
Almost all the algorithms we have studied thus far have been *polynomial-time algorithms*: on inputs of size n , their worst-case running time is $O(n^k)$ for some constant k . You might wonder whether *all* problems can be solved in polynomial time. The answer is no. For example, there are problems, such as Turing’s famous “Hailing Problem,” that cannot be solved by any computer, no matter how much time we allow. There are also problems that can be solved, but not in time $O(n^k)$ for any constant k . Generally, we think of problems that are solvable by polynomial-time algorithms as being tractable, or easy, and problems that require superpolynomial time as being intractable, or hard.

The subject of this chapter, however, is an interesting class of problems, called the “NP-complete” problems, whose status is unknown. No polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them. This so-called $P \neq NP$ question has been one of the deepest, most perplexing open research problems in theoretical computer science since it was first posed in 1971.

Several NP-complete problems are particularly tantalizing because they seem on the surface to be similar to problems that we know how to solve in polynomial time. In each of the following pairs of problems, one is solvable in polynomial time and the other is NP-complete, but the difference between problems appears to be slight:

Shortest vs. longest simple paths: In Chapter 24, we saw that even with negative edge weights, we can find *shortest* paths from a single source in a directed graph $G = (V, E)$ in $O(VE)$ time. Finding a *longest* simple path between two vertices is difficult, however. Merely determining whether a graph contains a simple path with at least a given number of edges is NP-complete.

Euler tour vs. hamiltonian cycle: An *Euler tour* of a connected, directed graph $G = (V, E)$ is a cycle that traverses each *edge* of G exactly once, although it is allowed to visit each vertex more than once. By Problem 22-3, we can determine whether a graph has an Euler tour in only $O(E)$ time and, in fact,

we can find the edges of the Euler tour in $O(E)$ time. A **hamiltonian cycle** of a directed graph $G = (V, E)$ is a simple cycle that contains each vertex in V . Determining whether a directed graph has a hamiltonian cycle is NP-complete. (Later in this chapter, we shall prove that determining whether an *undirected* graph has a hamiltonian cycle is NP-complete.)

2-CNF satisfiability vs. 3-CNF satisfiability: A boolean formula contains variables whose values are 0 or 1; boolean connectives such as \wedge (AND), \vee (OR), and \neg (NOT); and parentheses. A boolean formula is **satisfiable** if there exists some assignment of the values 0 and 1 to its variables that causes it to evaluate to 1. We shall define terms more formally later in this chapter, but informally, a boolean formula is in **k -conjunctive normal form**, or k -CNF, if it is the AND of clauses of ORs of exactly k variables or their negations. For example, the boolean formula $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ is in 2-CNF. (It has the satisfying assignment $x_1 = 1, x_2 = 0, x_3 = 1$.) Although we can determine in polynomial time whether a 2-CNF formula is satisfiable, we shall see later in this chapter that determining whether a 3-CNF formula is satisfiable is NP-complete.

NP-completeness and the classes P and NP

Throughout this chapter, we shall refer to three classes of problems: P, NP, and NPC, the latter class being the NP-complete problems. We describe them informally here, and we shall define them more formally later on.

The class P consists of those problems that are solvable in polynomial time. More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k , where n is the size of the input to the problem. Most of the problems examined in previous chapters are in P.

The class NP consists of those problems that are “verifiable” in polynomial time. What do we mean by a problem being verifiable? If we were somehow given a “certificate” of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem. For example, in the hamiltonian-cycle problem, given a directed graph $G = (V, E)$, a certificate would be a sequence $\langle v_1, v_2, v_3, \dots, v_{|V|} \rangle$ of $|V|$ vertices. We could easily check in polynomial time that $(v_i, v_{i+1}) \in E$ for $i = 1, 2, 3, \dots, |V| - 1$ and that $(v_{|V|}, v_1) \in E$ as well. As another example, for 3-CNF satisfiability, a certificate would be an assignment of values to variables. We could check in polynomial time that this assignment satisfies the boolean formula.

Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being supplied a certificate. We shall formalize this notion later in this chapter, but for now we can believe that $P \subseteq NP$. The open question is whether or not P is a proper subset of NP.

