CHAPTER 9

NP-COMPLETE PROBLEMS

Proving NP-completeness results is an important ingredient of our methodology for studying computational problems. It is also something of an art form.

9.1 PROBLEMS IN NP

We have defined **NP** as the class of languages decided by nondeterministic Turing machines in polynomial time. We shall next show an alternative way of looking at **NP**, somewhat akin to its characterization in terms of existential second-order logic (Theorem 8.3). Let $R \subseteq \Sigma^* \times \Sigma^*$ be a binary relation on strings. R is called *polynomially decidable* if there is a deterministic Turing machine deciding the language $\{x;y : (x,y) \in R\}$ in polynomial time. We say that R is *polynomially balanced* if $(x,y) \in R$ implies $|y| \leq |x|^k$ for some $k \geq 1$. That is, the length of the second component is always bounded by a polynomial in the length of the first (the other way is not important).

Proposition 9.1: Let $L \subseteq \Sigma^*$ be a language. $L \in$ **NP** if and only if there is a polynomially decidable and polynomially balanced relation R, such that $L = \{x : (x,y) \in R \text{ for some } y\}$.

Proof: Suppose that such an R exists. Then L is decided by the following nondeterministic machine M: On input x, M guesses a y of length at most $|x|^k$ (the polynomial balance bound for R), and then uses the polynomial algorithm on $x;y$ to test whether $(x,y) \in R$. If so, it accepts, otherwise it rejects. It is immediate that an accepting computation exists if and only if $x \in L$.

Conversely, suppose that $L \in$ **NP**; that is, there is a nondeterministic Turing machine N that decides L in time $|x|^k$, for some k. Define the following relation R: $(x,y) \in R$ if and only if y is the encoding of an accepting

computation of N on input x. It is clear that R is polynomially balanced (since N is polynomially bounded), and polynomially decidable (since it can be checked in linear time whether y indeed encodes an accepting computation of N on x). Furthermore, by our assumption that N decides L, we have that $L = \{x : (x, y) \in R \text{ for some } y\}$. □

Proposition 9.1 is perhaps the most intuitive way of understanding **NP**. Each problem in **NP** has a remarkable property: Any "yes" instance x of the problem has at least one *succinct certificate* (or *polynomial witness*) y of its being a "yes" instance. Naturally, "no" instances possess no such certificates. We may not know how to discover this certificate in polynomial time, but we are sure it exists if the instance is a "yes" instance. For SAT, the certificate of a Boolean expression ϕ is a truth assignment T that satisfies ϕ. T is succinct relative to ϕ (it assigns truth values to variables appearing in ϕ), and it exists if and only if the expression is satisfiable. In HAMILTON PATH, the certificate of a graph G is precisely a Hamilton path of G.

It is now easy to explain why **NP** is inhabited by such a tremendous wealth of practically important, natural computational problems (see the problems mentioned in this chapter, and the references). Many computational problems in several application areas call for the design of mathematical objects of various sorts (paths, truth assignments, solutions of equations, register allocations, traveling salesman routes, VLSI layouts, and so on). Sometimes we seek the optimum among all possible alternatives, and sometimes we are satisfied with any object that fits the design specifications (and we have seen in the example of TSP (D) that optimality can be couched in terms of constraint satisfaction by adding a "budget" to the problem). The object sought is thus the "certificate" that shows the problem is in **NP**. Often certificates are mathematical abstractions of actual, physical objects or real-life plans that will ultimately be constructed or implemented. Hence, it is only natural that in most applications the certificates are not astronomically large, in terms of the input data. And specifications are usually simple, checkable in polynomial time. One should therefore expect that most problems arising in computational practice are in **NP**.

And in fact they are. Although in later chapters we shall see several practically important, natural problems that are not believed to be in **NP**, such problems are not the rule. The study of the complexity of computational problems concerns itself for the most part with specimens in **NP**; it basically tries to sort out which of these problems can be solved in polynomial time, and which cannot. In this context, **NP**-completeness is a most important tool. Showing that the problem being studied is **NP**-complete establishes that it is among the least likely to be in **P**, those that can be solved in polynomial time only if $\mathbf{P} = \mathbf{NP}$. In this sense, **NP**-completeness is a valuable component of our methodology, one that complements algorithm design techniques. An aspect of

its importance is this: Once our problem has been shown **NP**-complete, it seems reasonable† to direct our efforts to the many *alternative approaches* available for such problems: Developing approximation algorithms, attacking special cases, studying the average performance of algorithms, developing randomized algorithms, designing exponential algorithms that are practical for small instances, resorting to local search and other heuristics, and so on. Many of these approaches are integral parts of the theory of algorithms and complexity (see the references and later chapters), and owe their existence and flourish precisely to **NP**-completeness.

