$O(n) + O(t^2(n))$ steps.

We have assumed that $t(n) \geq n$ (a reasonable assumption because M could not even read the entire input in less time). Therefore the running time of S is $O(t^2(n))$ and the proof is complete.

Next, we consider the analogous theorem for nondeterministic single-tape Turing machines. We show that any language that is decidable on such a machine is decidable on a deterministic single-tape Turing machine that requires significantly more time. Before doing so, we must define the running time of a nondeterministic Turing machine. Recall that a nondeterministic Turing machine is a decider if all its computation branches halt on all inputs.

DEFINITION 7.9

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f\colon \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.

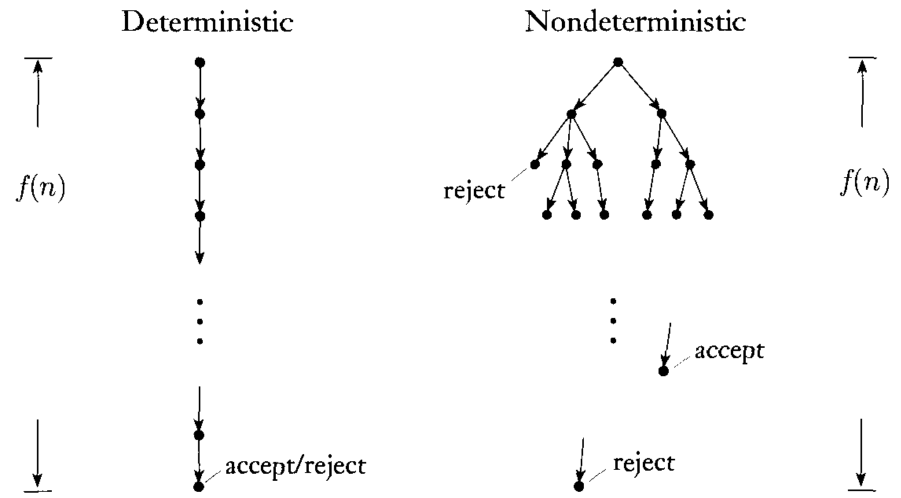

FIGURE 7.10
Measuring deterministic and nondeterministic time

The definition of the running time of a nondeterministic Turing machine is not intended to correspond to any real-world computing device. Rather, it is a useful mathematical definition that assists in characterizing the complexity of an important class of computational problems, as we demonstrate shortly.

THEOREM 7.11

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

PROOF Let N be a nondeterministic TM running in $t(n)$ time. We construct a deterministic TM D that simulates N as in the proof of Theorem 3.16 by searching N's nondeterministic computation tree. Now we analyze that simulation.

On an input of length n, every branch of N's nondeterministic computation tree has a length of at most $t(n)$. Every node in the tree can have at most b children, where b is the maximum number of legal choices given by N's transition function. Thus the total number of leaves in the tree is at most $b^{t(n)}$.

The simulation proceeds by exploring this tree breadth first. In other words, it visits all nodes at depth d before going on to any of the nodes at depth $d+1$. The algorithm given in the proof of Theorem 3.16 inefficiently starts at the root and travels down to a node whenever it visits that node, but eliminating this inefficiency doesn't alter the statement of the current theorem, so we leave it as is. The total number of nodes in the tree is less than twice the maximum number of leaves, so we bound it by $O(b^{t(n)})$. The time for starting from the root and traveling down to a node is $O(t(n))$. Therefore the running time of D is $O(t(n)b^{t(n)}) = 2^{O(t(n))}$.

As described in Theorem 3.16, the TM D has three tapes. Converting to a single-tape TM at most squares the running time, by Theorem 7.8. Thus the running time of the single-tape simulator is $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$, and the theorem is proved.

7.2 THE CLASS P

Theorems 7.8 and 7.11 illustrate an important distinction. On the one hand, we demonstrated at most a square or *polynomial* difference between the time complexity of problems measured on deterministic single-tape and multitape Turing machines. On the other hand, we showed at most an *exponential* difference between the time complexity of problems on deterministic and nondeterministic Turing machines.

POLYNOMIAL TIME

For our purposes, polynomial differences in running time are considered to be small, whereas exponential differences are considered to be large. Let's look at why we chose to make this separation between polynomials and exponentials rather than between some other classes of functions.

First, note the dramatic difference between the growth rate of typically occurring polynomials such as n^3 and typically occurring exponentials such as 2^n. For example, let n be 1000, the size of a reasonable input to an algorithm. In that case, n^3 is 1 billion, a large, but manageable number, whereas 2^n is a number much larger than the number of atoms in the universe. Polynomial time algorithms are fast enough for many purposes, but exponential time algorithms rarely are useful.

Exponential time algorithms typically arise when we solve problems by exhaustively searching through a space of solutions, called ***brute-force search***. For example, one way to factor a number into its constituent primes is to search through all potential divisors. The size of the search space is exponential, so this search uses exponential time. Sometimes, brute-force search may be avoided through a deeper understanding of a problem, which may reveal a polynomial time algorithm of greater utility.

All reasonable deterministic computational models are ***polynomially equivalent***. That is, any one of them can simulate another with only a polynomial increase in running time. When we say that all reasonable deterministic models are polynomially equivalent, we do not attempt to define *reasonable*. However, we have in mind a notion broad enough to include models that closely approximate running times on actual computers. For example, Theorem 7.8 shows that the deterministic single-tape and multitape Turing machine models are polynomially equivalent.

From here on we focus on aspects of time complexity theory that are unaffected by polynomial differences in running time. We consider such differences to be insignificant and ignore them. Doing so allows us to develop the theory in a way that doesn't depend on the selection of a particular model of computation. Remember, our aim is to present the fundamental properties of *computation*, rather than properties of Turing machines or any other special model.

You may feel that disregarding polynomial differences in running time is absurd. Real programmers certainly care about such differences and work hard just to make their programs run twice as quickly. However, we disregarded constant factors a while back when we introduced asymptotic notation. Now we propose to disregard the much greater polynomial differences, such as that between time n and time n^3.

Our decision to disregard polynomial differences doesn't imply that we consider such differences unimportant. On the contrary, we certainly do consider the difference between time n and time n^3 to be an important one. But some questions, such as the polynomiality or nonpolynomiality of the factoring problem, do not depend on polynomial differences and are important, too. We merely choose to focus on this type of question here. Ignoring the trees to see the forest doesn't mean that one is more important than the other—it just gives a different perspective.

Now we come to an important definition in complexity theory.

DEFINITION 7.12

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

The class P plays a central role in our theory and is important because

1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and
2. P roughly corresponds to the class of problems that are realistically solvable on a computer.

Item 1 indicates that P is a mathematically robust class. It isn't affected by the particulars of the model of computation that we are using.

Item 2 indicates that P is relevant from a practical standpoint. When a problem is in P, we have a method of solving it that runs in time n^k for some constant k. Whether this running time is practical depends on k and on the application. Of course, a running time of n^{100} is unlikely to be of any practical use. Nevertheless, calling polynomial time the threshold of practical solvability has proven to be useful. Once a polynomial time algorithm has been found for a problem that formerly appeared to require exponential time, some key insight into it has been gained, and further reductions in its complexity usually follow, often to the point of actual practical utility.

EXAMPLES OF PROBLEMS IN P

When we present a polynomial time algorithm, we give a high-level description of it without reference to features of a particular computational model. Doing so avoids tedious details of tapes and head motions. We need to follow certain conventions when describing an algorithm so that we can analyze it for polynomiality.

We describe algorithms with numbered stages. The notion of a stage of an algorithm is analogous to a step of a Turing machine, though of course, implementing one stage of an algorithm on a Turing machine, in general, will require many Turing machine steps.

