_r$, $\mathcal{P}_1 \equiv_r \mathcal{P}_2$.

It is easy to see that, for any pair of complexity classes C_1 and C_2 such that $C_1 \subset C_2$ and C_1 is closed with respect to a reducibility $\leq_r$, any C_2-complete problem $\mathcal{P}$ belongs to $C_2 - C_1$. In fact, for any problem $\mathcal{P}_1 \in C_2 - C_1$, $\mathcal{P}_1 \leq_r \mathcal{P}$ by the completeness of $\mathcal{P}$. Then, by the closure of C_1, $\mathcal{P} \in C_1$ would imply $\mathcal{P}_1 \in C_1$, which is a contradiction. This property suggests that if $C_1 \subseteq C_2$ then the best approach to determining whether $C_1 \subset C_2$ or $C_1 = C_2$ is to study the complexity of C_2-complete problems. In fact, if for any C_2-complete problem $\mathcal{P}$ we prove that $\mathcal{P} \in C_1$, then we have that $C_1 = C_2$; conversely, if we have that $\mathcal{P} \notin C_1$ then $C_1 \subset C_2$.

1.3.2 NP-complete problems

The above definitions turn out to be particularly relevant when applied to the class NP in order to define complete problems in NP with respect to Karp-reductions.

◀ Definition 1.15 *NP-complete problem*

A decision problem $\mathcal{P}$ is said to be NP*-complete if it is complete in* NP *with respect to $\leq_m^p$, that is, $\mathcal{P} \in$* NP *and, for any decision problem $\mathcal{P}_1 \in$* NP*, $\mathcal{P}_1 \leq_m^p \mathcal{P}$.*

Since P $\subseteq$ NP, by the closure of NP with respect to $\leq_m^p$ (see Exercise 1.10), P $\neq$ NP if and only if for each NP-complete problem $\mathcal{P}$, $\mathcal{P} \notin$ P. At the moment, it is not known whether polynomial-time algorithms exist for NP-complete problems and only superpolynomial time algorithms are currently available, even if no superpolynomial lower bound on the time complexity has been proved for any of these problems.

◀ Example 1.12

SATISFIABILITY is NP-complete. We have already seen in Example 1.9 that this problem is in NP. Showing its NP-completeness is quite a harder task, which we will present in Chap. 6.

A polynomial-time algorithm
for SATISFIABILITY
implies
P = NP
implies
1,000,000 $

By the definition of NP-completeness and by the transitive property of polynomial-time Karp reductions (see Exercise 1.7), any problem $\mathcal{P}$ can be proved to be NP-complete by first showing a polynomial-time nondeterministic algorithm for $\mathcal{P}$ (which proves that $\mathcal{P} \in$ NP) and then providing a polynomial-time Karp-reduction from some other problem $\mathcal{P}'$, already known to be NP-complete, to $\mathcal{P}$.

◀ Example 1.13

{0,1}-LINEAR PROGRAMMING is NP-complete. Indeed, this problem can be shown to be in NP by considering a polynomial-time nondeterministic algorithm similar to the one considered for SATISFIABILITY (that is, Program 1.3).

The NP-completeness can be shown by providing a polynomial-time Karp-reduction $\mathcal{R}$ from any other problem which is known to be NP-complete. In particular, a polynomial-time Karp-reduction from SATISFIABILITY has been shown in Example 1.10.

As said above, it is not known whether NP-complete problems can be solved efficiently or not. However, due to the fact that even after a wide effort from the whole computer science community, polynomial-time algorithms for such problems (now numbering in the thousands) are still lacking, NP-completeness is considered as a sign that the considered problem is intractable.

1.4 Complexity of optimization problems

LET US now turn our attention to optimization problems. As already observed, most of the basic concepts of complexity theory have been introduced with reference to decision problems. In order to extend the theoretical setting to optimization problems we need to reexamine such concepts and consider how they apply to optimization.

1.4.1 Optimization problems

The study of the cost of solving optimization problems is probably one of the most relevant practical aspects of complexity theory, due to the importance of such problems in many application areas.

In order to extend complexity theory from decision problems to optimization problems, suitable definitions have to be introduced. Also, the relationships between the complexity of optimization problems and the complexity of decision problems have to be discussed.

First, let us provide a formal definition of an optimization problem.

Definition 1.16 ▶ *Optimization problem*

An optimization problem $\mathcal{P}$ is characterized by the following quadruple of objects $(I_{\mathcal{P}}, \mathrm{SOL}_{\mathcal{P}}, m_{\mathcal{P}}, \mathrm{goal}_{\mathcal{P}})$, where:

1. *$I_{\mathcal{P}}$ is the set of instances of $\mathcal{P}$;*

2. *$\mathrm{SOL}_{\mathcal{P}}$ is a function that associates to any input instance $x \in I_{\mathcal{P}}$ the set of* feasible solutions *of x;*

3. *$m_{\mathcal{P}}$ is the* measure function, *defined for pairs (x,y) such that $x \in I_{\mathcal{P}}$ and $y \in \mathrm{SOL}_{\mathcal{P}}(x)$. For every such pair (x,y), $m_{\mathcal{P}}(x,y)$ provides a*

(also called "Objective function")

Problem 1.6: Minimum Vertex Cover

INSTANCE: Graph $G = (V, E)$.