9.2 VARIANTS OF SATISFIABILITY

Any computational problem, if generalized enough, will become **NP**-complete or worse. And any problem has special cases that are in **P**. The interesting part is to find the dividing line. SAT provides a very interesting example, pursued in this section.

There are many ways and styles for proving a special case of an **NP**-complete problem to be **NP**-complete. The simplest one (and perhaps the most useful) is when we just have to observe that the reduction we already know creates instances belonging to the special case considered. For example, let kSAT, where $k \geq 1$ is an integer, be the special case of SAT in which the formula is in conjunctive normal form, and all clauses have k literals.

Proposition 9.2: 3SAT is **NP**-complete.

Proof: Just notice that the reduction in Theorem 8.2 and Example 8.3 produces such expressions. Clauses with one or two literals can be made into equivalent clauses with three literals by duplicating a literal in them once or twice (for a direct reduction from SAT to 3SAT see Problem 9.5.2). □

Notice that in our variants of the satisfiability problem we allow repetitions of literals in the clauses. This is reasonable and simplifying, especially since our reductions from these problems do not assume that the literals in a clause are distinct. Naturally enough, 3SAT remains **NP**-complete even if all literals in a clause are required to be distinct (see Problem 9.5.5). In another front, satisfiability remains **NP**-complete even if we also bound the number of *occurrences* of the variables in the expression.

Proposition 9.3: 3SAT remains **NP**-complete even for expressions in which each each variable is restricted to appear at most three times, and each literal at most twice.

Proof: This is a special kind of a reduction, in which we must show that a

† There is nothing wrong with trying to prove that $\mathbf{P} = \mathbf{NP}$ by developing a polynomial-time algorithm for an **NP**-complete problem. The point is that without an **NP**-completeness proof we would be trying the same thing *without knowing it!*

problem remains **NP**-complete even when the instances are somehow restricted. We accomplish this by showing how to rewrite any instance so that the "undesirable features" of the instance (those that are forbidden in the restriction) go away. In the present case, the undesirable features are variables that appear many times. Consider such a variable x, appearing k times in the expression. We replace the first occurrence of x by x_1, the second by x_2, and so on, where $x_1, x_2, \ldots, x_k$ are k new variables. We must now somehow make sure that these k variables take the same truth value. It is easy to see that this is achieved by adding to our expression the clauses $(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge \ldots \wedge (\neg x_k \vee x_1)$. □

Notice however that, in order to achieve the restrictions of Proposition 9.3, we had to abandon our requirement that all clauses have *exactly* three literals; the reason behind this retreat is spelled out in Problem 9.5.4.

In analyzing the complexity of a problem, we are trying to define the precise boundary between the polynomial and **NP**-complete cases (although we should not be overconfident that such a boundary necessarily exists, see Section 14.1). For SAT this boundary is well-understood, at least along the dimension of literals per clause: We next show that 2SAT is in **P**. (For the boundary in terms of number of occurrences of variables, in the sense of Proposition 9.2, see Problem 9.5.4; the dividing line is again between two and three!)

Let ϕ be an instance of 2SAT, that is, a set of clauses with two literals each. We can define a graph $G(\phi)$ as follows: The vertices of G are the variables of ϕ and their negations; and there is an arc (α, β) if and only if there is a clause $(\neg\alpha \vee \beta)$ (or $(\beta \vee \neg\alpha)$) in ϕ. Intuitively, these edges capture the logical implications ($\Rightarrow$) of ϕ. As a result, $G(\phi)$ has a curious symmetry: If (α, β) is an edge, then so is $(\neg\beta, \neg\alpha)$; see Figure 9.1 for an example. Paths in $G(\phi)$ are also valid implications (by the transitivity of $\Rightarrow$). We can show the following:

Theorem 9.1: ϕ is unsatisfiable if and only if there is a variable x such that there are paths from x to $\neg x$ and from $\neg x$ to x in $G(\phi)$.