When we analyze an algorithm to show that it runs in polynomial time, we need to do two things. First, we have to give a polynomial upper bound (usually in big-O notation) on the number of stages that the algorithm uses when it runs on an input of length n. Then, we have to examine the individual stages in the description of the algorithm to be sure that each can be implemented in polynomial time on a reasonable deterministic model. We choose the stages when we describe the algorithm to make this second part of the analysis easy to do. When both tasks have been completed, we can conclude that the algorithm runs in polynomial time because we have demonstrated that it runs for a polynomial number of stages, each of which can be done in polynomial time, and the composition of polynomials is a polynomial.

One point that requires attention is the encoding method used for problems. We continue to use the angle-bracket notation $\langle\cdot\rangle$ to indicate a reasonable encoding of one or more objects into a string, without specifying any particular encoding method. Now, a reasonable method is one that allows for polynomial time encoding and decoding of objects into natural internal representations or into other reasonable encodings. Familiar encoding methods for graphs, automata, and the like all are reasonable. But note that unary notation for encoding numbers (as in the number 17 encoded by the unary string 11111111111111111) isn't reasonable because it is exponentially larger than truly reasonable encodings, such as base k notation for any $k \geq 2$.

Many computational problems you encounter in this chapter contain encodings of graphs. One reasonable encoding of a graph is a list of its nodes and edges. Another is the ***adjacency matrix***, where the (i, j)th entry is 1 if there is an edge from node i to node j and 0 if not. When we analyze algorithms on graphs, the running time may be computed in terms of the number of nodes instead of the size of the graph representation. In reasonable graph representations, the size of the representation is a polynomial in the number of nodes. Thus, if we analyze an algorithm and show that its running time is polynomial (or exponential) in the number of nodes, we know that it is polynomial (or exponential) in the size of the input.

The first problem concerns directed graphs. A directed graph G contains nodes s and t, as shown in the following figure. The *PATH* problem is to determine whether a directed path exists from s to t. Let

$$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}.$$

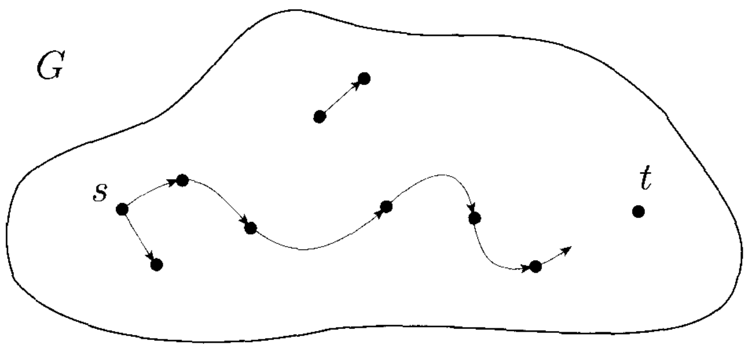

FIGURE 7.13
The *PATH* problem: Is there a path from s to t?

THEOREM 7.14

PATH $\in$ P.

PROOF IDEA We prove this theorem by presenting a polynomial time algorithm that decides *PATH*. Before describing that algorithm, let's observe that a brute-force algorithm for this problem isn't fast enough.

A brute-force algorithm for *PATH* proceeds by examining all potential paths in G and determining whether any is a directed path from s to t. A potential path is a sequence of nodes in G having a length of at most m, where m is the number of nodes in G. (If any directed path exists from s to t, one having a length of at most m exists because repeating a node never is necessary.) But the number of such potential paths is roughly m^m, which is exponential in the number of nodes in G. Therefore this brute-force algorithm uses exponential time.

To get a polynomial time algorithm for *PATH* we must do something that avoids brute force. One way is to use a graph-searching method such as breadth-first search. Here, we successively mark all nodes in G that are reachable from s by directed paths of length 1, then 2, then 3, through m. Bounding the running time of this strategy by a polynomial is easy.

PROOF A polynomial time algorithm M for *PATH* operates as follows.

```
M = "On input ⟨G, s, t⟩ where G is a directed graph with nodes s and t:
    1. Place a mark on node s.
    2. Repeat the following until no additional nodes are marked:
    3.    Scan all the edges of G. If an edge (a, b) is found going from
          a marked node a to an unmarked node b, mark node b.
    4. If t is marked, accept. Otherwise, reject."
```

Now we analyze this algorithm to show that it runs in polynomial time. Obviously, stages 1 and 4 are executed only once. Stage 3 runs at most m times because each time except the last it marks an additional node in G. Thus the total number of stages used is at most $1 + 1 + m$, giving a polynomial in the size of G.

Stages 1 and 4 of M are easily implemented in polynomial time on any reasonable deterministic model. Stage 3 involves a scan of the input and a test of whether certain nodes are marked, which also is easily implemented in polynomial time. Hence M is a polynomial time algorithm for *PATH*.

Let's turn to another example of a polynomial time algorithm. Say that two numbers are ***relatively prime*** if 1 is the largest integer that evenly divides them both. For example, 10 and 21 are relatively prime, even though neither of them is a prime number by itself, whereas 10 and 22 are not relatively prime because both are divisible by 2. Let *RELPRIME* be the problem of testing whether two numbers are relatively prime. Thus

$$\textit{RELPRIME} = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime}\}.$$

THEOREM 7.15

RELPRIME $\in$ P.

PROOF IDEA One algorithm that solves this problem searches through all possible divisors of both numbers and accepts if none are greater than 1. However, the magnitude of a number represented in binary, or in any other base k notation for $k \geq 2$, is exponential in the length of its representation. Therefore this brute-force algorithm searches through an exponential number of potential divisors and has an exponential running time.

Instead, we solve this problem with an ancient numerical procedure, called the ***Euclidean algorithm***, for computing the greatest common divisor. The ***greatest common divisor*** of natural numbers x and y, written $\gcd(x, y)$, is the largest integer that evenly divides both x and y. For example, $\gcd(18, 24) = 6$. Obviously, x and y are relatively prime iff $\gcd(x, y) = 1$. We describe the Euclidean algorithm as algorithm E in the proof. It uses the mod function, where $x \bmod y$ is the remainder after the integer division of x by y.

PROOF The Euclidean algorithm E is as follows.

```
E = "On input $\langle x, y\rangle$, where $x$ and $y$ are natural numbers in binary:
    1. Repeat until $y = 0$:
    2.    Assign $x \leftarrow x \bmod y$.
    3.    Exchange $x$ and $y$.
    4. Output $x$."
```

Algorithm R solves *RELPRIME*, using E as a subroutine.

```
R = "On input $\langle x, y\rangle$, where $x$ and $y$ are natural numbers in binary:
    1. Run $E$ on $\langle x, y\rangle$.
    2. If the result is 1, accept. Otherwise, reject."
```

Clearly, if E runs correctly in polynomial time, so does R and hence we only need to analyze E for time and correctness. The correctness of this algorithm is well known so we won't discuss it further here.

To analyze the time complexity of E, we first show that every execution of stage 2 (except possibly the first), cuts the value of x by at least half. After stage 2 is executed, $x < y$ because of the nature of the mod function. After stage 3, $x > y$ because the two have been exchanged. Thus, when stage 2 is subsequently executed, $x > y$. If $x/2 \geq y$, then $x \bmod y < y \leq x/2$ and x drops by at least half. If $x/2 < y$, then $x \bmod y = x - y < x/2$ and x drops by at least half.

The values of x and y are exchanged every time stage 3 is executed, so each of the original values of x and y are reduced by at least half every other time through the loop. Thus the maximum number of times that stages 2 and 3 are executed is the lesser of $2\log_2 x$ and $2\log_2 y$. These logarithms are proportional to the lengths of the representations, giving the number of stages executed as $O(n)$. Each stage of E uses only polynomial time, so the total running time is polynomial.