SOLUTION: A subset of nodes $U \subseteq V$ such that $\forall (v_i, v_j) \in E$, $v_i \in U$ or $v_j \in U$.

MEASURE: $|U|$.

positive integer which is the value of the feasible solution y;[1]

4. $\text{goal}_{\mathcal{P}} \in \{\text{MIN}, \text{MAX}\}$ *specifies whether $\mathcal{P}$ is a maximization or a minimization problem.*

Given an input instance x, we denote by $\text{SOL}^*_{\mathcal{P}}(x)$ the set of *optimal solutions* of x, that is the set of solutions whose value is optimal (minimum or maximum depending on whether $\text{goal}_{\mathcal{P}} = \text{MIN}$ or $\text{goal}_{\mathcal{P}} = \text{MAX}$). More formally, for every $y^*(x)$ such that $y^*(x) \in \text{SOL}^*_{\mathcal{P}}(x)$:

$$m_{\mathcal{P}}(x, y^*(x)) = \text{goal}_{\mathcal{P}}\{v \mid v = m_{\mathcal{P}}(x, z) \wedge z \in \text{SOL}_{\mathcal{P}}(x)\}.$$

The value of any optimal solution $y^*(x)$ of x will be denoted as $m^*_{\mathcal{P}}(x)$. In the following, whenever the problem we are referring to is clear from the context we will not use the subscript that explicitly refers to the problem $\mathcal{P}$.

◀ Example 1.14

Given a graph $G = (V, E)$, the MINIMUM VERTEX COVER problem is to find a vertex cover of minimum size, that is a minimum set U of nodes such that, for each edge $(v_i, v_j) \in E$, either $v_i \in U$ or $v_j \in U$ (that is to say, at least one among v_i and v_j must belong to U). Formally, this problem is defined as follows:

1. $I = \{G = (V, E) \mid G \text{ is a graph}\}$;
2. $\text{SOL}(G) = \{U \subseteq V \mid \forall (v_i, v_j) \in E[v_i \in U \vee v_j \in U]\}$;
3. $m(G, U) = |U|$;
4. $\text{goal} = \text{MIN}$.

In the following, however, whenever a new problem will be introduced, we will make use of the more intuitive notation shown in Problem 1.6.

It is important to notice that any optimization problem $\mathcal{P}$ has an associated decision problem $\mathcal{P}_D$. In the case that $\mathcal{P}$ is a minimization problem, $\mathcal{P}_D$

[1] Notice that, in practice, for several problems the measure function is defined to have values in $\mathbb{Q}$. It is however possible to transform any such optimization problem into an equivalent one satisfying our definition.

Problem 1.7: Vertex cover

INSTANCE: Graph $G = (V, E)$, $K \in \mathbb{N}$.

QUESTION: Does there exist a vertex cover on G of size $\leq K$, that is a subset $U \subseteq V$ such that $|U| \leq K$ and $\forall (u, v) \in E$, $u \in U$ or $v \in U$?

asks, for some $K > 0$, for the existence of a feasible solution y of instance x with value $m(x,y) \leq K$. Analogously, if $\mathcal{P}$ is a maximization problem, the associated decision problem asks, given $K > 0$, for the existence of a feasible solution y of x with $m(x,y) \geq K$. Moreover, an evaluation problem $\mathcal{P}_E$ can also be associated with $\mathcal{P}$, which asks for the value of an optimal solution of $\mathcal{P}$.

More precisely, we can say that the definition of an optimization problem $\mathcal{P}$ naturally leads to the following three different problems, corresponding to different ways of approaching its solution.

Constructive Problem ($\mathcal{P}_C$) – Given an instance $x \in I$, derive an optimal solution $y^*(x) \in \mathrm{SOL}^*(x)$ and its measure $m^*(x)$.

Evaluation Problem ($\mathcal{P}_E$) – Given an instance $x \in I$, derive the value $m^*(x)$.

Decision Problem ($\mathcal{P}_D$) – Given an instance $x \in I$ and a positive integer $K \in Z^+$, decide whether $m^*(x) \geq K$ (if goal = MAX) or whether $m^*(x) \leq K$ (if goal = MIN). If goal = MAX, the set $\{(x,K) \mid x \in I \wedge m^*(x) \geq K\}$ (or $\{(x,K) \mid x \in I \wedge m^*(x) \leq K\}$ if goal = MIN) is called the *underlying language* of $\mathcal{P}$.

Example 1.15 ▶ The decision problem relative to MINIMUM VERTEX COVER is Problem 1.7. This problem can be shown to be NP-complete (see Exercise 1.12).

Notice that, for any optimization problem $\mathcal{P}$, the corresponding decision problem $\mathcal{P}_D$ is not harder than the constructive problem $\mathcal{P}_C$. In fact, to answer $\mathcal{P}_D$ on instance x it is sufficient to run some algorithm for $\mathcal{P}_C$, thus obtaining the optimal solution $y^*(x)$ together with its value $m(x, y^*(x))$; then, it is sufficient to check whether $m(x, y^*(x)) \leq K$, in the minimization case, or whether $m(x, y^*(x)) \geq K$, in the maximization case.