Proof: Suppose that such paths exist, and still ϕ can be satisfied by a truth assignment T. Suppose that $T(x) =$ **true** (a similar argument works when $T(x) =$ **false**). Since there is a path from x to $\neg x$, and $T(x) =$ **true** while $T(\neg x) =$ **false**, there must be an edge (α, β) along this path such that $T(\alpha) =$ **true** and $T(\beta) =$ **false**. However, since (α, β) is an edge of $G(\phi)$, it follows that $(\neg\alpha \vee \beta)$ is a clause of ϕ. This clause is not satisfied by T, a contradiction.

Conversely, suppose that there is no variable with such paths in $G(\phi)$. We are going to construct a satisfying truth assignment, that is, a truth assignment such that no edge of $G(\phi)$ goes from **true** to **false**. We repeat the following step: We pick a node α whose truth value has not yet been defined, and such that there is no path from α to $\neg\alpha$. We consider all nodes reachable from α in $G(\phi)$, and assign them the value **true**. We also assign **false** to the negations of these nodes (the negations correspond to all these nodes from which $\neg\alpha$ is

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

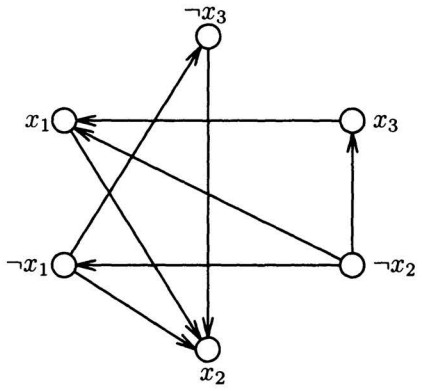

Figure 9-1. The algorithm for 2SAT.

reachable). This step is well-defined because, if there were paths from α to both β and $\neg\beta$, then there would be paths to $\neg\alpha$ from both of these (by the symmetry of $G(\phi)$), and therefore a path from α to $\neg\alpha$, a contradiction of our hypothesis. Furthermore, if there were a path from α to a node already assigned **false** in a previous step, then α is a predecessor of that node, and was also assigned **false** at that step.

We repeat this step until all nodes have a truth assignment. Since we assumed that there are no paths from any x to $\neg x$ and back, all nodes will be assigned a truth value. And since the steps are such that, whenever a node is assigned **true** all of its successors are also assigned **true**, and the opposite for **false**, there can be no edge from **true** to **false**. The truth assignment satisfies ϕ. □

Corollary: 2SAT is in **NL** (and therefore in **P**).

Proof: Since **NL** is closed under complement (Theorem 7.6), we need to to show that we can recognize *unsatisfiable* expressions in **NL**. In nondeterministic logarithmic space we can test the condition of the Theorem by guessing a variable x, and paths from x to $\neg x$ and back. □

A polynomial algorithm, like the one for 2SAT we just described, is not out of place in an **NP**-completeness chapter. Exploring the complexity of a problem typically involves switching back and forth between trying to develop a polynomial algorithm for the problem and trying to prove it **NP**-complete, until one of the approaches succeeds. Incidentally, recall that HORNSAT is another polynomial-time solvable special case of SAT (Theorem 4.2).

It is easy to see that 3SAT is a generalization of 2SAT: 2SAT can be thought of as the special case of 3SAT in which among the three literals in each clause there are at most two distinct ones (remember, we allow repetitions of literals in clauses). We already know that this generalization leads to an **NP**-complete problem. But we can generalize 2SAT in a different direction: 2SAT requires that *all* clauses be satisfied. It would be reasonable to ask whether there is a truth assignment that satisfies not necessarily all clauses, but some large number of them. That is, we are given a set of clauses, each with two literals in it, and an integer K; we are asked whether there is a truth assignment that satisfies at least K of the clauses. We call this problem MAX2SAT; it is obviously an optimization problem, turned into a "yes-no" problem by adding a *goal* K—the counterpart of a budget in maximization problems. MAX2SAT is a generalization of 2SAT, since 2SAT is the special case in which K equals the number of clauses. Notice also that we did not bother to define MAXSAT for clauses with three or more literals, since such a problem would be trivially **NP**-complete—it generalizes the **NP**-complete problem 3SAT.

It turns out that, by generalizing 2SAT to MAX2SAT, we have crossed the **NP**-completeness boundary once more:

Theorem 9.2: MAX2SAT is **NP**-complete.