Informally, a problem is in the class NPC—and we refer to it as being **NP-complete**—if it is in NP and is as “hard” as any problem in NP. We shall formally define what it means to be as hard as any problem in NP later in this chapter. In the meantime, we will state without proof that if *any* NP-complete problem can be solved in polynomial time, then *every* problem in NP has a polynomial-time algorithm. Most theoretical computer scientists believe that the NP-complete problems are intractable, since given the wide range of NP-complete problems that have been studied to date—without anyone having discovered a polynomial-time solution to any of them—it would be truly astounding if all of them could be solved in polynomial time. Yet, given the effort devoted thus far to proving that NP-complete problems are intractable—with a conclusive outcome—we cannot rule out the possibility that the NP-complete problems are in fact solvable in polynomial time.

To become a good algorithm designer, you must understand the rudiments of the theory of NP-completeness. If you can establish a problem as NP-complete, you provide good evidence for its intractability. As an engineer, you would then do better to spend your time developing an approximation algorithm (see Chapter 35) or solving a tractable special case, rather than searching for a fast algorithm that solves the problem exactly. Moreover, many natural and interesting problems that on the surface seem no harder than sorting, graph searching, or network flow are in fact NP-complete. Therefore, you should become familiar with this remarkable class of problems.

Overview of showing problems to be NP-complete

The techniques we use to show that a particular problem is NP-complete differ fundamentally from the techniques used throughout most of this book to design and analyze algorithms. When we demonstrate that a problem is NP-complete, we are making a statement about how hard it is (or at least how hard we think it is), rather than about how easy it is. We are not trying to prove the existence of an efficient algorithm, but instead that no efficient algorithm is likely to exist. In this way, NP-completeness proofs bear some similarity to the proof in Section 8.1 of an $\Omega(n \lg n)$ -time lower bound for any comparison sort algorithm; the specific techniques used for showing NP-completeness differ from the decision-tree method used in Section 8.1, however.

We rely on three key concepts in showing a problem to be NP-complete:

Decision problems vs. optimization problems

Many problems of interest are **optimization problems**, in which each feasible (i.e., “legal”) solution has an associated value, and we wish to find a feasible solution with the best value. For example, in a problem that we call **SHORTEST-PATH**,

we are given an undirected graph G and vertices u and v , and we wish to find a path from u to v that uses the fewest edges. In other words, SHORTEST-PATH is the single-pair shortest-path problem in an unweighted, undirected graph. NP-completeness applies directly not to optimization problems, however, but to **decision problems**, in which the answer is simply “yes” or “no” (or, more formally, “1” or “0”).

Although NP-complete problems are confined to the realm of decision problems, we can take advantage of a convenient relationship between optimization problems and decision problems. We usually can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized. For example, a decision problem related to SHORTEST-PATH is PATH: given a directed graph G , vertices u and v , and an integer k , does a path exist from u to v consisting of at most k edges?

The relationship between an optimization problem and its related decision problem works in our favor when we try to show that the optimization problem is “hard.” That is because the decision problem is in a sense “easier,” or at least “no harder.” As a specific example, we can solve PATH by solving SHORTEST-PATH and then comparing the number of edges in the shortest path found to the value of the decision-problem parameter k . In other words, if an optimization problem is easy, its related decision problem is easy as well. Stated in a way that has more relevance to NP-completeness, if we can provide evidence that a decision problem is hard, we also provide evidence that its related optimization problem is hard. Thus, even though it restricts attention to decision problems, the theory of NP-completeness often has implications for optimization problems as well.

Reductions

The above notion of showing that one problem is no harder or no easier than another applies even when both problems are decision problems. We take advantage of this idea in almost every NP-completeness proof, as follows. Let us consider a decision problem A , which we would like to solve in polynomial time. We call the input to a particular problem an **instance** of that problem; for example, in PATH, an instance would be a particular graph G , particular vertices u and v of G , and a particular integer k . Now suppose that we already know how to solve a different decision problem B in polynomial time. Finally, suppose that we have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:

- The transformation takes polynomial time.
- The answers are the same. That is, the answer for α is “yes” if and only if the answer for β is also “yes.”