Example 1.16 ▶ Let us consider MINIMUM TRAVELING SALESPERSON (TSP), that is, Problem 1.8. An instance of this problem can also be represented by a complete graph $G = (V, E)$ of n vertices with positive weights on the edges (the vertices represent the cities and the weight of an edge is equal to the distance between the corresponding pair of cities). Feasible solutions of the problem are then subsets I of

Problem 1.8: Minimum Traveling Salesperson

INSTANCE: Set of cities $\{c_1, \dots, c_n\}$, $n \times n$ matrix D of distances in Z^+.

SOLUTION: A (traveling salesperson) tour of all cities, that is, a permutation $\{c_{i_1}, \dots, c_{i_n}\}$.

MEASURE: $\sum_{k=1}^{n-1} D(i_k, i_{k+1}) + D(i_n, i_1)$.

Problem 1.9: Minimum Graph Coloring

INSTANCE: Graph $G = (V, E)$.

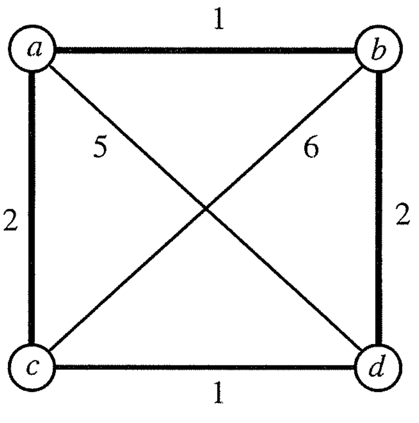
SOLUTION: An assignment $f : V \mapsto \{1, \dots, K\}$ of K colors to the vertices of G such that $\forall (u, v) \in E$, $f(u) \neq f(v)$.

MEASURE: K.

edges such that the graph (V, I) is a cycle. In Fig. 1.5 it is shown an instance of MINIMUM TRAVELING SALESPERSON in both ways (observe that in this case the distance matrix is symmetric): the thick edges in the graph denote the optimal tour. Since the problem is an optimization problem it cannot belong to NP, but it has a corresponding decision problem that is NP-complete (see Bibliographical notes).

	a	*b*	*c*	*d*
a	0	1	2	5
b	1	0	6	2
c	2	6	0	1
d	5	2	1	0

(a)



(b)

Figure 1.5
An instance of MINIMUM TRAVELING SALESPERSON represented (a) as a matrix and (b) as a graph

Example 1.17

The problem MINIMUM GRAPH COLORING is defined as follows (see Problem 1.9): given a graph $G = (V, E)$, find a vertex coloring with a minimum number of colors, that is a partition of V in a minimum number of classes $\{V_1, \dots, V_K\}$ such that for any edge $(u, v) \in E$, u and v are in different classes. For example, in the left side of Fig. 1.6 a sample graph is given with a coloring using 4 colors.

Note that such a coloring is not optimal: a minimum coloring for the same graph is shown in the right side of the same figure. As above, the problem cannot belong to NP, but the corresponding decision problem can be shown to be NP-complete (see Exercise 1.14).

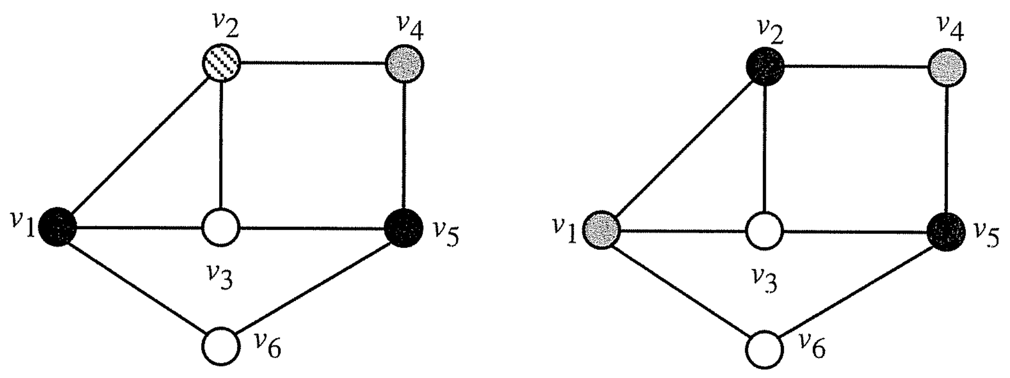


Figure 1.6
A coloring of a graph with 4 colors and an optimal coloring requiring 3 colors

1.4.2 PO and NPO problems

In order to characterize the complexity of optimization problems and to classify such problems accordingly, various approaches may be followed.

The most direct point of view is to consider the time needed for solving the given problem and to extend to optimization problems the theory developed for decision problems.

Of course, the most relevant issue is to characterize optimization problems $\mathcal{P}$ which can be considered tractable, i.e. such that there exists a polynomial-time computable algorithm $\mathcal{A}$ that, for any instance $x \in I$ returns an optimal solution $y \in \text{SOL}^*(x)$, together with its value $m^*(x)$. This means that our main concern will be the study of constructive versions of optimization problems: this will indeed be the approach that will be followed throughout the book (even though we will usually avoid to explicitly return the measure of the computed solution).

