above, we can express this as the set $\mathcal{P}$ of all problems X for which there exists an algorithm A with a polynomial running time that solves X.

Efficient Certification

Now, how should we formalize the idea that a solution to a problem can be *checked* efficiently, independently of whether it can be solved efficiently? A "checking algorithm" for a problem X has a different structure from an algorithm that actually seeks to solve the problem; in order to "check" a solution, we need the input string s, as well as a separate "certificate" string t that contains the evidence that s is a "yes" instance of X.

Thus we say that B is an *efficient certifier* for a problem X if the following properties hold.

- B is a polynomial-time algorithm that takes two input arguments s and t.
- There is a polynomial function p so that for every string s, we have $s \in X$ if and only if there exists a string t such that $|t| \leq p(|s|)$ and $B(s,t) = \texttt{yes}$.

It takes some time to really think through what this definition is saying. One should view an efficient certifier as approaching a problem X from a "managerial" point of view. It will not actually try to decide whether an input s belongs to X on its own. Rather, it is willing to efficiently evaluate proposed "proofs" t that s belongs to X—provided they are not too long—and it is a correct algorithm in the weak sense that s belongs to X if and only if there exists a proof that will convince it.

An efficient certifier B can be used as the core component of a "brute-force" algorithm for a problem X: On an input s, try all strings t of length $\leq p(|s|)$, and see if $B(s,t) = \texttt{yes}$ for any of these strings. But the existence of B does not provide us with any clear way to design an efficient algorithm that actually solves X; after all, it is still up to us to *find* a string t that will cause $B(s,t)$ to say "yes," and there are exponentially many possibilities for t.

NP: A Class of Problems

We define $\mathcal{NP}$ to be the set of all problems for which there exists an efficient certifier.[1] Here is one thing we can observe immediately.

(8.10) $\mathcal{P} \subseteq \mathcal{NP}$.

[1] The act of searching for a string t that will cause an efficient certifier to accept the input s is often viewed as a *nondeterministic search* over the space of possible proofs t; for this reason, $\mathcal{NP}$ was named as an acronym for "nondeterministic polynomial time."

Proof. Consider a problem $X \in \mathcal{P}$; this means that there is a polynomial-time algorithm A that solves X. To show that $X \in \mathcal{NP}$, we must show that there is an efficient certifier B for X.

This is very easy; we design B as follows. When presented with the input pair (s, t), the certifier B simply returns the value $A(s)$. (Think of B as a very "hands-on" manager that ignores the proposed proof t and simply solves the problem on its own.) Why is B an efficient certifier for X? Clearly it has polynomial running time, since A does. If a string $s \in X$, then for every t of length at most $p(|s|)$, we have $B(s, t) = \texttt{yes}$. On the other hand, if $s \notin X$, then for every t of length at most $p(|s|)$, we have $B(s, t) = \texttt{no}$. ■

We can easily check that the problems introduced in the first two sections belong to $\mathcal{NP}$: it is a matter of determining how an efficient certifier for each of them will make use of a "certificate" string t. For example:

- For the 3-Satisfiability Problem, the certificate t is an assignment of truth values to the variables; the certifier B evaluates the given set of clauses with respect to this assignment.
- For the Independent Set Problem, the certificate t is the identity of a set of at least k vertices; the certifier B checks that, for these vertices, no edge joins any pair of them.
- For the Set Cover Problem, the certificate t is a list of k sets from the given collection; the certifier checks that the union of these sets is equal to the underlying set U.

Yet we cannot prove that any of these problems require more than polynomial time to solve. Indeed, we cannot prove that there is any problem in $\mathcal{NP}$ that does not belong to $\mathcal{P}$. So in place of a concrete theorem, we can only ask a question:

(8.11) *Is there a problem in $\mathcal{NP}$ that does not belong to $\mathcal{P}$? Does $\mathcal{P} = \mathcal{NP}$?*

The question of whether $\mathcal{P} = \mathcal{NP}$ is fundamental in the area of algorithms, and it is one of the most famous problems in computer science. The general belief is that $\mathcal{P} \neq \mathcal{NP}$—and this is taken as a working hypothesis throughout the field—but there is not a lot of hard technical evidence for it. It is more based on the sense that $\mathcal{P} = \mathcal{NP}$ would be too amazing to be true. How could there be a general transformation from the task of *checking* a solution to the much harder task of actually *finding* a solution? How could there be a general means for designing efficient algorithms, powerful enough to handle all these hard problems, that we have somehow failed to discover? More generally, a huge amount of effort has gone into failed attempts at designing polynomial-time algorithms for hard problems in $\mathcal{NP}$; perhaps the most natural explanation

for this consistent failure is that these problems simply cannot be solved in polynomial time.