The final example of a polynomial time algorithm shows that every context-free language is decidable in polynomial time.

THEOREM 7.16

Every context-free language is a member of P.

PROOF IDEA In Theorem 4.9 we proved that every CFL is decidable. To do so we gave an algorithm for each CFL that decides it. If that algorithm runs in polynomial time, the current theorem follows as a corollary. Let's recall that algorithm and find out whether it runs quickly enough.

Let L be a CFL generated by CFG G that is in Chomsky normal form. From Problem 2.26, any derivation of a string w has $2n-1$ steps, where n is the length of w because G is in Chomsky normal form. The decider for L works by trying all possible derivations with $2n-1$ steps when its input is a string of length n. If any of these is a derivation of w, the decider accepts; if not, it rejects.

A quick analysis of this algorithm shows that it doesn't run in polynomial time. The number of derivations with k steps may be exponential in k, so this algorithm may require exponential time.

To get a polynomial time algorithm we introduce a powerful technique called ***dynamic programming***. This technique uses the accumulation of information about smaller subproblems to solve larger problems. We record the solution to any subproblem so that we need to solve it only once. We do so by making a table of all subproblems and entering their solutions systematically as we find them.

In this case, we consider the subproblems of determining whether each variable in G generates each substring of w. The algorithm enters the solution to this subproblem in an $n \times n$ table. For $i \leq j$ the (i,j)th entry of the table contains the collection of variables that generate the substring $w_i w_{i+1} \cdots w_j$. For $i > j$ the table entries are unused.

The algorithm fills in the table entries for each substring of w. First it fills in the entries for the substrings of length 1, then those of length 2, and so on. It uses the entries for the shorter lengths to assist in determining the entries for the longer lengths.

For example, suppose that the algorithm has already determined which variables generate all substrings up to length k. To determine whether a variable A generates a particular substring of length $k+1$ the algorithm splits that substring into two nonempty pieces in the k possible ways. For each split, the algorithm examines each rule $A \rightarrow BC$ to determine whether B generates the first piece and C generates the second piece, using table entries previously computed. If both B and C generate the respective pieces, A generates the substring and so is added to the associated table entry. The algorithm starts the process with the strings of length 1 by examining the table for the rules $A \rightarrow \mathtt{b}$.

PROOF The following algorithm D implements the proof idea. Let G be a CFG in Chomsky normal form generating the CFL L. Assume that S is the start variable. (Recall that the empty string is handled specially in a Chomsky normal form grammar. The algorithm handles the special case in which $w = \varepsilon$ in stage 1.) Comments appear inside double brackets.

```
$D =$ "On input $w = w_1 \cdots w_n$:
 1. If $w = \varepsilon$ and $S \rightarrow \varepsilon$ is a rule, accept.        [[ handle $w = \varepsilon$ case ]]
 2. For $i = 1$ to $n$:                  [[ examine each substring of length 1 ]]
 3.    For each variable $A$:
 4.       Test whether $A \rightarrow \mathtt{b}$ is a rule, where $\mathtt{b} = w_i$.
 5.       If so, place $A$ in $table(i, i)$.
 6. For $l = 2$ to $n$:                  [[ $l$ is the length of the substring ]]
 7.    For $i = 1$ to $n - l + 1$:       [[ $i$ is the start position of the substring ]]
 8.       Let $j = i + l - 1$,           [[ $j$ is the end position of the substring ]]
 9.       For $k = i$ to $j - 1$:        [[ $k$ is the split position ]]
10.          For each rule $A \rightarrow BC$:
11.             If $table(i, k)$ contains $B$ and $table(k + 1, j)$ contains
                $C$, put $A$ in $table(i, j)$.
12. If $S$ is in $table(1, n)$, accept. Otherwise, reject."
```

Now we analyze D. Each stage is easily implemented to run in polynomial time. Stages 4 and 5 run at most nv times, where v is the number of variables in G and is a fixed constant independent of n; hence these stages run $O(n)$ times. Stage 6 runs at most n times. Each time stage 6 runs, stage 7 runs at most n times. Each time stage 7 runs, stages 8 and 9 run at most n times. Each time stage 9 runs, stage 10 runs r times, where r is the number of rules of G and is another fixed constant. Thus stage 11, the inner loop of the algorithm, runs $O(n^3)$ times. Summing the total shows that D executes $O(n^3)$ stages.

7.3 THE CLASS NP

As we observed in Section 7.2, we can avoid brute-force search in many problems and obtain polynomial time solutions. However, attempts to avoid brute force in certain other problems, including many interesting and useful ones, haven't been successful, and polynomial time algorithms that solve them aren't known to exist.

Why have we been unsuccessful in finding polynomial time algorithms for these problems? We don't know the answer to this important question. Perhaps these problems have, as yet undiscovered, polynomial time algorithms that rest on unknown principles. Or possibly some of these problems simply *cannot* be solved in polynomial time. They may be intrinsically difficult.

One remarkable discovery concerning this question shows that the complexities of many problems are linked. A polynomial time algorithm for one such problem can be used to solve an entire class of problems. To understand this phenomenon, let's begin with an example.

A ***Hamiltonian path*** in a directed graph G is a directed path that goes through each node exactly once. We consider the problem of testing whether a directed graph contains a Hamiltonian path connecting two specified nodes, as shown in the following figure. Let

$$\begin{aligned} HAMPATH = \{\langle G, s, t\rangle \mid\ & G \text{ is a directed graph} \\ & \text{with a Hamiltonian path from } s \text{ to } t\}. \end{aligned}$$

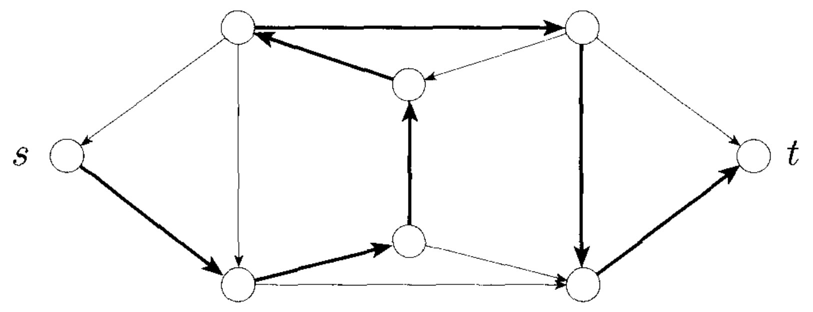

FIGURE 7.17
A Hamiltonian path goes through every node exactly once

We can easily obtain an exponential time algorithm for the *HAMPATH* problem by modifying the brute-force algorithm for *PATH* given in Theorem 7.14. We need only add a check to verify that the potential path is Hamiltonian. No one knows whether *HAMPATH* is solvable in polynomial time.

The *HAMPATH* problem does have a feature called ***polynomial verifiabil-***

ity that is important for understanding its complexity. Even though we don't know of a fast (i.e., polynomial time) way to determine whether a graph contains a Hamiltonian path, if such a path were discovered somehow (perhaps using the exponential time algorithm), we could easily convince someone else of its existence, simply by presenting it. In other words, *verifying* the existence of a Hamiltonian path may be much easier than *determining* its existence.

Another polynomially verifiable problem is compositeness. Recall that a natural number is ***composite*** if it is the product of two integers greater than 1 (i.e., a composite number is one that is not a prime number). Let

$$COMPOSITES = \{x \mid x = pq, \text{ for integers } p, q > 1\}.$$

We can easily verify that a number is composite—all that is needed is a divisor of that number. Recently, a polynomial time algorithm for testing whether a number is prime or composite was discovered, but it is considerably more complicated than the preceding method for verifying compositeness.