Example 1.18 ▶ The problem MINIMUM PATH is defined as follows (see Problem 1.10): given a graph $G = (V, E)$ and a pair of nodes v_s and v_d, derive the shortest path from v_s to v_d. This problem can be proved to be tractable by showing the polynomial-time Program 1.4 which constructs all minimum paths from all nodes in V to v_d. The algorithm visits the graph in a breadth-first order and constructs a tree with root v_d by setting in any node a pointer π to its parent node. An example of the behavior of the algorithm is shown in Fig. 1.7 where the lower part represents the resulting tree obtained with input the graph depicted in the upper part and $v_d = v_1$: the arrows denote the pointers to the parents while the values in the square brackets represent the visiting order.

Problem 1.10: Minimum Path

INSTANCE: Graph $G = (V, E)$, two nodes $v_s, v_d \in V$.

SOLUTION: A path $(v_s = v_{i_1}, v_{i_2}, \ldots, v_{i_k} = v_d)$ from v_s to v_d.

MEASURE: The number k of nodes in the path.

Program 1.4: Shortest path by breadth-first search

```
input Graph $G = (V, E)$ and two nodes $v_s, v_d \in V$;
output Minimum path from $v_s$ to $v_d$ in $G$;
begin
   Enqueue($v_d, Q$);
   Mark node $v_d$ as visited;
   $\pi[v_d]$ := nil ;
   while $Q$ is not empty do
   begin
      $v$ := Dequeue($Q$);
      for each node $u$ adjacent to $v$ do
         if $u$ has not been already visited then
         begin
            Enqueue($u, Q$);
            Mark $u$ as visited;
            $\pi[u]$ := $v$;
         end
   end;
   if $v_s$ has been visited then
      return the path from $v_s$ to $v_d$ by following pointers $\pi$;
end.
```

In this book we are mainly interested in those optimization problems which stand on the borderline between tractability and intractability and which, by analogy with the NP decision problems, are called NPO problems.

◀ Definition 1.17
Class NPO

An optimization problem $\mathcal{P} = (I, \mathrm{SOL}, m, \mathrm{goal})$ belongs to the class NPO *if the following holds:*

1. *the set of instances I is recognizable in polynomial time;*
2. *there exists a polynomial q such that, given an instance $x \in I$, for any $y \in \mathrm{SOL}(x)$, $|y| \leq q(|x|)$ and, besides, for any y such that $|y| \leq q(|x|)$, it is decidable in polynomial time whether $y \in \mathrm{SOL}(x)$;*
3. *the measure function m is computable in polynomial time.*

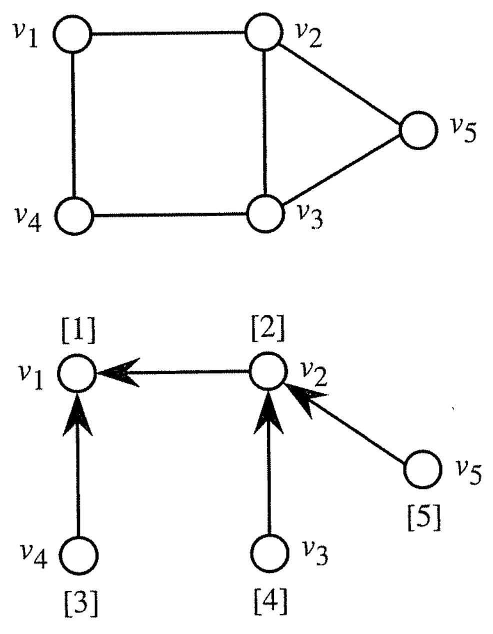


Figure 1.7
An example of application of Program 1.4

Example 1.19 ► MINIMUM VERTEX COVER belongs to NPO since:

1. the set of instances (any undirected graph) is clearly recognizable in polynomial time;
2. since any feasible solution is a subset of the set of nodes its size is smaller than the size of the instance; moreover, testing that a subset $U \subseteq V$ is a feasible solution requires testing whether any edge in E is incident to at least one node in U, which can be clearly performed in polynomial time;
3. given a feasible solution U, the measure function (size of U) is trivially computable in polynomial time.

Even if not explicitly introduced, underlying the definition of NPO problem there is a nondeterministic computation model. This is formally stated by the following result which basically shows that the class NPO is the natural optimization counterpart of the class NP.

Theorem 1.1 ► *For any optimization problem $\mathcal{P}$ in* NPO, *the corresponding decision problem $\mathcal{P}_D$ belongs to* NP.

PROOF Assume that $\mathcal{P}$ is a maximization problem (the proof in the minimization case is similar). Given an instance $x \in I$ and an integer K, we can solve $\mathcal{P}_D$ by performing the following nondeterministic algorithm. In time $q(|x|)$, where q is a polynomial, any string y such that $|y| \leq q(|x|)$ is guessed. Afterwards the string is tested for membership in $\mathrm{SOL}(x)$ in polynomial time. If the test is positive, $m(x,y)$ is computed (again in polynomial time) and if $m(x,y) \geq K$ the answer YES is returned. Otherwise (i.e., either y is not a feasible solution or $m(x,y) < K$), the answer NO is returned. QED

The relationship between classes NP and NPO, which holds in the case of nondeterministic computations, can be translated, in the case of deterministic algorithms, by the following definition, that introduces the class of NPO problems whose constructive versions are efficiently solvable.

