Chapter 8

NP and Computational Intractability

We now arrive at a major transition point in the book. Up until now, we've developed efficient algorithms for a wide range of problems and have even made some progress on informally categorizing the problems that admit efficient solutions—for example, problems expressible as minimum cuts in a graph, or problems that allow a dynamic programming formulation. But although we've often paused to take note of other problems that we don't see how to solve, we haven't yet made any attempt to actually quantify or characterize the range of problems that *can't be solved efficiently.*

Back when we were first laying out the fundamental definitions, we settled on polynomial time as our working notion of efficiency. One advantage of using a concrete definition like this, as we noted earlier, is that it gives us the opportunity to prove mathematically that certain problems cannot be solved by polynomial-time—and hence "efficient"—algorithms.

When people began investigating computational complexity in earnest, there was some initial progress in proving that certain *extremely hard* problems cannot be solved by efficient algorithms. But for many of the most fundamental discrete computational problems—arising in optimization, artificial intelligence, combinatorics, logic, and elsewhere—the question was too difficult to resolve, and it has remained open since then: We do not know of polynomial-time algorithms for these problems, and we cannot prove that no polynomial-time algorithm exists.

In the face of this formal ambiguity, which becomes increasingly hardened as years pass, people working in the study of complexity have made significant progress. A large class of problems in this "gray area" has been characterized, and it has been proved that they are equivalent in the following sense: a polynomial-time algorithm for any one of them would imply the existence of a

polynomial-time algorithm for all of them. These are the *NP-complete problems*, a name that will make more sense as we proceed a little further. There are literally thousands of NP-complete problems, arising in numerous areas, and the class seems to contain a large fraction of the fundamental problems whose complexity we can't resolve. So the formulation of NP-completeness, and the proof that all these problems are equivalent, is a powerful thing: it says that all these open questions are really a *single* open question, a single type of complexity that we don't yet fully understand.

From a pragmatic point of view, NP-completeness essentially means "computationally hard for all practical purposes, though we can't prove it." Discovering that a problem is NP-complete provides a compelling reason to stop searching for an efficient algorithm—you might as well search for an efficient algorithm for any of the famous computational problems already known to be NP-complete, for which many people have tried and failed to find efficient algorithms.

8.1 Polynomial-Time Reductions

Our plan is to explore the space of computationally hard problems, eventually arriving at a mathematical characterization of a large class of them. Our basic technique in this exploration is to compare the relative difficulty of different problems; we'd like to formally express statements like, "Problem X is at least as hard as problem Y." We will formalize this through the notion of *reduction*: we will show that a particular problem X is at least as hard as some other problem Y by arguing that, if we had a "black box" capable of solving X, then we could also solve Y. (In other words, X is powerful enough to let us solve Y.)

To make this precise, we add the assumption that X can be solved in polynomial time directly to our model of computation. Suppose we had a *black box* that could solve instances of a problem X; if we write down the input for an instance of X, then in a single step, the black box will return the correct answer. We can now ask the following question:

> *($*) Can arbitrary instances of problem Y be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to a black box that solves problem X?*

If the answer to this question is yes, then we write $Y \leq_p X$; we read this as "Y is polynomial-time reducible to X," or "X is at least as hard as Y (with respect to polynomial time)." Note that in this definition, we still pay for the time it takes to write down the input to the black box solving X, and to read the answer that the black box provides.

This formulation of reducibility is very natural. When we ask about reductions to a problem X, it is as though we've supplemented our computational model with a piece of specialized hardware that solves instances of X in a single step. We can now explore the question: How much extra power does this piece of hardware give us?

An important consequence of our definition of $\leq_P$ is the following. Suppose $Y \leq_P X$ and there actually *exists* a polynomial-time algorithm to solve X. Then our specialized black box for X is actually not so valuable; we can replace it with a polynomial-time algorithm for X. Consider what happens to our algorithm for problem Y that involved a polynomial number of steps plus a polynomial number of calls to the black box. It now becomes an algorithm that involves a polynomial number of steps, plus a polynomial number of calls to a subroutine that runs in polynomial time; in other words, it has become a polynomial-time algorithm. We have therefore proved the following fact.

(8.1) *Suppose $Y \leq_P X$. If X can be solved in polynomial time, then Y can be solved in polynomial time.*

We've made use of precisely this fact, implicitly, at a number of earlier points in the book. Recall that we solved the Bipartite Matching Problem using a polynomial amount of preprocessing plus the solution of a single instance of the Maximum-Flow Problem. Since the Maximum-Flow Problem can be solved in polynomial time, we concluded that Bipartite Matching could as well. Similarly, we solved the foreground/background Image Segmentation Problem using a polynomial amount of preprocessing plus the solution of a single instance of the Minimum-Cut Problem, with the same consequences. Both of these can be viewed as direct applications of (8.1). Indeed, (8.1) summarizes a great way to design polynomial-time algorithms for new problems: by reduction to a problem we already know how to solve in polynomial time.

In this chapter, however, we will be using (8.1) to establish the computational *intractability* of various problems. We will be engaged in the somewhat subtle activity of relating the tractability of problems even when we don't know how to solve *either* of them in polynomial time. For this purpose, we will really be using the contrapositive of (8.1), which is sufficiently valuable that we'll state it as a separate fact.

(8.2) *Suppose $Y \leq_P X$. If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.*

Statement (8.2) is transparently equivalent to (8.1), but it emphasizes our overall plan: If we have a problem Y that is known to be hard, and we show

that $Y \leq_P X$, then the hardness has "spread" to X; X must be hard or else it could be used to solve Y.

In reality, given that we don't actually know whether the problems we're studying can be solved in polynomial time or not, we'll be using $\leq_P$ to establish relative levels of difficulty among problems.

With this in mind, we now establish some reducibilities among an initial collection of fundamental hard problems.

A First Reduction: Independent Set and Vertex Cover

The Independent Set Problem, which we introduced as one of our five representative problems in Chapter 1, will serve as our first prototypical example of a hard problem. We don't know a polynomial-time algorithm for it, but we also don't know how to prove that none exists.

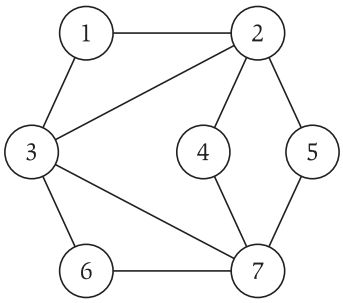

Figure 8.1 A graph whose largest independent set has size 4, and whose smallest vertex cover has size 3.

Let's review the formulation of Independent Set, because we're going to add one wrinkle to it. Recall that in a graph $G = (V, E)$, we say a set of nodes $S \subseteq V$ is *independent* if no two nodes in S are joined by an edge. It is easy to find small independent sets in a graph (for example, a single node forms an independent set); the hard part is to find a large independent set, since you need to build up a large collection of nodes without ever including two neighbors. For example, the set of nodes $\{3, 4, 5\}$ is an independent set of size 3 in the graph in Figure 8.1, while the set of nodes $\{1, 4, 5, 6\}$ is a larger independent set.

In Chapter 1, we posed the problem of finding the *largest* independent set in a graph G. For purposes of our current exploration in terms of reducibility, it will be much more convenient to work with problems that have yes/no answers only, and so we phrase Independent Set as follows.

> *Given a graph G and a number k, does G contain an independent set of size at least k?*

In fact, from the point of view of polynomial-time solvability, there is not a significant difference between the *optimization version* of the problem (find the maximum size of an independent set) and the *decision version* (decide, yes or no, whether G has an independent set of size at least a given k). Given a method to solve the optimization version, we automatically solve the decision version (for any k) as well. But there is also a slightly less obvious converse to this: If we can solve the decision version of Independent Set for every k, then we can also find a maximum independent set. For given a graph G on n nodes, we simply solve the decision version of Independent Set for each k; the largest k for which the answer is "yes" is the size of the largest independent set in G. (And using binary search, we need only solve the decision version

for $O(\log n)$ different values of k.) This simple equivalence between decision and optimization will also hold in the problems we discuss below.

Now, to illustrate our basic strategy for relating hard problems to one another, we consider another fundamental graph problem for which no efficient algorithm is known: *Vertex Cover*. Given a graph $G = (V, E)$, we say that a set of nodes $S \subseteq V$ is a *vertex cover* if every edge $e \in E$ has at least one end in S. Note the following fact about this use of terminology: In a vertex cover, the vertices do the "covering," and the edges are the objects being "covered." Now, it is easy to find large vertex covers in a graph (for example, the full vertex set is one); the hard part is to find small ones. We formulate the Vertex Cover Problem as follows.

> *Given a graph G and a number k, does G contain a vertex cover of size at most k?*

For example, in the graph in Figure 8.1, the set of nodes $\{1, 2, 6, 7\}$ is a vertex cover of size 4, while the set $\{2, 3, 7\}$ is a vertex cover of size 3.

We don't know how to solve either Independent Set or Vertex Cover in polynomial time; but what can we say about their relative difficulty? We now show that they are equivalently hard, by establishing that Independent Set $\leq_P$ Vertex Cover and also that Vertex Cover $\leq_P$ Independent Set. This will be a direct consequence of the following fact.

(8.3) *Let $G = (V, E)$ be a graph. Then S is an independent set if and only if its complement $V - S$ is a vertex cover.*

Proof. First, suppose that S is an independent set. Consider an arbitrary edge $e = (u, v)$. Since S is independent, it cannot be the case that both u and v are in S; so one of them must be in $V - S$. It follows that every edge has at least one end in $V - S$, and so $V - S$ is a vertex cover.

Conversely, suppose that $V - S$ is a vertex cover. Consider any two nodes u and v in S. If they were joined by edge e, then neither end of e would lie in $V - S$, contradicting our assumption that $V - S$ is a vertex cover. It follows that no two nodes in S are joined by an edge, and so S is an independent set. ■

Reductions in each direction between the two problems follow immediately from (8.3).

(8.4) Independent Set $\leq_P$ Vertex Cover.

Proof. If we have a black box to solve Vertex Cover, then we can decide whether G has an independent set of size at least k by asking the black box whether G has a vertex cover of size at most $n - k$. ■

(8.5) Vertex Cover $\leq_P$ Independent Set.

Proof. If we have a black box to solve Independent Set, then we can decide whether G has a vertex cover of size at most k by asking the black box whether G has an independent set of size at least $n - k$. ■

To sum up, this type of analysis illustrates our plan in general: although we don't know how to solve either Independent Set or Vertex Cover efficiently, (8.4) and (8.5) tell us how we could solve either given an efficient solution to the other, and hence these two facts establish the relative levels of difficulty of these problems.

We now pursue this strategy for a number of other problems.

Reducing to a More General Case: Vertex Cover to Set Cover

Independent Set and Vertex Cover represent two different genres of problems. Independent Set can be viewed as a *packing problem*: The goal is to "pack in" as many vertices as possible, subject to conflicts (the edges) that try to prevent one from doing this. Vertex Cover, on the other hand, can be viewed as a *covering problem*: The goal is to parsimoniously "cover" all the edges in the graph using as few vertices as possible.

Vertex Cover is a covering problem phrased specifically in the language of graphs; there is a more general covering problem, *Set Cover*, in which you seek to cover an arbitrary set of objects using a collection of smaller sets. We can phrase Set Cover as follows.

> *Given a set U of n elements, a collection $S_1, \ldots, S_m$ of subsets of U, and a number k, does there exist a collection of at most k of these sets whose union is equal to all of U?*

Imagine, for example, that we have m available pieces of software, and a set U of n *capabilities* that we would like our system to have. The i^{th} piece of software includes the set $S_i \subseteq U$ of capabilities. In the Set Cover Problem, we seek to include a small number of these pieces of software on our system, with the property that our system will then have all n capabilities.