Proof: Consider the following ten clauses:

$$(x)(y)(z)(w)$$
$$(\neg x \vee \neg y)(\neg y \vee \neg z)(\neg z \vee \neg x)$$
$$(x \vee \neg w)(y \vee \neg w)(z \vee \neg w)$$

There is no way to satisfy all these clauses (for example, to satisfy all clauses in the first row we must lose all clauses in the second). But how many *can* we satisfy? Notice first that the clauses are symmetric with respect to x, y, and z (but not w). So, asume that all three of x, y, z are **true**. Then the second row is lost, and we can get all the rest by setting w to **true**. If just two of x, y, z are **true**, then we lose a clause from the first row, and one clause from the second row. Then we have a choice: If we set w to **true**, we get one extra clause from the first row; if we set it to **false**, we get one from the third. So, we can again satisfy seven clauses, and no more. If only one of x, y, z is **true**, then we have one clause from the first row and the whole second row. For the third row, we can satisfy all three clauses by setting w to **false**, but then we lose (w). The maximum is again seven. However, suppose that all three are **false**. Then it is easy to see that we can satisfy at most *six* clauses: The second and third row.

In other words, these ten clauses have the following interesting property: Any truth assignment that satisfies $(x \vee y \vee z)$ can be extended to satisfy seven of them and no more, while the remaining truth assignment can be extended to satisfy only six of them. This suggests an immediate reduction from 3SAT

to MAX2SAT: Given any instance ϕ of 3SAT, we construct an instance $R(\phi)$ of MAX2SAT as follows: For each clause $C_i = (\alpha \vee \beta \vee \gamma)$ of ϕ we add to $R(\phi)$ the ten clauses above, *with* α, β, *and* γ *replacing* x, y, *and* z; w is replaced by a new variable w_i, particular to C_i. We call the ten clauses of $R(\phi)$ corresponding to a clause of ϕ a *group*. If ϕ has m clauses, then obviously $R(\phi)$ has $10m$. The goal is set at $K = 7m$.

We claim that the goal can be achieved in $R(\phi)$ if and only if ϕ is satisfiable. Suppose that $7m$ clauses can be satisfied in $R(\phi)$. Since we know that in each group we can satisfy at most seven clauses, and there are m groups, seven clauses must be satisfied in each group. However, such an assignment would satisfy all clauses in ϕ. Conversely, any assignment that satisfies all clauses in ϕ can be turned into one that satisfies $7m$ clauses in $R(\phi)$ by defining the truth value of w_i in each group according to how many literals of the corresponding clause of ϕ are **true**.

Finally, it is easy to check that MAX2SAT is in **NP**, and that the reduction can be carried out in logarithmic space (since these important prerequisites will be very clear in most of the subsequent reductions, we shall often omit mentioning them explicitly). □

The style of this proof is quite instructive: To show the problem **NP**-complete we start by toying with small instances of the problem, until we isolate one with an interesting behavior (the ten clauses above). Sometimes the properties of this instance immediately enable a simple **NP**-completeness proof. We shall see more uses of this method, sometimes called "gadget construction," in the next section.

We shall end this section with yet another interesting variant of SAT. In 3SAT we are given a set of clauses with three literals in each, and we are asked whether there is a truth assignment T such that no clause has all three literals **false**. Any other combination of truth values in a clause is allowed; in particular, all three literals might very well be **true**. *Suppose now that we disallow this.* That is, we insist that in no clause are all three literals *equal* in truth value (neither all **true**, nor al **false**). We call this problem NAESAT (for "not-all-equal SAT").

Theorem 9.3: NAESAT is **NP**-complete.

Proof: Let us look back at the reduction from CIRCUIT SAT to SAT (Example 8.3). We shall argue that it is essentially also a reduction from CIRCUIT SAT to NAESAT! To see why, consider the clauses created in that reduction. We add to all one- or two-literal clauses among them the same literal, call it z. We claim that the resulting set of clauses, considered as an instance of NAESAT (not 3SAT) is satisfiable if and only if the original circuit is satisfiable.

Suppose that there is a truth assignment T that satisfies all clauses in the sense of NAESAT. It is easy to see that the *complementary* truth assignment $\overline{T}$

also satisfies all clauses in the NAESAT sense. In one of these truth assignments z takes the value **false**. This truth assignment then satisfies all original clauses (before the addition of z), and therefore—by the reduction in Example 8.3—there is a satisfying truth assignment for the circuit.