8.4 NP-Complete Problems

In the absence of progress on the $\mathcal{P} = \mathcal{NP}$ question, people have turned to a related but more approachable question: What are the hardest problems in $\mathcal{NP}$? Polynomial-time reducibility gives us a way of addressing this question and gaining insight into the structure of $\mathcal{NP}$.

Arguably the most natural way to define a "hardest" problem X is via the following two properties: (i) $X \in \mathcal{NP}$; and (ii) for all $Y \in \mathcal{NP}$, $Y \leq_P X$. In other words, we require that every problem in $\mathcal{NP}$ can be reduced to X. We will call such an X an *NP-complete* problem.

The following fact helps to further reinforce our use of the term *hardest*.

(8.12) *Suppose X is an NP-complete problem. Then X is solvable in polynomial time if and only if $\mathcal{P} = \mathcal{NP}$.*

Proof. Clearly, if $\mathcal{P} = \mathcal{NP}$, then X can be solved in polynomial time since it belongs to $\mathcal{NP}$. Conversely, suppose that X can be solved in polynomial time. If Y is any other problem in $\mathcal{NP}$, then $Y \leq_P X$, and so by (8.1), it follows that Y can be solved in polynomial time. Hence $\mathcal{NP} \subseteq \mathcal{P}$; combined with (8.10), we have the desired conclusion. ■

A crucial consequence of (8.12) is the following: If there is *any* problem in $\mathcal{NP}$ that cannot be solved in polynomial time, then no NP-complete problem can be solved in polynomial time.

Circuit Satisfiability: A First NP-Complete Problem

Our definition of NP-completeness has some very nice properties. But before we get too carried away in thinking about this notion, we should stop to notice something: it is not at all obvious that NP-complete problems should even *exist*. Why couldn't there exist two incomparable problems X' and X'', so that there is no $X \in \mathcal{NP}$ with the property that $X' \leq_P X$ and $X'' \leq_P X$? Why couldn't there exist an infinite sequence of problems $X_1, X_2, X_3, \ldots$ in $\mathcal{NP}$, each strictly harder than the previous one? To prove a problem is NP-complete, one must show how it could encode *any* problem in $\mathcal{NP}$. This is a much trickier matter than what we encountered in Sections 8.1 and 8.2, where we sought to encode specific, individual problems in terms of others.

In 1971, Cook and Levin independently showed how to do this for very natural problems in $\mathcal{NP}$. Maybe the most natural problem choice for a first NP-complete problem is the following *Circuit Satisfiability Problem.*

To specify this problem, we need to make precise what we mean by a *circuit*. Consider the standard Boolean operators that we used to define the Satisfiability Problem: $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT). Our definition of a circuit is designed to represent a physical circuit built out of gates that implement these operators. Thus we define a circuit K to be a labeled, directed acyclic graph such as the one shown in the example of Figure 8.4.

- The *sources* in K (the nodes with no incoming edges) are labeled either with one of the constants 0 or 1, or with the name of a distinct variable. The nodes of the latter type will be referred to as the *inputs* to the circuit.
- Every other node is labeled with one of the Boolean operators $\wedge$, $\vee$, or $\neg$; nodes labeled with $\wedge$ or $\vee$ will have two incoming edges, and nodes labeled with $\neg$ will have one incoming edge.
- There is a single node with no outgoing edges, and it will represent the *output:* the result that is computed by the circuit.

A circuit computes a function of its inputs in the following natural way. We imagine the edges as "wires" that carry the 0/1 value at the node they emanate from. Each node v other than the sources will take the values on its incoming edge(s) and apply the Boolean operator that labels it. The result of this $\wedge$, $\vee$, or $\neg$ operation will be passed along the edge(s) leaving v. The overall value computed by the circuit will be the value computed at the output node.