◀ Definition 1.18
Class PO

An optimization problem $\mathcal{P}$ belongs to the class PO *if it is in* NPO *and there exists a polynomial-time computable algorithm $\mathcal{A}$ that, for any instance $x \in I$, returns an optimal solution $y \in \mathrm{SOL}^*(x)$, together with its value $m^*(x)$.*

◀ Example 1.20

MINIMUM PATH belongs to PO. Indeed, it is easy to see that this problem satisfies the three conditions of Def. 1.17. Moreover, as we have seen in Example 1.18, MINIMUM PATH is solvable in polynomial time.

Practically all interesting optimization problems belong to the class NPO: in addition to all optimization problems in PO (such as MINIMUM PATH), several other problems of great practical relevance belong to NPO. Beside MINIMUM VERTEX COVER, MINIMUM TRAVELING SALESPERSON, and MINIMUM GRAPH COLORING, many other graph optimization problems, most packing and scheduling problems, and the general formulations of integer and binary linear programming belong to NPO but are not known to be in PO since no polynomial-time algorithm for them is known (and it is likely that none exists). For some of them, not only does the exact solution appear to be extremely complex to obtain but, as we will see, even to achieve good approximate solutions may be computationally hard.

Indeed, for all the optimization problems in NPO − PO the intrinsic complexity is not precisely known. Just as with the decision problems in NP − P, no polynomial-time algorithms for them have been found but no proof of intractability is known either. In fact, the question "PO = NPO?" is strictly related to the question "P = NP?" since it can be proved that the two questions are equivalent in the sense that a positive answer to the first would imply a positive answer to the second and vice versa. In order to establish such relationships between the two questions we have to make more precise how the complexity of decision problems may be related to the complexity of optimization problems.

1.4.3 NP-hard optimization problems

In order to assess the intrinsic complexity of optimization problems we may think of proceeding as in the case of decision problems. In that case the relative complexity of problems was established by making use of polynomial-time Karp reductions and of the related notion of NP-completeness. Unfortunately, Karp reductions are defined for decision problems and cannot be applied to optimization problems. In this case, instead, we can make use of the polynomial-time Turing reducibility (see Sect. 1.3).

Definition 1.19 ► *NP-hard problem* *An optimization problem $\mathcal{P}$ is called* NP*-hard if, for every decision problem $\mathcal{P}' \in$* NP*, $\mathcal{P}' \leq_T^p \mathcal{P}$, that is, $\mathcal{P}'$ can be solved in polynomial time by an algorithm which uses an oracle that, for any instance $x \in I_{\mathcal{P}}$, returns an optimal solution $y^*(x)$ of x along with its value $m^*_{\mathcal{P}}(x)$.*[2]

Thus, a problem is NP-hard if it is at least as difficult to solve (in terms of time complexity and apart from a polynomial-time reduction) as any problem in NP. As a consequence of the definition of NP-completeness, in order to prove that an optimization problem $\mathcal{P}$ is NP-hard it is enough to show that $\mathcal{P}' \leq_T^p \mathcal{P}$ for an NP-complete problem $\mathcal{P}'$. Besides, if a problem $\mathcal{P}$ is NP-hard, $\mathcal{P} \in$ PO implies P=NP.

Indeed, many interesting problems are NP-hard. For example, this happens with all problems in NPO whose underlying language is NP-complete.

Theorem 1.2 ► *Let a problem $\mathcal{P} \in$* NPO *be given; if the underlying language of $\mathcal{P}$ is* NP*-complete then $\mathcal{P}$ is* NP*-hard.* = decision problem

PROOF Clearly, the solution of the decision problem could be obtained for free if an oracle would give us the solution of the constructive optimization problem.
QED

From the preceding result we may easily derive a first consequence concerning the complexity of NPO problems. In fact it turns out that if we could solve the problem $\mathcal{P}$ of Theorem 1.2 in polynomial time we could also solve its underlying decision problem in polynomial time. Hence, unless P = NP, no problem in NPO whose underlying language is NP-complete (e.g., MINIMUM TRAVELING SALESPERSON) can belong to PO and the next result follows.

Corollary 1.3 ► *If* P $\neq$ NP *then* PO $\neq$ NPO.

The fact that an NPO problem $\mathcal{P}$ is NP-hard naturally places $\mathcal{P}$ at the highest level of complexity in the class NPO, like what happens in NP with the NP-complete problems.

Example 1.21 ► As a consequence of Exercises 1.12 and 1.14 and of Example 1.16, we have that MINIMUM VERTEX COVER, MINIMUM GRAPH COLORING, and MINIMUM TRAVELING SALESPERSON are NP-hard.

[2]More formally, and according to the definition of Turing reducibility (i.e., Def. 1.12), we should write $\mathcal{P}' \leq_T^p \text{SOL}^*_{\mathcal{P}}$ instead of $\mathcal{P}' \leq_T^p \mathcal{P}$.

