

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 "Halting 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 strongly 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—without 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”).

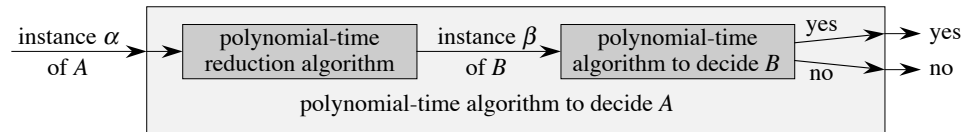
Although NP-complete problems are confined to the realm of decision problems, we can take advantage of a convenient relationship between optimization problems and decision problems. We usually can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized. For example, a decision problem related to SHORTEST-PATH is PATH: given an undirected 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  $\alpha$  of  $A$  into some instance  $\beta$  of  $B$  with the following characteristics:

- The transformation takes polynomial time.
- The answers are the same. That is, the answer for  $\alpha$  is “yes” if and only if the answer for  $\beta$  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  $\alpha$  of  $A$  into an instance  $\beta$  of  $B$ , we solve  $B$  in polynomial time, and we use the answer for  $\beta$  as the answer for  $\alpha$ .

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  $\alpha$  of problem  $A$ , use a polynomial-time reduction algorithm to transform it to an instance  $\beta$  of problem  $B$ .
2. Run the polynomial-time decision algorithm for  $B$  on the instance  $\beta$ .
3. Use the answer for  $\beta$  as the answer for  $\alpha$ .

As long as each of these steps takes polynomial time, all three together do also, and so we have a way to decide on  $\alpha$  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.

Second, for many reasonable models of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another. For example, the class of problems solvable in polynomial time by the serial random-access machine used throughout most of this book is the same as the class of problems solvable in polynomial time on abstract Turing machines.<sup>1</sup> It is also the same as the class of problems solvable in polynomial time on a parallel computer when the number of processors grows polynomially with the input size.

Third, the class of polynomial-time solvable problems has nice closure properties, since polynomials are closed under addition, multiplication, and composition. For example, if the output of one polynomial-time algorithm is fed into the input of another, the composite algorithm is polynomial. Exercise 34.1-5 asks you to show that if an algorithm makes a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then the running time of the composite algorithm is polynomial.

### Abstract problems

To understand the class of polynomial-time solvable problems, we must first have a formal notion of what a “problem” is. We define an **abstract problem**  $Q$  to be a binary relation on a set  $I$  of problem **instances** and a set  $S$  of problem **solutions**. For example, an instance for SHORTEST-PATH is a triple consisting of a graph and two vertices. A solution is a sequence of vertices in the graph, with perhaps the empty sequence denoting that no path exists. The problem SHORTEST-PATH itself is the relation that associates each instance of a graph and two vertices with a shortest path in the graph that connects the two vertices. Since shortest paths are not necessarily unique, a given problem instance may have more than one solution.

This formulation of an abstract problem is more general than we need for our purposes. As we saw above, the theory of NP-completeness restricts attention to **decision problems**: those having a yes/no solution. In this case, we can view an abstract decision problem as a function that maps the instance set  $I$  to the solution set  $\{0, 1\}$ . For example, a decision problem related to SHORTEST-PATH is the problem PATH that we saw earlier. If  $i = \langle G, u, v, k \rangle$  is an instance of the decision problem PATH, then  $\text{PATH}(i) = 1$  (yes) if a shortest path from  $u$  to  $v$  has at most  $k$  edges, and  $\text{PATH}(i) = 0$  (no) otherwise. Many abstract problems are not decision problems, but rather **optimization problems**, which require some value to be minimized or maximized. As we saw above, however, we can usually recast an optimization problem as a decision problem that is no harder.

---

<sup>1</sup>See Hopcroft and Ullman [180] or Lewis and Papadimitriou [236] for a thorough treatment of the Turing-machine model.

## Encodings

In order for a computer program to solve an abstract problem, we must represent problem instances in a way that the program understands. An **encoding** of a set  $S$  of abstract objects is a mapping  $e$  from  $S$  to the set of binary strings.<sup>2</sup> For example, we are all familiar with encoding the natural numbers  $\mathbb{N} = \{0, 1, 2, 3, 4, \dots\}$  as the strings  $\{0, 1, 10, 11, 100, \dots\}$ . Using this encoding,  $e(17) = 10001$ . If you have looked at computer representations of keyboard characters, you probably have seen the ASCII code, where, for example, the encoding of A is 1000001. We can encode a compound object as a binary string by combining the representations of its constituent parts. Polygons, graphs, functions, ordered pairs, programs—all can be encoded as binary strings.

Thus, a computer algorithm that “solves” some abstract decision problem actually takes an encoding of a problem instance as input. We call a problem whose instance set is the set of binary strings a **concrete problem**. We say that an algorithm **solves** a concrete problem in time  $O(T(n))$  if, when it is provided a problem instance  $i$  of length  $n = |i|$ , the algorithm can produce the solution in  $O(T(n))$  time.<sup>3</sup> A concrete problem is **polynomial-time solvable**, therefore, if there exists an algorithm to solve it in time  $O(n^k)$  for some constant  $k$ .

We can now formally define the **complexity class P** as the set of concrete decision problems that are polynomial-time solvable.

We can use encodings to map abstract problems to concrete problems. Given an abstract decision problem  $Q$  mapping an instance set  $I$  to  $\{0, 1\}$ , an encoding  $e : I \rightarrow \{0, 1\}^*$  can induce a related concrete decision problem, which we denote by  $e(Q)$ .<sup>4</sup> If the solution to an abstract-problem instance  $i \in I$  is  $Q(i) \in \{0, 1\}$ , then the solution to the concrete-problem instance  $e(i) \in \{0, 1\}^*$  is also  $Q(i)$ . As a technicality, some binary strings might represent no meaningful abstract-problem instance. For convenience, we shall assume that any such string maps arbitrarily to 0. Thus, the concrete problem produces the same solutions as the abstract problem on binary-string instances that represent the encodings of abstract-problem instances.

We would like to extend the definition of polynomial-time solvability from concrete problems to abstract problems by using encodings as the bridge, but we would

<sup>2</sup>The codomain of  $e$  need not be *binary* strings; any set of strings over a finite alphabet having at least 2 symbols will do.

<sup>3</sup>We assume that the algorithm’s output is separate from its input. Because it takes at least one time step to produce each bit of the output and the algorithm takes  $O(T(n))$  time steps, the size of the output is  $O(T(n))$ .

<sup>4</sup>We denote by  $\{0, 1\}^*$  the set of all strings composed of symbols from the set  $\{0, 1\}$ .



like the definition to be independent of any particular encoding. That is, the efficiency of solving a problem should not depend on how the problem is encoded. Unfortunately, it depends quite heavily on the encoding. For example, suppose that an integer  $k$  is to be provided as the sole input to an algorithm, and suppose that the running time of the algorithm is  $\Theta(k)$ . If the integer  $k$  is provided in **unary**—a string of  $k$  1s—then the running time of the algorithm is  $O(n)$  on length- $n$  inputs, which is polynomial time. If we use the more natural binary representation of the integer  $k$ , however, then the input length is  $n = \lfloor \lg k \rfloor + 1$ . In this case, the running time of the algorithm is  $\Theta(k) = \Theta(2^n)$ , which is exponential in the size of the input. Thus, depending on the encoding, the algorithm runs in either polynomial or superpolynomial time.

How we encode an abstract problem matters quite a bit to how we understand polynomial time. We cannot really talk about solving an abstract problem without first specifying an encoding. Nevertheless, in practice, if we rule out “expensive” encodings such as unary ones, the actual encoding of a problem makes little difference to whether the problem can be solved in polynomial time. For example, representing integers in base 3 instead of binary has no effect on whether a problem is solvable in polynomial time, since we can convert an integer represented in base 3 to an integer represented in base 2 in polynomial time.

We say that a function  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  is **polynomial-time computable** if there exists a polynomial-time algorithm  $A$  that, given any input  $x \in \{0, 1\}^*$ , produces as output  $f(x)$ . For some set  $I$  of problem instances, we say that two encodings  $e_1$  and  $e_2$  are **polynomially related** if there exist two polynomial-time computable functions  $f_{12}$  and  $f_{21}$  such that for any  $i \in I$ , we have  $f_{12}(e_1(i)) = e_2(i)$  and  $f_{21}(e_2(i)) = e_1(i)$ .<sup>5</sup> That is, a polynomial-time algorithm can compute the encoding  $e_2(i)$  from the encoding  $e_1(i)$ , and vice versa. If two encodings  $e_1$  and  $e_2$  of an abstract problem are polynomially related, whether the problem is polynomial-time solvable or not is independent of which encoding we use, as the following lemma shows.

### **Lemma 34.1**

Let  $Q$  be an abstract decision problem on an instance set  $I$ , and let  $e_1$  and  $e_2$  be polynomially related encodings on  $I$ . Then,  $e_1(Q) \in P$  if and only if  $e_2(Q) \in P$ .

---

<sup>5</sup>Technically, we also require the functions  $f_{12}$  and  $f_{21}$  to “map noninstances to noninstances.” A **noninstance** of an encoding  $e$  is a string  $x \in \{0, 1\}^*$  such that there is no instance  $i$  for which  $e(i) = x$ . We require that  $f_{12}(x) = y$  for every noninstance  $x$  of encoding  $e_1$ , where  $y$  is some noninstance of  $e_2$ , and that  $f_{21}(x') = y'$  for every noninstance  $x'$  of  $e_2$ , where  $y'$  is some noninstance of  $e_1$ .

**Proof** We need only prove the forward direction, since the backward direction is symmetric. Suppose, therefore, that  $e_1(Q)$  can be solved in time  $O(n^k)$  for some constant  $k$ . Further, suppose that for any problem instance  $i$ , the encoding  $e_1(i)$  can be computed from the encoding  $e_2(i)$  in time  $O(n^c)$  for some constant  $c$ , where  $n = |e_2(i)|$ . To solve problem  $e_2(Q)$ , on input  $e_2(i)$ , we first compute  $e_1(i)$  and then run the algorithm for  $e_1(Q)$  on  $e_1(i)$ . How long does this take? Converting encodings takes time  $O(n^c)$ , and therefore  $|e_1(i)| = O(n^c)$ , since the output of a serial computer cannot be longer than its running time. Solving the problem on  $e_1(i)$  takes time  $O(|e_1(i)|^k) = O(n^{ck})$ , which is polynomial since both  $c$  and  $k$  are constants. ■

Thus, whether an abstract problem has its instances encoded in binary or base 3 does not affect its “complexity,” that is, whether it is polynomial-time solvable or not; but if instances are encoded in unary, its complexity may change. In order to be able to converse in an encoding-independent fashion, we shall generally assume that problem instances are encoded in any reasonable, concise fashion, unless we specifically say otherwise. To be precise, we shall assume that the encoding of an integer is polynomially related to its binary representation, and that the encoding of a finite set is polynomially related to its encoding as a list of its elements, enclosed in braces and separated by commas. (ASCII is one such encoding scheme.) With such a “standard” encoding in hand, we can derive reasonable encodings of other mathematical objects, such as tuples, graphs, and formulas. To denote the standard encoding of an object, we shall enclose the object in angle braces. Thus,  $\langle G \rangle$  denotes the standard encoding of a graph  $G$ .

As long as we implicitly use an encoding that is polynomially related to this standard encoding, we can talk directly about abstract problems without reference to any particular encoding, knowing that the choice of encoding has no effect on whether the abstract problem is polynomial-time solvable. Henceforth, we shall generally assume that all problem instances are binary strings encoded using the standard encoding, unless we explicitly specify the contrary. We shall also typically neglect the distinction between abstract and concrete problems. You should watch out for problems that arise in practice, however, in which a standard encoding is not obvious and the encoding does make a difference.

### A formal-language framework

By focusing on decision problems, we can take advantage of the machinery of formal-language theory. Let’s review some definitions from that theory. An **alphabet**  $\Sigma$  is a finite set of symbols. A **language**  $L$  over  $\Sigma$  is any set of strings made up of symbols from  $\Sigma$ . For example, if  $\Sigma = \{0, 1\}$ , the set  $L = \{10, 11, 101, 111, 1011, 1101, 10001, \dots\}$  is the language of binary represen-

tations of prime numbers. We denote the *empty string* by  $\varepsilon$ , the *empty language* by  $\emptyset$ , and the language of all strings over  $\Sigma$  by  $\Sigma^*$ . For example, if  $\Sigma = \{0, 1\}$ , then  $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$  is the set of all binary strings. Every language  $L$  over  $\Sigma$  is a subset of  $\Sigma^*$ .

We can perform a variety of operations on languages. Set-theoretic operations, such as *union* and *intersection*, follow directly from the set-theoretic definitions. We define the *complement* of  $L$  by  $\bar{L} = \Sigma^* - L$ . The *concatenation*  $L_1 L_2$  of two languages  $L_1$  and  $L_2$  is the language

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\}.$$

The *closure* or *Kleene star* of a language  $L$  is the language

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \dots,$$

where  $L^k$  is the language obtained by concatenating  $L$  to itself  $k$  times.

From the point of view of language theory, the set of instances for any decision problem  $Q$  is simply the set  $\Sigma^*$ , where  $\Sigma = \{0, 1\}$ . Since  $Q$  is entirely characterized by those problem instances that produce a 1 (yes) answer, we can view  $Q$  as a language  $L$  over  $\Sigma = \{0, 1\}$ , where

$$L = \{x \in \Sigma^* : Q(x) = 1\}.$$

For example, the decision problem PATH has the corresponding language

$$\begin{aligned} \text{PATH} = \{ \langle G, u, v, k \rangle : & G = (V, E) \text{ is an undirected graph,} \\ & u, v \in V, \\ & k \geq 0 \text{ is an integer, and} \\ & \text{there exists a path from } u \text{ to } v \text{ in } G \\ & \text{consisting of at most } k \text{ edges} \} . \end{aligned}$$

(Where convenient, we shall sometimes use the same name—PATH in this case—to refer to both a decision problem and its corresponding language.)

The formal-language framework allows us to express concisely the relation between decision problems and algorithms that solve them. We say that an algorithm  $A$  *accepts* a string  $x \in \{0, 1\}^*$  if, given input  $x$ , the algorithm's output  $A(x)$  is 1. The language *accepted* by an algorithm  $A$  is the set of strings  $L = \{x \in \{0, 1\}^* : A(x) = 1\}$ , that is, the set of strings that the algorithm accepts. An algorithm  $A$  *rejects* a string  $x$  if  $A(x) = 0$ .

Even if language  $L$  is accepted by an algorithm  $A$ , the algorithm will not necessarily reject a string  $x \notin L$  provided as input to it. For example, the algorithm may loop forever. A language  $L$  is *decided* by an algorithm  $A$  if every binary string in  $L$  is accepted by  $A$  and every binary string not in  $L$  is rejected by  $A$ . A language  $L$  is *accepted in polynomial time* by an algorithm  $A$  if it is accepted by  $A$  and if in addition there exists a constant  $k$  such that for any length- $n$  string  $x \in L$ ,

algorithm  $A$  accepts  $x$  in time  $O(n^k)$ . A language  $L$  is **decided in polynomial time** by an algorithm  $A$  if there exists a constant  $k$  such that for any length- $n$  string  $x \in \{0, 1\}^*$ , the algorithm correctly decides whether  $x \in L$  in time  $O(n^k)$ . Thus, to accept a language, an algorithm need only produce an answer when provided a string in  $L$ , but to decide a language, it must correctly accept or reject every string in  $\{0, 1\}^*$ .

As an example, the language PATH can be accepted in polynomial time. One polynomial-time accepting algorithm verifies that  $G$  encodes an undirected graph, verifies that  $u$  and  $v$  are vertices in  $G$ , uses breadth-first search to compute a shortest path from  $u$  to  $v$  in  $G$ , and then compares the number of edges on the shortest path obtained with  $k$ . If  $G$  encodes an undirected graph and the path found from  $u$  to  $v$  has at most  $k$  edges, the algorithm outputs 1 and halts. Otherwise, the algorithm runs forever. This algorithm does not decide PATH, however, since it does not explicitly output 0 for instances in which a shortest path has more than  $k$  edges. A decision algorithm for PATH must explicitly reject binary strings that do not belong to PATH. For a decision problem such as PATH, such a decision algorithm is easy to design: instead of running forever when there is not a path from  $u$  to  $v$  with at most  $k$  edges, it outputs 0 and halts. (It must also output 0 and halt if the input encoding is faulty.) For other problems, such as Turing's Halting Problem, there exists an accepting algorithm, but no decision algorithm exists.