For example, consider the circuit in Figure 8.4. The leftmost two sources are preassigned the values 1 and 0, and the next three sources constitute the

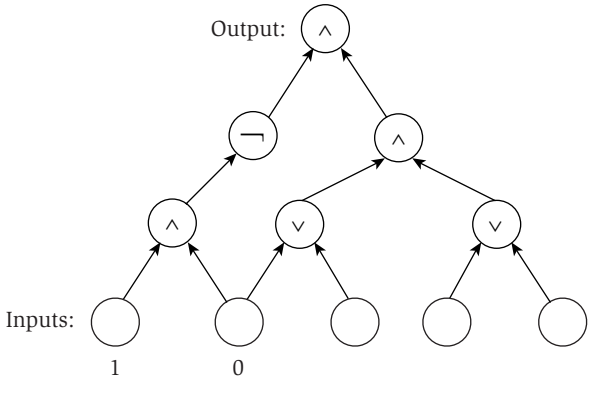

Figure 8.4 A circuit with three inputs, two additional sources that have assigned truth values, and one output.

inputs. If the inputs are assigned the values 1, 0, 1 from left to right, then we get values 0, 1, 1 for the gates in the second row, values 1, 1 for the gates in the third row, and the value 1 for the output.

Now, the Circuit Satisfiability Problem is the following. We are given a circuit as input, and we need to decide whether there is an assignment of values to the inputs that causes the output to take the value 1. (If so, we will say that the given circuit is *satisfiable*, and a *satisfying assignment* is one that results in an output of 1.) In our example, we have just seen—via the assignment 1, 0, 1 to the inputs—that the circuit in Figure 8.4 is satisfiable.

We can view the theorem of Cook and Levin as saying the following.

(8.13) Circuit Satisfiability *is NP-complete.*

As discussed above, the proof of (8.13) requires that we consider an arbitrary problem X in $\mathcal{NP}$, and show that $X \leq_P$ Circuit Satisfiability. We won't describe the proof of (8.13) in full detail, but it is actually not so hard to follow the basic idea that underlies it. We use the fact that any algorithm that takes a fixed number n of bits as input and produces a yes/no answer can be represented by a circuit of the type we have just defined: This circuit is equivalent to the algorithm in the sense that its output is 1 on precisely the inputs for which the algorithm outputs `yes`. Moreover, if the algorithm takes a number of steps that is polynomial in n, then the circuit has polynomial size. This transformation from an algorithm to a circuit is the part of the proof of (8.13) that we won't go into here, though it is quite natural given the fact that algorithms implemented on physical computers can be reduced to their operations on an underlying set of $\wedge$, $\vee$, and $\neg$ gates. (Note that fixing the number of input bits is important, since it reflects a basic distinction between algorithms and circuits: an algorithm typically has no trouble dealing with different inputs of varying lengths, but a circuit is structurally hard-coded with the size of the input.)

How should we use this relationship between algorithms and circuits? We are trying to show that $X \leq_P$ Circuit Satisfiability—that is, given an input s, we want to decide whether $s \in X$ using a black box that can solve instances of Circuit Satisfiability. Now, all we know about X is that it has an efficient certifier $B(\cdot, \cdot)$. So to determine whether $s \in X$, for some specific input s of length n, we need to answer the following question: Is there a t of length $p(n)$ so that $B(s, t) =$ `yes`?

We will answer this question by appealing to a black box for Circuit Satisfiability as follows. Since we only care about the answer for a specific input s, we view $B(\cdot, \cdot)$ as an algorithm on $n + p(n)$ bits (the input s and the

certificate t), and we convert it to a polynomial-size circuit K with $n + p(n)$ sources. The first n sources will be hard-coded with the values of the bits in s, and the remaining $p(n)$ sources will be labeled with variables representing the bits of t; these latter sources will be the inputs to K.

Now we simply observe that $s \in X$ if and only if there is a way to set the input bits to K so that the circuit produces an output of 1—in other words, if and only if K is satisfiable. This establishes that $X \leq_P$ *Circuit Satisfiability*, and completes the proof of (8.13).

An Example To get a better sense for what's going on in the proof of (8.13), we consider a simple, concrete example. Suppose we have the following problem.

Given a graph G, does it contain a two-node independent set?

Note that this problem belongs to $\mathcal{NP}$. Let's see how an instance of this problem can be solved by constructing an equivalent instance of Circuit Satisfiability.