1.4.4 Optimization problems and evaluation problems

As we have seen, the results of the preceding section have provided us with some information on the relative complexity of decision problems and optimization problems in some particular cases.

Let us now see the mutual relations existing among the various versions of optimization problems (decision, evaluation, and constructive problems) in a more systematic manner.

First of all, let us state more formally three results that have already been mentioned in the preceding section.

◀ Theorem 1.4

For any problem $\mathcal{P} \in$ NPO, $\mathcal{P}_D \equiv_T^p \mathcal{P}_E \leq_T^p \mathcal{P}_C$.

PROOF

The proofs of $\mathcal{P}_D \leq_T^p \mathcal{P}_E$ and of $\mathcal{P}_E \leq_T^p \mathcal{P}_C$ are immediate. Regarding $\mathcal{P}_E \leq_T^p \mathcal{P}_D$, due to the properties of NPO problems we have that, for any $x \in I$ and for any $y \in \mathrm{SOL}(x)$, the range of possible values of $m(x,y)$ is bounded by $M = 2^{p(|x|)}$ for some polynomial p. Hence, by applying binary search, the evaluation problem can be solved by at most $\log M = p(|x|)$ queries to the oracle $\mathcal{P}_D$.

QED

Less clear is the possibility of deriving the constructive solution when knowing only the solution of the evaluation problem.

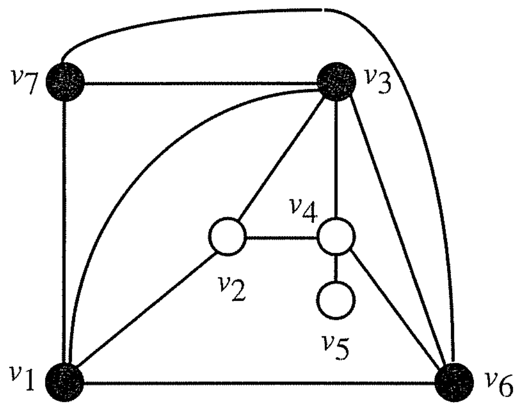


Figure 1.8
A clique of size 4

◀ Example 1.22

Let us consider MAXIMUM CLIQUE, that is, Problem 1.11. An example of a clique of size 4 is shown in Fig. 1.8. Given the solution of the evaluation problem for MAXIMUM CLIQUE we can construct an optimal solution in polynomial time as described by Program 1.5 where, given a node v, $G(v)$ denotes the subgraph induced by v and the set of nodes adjacent to v and $G^-(v)$ denotes the subgraph induced by the set of nodes adjacent to v.

The cost of running Program 1.5 is given by the following recurrence relation as a function T of the number of nodes in the graph:

1. $T(1)$ is $O(1)$,

Problem 1.11: Maximum Clique

INSTANCE: Graph $G = (V, E)$.

SOLUTION: A clique in G, i.e., a subset of nodes $U \subseteq V$ such that $\forall (v_i, v_j) \in U \times U, (v_i, v_j) \in E$ or $v_i = v_j$.

= complete graph

MEASURE: $|U|$.

Program 1.5: Maxclique

```
input Graph $G = (V, E)$;
output Maximum clique in $G$;
begin
   $k$ := MAXIMUM CLIQUE$_E(G)$;
   if $k = 1$ then return any node in $V$;
   Find node $v$ such that MAXIMUM CLIQUE$_E(G(v)) = k$;
   return $\{v\} \cup$ Maxclique$(G^-(v))$
end.
```

2. $T(n) = (n+1) + T(n-1)$

(recall that querying the oracle MAXIMUM CLIQUE$_E$ costs only one step). The solution of the recurrence relation is $O(n^2)$.

Unfortunately, the straightforward construction of the previous example cannot be applied in general. Intuitively it may be that the constructive problem is indeed more complex than the evaluation problem since it yields additional information.

The next result, however, shows that whenever the decision problem is NP-complete the constructive, evaluation, and decision problems are equivalent.

Theorem 1.5 ► *Let $\mathcal{P}$ be an* NPO *problem whose underlying language $\mathcal{P}_D$ is* NP-*complete. Then $\mathcal{P}_C \leq_T^p \mathcal{P}_D$.*

PROOF Let us assume, without loss of generality, that $\mathcal{P}$ is a maximization problem. To prove the theorem we will derive an NPO problem $\mathcal{P}'$ such that $\mathcal{P}_C \leq_T^p \mathcal{P}'_D$, and then use the fact that, since $\mathcal{P}_D$ is NP-complete, $\mathcal{P}'_D \leq_T^p \mathcal{P}_D$ (actually, $\mathcal{P}'_D \leq_m^p \mathcal{P}_D$).

Problem $\mathcal{P}'$ has the same definition of $\mathcal{P}$ except for the measure function $m_{\mathcal{P}'}$, which is defined as follows. Let p be a polynomial which bounds the length of the solutions of $\mathcal{P}$ with respect to the length of the corresponding instances and, for any solution $y \in \text{SOL}_{\mathcal{P}}$, let $\lambda(y)$ denote the rank of y

in the lexicographic order. Then, for any instance $x \in I_{\mathcal{P}'} = I_{\mathcal{P}}$ and for any solution $y \in \mathrm{SOL}_{\mathcal{P}'}(x) = \mathrm{SOL}_{\mathcal{P}}(x)$, let $m_{\mathcal{P}'}(x,y) = 2^{p(|x|)+1} m_{\mathcal{P}}(x,y) + \lambda(y)$.