Figure 8.2 shows a sample instance of the Set Cover Problem: The ten circles represent the elements of the underlying set U, and the seven ovals and polygons represent the sets $S_1, S_2, \ldots, S_7$. In this instance, there is a collection

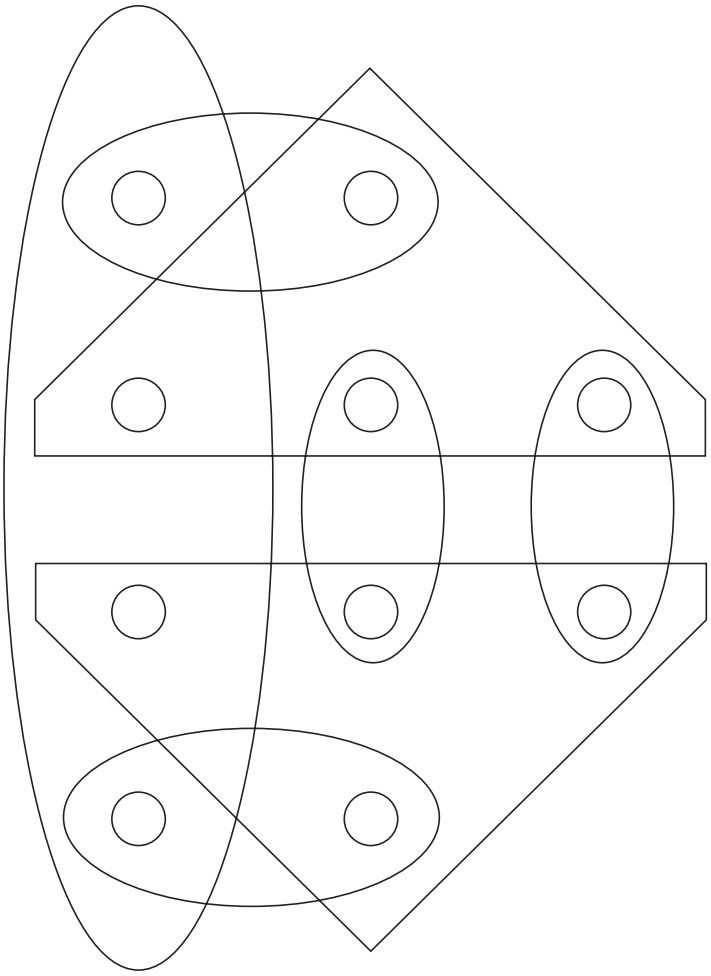

Figure 8.2 An instance of the Set Cover Problem.

of three of the sets whose union is equal to all of U: We can choose the tall thin oval on the left, together with the two polygons.

Intuitively, it feels like Vertex Cover is a special case of Set Cover: in the latter case, we are trying to cover an arbitrary set using arbitrary subsets, while in the former case, we are specifically trying to cover edges of a graph using sets of edges incident to vertices. In fact, we can show the following reduction.

(8.6) Vertex Cover $\leq_P$ Set Cover.

Proof. Suppose we have access to a black box that can solve Set Cover, and consider an arbitrary instance of Vertex Cover, specified by a graph $G = (V, E)$ and a number k. How can we use the black box to help us?

Our goal is to cover the edges in E, so we formulate an instance of Set Cover in which the ground set U is equal to E. Each time we pick a vertex in the Vertex Cover Problem, we cover all the edges incident to it; thus, for each vertex $i \in V$, we add a set $S_i \subseteq U$ to our Set Cover instance, consisting of all the edges in G incident to i.

We now claim that U can be covered with at most k of the sets $S_1, \ldots, S_n$ if and only if G has a vertex cover of size at most k. This can be proved very easily. For if $S_{i_1}, \ldots, S_{i_\ell}$ are $\ell \leq k$ sets that cover U, then every edge in G is incident to one of the vertices $i_1, \ldots, i_\ell$, and so the set $\{i_1, \ldots, i_\ell\}$ is a vertex cover in G of size $\ell \leq k$. Conversely, if $\{i_1, \ldots, i_\ell\}$ is a vertex cover in G of size $\ell \leq k$, then the sets $S_{i_1}, \ldots, S_{i_\ell}$ cover U.

Thus, given our instance of Vertex Cover, we formulate the instance of Set Cover described above, and pass it to our black box. We answer yes if and only if the black box answers yes.

(You can check that the instance of Set Cover pictured in Figure 8.2 is actually the one you'd get by following the reduction in this proof, starting from the graph in Figure 8.1.) ■

Here is something worth noticing, both about this proof and about the previous reductions in (8.4) and (8.5). Although the definition of $\leq_P$ allows us to issue many calls to our black box for Set Cover, we issued only one. Indeed, our algorithm for Vertex Cover consisted simply of encoding the problem as a single instance of Set Cover and then using the answer to this instance as our overall answer. This will be true of essentially all the reductions that we consider; they will consist of establishing $Y \leq_P X$ by transforming our instance of Y to a single instance of X, invoking our black box for X on this instance, and reporting the box's answer as our answer for the instance of Y.

Just as Set Cover is a natural generalization of Vertex Cover, there is a natural generalization of Independent Set as a packing problem for arbitrary sets. Specifically, we define the *Set Packing Problem* as follows.

> *Given a set U of n elements, a collection $S_1, \ldots, S_m$ of subsets of U, and a number k, does there exist a collection of at least k of these sets with the property that no two of them intersect?*

In other words, we wish to "pack" a large number of sets together, with the constraint that no two of them are overlapping.

As an example of where this type of issue might arise, imagine that we have a set U of n non-sharable *resources*, and a set of m software processes. The i^{th} process requires the set $S_i \subseteq U$ of resources in order to run. Then the Set Packing Problem seeks a large collection of these processes that can be run

simultaneously, with the property that none of their resource requirements overlap (i.e., represent a conflict).

There is a natural analogue to (8.6), and its proof is almost the same as well; we will leave the details as an exercise.

(8.7) Independent Set $\leq_P$ Set Packing.

8.2 Reductions via "Gadgets": The Satisfiability Problem

We now introduce a somewhat more abstract set of problems, which are formulated in Boolean notation. As such, they model a wide range of problems in which we need to set decision variables so as to satisfy a given set of constraints; such formalisms are common, for example, in artificial intelligence. After introducing these problems, we will relate them via reduction to the graph- and set-based problems that we have been considering thus far.

The SAT and 3-SAT Problems

Suppose we are given a set X of n *Boolean variables* $x_1, \ldots, x_n$; each can take the value 0 or 1 (equivalently, "false" or "true"). By a *term* over X, we mean one of the variables x_i or its negation $\overline{x_i}$. Finally, a *clause* is simply a disjunction of distinct terms

$$t_1 \vee t_2 \vee \cdots \vee t_\ell.$$

(Again, each $t_i \in \{x_1, x_2, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\}$.) We say the clause has length ℓ if it contains ℓ terms.

We now formalize what it means for an assignment of values to satisfy a collection of clauses. A *truth assignment* for X is an assignment of the value 0 or 1 to each x_i; in other words, it is a function $v : X \to \{0, 1\}$. The assignment v implicitly gives $\overline{x_i}$ the opposite truth value from x_i. An assignment *satisfies* a clause C if it causes C to evaluate to 1 under the rules of Boolean logic; this is equivalent to requiring that at least one of the terms in C should receive the value 1. An assignment satisfies a collection of clauses $C_1, \ldots, C_k$ if it causes all of the C_i to evaluate to 1; in other words, if it causes the conjunction

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k$$

to evaluate to 1. In this case, we will say that v is a *satisfying assignment* with respect to $C_1, \ldots, C_k$; and that the set of clauses $C_1, \ldots, C_k$ is *satisfiable*.

Here is a simple example. Suppose we have the three clauses

$$(x_1 \vee \overline{x_2}), (\overline{x_1} \vee \overline{x_3}), (x_2 \vee \overline{x_3}).$$

Then the truth assignment v that sets all variables to 1 is not a satisfying assignment, because it does not satisfy the second of these clauses; but the truth assignment v' that sets all variables to 0 is a satisfying assignment.

We can now state the *Satisfiability Problem*, also referred to as *SAT*:

Given a set of clauses $C_1, \ldots, C_k$ over a set of variables $X = \{x_1, \ldots, x_n\}$, does there exist a satisfying truth assignment?

There is a special case of SAT that will turn out to be equivalently difficult and is somewhat easier to think about; this is the case in which all clauses contain exactly three terms (corresponding to distinct variables). We call this problem *3-Satisfiability*, or *3-SAT*:

Given a set of clauses $C_1, \ldots, C_k$, each of length 3, over a set of variables $X = \{x_1, \ldots, x_n\}$, does there exist a satisfying truth assignment?

Satisfiability and 3-Satisfiability are really fundamental combinatorial search problems; they contain the basic ingredients of a hard computational problem in very "bare-bones" fashion. We have to make n independent decisions (the assignments for each x_i) so as to satisfy a set of constraints. There are several ways to satisfy each constraint in isolation, but we have to arrange our decisions so that all constraints are satisfied simultaneously.

Reducing 3-SAT to Independent Set

We now relate the type of computational hardness embodied in SAT and 3-SAT to the superficially different sort of hardness represented by the search for independent sets and vertex covers in graphs. Specifically, we will show that 3-SAT $\leq_P$ Independent Set. The difficulty in proving a thing like this is clear; 3-SAT is about setting Boolean variables in the presence of constraints, while Independent Set is about selecting vertices in a graph. To solve an instance of 3-SAT using a black box for Independent Set, we need a way to encode all these Boolean constraints in the nodes and edges of a graph, so that satisfiability corresponds to the existence of a large independent set.

Doing this illustrates a general principle for designing complex reductions $Y \leq_P X$: building "gadgets" out of components in problem X to represent what is going on in problem Y.

(8.8) 3-SAT $\leq_P$ Independent Set.

Proof. We have a black box for Independent Set and want to solve an instance of 3-SAT consisting of variables $X = \{x_1, \ldots, x_n\}$ and clauses $C_1, \ldots, C_k$.

The key to thinking about the reduction is to realize that there are two conceptually distinct ways of thinking about an instance of 3-SAT.

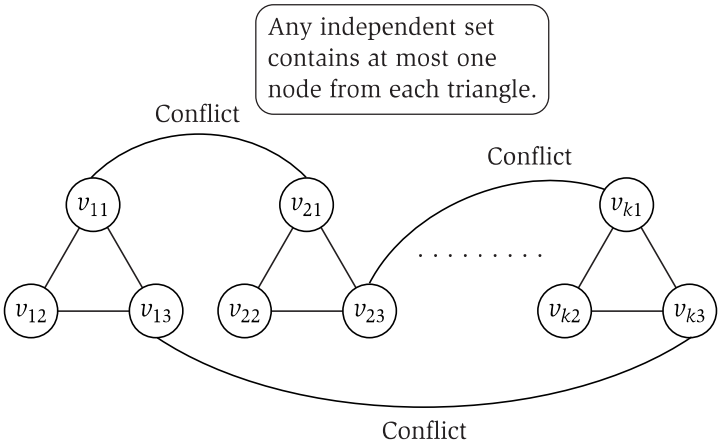

Figure 8.3 The reduction from 3-SAT to Independent Set.

- One way to picture the 3-SAT instance was suggested earlier: You have to make an independent 0/1 decision for each of the n variables, and you succeed if you manage to achieve one of three ways of satisfying each clause.
- A different way to picture the same 3-SAT instance is as follows: You have to choose one term from each clause, and then find a truth assignment that causes all these terms to evaluate to 1, thereby satisfying all clauses. So you succeed if you can select a term from each clause in such a way that no two selected terms "conflict"; we say that two terms *conflict* if one is equal to a variable x_i and the other is equal to its negation $\overline{x_i}$. If we avoid conflicting terms, we can find a truth assignment that makes the selected terms from each clause evaluate to 1.

Our reduction will be based on this second view of the 3-SAT instance; here is how we encode it using independent sets in a graph. First, construct a graph $G = (V, E)$ consisting of $3k$ nodes grouped into k triangles as shown in Figure 8.3. That is, for $i = 1, 2, \ldots, k$, we construct three vertices v_{i1}, v_{i2}, v_{i3} joined to one another by edges. We give each of these vertices a *label*; v_{ij} is labeled with the j^{th} term from the clause C_i of the 3-SAT instance.

Before proceeding, consider what the independent sets of size k look like in this graph: Since two vertices cannot be selected from the same triangle, they consist of all ways of choosing one vertex from each of the triangles. This is implementing our goal of choosing a term in each clause that will evaluate to 1; but we have so far not prevented ourselves from choosing two terms that conflict.

We encode conflicts by adding some more edges to the graph: For each pair of vertices whose labels correspond to terms that conflict, we add an edge between them. Have we now destroyed all the independent sets of size k, or does one still exist? It's not clear; it depends on whether we can still select one node from each triangle so that no conflicting pairs of vertices are chosen. But this is precisely what the 3-SAT instance required.