Following the proof outline above, we first consider an efficient certifier for this problem. The input s is a graph on n nodes, which will be specified by $\binom{n}{2}$ bits: For each pair of nodes, there will be a bit saying whether there is an edge joining this pair. The certificate t can be specified by n bits: For each node, there will be a bit saying whether this node belongs to the proposed independent set. The efficient certifier now needs to check two things: that at least two of the bits in t are set to 1, and that no two bits in t are both set to 1 if they form the two ends of an edge (as determined by the corresponding bit in s).

Now, for the specific input length n corresponding to the s that we are interested in, we construct an equivalent circuit K. Suppose, for example, that we are interested in deciding the answer to this problem for a graph G on the three nodes u, v, w, in which v is joined to both u and w. This means that we are concerned with an input of length $n = 3$. Figure 8.5 shows a circuit that is equivalent to an efficient certifier for our problem on arbitrary three-node graphs. (Essentially, the right-hand side of the circuit checks that at least two nodes have been selected, and the left-hand side checks that we haven't selected both ends of any edge.) We encode the edges of G as constants in the first three sources, and we leave the remaining three sources (representing the choice of nodes to put in the independent set) as variables. Now observe that this instance of Circuit Satisfiability is satisfiable, by the assignment $1, 0, 1$ to the inputs. This corresponds to choosing nodes u and w, which indeed form a two-node independent set in our three-node graph G.

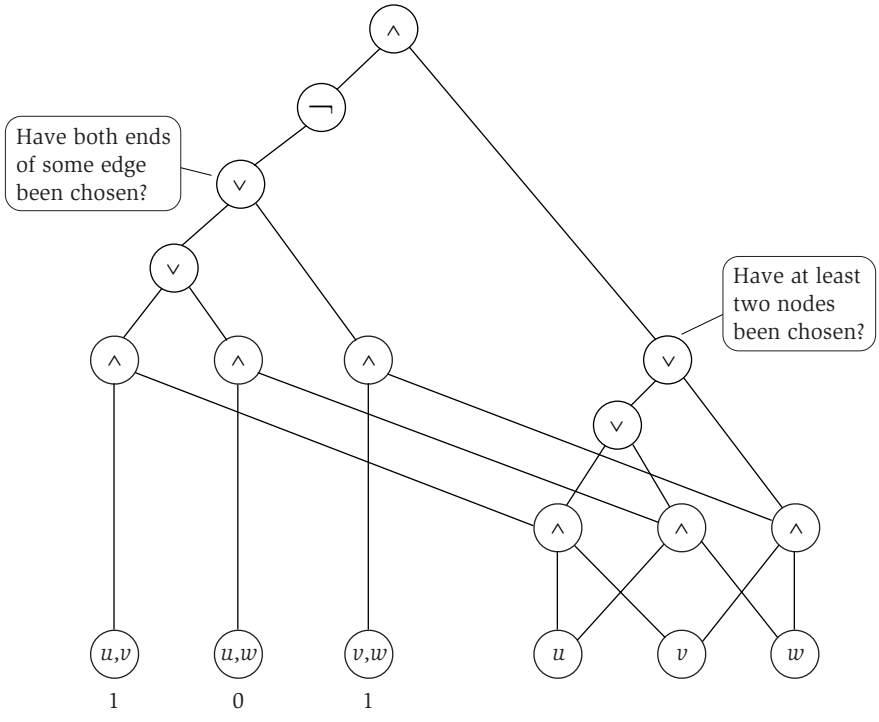

Figure 8.5 A circuit to verify whether a 3-node graph contains a 2-node independent set.

Proving Further Problems NP-Complete

Statement (8.13) opens the door to a much fuller understanding of hard problems in $\mathcal{NP}$: Once we have our hands on a first NP-complete problem, we can discover many more via the following simple observation.

(8.14) *If Y is an NP-complete problem, and X is a problem in $\mathcal{NP}$ with the property that $Y \leq_P X$, then X is NP-complete.*

Proof. Since $X \in \mathcal{NP}$, we need only verify property (ii) of the definition. So let Z be any problem in $\mathcal{NP}$. We have $Z \leq_P Y$, by the NP-completeness of Y, and $Y \leq_P X$ by assumption. By (8.9), it follows that $Z \leq_P X$. ■

So while proving (8.13) required the hard work of considering any possible problem in $\mathcal{NP}$, proving further problems NP-complete only requires a reduction from a single problem already known to be NP-complete, thanks to (8.14).