Notice that, for all instances x of $\mathcal{P}'$ and for any pair $y_1, y_2 \in \mathrm{SOL}_{\mathcal{P}'}(x)$, we have that $m_{\mathcal{P}'}(x,y_1) \neq m_{\mathcal{P}'}(x,y_2)$. Therefore there exists a unique optimal feasible solution $y^*_{\mathcal{P}'}(x)$ in $\mathrm{SOL}^*_{\mathcal{P}'}(x)$. Note also that, by definition, if $m_{\mathcal{P}'}(x,y_1) > m_{\mathcal{P}'}(x,y_2)$ then $m_{\mathcal{P}}(x,y_1) \geq m_{\mathcal{P}}(x,y_2)$, thus implying that $y^*_{\mathcal{P}'}(x) \in \mathrm{SOL}^*_{\mathcal{P}}(x)$.

The optimal solution $y^*_{\mathcal{P}'}(x)$ can be easily derived in polynomial time by means of an oracle for $\mathcal{P}'_E$, since, given $m^*_{\mathcal{P}'}(x)$, the position of $y^*_{\mathcal{P}'}(x)$ in the lexicographic order can be derived by computing the remainder of the division between $m^*_{\mathcal{P}'}(x)$ and $2^{p(|x|)+1}$.

We know that an oracle for $\mathcal{P}'_D$ can be used to simulate $\mathcal{P}'_E$ in polynomial time. Thus we can construct an optimal solution of $\mathcal{P}$ in polynomial time using an oracle for $\mathcal{P}'_D$, and since $\mathcal{P}'_D \in$ NP and $\mathcal{P}_D$ is NP-complete, an oracle for $\mathcal{P}_D$ can be used to simulate the oracle for $\mathcal{P}'_D$. QED

The following question still remains open: Is there an NPO problem $\mathcal{P}$ whose corresponding constructive problem is harder than the evaluation problem $\mathcal{P}_E$? Indeed, there is some evidence that the answer to this question may be affirmative (see Bibliographical notes).

1.5 Exercises

Exercise 1.1 Prove the cost evaluations given in Example 1.1.

Exercise 1.2 Program 1.6 is the Euclidean algorithm for the greatest common divisor of two integers $x, y \in \mathbb{Z}$. Determine its execution cost under the uniform cost model and the logarithmic cost model. Moreover, show which are the dominant operations and derive the asymptotic execution cost.

Exercise 1.3 Given a set P of points in the plane, the convex hull of P is defined as the minimal size convex polygon including all points in P. It can be easily proved that the set of vertices of the convex hull is a subset of P. Prove that the time complexity of the problem of computing the convex hull of a set of n points is:

1. $O(n^2)$;
2. $\Omega(n \log n)$ (by reduction from sorting);
3. (*) $\Theta(n \log n)$.

Program 1.6: Euclid

```
input Integers $x, y$;
output Greatest common divisor $z$ of $x$ and $y$;
begin
  while $x > 0$ do
    if $x > y$ then
      $x := x - y$
    else if $x < y$ then
      $y := y - x$
    else begin
      $z := x$;
      $x := 0$
    end;
  return $z$
end.
```

Problem 1.12: Maximum Path in a Tree

INSTANCE: Tree T, integer $K > 0$.

QUESTION: Is the length of the longest path in T less than K?

Exercise 1.4 Let us denote as LOGSPACE the class of all problems solvable in *work space* proportional to the logarithm of the input size where the work space is the number of memory locations used for performing the computation, not considering the space required for the initial description of the problem instance (for example, we may assume that the problem instance is represented on a read-only memory device, whose size is not considered in the space complexity evaluation). Prove that SATISFYING TRUTH ASSIGNMENT is in LOGSPACE.

Exercise 1.5 Show that Problem 1.12 is in LOGSPACE.

Exercise 1.6 Derive a polynomial-space algorithm solving QUANTIFIED BOOLEAN FORMULAS.

$\longrightarrow$ Exercise 1.7 Show that polynomial-time Karp reductions have the transitive property, that is, if $\mathcal{P}_1 \leq^p_m \mathcal{P}_2$ and $\mathcal{P}_2 \leq^p_m \mathcal{P}_3$, then $\mathcal{P}_1 \leq^p_m \mathcal{P}_3$.

$\longrightarrow$ Exercise 1.8 Define a polynomial-time nondeterministic algorithm for the Problem 1.13.

Exercise 1.9 Recall that a *disjunctive normal form* (DNF) formula is defined as a collection of conjunctions $C = \{c_1, c_2, \ldots, c_m\}$ and is assumed to

Problem 1.13: Subset Sum

INSTANCE: Set $S = \{s_1, s_2, \ldots, s_n\}$, weight function $w : S \mapsto \mathbb{N}$, $K \in \mathbb{N}$.