Let's claim, precisely, that the original 3-SAT instance is satisfiable if and only if the graph G we have constructed has an independent set of size at least k. First, if the 3-SAT instance is satisfiable, then each triangle in our graph contains at least one node whose label evaluates to 1. Let S be a set consisting of one such node from each triangle. We claim S is independent; for if there were an edge between two nodes $u, v \in S$, then the labels of u and v would have to conflict; but this is not possible, since they both evaluate to 1.

Conversely, suppose our graph G has an independent set S of size at least k. Then, first of all, the size of S is exactly k, and it must consist of one node from each triangle. Now, we claim that there is a truth assignment ν for the variables in the 3-SAT instance with the property that the labels of all nodes in S evaluate to 1. Here is how we could construct such an assignment ν. For each variable x_i, if neither x_i nor $\overline{x_i}$ appears as a label of a node in S, then we arbitrarily set $\nu(x_i) = 1$. Otherwise, exactly one of x_i or $\overline{x_i}$ appears as a label of a node in S; for if one node in S were labeled x_i and another were labeled $\overline{x_i}$, then there would be an edge between these two nodes, contradicting our assumption that S is an independent set. Thus, if x_i appears as a label of a node in S, we set $\nu(x_i) = 1$, and otherwise we set $\nu(x_i) = 0$. By constructing ν in this way, all labels of nodes in S will evaluate to 1.

Since G has an independent set of size at least k if and only if the original 3-SAT instance is satisfiable, the reduction is complete. ■

Some Final Observations: Transitivity of Reductions

We've now seen a number of different hard problems, of various flavors, and we've discovered that they are closely related to one another. We can infer a number of additional relationships using the following fact: $\leq_P$ is a *transitive* relation.

(8.9) *If $Z \leq_P Y$, and $Y \leq_P X$, then $Z \leq_P X$.*

Proof. Given a black box for X, we show how to solve an instance of Z; essentially, we just compose the two algorithms implied by $Z \leq_P Y$ and $Y \leq_P X$. We run the algorithm for Z using a black box for Y; but each time the black box for Y is called, we *simulate* it in a polynomial number of steps using the algorithm that solves instances of Y using a black box for X. ■

Transitivity can be quite useful. For example, since we have proved

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

we can conclude that 3-SAT $\leq_P$ Set Cover.

8.3 Efficient Certification and the Definition of NP

Reducibility among problems was the first main ingredient in our study of computational intractability. The second ingredient is a characterization of the class of problems that we are dealing with. Combining these two ingredients, together with a powerful theorem of Cook and Levin, will yield some surprising consequences.

Recall that in Chapter 1, when we first encountered the Independent Set Problem, we asked: Can we say anything *good* about it, from a computational point of view? And, indeed, there was something: If a graph does contain an independent set of size at least k, then we could give you an easy proof of this fact by exhibiting such an independent set. Similarly, if a 3-SAT instance is satisfiable, we can prove this to you by revealing the satisfying assignment. It may be an enormously difficult task to actually *find* such an assignment; but if we've done the hard work of finding one, it's easy for you to plug it into the clauses and check that they are all satisfied.

The issue here is the contrast between *finding* a solution and *checking* a proposed solution. For Independent Set or 3-SAT, we do not know a polynomial-time algorithm to find solutions; but *checking* a proposed solution to these problems can be easily done in polynomial time. To see that this is not an entirely trivial issue, consider the problem we'd face if we had to prove that a 3-SAT instance was *not* satisfiable. What "evidence" could we show that would convince you, in polynomial time, that the instance was unsatisfiable?

Problems and Algorithms

This will be the crux of our characterization; we now proceed to formalize it. The input to a computational problem will be encoded as a finite binary string s. We denote the length of a string s by $|s|$. We will identify a decision problem X with the *set* of strings on which the answer is "yes." An algorithm A for a decision problem receives an input string s and returns the value "yes" or "no"—we will denote this returned value by $A(s)$. We say that A *solves* the problem X if for all strings s, we have $A(s) = \texttt{yes}$ if and only if $s \in X$.

As always, we say that A has a *polynomial running time* if there is a polynomial function $p(\cdot)$ so that for every input string s, the algorithm A terminates on s in at most $O(p(|s|))$ steps. Thus far in the book, we have been concerned with problems solvable in polynomial time. In the notation

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

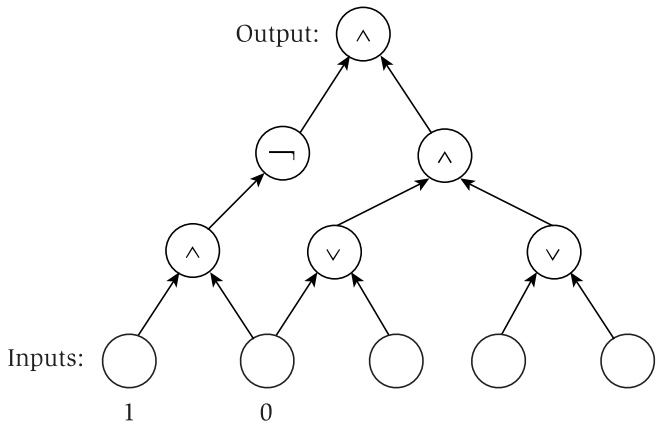

Figure 8.4 A circuit with three inputs, two additional sources that have assigned truth values, and one output.

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.

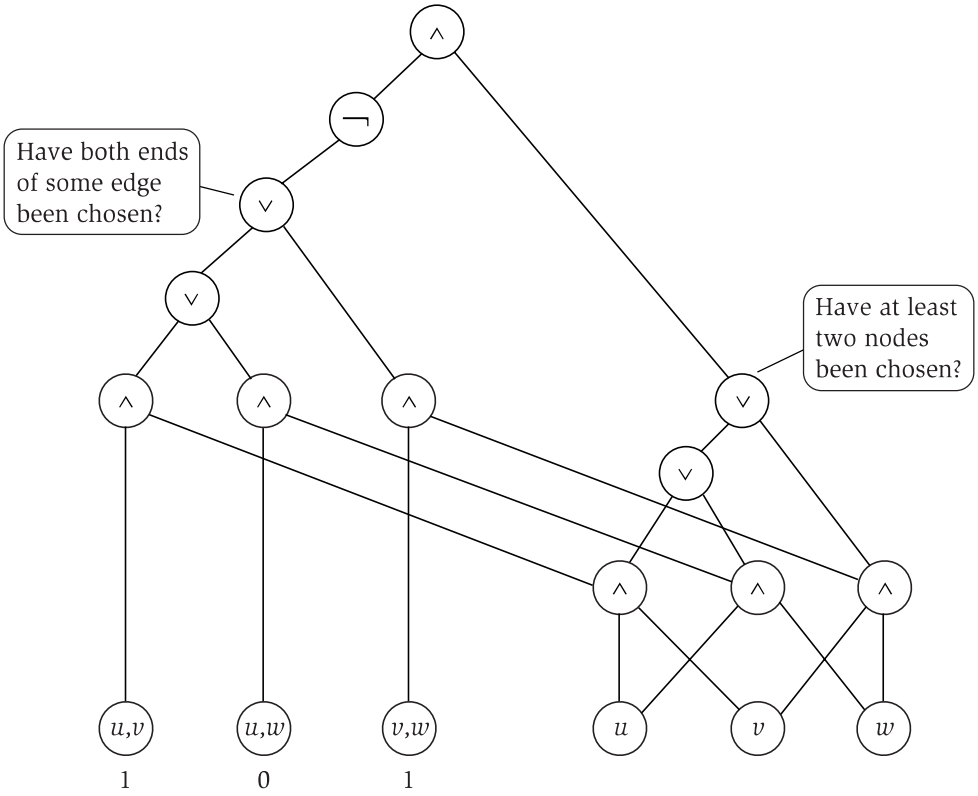

Figure 8.5 A circuit to verify whether a 3-node graph contains a 2-node independent set.

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?

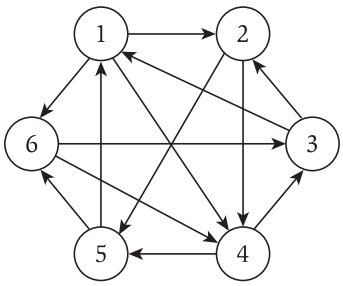

Figure 8.6 A directed graph containing a Hamiltonian cycle.

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}.

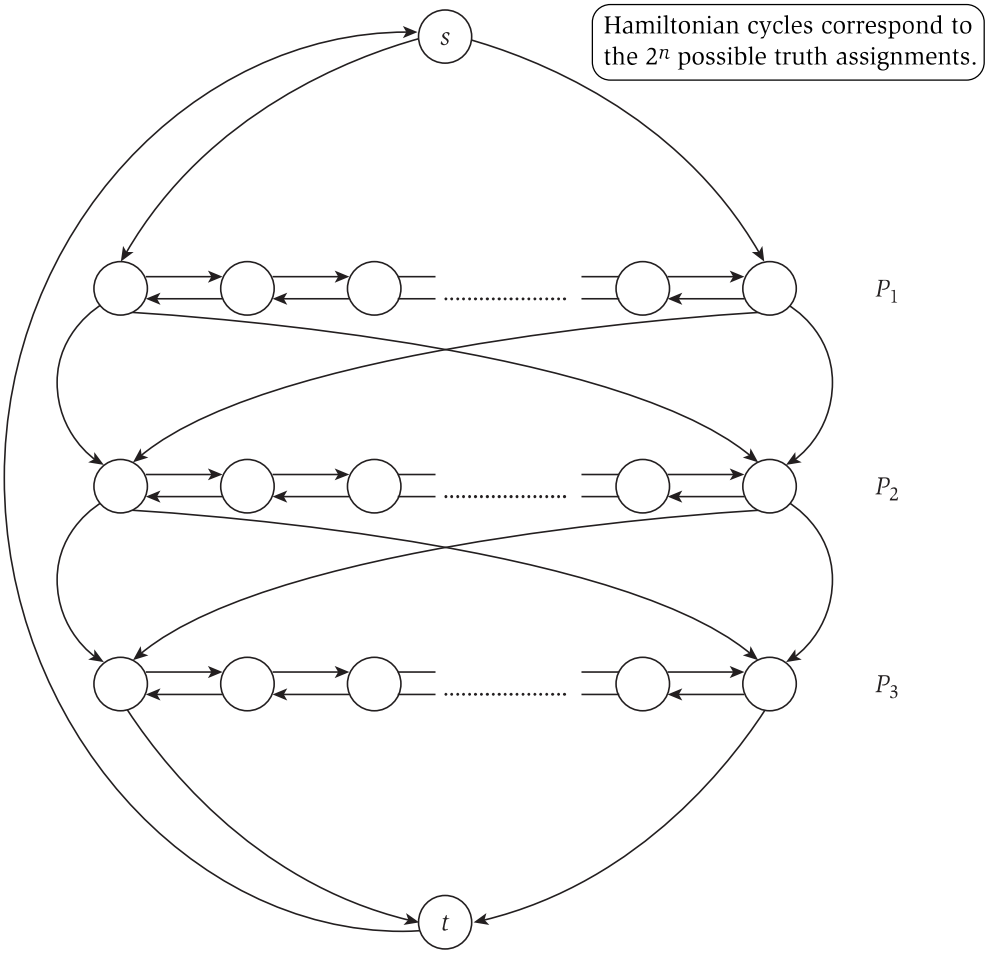

Figure 8.7 The reduction from 3-SAT to Hamiltonian Cycle: part 1.

We hook these paths together as follows. For each $i = 1, 2, \ldots, n-1$, we define edges from v_{i1} to $v_{i+1,1}$ and to $v_{i+1,b}$. We also define edges from v_{ib} to $v_{i+1,1}$ and to $v_{i+1,b}$. We add two extra nodes s and t; we define edges from s to v_{11} and v_{1b}; from v_{n1} and v_{nb} to t; and from t to s.

The construction up to this point is pictured in Figure 8.7. It's important to pause here and consider what the Hamiltonian cycles in our graph look like. Since only one edge leaves t, we know that any Hamiltonian cycle C must use the edge (t, s). After entering s, the cycle C can then traverse P_1 either left to right or right to left; regardless of what it does here, it can then traverse P_2 either left to right or right to left; and so forth, until it finishes traversing P_n and enters t. In other words, there are exactly 2^n different Hamiltonian cycles, and they correspond to the n independent choices of how to traverse each P_i.