We can informally define a **complexity class** as a set of languages, membership in which is determined by a **complexity measure**, such as running time, of an algorithm that determines whether a given string  $x$  belongs to language  $L$ . The actual definition of a complexity class is somewhat more technical.<sup>6</sup>

Using this language-theoretic framework, we can provide an alternative definition of the complexity class P:

$$P = \{L \subseteq \{0, 1\}^* : \text{there exists an algorithm } A \text{ that decides } L \text{ in polynomial time}\}.$$

In fact, P is also the class of languages that can be accepted in polynomial time.

### Theorem 34.2

$$P = \{L : L \text{ is accepted by a polynomial-time algorithm}\}.$$

**Proof** Because the class of languages decided by polynomial-time algorithms is a subset of the class of languages accepted by polynomial-time algorithms, we need only show that if  $L$  is accepted by a polynomial-time algorithm, it is decided by a polynomial-time algorithm. Let  $L$  be the language accepted by some

---

<sup>6</sup>For more on complexity classes, see the seminal paper by Hartmanis and Stearns [162].

polynomial-time algorithm  $A$ . We shall use a classic “simulation” argument to construct another polynomial-time algorithm  $A'$  that decides  $L$ . Because  $A$  accepts  $L$  in time  $O(n^k)$  for some constant  $k$ , there also exists a constant  $c$  such that  $A$  accepts  $L$  in at most  $cn^k$  steps. For any input string  $x$ , the algorithm  $A'$  simulates  $cn^k$  steps of  $A$ . After simulating  $cn^k$  steps, algorithm  $A'$  inspects the behavior of  $A$ . If  $A$  has accepted  $x$ , then  $A'$  accepts  $x$  by outputting a 1. If  $A$  has not accepted  $x$ , then  $A'$  rejects  $x$  by outputting a 0. The overhead of  $A'$  simulating  $A$  does not increase the running time by more than a polynomial factor, and thus  $A'$  is a polynomial-time algorithm that decides  $L$ . ■

Note that the proof of Theorem 34.2 is nonconstructive. For a given language  $L \in P$ , we may not actually know a bound on the running time for the algorithm  $A$  that accepts  $L$ . Nevertheless, we know that such a bound exists, and therefore, that an algorithm  $A'$  exists that can check the bound, even though we may not be able to find the algorithm  $A'$  easily.

## Exercises

### 34.1-1

Define the optimization problem LONGEST-PATH-LENGTH as the relation that associates each instance of an undirected graph and two vertices with the number of edges in a longest simple path between the two vertices. Define the decision problem LONGEST-PATH =  $\{\langle G, u, v, k \rangle : G = (V, E) \text{ is an undirected graph, } u, v \in V, k \geq 0 \text{ is an integer, and there exists a simple path from } u \text{ to } v \text{ in } G \text{ consisting of at least } k \text{ edges}\}$ . Show that the optimization problem LONGEST-PATH-LENGTH can be solved in polynomial time if and only if LONGEST-PATH  $\in P$ .

### 34.1-2

Give a formal definition for the problem of finding the longest simple cycle in an undirected graph. Give a related decision problem. Give the language corresponding to the decision problem.

### 34.1-3

Give a formal encoding of directed graphs as binary strings using an adjacency-matrix representation. Do the same using an adjacency-list representation. Argue that the two representations are polynomially related.

### 34.1-4

Is the dynamic-programming algorithm for the 0-1 knapsack problem that is asked for in Exercise 16.2-2 a polynomial-time algorithm? Explain your answer.

**34.1-5**

Show that if an algorithm makes at most a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then it runs in polynomial time. Also show that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

**34.1-6**

Show that the class  $P$ , viewed as a set of languages, is closed under union, intersection, concatenation, complement, and Kleene star. That is, if  $L_1, L_2 \in P$ , then  $L_1 \cup L_2 \in P$ ,  $L_1 \cap L_2 \in P$ ,  $L_1 L_2 \in P$ ,  $\overline{L_1} \in P$ , and  $L_1^* \in P$ .

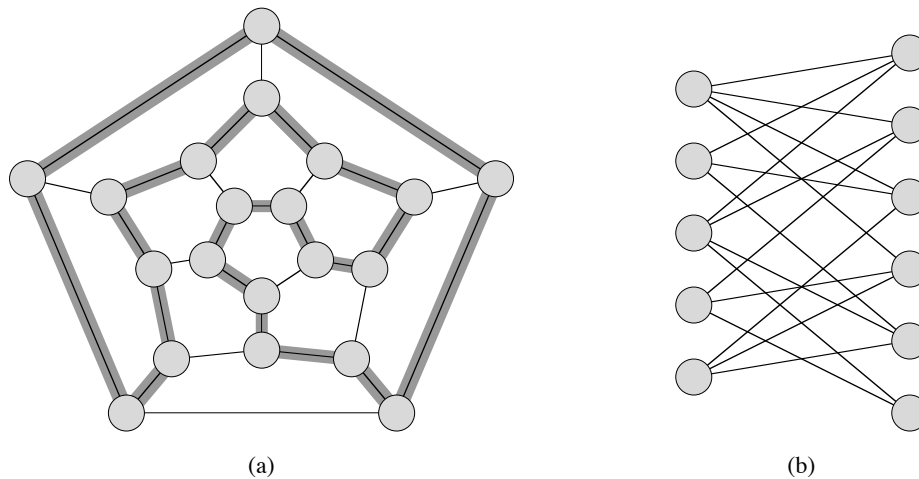
---

## 34.2 Polynomial-time verification

We now look at algorithms that verify membership in languages. For example, suppose that for a given instance  $\langle G, u, v, k \rangle$  of the decision problem PATH, we are also given a path  $p$  from  $u$  to  $v$ . We can easily check whether  $p$  is a path in  $G$  and whether the length of  $p$  is at most  $k$ , and if so, we can view  $p$  as a “certificate” that the instance indeed belongs to PATH. For the decision problem PATH, this certificate doesn’t seem to buy us much. After all, PATH belongs to  $P$ —in fact, we can solve PATH in linear time—and so verifying membership from a given certificate takes as long as solving the problem from scratch. We shall now examine a problem for which we know of no polynomial-time decision algorithm and yet, given a certificate, verification is easy.

### Hamiltonian cycles

The problem of finding a hamiltonian cycle in an undirected graph has been studied for over a hundred years. Formally, a **hamiltonian cycle** of an undirected graph  $G = (V, E)$  is a simple cycle that contains each vertex in  $V$ . A graph that contains a hamiltonian cycle is said to be **hamiltonian**; otherwise, it is **nonhamiltonian**. The name honors W. R. Hamilton, who described a mathematical game on the dodecahedron (Figure 34.2(a)) in which one player sticks five pins in any five consecutive vertices and the other player must complete the path to form a cycle



**Figure 34.2** (a) A graph representing the vertices, edges, and faces of a dodecahedron, with a hamiltonian cycle shown by shaded edges. (b) A bipartite graph with an odd number of vertices. Any such graph is nonhamiltonian.

containing all the vertices.<sup>7</sup> The dodecahedron is hamiltonian, and Figure 34.2(a) shows one hamiltonian cycle. Not all graphs are hamiltonian, however. For example, Figure 34.2(b) shows a bipartite graph with an odd number of vertices. Exercise 34.2-2 asks you to show that all such graphs are nonhamiltonian.

We can define the **hamiltonian-cycle problem**, “Does a graph  $G$  have a hamiltonian cycle?” as a formal language:

$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ is a hamiltonian graph}\}.$$

How might an algorithm decide the language HAM-CYCLE? Given a problem instance  $\langle G \rangle$ , one possible decision algorithm lists all permutations of the vertices of  $G$  and then checks each permutation to see if it is a hamiltonian cycle. What is the running time of this algorithm? If we use the “reasonable” encoding of a graph as its adjacency matrix, the number  $m$  of vertices in the graph is  $\Omega(\sqrt{n})$ , where  $n = |\langle G \rangle|$  is the length of the encoding of  $G$ . There are  $m!$  possible permutations

<sup>7</sup>In a letter dated 17 October 1856 to his friend John T. Graves, Hamilton [157, p. 624] wrote, “I have found that some young persons have been much amused by trying a new mathematical game which the Icosion furnishes, one person sticking five pins in any five consecutive points ... and the other player then aiming to insert, which by the theory in this letter can always be done, fifteen other pins, in cyclical succession, so as to cover all the other points, and to end in immediate proximity to the pin wherewith his antagonist had begun.”

of the vertices, and therefore the running time is  $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$ , which is not  $O(n^k)$  for any constant  $k$ . Thus, this naive algorithm does not run in polynomial time. In fact, the hamiltonian-cycle problem is NP-complete, as we shall prove in Section 34.5.

### Verification algorithms

Consider a slightly easier problem. Suppose that a friend tells you that a given graph  $G$  is hamiltonian, and then offers to prove it by giving you the vertices in order along the hamiltonian cycle. It would certainly be easy enough to verify the proof: simply verify that the provided cycle is hamiltonian by checking whether it is a permutation of the vertices of  $V$  and whether each of the consecutive edges along the cycle actually exists in the graph. You could certainly implement this verification algorithm to run in  $O(n^2)$  time, where  $n$  is the length of the encoding of  $G$ . Thus, a proof that a hamiltonian cycle exists in a graph can be verified in polynomial time.

We define a **verification algorithm** as being a two-argument algorithm  $A$ , where one argument is an ordinary input string  $x$  and the other is a binary string  $y$  called a **certificate**. A two-argument algorithm  $A$  **verifies** an input string  $x$  if there exists a certificate  $y$  such that  $A(x, y) = 1$ . The **language verified** by a verification algorithm  $A$  is

$$L = \{x \in \{0, 1\}^* : \text{there exists } y \in \{0, 1\}^* \text{ such that } A(x, y) = 1\}.$$

Intuitively, an algorithm  $A$  verifies a language  $L$  if for any string  $x \in L$ , there exists a certificate  $y$  that  $A$  can use to prove that  $x \in L$ . Moreover, for any string  $x \notin L$ , there must be no certificate proving that  $x \in L$ . For example, in the hamiltonian-cycle problem, the certificate is the list of vertices in some hamiltonian cycle. If a graph is hamiltonian, the hamiltonian cycle itself offers enough information to verify this fact. Conversely, if a graph is not hamiltonian, there can be no list of vertices that fools the verification algorithm into believing that the graph is hamiltonian, since the verification algorithm carefully checks the proposed “cycle” to be sure.



### The complexity class NP

The **complexity class NP** is the class of languages that can be verified by a polynomial-time algorithm.<sup>8</sup> More precisely, a language  $L$  belongs to NP if and only if there exist a two-input polynomial-time algorithm  $A$  and a constant  $c$  such that

$$L = \{x \in \{0, 1\}^* : \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \\ \text{such that } A(x, y) = 1\}.$$

We say that algorithm  $A$  **verifies** language  $L$  *in polynomial time*.

From our earlier discussion on the hamiltonian-cycle problem, we now see that HAM-CYCLE  $\in$  NP. (It is always nice to know that an important set is nonempty.) Moreover, if  $L \in P$ , then  $L \in$  NP, since if there is a polynomial-time algorithm to decide  $L$ , the algorithm can be easily converted to a two-argument verification algorithm that simply ignores any certificate and accepts exactly those input strings it determines to be in  $L$ . Thus,  $P \subseteq$  NP.

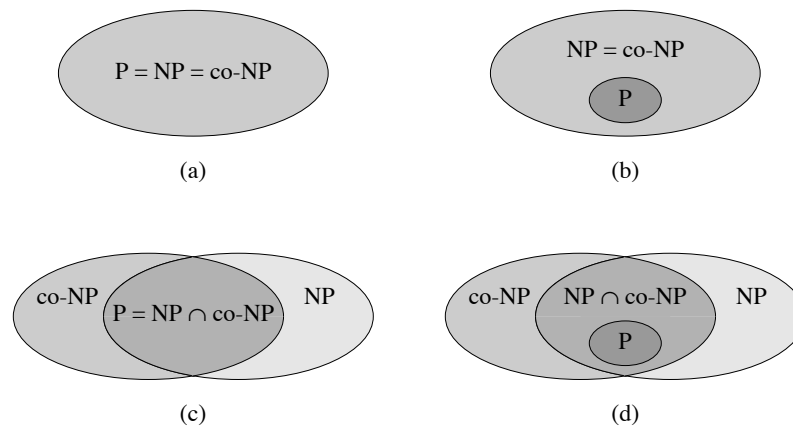
It is unknown whether  $P =$  NP, but most researchers believe that  $P$  and NP are not the same class. Intuitively, the class  $P$  consists of problems that can be solved quickly. The class NP consists of problems for which a solution can be verified quickly. You may have learned from experience that it is often more difficult to solve a problem from scratch than to verify a clearly presented solution, especially when working under time constraints. Theoretical computer scientists generally believe that this analogy extends to the classes  $P$  and NP, and thus that NP includes languages that are not in  $P$ .

There is more compelling, though not conclusive, evidence that  $P \neq$  NP—the existence of languages that are “NP-complete.” We shall study this class in Section 34.3.

Many other fundamental questions beyond the  $P \neq$  NP question remain unresolved. Figure 34.3 shows some possible scenarios. Despite much work by many researchers, no one even knows whether the class NP is closed under complement. That is, does  $L \in$  NP imply  $\overline{L} \in$  NP? We can define the **complexity class co-NP** as the set of languages  $L$  such that  $\overline{L} \in$  NP. We can restate the question of whether NP is closed under complement as whether  $\text{NP} = \text{co-NP}$ . Since  $P$  is closed under complement (Exercise 34.1-6), it follows from Exercise 34.2-9 that  $P \subseteq \text{NP} \cap \text{co-NP}$ . Once again, however, no one knows whether  $P = \text{NP} \cap \text{co-NP}$  or whether there is some language in  $\text{NP} \cap \text{co-NP} - P$ .

---

<sup>8</sup>The name “NP” stands for “nondeterministic polynomial time.” The class NP was originally studied in the context of nondeterminism, but this book uses the somewhat simpler yet equivalent notion of verification. Hopcroft and Ullman [180] give a good presentation of NP-completeness in terms of nondeterministic models of computation.



**Figure 34.3** Four possibilities for relationships among complexity classes. In each diagram, one region enclosing another indicates a proper-subset relation. (a)  $P = NP = co-NP$ . Most researchers regard this possibility as the most unlikely. (b) If  $NP$  is closed under complement, then  $NP = co-NP$ , but it need not be the case that  $P = NP$ . (c)  $P = NP \cap co-NP$ , but  $NP$  is not closed under complement. (d)  $NP \neq co-NP$  and  $P \neq NP \cap co-NP$ . Most researchers regard this possibility as the most likely.

Thus, our understanding of the precise relationship between  $P$  and  $NP$  is woefully incomplete. Nevertheless, even though we might not be able to prove that a particular problem is intractable, if we can prove that it is  $NP$ -complete, then we have gained valuable information about it.

## Exercises

### 34.2-1

Consider the language  $GRAPH-ISOMORPHISM = \{\langle G_1, G_2 \rangle : G_1 \text{ and } G_2 \text{ are isomorphic graphs}\}$ . Prove that  $GRAPH-ISOMORPHISM \in NP$  by describing a polynomial-time algorithm to verify the language.

### 34.2-2

Prove that if  $G$  is an undirected bipartite graph with an odd number of vertices, then  $G$  is nonhamiltonian.

### 34.2-3

Show that if  $HAM-CYCLE \in P$ , then the problem of listing the vertices of a hamiltonian cycle, in order, is polynomial-time solvable.

**34.2-4**

Prove that the class NP of languages is closed under union, intersection, concatenation, and Kleene star. Discuss the closure of NP under complement.

**34.2-5**

Show that any language in NP can be decided by an algorithm running in time  $2^{O(n^k)}$  for some constant  $k$ .

**34.2-6**

A **hamiltonian path** in a graph is a simple path that visits every vertex exactly once. Show that the language HAM-PATH =  $\{\langle G, u, v \rangle : \text{there is a hamiltonian path from } u \text{ to } v \text{ in graph } G\}$  belongs to NP.

**34.2-7**

Show that the hamiltonian-path problem from Exercise 34.2-6 can be solved in polynomial time on directed acyclic graphs. Give an efficient algorithm for the problem.

**34.2-8**

Let  $\phi$  be a boolean formula constructed from the boolean input variables  $x_1, x_2, \dots, x_k$ , negations ( $\neg$ ), ANDs ( $\wedge$ ), ORs ( $\vee$ ), and parentheses. The formula  $\phi$  is a **tautology** if it evaluates to 1 for every assignment of 1 and 0 to the input variables. Define TAUTOLOGY as the language of boolean formulas that are tautologies. Show that TAUTOLOGY  $\in$  co-NP.

**34.2-9**

Prove that  $P \subseteq \text{co-NP}$ .

**34.2-10**

Prove that if NP  $\neq$  co-NP, then  $P \neq \text{NP}$ .

