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.

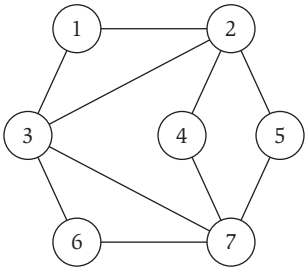

Figure 8.1 A graph whose largest independent set has size 4, and whose smallest vertex cover has size 3.

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

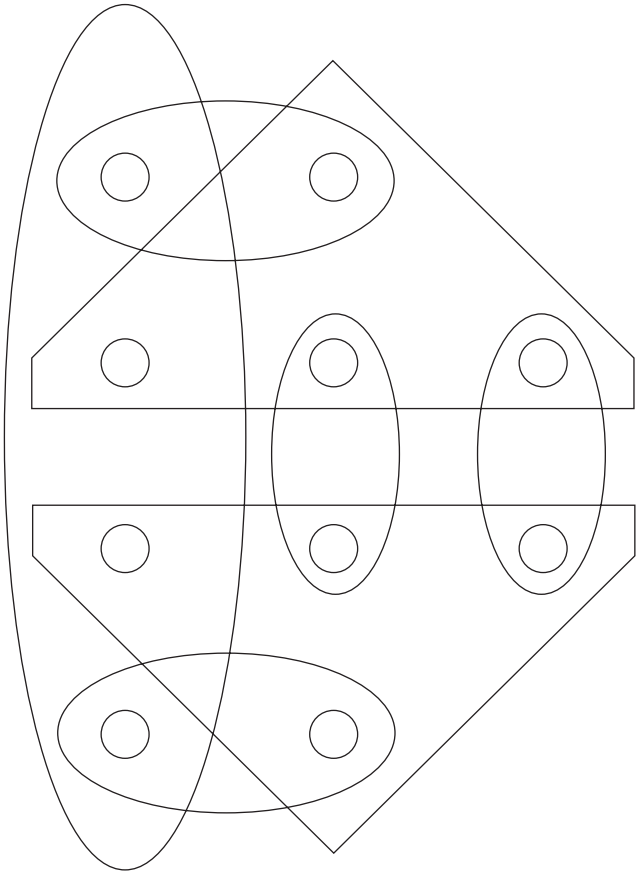

Figure 8.2 An instance of the Set Cover Problem.

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 $\nu : X \to \{0, 1\}$. The assignment ν 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 ν 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.

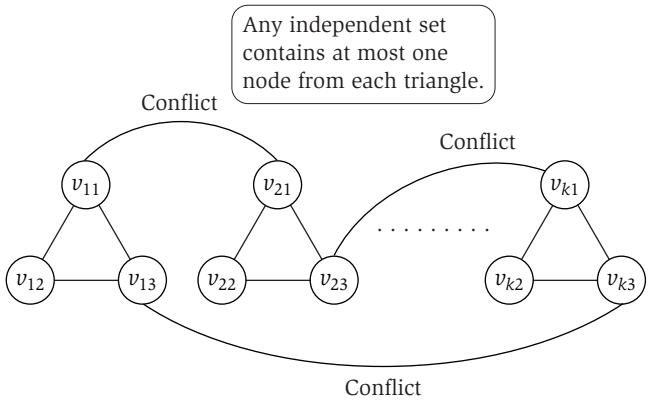

Figure 8.3 The reduction from 3-SAT to Independent Set.

- 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