This naturally models the n independent choices of how to set each variables $x_1, \ldots, x_n$ in the 3-SAT instance. Thus we will identify each Hamiltonian cycle uniquely with a truth assignment as follows: If $\mathcal{C}$ traverses P_i left to right, then x_i is set to 1; otherwise, x_i is set to 0.

Now we add nodes to model the clauses; the 3-SAT instance will turn out to be satisfiable if and only if any Hamiltonian cycle survives. Let's consider, as a concrete example, a clause

$$C_1 = x_1 \vee \overline{x_2} \vee x_3.$$

In the language of Hamiltonian cycles, this clause says, "The cycle should traverse P_1 left to right; or it should traverse P_2 right to left; or it should traverse P_3 left to right." So we add a node c_1, as in Figure 8.8, that does just this. (Note that certain edges have been eliminated from this drawing, for the sake of clarity.) For some value of ℓ, node c_1 will have edges *from* $v_{1\ell}$, $v_{2,\ell+1}$, and $v_{3\ell}$; it will have edges *to* $v_{1,\ell+1}$, $v_{2,\ell}$, and $v_{3,\ell+1}$. Thus it can be easily spliced into any Hamiltonian cycle that traverses P_1 left to right by visiting node c_1 between $v_{1\ell}$ and $v_{1,\ell+1}$; similarly, c_1 can be spliced into any Hamiltonian cycle that traverses P_2 right to left, or P_3 left to right. It cannot be spliced into a Hamiltonian cycle that does not do any of these things.

More generally, we will define a node c_j for each clause C_j. We will reserve node positions $3j$ and $3j+1$ in each path P_i for variables that participate in clause C_j. Suppose clause C_j contains a term t. Then if $t = x_i$, we will add edges $(v_{i,3j}, c_j)$ and $(c_j, v_{i,3j+1})$; if $t = \overline{x_i}$, we will add edges $(v_{i,3j+1}, c_j)$ and $(c_j, v_{i,3j})$.

This completes the construction of the graph G. Now, following our generic outline for NP-completeness proofs, we claim that the 3-SAT instance is satisfiable if and only if G has a Hamiltonian cycle.

First suppose there is a satisfying assignment for the 3-SAT instance. Then we define a Hamiltonian cycle following our informal plan above. If x_i is assigned 1 in the satisfying assignment, then we traverse the path P_i left to right; otherwise we traverse P_i right to left. For each clause C_j, since it is satisfied by the assignment, there will be at least one path P_i in which we will be going in the "correct" direction relative to the node c_j, and we can splice it into the tour there via edges incident on $v_{i,3j}$ and $v_{i,3j+1}$.

Conversely, suppose that there is a Hamiltonian cycle $\mathcal{C}$ in G. The crucial thing to observe is the following. If $\mathcal{C}$ enters a node c_j on an edge from $v_{i,3j}$, it must depart on an edge to $v_{i,3j+1}$. For if not, then $v_{i,3j+1}$ will have only one unvisited neighbor left, namely, $v_{i,3j+2}$, and so the tour will not be able to visit this node and still maintain the Hamiltonian property. Symmetrically, if it enters from $v_{i,3j+1}$, it must depart immediately to $v_{i,3j}$. Thus, for each node c_j,

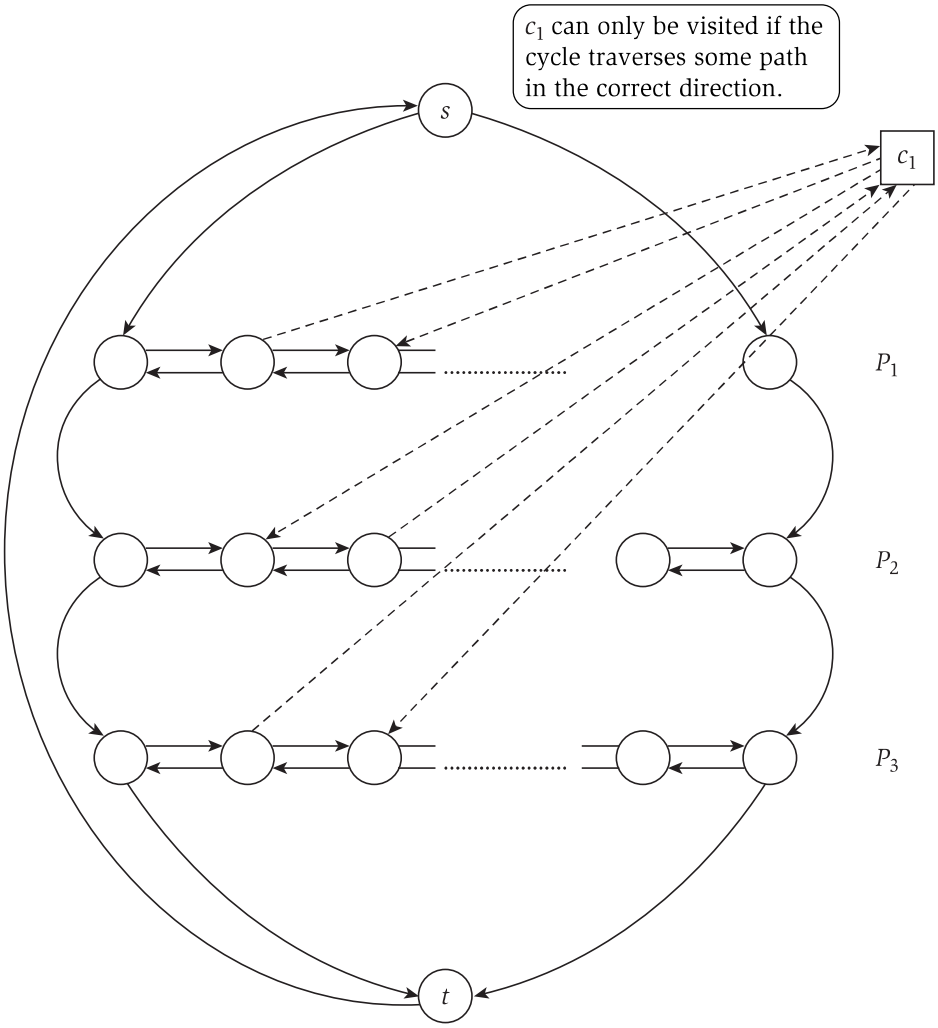

Figure 8.8 The reduction from 3-SAT to Hamiltonian Cycle: part 2.

the nodes immediately before and after c_j in the cycle $\mathcal{C}$ are joined by an edge e in G; thus, if we remove c_j from the cycle and insert this edge e for each j, then we obtain a Hamiltonian cycle $\mathcal{C}'$ on the subgraph $G - \{c_1, \ldots, c_k\}$. This is our original subgraph, before we added the clause nodes; as we noted above, any Hamiltonian cycle in this subgraph must traverse each P_i fully in one direction or the other. We thus use $\mathcal{C}'$ to define the following truth assignment for the 3-SAT instance. If $\mathcal{C}'$ traverses P_i left to right, then we set $x_i = 1$; otherwise we set $x_i = 0$. Since the larger cycle $\mathcal{C}$ was able to visit each clause node c_j, at least one of the paths was traversed in the "correct" direction relative to the node c_j, and so the assignment we have defined satisfies all the clauses.

Having established that the 3-SAT instance is satisfiable if and only if G has a Hamiltonian cycle, our proof is complete. ■

Proving Traveling Salesman is NP-Complete

Armed with our basic hardness result for Hamiltonian Cycle, we can move on to show the hardness of Traveling Salesman.

(8.18) Traveling Salesman *is NP-complete.*

Proof. It is easy to see that Traveling Salesman is in $\mathcal{NP}$: The certificate is a permutation of the cities, and a certifier checks that the length of the corresponding tour is at most the given bound.

We now show that Hamiltonian Cycle $\leq_P$ Traveling Salesman. Given a directed graph $G = (V, E)$, we define the following instance of Traveling Salesman. We have a city v_i' for each node v_i of the graph G. We define $d(v_i', v_j')$ to be 1 if there is an edge (v_i, v_j) in G, and we define it to be 2 otherwise.

Now we claim that G has a Hamiltonian cycle if and only if there is tour of length at most n in our Traveling Salesman instance. For if G has a Hamiltonian cycle, then this ordering of the corresponding cities defines a tour of length n. Conversely, suppose there is a tour of length at most n. The expression for the length of this tour is a sum of n terms, each of which is at least 1; thus it must be the case that all the terms are equal to 1. Hence each pair of nodes in G that correspond to consecutive cities on the tour must be connected by an edge; it follows that the ordering of these corresponding nodes must form a Hamiltonian cycle. ■

Note that allowing *asymmetric* distances in the Traveling Salesman Problem ($d(v_i', v_j') \neq d(v_j', v_i')$) played a crucial role; since the graph in the Hamiltonian Cycle instance is directed, our reduction yielded a Traveling Salesman instance with asymmetric distances.

In fact, the analogue of the Hamiltonian Cycle Problem for undirected graphs is also NP-complete; although we will not prove this here, it follows via a not-too-difficult reduction from directed Hamiltonian Cycle. Using this undirected Hamiltonian Cycle Problem, an exact analogue of (8.18) can be used to prove that the Traveling Salesman Problem with symmetric distances is also NP-complete.

Of course, the most famous special case of the Traveling Salesman Problem is the one in which the distances are defined by a set of n points in the plane. It is possible to reduce Hamiltonian Cycle to this special case as well, though this is much trickier.

Extensions: The Hamiltonian Path Problem

It is also sometimes useful to think about a variant of Hamiltonian Cycle in which it is not necessary to return to one's starting point. Thus, given a directed graph $G = (V, E)$, we say that a path P in G is a *Hamiltonian path* if it contains each vertex exactly once. (The path is allowed to start at any node and end at any node, provided it respects this constraint.) Thus such a path consists of distinct nodes $v_{i_1}, v_{i_2}, \ldots, v_{i_n}$ in order, such that they collectively constitute the entire vertex set V; by way of contrast with a Hamiltonian cycle, it is not necessary for there to be an edge from v_{i_n} back to v_{i_1}. Now, the *Hamiltonian Path Problem* asks:

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

Using the hardness of Hamiltonian Cycle, we show the following.

(8.19) Hamiltonian Path *is NP-complete.*

Proof. First of all, Hamiltonian Path is in $\mathcal{NP}$: A certificate could be a path in G, and a certifier could then check that it is indeed a path and that it contains each node exactly once.

One way to show that Hamiltonian Path is NP-complete is to use a reduction from 3-SAT that is almost identical to the one we used for Hamiltonian Cycle: We construct the same graph that appears in Figure 8.7, *except* that we do not include an edge from t to s. If there is any Hamiltonian path in this modified graph, it must begin at s (since s has no incoming edges) and end at t (since t has no outgoing edges). With this one change, we can adapt the argument used in the Hamiltonian Cycle reduction more or less word for word to argue that there is a satisfying assignment for the instance of 3-SAT if and only if there is a Hamiltonian path.

An alternate way to show that Hamiltonian Path is NP-complete is to prove that Hamiltonian Cycle $\leq_P$ Hamiltonian Path. Given an instance of Hamiltonian Cycle, specified by a directed graph G, we construct a graph G' as follows. We choose an arbitrary node v in G and replace it with two new nodes v' and v''. All edges out of v in G are now out of v'; and all edges into v in G are now into v''. More precisely, each edge (v, w) in G is replaced by an edge (v', w); and each edge (u, v) in G is replaced by an edge (u, v''). This completes the construction of G'.

We claim that G' contains a Hamiltonian path if and only if G contains a Hamiltonian cycle. Indeed, suppose C is a Hamiltonian cycle in G, and consider traversing it beginning and ending at node v. It is easy to see that the same ordering of nodes forms a Hamiltonian path in G' that begins at v' and ends at v''. Conversely, suppose P is a Hamiltonian path in G'. Clearly P must begin

at v' (since v' has no incoming edges) and end at v'' (since v'' has no outgoing edges). If we replace v' and v'' with v, then this ordering of nodes forms a Hamiltonian cycle in G. ■

8.6 Partitioning Problems

In the next two sections, we consider two fundamental *partitioning* problems, in which we are searching over ways of dividing a collection of objects into subsets. Here we show the NP-completeness of a problem that we call *3-Dimensional Matching*. In the next section we consider *Graph Coloring*, a problem that involves partitioning the nodes of a graph.