**34.2-11**

Let  $G$  be a connected, undirected graph with at least 3 vertices, and let  $G^3$  be the graph obtained by connecting all pairs of vertices that are connected by a path in  $G$  of length at most 3. Prove that  $G^3$  is hamiltonian. (*Hint:* Construct a spanning tree for  $G$ , and use an inductive argument.)

### 34.3 NP-completeness and reducibility

Perhaps the most compelling reason why theoretical computer scientists believe that  $P \neq NP$  comes from the existence of the class of “NP-complete” problems. This class has the intriguing property that if *any* NP-complete problem can be solved in polynomial time, then *every* problem in NP has a polynomial-time solution, that is,  $P = NP$ . Despite years of study, though, no polynomial-time algorithm has ever been discovered for any NP-complete problem.

The language HAM-CYCLE is one NP-complete problem. If we could decide HAM-CYCLE in polynomial time, then we could solve every problem in NP in polynomial time. In fact, if  $NP - P$  should turn out to be nonempty, we could say with certainty that  $HAM-CYCLE \in NP - P$ .

The NP-complete languages are, in a sense, the “hardest” languages in NP. In this section, we shall show how to compare the relative “hardness” of languages using a precise notion called “polynomial-time reducibility.” Then we formally define the NP-complete languages, and we finish by sketching a proof that one such language, called CIRCUIT-SAT, is NP-complete. In Sections 34.4 and 34.5, we shall use the notion of reducibility to show that many other problems are NP-complete.

#### Reducibility

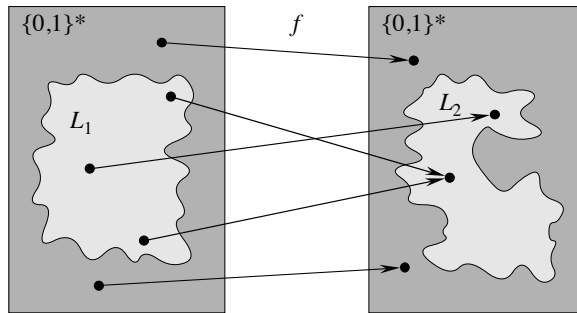
Intuitively, a problem  $Q$  can be reduced to another problem  $Q'$  if any instance of  $Q$  can be “easily rephrased” as an instance of  $Q'$ , the solution to which provides a solution to the instance of  $Q$ . For example, the problem of solving linear equations in an indeterminate  $x$  reduces to the problem of solving quadratic equations. Given a linear-equation instance  $ax + b = 0$  (with solution  $x = -b/a$ ), we transform it to the quadratic equation  $ax^2 + bx + 0 = 0$ , whose solutions are  $x = 0$  and  $x = -b/a$ , thereby providing a solution to  $ax + b = 0$ . Thus, if a problem  $Q$  reduces to another problem  $Q'$ , then  $Q$  is, in a sense, “no harder to solve” than  $Q'$ .

Returning to our formal-language framework for decision problems, we say that a language  $L_1$  is **polynomial-time reducible** to a language  $L_2$ , written  $L_1 \leq_P L_2$ , if there exists a polynomial-time computable function  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  such that for all  $x \in \{0, 1\}^*$ ,

$$x \in L_1 \text{ if and only if } f(x) \in L_2. \quad (34.1)$$

We call the function  $f$  the **reduction function**, and a polynomial-time algorithm  $F$  that computes  $f$  is a **reduction algorithm**.

Figure 34.4 illustrates the idea of a polynomial-time reduction from a language  $L_1$  to another language  $L_2$ . Each language is a subset of  $\{0, 1\}^*$ . The reduction function  $f$  provides a polynomial-time mapping such that if  $x \in L_1$ ,



**Figure 34.4** An illustration of a polynomial-time reduction from a language  $L_1$  to a language  $L_2$  via a reduction function  $f$ . For any input  $x \in \{0,1\}^*$ , the question of whether  $x \in L_1$  has the same answer as the question of whether  $f(x) \in L_2$ .

then  $f(x) \in L_2$ . Moreover, if  $x \notin L_1$ , then  $f(x) \notin L_2$ . Thus, the reduction function maps any instance  $x$  of the decision problem represented by the language  $L_1$  to an instance  $f(x)$  of the problem represented by  $L_2$ . Providing an answer to whether  $f(x) \in L_2$  directly provides the answer to whether  $x \in L_1$ .

Polynomial-time reductions give us a powerful tool for proving that various languages belong to P.

### Lemma 34.3

If  $L_1, L_2 \subseteq \{0,1\}^*$  are languages such that  $L_1 \leq_P L_2$ , then  $L_2 \in \text{P}$  implies  $L_1 \in \text{P}$ .

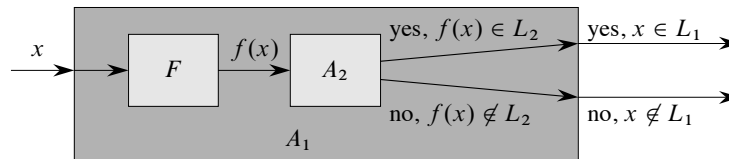
**Proof** Let  $A_2$  be a polynomial-time algorithm that decides  $L_2$ , and let  $F$  be a polynomial-time reduction algorithm that computes the reduction function  $f$ . We shall construct a polynomial-time algorithm  $A_1$  that decides  $L_1$ .

Figure 34.5 illustrates how we construct  $A_1$ . For a given input  $x \in \{0,1\}^*$ , algorithm  $A_1$  uses  $F$  to transform  $x$  into  $f(x)$ , and then it uses  $A_2$  to test whether  $f(x) \in L_2$ . Algorithm  $A_1$  takes the output from algorithm  $A_2$  and produces that answer as its own output.

The correctness of  $A_1$  follows from condition (34.1). The algorithm runs in polynomial time, since both  $F$  and  $A_2$  run in polynomial time (see Exercise 34.1-5). ■

## NP-completeness

Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within a polynomial-time factor. That is, if  $L_1 \leq_P L_2$ , then  $L_1$  is not more than a polynomial factor harder than  $L_2$ , which is



**Figure 34.5** The proof of Lemma 34.3. The algorithm  $F$  is a reduction algorithm that computes the reduction function  $f$  from  $L_1$  to  $L_2$  in polynomial time, and  $A_2$  is a polynomial-time algorithm that decides  $L_2$ . Algorithm  $A_1$  decides whether  $x \in L_1$  by using  $F$  to transform any input  $x$  into  $f(x)$  and then using  $A_2$  to decide whether  $f(x) \in L_2$ .

why the “less than or equal to” notation for reduction is mnemonic. We can now define the set of NP-complete languages, which are the hardest problems in NP.

A language  $L \subseteq \{0, 1\}^*$  is **NP-complete** if

1.  $L \in \text{NP}$ , and
2.  $L' \leq_P L$  for every  $L' \in \text{NP}$ .

If a language  $L$  satisfies property 2, but not necessarily property 1, we say that  $L$  is **NP-hard**. We also define NPC to be the class of NP-complete languages.

As the following theorem shows, NP-completeness is at the crux of deciding whether P is in fact equal to NP.

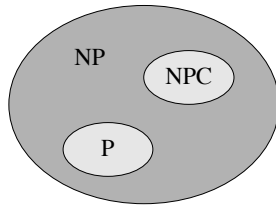
#### Theorem 34.4

If any NP-complete problem is polynomial-time solvable, then  $P = \text{NP}$ . Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.

**Proof** Suppose that  $L \in P$  and also that  $L \in \text{NPC}$ . For any  $L' \in \text{NP}$ , we have  $L' \leq_P L$  by property 2 of the definition of NP-completeness. Thus, by Lemma 34.3, we also have that  $L' \in P$ , which proves the first statement of the theorem.

To prove the second statement, note that it is the contrapositive of the first statement. ■

It is for this reason that research into the  $P \neq \text{NP}$  question centers around the NP-complete problems. Most theoretical computer scientists believe that  $P \neq \text{NP}$ , which leads to the relationships among P, NP, and NPC shown in Figure 34.6. But, for all we know, someone may yet come up with a polynomial-time algorithm for an NP-complete problem, thus proving that  $P = \text{NP}$ . Nevertheless, since no polynomial-time algorithm for any NP-complete problem has yet been discov-



**Figure 34.6** How most theoretical computer scientists view the relationships among  $P$ ,  $NP$ , and  $NPC$ . Both  $P$  and  $NPC$  are wholly contained within  $NP$ , and  $P \cap NPC = \emptyset$ .

ered, a proof that a problem is NP-complete provides excellent evidence that it is intractable.

### Circuit satisfiability

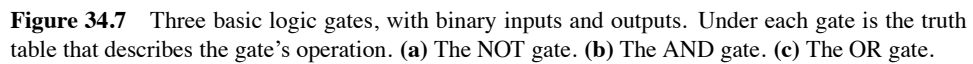
We have defined the notion of an NP-complete problem, but up to this point, we have not actually proved that any problem is NP-complete. Once we prove that at least one problem is NP-complete, we can use polynomial-time reducibility as a tool to prove other problems to be NP-complete. Thus, we now focus on demonstrating the existence of an NP-complete problem: the circuit-satisfiability problem.

Unfortunately, the formal proof that the circuit-satisfiability problem is NP-complete requires technical detail beyond the scope of this text. Instead, we shall informally describe a proof that relies on a basic understanding of boolean combinational circuits.

Boolean combinational circuits are built from boolean combinational elements that are interconnected by wires. A **boolean combinational element** is any circuit element that has a constant number of boolean inputs and outputs and that performs a well-defined function. Boolean values are drawn from the set  $\{0, 1\}$ , where 0 represents FALSE and 1 represents TRUE.

The boolean combinational elements that we use in the circuit-satisfiability problem compute simple boolean functions, and they are known as **logic gates**. Figure 34.7 shows the three basic logic gates that we use in the circuit-satisfiability problem: the **NOT gate** (or **inverter**), the **AND gate**, and the **OR gate**. The NOT gate takes a single binary **input**  $x$ , whose value is either 0 or 1, and produces a binary **output**  $z$  whose value is opposite that of the input value. Each of the other two gates takes two binary inputs  $x$  and  $y$  and produces a single binary output  $z$ .

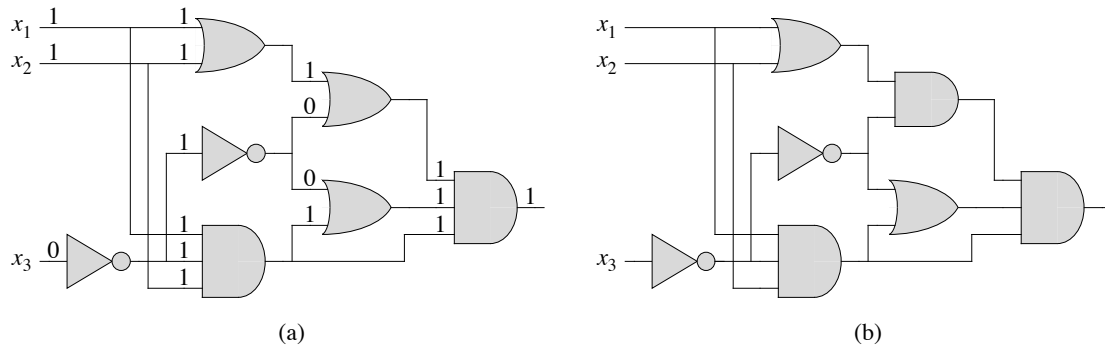
We can describe the operation of each gate, and of any boolean combinational element, by a **truth table**, shown under each gate in Figure 34.7. A truth table gives the outputs of the combinational element for each possible setting of the inputs. For



We can generalize AND and OR gates to take more than two inputs. An AND gate's output is 1 if all of its inputs are 1, and its output is 0 otherwise. An OR gate's output is 1 if any of its inputs are 1, and its output is 0 otherwise.

Boolean combinational circuits contain no cycles. In other words, suppose we create a directed graph  $G = (V, E)$  with one vertex for each combinational element and with  $k$  directed edges for each wire whose fan-out is  $k$ ; the graph contains a directed edge  $(u, v)$  if a wire connects the output of element  $u$  to an input of element  $v$ . Then  $G$  must be acyclic.





**Figure 34.8** Two instances of the circuit-satisfiability problem. **(a)** The assignment  $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$  to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. **(b)** No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.

A **truth assignment** for a boolean combinational circuit is a set of boolean input values. We say that a one-output boolean combinational circuit is **satisfiable** if it has a **satisfying assignment**: a truth assignment that causes the output of the circuit to be 1. For example, the circuit in Figure 34.8(a) has the satisfying assignment  $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$ , and so it is satisfiable. As Exercise 34.3-1 asks you to show, no assignment of values to  $x_1, x_2$ , and  $x_3$  causes the circuit in Figure 34.8(b) to produce a 1 output; it always produces 0, and so it is unsatisfiable.

The **circuit-satisfiability problem** is, “Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?” In order to pose this question formally, however, we must agree on a standard encoding for circuits. The **size** of a boolean combinational circuit is the number of boolean combinational elements plus the number of wires in the circuit. We could devise a graphlike encoding that maps any given circuit  $C$  into a binary string  $\langle C \rangle$  whose length is polynomial in the size of the circuit itself. As a formal language, we can therefore define

$$\text{CIRCUIT-SAT} = \{ \langle C \rangle : C \text{ is a satisfiable boolean combinational circuit} \}.$$

The circuit-satisfiability problem arises in the area of computer-aided hardware optimization. If a subcircuit always produces 0, that subcircuit is unnecessary; the designer can replace it by a simpler subcircuit that omits all logic gates and provides the constant 0 value as its output. You can see why we would like to have a polynomial-time algorithm for this problem.

Given a circuit  $C$ , we might attempt to determine whether it is satisfiable by simply checking all possible assignments to the inputs. Unfortunately, if the circuit has  $k$  inputs, then we would have to check up to  $2^k$  possible assignments. When

the size of  $C$  is polynomial in  $k$ , checking each one takes  $\Omega(2^k)$  time, which is superpolynomial in the size of the circuit.<sup>9</sup> In fact, as we have claimed, there is strong evidence that no polynomial-time algorithm exists that solves the circuit-satisfiability problem because circuit satisfiability is NP-complete. We break the proof of this fact into two parts, based on the two parts of the definition of NP-completeness.

### **Lemma 34.5**

The circuit-satisfiability problem belongs to the class NP.

**Proof** We shall provide a two-input, polynomial-time algorithm  $A$  that can verify CIRCUIT-SAT. One of the inputs to  $A$  is (a standard encoding of) a boolean combinational circuit  $C$ . The other input is a certificate corresponding to an assignment of boolean values to the wires in  $C$ . (See Exercise 34.3-4 for a smaller certificate.)

We construct the algorithm  $A$  as follows. For each logic gate in the circuit, it checks that the value provided by the certificate on the output wire is correctly computed as a function of the values on the input wires. Then, if the output of the entire circuit is 1, the algorithm outputs 1, since the values assigned to the inputs of  $C$  provide a satisfying assignment. Otherwise,  $A$  outputs 0.

Whenever a satisfiable circuit  $C$  is input to algorithm  $A$ , there exists a certificate whose length is polynomial in the size of  $C$  and that causes  $A$  to output a 1. Whenever an unsatisfiable circuit is input, no certificate can fool  $A$  into believing that the circuit is satisfiable. Algorithm  $A$  runs in polynomial time: with a good implementation, linear time suffices. Thus, we can verify CIRCUIT-SAT in polynomial time, and CIRCUIT-SAT  $\in$  NP. ■

The second part of proving that CIRCUIT-SAT is NP-complete is to show that the language is NP-hard. That is, we must show that every language in NP is polynomial-time reducible to CIRCUIT-SAT. The actual proof of this fact is full of technical intricacies, and so we shall settle for a sketch of the proof based on some understanding of the workings of computer hardware.

A computer program is stored in the computer memory as a sequence of instructions. A typical instruction encodes an operation to be performed, addresses of operands in memory, and an address where the result is to be stored. A special memory location, called the **program counter**, keeps track of which instruc-

---

<sup>9</sup>On the other hand, if the size of the circuit  $C$  is  $\Theta(2^k)$ , then an algorithm whose running time is  $O(2^k)$  has a running time that is polynomial in the circuit size. Even if  $P \neq NP$ , this situation would not contradict the NP-completeness of the problem; the existence of a polynomial-time algorithm for a special case does not imply that there is a polynomial-time algorithm for all cases.

tion is to be executed next. The program counter automatically increments upon fetching each instruction, thereby causing the computer to execute instructions sequentially. The execution of an instruction can cause a value to be written to the program counter, however, which alters the normal sequential execution and allows the computer to loop and perform conditional branches.

At any point during the execution of a program, the computer's memory holds the entire state of the computation. (We take the memory to include the program itself, the program counter, working storage, and any of the various bits of state that a computer maintains for bookkeeping.) We call any particular state of computer memory a **configuration**. We can view the execution of an instruction as mapping one configuration to another. The computer hardware that accomplishes this mapping can be implemented as a boolean combinational circuit, which we denote by  $M$  in the proof of the following lemma.