Conversely, suppose that there is a truth assignment that satisfies the circuit. Then there is a truth assignment T that satisfies all clauses in the ordinary, 3SAT sense; we take $T(z) =$ **false**. We claim that in no clause all literals are **true** under T (we know they are not all **false**). To see why, recall that clauses come in groups corresponding to gates. **true**, **false**, NOT gates and variable gates have clauses involving z, and hence T does not make all of their literals **true**. For an AND gate we have the clauses $(\neg g \vee h \vee z)$, $(\neg g \vee h' \vee z)$, and $(\neg h \vee \neg h' \vee g)$. It is also easy to see that T cannot satisfy all three literals in any clause: This is trivial for the first two clauses since they contain z; and if all literals in the third are **true**, then the other clauses are not satisfied. The case for OR gates is very similar. □

9.3 GRAPH-THEORETIC PROBLEMS

Many interesting graph-theoretic problems are defined in terms of *undirected* graphs. Technically, an undirected graph is just an ordinary graph which happens to be symmetric and have no self-loops; that is, whenever (i, j) is an edge then $i \neq j$, and (j, i) is also an edge. However, since we shall deal with such graphs extensively, we must develop a better notation for them. An undirected graph is a pair $G = (V, E)$, where V is a finite set of nodes and E is a set of *unordered* pairs of nodes in V, called edges; an edge between i and j is denoted $[i, j]$. An edge will be depicted as a line (no arrows). *All graphs in this section are undirected.*

Let $G = (V, E)$ be an undirected graph, and let $I \subseteq V$. We say that the set I is *independent* if whenever $i, j \in I$ then there is no edge between i and j. All graphs (except for the one with no nodes) have non-empty independent sets; the interesting question is, what is the largest independent set in a graph. The INDEPENDENT SET problem is this: Given an undirected graph $G = (V, E)$, and a goal K is there an independent set I with $|I| = K$?.

Theorem 9.4: INDEPENDENT SET is **NP**-complete.

Proof: The proof uses a simple gadget, the *triangle*. The point is that if a graph contains a triangle, then any independent set can obviously contain at most one node of the triangle. But there is more to the construction.

Interestingly, to prove that INDEPENDENT SET is **NP**-complete it is best to *restrict the class of graphs* we consider. Although restricting the domain makes a problem easier, in this case the restriction is such that it retains the complexity of the problem, while making the issues clearer. We consider only graphs *whose nodes can be partitioned in m disjoint triangles (see Figure 9.2).*

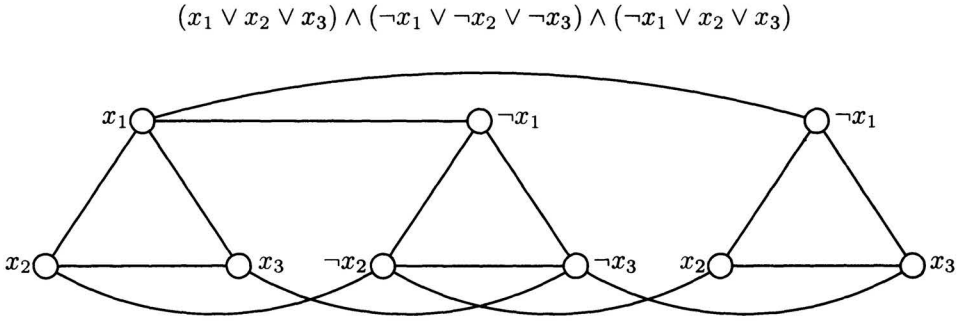

Figure 9-2. Reduction to INDEPENDENT SET.

Then obviously an independent set can contain at most m nodes (one from each triangle); an independent set of size m exists if and only if the other edges of the graph allow us to choose one node from each triangle.

Once we look at such graphs, the combinatorics of the INDEPENDENT SET problem seem easier to understand (and yet no easier to resolve computationally). In fact, a reduction from 3SAT is immediate: For each one of the m clauses of the given expression ϕ we create a separate triangle in the graph G. Each node of the triangle corresponds to a literal in the clause. The structure of ϕ is taken into account in this simple manner: We add an edge between two nodes in different triangles *if and only if the nodes correspond to opposite literals* (see Figure 9.2). The construction is completed by taking the goal to be $K = m$.