Some problems may not be polynomially verifiable. For example, take $\overline{HAMPATH}$, the complement of the *HAMPATH* problem. Even if we could determine (somehow) that a graph did *not* have a Hamiltonian path, we don't know of a way for someone else to verify its nonexistence without using the same exponential time algorithm for making the determination in the first place. A formal definition follows.

DEFINITION 7.18

A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

We measure the time of a verifier only in terms of the length of w, so a ***polynomial time verifier*** runs in polynomial time in the length of w. A language A is ***polynomially verifiable*** if it has a polynomial time verifier.

A verifier uses additional information, represented by the symbol c in Definition 7.18, to verify that a string w is a member of A. This information is called a ***certificate***, or ***proof***, of membership in A. Observe that, for polynomial verifiers, the certificate has polynomial length (in the length of w) because that is all the verifier can access in its time bound. Let's apply this definition to the languages *HAMPATH* and *COMPOSITES*.

For the *HAMPATH* problem, a certificate for a string $\langle G, s, t\rangle \in HAMPATH$ simply is the Hamiltonian path from s to t. For the *COMPOSITES* problem, a certificate for the composite number x simply is one of its divisors. In both cases the verifier can check in polynomial time that the input is in the language when it is given the certificate.

DEFINITION 7.19

NP is the class of languages that have polynomial time verifiers.

The class NP is important because it contains many problems of practical interest. From the preceding discussion, both *HAMPATH* and *COMPOSITES* are members of NP. As we mentioned, *COMPOSITES* is also a member of P which is a subset of NP, but proving this stronger result is much more difficult. The term NP comes from ***nondeterministic polynomial time*** and is derived from an alternative characterization by using nondeterministic polynomial time Turing machines. Problems in NP are sometimes called NP-problems.

The following is a nondeterministic Turing machine (NTM) that decides the *HAMPATH* problem in nondeterministic polynomial time. Recall that in Definition 7.9 we defined the time of a nondeterministic machine to be the time used by the longest computation branch.

```
$N_1$ = "On input $\langle G, s, t\rangle$, where $G$ is a directed graph with nodes $s$ and $t$:
    1. Write a list of $m$ numbers, $p_1, \ldots, p_m$, where $m$ is the number
       of nodes in $G$. Each number in the list is nondeterministically
       selected to be between 1 and $m$.
    2. Check for repetitions in the list. If any are found, reject.
    3. Check whether $s = p_1$ and $t = p_m$. If either fail, reject.
    4. For each $i$ between 1 and $m - 1$, check whether $(p_i, p_{i+1})$ is an
       edge of $G$. If any are not, reject. Otherwise, all tests have been
       passed, so accept."
```

To analyze this algorithm and verify that it runs in nondeterministic polynomial time, we examine each of its stages. In stage 1, the nondeterministic selection clearly runs in polynomial time. In stages 2 and 3, each part is a simple check, so together they run in polynomial time. Finally, stage 4 also clearly runs in polynomial time. Thus this algorithm runs in nondeterministic polynomial time.

THEOREM 7.20

A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

PROOF IDEA We show how to convert a polynomial time verifier to an equivalent polynomial time NTM and vice versa. The NTM simulates the verifier by guessing the certificate. The verifier simulates the NTM by using the accepting branch as the certificate.

PROOF For the forward direction of this theorem, let $A \in$ NP and show that A is decided by a polynomial time NTM N. Let V be the polynomial time verifier for A that exists by the definition of NP. Assume that V is a TM that runs in time n^k and construct N as follows.

```
N = "On input $w$ of length $n$:
    1. Nondeterministically select string $c$ of length at most $n^k$.
    2. Run $V$ on input $\langle w, c\rangle$.
    3. If $V$ accepts, accept; otherwise, reject."
```

To prove the other direction of the theorem, assume that A is decided by a polynomial time NTM N and construct a polynomial time verifier V as follows.

```
V = "On input $\langle w, c\rangle$, where $w$ and $c$ are strings:
    1. Simulate $N$ on input $w$, treating each symbol of $c$ as a description
       of the nondeterministic choice to make at each step (as in
       the proof of Theorem 3.16).
    2. If this branch of $N$'s computation accepts, accept; otherwise,
       reject."
```

We define the nondeterministic time complexity class NTIME($t(n)$) as analogous to the deterministic time complexity class TIME($t(n)$).

DEFINITION 7.21

NTIME(*t(n)*) $= \{L \mid L$ is a language decided by a $O(t(n))$ time nondeterministic Turing machine$\}$.

COROLLARY 7.22

$\text{NP} = \bigcup_k \text{NTIME}(n^k)$.

The class NP is insensitive to the choice of reasonable nondeterministic computational model because all such models are polynomially equivalent. When describing and analyzing nondeterministic polynomial time algorithms, we follow the preceding conventions for deterministic polynomial time algorithms. Each stage of a nondeterministic polynomial time algorithm must have an obvious implementation in nondeterministic polynomial time on a reasonable nondeterministic computational model. We analyze the algorithm to show that every branch uses at most polynomially many stages.

EXAMPLES OF PROBLEMS IN NP

A ***clique*** in an undirected graph is a subgraph, wherein every two nodes are connected by an edge. A ***k-clique*** is a clique that contains k nodes. Figure 7.23 illustrates a graph having a 5-clique

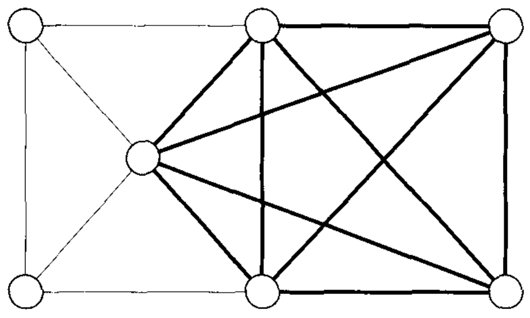

FIGURE 7.23
A graph with a 5-clique

The clique problem is to determine whether a graph contains a clique of a specified size. Let

$$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}.$$

THEOREM 7.24

CLIQUE is in NP.

PROOF IDEA The clique is the certificate.

PROOF The following is a verifier V for *CLIQUE*.

```
V = "On input ⟨⟨G, k⟩, c⟩:
    1. Test whether $c$ is a set of $k$ nodes in $G$
    2. Test whether $G$ contains all edges connecting nodes in $c$.
    3. If both pass, accept; otherwise, reject."
```

ALTERNATIVE PROOF If you prefer to think of NP in terms of nondeterministic polynomial time Turing machines, you may prove this theorem by giving one that decides *CLIQUE*. Observe the similarity between the two proofs.

```
N = "On input ⟨G, k⟩, where $G$ is a graph:
    1. Nondeterministically select a subset $c$ of $k$ nodes of $G$.
    2. Test whether $G$ contains all edges connecting nodes in $c$.
    3. If yes, accept; otherwise, reject."
```

Next we consider the *SUBSET-SUM* problem concerning integer arithmetic. In this problem we have a collection of numbers $x_1, \ldots, x_k$ and a target number t. We want to determine whether the collection contains a subcollection that adds up to t. Thus

$$SUBSET\text{-}SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\} \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\}.$$

For example, $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in$ *SUBSET-SUM* because $4 + 21 = 25$. Note that $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_l\}$ are considered to be ***multisets*** and so allow repetition of elements.

THEOREM 7.25

SUBSET-SUM is in NP.

PROOF IDEA The subset is the certificate.