### **Lemma 34.6**

The circuit-satisfiability problem is NP-hard.

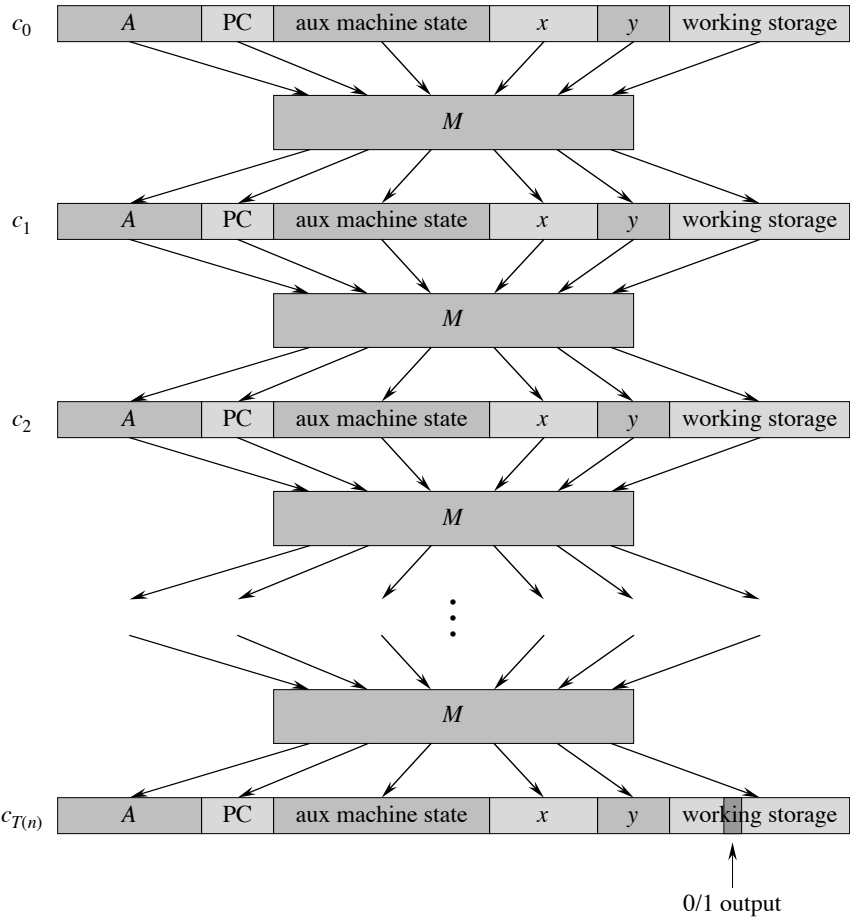
**Proof** Let  $L$  be any language in NP. We shall describe a polynomial-time algorithm  $F$  computing a reduction function  $f$  that maps every binary string  $x$  to a circuit  $C = f(x)$  such that  $x \in L$  if and only if  $C \in \text{CIRCUIT-SAT}$ .

Since  $L \in \text{NP}$ , there must exist an algorithm  $A$  that verifies  $L$  in polynomial time. The algorithm  $F$  that we shall construct uses the two-input algorithm  $A$  to compute the reduction function  $f$ .

Let  $T(n)$  denote the worst-case running time of algorithm  $A$  on length- $n$  input strings, and let  $k \geq 1$  be a constant such that  $T(n) = O(n^k)$  and the length of the certificate is  $O(n^k)$ . (The running time of  $A$  is actually a polynomial in the total input size, which includes both an input string and a certificate, but since the length of the certificate is polynomial in the length  $n$  of the input string, the running time is polynomial in  $n$ .)

The basic idea of the proof is to represent the computation of  $A$  as a sequence of configurations. As Figure 34.9 illustrates, we can break each configuration into parts consisting of the program for  $A$ , the program counter and auxiliary machine state, the input  $x$ , the certificate  $y$ , and working storage. The combinational circuit  $M$ , which implements the computer hardware, maps each configuration  $c_i$  to the next configuration  $c_{i+1}$ , starting from the initial configuration  $c_0$ . Algorithm  $A$  writes its output—0 or 1—to some designated location by the time it finishes executing, and if we assume that thereafter  $A$  halts, the value never changes. Thus, if the algorithm runs for at most  $T(n)$  steps, the output appears as one of the bits in  $c_{T(n)}$ .

The reduction algorithm  $F$  constructs a single combinational circuit that computes all configurations produced by a given initial configuration. The idea is to



**Figure 34.9** The sequence of configurations produced by an algorithm  $A$  running on an input  $x$  and certificate  $y$ . Each configuration represents the state of the computer for one step of the computation and, besides  $A$ ,  $x$ , and  $y$ , includes the program counter ( $PC$ ), auxiliary machine state, and working storage. Except for the certificate  $y$ , the initial configuration  $c_0$  is constant. A boolean combinational circuit  $M$  maps each configuration to the next configuration. The output is a distinguished bit in the working storage.

paste together  $T(n)$  copies of the circuit  $M$ . The output of the  $i$ th circuit, which produces configuration  $c_i$ , feeds directly into the input of the  $(i + 1)$ st circuit. Thus, the configurations, rather than being stored in the computer's memory, simply reside as values on the wires connecting copies of  $M$ .

Recall what the polynomial-time reduction algorithm  $F$  must do. Given an input  $x$ , it must compute a circuit  $C = f(x)$  that is satisfiable if and only if there exists a certificate  $y$  such that  $A(x, y) = 1$ . When  $F$  obtains an input  $x$ , it first computes  $n = |x|$  and constructs a combinational circuit  $C'$  consisting of  $T(n)$  copies of  $M$ . The input to  $C'$  is an initial configuration corresponding to a computation on  $A(x, y)$ , and the output is the configuration  $c_{T(n)}$ .

Algorithm  $F$  modifies circuit  $C'$  slightly to construct the circuit  $C = f(x)$ . First, it wires the inputs to  $C'$  corresponding to the program for  $A$ , the initial program counter, the input  $x$ , and the initial state of memory directly to these known values. Thus, the only remaining inputs to the circuit correspond to the certificate  $y$ . Second, it ignores all outputs from  $C'$ , except for the one bit of  $c_{T(n)}$  corresponding to the output of  $A$ . This circuit  $C$ , so constructed, computes  $C(y) = A(x, y)$  for any input  $y$  of length  $O(n^k)$ . The reduction algorithm  $F$ , when provided an input string  $x$ , computes such a circuit  $C$  and outputs it.

We need to prove two properties. First, we must show that  $F$  correctly computes a reduction function  $f$ . That is, we must show that  $C$  is satisfiable if and only if there exists a certificate  $y$  such that  $A(x, y) = 1$ . Second, we must show that  $F$  runs in polynomial time.

To show that  $F$  correctly computes a reduction function, let us suppose that there exists a certificate  $y$  of length  $O(n^k)$  such that  $A(x, y) = 1$ . Then, if we apply the bits of  $y$  to the inputs of  $C$ , the output of  $C$  is  $C(y) = A(x, y) = 1$ . Thus, if a certificate exists, then  $C$  is satisfiable. For the other direction, suppose that  $C$  is satisfiable. Hence, there exists an input  $y$  to  $C$  such that  $C(y) = 1$ , from which we conclude that  $A(x, y) = 1$ . Thus,  $F$  correctly computes a reduction function.

To complete the proof sketch, we need only show that  $F$  runs in time polynomial in  $n = |x|$ . The first observation we make is that the number of bits required to represent a configuration is polynomial in  $n$ . The program for  $A$  itself has constant size, independent of the length of its input  $x$ . The length of the input  $x$  is  $n$ , and the length of the certificate  $y$  is  $O(n^k)$ . Since the algorithm runs for at most  $O(n^k)$  steps, the amount of working storage required by  $A$  is polynomial in  $n$  as well. (We assume that this memory is contiguous; Exercise 34.3-5 asks you to extend the argument to the situation in which the locations accessed by  $A$  are scattered across a much larger region of memory and the particular pattern of scattering can differ for each input  $x$ .)

The combinational circuit  $M$  implementing the computer hardware has size polynomial in the length of a configuration, which is  $O(n^k)$ ; hence, the size of  $M$  is polynomial in  $n$ . (Most of this circuitry implements the logic of the memory

system.) The circuit  $C$  consists of at most  $t = O(n^k)$  copies of  $M$ , and hence it has size polynomial in  $n$ . The reduction algorithm  $F$  can construct  $C$  from  $x$  in polynomial time, since each step of the construction takes polynomial time. ■

The language CIRCUIT-SAT is therefore at least as hard as any language in NP, and since it belongs to NP, it is NP-complete.

### **Theorem 34.7**

The circuit-satisfiability problem is NP-complete.

**Proof** Immediate from Lemmas 34.5 and 34.6 and from the definition of NP-completeness. ■

## **Exercises**

### **34.3-1**

Verify that the circuit in Figure 34.8(b) is unsatisfiable.

### **34.3-2**

Show that the  $\leq_P$  relation is a transitive relation on languages. That is, show that if  $L_1 \leq_P L_2$  and  $L_2 \leq_P L_3$ , then  $L_1 \leq_P L_3$ .

### **34.3-3**

Prove that  $L \leq_P \bar{L}$  if and only if  $\bar{L} \leq_P L$ .

### **34.3-4**

Show that we could have used a satisfying assignment as a certificate in an alternative proof of Lemma 34.5. Which certificate makes for an easier proof?

### **34.3-5**

The proof of Lemma 34.6 assumes that the working storage for algorithm  $A$  occupies a contiguous region of polynomial size. Where in the proof do we exploit this assumption? Argue that this assumption does not involve any loss of generality.

### **34.3-6**

A language  $L$  is **complete** for a language class  $C$  with respect to polynomial-time reductions if  $L \in C$  and  $L' \leq_P L$  for all  $L' \in C$ . Show that  $\emptyset$  and  $\{0, 1\}^*$  are the only languages in P that are not complete for P with respect to polynomial-time reductions.

**34.3-7**

Show that, with respect to polynomial-time reductions (see Exercise 34.3-6),  $L$  is complete for NP if and only if  $\overline{L}$  is complete for co-NP.

**34.3-8**

The reduction algorithm  $F$  in the proof of Lemma 34.6 constructs the circuit  $C = f(x)$  based on knowledge of  $x$ ,  $A$ , and  $k$ . Professor Sartre observes that the string  $x$  is input to  $F$ , but only the existence of  $A$ ,  $k$ , and the constant factor implicit in the  $O(n^k)$  running time is known to  $F$  (since the language  $L$  belongs to NP), not their actual values. Thus, the professor concludes that  $F$  can't possibly construct the circuit  $C$  and that the language CIRCUIT-SAT is not necessarily NP-hard. Explain the flaw in the professor's reasoning.

---

**34.4 NP-completeness proofs**

We proved that the circuit-satisfiability problem is NP-complete by a direct proof that  $L \leq_P \text{CIRCUIT-SAT}$  for every language  $L \in \text{NP}$ . In this section, we shall show how to prove that languages are NP-complete without directly reducing *every* language in NP to the given language. We shall illustrate this methodology by proving that various formula-satisfiability problems are NP-complete. Section 34.5 provides many more examples of the methodology.

The following lemma is the basis of our method for showing that a language is NP-complete.

**Lemma 34.8**

If  $L$  is a language such that  $L' \leq_P L$  for some  $L' \in \text{NPC}$ , then  $L$  is NP-hard. If, in addition,  $L \in \text{NP}$ , then  $L \in \text{NPC}$ .

**Proof** Since  $L'$  is NP-complete, for all  $L'' \in \text{NP}$ , we have  $L'' \leq_P L'$ . By supposition,  $L' \leq_P L$ , and thus by transitivity (Exercise 34.3-2), we have  $L'' \leq_P L$ , which shows that  $L$  is NP-hard. If  $L \in \text{NP}$ , we also have  $L \in \text{NPC}$ . ■

In other words, by reducing a known NP-complete language  $L'$  to  $L$ , we implicitly reduce every language in NP to  $L$ . Thus, Lemma 34.8 gives us a method for proving that a language  $L$  is NP-complete:

1. Prove  $L \in \text{NP}$ .
2. Select a known NP-complete language  $L'$ .

3. Describe an algorithm that computes a function  $f$  mapping every instance  $x \in \{0, 1\}^*$  of  $L'$  to an instance  $f(x)$  of  $L$ .
4. Prove that the function  $f$  satisfies  $x \in L'$  if and only if  $f(x) \in L$  for all  $x \in \{0, 1\}^*$ .
5. Prove that the algorithm computing  $f$  runs in polynomial time.

(Steps 2–5 show that  $L$  is NP-hard.) This methodology of reducing from a single known NP-complete language is far simpler than the more complicated process of showing directly how to reduce from every language in NP. Proving  $\text{CIRCUIT-SAT} \in \text{NPC}$  has given us a “foot in the door.” Because we know that the circuit-satisfiability problem is NP-complete, we now can prove much more easily that other problems are NP-complete. Moreover, as we develop a catalog of known NP-complete problems, we will have more and more choices for languages from which to reduce.

### Formula satisfiability

We illustrate the reduction methodology by giving an NP-completeness proof for the problem of determining whether a boolean formula, not a circuit, is satisfiable. This problem has the historical honor of being the first problem ever shown to be NP-complete.

We formulate the (*formula*) *satisfiability* problem in terms of the language SAT as follows. An instance of SAT is a boolean formula  $\phi$  composed of

1.  $n$  boolean variables:  $x_1, x_2, \dots, x_n$ ;
2.  $m$  boolean connectives: any boolean function with one or two inputs and one output, such as  $\wedge$  (AND),  $\vee$  (OR),  $\neg$  (NOT),  $\rightarrow$  (implication),  $\leftrightarrow$  (if and only if); and
3. parentheses. (Without loss of generality, we assume that there are no redundant parentheses, i.e., a formula contains at most one pair of parentheses per boolean connective.)

We can easily encode a boolean formula  $\phi$  in a length that is polynomial in  $n + m$ . As in boolean combinational circuits, a *truth assignment* for a boolean formula  $\phi$  is a set of values for the variables of  $\phi$ , and a *satisfying assignment* is a truth assignment that causes it to evaluate to 1. A formula with a satisfying assignment is a *satisfiable* formula. The satisfiability problem asks whether a given boolean formula is satisfiable; in formal-language terms,

$$\text{SAT} = \{ \langle \phi \rangle : \phi \text{ is a satisfiable boolean formula} \} .$$

As an example, the formula



$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

has the satisfying assignment  $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1 \rangle$ , since

$$\begin{aligned} \phi &= ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1, \end{aligned} \tag{34.2}$$

and thus this formula  $\phi$  belongs to SAT.

The naive algorithm to determine whether an arbitrary boolean formula is satisfiable does not run in polynomial time. A formula with  $n$  variables has  $2^n$  possible assignments. If the length of  $\langle \phi \rangle$  is polynomial in  $n$ , then checking every assignment requires  $\Omega(2^n)$  time, which is superpolynomial in the length of  $\langle \phi \rangle$ . As the following theorem shows, a polynomial-time algorithm is unlikely to exist.

### **Theorem 34.9**

Satisfiability of boolean formulas is NP-complete.

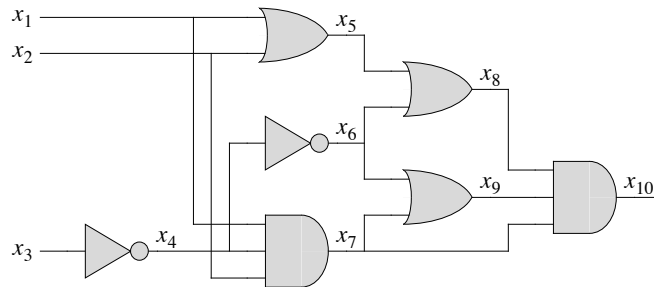
**Proof** We start by arguing that  $\text{SAT} \in \text{NP}$ . Then we prove that SAT is NP-hard by showing that  $\text{CIRCUIT-SAT} \leq_P \text{SAT}$ ; by Lemma 34.8, this will prove the theorem.

To show that SAT belongs to NP, we show that a certificate consisting of a satisfying assignment for an input formula  $\phi$  can be verified in polynomial time. The verifying algorithm simply replaces each variable in the formula with its corresponding value and then evaluates the expression, much as we did in equation (34.2) above. This task is easy to do in polynomial time. If the expression evaluates to 1, then the algorithm has verified that the formula is satisfiable. Thus, SAT is in NP.