The 3-Dimensional Matching Problem

We begin by discussing the 3-Dimensional Matching Problem, which can be motivated as a harder version of the Bipartite Matching Problem that we considered earlier. We can view the Bipartite Matching Problem in the following way: We are given two sets X and Y, each of size n, and a set P of pairs drawn from $X \times Y$. The question is: Does there exist a set of n pairs in P so that each element in $X \cup Y$ is contained in exactly one of these pairs? The relation to Bipartite Matching is clear: the set P of pairs is simply the edges of the bipartite graph.

Now Bipartite Matching is a problem we know how to solve in polynomial time. But things get much more complicated when we move from ordered pairs to ordered triples. Consider the following 3-Dimensional Matching Problem:

> *Given disjoint sets X, Y, and Z, each of size n, and given a set $T \subseteq X \times Y \times Z$ of ordered triples, does there exist a set of n triples in T so that each element of $X \cup Y \cup Z$ is contained in exactly one of these triples?*

Such a set of triples is called a *perfect three-dimensional matching*.

An interesting thing about 3-Dimensional Matching, beyond its relation to Bipartite Matching, is that it simultaneously forms a special case of both Set Cover and Set Packing: we are seeking to *cover* the ground set $X \cup Y \cup Z$ with a collection of *disjoint* sets. More concretely, *3-Dimensional Matching* is a special case of *Set Cover* since we seek to cover the ground set $U = X \cup Y \cup Z$ using at most n sets from a given collection (the triples). Similarly, 3-Dimensional Matching is a special case of Set Packing, since we are seeking n disjoint subsets of the ground set $U = X \cup Y \cup Z$.

Proving 3-Dimensional Matching Is NP-Complete

The arguments above can be turned quite easily into proofs that 3-Dimensional Matching $\leq_P$ Set Cover and that 3-Dimensional Matching $\leq_P$ Set Packing.

But this doesn't help us establish the NP-completeness of 3-Dimensional Matching, since these reductions simply show that 3-Dimensional Matching can be reduced to some very hard problems. What we need to show is the other direction: that a known NP-complete problem can be reduced to 3-Dimensional Matching.

(8.20) 3-Dimensional Matching *is NP-complete.*

Proof. Not surprisingly, it is easy to prove that 3-Dimensional Matching is in $\mathcal{NP}$. Given a collection of triples $T \subset X \times Y \times Z$, a certificate that there is a solution could be a collection of triples $T' \subseteq T$. In polynomial time, one could verify that each element in $X \cup Y \cup Z$ belongs to exactly one of the triples in T'.

For the reduction, we again return all the way to 3-SAT. This is perhaps a little more curious than in the case of Hamiltonian Cycle, since 3-Dimensional Matching is so closely related to both Set Packing and Set Cover; but in fact the partitioning requirement is very hard to encode using either of these problems.

Thus, consider an arbitrary instance of 3-SAT, with n variables $x_1, \ldots, x_n$ and k clauses $C_1, \ldots, C_k$. We will show how to solve it, given the ability to detect perfect three-dimensional matchings.

The overall strategy in this reduction will be similar (at a very high level) to the approach we followed in the reduction from 3-SAT to Hamiltonian Cycle. We will first design gadgets that encode the independent choices involved in the truth assignment to each variable; we will then add gadgets that encode the constraints imposed by the clauses. In performing this construction, we will initially describe all the elements in the 3-Dimensional Matching instance simply as "elements," without trying to specify for each one whether it comes from X, Y, or Z. At the end, we will observe that they naturally decompose into these three sets.

Here is the basic gadget associated with variable x_i. We define elements $A_i = \{a_{i1}, a_{i2}, \ldots, a_{i,2k}\}$ that constitute the *core* of the gadget; we define elements $B_i = \{b_{i1}, \ldots, b_{i,2k}\}$ at the *tips* of the gadget. For each $j = 1, 2, \ldots, 2k$, we define a triple $t_{ij} = (a_{ij}, a_{i,j+1}, b_{ij})$, where we interpret addition modulo $2k$. Three of these gadgets are pictured in Figure 8.9. In gadget i, we will call a triple t_{ij} *even* if j is even, and *odd* if j is odd. In an analogous way, we will refer to a tip b_{ij} as being either *even* or *odd*.

These will be the only triples that contain the elements in A_i, so we can already say something about how they must be covered in any perfect matching: we must either use all the even triples in gadget i, or all the odd triples in gadget i. This will be our basic way of encoding the idea that x_i can

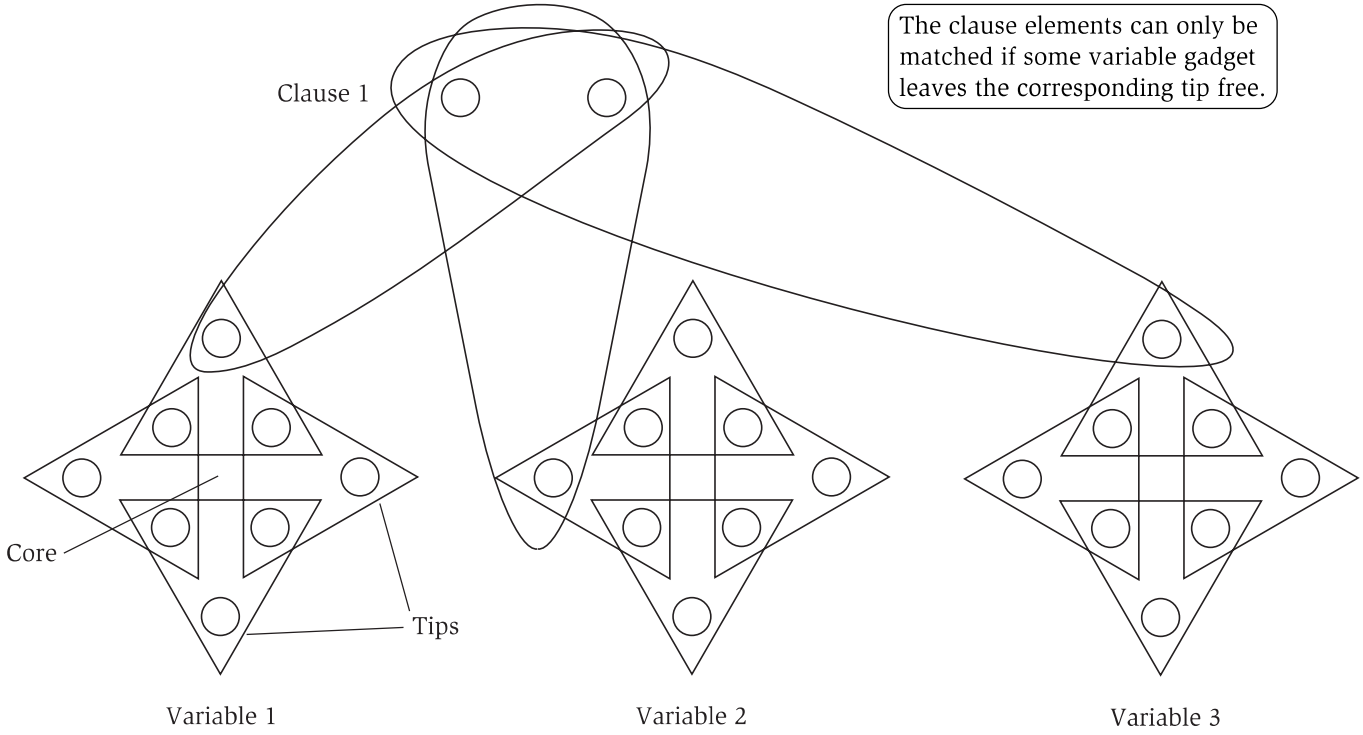

Figure 8.9 The reduction from 3-SAT to 3-Dimensional Matching.

be set to either 0 or 1; if we select all the even triples, this will represent setting $x_i = 0$, and if we select all the odd triples, this will represent setting $x_i = 1$.

Here is another way to view the odd/even decision, in terms of the tips of the gadget. If we decide to use the even triples, we cover the even tips of the gadget and leave the odd tips free. If we decide to use the odd triples, we cover the odd tips of the gadget and leave the even tips free. Thus our decision of how to set x_i can be viewed as follows: Leaving the odd tips free corresponds to 0, while leaving the even tips free corresponds to 1. This will actually be the more useful way to think about things in the remainder of the construction.

So far we can make this even/odd choice independently for each of the n variable gadgets. We now add elements to model the clauses and to constrain the assignments we can choose. As in the proof of (8.17), let's consider the example of a clause

$$C_1 = x_1 \vee \overline{x_2} \vee x_3.$$

In the language of three-dimensional matchings, it tells us, "The matching on the cores of the gadgets should leave the even tips of the first gadget free; or it should leave the odd tips of the second gadget free; or it should leave the even tips of the third gadget free." So we add a *clause gadget* that does precisely

this. It consists of a set of two *core* elements $P_1 = \{p_1, p_1'\}$, and three triples that contain them. One has the form (p_1, p_1', b_{1j}) for an even tip b_{1j}; another includes p_1, p_1', and an odd tip $b_{2,j'}$; and a third includes p_1, p_1', and an even tip $b_{3,j''}$. These are the only three triples that cover P_1, so we know that one of them must be used; this enforces the clause constraint exactly.

In general, for clause C_j, we create a gadget with two core elements $P_j = \{p_j, p_j'\}$, and we define three triples containing P_j as follows. Suppose clause C_j contains a term t. If $t = x_i$, we define a triple $(p_j, p_j', b_{i,2j})$; if $t = \overline{x_i}$, we define a triple $(p_j, p_j', b_{i,2j-1})$. Note that only clause gadget j makes use of tips b_{im} with $m = 2j$ or $m = 2j - 1$; thus, the clause gadgets will never "compete" with each other for free tips.

We are almost done with the construction, but there's still one problem. Suppose the set of clauses has a satisfying assignment. Then we make the corresponding choices of odd/even for each variable gadget; this leaves at least one free tip for each clause gadget, and so all the core elements of the clause gadgets get covered as well. The problem is that *we haven't covered all the tips.* We started with $n \cdot 2k = 2nk$ tips; the triples $\{t_{ij}\}$ covered nk of them; and the clause gadgets covered an additional k of them. This leaves $(n - 1)k$ tips left to be covered.

We handle this problem with a very simple trick: we add $(n - 1)k$ "cleanup gadgets" to the construction. Cleanup gadget i consists of two core elements $Q_i = \{q_i, q_i'\}$, and there is a triple (q_i, q_i, b) for *every* tip b in every variable gadget. This is the final piece of the construction.

Thus, if the set of clauses has a satisfying assignment, then we make the corresponding choices of odd/even for each variable gadget; as before, this leaves at least one free tip for each clause gadget. Using the cleanup gadgets to cover the remaining tips, we see that all core elements in the variable, clause, and cleanup gadgets have been covered, and all tips have been covered as well.

Conversely, suppose there is a perfect three-dimensional matching in the instance we have constructed. Then, as we argued above, in each variable gadget the matching chooses either all the even $\{t_{ij}\}$ or all the odd $\{t_{ij}\}$. In the former case, we set $x_i = 0$ in the 3-SAT instance; and in the latter case, we set $x_i = 1$. Now consider clause C_j; has it been satisfied? Because the two core elements in P_j have been covered, at least one of the three variable gadgets corresponding to a term in C_j made the "correct" odd/even decision, and this induces a variable assignment that satisfies C_j.

This concludes the proof, except for one last thing to worry about: Have we really constructed an instance of 3-Dimensional Matching? We have a collection of elements, and triples containing certain of them, but can the elements really be partitioned into appropriate sets X, Y, and Z of equal size?

Fortunately, the answer is yes. We can define X to be set of all a_{ij} with j even, the set of all p_j, and the set of all q_i. We can define Y to be set of all a_{ij} with j odd, the set of all p'_j, and the set of all q'_i. Finally, we can define Z to be the set of all tips b_{ij}. It is now easy to check that each triple consists of one element from each of X, Y, and Z. ■

8.7 Graph Coloring

When you color a map (say, the states in a U.S. map or the countries on a globe), the goal is to give neighboring regions different colors so that you can see their common borders clearly while minimizing visual distraction by using only a few colors. In the middle of the 19th century, Francis Guthrie noticed that you could color a map of the counties of England this way with only four colors, and he wondered whether the same was true for every map. He asked his brother, who relayed the question to one of his professors, and thus a famous mathematical problem was born: the *Four-Color Conjecture*.