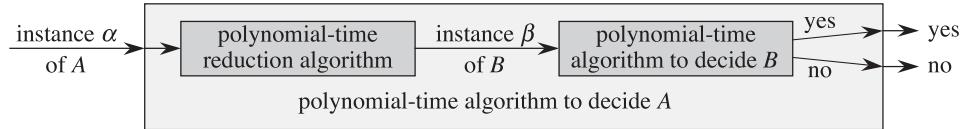


Figure 34.1 How to use a polynomial-time reduction algorithm to solve a decision problem A in polynomial time, given a polynomial-time decision algorithm for another problem B . In polynomial time, we transform an instance α of A into an instance β of B , we solve B in polynomial time, and we use the answer for β as the answer for α .

We call such a procedure a polynomial-time **reduction algorithm** and, as Figure 34.1 shows, it provides us a way to solve problem A in polynomial time:

1. Given an instance α of problem A , use a polynomial-time reduction algorithm to transform it to an instance β of problem B .
2. Run the polynomial-time decision algorithm for B on the instance β .
3. Use the answer for β as the answer for α .

As long as each of these steps takes polynomial time, all three together do also, and so we have a way to decide on α in polynomial time. In other words, by “reducing” solving problem A to solving problem B , we use the “easiness” of B to prove the “easiness” of A .

Recalling that NP-completeness is about showing how hard a problem is rather than how easy it is, we use polynomial-time reductions in the opposite way to show that a problem is NP-complete. Let us take the idea a step further, and show how we could use polynomial-time reductions to show that no polynomial-time algorithm can exist for a particular problem B . Suppose we have a decision problem A for which we already know that no polynomial-time algorithm can exist. (Let us not concern ourselves for now with how to find such a problem A .) Suppose further that we have a polynomial-time reduction transforming instances of A to instances of B . Now we can use a simple proof by contradiction to show that no polynomial-time algorithm can exist for B . Suppose otherwise; i.e., suppose that B has a polynomial-time algorithm. Then, using the method shown in Figure 34.1, we would have a way to solve problem A in polynomial time, which contradicts our assumption that there is no polynomial-time algorithm for A .

For NP-completeness, we cannot assume that there is absolutely no polynomial-time algorithm for problem A . The proof methodology is similar, however, in that we prove that problem B is NP-complete on the assumption that problem A is also NP-complete.

A first NP-complete problem

Because the technique of reduction relies on having a problem already known to be NP-complete in order to prove a different problem NP-complete, we need a “first” NP-complete problem. The problem we shall use is the circuit-satisfiability problem, in which we are given a boolean combinational circuit composed of AND, OR, and NOT gates, and we wish to know whether there exists some set of boolean inputs to this circuit that causes its output to be 1. We shall prove that this first problem is NP-complete in Section 34.3.

Chapter outline

This chapter studies the aspects of NP-completeness that bear most directly on the analysis of algorithms. In Section 34.1, we formalize our notion of “problem” and define the complexity class P of polynomial-time solvable decision problems. We also see how these notions fit into the framework of formal-language theory. Section 34.2 defines the class NP of decision problems whose solutions are verifiable in polynomial time. It also formally poses the $P \neq NP$ question.

Section 34.3 shows we can relate problems via polynomial-time “reductions.” It defines NP-completeness and sketches a proof that one problem, called “circuit satisfiability,” is NP-complete. Having found one NP-complete problem, we show in Section 34.4 how to prove other problems to be NP-complete much more simply by the methodology of reductions. We illustrate this methodology by showing that two formula-satisfiability problems are NP-complete. With additional reductions, we show in Section 34.5 a variety of other problems to be NP-complete.

34.1 Polynomial time

We begin our study of NP-completeness by formalizing our notion of polynomial-time solvable problems. We generally regard these problems as tractable, but for philosophical, not mathematical, reasons. We can offer three supporting arguments.

First, although we may reasonably regard a problem that requires time $\Theta(n^{100})$ to be intractable, very few practical problems require time on the order of such a high-degree polynomial. The polynomial-time computable problems encountered in practice typically require much less time. Experience has shown that once the first polynomial-time algorithm for a problem has been discovered, more efficient algorithms often follow. Even if the current best algorithm for a problem has a running time of $\Theta(n^{100})$, an algorithm with a much better running time will likely soon be discovered.