To prove that SAT is NP-hard, we show that  $\text{CIRCUIT-SAT} \leq_P \text{SAT}$ . In other words, we need to show how to reduce any instance of circuit satisfiability to an instance of formula satisfiability in polynomial time. We can use induction to express any boolean combinational circuit as a boolean formula. We simply look at the gate that produces the circuit output and inductively express each of the gate's inputs as formulas. We then obtain the formula for the circuit by writing an expression that applies the gate's function to its inputs' formulas.

Unfortunately, this straightforward method does not amount to a polynomial-time reduction. As Exercise 34.4-1 asks you to show, shared subformulas—which arise from gates whose output wires have fan-out of 2 or more—can cause the size of the generated formula to grow exponentially. Thus, the reduction algorithm must be somewhat more clever.

Figure 34.10 illustrates how we overcome this problem, using as an example the circuit from Figure 34.8(a). For each wire  $x_i$  in the circuit  $C$ , the formula  $\phi$



has a variable  $x_i$ . We can now express how each gate operates as a small formula involving the variables of its incident wires. For example, the operation of the output AND gate is  $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$ . We call each of these small formulas a *clause*.

The formula  $\phi$  produced by the reduction algorithm is the AND of the circuit-output variable with the conjunction of clauses describing the operation of each gate. For the circuit in the figure, the formula is

$$\begin{aligned} \phi = & x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \\ & \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\ & \wedge (x_6 \leftrightarrow \neg x_4) \\ & \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\ & \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\ & \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\ & \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)). \end{aligned}$$

Given a circuit  $C$ , it is straightforward to produce such a formula  $\phi$  in polynomial time.

Why is the circuit  $C$  satisfiable exactly when the formula  $\phi$  is satisfiable? If  $C$  has a satisfying assignment, then each wire of the circuit has a well-defined value, and the output of the circuit is 1. Therefore, when we assign wire values to variables in  $\phi$ , each clause of  $\phi$  evaluates to 1, and thus the conjunction of all evaluates to 1. Conversely, if some assignment causes  $\phi$  to evaluate to 1, the circuit  $C$  is satisfiable by an analogous argument. Thus, we have shown that  $\text{CIRCUIT-SAT} \leq_p \text{SAT}$ , which completes the proof. ■

### 3-CNF satisfiability

We can prove many problems NP-complete by reducing from formula satisfiability. The reduction algorithm must handle any input formula, though, and this requirement can lead to a huge number of cases that we must consider. We often prefer to reduce from a restricted language of boolean formulas, so that we need to consider fewer cases. Of course, we must not restrict the language so much that it becomes polynomial-time solvable. One convenient language is 3-CNF satisfiability, or 3-CNF-SAT.

We define 3-CNF satisfiability using the following terms. A **literal** in a boolean formula is an occurrence of a variable or its negation. A boolean formula is in **conjunctive normal form**, or **CNF**, if it is expressed as an AND of **clauses**, each of which is the OR of one or more literals. A boolean formula is in **3-conjunctive normal form**, or **3-CNF**, if each clause has exactly three distinct literals.

For example, the boolean formula

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

is in 3-CNF. The first of its three clauses is  $(x_1 \vee \neg x_1 \vee \neg x_2)$ , which contains the three literals  $x_1$ ,  $\neg x_1$ , and  $\neg x_2$ .

In 3-CNF-SAT, we are asked whether a given boolean formula  $\phi$  in 3-CNF is satisfiable. The following theorem shows that a polynomial-time algorithm that can determine the satisfiability of boolean formulas is unlikely to exist, even when they are expressed in this simple normal form.

#### **Theorem 34.10**

Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.

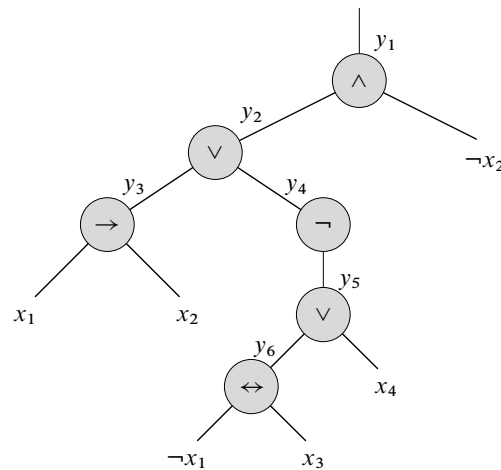
**Proof** The argument we used in the proof of Theorem 34.9 to show that  $\text{SAT} \in \text{NP}$  applies equally well here to show that  $3\text{-CNF-SAT} \in \text{NP}$ . By Lemma 34.8, therefore, we need only show that  $\text{SAT} \leq_p 3\text{-CNF-SAT}$ .

We break the reduction algorithm into three basic steps. Each step progressively transforms the input formula  $\phi$  closer to the desired 3-conjunctive normal form.

The first step is similar to the one used to prove  $\text{CIRCUIT-SAT} \leq_p \text{SAT}$  in Theorem 34.9. First, we construct a binary “parse” tree for the input formula  $\phi$ , with literals as leaves and connectives as internal nodes. Figure 34.11 shows such a parse tree for the formula

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2. \quad (34.3)$$

Should the input formula contain a clause such as the OR of several literals, we use associativity to parenthesize the expression fully so that every internal node in the resulting tree has 1 or 2 children. We can now think of the binary parse tree as a circuit for computing the function.



Mimicking the reduction in the proof of Theorem 34.9, we introduce a variable  $y_i$  for the output of each internal node. Then, we rewrite the original formula  $\phi$  as the AND of the root variable and a conjunction of clauses describing the operation of each node. For the formula (34.3), the resulting expression is

$$\begin{aligned}\phi' = & y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ & \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ & \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ & \wedge (y_4 \leftrightarrow \neg y_5) \\ & \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ & \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)).\end{aligned}$$

Observe that the formula  $\phi'$  thus obtained is a conjunction of clauses  $\phi'_i$ , each of which has at most 3 literals. The only requirement that we might fail to meet is that each clause has to be an OR of 3 literals.

The second step of the reduction converts each clause  $\phi'_i$  into conjunctive normal form. We construct a truth table for  $\phi'_i$  by evaluating all possible assignments to its variables. Each row of the truth table consists of a possible assignment of the variables of the clause, together with the value of the clause under that assignment. Using the truth-table entries that evaluate to 0, we build a formula in *disjunctive normal form* (or *DNF*)—an OR of ANDs—that is equivalent to  $\neg\phi'_i$ . We then negate this formula and convert it into a CNF formula  $\phi''_i$  by using *DeMorgan's*

| $y_1$ | $y_2$ | $x_2$ | $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ |
|-------|-------|-------|---|
| 1     | 1     | 1     | 0   |
| 1     | 1     | 0     | 1   |
| 1     | 0     | 1     | 0   |
| 1     | 0     | 0     | 0   |
| 0     | 1     | 1     | 1   |
| 0     | 1     | 0     | 0   |
| 0     | 0     | 1     | 1   |
| 0     | 0     | 0     | 1   |

**Figure 34.12** The truth table for the clause  $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ .

*laws* for propositional logic,

$$\neg(a \wedge b) = \neg a \vee \neg b ,$$

$$\neg(a \vee b) = \neg a \wedge \neg b ,$$

to complement all literals, change ORs into ANDs, and change ANDs into ORs.

In our example, we convert the clause  $\phi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$  into CNF as follows. The truth table for  $\phi'_1$  appears in Figure 34.12. The DNF formula equivalent to  $\neg\phi'_1$  is

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2) .$$

Negating and applying DeMorgan's laws, we get the CNF formula

$$\begin{aligned} \phi''_1 = & (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \\ & \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2) , \end{aligned}$$

which is equivalent to the original clause  $\phi'_1$ .

At this point, we have converted each clause  $\phi'_i$  of the formula  $\phi'$  into a CNF formula  $\phi''_i$ , and thus  $\phi'$  is equivalent to the CNF formula  $\phi''$  consisting of the conjunction of the  $\phi''_i$ . Moreover, each clause of  $\phi''$  has at most 3 literals.

The third and final step of the reduction further transforms the formula so that each clause has *exactly* 3 distinct literals. We construct the final 3-CNF formula  $\phi'''$  from the clauses of the CNF formula  $\phi''$ . The formula  $\phi'''$  also uses two auxiliary variables that we shall call  $p$  and  $q$ . For each clause  $C_i$  of  $\phi''$ , we include the following clauses in  $\phi'''$ :

- If  $C_i$  has 3 distinct literals, then simply include  $C_i$  as a clause of  $\phi'''$ .
- If  $C_i$  has 2 distinct literals, that is, if  $C_i = (l_1 \vee l_2)$ , where  $l_1$  and  $l_2$  are literals, then include  $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$  as clauses of  $\phi'''$ . The literals  $p$  and  $\neg p$  merely fulfill the syntactic requirement that each clause of  $\phi'''$  has

- If  $C_i$  has just 1 distinct literal  $l$ , then include  $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$  as clauses of  $\phi'''$ . Regardless of the values of  $p$  and  $q$ , one of the four clauses is equivalent to  $l$ , and the other 3 evaluate to 1.

We must also show that the reduction can be computed in polynomial time. Constructing  $\phi'$  from  $\phi$  introduces at most 1 variable and 1 clause per connective in  $\phi$ . Constructing  $\phi''$  from  $\phi'$  can introduce at most 8 clauses into  $\phi''$  for each clause from  $\phi'$ , since each clause of  $\phi'$  has at most 3 variables, and the truth table for each clause has at most  $2^3 = 8$  rows. The construction of  $\phi'''$  from  $\phi''$  introduces at most 4 clauses into  $\phi'''$  for each clause of  $\phi''$ . Thus, the size of the resulting formula  $\phi'''$  is polynomial in the length of the original formula. Each of the constructions can easily be accomplished in polynomial time. ■

**34.4-1**

Consider the straightforward (nonpolynomial-time) reduction in the proof of Theorem 34.9. Describe a circuit of size  $n$  that, when converted to a formula by this method, yields a formula whose size is exponential in  $n$ .

Show the 3-CNF formula that results when we use the method of Theorem 34.10 on the formula (34.3).

Professor Jagger proposes to show that  $\text{SAT} \leq_p \text{3-CNF-SAT}$  by using only the truth-table technique in the proof of Theorem 34.10, and not the other steps. That is, the professor proposes to take the boolean formula  $\phi$ , form a truth table for its variables, derive from the truth table a formula in 3-DNF that is equivalent to  $\neg\phi$ , and then negate and apply DeMorgan's laws to produce a 3-CNF formula equivalent to  $\phi$ . Show that this strategy does not yield a polynomial-time reduction.

**34.4-4**

Show that the problem of determining whether a boolean formula is a tautology is complete for co-NP. (*Hint:* See Exercise 34.3-7.)

**34.4-5**

Show that the problem of determining the satisfiability of boolean formulas in disjunctive normal form is polynomial-time solvable.

**34.4-6**

Suppose that someone gives you a polynomial-time algorithm to decide formula satisfiability. Describe how to use this algorithm to find satisfying assignments in polynomial time.

**34.4-7**

Let 2-CNF-SAT be the set of satisfiable boolean formulas in CNF with exactly 2 literals per clause. Show that 2-CNF-SAT  $\in$  P. Make your algorithm as efficient as possible. (*Hint:* Observe that  $x \vee y$  is equivalent to  $\neg x \rightarrow y$ . Reduce 2-CNF-SAT to an efficiently solvable problem on a directed graph.)

---

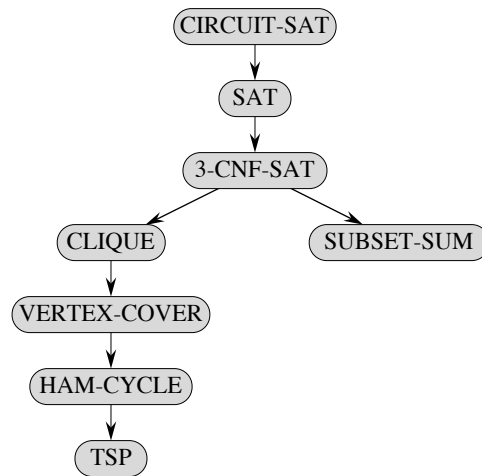
## 34.5 NP-complete problems

NP-complete problems arise in diverse domains: boolean logic, graphs, arithmetic, network design, sets and partitions, storage and retrieval, sequencing and scheduling, mathematical programming, algebra and number theory, games and puzzles, automata and language theory, program optimization, biology, chemistry, physics, and more. In this section, we shall use the reduction methodology to provide NP-completeness proofs for a variety of problems drawn from graph theory and set partitioning.

Figure 34.13 outlines the structure of the NP-completeness proofs in this section and Section 34.4. We prove each language in the figure to be NP-complete by reduction from the language that points to it. At the root is CIRCUIT-SAT, which we proved NP-complete in Theorem 34.7.

**34.5.1 The clique problem**

A *clique* in an undirected graph  $G = (V, E)$  is a subset  $V' \subseteq V$  of vertices, each pair of which is connected by an edge in  $E$ . In other words, a clique is a complete subgraph of  $G$ . The *size* of a clique is the number of vertices it contains. The *clique problem* is the optimization problem of finding a clique of maximum size in



**Figure 34.13** The structure of NP-completeness proofs in Sections 34.4 and 34.5. All proofs ultimately follow by reduction from the NP-completeness of CIRCUI-T-SAT.

a graph. As a decision problem, we ask simply whether a clique of a given size  $k$  exists in the graph. The formal definition is

$\text{CLIQUE} = \{ \langle G, k \rangle : G \text{ is a graph containing a clique of size } k \} .$

A naive algorithm for determining whether a graph  $G = (V, E)$  with  $|V|$  vertices has a clique of size  $k$  is to list all  $k$ -subsets of  $V$ , and check each one to see whether it forms a clique. The running time of this algorithm is  $\Omega(k^2 \binom{|V|}{k})$ , which is polynomial if  $k$  is a constant. In general, however,  $k$  could be near  $|V|/2$ , in which case the algorithm runs in superpolynomial time. Indeed, an efficient algorithm for the clique problem is unlikely to exist.

### Theorem 34.11

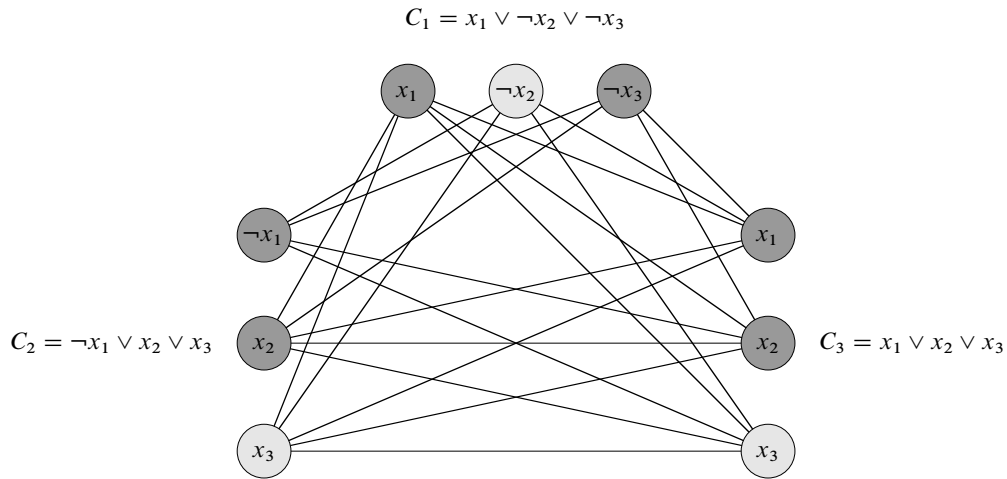
The clique problem is NP-complete.

**Proof** To show that  $\text{CLIQUE} \in \text{NP}$ , for a given graph  $G = (V, E)$ , we use the set  $V' \subseteq V$  of vertices in the clique as a certificate for  $G$ . We can check whether  $V'$  is a clique in polynomial time by checking whether, for each pair  $u, v \in V'$ , the edge  $(u, v)$  belongs to  $E$ .

We next prove that  $3\text{-CNF-SAT} \leq_p \text{CLIQUE}$ , which shows that the clique problem is NP-hard. You might be surprised that we should be able to prove such a result, since on the surface logical formulas seem to have little to do with graphs.

The reduction algorithm begins with an instance of 3-CNF-SAT. Let  $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$  be a boolean formula in 3-CNF with  $k$  clauses. For  $r =$





**Figure 34.14** The graph  $G$  derived from the 3-CNF formula  $\phi = C_1 \wedge C_2 \wedge C_3$ , where  $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$ ,  $C_2 = (\neg x_1 \vee x_2 \vee x_3)$ , and  $C_3 = (x_1 \vee x_2 \vee x_3)$ , in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has  $x_2 = 0$ ,  $x_3 = 1$ , and  $x_1$  either 0 or 1. This assignment satisfies  $C_1$  with  $\neg x_2$ , and it satisfies  $C_2$  and  $C_3$  with  $x_3$ , corresponding to the clique with lightly shaded vertices.