Formally, we are given an instance ϕ of 3SAT with m clauses $C_1, \ldots, C_m$, with each clause being $C_i = (\alpha_{i1} \vee \alpha_{i2} \vee \alpha_{i3})$, with the α_{ij}'s being either Boolean variables or negations thereof. Our reduction constructs a graph $R(\phi) = (G, K)$, where $K = m$, and $G = (V, E)$ is the following graph: $V = \{v_{ij} : i = 1, \ldots, m; j = 1, 2, 3\}$; $E = \{[v_{ij}, v_{ik}] : i = 1, \ldots, m; j \neq k\} \cup \{[v_{ij}, v_{\ell k}] : i \neq \ell, \alpha_{ij} = \neg\alpha_{\ell k}\}$. (There is a node for every appearance of a literal in a clause; the first set of edges defines the m triangles, and the second group joins opposing literals.)

We claim that there is an independent set of K nodes in G if and only if ϕ is satisfiable. Suppose that such a set I exists. Since $K = m$, I must contain a node from each triangle. Since the nodes are labeled with literals, and I contains no two nodes corresponding to opposite literals, I is a truth assignment that satisfies ϕ: The **true** literals are just those which are labels of nodes of I (variables left unassigned by this rule can take any value). We know that this gives a truth assignment because any two contradictory literals are connected by an edge in G, and so they cannot both be in I. And since I has a node from every triangle, the truth assignment satisfies all clauses.

Conversely, if a satisfying truth assignment exists, then we identify a **true** literal in each clause, and pick the node in the triangle of this clause labeled by this literal: This way we collect $m = K$ independent nodes. □

By Proposition 9.3, in our proof above we could assume that the original Boolean expression has at most two occurrences of each literal. Thus, each node in the graph constructed in the proof of Theorem 9.4 is adjacent to at most four nodes (that is, it has *degree* at most four): The other two nodes of its triangle, and the two occurrences of the opposite literal. There is a slight complication, because there are clauses now that contain just two literals (recall the proof of Proposition 9.3). But this is easy to fix: Such clauses are represented by a single edge joining the two literals, instead of a triangle. Let k-DEGREE INDEPENDENT SET be the special case of INDEPENDENT SET problem in which all degrees are at most k, an integer; we have shown the following:

Corollary 1: 4-DEGREE INDEPENDENT SET is **NP**-complete. □

Even if the graph is planar, the INDEPENDENT SET problem remains **NP**-complete (see Problem 9.5.9). However, it is polynomially solvable when the graph is *bipartite* (see Problem 9.5.25). The reason for this is that, in bipartite graphs, INDEPENDENT SET is closely related to MATCHING, which in turn is a special case of MAX FLOW.

This brings about an interesting point: Problems in graph theory can be guises of one another in confusing ways; sometimes this suggests trivial reductions from a problem to another. In the CLIQUE problem we are given a graph G and a goal K, and we ask whether there is a set of K nodes that form a *clique* by having all possible edges between them. Also, NODE COVER asks whether there is a set C with B or fewer nodes (where B is a given "budget;" this is a minimization problem) such that each edge of G has at least one of its endpoints in C.

It is easy to see that CLIQUE is a clumsy disguise of INDEPENDENT SET: If we take the *complement* of the graph, that is, a graph that has precisely those edges that are missing from this one, cliques become independent sets and vice-versa. Also, I is an independent set of graph $G = (V, E)$ if and only if $V - I$ is a node cover of the same graph (and, moreover, NODE COVER is, conveniently, a minimization problem). These observations establish the following result:

Corollary 2: CLIQUE and NODE COVER are **NP**-complete. □

A *cut* in an undirected graph $G = (V, E)$ is a partition of the nodes into two nonempty sets S and $V - S$. The *size* of a cut $(S, V - S)$ is the number of edges between S and $V - S$. It is an interesting problem to find the cut with the smallest size in a graph. It turns out that this problem, called MIN CUT, is in **P**. To see why, recall that the smallest cut *that separates two given nodes s and t* equals the maximum flow from s to t (Problem 1.4.11). Thus, to find the minimum overall cut, we just need to find the maximum flow between

some fixed node s and each of the other nodes of V, and pick the smallest value found.

But *maximizing* the size of a cut is much harder:

Theorem 9.5: MAX CUT is **NP**-complete.