PROOF The following is a verifier V for *SUBSET-SUM*.

```
V = "On input ⟨⟨S, t⟩, c⟩:
    1. Test whether c is a collection of numbers that sum to t.
    2. Test whether S contains all the numbers in c.
    3. If both pass, accept; otherwise, reject."
```

ALTERNATIVE PROOF We can also prove this theorem by giving a nondeterministic polynomial time Turing machine for *SUBSET-SUM* as follows.

```
N = "On input ⟨S, t⟩:
    1. Nondeterministically select a subset c of the numbers in S.
    2. Test whether c is a collection of numbers that sum to t.
    3. If the test passes, accept; otherwise, reject."
```

Observe that the complements of these sets, $\overline{CLIQUE}$ and $\overline{SUBSET\text{-}SUM}$, are not obviously members of NP. Verifying that something is *not* present seems to be more difficult than verifying that it *is* present. We make a separate complexity class, called coNP, which contains the languages that are complements of languages in NP. We don't know whether coNP is different from NP.

THE P VERSUS NP QUESTION

As we have been saying, NP is the class of languages that are solvable in polynomial time on a nondeterministic Turing machine, or, equivalently, it is the class of languages whereby membership in the language can be verified in polynomial time. P is the class of languages where membership can be tested in polynomial time. We summarize this information as follows, where we loosely refer to polynomial time solvable as solvable "quickly."

P = the class of languages for which membership can be *decided* quickly.
NP = the class of languages for which membership can be *verified* quickly.

We have presented examples of languages, such as *HAMPATH* and *CLIQUE*, that are members of NP but that are not known to be in P. The power of polynomial verifiability seems to be much greater than that of polynomial decidability. But, hard as it may be to imagine, P and NP could be equal. We are unable to *prove* the existence of a single language in NP that is not in P.

The question of whether P = NP is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics. If these classes were equal, any polynomially verifiable problem would be polynomially decidable. Most researchers believe that the two classes are not equal because people have invested enormous effort to find polynomial time algorithms for certain problems in NP, without success. Researchers also have tried proving that the classes are unequal, but that would entail showing that no fast algorithm exists to replace brute-force search. Doing so is presently beyond scientific reach. The following figure shows the two possibilities.

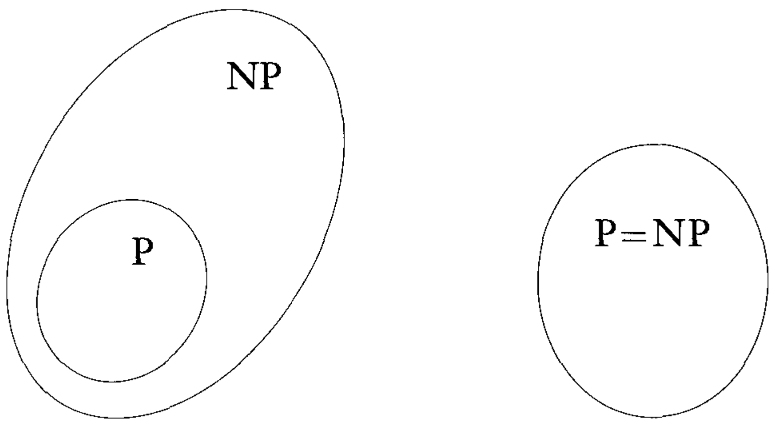

FIGURE 7.26
One of these two possibilities is correct

The best method known for solving languages in NP deterministically uses exponential time. In other words, we can prove that

$$\text{NP} \subseteq \text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k}),$$

but we don't know whether NP is contained in a smaller deterministic time complexity class.

7.4 NP-COMPLETENESS

One important advance on the P versus NP question came in the early 1970s with the work of Stephen Cook and Leonid Levin. They discovered certain problems in NP whose individual complexity is related to that of the entire class. If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called ***NP-complete***. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

On the theoretical side, a researcher trying to show that P is unequal to NP may focus on an NP-complete problem. If any problem in NP requires more than polynomial time, an NP-complete one does. Furthermore, a researcher attempting to prove that P equals NP only needs to find a polynomial time algorithm for an NP-complete problem to achieve this goal.

On the practical side, the phenomenon of NP-completeness may prevent wasting time searching for a nonexistent polynomial time algorithm to solve a particular problem. Even though we may not have the necessary mathematics to prove that the problem is unsolvable in polynomial time, we believe that P is unequal to NP, so proving that a problem is NP-complete is strong evidence of its nonpolynomiality.

The first NP-complete problem that we present is called the ***satisfiability problem***. Recall that variables that can take on the values TRUE and FALSE are called ***Boolean variables*** (see Section 0.2). Usually, we represent TRUE by 1 and FALSE by 0. The ***Boolean operations*** AND, OR, and NOT, represented by the symbols $\wedge$, $\vee$, and $\neg$, respectively, are described in the following list. We use the overbar as a shorthand for the $\neg$ symbol, so $\overline{x}$ means $\neg\, x$.

$$\begin{array}{lll}
0 \wedge 0 = 0 & 0 \vee 0 = 0 & \overline{0} = 1 \\
0 \wedge 1 = 0 & 0 \vee 1 = 1 & \overline{1} = 0 \\
1 \wedge 0 = 0 & 1 \vee 0 = 1 & \\
1 \wedge 1 = 1 & 1 \vee 1 = 1 &
\end{array}$$

A ***Boolean formula*** is an expression involving Boolean variables and operations. For example,

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

is a Boolean formula. A Boolean formula is ***satisfiable*** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1. The preceding formula is satisfiable because the assignment $x = 0$, $y = 1$, and $z = 0$ makes ϕ evaluate to 1. We say the assignment *satisfies* ϕ. The ***satisfiability problem*** is to test whether a Boolean formula is satisfiable. Let

$$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}.$$

Now we state the Cook–Levin theorem, which links the complexity of the SAT problem to the complexities of all problems in NP.

THEOREM 7.27

Cook–Levin theorem $SAT \in \text{P}$ iff P = NP.

Next, we develop the method that is central to the proof of the Cook–Levin theorem.

POLYNOMIAL TIME REDUCIBILITY

In Chapter 5 we defined the concept of reducing one problem to another. When problem A reduces to problem B, a solution to B can be used to solve A. Now we define a version of reducibility that takes the efficiency of computation into account. When problem A is *efficiently* reducible to problem B, an efficient solution to B can be used to solve A efficiently.

DEFINITION 7.28

A function $f\colon \Sigma^* \longrightarrow \Sigma^*$ is a ***polynomial time computable function*** if some polynomial time Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input w.

DEFINITION 7.29

Language A is ***polynomial time mapping reducible***,[1] or simply ***polynomial time reducible***, to language B, written $A \leq_{\text{P}} B$, if a polynomial time computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$ exists, where for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the ***polynomial time reduction*** of A to B.

Polynomial time reducibility is the efficient analog to mapping reducibility as defined in Section 5.3. Other forms of efficient reducibility are available, but polynomial time reducibility is a simple form that is adequate for our purposes so we won't discuss the others here. The following figure illustrates polynomial time reducibility.

[1] It is called ***polynomial time many–one reducibility*** in some other textbooks.

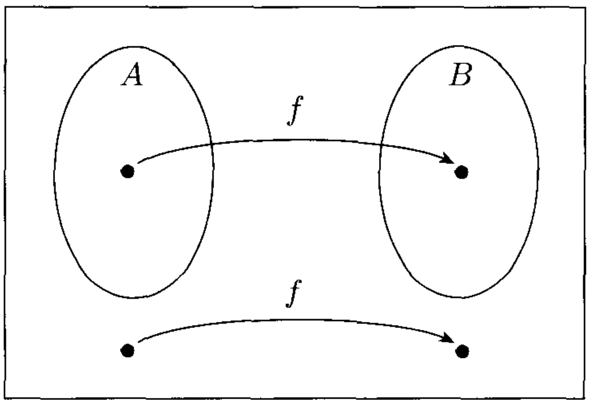

FIGURE 7.30
Polynomial time function f reducing A to B

As with an ordinary mapping reduction, a polynomial time reduction of A to B provides a way to convert membership testing in A to membership testing in B, but now the conversion is done efficiently. To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$.