$1, 2, \dots, k$ , each clause  $C_r$  has exactly three distinct literals  $l_1^r$ ,  $l_2^r$ , and  $l_3^r$ . We shall construct a graph  $G$  such that  $\phi$  is satisfiable if and only if  $G$  has a clique of size  $k$ .

We construct the graph  $G = (V, E)$  as follows. For each clause  $C_r = (l_1^r \vee l_2^r \vee l_3^r)$  in  $\phi$ , we place a triple of vertices  $v_1^r$ ,  $v_2^r$ , and  $v_3^r$  into  $V$ . We put an edge between two vertices  $v_i^r$  and  $v_j^s$  if both of the following hold:

- $v_i^r$  and  $v_j^s$  are in different triples, that is,  $r \neq s$ , and
- their corresponding literals are **consistent**, that is,  $l_i^r$  is not the negation of  $l_j^s$ .

We can easily build this graph from  $\phi$  in polynomial time. As an example of this construction, if we have

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3),$$

then  $G$  is the graph shown in Figure 34.14.

We must show that this transformation of  $\phi$  into  $G$  is a reduction. First, suppose that  $\phi$  has a satisfying assignment. Then each clause  $C_r$  contains at least one literal  $l_i^r$  that is assigned 1, and each such literal corresponds to a vertex  $v_i^r$ . Picking one such “true” literal from each clause yields a set  $V'$  of  $k$  vertices. We claim that  $V'$  is a clique. For any two vertices  $v_i^r, v_j^s \in V'$ , where  $r \neq s$ , both corresponding literals  $l_i^r$  and  $l_j^s$  map to 1 by the given satisfying assignment, and thus the literals

cannot be complements. Thus, by the construction of  $G$ , the edge  $(v_i^r, v_j^s)$  belongs to  $E$ .

Conversely, suppose that  $G$  has a clique  $V'$  of size  $k$ . No edges in  $G$  connect vertices in the same triple, and so  $V'$  contains exactly one vertex per triple. We can assign 1 to each literal  $l_i^r$  such that  $v_i^r \in V'$  without fear of assigning 1 to both a literal and its complement, since  $G$  contains no edges between inconsistent literals. Each clause is satisfied, and so  $\phi$  is satisfied. (Any variables that do not correspond to a vertex in the clique may be set arbitrarily.) ■

In the example of Figure 34.14, a satisfying assignment of  $\phi$  has  $x_2 = 0$  and  $x_3 = 1$ . A corresponding clique of size  $k = 3$  consists of the vertices corresponding to  $\neg x_2$  from the first clause,  $x_3$  from the second clause, and  $x_3$  from the third clause. Because the clique contains no vertices corresponding to either  $x_1$  or  $\neg x_1$ , we can set  $x_1$  to either 0 or 1 in this satisfying assignment.

Observe that in the proof of Theorem 34.11, we reduced an arbitrary instance of 3-CNF-SAT to an instance of CLIQUE with a particular structure. You might think that we have shown only that CLIQUE is NP-hard in graphs in which the vertices are restricted to occur in triples and in which there are no edges between vertices in the same triple. Indeed, we have shown that CLIQUE is NP-hard only in this restricted case, but this proof suffices to show that CLIQUE is NP-hard in general graphs. Why? If we had a polynomial-time algorithm that solved CLIQUE on general graphs, it would also solve CLIQUE on restricted graphs.

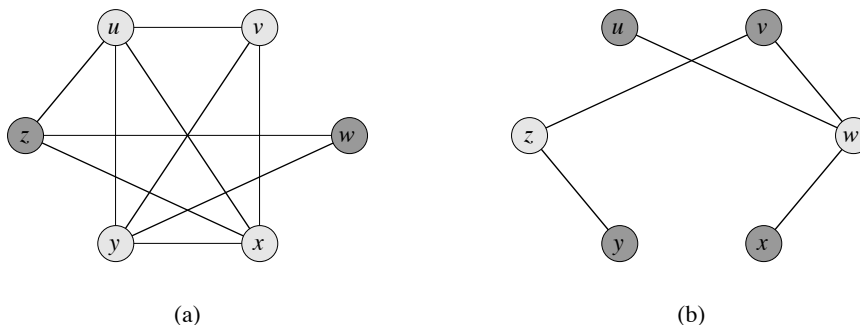
The opposite approach—reducing instances of 3-CNF-SAT with a special structure to general instances of CLIQUE—would not have sufficed, however. Why not? Perhaps the instances of 3-CNF-SAT that we chose to reduce from were “easy,” and so we would not have reduced an NP-hard problem to CLIQUE.

Observe also that the reduction used the instance of 3-CNF-SAT, but not the solution. We would have erred if the polynomial-time reduction had relied on knowing whether the formula  $\phi$  is satisfiable, since we do not know how to decide whether  $\phi$  is satisfiable in polynomial time.

### 34.5.2 The vertex-cover problem

A **vertex cover** of an undirected graph  $G = (V, E)$  is a subset  $V' \subseteq V$  such that if  $(u, v) \in E$ , then  $u \in V'$  or  $v \in V'$  (or both). That is, each vertex “covers” its incident edges, and a vertex cover for  $G$  is a set of vertices that covers all the edges in  $E$ . The **size** of a vertex cover is the number of vertices in it. For example, the graph in Figure 34.15(b) has a vertex cover  $\{w, z\}$  of size 2.

The **vertex-cover problem** is to find a vertex cover of minimum size in a given graph. Restating this optimization problem as a decision problem, we wish to



**Figure 34.15** Reducing CLIQUE to VERTEX-COVER. **(a)** An undirected graph  $G = (V, E)$  with clique  $V' = \{u, v, x, y\}$ . **(b)** The graph  $\bar{G}$  produced by the reduction algorithm that has vertex cover  $V - V' = \{w, z\}$ .

determine whether a graph has a vertex cover of a given size  $k$ . As a language, we define

$$\text{VERTEX-COVER} = \{\langle G, k \rangle : \text{graph } G \text{ has a vertex cover of size } k\}.$$

The following theorem shows that this problem is NP-complete.

**Theorem 34.12**

The vertex-cover problem is NP-complete.

**Proof** We first show that VERTEX-COVER  $\in$  NP. Suppose we are given a graph  $G = (V, E)$  and an integer  $k$ . The certificate we choose is the vertex cover  $V' \subseteq V$  itself. The verification algorithm affirms that  $|V'| = k$ , and then it checks, for each edge  $(u, v) \in E$ , that  $u \in V'$  or  $v \in V'$ . We can easily verify the certificate in polynomial time.

We prove that the vertex-cover problem is NP-hard by showing that CLIQUE  $\leq_p$  VERTEX-COVER. This reduction relies on the notion of the “complement” of a graph. Given an undirected graph  $G = (V, E)$ , we define the **complement** of  $G$  as  $\bar{G} = (V, \bar{E})$ , where  $\bar{E} = \{(u, v) : u, v \in V, u \neq v, \text{ and } (u, v) \notin E\}$ . In other words,  $\bar{G}$  is the graph containing exactly those edges that are not in  $G$ . Figure 34.15 shows a graph and its complement and illustrates the reduction from CLIQUE to VERTEX-COVER.

The reduction algorithm takes as input an instance  $\langle G, k \rangle$  of the clique problem. It computes the complement  $\bar{G}$ , which we can easily do in polynomial time. The output of the reduction algorithm is the instance  $\langle \bar{G}, |V| - k \rangle$  of the vertex-cover problem. To complete the proof, we show that this transformation is indeed a

reduction: the graph  $G$  has a clique of size  $k$  if and only if the graph  $\overline{G}$  has a vertex cover of size  $|V| - k$ .

Suppose that  $G$  has a clique  $V' \subseteq V$  with  $|V'| = k$ . We claim that  $V - V'$  is a vertex cover in  $\overline{G}$ . Let  $(u, v)$  be any edge in  $\overline{E}$ . Then,  $(u, v) \notin E$ , which implies that at least one of  $u$  or  $v$  does not belong to  $V'$ , since every pair of vertices in  $V'$  is connected by an edge of  $E$ . Equivalently, at least one of  $u$  or  $v$  is in  $V - V'$ , which means that edge  $(u, v)$  is covered by  $V - V'$ . Since  $(u, v)$  was chosen arbitrarily from  $\overline{E}$ , every edge of  $\overline{E}$  is covered by a vertex in  $V - V'$ . Hence, the set  $V - V'$ , which has size  $|V| - k$ , forms a vertex cover for  $\overline{G}$ .

Conversely, suppose that  $\overline{G}$  has a vertex cover  $V' \subseteq V$ , where  $|V'| = |V| - k$ . Then, for all  $u, v \in V$ , if  $(u, v) \in \overline{E}$ , then  $u \in V'$  or  $v \in V'$  or both. The contrapositive of this implication is that for all  $u, v \in V$ , if  $u \notin V'$  and  $v \notin V'$ , then  $(u, v) \in E$ . In other words,  $V - V'$  is a clique, and it has size  $|V| - |V'| = k$ . ■

Since VERTEX-COVER is NP-complete, we don't expect to find a polynomial-time algorithm for finding a minimum-size vertex cover. Section 35.1 presents a polynomial-time “approximation algorithm,” however, which produces “approximate” solutions for the vertex-cover problem. The size of a vertex cover produced by the algorithm is at most twice the minimum size of a vertex cover.

Thus, we shouldn't give up hope just because a problem is NP-complete. We may be able to design a polynomial-time approximation algorithm that obtains near-optimal solutions, even though finding an optimal solution is NP-complete. Chapter 35 gives several approximation algorithms for NP-complete problems.

### 34.5.3 The hamiltonian-cycle problem

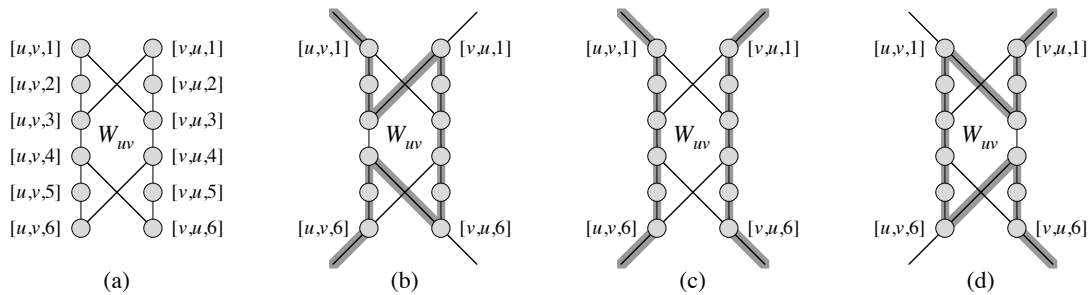
We now return to the hamiltonian-cycle problem defined in Section 34.2.

#### **Theorem 34.13**

The hamiltonian cycle problem is NP-complete.

**Proof** We first show that HAM-CYCLE belongs to NP. Given a graph  $G = (V, E)$ , our certificate is the sequence of  $|V|$  vertices that makes up the hamiltonian cycle. The verification algorithm checks that this sequence contains each vertex in  $V$  exactly once and that with the first vertex repeated at the end, it forms a cycle in  $G$ . That is, it checks that there is an edge between each pair of consecutive vertices and between the first and last vertices. We can verify the certificate in polynomial time.

We now prove that VERTEX-COVER  $\leq_P$  HAM-CYCLE, which shows that HAM-CYCLE is NP-complete. Given an undirected graph  $G = (V, E)$  and an

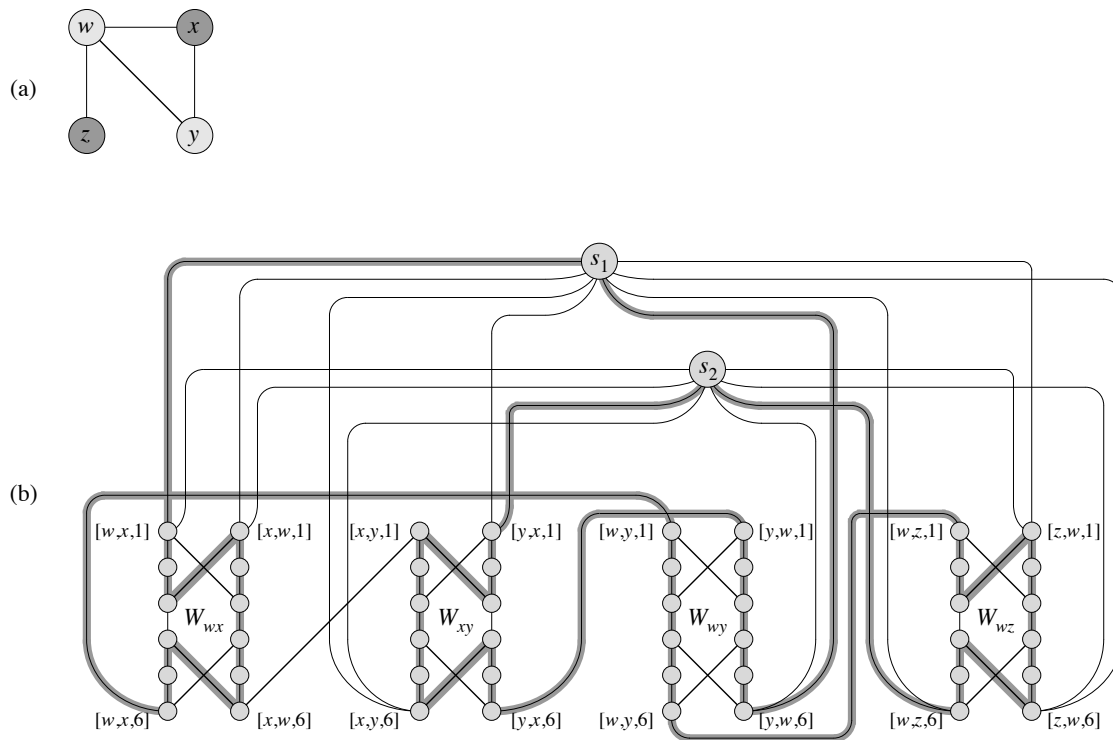


**Figure 34.16** The widget used in reducing the vertex-cover problem to the hamiltonian-cycle problem. An edge  $(u, v)$  of graph  $G$  corresponds to widget  $W_{uv}$  in the graph  $G'$  created in the reduction. (a) The widget, with individual vertices labeled. (b)–(d) The shaded paths are the only possible ones through the widget that include all vertices, assuming that the only connections from the widget to the remainder of  $G'$  are through vertices  $[u, v, 1]$ ,  $[u, v, 6]$ ,  $[v, u, 1]$ , and  $[v, u, 6]$ .

integer  $k$ , we construct an undirected graph  $G' = (V', E')$  that has a hamiltonian cycle if and only if  $G$  has a vertex cover of size  $k$ .

Our construction uses a **widget**, which is a piece of a graph that enforces certain properties. Figure 34.16(a) shows the widget we use. For each edge  $(u, v) \in E$ , the graph  $G'$  that we construct will contain one copy of this widget, which we denote by  $W_{uv}$ . We denote each vertex in  $W_{uv}$  by  $[u, v, i]$  or  $[v, u, i]$ , where  $1 \leq i \leq 6$ , so that each widget  $W_{uv}$  contains 12 vertices. Widget  $W_{uv}$  also contains the 14 edges shown in Figure 34.16(a).

Along with the internal structure of the widget, we enforce the properties we want by limiting the connections between the widget and the remainder of the graph  $G'$  that we construct. In particular, only vertices  $[u, v, 1]$ ,  $[u, v, 6]$ ,  $[v, u, 1]$ , and  $[v, u, 6]$  will have edges incident from outside  $W_{uv}$ . Any hamiltonian cycle of  $G'$  must traverse the edges of  $W_{uv}$  in one of the three ways shown in Figures 34.16(b)–(d). If the cycle enters through vertex  $[u, v, 1]$ , it must exit through vertex  $[u, v, 6]$ , and it either visits all 12 of the widget's vertices (Figure 34.16(b)) or the six vertices  $[u, v, 1]$  through  $[u, v, 6]$  (Figure 34.16(c)). In the latter case, the cycle will have to reenter the widget to visit vertices  $[v, u, 1]$  through  $[v, u, 6]$ . Similarly, if the cycle enters through vertex  $[v, u, 1]$ , it must exit through vertex  $[v, u, 6]$ , and it either visits all 12 of the widget's vertices (Figure 34.16(d)) or the six vertices  $[v, u, 1]$  through  $[v, u, 6]$  (Figure 34.16(c)). No other paths through the widget that visit all 12 vertices are possible. In particular, it is impossible to construct two vertex-disjoint paths, one of which connects  $[u, v, 1]$  to  $[v, u, 6]$  and the other of which connects  $[v, u, 1]$  to  $[u, v, 6]$ , such that the union of the two paths contains all of the widget's vertices.