In earlier sections, we have seen a number of reductions among some basic hard problems. To establish their NP-completeness, we need to connect Circuit Satisfiability to this set of problems. The easiest way to do this is by relating it to the problem it most closely resembles, 3-Satisfiability.

(8.15) 3-Satisfiability *is NP-complete.*

Proof. Clearly 3-Satisfiability is in $\mathcal{NP}$, since we can verify in polynomial time that a proposed truth assignment satisfies the given set of clauses. We will prove that it is NP-complete via the reduction Circuit Satisfiability $\leq_P$ 3-SAT.

Given an arbitrary instance of Circuit Satisfiability, we will first construct an equivalent instance of SAT in which each clause contains *at most* three variables. Then we will convert this SAT instance to an equivalent one in which each clause has *exactly* three variables. This last collection of clauses will thus be an instance of 3-SAT, and hence will complete the reduction.

So consider an arbitrary circuit K. We associate a variable x_v with each node v of the circuit, to encode the truth value that the circuit holds at that node. Now we will define the clauses of the SAT problem. First we need to encode the requirement that the circuit computes values correctly at each gate from the input values. There will be three cases depending on the three types of gates.

- If node v is labeled with $\neg$, and its only entering edge is from node u, then we need to have $x_v = \overline{x_u}$. We guarantee this by adding two clauses $(x_v \vee x_u)$, and $(\overline{x_v} \vee \overline{x_u})$.
- If node v is labeled with $\vee$, and its two entering edges are from nodes u and w, we need to have $x_v = x_u \vee x_w$. We guarantee this by adding the following clauses: $(x_v \vee \overline{x_u})$, $(x_v \vee \overline{x_w})$, and $(\overline{x_v} \vee x_u \vee x_w)$.
- If node v is labeled with $\wedge$, and its two entering edges are from nodes u and w, we need to have $x_v = x_u \wedge x_w$. We guarantee this by adding the following clauses: $(\overline{x_v} \vee x_u)$, $(\overline{x_v} \vee x_w)$, and $(x_v \vee \overline{x_u} \vee \overline{x_w})$.

Finally, we need to guarantee that the constants at the sources have their specified values, and that the output evaluates to 1. Thus, for a source v that has been labeled with a constant value, we add a clause with the single variable x_v or $\overline{x_v}$, which forces x_v to take the designated value. For the output node o, we add the single-variable clause x_o, which requires that o take the value 1. This concludes the construction.

It is not hard to show that the SAT instance we just constructed is equivalent to the given instance of Circuit Satisfiability. To show the equivalence, we need to argue two things. First suppose that the given circuit K is satisfiable. The satisfying assignment to the circuit inputs can be propagated to create

values at all nodes in K (as we did in the example of Figure 8.4). This set of values clearly satisfies the SAT instance we constructed.

To argue the other direction, we suppose that the SAT instance we constructed is satisfiable. Consider a satisfying assignment for this instance, and look at the values of the variables corresponding to the circuit K's inputs. We claim that these values constitute a satisfying assignment for the circuit K. To see this, simply note that the SAT clauses ensure that the values assigned to all nodes of K are the same as what the circuit computes for these nodes. In particular, a value of 1 will be assigned to the output, and so the assignment to inputs satisfies K.

Thus we have shown how to create a SAT instance that is equivalent to the Circuit Satisfiability Problem. But we are not quite done, since our goal was to create an instance of 3-SAT, which requires that all clauses have length exactly 3—in the instance we constructed, some clauses have lengths of 1 or 2. So to finish the proof, we need to convert this instance of SAT to an equivalent instance in which each clause has exactly three variables.

To do this, we create four new variables: z_1, z_2, z_3, z_4. The idea is to ensure that in any satisfying assignment, we have $z_1 = z_2 = 0$, and we do this by adding the clauses $(\overline{z_i} \vee z_3 \vee z_4)$, $(\overline{z_i} \vee \overline{z_3} \vee z_4)$, $(\overline{z_i} \vee z_3 \vee \overline{z_4})$, and $(\overline{z_i} \vee \overline{z_3} \vee \overline{z_4})$ for each of $i = 1$ and $i = 2$. Note that there is no way to satisfy all these clauses unless we set $z_1 = z_2 = 0$.