If one language is polynomial time reducible to a language already known to have a polynomial time solution, we obtain a polynomial time solution to the original language, as in the following theorem.

THEOREM 7.31

If $A \leq_{\mathrm{P}} B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$.

PROOF Let M be the polynomial time algorithm deciding B and f be the polynomial time reduction from A to B. We describe a polynomial time algorithm N deciding A as follows.

```
N = "On input w:
    1. Compute f(w).
    2. Run M on input f(w) and output whatever M outputs."
```

We have $w \in A$ whenever $f(w) \in B$ because f is a reduction from A to B. Thus M accepts $f(w)$ whenever $w \in A$. Moreover, N runs in polynomial time because each of its two stages runs in polynomial time. Note that stage 2 runs in polynomial time because the composition of two polynomials is a polynomial.

Before demonstrating a polynomial time reduction we introduce *3SAT*, a special case of the satisfiability problem whereby all formulas are in a special form. A

literal is a Boolean variable or a negated Boolean variable, as in x or $\overline{x}$. A ***clause*** is several literals connected with $\vee$s, as in $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$. A Boolean formula is in ***conjunctive normal form***, called a ***cnf-formula***, if it comprises several clauses connected with $\wedge$s, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6}).$$

It is a ***3cnf-formula*** if all the clauses have three literals, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6).$$

Let *3SAT* $= \{\langle\phi\rangle \mid \phi$ is a satisfiable 3cnf-formula$\}$. In a satisfiable cnf-formula, each clause must contain at least one literal that is assigned 1.

The following theorem presents a polynomial time reduction from the *3SAT* problem to the *CLIQUE* problem.

THEOREM 7.32

3SAT is polynomial time reducible to *CLIQUE*.

PROOF IDEA The polynomial time reduction f that we demonstrate from *3SAT* to *CLIQUE* converts formulas to graphs. In the constructed graphs, cliques of a specified size correspond to satisfying assignments of the formula. Structures within the graph are designed to mimic the behavior of the variables and clauses.

PROOF Let ϕ be a formula with k clauses such as

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \quad \cdots \quad \wedge (a_k \vee b_k \vee c_k).$$

The reduction f generates the string $\langle G, k\rangle$, where G is an undirected graph defined as follows.

The nodes in G are organized into k groups of three nodes each called the ***triples***, $t_1, \ldots, t_k$. Each triple corresponds to one of the clauses in ϕ, and each node in a triple corresponds to a literal in the associated clause. Label each node of G with its corresponding literal in ϕ.

The edges of G connect all but two types of pairs of nodes in G. No edge is present between nodes in the same triple and no edge is present between two nodes with contradictory labels, as in x_2 and $\overline{x_2}$. The following figure illustrates this construction when $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.

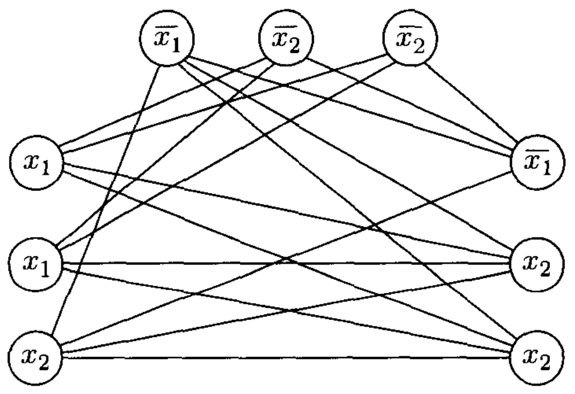

FIGURE 7.33
The graph that the reduction produces from
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

Now we demonstrate why this construction works. We show that ϕ is satisfiable iff G has a k-clique.

Suppose that ϕ has a satisfying assignment. In that satisfying assignment, at least one literal is true in every clause. In each triple of G, we select one node corresponding to a true literal in the satisfying assignment. If more than one literal is true in a particular clause, we choose one of the true literals arbitrarily. The nodes just selected form a k-clique. The number of nodes selected is k, because we chose one for each of the k triples. Each pair of selected nodes is joined by an edge because no pair fits one of the exceptions described previously. They could not be from the same triple because we selected only one node per triple. They could not have contradictory labels because the associated literals were both true in the satisfying assignment. Therefore G contains a k-clique.

Suppose that G has a k-clique. No two of the clique's nodes occur in the same triple because nodes in the same triple aren't connected by edges. Therefore each of the k triples contains exactly one of the k clique nodes. We assign truth values to the variables of ϕ so that each literal labeling a clique node is made true. Doing so is always possible because two nodes labeled in a contradictory way are not connected by an edge and hence both can't be in the clique. This assignment to the variables satisfies ϕ because each triple contains a clique node and hence each clause contains a literal that is assigned TRUE. Therefore ϕ is satisfiable.

Theorems 7.31 and 7.32 tell us that, if *CLIQUE* is solvable in polynomial time, so is *3SAT*. At first glance, this connection between these two problems appears quite remarkable because, superficially, they are rather different. But polynomial time reducibility allows us to link their complexities. Now we turn to a definition that will allow us similarly to link the complexities of an entire class of problems.

DEFINITION OF NP-COMPLETENESS

DEFINITION 7.34

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

THEOREM 7.35

If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.

PROOF This theorem follows directly from the definition of polynomial time reducibility.

THEOREM 7.36

If B is NP-complete and $B \leq_{\mathrm{P}} C$ for C in NP, then C is NP-complete.

PROOF We already know that C is in NP, so we must show that every A in NP is polynomial time reducible to C. Because B is NP-complete, every language in NP is polynomial time reducible to B, and B in turn is polynomial time reducible to C. Polynomial time reductions compose; that is, if A is polynomial time reducible to B and B is polynomial time reducible to C, then A is polynomial time reducible to C. Hence every language in NP is polynomial time reducible to C.

THE COOK–LEVIN THEOREM

Once we have one NP-complete problem, we may obtain others by polynomial time reduction from it. However, establishing the first NP-complete problem is more difficult. Now we do so by proving that *SAT* is NP-complete.

THEOREM 7.37

SAT is NP-complete.[2]

This theorem restates Theorem 7.27, the Cook–Levin theorem, in another form.

[2]An alternative proof of this theorem appears in Section 9.3 on page 351.

PROOF IDEA Showing that *SAT* is in NP is easy, and we do so shortly. The hard part of the proof is showing that any language in NP is polynomial time reducible to *SAT*.

To do so we construct a polynomial time reduction for each language A in NP to *SAT*. The reduction for A takes a string w and produces a Boolean formula ϕ that simulates the NP machine for A on input w. If the machine accepts, ϕ has a satisfying assignment that corresponds to the accepting computation. If the machine doesn't accept, no assignment satisfies ϕ. Therefore w is in A if and only if ϕ is satisfiable.

Actually constructing the reduction to work in this way is a conceptually simple task, though we must cope with many details. A Boolean formula may contain the Boolean operations AND, OR, and NOT, and these operations form the basis for the circuitry used in electronic computers. Hence the fact that we can design a Boolean formula to simulate a Turing machine isn't surprising. The details are in the implementation of this idea.