**Figure 34.17** Reducing an instance of the vertex-cover problem to an instance of the hamiltonian-cycle problem. **(a)** An undirected graph  $G$  with a vertex cover of size 2, consisting of the lightly shaded vertices  $w$  and  $y$ . **(b)** The undirected graph  $G'$  produced by the reduction, with the hamiltonian cycle corresponding to the vertex cover shaded. The vertex cover  $\{w, y\}$  corresponds to edges  $(s_1, [w, x, 1])$  and  $(s_2, [y, x, 1])$  appearing in the hamiltonian cycle.

The only other vertices in  $V'$  other than those of widgets are *selector vertices*  $s_1, s_2, \dots, s_k$ . We use edges incident on selector vertices in  $G'$  to select the  $k$  vertices of the cover in  $G$ .

In addition to the edges in widgets,  $E'$  contains two other types of edges, which Figure 34.17 shows. First, for each vertex  $u \in V$ , we add edges to join pairs of widgets in order to form a path containing all widgets corresponding to edges incident on  $u$  in  $G$ . We arbitrarily order the vertices adjacent to each vertex  $u \in V$  as  $u^{(1)}, u^{(2)}, \dots, u^{(\text{degree}(u))}$ , where  $\text{degree}(u)$  is the number of vertices adjacent to  $u$ . We create a path in  $G'$  through all the widgets corresponding to edges incident on  $u$  by adding to  $E'$  the edges  $\{([u, u^{(i)}, 6], [u, u^{(i+1)}, 1]) : 1 \leq i \leq \text{degree}(u) - 1\}$ . In Figure 34.17, for example, we order the vertices adjacent to  $w$  as  $x, y, z$ , and so graph  $G'$  in part (b) of the figure includes the edges

$([w, x, 6], [w, y, 1])$  and  $([w, y, 6], [w, z, 1])$ . For each vertex  $u \in V$ , these edges in  $G'$  fill in a path containing all widgets corresponding to edges incident on  $u$  in  $G$ .

The intuition behind these edges is that if we choose a vertex  $u \in V$  in the vertex cover of  $G$ , we can construct a path from  $[u, u^{(1)}, 1]$  to  $[u, u^{(\text{degree}(u))}, 6]$  in  $G'$  that “covers” all widgets corresponding to edges incident on  $u$ . That is, for each of these widgets, say  $W_{u, u^{(i)}}$ , the path either includes all 12 vertices (if  $u$  is in the vertex cover but  $u^{(i)}$  is not) or just the six vertices  $[u, u^{(i)}, 1], [u, u^{(i)}, 2], \dots, [u, u^{(i)}, 6]$  (if both  $u$  and  $u^{(i)}$  are in the vertex cover).

The final type of edge in  $E'$  joins the first vertex  $[u, u^{(1)}, 1]$  and the last vertex  $[u, u^{(\text{degree}(u))}, 6]$  of each of these paths to each of the selector vertices. That is, we include the edges

$$\begin{aligned} & \{(s_j, [u, u^{(1)}, 1]) : u \in V \text{ and } 1 \leq j \leq k\} \\ & \cup \{(s_j, [u, u^{(\text{degree}(u))}, 6]) : u \in V \text{ and } 1 \leq j \leq k\} . \end{aligned}$$

Next, we show that the size of  $G'$  is polynomial in the size of  $G$ , and hence we can construct  $G'$  in time polynomial in the size of  $G$ . The vertices of  $G'$  are those in the widgets, plus the selector vertices. With 12 vertices per widget, plus  $k \leq |V|$  selector vertices, we have a total of

$$\begin{aligned} |V'| &= 12 |E| + k \\ &\leq 12 |E| + |V| \end{aligned}$$

vertices. The edges of  $G'$  are those in the widgets, those that go between widgets, and those connecting selector vertices to widgets. Each widget contains 14 edges, totaling  $14 |E|$  in all widgets. For each vertex  $u \in V$ , graph  $G'$  has  $\text{degree}(u) - 1$  edges going between widgets, so that summed over all vertices in  $V$ ,

$$\sum_{u \in V} (\text{degree}(u) - 1) = 2 |E| - |V|$$

edges go between widgets. Finally,  $G'$  has two edges for each pair consisting of a selector vertex and a vertex of  $V$ , totaling  $2k |V|$  such edges. The total number of edges of  $G'$  is therefore

$$\begin{aligned} |E'| &= (14 |E|) + (2 |E| - |V|) + (2k |V|) \\ &= 16 |E| + (2k - 1) |V| \\ &\leq 16 |E| + (2 |V| - 1) |V| . \end{aligned}$$

Now we show that the transformation from graph  $G$  to  $G'$  is a reduction. That is, we must show that  $G$  has a vertex cover of size  $k$  if and only if  $G'$  has a hamiltonian cycle.

Suppose that  $G = (V, E)$  has a vertex cover  $V^* \subseteq V$  of size  $k$ . Let  $V^* = \{u_1, u_2, \dots, u_k\}$ . As Figure 34.17 shows, we form a hamiltonian cycle in  $G'$  by including the following edges<sup>10</sup> for each vertex  $u_j \in V^*$ . Include edges  $\{([u_j, u_j^{(i)}, 6], [u_j, u_j^{(i+1)}, 1]) : 1 \leq i \leq \text{degree}(u_j) - 1\}$ , which connect all widgets corresponding to edges incident on  $u_j$ . We also include the edges within these widgets as Figures 34.16(b)–(d) show, depending on whether the edge is covered by one or two vertices in  $V^*$ . The hamiltonian cycle also includes the edges

$$\begin{aligned} &\{(s_j, [u_j, u_j^{(1)}, 1]) : 1 \leq j \leq k\} \\ &\cup \{(s_{j+1}, [u_j, u_j^{(\text{degree}(u_j))}, 6]) : 1 \leq j \leq k-1\} \\ &\cup \{(s_1, [u_k, u_k^{(\text{degree}(u_k))}, 6])\}. \end{aligned}$$

By inspecting Figure 34.17, you can verify that these edges form a cycle. The cycle starts at  $s_1$ , visits all widgets corresponding to edges incident on  $u_1$ , then visits  $s_2$ , visits all widgets corresponding to edges incident on  $u_2$ , and so on, until it returns to  $s_1$ . The cycle visits each widget either once or twice, depending on whether one or two vertices of  $V^*$  cover its corresponding edge. Because  $V^*$  is a vertex cover for  $G$ , each edge in  $E$  is incident on some vertex in  $V^*$ , and so the cycle visits each vertex in each widget of  $G'$ . Because the cycle also visits every selector vertex, it is hamiltonian.

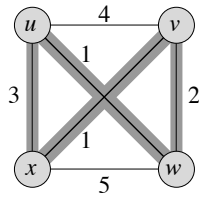
Conversely, suppose that  $G' = (V', E')$  has a hamiltonian cycle  $C \subseteq E'$ . We claim that the set

$$V^* = \{u \in V : (s_j, [u, u^{(1)}, 1]) \in C \text{ for some } 1 \leq j \leq k\} \quad (34.4)$$

is a vertex cover for  $G$ . To see why, partition  $C$  into maximal paths that start at some selector vertex  $s_i$ , traverse an edge  $(s_i, [u, u^{(1)}, 1])$  for some  $u \in V$ , and end at a selector vertex  $s_j$  without passing through any other selector vertex. Let us call each such path a “cover path.” From how  $G'$  is constructed, each cover path must start at some  $s_i$ , take the edge  $(s_i, [u, u^{(1)}, 1])$  for some vertex  $u \in V$ , pass through all the widgets corresponding to edges in  $E$  incident on  $u$ , and then end at some selector vertex  $s_j$ . We refer to this cover path as  $p_u$ , and by equation (34.4), we put  $u$  into  $V^*$ . Each widget visited by  $p_u$  must be  $W_{uv}$  or  $W_{vu}$  for some  $v \in V$ . For each widget visited by  $p_u$ , its vertices are visited by either one or two cover paths. If they are visited by one cover path, then edge  $(u, v) \in E$  is covered in  $G$  by vertex  $u$ . If two cover paths visit the widget, then the other cover path must be  $p_v$ , which implies that  $v \in V^*$ , and edge  $(u, v) \in E$  is covered by both  $u$  and  $v$ .

<sup>10</sup>Technically, we define a cycle in terms of vertices rather than edges (see Section B.4). In the interest of clarity, we abuse notation here and define the hamiltonian cycle in terms of edges.





**Figure 34.18** An instance of the traveling-salesman problem. Shaded edges represent a minimum-cost tour, with cost 7.

Because each vertex in each widget is visited by some cover path, we see that each edge in  $E$  is covered by some vertex in  $V^*$ . ■

### 34.5.4 The traveling-salesman problem

In the *traveling-salesman problem*, which is closely related to the hamiltonian-cycle problem, a salesman must visit  $n$  cities. Modeling the problem as a complete graph with  $n$  vertices, we can say that the salesman wishes to make a *tour*, or hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. The salesman incurs a nonnegative integer cost  $c(i, j)$  to travel from city  $i$  to city  $j$ , and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour. For example, in Figure 34.18, a minimum-cost tour is  $\langle u, w, v, x, u \rangle$ , with cost 7. The formal language for the corresponding decision problem is

TSP =  $\{ \langle G, c, k \rangle : G = (V, E) \text{ is a complete graph,}$   
 $c \text{ is a function from } V \times V \rightarrow \mathbb{N},$   
 $k \in \mathbb{N}, \text{ and}$   
 $G \text{ has a traveling-salesman tour with cost at most } k \} .$

The following theorem shows that a fast algorithm for the traveling-salesman problem is unlikely to exist.

#### **Theorem 34.14**

The traveling-salesman problem is NP-complete.

**Proof** We first show that TSP belongs to NP. Given an instance of the problem, we use as a certificate the sequence of  $n$  vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs, and checks whether the sum is at most  $k$ . This process can certainly be done in polynomial time.

To prove that TSP is NP-hard, we show that HAM-CYCLE  $\leq_p$  TSP. Let  $G = (V, E)$  be an instance of HAM-CYCLE. We construct an instance of TSP as follows. We form the complete graph  $G' = (V, E')$ , where  $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$ , and we define the cost function  $c$  by

$$c(i, j) = \begin{cases} 0 & \text{if } (i, j) \in E, \\ 1 & \text{if } (i, j) \notin E. \end{cases}$$

(Note that because  $G$  is undirected, it has no self-loops, and so  $c(v, v) = 1$  for all vertices  $v \in V$ .) The instance of TSP is then  $\langle G', c, 0 \rangle$ , which we can easily create in polynomial time.

We now show that graph  $G$  has a hamiltonian cycle if and only if graph  $G'$  has a tour of cost at most 0. Suppose that graph  $G$  has a hamiltonian cycle  $h$ . Each edge in  $h$  belongs to  $E$  and thus has cost 0 in  $G'$ . Thus,  $h$  is a tour in  $G'$  with cost 0. Conversely, suppose that graph  $G'$  has a tour  $h'$  of cost at most 0. Since the costs of the edges in  $E'$  are 0 and 1, the cost of tour  $h'$  is exactly 0 and each edge on the tour must have cost 0. Therefore,  $h'$  contains only edges in  $E$ . We conclude that  $h'$  is a hamiltonian cycle in graph  $G$ . ■

### 34.5.5 The subset-sum problem

We next consider an arithmetic NP-complete problem. In the **subset-sum problem**, we are given a finite set  $S$  of positive integers and an integer **target**  $t > 0$ . We ask whether there exists a subset  $S' \subseteq S$  whose elements sum to  $t$ . For example, if  $S = \{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993\}$  and  $t = 138457$ , then the subset  $S' = \{1, 2, 7, 98, 343, 686, 2409, 17206, 117705\}$  is a solution.

As usual, we define the problem as a language:

SUBSET-SUM =  $\{\langle S, t \rangle : \text{there exists a subset } S' \subseteq S \text{ such that } t = \sum_{s \in S'} s\}$ .

As with any arithmetic problem, it is important to recall that our standard encoding assumes that the input integers are coded in binary. With this assumption in mind, we can show that the subset-sum problem is unlikely to have a fast algorithm.

#### Theorem 34.15

The subset-sum problem is NP-complete.

**Proof** To show that SUBSET-SUM is in NP, for an instance  $\langle S, t \rangle$  of the problem, we let the subset  $S'$  be the certificate. A verification algorithm can check whether  $t = \sum_{s \in S'} s$  in polynomial time.

We now show that 3-CNF-SAT  $\leq_p$  SUBSET-SUM. Given a 3-CNF formula  $\phi$  over variables  $x_1, x_2, \dots, x_n$  with clauses  $C_1, C_2, \dots, C_k$ , each containing exactly

three distinct literals, the reduction algorithm constructs an instance  $\langle S, t \rangle$  of the subset-sum problem such that  $\phi$  is satisfiable if and only if there exists a subset of  $S$  whose sum is exactly  $t$ . Without loss of generality, we make two simplifying assumptions about the formula  $\phi$ . First, no clause contains both a variable and its negation, for such a clause is automatically satisfied by any assignment of values to the variables. Second, each variable appears in at least one clause, because it does not matter what value is assigned to a variable that appears in no clauses.

The reduction creates two numbers in set  $S$  for each variable  $x_i$  and two numbers in  $S$  for each clause  $C_j$ . We shall create numbers in base 10, where each number contains  $n+k$  digits and each digit corresponds to either one variable or one clause. Base 10 (and other bases, as we shall see) has the property we need of preventing carries from lower digits to higher digits.

As Figure 34.19 shows, we construct set  $S$  and target  $t$  as follows. We label each digit position by either a variable or a clause. The least significant  $k$  digits are labeled by the clauses, and the most significant  $n$  digits are labeled by variables.

- The target  $t$  has a 1 in each digit labeled by a variable and a 4 in each digit labeled by a clause.
- For each variable  $x_i$ , set  $S$  contains two integers  $v_i$  and  $v'_i$ . Each of  $v_i$  and  $v'_i$  has a 1 in the digit labeled by  $x_i$  and 0s in the other variable digits. If literal  $x_i$  appears in clause  $C_j$ , then the digit labeled by  $C_j$  in  $v_i$  contains a 1. If literal  $\neg x_i$  appears in clause  $C_j$ , then the digit labeled by  $C_j$  in  $v'_i$  contains a 1. All other digits labeled by clauses in  $v_i$  and  $v'_i$  are 0.

All  $v_i$  and  $v'_i$  values in set  $S$  are unique. Why? For  $l \neq i$ , no  $v_l$  or  $v'_l$  values can equal  $v_i$  and  $v'_i$  in the most significant  $n$  digits. Furthermore, by our simplifying assumptions above, no  $v_i$  and  $v'_i$  can be equal in all  $k$  least significant digits. If  $v_i$  and  $v'_i$  were equal, then  $x_i$  and  $\neg x_i$  would have to appear in exactly the same set of clauses. But we assume that no clause contains both  $x_i$  and  $\neg x_i$  and that either  $x_i$  or  $\neg x_i$  appears in some clause, and so there must be some clause  $C_j$  for which  $v_i$  and  $v'_i$  differ.

- For each clause  $C_j$ , set  $S$  contains two integers  $s_j$  and  $s'_j$ . Each of  $s_j$  and  $s'_j$  has 0s in all digits other than the one labeled by  $C_j$ . For  $s_j$ , there is a 1 in the  $C_j$  digit, and  $s'_j$  has a 2 in this digit. These integers are “slack variables,” which we use to get each clause-labeled digit position to add to the target value of 4.

Simple inspection of Figure 34.19 demonstrates that all  $s_j$  and  $s'_j$  values in  $S$  are unique in set  $S$ .