QUESTION: Does there exist any $S' \subseteq S$ such that $\sum_{s_i \in S'} w(s_i) = K$?

Problem 1.14: Tautology

INSTANCE: Boolean formula $\mathcal{F}$ in DNF.

QUESTION: Is it true that every truth assignment on $\mathcal{F}$ is a satisfying assignment?

be satisfied by a truth assignment f if and only if at least one conjunction c_i, with $1 \leq i \leq m$, is satisfied by f. Show that Problem 1.14 is in co–NP.

Exercise 1.10 Show that NP is closed with respect to polynomial-time Karp-reducibility. Is NP closed also with respect to polynomial-time Turing reducibility?

Exercise 1.11 Prove that if there exists a problem $\mathcal{P}$ such that both $\mathcal{P}$ and $\mathcal{P}^c$ are NP-complete then NP = co–NP.

Exercise 1.12 (*) Prove that VERTEX COVER is NP-complete. (Hint: use the NP-completeness of SATISFIABILITY.)

Exercise 1.13 Prove that the decision problem corresponding to the MAXIMUM CLIQUE problem is NP-complete. (Hint: use the previous exercise).

Exercise 1.14 (*) Prove that the decision problem corresponding to the MINIMUM GRAPH COLORING problem is NP-complete. (Hint: use the NP-completeness of SATISFIABILITY.)

Exercise 1.15 Prove that the problem of deciding whether a graph is colorable with two colors is in P. (Hint: assign a color to a vertex and compute the consequences of this assignment.)

Exercise 1.16 Prove that the optimization problem corresponding to Problem 1.15 is in NPO.

Exercise 1.17 A problem $\mathcal{P}$ is called NP-easy if there exists a decision problem $\mathcal{P}' \in$ NP such that $\mathcal{P} \leq_T^p \mathcal{P}'$. Show that MINIMUM TRAVELING SALESPERSON is NP-easy. (Hint: Use the language $\{(G, p, k) \mid G$ is a graph, p is a path that can be extended to a Hamiltonian tour of cost $\leq k\}$ as the problem $\mathcal{P}'$ in the definition.)

Problem 1.15: 4-th vertex cover

INSTANCE: Graph $G = (V, E)$, integer K.

QUESTION: Is the size of the 4-th smallest vertex cover in G at most equal to K?

Problem 1.16: MAXIMUM SATISFIABILITY

INSTANCE: Set C of disjunctive clauses on a set of variables V.

SOLUTION: A truth assignment $f : V \mapsto \{\text{TRUE}, \text{FALSE}\}$.

MEASURE: The number of clauses in C which are satisfied by f.

Exercise 1.18 Given an oracle for the evaluation version of Problem 1.16, show how it can be exploited to solve the constructive version of the same problem.

1.6 Bibliographical notes

GENERAL INTRODUCTIONS to the theory of algorithms and techniques for their analysis are given in [Knuth, 1969, Knuth, 1971, Knuth, 1973]. Other fundamental references to techniques for the design and analysis of algorithms are [Aho, Hopcroft, and Ullman, 1974] and [Cormen, Leiserson, and Rivest, 1990]. A recent text entirely devoted to formal methods for the analysis of combinatorial algorithms is [Sedgewick and Flajolet, 1996].

A detailed exposition of the theory of computational complexity of decision problems and of structural properties of related complexity classes is provided in several textbooks such as [Balcàzar, Diaz, and Gabarrò, 1988, Bovet and Crescenzi, 1994, Papadimitriou, 1994]. An excellent textbook on the theory of computability is [Rogers, 1987], where a detailed exposition of several types of reducibilities can be found.

The theory of NP-completeness is one of the bases for the topics developed in this textbook: the original concept of NP-complete problems has been introduced independently in [Cook, 1971], where SATISFIABILITY was proved to be NP-complete, and in [Levin, 1973]. The theory was further refined in [Karp, 1972], where a first set of problems were shown to be NP-complete by using reductions.

[Garey and Johnson, 1979] represents a landmark in the literature on NP-completeness and provides a detailed survey of the theory and an extensive list of NP-complete problems. Still twenty years later, this book is a fundamental reference for people interested in the assessing of the computational tractability or intractability of decision problems. The compendium of optimization problems contained in this book and the notation therein adopted are directly inspired by the list of NP-complete problems given in [Garey and Johnson, 1979]. This book is also a source of information on PSPACE-completeness and on the first examples of PSPACE-complete problems, such as QUANTIFIED BOOLEAN FORMULAS.

The characterization of the complexity of optimization problems has been first addressed in [Johnson, 1974a, Ausiello, Marchetti-Spaccamela, and Protasi, 1980, Paz and Moran, 1981], by establishing connections between combinatorial properties and complexity of decision and optimization problems. In particular, the concept of strong NP-completeness introduced in [Garey and Johnson, 1978] proved to be very fruitful in the theory of approximation of optimization problems (see Chap. 3). In [Krentel, 1988] the first structural approach to the characterization of the complexity of optimization problems is provided, by introducing a suitable computation model and the corresponding complexity classes (see Chap. 6). Further studies on the relationship between decision, evaluation, and optimization problems, including Theorem 1.5, are discussed in [Crescenzi and Silvestri, 1990].