Now consider a clause in the SAT instance we constructed that has a single term t (where the term t can be either a variable or the negation of a variable). We replace each such term by the clause $(t \vee z_1 \vee z_2)$. Similarly, we replace each clause that has two terms, say, $(t \vee t')$, with the clause $(t \vee t' \vee z_1)$. The resulting 3-SAT formula is clearly equivalent to the SAT formula with at most three variables in each clause, and this finishes the proof. ■

Using this NP-completeness result, and the sequence of reductions

$$\text{3-SAT} \leq_P \text{Independent Set} \leq_P \text{Vertex Cover} \leq_P \text{Set Cover},$$

summarized earlier, we can use (8.14) to conclude the following.

(8.16) *All of the following problems are NP-complete:* Independent Set, Set Packing, Vertex Cover, *and* Set Cover.

Proof. Each of these problems has the property that it is in $\mathcal{NP}$ and that 3-SAT (and hence Circuit Satisfiability) can be reduced to it. ■

General Strategy for Proving New Problems NP-Complete

For most of the remainder of this chapter, we will take off in search of further NP-complete problems. In particular, we will discuss further genres of hard computational problems and prove that certain examples of these genres are NP-complete. As we suggested initially, there is a very practical motivation in doing this: since it is widely believed that $\mathcal{P} \neq \mathcal{NP}$, the discovery that a problem is NP-complete can be taken as a strong indication that it cannot be solved in polynomial time.

Given a new problem X, here is the basic strategy for proving it is NP-complete.

1. Prove that $X \in \mathcal{NP}$.
2. Choose a problem Y that is known to be NP-complete.
3. Prove that $Y \leq_P X$.

We noticed earlier that most of our reductions $Y \leq_P X$ consist of transforming a given instance of Y into a *single* instance of X with the same answer. This is a particular way of using a black box to solve X; in particular, it requires only a single invocation of the black box. When we use this style of reduction, we can refine the strategy above to the following outline of an NP-completeness proof.

1. Prove that $X \in \mathcal{NP}$.
2. Choose a problem Y that is known to be NP-complete.
3. Consider an arbitrary instance s_Y of problem Y, and show how to construct, in polynomial time, an instance s_X of problem X that satisfies the following properties:
   (a) If s_Y is a "yes" instance of Y, then s_X is a "yes" instance of X.
   (b) If s_X is a "yes" instance of X, then s_Y is a "yes" instance of Y.
   In other words, this establishes that s_Y and s_X have the same answer.

There has been research aimed at understanding the distinction between polynomial-time reductions with this special structure—asking the black box a single question and using its answer verbatim—and the more general notion of polynomial-time reduction that can query the black box multiple times. (The more restricted type of reduction is known as a *Karp reduction*, while the more general type is known as a *Cook reduction* and also as a *polynomial-time Turing reduction*.) We will not be pursuing this distinction further here.

8.5 Sequencing Problems

Thus far we have seen problems that (like Independent Set and Vertex Cover) have involved searching over subsets of a collection of objects; we have also

seen problems that (like 3-SAT) have involved searching over 0/1 settings to a collection of variables. Another type of computationally hard problem involves searching over the set of all *permutations* of a collection of objects.

The Traveling Salesman Problem

Probably the most famous such sequencing problem is the *Traveling Salesman Problem*. Consider a salesman who must visit n cities labeled $v_1, v_2, \ldots, v_n$. The salesman starts in city v_1, his home, and wants to find a *tour*—an order in which to visit all the other cities and return home. His goal is to find a tour that causes him to travel as little total distance as possible.

To formalize this, we will take a very general notion of distance: for each ordered pair of cities (v_i, v_j), we will specify a nonnegative number $d(v_i, v_j)$ as the distance from v_i to v_j. We will not require the distance to be symmetric (so it may happen that $d(v_i, v_j) \neq d(v_j, v_i)$), nor will we require it to satisfy the triangle inequality (so it may happen that $d(v_i, v_j)$ plus $d(v_j, v_k)$ is actually less than the "direct" distance $d(v_i, v_k)$). The reason for this is to make our formulation as general as possible. Indeed, Traveling Salesman arises naturally in many applications where the points are not cities and the traveler is not a salesman. For example, people have used Traveling Salesman formulations for problems such as planning the most efficient motion of a robotic arm that drills holes in n points on the surface of a VLSI chip; or for serving I/O requests on a disk; or for sequencing the execution of n software modules to minimize the context-switching time.