Note that the greatest sum of digits in any one digit position is 6, which occurs in the digits labeled by clauses (three 1s from the  $v_i$  and  $v'_i$  values, plus 1 and 2 from

|        |   | $x_1$ | $x_2$ | $x_3$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ |
|--------|---|-------|-------|-------|-------|-------|-------|-------|
| $v_1$  | = | 1     | 0     | 0     | 1     | 0     | 0     | 1     |
| $v'_1$ | = | 1     | 0     | 0     | 0     | 1     | 1     | 0     |
| $v_2$  | = | 0     | 1     | 0     | 0     | 0     | 0     | 1     |
| $v'_2$ | = | 0     | 1     | 0     | 1     | 1     | 1     | 0     |
| $v_3$  | = | 0     | 0     | 1     | 0     | 0     | 1     | 1     |
| $v'_3$ | = | 0     | 0     | 1     | 1     | 1     | 0     | 0     |
| $s_1$  | = | 0     | 0     | 0     | 1     | 0     | 0     | 0     |
| $s'_1$ | = | 0     | 0     | 0     | 2     | 0     | 0     | 0     |
| $s_2$  | = | 0     | 0     | 0     | 0     | 1     | 0     | 0     |
| $s'_2$ | = | 0     | 0     | 0     | 0     | 2     | 0     | 0     |
| $s_3$  | = | 0     | 0     | 0     | 0     | 0     | 1     | 0     |
| $s'_3$ | = | 0     | 0     | 0     | 0     | 0     | 2     | 0     |
| $s_4$  | = | 0     | 0     | 0     | 0     | 0     | 0     | 1     |
| $s'_4$ | = | 0     | 0     | 0     | 0     | 0     | 0     | 2     |
| $t$    | = | 1     | 1     | 1     | 4     | 4     | 4     | 4     |

**Figure 34.19** The reduction of 3-CNF-SAT to SUBSET-SUM. The formula in 3-CNF is  $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$ , where  $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$ ,  $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$ ,  $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$ , and  $C_4 = (x_1 \vee x_2 \vee x_3)$ . A satisfying assignment of  $\phi$  is  $\langle x_1 = 0, x_2 = 0, x_3 = 1 \rangle$ . The set  $S$  produced by the reduction consists of the base-10 numbers shown; reading from top to bottom,  $S = \{1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2\}$ . The target  $t$  is 1114444. The subset  $S' \subseteq S$  is lightly shaded, and it contains  $v'_1, v'_2$ , and  $v_3$ , corresponding to the satisfying assignment. It also contains slack variables  $s_1, s'_1, s'_2, s_3, s_4$ , and  $s'_4$  to achieve the target value of 4 in the digits labeled by  $C_1$  through  $C_4$ .

the  $s_j$  and  $s'_j$  values). Interpreting these numbers in base 10, therefore, no carries can occur from lower digits to higher digits.<sup>11</sup>

We can perform the reduction in polynomial time. The set  $S$  contains  $2n + 2k$  values, each of which has  $n + k$  digits, and the time to produce each digit is polynomial in  $n + k$ . The target  $t$  has  $n + k$  digits, and the reduction produces each in constant time.

We now show that the 3-CNF formula  $\phi$  is satisfiable if and only if there exists a subset  $S' \subseteq S$  whose sum is  $t$ . First, suppose that  $\phi$  has a satisfying assignment. For  $i = 1, 2, \dots, n$ , if  $x_i = 1$  in this assignment, then include  $v_i$  in  $S'$ . Otherwise, include  $v'_i$ . In other words, we include in  $S'$  exactly the  $v_i$  and  $v'_i$  values that cor-

<sup>11</sup>In fact, any base  $b$ , where  $b \geq 7$ , would work. The instance at the beginning of this subsection is the set  $S$  and target  $t$  in Figure 34.19 interpreted in base 7, with  $S$  listed in sorted order.

respond to literals with the value 1 in the satisfying assignment. Having included either  $v_i$  or  $v'_i$ , but not both, for all  $i$ , and having put 0 in the digits labeled by variables in all  $s_j$  and  $s'_j$ , we see that for each variable-labeled digit, the sum of the values of  $S'$  must be 1, which matches those digits of the target  $t$ . Because each clause is satisfied, the clause contains some literal with the value 1. Therefore, each digit labeled by a clause has at least one 1 contributed to its sum by a  $v_i$  or  $v'_i$  value in  $S'$ . In fact, 1, 2, or 3 literals may be 1 in each clause, and so each clause-labeled digit has a sum of 1, 2, or 3 from the  $v_i$  and  $v'_i$  values in  $S'$ . In Figure 34.19 for example, literals  $\neg x_1$ ,  $\neg x_2$ , and  $x_3$  have the value 1 in a satisfying assignment. Each of clauses  $C_1$  and  $C_4$  contains exactly one of these literals, and so together  $v'_1$ ,  $v'_2$ , and  $v_3$  contribute 1 to the sum in the digits for  $C_1$  and  $C_4$ . Clause  $C_2$  contains two of these literals, and  $v'_1$ ,  $v'_2$ , and  $v_3$  contribute 2 to the sum in the digit for  $C_2$ . Clause  $C_3$  contains all three of these literals, and  $v'_1$ ,  $v'_2$ , and  $v_3$  contribute 3 to the sum in the digit for  $C_3$ . We achieve the target of 4 in each digit labeled by clause  $C_j$  by including in  $S'$  the appropriate nonempty subset of slack variables  $\{s_j, s'_j\}$ . In Figure 34.19,  $S'$  includes  $s_1, s'_1, s'_2, s_3, s_4$ , and  $s'_4$ . Since we have matched the target in all digits of the sum, and no carries can occur, the values of  $S'$  sum to  $t$ .

Now, suppose that there is a subset  $S' \subseteq S$  that sums to  $t$ . The subset  $S'$  must include exactly one of  $v_i$  and  $v'_i$  for each  $i = 1, 2, \dots, n$ , for otherwise the digits labeled by variables would not sum to 1. If  $v_i \in S'$ , we set  $x_i = 1$ . Otherwise,  $v'_i \in S'$ , and we set  $x_i = 0$ . We claim that every clause  $C_j$ , for  $j = 1, 2, \dots, k$ , is satisfied by this assignment. To prove this claim, note that to achieve a sum of 4 in the digit labeled by  $C_j$ , the subset  $S'$  must include at least one  $v_i$  or  $v'_i$  value that has a 1 in the digit labeled by  $C_j$ , since the contributions of the slack variables  $s_j$  and  $s'_j$  together sum to at most 3. If  $S'$  includes a  $v_i$  that has a 1 in  $C_j$ 's position, then the literal  $x_i$  appears in clause  $C_j$ . Since we have set  $x_i = 1$  when  $v_i \in S'$ , clause  $C_j$  is satisfied. If  $S'$  includes a  $v'_i$  that has a 1 in that position, then the literal  $\neg x_i$  appears in  $C_j$ . Since we have set  $x_i = 0$  when  $v'_i \in S'$ , clause  $C_j$  is again satisfied. Thus, all clauses of  $\phi$  are satisfied, which completes the proof. ■

## Exercises

### 34.5-1

The **subgraph-isomorphism problem** takes two undirected graphs  $G_1$  and  $G_2$ , and it asks whether  $G_1$  is isomorphic to a subgraph of  $G_2$ . Show that the subgraph-isomorphism problem is NP-complete.

### 34.5-2

Given an integer  $m \times n$  matrix  $A$  and an integer  $m$ -vector  $b$ , the **0-1 integer-programming problem** asks whether there exists an integer  $n$ -vector  $x$  with ele-

ments in the set  $\{0, 1\}$  such that  $Ax \leq b$ . Prove that 0-1 integer programming is NP-complete. (*Hint*: Reduce from 3-CNF-SAT.)

### 34.5-3

The *integer linear-programming problem* is like the 0-1 integer-programming problem given in Exercise 34.5-2, except that the values of the vector  $x$  may be any integers rather than just 0 or 1. Assuming that the 0-1 integer-programming problem is NP-hard, show that the integer linear-programming problem is NP-complete.

### 34.5-4

Show how to solve the subset-sum problem in polynomial time if the target value  $t$  is expressed in unary.

### 34.5-5

The *set-partition problem* takes as input a set  $S$  of numbers. The question is whether the numbers can be partitioned into two sets  $A$  and  $\bar{A} = S - A$  such that  $\sum_{x \in A} x = \sum_{x \in \bar{A}} x$ . Show that the set-partition problem is NP-complete.

### 34.5-6

Show that the hamiltonian-path problem is NP-complete.

### 34.5-7

The *longest-simple-cycle problem* is the problem of determining a simple cycle (no repeated vertices) of maximum length in a graph. Formulate a related decision problem, and show that the decision problem is NP-complete.

### 34.5-8

In the *half 3-CNF satisfiability* problem, we are given a 3-CNF formula  $\phi$  with  $n$  variables and  $m$  clauses, where  $m$  is even. We wish to determine whether there exists a truth assignment to the variables of  $\phi$  such that exactly half the clauses evaluate to 0 and exactly half the clauses evaluate to 1. Prove that the half 3-CNF satisfiability problem is NP-complete.

---

## Problems

### 34-1 Independent set

An *independent set* of a graph  $G = (V, E)$  is a subset  $V' \subseteq V$  of vertices such that each edge in  $E$  is incident on at most one vertex in  $V'$ . The *independent-set problem* is to find a maximum-size independent set in  $G$ .

- a. Formulate a related decision problem for the independent-set problem, and prove that it is NP-complete. (*Hint*: Reduce from the clique problem.)
- b. Suppose that you are given a “black-box” subroutine to solve the decision problem you defined in part (a). Give an algorithm to find an independent set of maximum size. The running time of your algorithm should be polynomial in  $|V|$  and  $|E|$ , counting queries to the black box as a single step.

Although the independent-set decision problem is NP-complete, certain special cases are polynomial-time solvable.

- c. Give an efficient algorithm to solve the independent-set problem when each vertex in  $G$  has degree 2. Analyze the running time, and prove that your algorithm works correctly.
- d. Give an efficient algorithm to solve the independent-set problem when  $G$  is bipartite. Analyze the running time, and prove that your algorithm works correctly. (*Hint*: Use the results of Section 26.3.)

### 34-2 Bonnie and Clyde

Bonnie and Clyde have just robbed a bank. They have a bag of money and want to divide it up. For each of the following scenarios, either give a polynomial-time algorithm, or prove that the problem is NP-complete. The input in each case is a list of the  $n$  items in the bag, along with the value of each.

- a. The bag contains  $n$  coins, but only 2 different denominations: some coins are worth  $x$  dollars, and some are worth  $y$  dollars. Bonnie and Clyde wish to divide the money exactly evenly.
- b. The bag contains  $n$  coins, with an arbitrary number of different denominations, but each denomination is a nonnegative integer power of 2, i.e., the possible denominations are 1 dollar, 2 dollars, 4 dollars, etc. Bonnie and Clyde wish to divide the money exactly evenly.
- c. The bag contains  $n$  checks, which are, in an amazing coincidence, made out to “Bonnie or Clyde.” They wish to divide the checks so that they each get the exact same amount of money.
- d. The bag contains  $n$  checks as in part (c), but this time Bonnie and Clyde are willing to accept a split in which the difference is no larger than 100 dollars.

### 34-3 Graph coloring

Mapmakers try to use as few colors as possible when coloring countries on a map, as long as no two countries that share a border have the same color. We can model this problem with an undirected graph  $G = (V, E)$  in which each vertex represents a country and vertices whose respective countries share a border are adjacent. Then, a ***k*-coloring** is a function  $c : V \rightarrow \{1, 2, \dots, k\}$  such that  $c(u) \neq c(v)$  for every edge  $(u, v) \in E$ . In other words, the numbers  $1, 2, \dots, k$  represent the  $k$  colors, and adjacent vertices must have different colors. The **graph-coloring problem** is to determine the minimum number of colors needed to color a given graph.

- a. Give an efficient algorithm to determine a 2-coloring of a graph, if one exists.
- b. Cast the graph-coloring problem as a decision problem. Show that your decision problem is solvable in polynomial time if and only if the graph-coloring problem is solvable in polynomial time.
- c. Let the language 3-COLOR be the set of graphs that can be 3-colored. Show that if 3-COLOR is NP-complete, then your decision problem from part (b) is NP-complete.

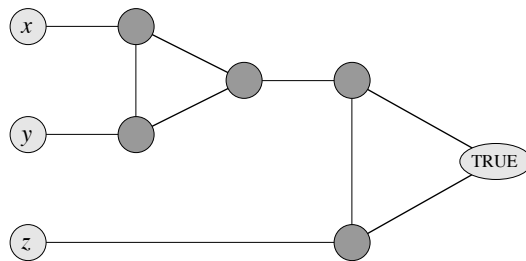
To prove that 3-COLOR is NP-complete, we use a reduction from 3-CNF-SAT. Given a formula  $\phi$  of  $m$  clauses on  $n$  variables  $x_1, x_2, \dots, x_n$ , we construct a graph  $G = (V, E)$  as follows. The set  $V$  consists of a vertex for each variable, a vertex for the negation of each variable, 5 vertices for each clause, and 3 special vertices: TRUE, FALSE, and RED. The edges of the graph are of two types: “literal” edges that are independent of the clauses and “clause” edges that depend on the clauses. The literal edges form a triangle on the special vertices and also form a triangle on  $x_i, \neg x_i$ , and RED for  $i = 1, 2, \dots, n$ .

- d. Argue that in any 3-coloring  $c$  of a graph containing the literal edges, exactly one of a variable and its negation is colored  $c(\text{TRUE})$  and the other is colored  $c(\text{FALSE})$ . Argue that for any truth assignment for  $\phi$ , there exists a 3-coloring of the graph containing just the literal edges.

The widget shown in Figure 34.20 helps to enforce the condition corresponding to a clause  $(x \vee y \vee z)$ . Each clause requires a unique copy of the 5 vertices that are heavily shaded in the figure; they connect as shown to the literals of the clause and the special vertex TRUE.

- e. Argue that if each of  $x$ ,  $y$ , and  $z$  is colored  $c(\text{TRUE})$  or  $c(\text{FALSE})$ , then the widget is 3-colorable if and only if at least one of  $x$ ,  $y$ , or  $z$  is colored  $c(\text{TRUE})$ .
- f. Complete the proof that 3-COLOR is NP-complete.





**Figure 34.20** The widget corresponding to a clause  $(x \vee y \vee z)$ , used in Problem 34-3.

### 34-4 Scheduling with profits and deadlines

Suppose that we have one machine and a set of  $n$  tasks  $a_1, a_2, \dots, a_n$ , each of which requires time on the machine. Each task  $a_j$  requires  $t_j$  time units on the machine (its processing time), yields a profit of  $p_j$ , and has a deadline  $d_j$ . The machine can process only one task at a time, and task  $a_j$  must run without interruption for  $t_j$  consecutive time units. If we complete task  $a_j$  by its deadline  $d_j$ , we receive a profit  $p_j$ , but if we complete it after its deadline, we receive no profit. As an optimization problem, we are given the processing times, profits, and deadlines for a set of  $n$  tasks, and we wish to find a schedule that completes all the tasks and returns the greatest amount of profit. The processing times, profits, and deadlines are all nonnegative numbers.

- State this problem as a decision problem.
- Show that the decision problem is NP-complete.
- Give a polynomial-time algorithm for the decision problem, assuming that all processing times are integers from 1 to  $n$ . (*Hint: Use dynamic programming.*)
- Give a polynomial-time algorithm for the optimization problem, assuming that all processing times are integers from 1 to  $n$ .

---

## Chapter notes

The book by Garey and Johnson [129] provides a wonderful guide to NP-completeness, discussing the theory at length and providing a catalogue of many problems that were known to be NP-complete in 1979. The proof of Theorem 34.13 is adapted from their book, and the list of NP-complete problem domains at the beginning of Section 34.5 is drawn from their table of contents. Johnson wrote a series

of 23 columns in the *Journal of Algorithms* between 1981 and 1992 reporting new developments in NP-completeness. Hopcroft, Motwani, and Ullman [177], Lewis and Papadimitriou [236], Papadimitriou [270], and Sipser [317] have good treatments of NP-completeness in the context of complexity theory. NP-completeness and several reductions also appear in books by Aho, Hopcroft, and Ullman [5]; Dasgupta, Papadimitriou, and Vazirani [82]; Johnsonbaugh and Schaefer [193]; and Kleinberg and Tardos [208].

The class P was introduced in 1964 by Cobham [72] and, independently, in 1965 by Edmonds [100], who also introduced the class NP and conjectured that  $P \neq NP$ . The notion of NP-completeness was proposed in 1971 by Cook [75], who gave the first NP-completeness proofs for formula satisfiability and 3-CNF satisfiability. Levin [234] independently discovered the notion, giving an NP-completeness proof for a tiling problem. Karp [199] introduced the methodology of reductions in 1972 and demonstrated the rich variety of NP-complete problems. Karp's paper included the original NP-completeness proofs of the clique, vertex-cover, and hamiltonian-cycle problems. Since then, thousands of problems have been proven to be NP-complete by many researchers. In a talk at a meeting celebrating Karp's 60th birthday in 1995, Papadimitriou remarked, "about 6000 papers each year have the term 'NP-complete' on their title, abstract, or list of keywords. This is more than each of the terms 'compiler,' 'database,' 'expert,' 'neural network,' or 'operating system.' "

Recent work in complexity theory has shed light on the complexity of computing approximate solutions. This work gives a new definition of NP using "probabilistically checkable proofs." This new definition implies that for problems such as clique, vertex cover, the traveling-salesman problem with the triangle inequality, and many others, computing good approximate solutions is NP-hard and hence no easier than computing optimal solutions. An introduction to this area can be found in Arora's thesis [20]; a chapter by Arora and Lund in Hochbaum [172]; a survey article by Arora [21]; a book edited by Mayr, Prömel, and Steger [246]; and a survey article by Johnson [191].