The Graph Coloring Problem

Graph coloring refers to the same process on an undirected graph G, with the nodes playing the role of the regions to be colored, and the edges representing pairs that are neighbors. We seek to assign a *color* to each node of G so that if (u, v) is an edge, then u and v are assigned different colors; and the goal is to do this while using a small set of colors. More formally, a *k-coloring* of G is a function $f : V \to \{1, 2, \ldots, k\}$ so that for every edge (u, v), we have $f(u) \neq f(v)$. (So the available colors here are named $1, 2, \ldots, k$, and the function f represents our choice of a color for each node.) If G has a k-coloring, then we will say that it is a *k-colorable graph*.

In contrast with the case of maps in the plane, it's clear that there's not some fixed constant k so that every graph has a k-coloring: For example, if we take a set of n nodes and join each pair of them by an edge, the resulting graph needs n colors. However, the algorithmic version of the problem is very interesting:

Given a graph G and a bound k, does G have a k-coloring?

We will refer to this as the *Graph Coloring Problem*, or as *k-Coloring* when we wish to emphasize a particular choice of k.

Graph Coloring turns out to be a problem with a wide range of applications. While it's not clear there's ever been much genuine demand from cartographers, the problem arises naturally whenever one is trying to allocate resources in the presence of conflicts.

- Suppose, for example, that we have a collection of n processes on a system that can run multiple jobs concurrently, but certain pairs of jobs cannot be scheduled at the same time because they both need a particular resource. Over the next k time steps of the system, we'd like to schedule each process to run in at least one of them. Is this possible? If we construct a graph G on the set of processes, joining two by an edge if they have a conflict, then a k-coloring of G represents a conflict-free schedule of the processes: all nodes colored j can be scheduled in step j, and there will never be contention for any of the resources.
- Another well-known application arises in the design of compilers. Suppose we are compiling a program and are trying to assign each variable to one of k registers. If two variables are in use at a common point in time, then they cannot be assigned to the same register. (Otherwise one would end up overwriting the other.) Thus we can build a graph G on the set of variables, joining two by an edge if they are both in use at the same time. Now a k-coloring of G corresponds to a safe way of allocating variables to registers: All nodes colored j can be assigned to register j, since no two of them are in use at the same time.
- A third application arises in wavelength assignment for wireless communication devices: We'd like to assign one of k transmitting wavelengths to each of n devices; but if two devices are sufficiently close to each other, then they need to be assigned different wavelengths to prevent interference. To deal with this, we build a graph G on the set of devices, joining two nodes if they're close enough to interfere with each other; a k-coloring of this graph is now an assignment of wavelengths so that any nodes assigned the same wavelength are far enough apart that interference won't be a problem. (Interestingly, this is an application of graph coloring where the "colors" being assigned to nodes are positions on the electromagnetic spectrum—in other words, under a slightly liberal interpretation, they're actually colors.)

The Computational Complexity of Graph Coloring

What is the complexity of k-Coloring? First of all, the case $k = 2$ is a problem we've already seen in Chapter 3. Recall, there, that we considered the problem of determining whether a graph G is bipartite, and we showed that this is equivalent to the following question: Can one color the nodes of G red and blue so that every edge has one red end and one blue end?

But this latter question is precisely the Graph Coloring Problem in the case when there are $k = 2$ colors (i.e., red and blue) available. Thus we have argued that

(8.21) *A graph G is 2-colorable if and only if it is bipartite.*

This means we can use the algorithm from Section 3.4 to decide whether an input graph G is 2-colorable in $O(m+n)$ time, where n is the number of nodes of G and m is the number of edges.

As soon as we move up to $k=3$ colors, things become much harder. No simple efficient algorithm for the 3-Coloring Problem suggests itself, as it did for 2-Coloring, and it is also a very difficult problem to reason about. For example, one might initially suspect that any graph that is not 3-colorable will contain a "proof" in the form of four nodes that are all mutually adjacent (and hence would need four different colors)—but this is not true. The graph in Figure 8.10, for instance, is not 3-colorable for a somewhat more subtle (though still explainable) reason, and it is possible to draw much more complicated graphs that are not 3-colorable for reasons that seem very hard to state succinctly.

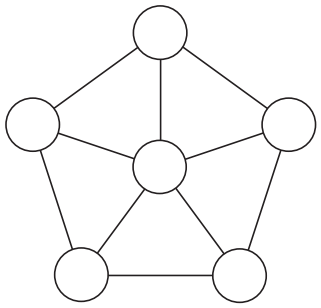

Figure 8.10 A graph that is not 3-colorable.

In fact, the case of three colors is already a very hard problem, as we show now.

Proving 3-Coloring Is NP-Complete

(8.22) 3-Coloring *is NP-complete.*

Proof. It is easy to see why the problem is in $\mathcal{NP}$. Given G and k, one certificate that the answer is yes is simply a k-coloring: One can verify in polynomial time that at most k colors are used, and that no pair of nodes joined by an edge receive the same color.

Like the other problems in this section, 3-Coloring is a problem that is hard to relate at a superficial level to other NP-complete problems we've seen. So once again, we're going to reach all the way back to 3-SAT. Given an arbitrary instance of 3-SAT, with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_k$, we will solve it using a black box for 3-Coloring.

The beginning of the reduction is quite intuitive. Perhaps the main power of 3-Coloring for encoding Boolean expressions lies in the fact that we can associate graph nodes with particular terms, and by joining them with edges we ensure that they get different colors; this can be used to set one true and the other false. So with this in mind, we define nodes v_i and $\overline{v_i}$ corresponding to each variable x_i and its negation $\overline{x_i}$. We also define three "special nodes" T, F, and B, which we refer to as *True*, *False*, and *Base*.

To begin, we join each pair of nodes v_i, $\overline{v_i}$ to each other by an edge, and we join both these nodes to *Base*. (This forms a triangle on v_i, $\overline{v_i}$, and *Base*, for each i.) We also join *True*, *False*, and *Base* into a triangle. The simple graph

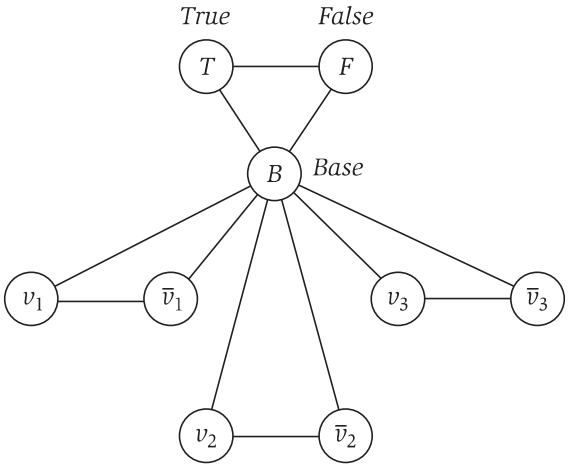

Figure 8.11 The beginning of the reduction for 3-Coloring.

G we have defined thus far is pictured in Figure 8.11, and it already has some useful properties.

- In any 3-coloring of G, the nodes v_i and $\overline{v_i}$ must get different colors, and both must be different from *Base*.
- In any 3-coloring of G, the nodes *True*, *False*, and *Base* must get all three colors in some permutation. Thus we can refer to the three colors as the *True* color, the *False* color, and the *Base* color, based on which of these three nodes gets which color. In particular, this means that for each i, one of v_i or $\overline{v_i}$ gets the *True* color, and the other gets the *False* color. For the remainder of the construction, we will consider the variable x_i to be set to 1 in the given instance of 3-SAT if and only if the node v_i gets assigned the *True* color.

So in summary, we now have a graph G in which any 3-coloring implicitly determines a truth assignment for the variables in the 3-SAT instance. We now need to grow G so that only satisfying assignments can be extended to 3-colorings of the full graph. How should we do this?

As in other 3-SAT reductions, let's consider a clause like $x_1 \vee \overline{x_2} \vee x_3$. In the language of 3-colorings of G, it says, "At least one of the nodes v_1, $\overline{v_2}$, or v_3 should get the *True* color." So what we need is a little subgraph that we can plug into G, so that any 3-coloring that extends into this subgraph must have the property of assigning the *True* color to at least one of v_1, $\overline{v_2}$, or v_3. It takes some experimentation to find such a subgraph, but one that works is depicted in Figure 8.12.

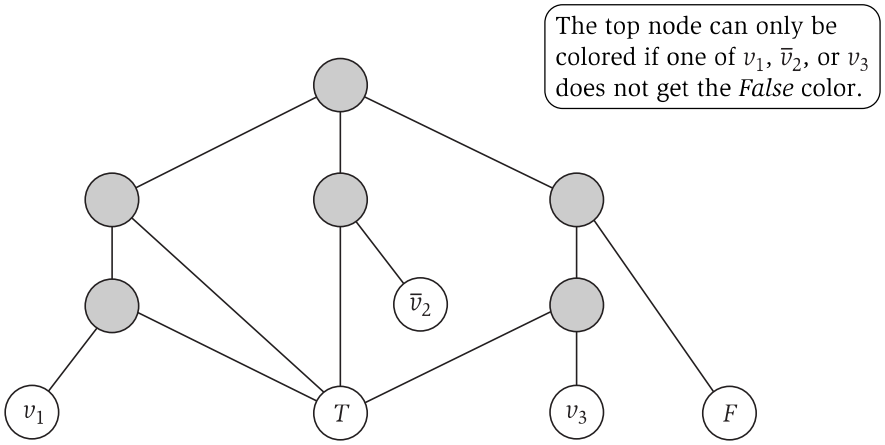

Figure 8.12 Attaching a subgraph to represent the clause $x_1 \vee \overline{x}_2 \vee x_3$.

This six-node subgraph "attaches" to the rest of G at five existing nodes: *True*, *False*, and those corresponding to the three terms in the clause that we're trying to represent (in this case, v_1, $\overline{v_2}$, and v_3.) Now suppose that in some 3-coloring of G all three of v_1, $\overline{v_2}$, and v_3 are assigned the *False* color. Then the lowest two shaded nodes in the subgraph must receive the *Base* color, the three shaded nodes above them must receive, respectively, the *False*, *Base*, and *True* colors, and hence there's no color that can be assigned to the topmost shaded node. In other words, a 3-coloring in which none of v_1, $\overline{v_2}$, or v_3 is assigned the *True* color cannot be extended to a 3-coloring of this subgraph.[2]

Finally, and conversely, some hand-checking of cases shows that as long as one of v_1, $\overline{v_2}$, or v_3 is assigned the *True* color, the full subgraph can be 3-colored.

So from this, we can complete the construction: We start with the graph G defined above, and for each clause in the 3-SAT instance, we attach a six-node subgraph as shown in Figure 8.12. Let us call the resulting graph G'.

[2] This argument actually gives considerable insight into how one comes up with this subgraph in the first place. The goal is to have a node like the topmost one that cannot receive any color. So we start by "plugging in" three nodes corresponding to the terms, all colored *False*, at the bottom. For each one, we then work upward, pairing it off with a node of a known color to force the node above to have the third color. Proceeding in this way, we can arrive at a node that is forced to have any color we want. So we force each of the three different colors, starting from each of the three different terms, and then we plug all three of these differently colored nodes into our topmost node, arriving at the impossibility.

We now claim that the given 3-SAT instance is satisfiable if and only if G' has a 3-coloring. First, suppose that there is a satisfying assignment for the 3-SAT instance. We define a coloring of G' by first coloring *Base*, *True*, and *False* arbitrarily with the three colors, then, for each i, assigning v_i the *True* color if $x_i = 1$ and the *False* color if $x_i = 0$. We then assign $\overline{v_i}$ the only available color. Finally, as argued above, it is now possible to extend this 3-coloring into each six-node clause subgraph, resulting in a 3-coloring of all of G'.