Thus, given the set of distances, we ask: Order the cities into a *tour* $v_{i_1}, v_{i_2}, \ldots, v_{i_n}$, with $i_1 = 1$, so as to minimize the total distance $\sum_j d(v_{i_j}, v_{i_{j+1}}) + d(v_{i_n}, v_{i_1})$. The requirement $i_1 = 1$ simply "orients" the tour so that it starts at the home city, and the terms in the sum simply give the distance from each city on the tour to the next one. (The last term in the sum is the distance required for the salesman to return home at the end.)

Here is a decision version of the Traveling Salesman Problem.

Given a set of distances on n cities, and a bound D, is there a tour of length at most D?

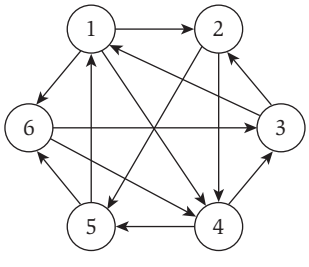

Figure 8.6 A directed graph containing a Hamiltonian cycle.

The Hamiltonian Cycle Problem

The Traveling Salesman Problem has a natural graph-based analogue, which forms one of the fundamental problems in graph theory. Given a directed graph $G = (V, E)$, we say that a cycle C in G is a *Hamiltonian cycle* if it visits each vertex exactly once. In other words, it constitutes a "tour" of all the vertices, with no repetitions. For example, the directed graph pictured in Figure 8.6 has

several Hamiltonian cycles; one visits the nodes in the order 1, 6, 4, 3, 2, 5, 1, while another visits the nodes in the order 1, 2, 4, 5, 6, 3, 1.

The Hamiltonian Cycle Problem is then simply the following:

Given a directed graph G, does it contain a Hamiltonian cycle?

Proving *Hamiltonian Cycle* is NP-Complete

We now show that both these problems are NP-complete. We do this by first establishing the NP-completeness of Hamiltonian Cycle, and then proceeding to reduce from Hamiltonian Cycle to Traveling Salesman.

(8.17) Hamiltonian Cycle *is NP-complete.*

Proof. We first show that Hamiltonian Cycle is in $\mathcal{NP}$. Given a directed graph $G = (V, E)$, a certificate that there is a solution would be the ordered list of the vertices on a Hamiltonian cycle. We could then check, in polynomial time, that this list of vertices does contain each vertex exactly once, and that each consecutive pair in the ordering is joined by an edge; this would establish that the ordering defines a Hamiltonian cycle.

We now show that 3-SAT $\leq_P$ Hamiltonian Cycle. Why are we reducing from 3-SAT? Essentially, faced with Hamiltonian Cycle, we really have no idea *what* to reduce from; it's sufficiently different from all the problems we've seen so far that there's no real basis for choosing. In such a situation, one strategy is to go back to 3-SAT, since its combinatorial structure is very basic. Of course, this strategy guarantees at least a certain level of complexity in the reduction, since we need to encode variables and clauses in the language of graphs.

So consider an arbitrary instance of 3-SAT, with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_k$. We must show how to solve it, given the ability to detect Hamiltonian cycles in directed graphs. As always, it helps to focus on the essential ingredients of 3-SAT: We can set the values of the variables however we want, and we are given three chances to satisfy each clause.

We begin by describing a graph that contains 2^n different Hamiltonian cycles that correspond very naturally to the 2^n possible truth assignments to the variables. After this, we will add nodes to model the constraints imposed by the clauses.

We construct n paths $P_1, \ldots, P_n$, where P_i consists of nodes $v_{i1}, v_{i2}, \ldots, v_{ib}$ for a quantity b that we take to be somewhat larger than the number of clauses k; say, $b = 3k + 3$. There are edges from v_{ij} to $v_{i,j+1}$ and in the other direction from $v_{i,j+1}$ to v_{ij}. Thus P_i can be traversed "left to right," from v_{i1} to v_{ib}, or "right to left," from v_{ib} to v_{i1}.