PROOF First, we show that *SAT* is in NP. A nondeterministic polynomial time machine can guess an assignment to a given formula ϕ and accept if the assignment satisfies ϕ.

Next, we take any language A in NP and show that A is polynomial time reducible to *SAT*. Let N be a nondeterministic Turing machine that decides A in n^k time for some constant k. (For convenience we actually assume that N runs in time $n^k - 3$, but only those readers interested in details should worry about this minor point.) The following notion helps to describe the reduction.

A ***tableau*** for N on w is an $n^k \times n^k$ table whose rows are the configurations of a branch of the computation of N on input w, as shown in the following figure.

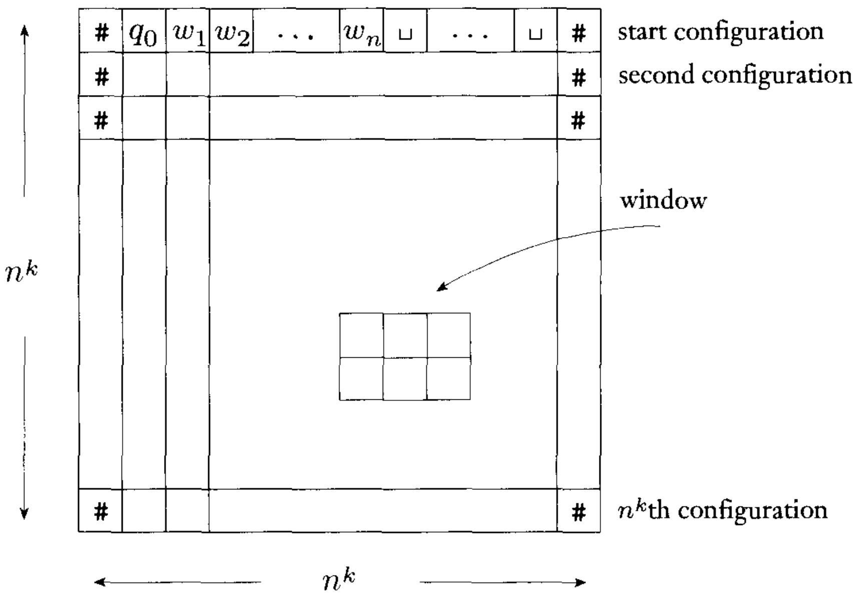

FIGURE 7.38
A tableau is an $n^k \times n^k$ table of configurations

For convenience later we assume that each configuration starts and ends with a # symbol, so the first and last columns of a tableau are all #s. The first row of the tableau is the starting configuration of N on w, and each row follows the previous one according to N's transition function. A tableau is ***accepting*** if any row of the tableau is an accepting configuration.

Every accepting tableau for N on w corresponds to an accepting computation branch of N on w. Thus the problem of determining whether N accepts w is equivalent to the problem of determining whether an accepting tableau for N on w exists.

Now we get to the description of the polynomial time reduction f from A to *SAT*. On input w, the reduction produces a formula ϕ. We begin by describing the variables of ϕ. Say that Q and Γ are the state set and tape alphabet of N. Let $C = Q \cup \Gamma \cup \{\#\}$. For each i and j between 1 and n^k and for each s in C we have a variable, $x_{i,j,s}$.

Each of the $(n^k)^2$ entries of a tableau is called a ***cell***. The cell in row i and column j is called $cell[i,j]$ and contains a symbol from C. We represent the contents of the cells with the variables of ϕ. If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s.

Now we design ϕ so that a satisfying assignment to the variables does correspond to an accepting tableau for N on w. The formula ϕ is the AND of four parts $\phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$. We describe each part in turn.

As we mentioned previously, turning variable $x_{i,j,s}$ on corresponds to placing symbol s in $cell[i,j]$. The first thing we must guarantee in order to obtain a correspondence between an assignment and a tableau is that the assignment turns on exactly one variable for each cell. Formula ϕ_{cell} ensures this requirement by expressing it in terms of Boolean operations:

$$\phi_{\text{cell}} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right].$$

The symbols $\bigwedge$ and $\bigvee$ stand for iterated AND and OR. For example, the expression in the preceding formula

$$\bigvee_{s \in C} x_{i,j,s}$$

is shorthand for

$$x_{i,j,s_1} \vee x_{i,j,s_2} \vee \cdots \vee x_{i,j,s_l}$$

where $C = \{s_1, s_2, \ldots, s_l\}$. Hence ϕ_{cell} is actually a large expression that contains a fragment for each cell in the tableau because i and j range from 1 to n^k. The first part of each fragment says that at least one variable is turned on in the corresponding cell. The second part of each fragment says that no more than one variable is turned on (literally, it says that in each pair of variables, at least one is turned off) in the corresponding cell. These fragments are connected by $\wedge$ operations.

The first part of ϕ_{cell} inside the brackets stipulates that at least one variable that is associated to each cell is on, whereas the second part stipulates that no more than one variable is on for each cell. Any assignment to the variables that satisfies ϕ (and therefore ϕ_{cell}) must have exactly one variable on for every cell. Thus any satisfying assignment specifies one symbol in each cell of the table. Parts ϕ_{start}, ϕ_{move}, and ϕ_{accept} ensure that these symbols actually correspond to an accepting tableau as follows.

Formula ϕ_{start} ensures that the first row of the table is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on:

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} .\end{aligned}$$

Formula ϕ_{accept} guarantees that an accepting configuration occurs in the tableau. It ensures that q_{accept}, the symbol for the accept state, appears in one of the cells of the tableau, by stipulating that one of the corresponding variables is on:

$$\phi_{\text{accept}} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{\text{accept}}} .$$

Finally, formula ϕ_{move} guarantees that each row of the table corresponds to a configuration that legally follows the preceding row's configuration according to N's rules. It does so by ensuring that each 2×3 window of cells is legal. We say that a 2×3 window is ***legal*** if that window does not violate the actions specified by N's transition function. In other words, a window is legal if it might appear when one configuration correctly follows another.[3]

For example, say that a, b, and c are members of the tape alphabet and q_1 and q_2 are states of N. Assume that, when in state q_1 with the head reading an a, N writes a b, stays in state q_1 and moves right, and that when in state q_1 with the head reading a b, N nondeterministically either

1. writes a c, enters q_2 and moves to the left, or
2. writes an a, enters q_2 and moves to the right.

Expressed formally, $\delta(q_1, \text{a}) = \{(q_1,\text{b},\text{R})\}$ and $\delta(q_1, \text{b}) = \{(q_2,\text{c},\text{L}), (q_2,\text{a},\text{R})\}$. Examples of legal windows for this machine are shown in Figure 7.39.

[3]We could give a precise definition of ***legal window*** here, in terms of the transition function. But doing so is quite tedious and would distract us from the main thrust of the proof argument. Anyone desiring more precision should refer to the related analysis in the proof of Theorem 5.15, the undecidability of the Post Correspondence Problem.

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

FIGURE 7.39
Examples of legal windows

In Figure 7.39, windows (a) and (b) are legal because the transition function allows N to move in the indicated way. Window (c) is legal because, with q_1 appearing on the right side of the top row, we don't know what symbol the head is over. That symbol could be an a, and q_1 might change it to a b and move to the right. That possibility would give rise to this window, so it doesn't violate N's rules. Window (d) is obviously legal because the top and bottom are identical, which would occur if the head weren't adjacent to the location of the window. Note that # may appear on the left or right of both the top and bottom rows in a legal window. Window (e) is legal because state q_1 reading a b might have been immediately to the right of the top row, and it would then have moved to the left in state q_2 to appear on the right-hand end of the bottom row. Finally, window (f) is legal because state q_1 might have been immediately to the left of the top row and it might have changed the b to a c and moved to the left.