Conversely, suppose G' has a 3-coloring. In this coloring, each node v_i is assigned either the *True* color or the *False* color; we set the variable x_i correspondingly. Now we claim that in each clause of the 3-SAT instance, at least one of the terms in the clause has the truth value 1. For if not, then all three of the corresponding nodes has the *False* color in the 3-coloring of G' and, as we have seen above, there is no 3-coloring of the corresponding clause subgraph consistent with this—a contradiction. ■

When $k > 3$, it is very easy to reduce the 3-Coloring Problem to k-Coloring. Essentially, all we do is to take an instance of 3-Coloring, represented by a graph G, add $k - 3$ new nodes, and join these new nodes to each other and to every node in G. The resulting graph is k-colorable if and only if the original graph G is 3-colorable. Thus k-Coloring for any $k > 3$ is NP-complete as well.

Coda: The Resolution of the Four-Color Conjecture

To conclude this section, we should finish off the story of the Four-Color Conjecture for maps in the plane as well. After more than a hundred years, the conjecture was finally proved by Appel and Haken in 1976. The structure of the proof was a simple induction on the number of regions, but the induction step involved nearly two thousand fairly complicated cases, and the verification of these cases had to be carried out by a computer. This was not a satisfying outcome for most mathematicians: Hoping for a proof that would yield some insight into why the result was true, they instead got a case analysis of enormous complexity whose proof could not be checked by hand. The problem of finding a reasonably short, human-readable proof still remains open.

8.8 Numerical Problems

We now consider some computationally hard problems that involve arithmetic operations on numbers. We will see that the intractability here comes from the way in which some of the problems we have seen earlier in the chapter can be encoded in the representations of very large integers.

The Subset Sum Problem

Our basic problem in this genre will be *Subset Sum*, a special case of the Knapsack Problem that we saw before in Section 6.4 when we covered dynamic programming. We can formulate a decision version of this problem as follows.

> *Given natural numbers $w_1, \ldots, w_n$, and a target number W, is there a subset of $\{w_1, \ldots, w_n\}$ that adds up to precisely W?*

We have already seen an algorithm to solve this problem; why are we now including it on our list of computationally hard problems? This goes back to an issue that we raised the first time we considered Subset Sum in Section 6.4. The algorithm we developed there has running time $O(nW)$, which is reasonable when W is small, but becomes hopelessly impractical as W (and the numbers w_i) grow large. Consider, for example, an instance with 100 numbers, each of which is 100 bits long. Then the input is only $100 \times 100 = 10{,}000$ digits, but W is now roughly 2^{100}.

To phrase this more generally, since integers will typically be given in bit representation, or base-10 representation, the quantity W is really *exponential* in the size of the input; our algorithm was not a polynomial-time algorithm. (We referred to it as *pseudo-polynomial*, to indicate that it ran in time polynomial in the magnitude of the input numbers, but not polynomial in the size of their representation.)

This is an issue that comes up in many settings; for example, we encountered it in the context of network flow algorithms, where the capacities had integer values. Other settings may be familiar to you as well. For example, the security of a cryptosystem such as RSA is motivated by the sense that factoring a 1,000-bit number is difficult. But if we considered a running time of 2^{1000} steps feasible, factoring such a number would not be difficult at all.

It is worth pausing here for a moment and asking: Is this notion of polynomial time for numerical operations too severe a restriction? For example, given two natural numbers w_1 and w_2 represented in base-d notation for some $d > 1$, how long does it take to add, subtract, or multiply them? This is an issue we touched on in Section 5.5, where we noted that the standard ways that kids in elementary school learn to perform these operations have (low-degree) polynomial running times. Addition and subtraction (with carries) take $O(\log w_1 + \log w_2)$ time, while the standard multiplication algorithm runs in $O(\log w_1 \cdot \log w_2)$ time. (Recall that in Section 5.5 we discussed the design of an asymptotically faster multiplication algorithm that elementary schoolchildren are unlikely to invent on their own.)

So a basic question is: Can Subset Sum be solved by a (genuinely) polynomial-time algorithm? In other words, could there be an algorithm with running time polynomial in n and $\log W$? Or polynomial in n alone?

Proving Subset Sum Is NP-Complete

The following result suggests that this is not likely to be the case.

(8.23) Subset Sum *is NP-complete.*

Proof. We first show that Subset Sum is in $\mathcal{NP}$. Given natural numbers $w_1, \ldots, w_n$, and a target W, a certificate that there is a solution would be the subset $w_{i_1}, \ldots, w_{i_k}$ that is purported to add up to W. In polynomial time, we can compute the sum of these numbers and verify that it is equal to W.

We now reduce a known NP-complete problem to Subset Sum. Since we are seeking a set that adds up to *exactly* a given quantity (as opposed to being bounded above or below by this quantity), we look for a combinatorial problem that is based on meeting an *exact* bound. The 3-Dimensional Matching Problem is a natural choice; we show that 3-Dimensional Matching $\leq_P$ Subset Sum. The trick will be to encode the manipulation of sets via the addition of integers.

So consider an instance of 3-Dimensional Matching specified by sets X, Y, Z, each of size n, and a set of m triples $T \subseteq X \times Y \times Z$. A common way to represent sets is via *bit-vectors*: Each entry in the vector corresponds to a different element, and it holds a 1 if and only if the set contains that element. We adopt this type of approach for representing each triple $t = (x_i, y_j, z_k) \in T$: we construct a number w_t with $3n$ digits that has a 1 in positions i, $n + j$, and $2n + k$, and a 0 in all other positions. In other words, for some base $d > 1$, $w_t = d^{i-1} + d^{n+j-1} + d^{2n+k-1}$.

Note how taking the union of triples *almost* corresponds to integer addition: The 1s fill in the places where there is an element in any of the sets. But we say *almost* because addition includes *carries*: too many 1s in the same column will "roll over" and produce a nonzero entry in the next column. This has no analogue in the context of the union operation.

In the present situation, we handle this problem by a simple trick. We have only m numbers in all, and each has digits equal to 0 or 1; so if we assume that our numbers are written in base $d = m + 1$, then there will be no carries at all.

Thus we construct the following instance of Subset Sum. For each triple $t = (x_i, y_j, z_k) \in T$, we construct a number w_t in base $m + 1$ as defined above. We define W to be the number in base $m + 1$ with $3n$ digits, each of which is equal to 1, that is, $W = \sum_{i=0}^{3n-1} (m + 1)^i$.

We claim that the set T of triples contains a perfect three-dimensional matching if and only if there is a subset of the numbers $\{w_t\}$ that adds up to W. For suppose there is a perfect three-dimensional matching consisting of

triples $t_1, \ldots, t_n$. Then in the sum $w_{t_1} + \cdots + w_{t_n}$, there is a single 1 in each of the $3n$ digit positions, and so the result is equal to W.

Conversely, suppose there exists a set of numbers $w_{t_1}, \ldots, w_{t_k}$ that adds up to W. Then since each w_{t_i} has three 1s in its representation, and there are no carries, we know that $k = n$. It follows that for each of the $3n$ digit positions, exactly one of the w_{t_i} has a 1 in that position. Thus, $t_1, \ldots, t_k$ constitute a perfect three-dimensional matching. ■

Extensions: The Hardness of Certain Scheduling Problems

The hardness of Subset Sum can be used to establish the hardness of a range of scheduling problems—including some that do not obviously involve the addition of numbers. Here is a nice example, a natural (but much harder) generalization of a scheduling problem we solved in Section 4.2 using a greedy algorithm.

Suppose we are given a set of n jobs that must be run on a single machine. Each job i has a *release time* r_i when it is first available for processing; a *deadline* d_i by which it must be completed; and a *processing duration* t_i. We will assume that all of these parameters are natural numbers. In order to be completed, job i must be allocated a contiguous slot of t_i time units somewhere in the interval $[r_i, d_i]$. The machine can run only one job at a time. The question is: Can we schedule all jobs so that each completes by its deadline? We will call this an instance of *Scheduling with Release Times and Deadlines*.

(8.24) Scheduling with Release Times and Deadlines *is NP-complete.*

Proof. Given an instance of the problem, a certificate that it is solvable would be a specification of the starting time for each job. We could then check that each job runs for a distinct interval of time, between its release time and deadline. Thus the problem is in $\mathcal{NP}$.

We now show that Subset Sum is reducible to this scheduling problem. Thus, consider an instance of Subset Sum with numbers $w_1, \ldots, w_n$ and a target W. In constructing an equivalent scheduling instance, one is struck initially by the fact that we have so many parameters to manage: release times, deadlines, and durations. The key is to sacrifice most of this flexibility, producing a "skeletal" instance of the problem that still encodes the Subset Sum Problem.

Let $S = \sum_{i=1}^{n} w_i$. We define jobs $1, 2, \ldots, n$; job i has a release time of 0, a deadline of $S + 1$, and a duration of w_i. For this set of jobs, we have the freedom to arrange them in any order, and they will all finish on time.

We now further constrain the instance so that the only way to solve it will be to group together a subset of the jobs whose durations add up precisely to W. To do this, we define an $(n+1)^{\text{st}}$ job; it has a release time of W, a deadline of $W+1$, and a duration of 1.

Now consider any feasible solution to this instance of the scheduling problem. The $(n+1)^{\text{st}}$ job must be run in the interval $[W, W+1]$. This leaves S available time units between the common release time and the common deadline; and there are S time units worth of jobs to run. Thus the machine must not have any idle time, when no jobs are running. In particular, if jobs $i_1, \ldots, i_k$ are the ones that run before time W, then the corresponding numbers $w_{i_1}, \ldots, w_{i_k}$ in the Subset Sum instance add up to exactly W.

Conversely, if there are numbers $w_{i_1}, \ldots, w_{i_k}$ that add up to exactly W, then we can schedule these before job $n+1$ and the remainder after job $n+1$; this is a feasible solution to the scheduling instance. ■

Caveat: Subset Sum with Polynomially Bounded Numbers

There is a very common source of pitfalls involving the Subset Sum Problem, and while it is closely connected to the issues we have been discussing already, we feel it is worth discussing explicitly. The pitfall is the following.

> *Consider the special case of Subset Sum, with n input numbers, in which W is bounded by a polynomial function of n. Assuming $\mathcal{P} \neq \mathcal{NP}$, this special case is* not *NP-complete.*

It is not NP-complete for the simple reason that it can be solved in time $O(nW)$, by our dynamic programming algorithm from Section 6.4; when W is bounded by a polynomial function of n, this is a polynomial-time algorithm.

All this is very clear; so you may ask: Why dwell on it? The reason is that there is a genre of problem that is often wrongly claimed to be NP-complete (even in published papers) via reduction from this special case of Subset Sum. Here is a basic example of such a problem, which we will call *Component Grouping*.

> *Given a graph G that is not connected, and a number k, does there exist a subset of its connected components whose union has size exactly k?*

Incorrect Claim. Component Grouping is NP-complete.

Incorrect Proof. Component Grouping is in $\mathcal{NP}$, and we'll skip the proof of this. We now attempt to show that Subset Sum $\leq_P$ Component Grouping. Given an instance of Subset Sum with numbers $w_1, \ldots, w_n$ and target W, we construct an instance of Component Grouping as follows. For each i, we construct a path P_i of length w_i. The graph G will be the union of the paths

$P_1, \ldots, P_n$, each of which is a separate connected component. We set $k = W$. It is clear that G has a set of connected components whose union has size k if and only if some subset of the numbers $w_1, \ldots, w_n$ adds up to W. ■

The error here is subtle; in particular, the claim in the last sentence is correct. The problem is that the construction described above does not establish that Subset Sum $\leq_P$ Component Grouping, because it requires more than polynomial time. In constructing the input to our black box that solves Component Grouping, we had to build the encoding of a graph of size $w_1 + \cdots + w_n$, and this takes time exponential in the size of the input to the Subset Sum instance. In effect, Subset Sum works with the numbers $w_1, \ldots, w_n$ in a very compact representation, but Component Grouping does not accept "compact" encodings of graphs.

The problem is more fundamental than the incorrectness of this proof; in fact, Component Grouping is a problem that can be solved in polynomial time. If $n_1, n_2, \ldots, n_c$ denote the sizes of the connected components of G, we simply use our dynamic programming algorithm for Subset Sum to decide whether some subset of these numbers $\{n_i\}$ adds up to k. The running time required for this is $O(ck)$; and since c and k are both bounded by n, this is $O(n^2)$ time.

Thus we have discovered a new polynomial-time algorithm by reducing in the other direction, to a polynomial-time solvable special case of Subset Sum.

8.9 Co-NP and the Asymmetry of NP

As a further perspective on this general class of problems, let's return to the definitions underlying the class $\mathcal{NP}$. We've seen that the notion of an efficient certifier doesn't suggest a concrete algorithm for actually solving the problem that's better than brute-force search.

Now here's another observation: The definition of efficient certification, and hence of $\mathcal{NP}$, is fundamentally *asymmetric*. An input string s is a "yes" instance if and only if there exists a short t so that $B(s, t) = \texttt{yes}$. Negating this statement, we see that an input string s is a "no" instance if and only if *for all* short t, it's the case that $B(s, t) = \texttt{no}$.

This relates closely to our intuition about $\mathcal{NP}$: When we have a "yes" instance, we can provide a short proof of this fact. But when we have a "no" instance, no correspondingly short proof is guaranteed by the definition; the answer is no simply because there is no string that will serve as a proof. In concrete terms, recall our question from Section 8.3: Given an unsatisfiable set of clauses, what evidence could we show to quickly convince you that there is no satisfying assignment?

For every problem X, there is a natural *complementary* problem $\overline{X}$: For all input strings s, we say $s \in \overline{X}$ if and only if $s \notin X$. Note that if $X \in \mathcal{P}$, then $\overline{X} \in \mathcal{P}$, since from an algorithm A that solves X, we can simply produce an algorithm $\overline{A}$ that runs A and then flips its answer.

But it is far from clear that if $X \in \mathcal{NP}$, it should follow that $\overline{X} \in \mathcal{NP}$. The problem $\overline{X}$, rather, has a different property: for all s, we have $s \in \overline{X}$ if and only if for all t of length at most $p(|s|)$, $B(s, t) = \texttt{no}$. This is a fundamentally different definition, and it can't be worked around by simply "inverting" the output of the efficient certifier B to produce $\overline{B}$. The problem is that the "exists t" in the definition of $\mathcal{NP}$ has become a "for all t," and this is a serious change.

There is a class of problems parallel to $\mathcal{NP}$ that is designed to model this issue; it is called, naturally enough, co-$\mathcal{NP}$. A problem X belongs to co-$\mathcal{NP}$ if and only if the complementary problem $\overline{X}$ belongs to $\mathcal{NP}$. We do not know for sure that $\mathcal{NP}$ and co-$\mathcal{NP}$ are different; we can only ask

(8.25) *Does* $\mathcal{NP} = \text{co-}\mathcal{NP}$*?*

Again, the widespread belief is that $\mathcal{NP} \neq \text{co-}\mathcal{NP}$: Just because the "yes" instances of a problem have short proofs, it is not clear why we should believe that the "no" instances have short proofs as well.

Proving $\mathcal{NP} \neq \text{co-}\mathcal{NP}$ would be an even bigger step than proving $\mathcal{P} \neq \mathcal{NP}$, for the following reason:

(8.26) *If* $\mathcal{NP} \neq \text{co-}\mathcal{NP}$*, then* $\mathcal{P} \neq \mathcal{NP}$*.*

Proof. We'll actually prove the contrapositive statement: $\mathcal{P} = \mathcal{NP}$ implies $\mathcal{NP} = \text{co-}\mathcal{NP}$. Essentially, the point is that $\mathcal{P}$ is closed under complementation; so if $\mathcal{P} = \mathcal{NP}$, then $\mathcal{NP}$ would be closed under complementation as well. More formally, starting from the assumption $\mathcal{P} = \mathcal{NP}$, we have

$$X \in \mathcal{NP} \Longrightarrow X \in \mathcal{P} \Longrightarrow \overline{X} \in \mathcal{P} \Longrightarrow \overline{X} \in \mathcal{NP} \Longrightarrow X \in \text{co-}\mathcal{NP}$$

and

$$X \in \text{co-}\mathcal{NP} \Longrightarrow \overline{X} \in \mathcal{NP} \Longrightarrow \overline{X} \in \mathcal{P} \Longrightarrow X \in \mathcal{P} \Longrightarrow X \in \mathcal{NP}.$$

Hence it would follow that $\mathcal{NP} \subseteq \text{co-}\mathcal{NP}$ and co-$\mathcal{NP} \subseteq \mathcal{NP}$, whence $\mathcal{NP} = \text{co-}\mathcal{NP}$. ■

Good Characterizations: The Class NP ∩ co-NP

If a problem X belongs to both $\mathcal{NP}$ and co-$\mathcal{NP}$, then it has the following nice property: When the answer is yes, there is a short proof; and when the answer is no, there is also a short proof. Thus problems that belong to this intersection

$\mathcal{NP} \cap$ co-$\mathcal{NP}$ are said to have a *good characterization*, since there is always a nice certificate for the solution.

This notion corresponds directly to some of the results we have seen earlier. For example, consider the problem of determining whether a flow network contains a flow of value at least ν, for some quantity ν. To prove that the answer is yes, we could simply exhibit a flow that achieves this value; this is consistent with the problem belonging to $\mathcal{NP}$. But we can also prove the answer is no: We can exhibit a cut whose capacity is strictly less than ν. This duality between "yes" and "no" instances is the crux of the Max-Flow Min-Cut Theorem.

Similarly, Hall's Theorem for matchings from Section 7.5 proved that the Bipartite Perfect Matching Problem is in $\mathcal{NP} \cap$ co-$\mathcal{NP}$: We can exhibit either a perfect matching, or a set of vertices $A \subseteq X$ such that the total number of neighbors of A is strictly less than $|A|$.

Now, if a problem X is in $\mathcal{P}$, then it belongs to both $\mathcal{NP}$ and co-$\mathcal{NP}$; thus, $\mathcal{P} \subseteq \mathcal{NP} \cap$ co-$\mathcal{NP}$. Interestingly, both our proof of the Max-Flow Min-Cut Theorem and our proof of Hall's Theorem came hand in hand with proofs of the stronger results that Maximum Flow and Bipartite Matching are problems in $\mathcal{P}$. Nevertheless, the good characterizations themselves are so clean that formulating them separately still gives us a lot of conceptual leverage in reasoning about these problems.

Naturally, one would like to know whether there's a problem that has a good characterization but no polynomial-time algorithm. But this too is an open question:

(8.27) *Does* $\mathcal{P} = \mathcal{NP} \cap$ co-$\mathcal{NP}$*?*

Unlike questions (8.11) and (8.25), general opinion seems somewhat mixed on this one. In part, this is because there are many cases in which a problem was found to have a nontrivial good characterization; and then (sometimes many years later) it was also discovered to have a polynomial-time algorithm.

8.10 A Partial Taxonomy of Hard Problems

We've now reached the end of the chapter, and we've encountered a fairly rich array of NP-complete problems. In a way, it's useful to know a good number of different NP-complete problems: When you encounter a new problem X and want to try proving it's NP-complete, you want to show $Y \leq_P X$ for some known NP-complete problem Y—so the more options you have for Y, the better.

At the same time, the more options you have for Y, the more bewildering it can be to try choosing the right one to use in a particular reduction. Of course, the whole point of NP-completeness is that one of these problems will work in your reduction if and only if any of them will (since they're all equivalent with respect to polynomial-time reductions); but the reduction to a given problem X can be much, much easier starting from some problems than from others.

With this in mind, we spend this concluding section on a review of the NP-complete problems we've come across in the chapter, grouped into six basic genres. Together with this grouping, we offer some suggestions as to how to choose a starting problem for use in a reduction.

Packing Problems

Packing problems tend to have the following structure: You're given a collection of objects, and you want to choose at least k of them; making your life difficult is a set of conflicts among the objects, preventing you from choosing certain groups simultaneously.

We've seen two basic packing problems in this chapter.

- *Independent Set*: Given a graph G and a number k, does G contain an independent set of size at least k?
- *Set Packing*: Given a set U of n elements, a collection $S_1, \ldots, S_m$ of subsets of U, and a number k, does there exist a collection of at least k of these sets with the property that no two of them intersect?

Covering Problems

Covering problems form a natural contrast to packing problems, and one typically recognizes them as having the following structure: you're given a collection of objects, and you want to choose a subset that collectively achieves a certain goal; the challenge is to achieve this goal while choosing only k of the objects.

We've seen two basic covering problems in this chapter.

- *Vertex Cover*: Given a graph G and a number k, does G contain a vertex cover of size at most k?
- *Set Cover*: Given a set U of n elements, a collection $S_1, \ldots, S_m$ of subsets of U, and a number k, does there exist a collection of at most k of these sets whose union is equal to all of U?

Partitioning Problems

Partitioning problems involve a search over all ways to divide up a collection of objects into subsets so that each object appears in exactly one of the subsets.

One of our two basic partitioning problems, 3-Dimensional Matching, arises naturally whenever you have a collection of sets and you want to solve a covering problem and a packing problem simultaneously: Choose some of the sets in such a way that they are disjoint, yet completely cover the ground set.

- *3-Dimensional Matching*: Given disjoint sets X, Y, and Z, each of size n, and given a set $T \subseteq X \times Y \times Z$ of ordered triples, does there exist a set of n triples in T so that each element of $X \cup Y \cup Z$ is contained in exactly one of these triples?

Our other basic partitioning problem, Graph Coloring, is at work whenever you're seeking to partition objects in the presence of conflicts, and conflicting objects aren't allowed to go into the same set.

- *Graph Coloring*: Given a graph G and a bound k, does G have a k-coloring?

Sequencing Problems

Our first three types of problems have involved searching over subsets of a collection of objects. Another type of computationally hard problem involves searching over the set of all *permutations* of a collection of objects.

Two of our basic sequencing problems draw their difficulty from the fact that you are required to order n objects, but there are restrictions preventing you from placing certain objects after certain others.

- *Hamiltonian Cycle*: Given a directed graph G, does it contain a Hamiltonian cycle?
- *Hamiltonian Path*: Given a directed graph G, does it contain a Hamiltonian path?

Our third basic sequencing problem is very similar; it softens these restrictions by simply imposing a cost for placing one object after another.

- *Traveling Salesman*: Given a set of distances on n cities, and a bound D, is there a tour of length at most D?

Numerical Problems

The hardness of the numerical problems considered in this chapter flowed principally from Subset Sum, the special case of the Knapsack Problem that we considered in Section 8.8.

- *Subset Sum*: Given natural numbers $w_1, \ldots, w_n$, and a target number W, is there a subset of $\{w_1, \ldots, w_n\}$ that adds up to precisely W?

It is natural to try reducing from *Subset Sum* whenever one has a problem with weighted objects and the goal is to select objects conditioned on a constraint on

the total weight of the objects selected. This, for example, is what happened in the proof of (8.24), showing that Scheduling with Release Times and Deadlines is NP-complete.

At the same time, one must heed the warning that Subset Sum only becomes hard with truly large integers; when the magnitudes of the input numbers are bounded by a polynomial function of n, the problem is solvable in polynomial time by dynamic programming.

Constraint Satisfaction Problems

Finally, we considered basic constraint satisfaction problems, including Circuit Satisfiability, SAT, and 3-SAT. Among these, the most useful for the purpose of designing reductions is 3-SAT.

- *3-SAT*: Given a set of clauses $C_1, \ldots, C_k$, each of length 3, over a set of variables $X = \{x_1, \ldots, x_n\}$, does there exist a satisfying truth assignment?

Because of its expressive flexibility, 3-SAT is often a useful starting point for reductions where none of the previous five categories seem to fit naturally onto the problem being considered. In designing 3-SAT reductions, it helps to recall the advice given in the proof of (8.8), that there are two distinct ways to view an instance of 3-SAT: (a) as a search over assignments to the variables, subject to the constraint that all clauses must be satisfied, and (b) as a search over ways to choose a single term (to be satisfied) from each clause, subject to the constraint that one mustn't choose conflicting terms from different clauses. Each of these perspectives on 3-SAT is useful, and each forms the key idea behind a large number of reductions.

Solved Exercises

Solved Exercise 1

You're consulting for a small high-tech company that maintains a high-security computer system for some sensitive work that it's doing. To make sure this system is not being used for any illicit purposes, they've set up some logging software that records the IP addresses that all their users are accessing over time. We'll assume that each user accesses at most one IP address in any given minute; the software writes a log file that records, for each user u and each minute m, a value $I(u, m)$ that is equal to the IP address (if any) accessed by user u during minute m. It sets $I(u, m)$ to the null symbol $\perp$ if u did not access any IP address during minute m.

The company management just learned that yesterday the system was used to launch a complex attack on some remote sites. The attack was carried out by accessing t distinct IP addresses over t consecutive minutes: In minute