The windows shown in the following figure aren't legal for machine N.

(a)

a	b	a
a	a	a

(b)

a	q_1	b
q_1	a	a

(c)

b	q_1	b
q_2	b	q_2

FIGURE 7.40
Examples of illegal windows

In window (a) the central symbol in the top row can't change because a state wasn't adjacent to it. Window (b) isn't legal because the transition function specifies that the b gets changed to a c but not to an a. Window (c) isn't legal because two states appear in the bottom row.

CLAIM 7.41

If the top row of the table is the start configuration and every window in the table is legal, each row of the table is a configuration that legally follows the preceding one.

We prove this claim by considering any two adjacent configurations in the table, called the upper configuration and the lower configuration. In the upper configuration, every cell that isn't adjacent to a state symbol and that doesn't contain the boundary symbol #, is the center top cell in a window whose top row contains no states. Therefore that symbol must appear unchanged in the center bottom of the window. Hence it appears in the same position in the bottom configuration.

The window containing the state symbol in the center top cell guarantees that the corresponding three positions are updated consistently with the transition function. Therefore, if the upper configuration is a legal configuration, so is the lower configuration, and the lower one follows the upper one according to N's rules. Note that this proof, though straightforward, depends crucially on our choice of a 2×3 window size, as Exercise 7.39 shows.

Now we return to the construction of ϕ_{move}. It stipulates that all the windows in the tableau are legal. Each window contains six cells, which may be set in a fixed number of ways to yield a legal window. Formula ϕ_{move} says that the settings of those six cells must be one of these ways, or

$$\phi_{\text{move}} = \bigwedge_{1<i\leq n^k,\ 1<j<n^k} \left(\text{the } (i,j) \text{ window is legal}\right)$$

We replace the text "the (i,j) window is legal" in this formula with the following formula. We write the contents of six cells of a window as $a_1, \ldots, a_6$.

$$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is a legal window}}} \left(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}\right)$$

Next we analyze the complexity of the reduction to show that it operates in polynomial time. To do so we examine the size of ϕ. First, we estimate the number of variables it has. Recall that the tableau is an $n^k \times n^k$ table, so it contains n^{2k} cells. Each cell has l variables associated with it, where l is the number of symbols in C. Because l depends only on the TM N and not on the length of the input n, the total number of variables is $O(n^{2k})$.

We estimate the size of each of the parts of ϕ. Formula ϕ_{cell} contains a fixed-size fragment of the formula for each cell of the tableau, so its size is $O(n^{2k})$. Formula ϕ_{start} has a fragment for each cell in the top row, so its size is $O(n^k)$. Formulas ϕ_{move} and ϕ_{accept} each contain a fixed-size fragment of the formula for each cell of the tableau, so their size is $O(n^{2k})$. Thus ϕ's total size is $O(n^{2k})$. That bound is sufficient for our purposes because it shows that the the size of ϕ is polynomial in n. If it were more than polynomial, the reduction wouldn't have any chance of generating it in polynomial time. (Actually our estimates are low by a factor of $O(\log n)$ because each variable has indices that can range up to n^k and so may require $O(\log n)$ symbols to write into the formula, but this additional factor doesn't change the polynomiality of the result.)

To see that we can generate the formula in polynomial time, observe its highly repetitive nature. Each component of the formula is composed of many nearly identical fragments, which differ only at the indices in a simple way. Therefore we may easily construct a reduction that produces ϕ in polynomial time from the input w.

Thus we have concluded the proof of the Cook–Levin theorem, showing that *SAT* is NP-complete. Showing the NP-completeness of other languages generally doesn't require such a lengthy proof. Instead NP-completeness can be proved with a polynomial time reduction from a language that is already known to be NP-complete. We can use *SAT* for this purpose, but using *3SAT*, the special case of *SAT* that we defined on page 274, is usually easier. Recall that the formulas in *3SAT* are in conjunctive normal form (cnf) with three literals per clause. First, we must show that *3SAT* itself is NP-complete. We prove this assertion as a corollary to Theorem 7.37.

COROLLARY 7.42

3SAT is NP-complete.

PROOF Obviously *3SAT* is in NP, so we only need to prove that all languages in NP reduce to *3SAT* in polynomial time. One way to do so is by showing that *SAT* polynomial time reduces to *3SAT*. Instead, we modify the proof of Theorem 7.37 so that it directly produces a formula in conjunctive normal form with three literals per clause.

Theorem 7.37 produces a formula that is already almost in conjunctive normal form. Formula ϕ_{cell} is a big AND of subformulas, each of which contains a big OR and a big AND of ORs. Thus ϕ_{cell} is an AND of clauses and so is already in cnf. Formula ϕ_{start} is a big AND of variables. Taking each of these variables to be a clause of size 1 we see that ϕ_{start} is in cnf. Formula ϕ_{accept} is a big OR of variables and is thus a single clause. Formula ϕ_{move} is the only one that isn't already in cnf, but we may easily convert it into a formula that is in cnf as follows.

Recall that ϕ_{move} is a big AND of subformulas, each of which is an OR of ANDs that describes all possible legal windows. The distributive laws, as described in Chapter 0, state that we can replace an OR of ANDs with an equivalent AND of ORs. Doing so may significantly increase the size of each subformula, but it can only increase the total size of ϕ_{move} by a constant factor because the size of each subformula depends only on N. The result is a formula that is in conjunctive normal form.

Now that we have written the formula in cnf, we convert it to one with three literals per clause. In each clause that currently has one or two literals, we replicate one of the literals until the total number is three. In each clause that has more than three literals, we split it into several clauses and add additional variables to preserve the satisfiability or nonsatisfiability of the original.

For example, we replace clause $(a_1 \vee a_2 \vee a_3 \vee a_4)$, wherein each a_i is a literal, with the two-clause expression $(a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$, wherein z is a new variable. If some setting of the a_i's satisfies the original clause, we can find some setting of z so that the two new clauses are satisfied. In general, if the clause contains l literals,

$$(a_1 \vee a_2 \vee \cdots \vee a_l),$$

we can replace it with the $l-2$ clauses

$$(a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \wedge \cdots \wedge (\overline{z_{l-3}} \vee a_{l-1} \vee a_l).$$

We may easily verify that the new formula is satisfiable iff the original formula was, so the proof is complete.

7.5 ADDITIONAL NP-COMPLETE PROBLEMS

The phenomenon of NP-completeness is widespread. NP-complete problems appear in many fields. For reasons that are not well understood, most naturally occurring NP-problems are known either to be in P or to be NP-complete. If you seek a polynomial time algorithm for a new NP-problem, spending part of your effort attempting to prove it NP-complete is sensible because doing so may prevent you from working to find a polynomial time algorithm that doesn't exist.

In this section we present additional theorems showing that various languages are NP-complete. These theorems provide examples of the techniques that are used in proofs of this kind. Our general strategy is to exhibit a polynomial time reduction from *3SAT* to the language in question, though we sometimes reduce from other NP-complete languages when that is more convenient.

When constructing a polynomial time reduction from *3SAT* to a language, we look for structures in that language that can simulate the variables and clauses in Boolean formulas. Such structures are sometimes called ***gadgets***. For example, in the reduction from *3SAT* to *CLIQUE* presented in Theorem 7.32, individual nodes simulate variables and triples of nodes simulate clauses. An individual node may or may not be a member of the clique, which corresponds to a variable that may or may not be true in a satisfying assignment. Each clause must contain a literal that is assigned TRUE and that corresponds to the way each triple must contain a node in the clique if the target size is to be reached. The following corollary to Theorem 7.32 states that *CLIQUE* is NP-complete.

COROLLARY 7.43

CLIQUE is NP-complete.