Arguably the most important decision in designing an **NP**-completeness proof is, *from which* **NP***-complete problem to start.* Naturally, there are no easy rules here. One should always start by getting as much experience as possible with the problem in hand, examining small and interesting examples. Only then it is perhaps worth going through a (mental or not, see the references) list of known **NP**-complete problems, to see if any problem in there seems very close to the problem in hand. Others always start with 3SAT, an extremely versatile problem which can be easily reduced to a surprising range of **NP**-complete problems (several examples follow). However, in some cases there is tangible dividend from finding the right problem to start: There is a reduction so elegant and simple, that resorting to 3SAT would obviously have been a waste. The following proof is a good example.

Proof: We shall reduce NAESAT to MAX CUT. We are given m clauses with three literals each. We shall construct a graph $G = (V, E)$ and a goal K such that there is a way to separate the nodes of G into two sets S and $V - S$ with K or more edges going from one set to another, if and only if there is a truth assignment that makes at least one literal **true** and at least one literal **false** in each clause. In our construction we shall stretch our definition of a graph by allowing *multiple edges between two nodes;* that is, there may be more than one edge from a node to another, and each of these edges will contribute one to the cut—*if* these nodes are separated.

Suppose that the clauses are $C_1, \ldots, C_m$, and the variables appearing in them $x_1, \ldots, x_n$. G has $2n$ nodes, namely $x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n$. The gadget employed *is again the triangle,* but this time it is used in a very different way: The property of a triangle we need here is that *the size of its maximum cut is two,* and this is achieved by splitting the nodes in any way. For each clause, say $C_i = (\alpha \vee \beta \vee \gamma)$ (recall that α, β, and γ are also nodes of G), we add to E the three edges of the triangle $[\alpha, \beta, \gamma]$. If two of these literals coincide, we omit the third edge, and the triangle degenerates to two parallel edges between the two distinct literals. Finally, for each variable x_i we add n_i copies of the edge $[x_i, \neg x_i]$, *where n_i is the number of occurrences of x_i or $\neg x_i$ in the clauses.* This completes the construction of G (see Figure 9.3). As for K, it is equal to $5m$.

Suppose that there is a cut $(S, V - S)$ of size $5m$ or more. We claim that it is no loss of generality to assume that all variables are separated from their negations. Because if both x_i and $\neg x_i$ are on the same side of the cut, then together they contribute at most $2n_i$ adjacent edges to the cut, and thus we

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

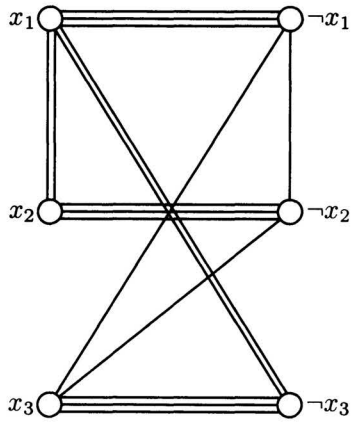

Figure 9-3. Reduction to MAX CUT.

would be able to change the side of one of them without decreasing the size of the cut. Thus, we think of the literals in S as being **true**, and those in $V - S$ as being **false**.

The total number of edges in the cut that join opposite literals is $3m$ (as many as there are occurrences of literals). The remaining $2m$ edges must be obtained from the triangles that correspond to the clauses. And since each triangle can contribute at most two to the size of the cut, all m triangles must be split. However, that a triangle is split means that at least one of its literals is **false**, and at least one **true**; hence, all clauses are satisfied by the truth assignment in the sense of NAESAT.

Conversely, it is easy to translate a truth assignment that satisfies all clauses into a cut of size $5m$. □

In many interesting applications of graph partitioning, the two sets S and $V - S$ cannot be arbitrarily small or large. Suppose that we are looking for a cut $S, V - S$ of size K or more *such that* $|S| = |V - S|$ (if there is an odd number of nodes in V, this problem is *very* easy...). We call this problem MAX BISECTION.

Is MAX BISECTION easier or harder than MAX CUT? Imposing an extra restriction (that $|S| = |V-S|$) certainly makes the problem *conceptually* harder. It also *adversely affects the outcome*, in the sense that the maximum may become smaller. However, an extra requirement could affect the *computational* complexity of the problem both ways. You should have no difficulty recalling (or

devising) problems in **P** for which imposing an extra constraint on the solution space leads to **NP**-completeness, and examples in which exactly the opposite happens. In the case of NAESAT (which results from 3SAT by adding an extra restriction to what it means for a truth assignment to be satisfying) we saw that the problem remains just as hard. This is also the case presently:

Lemma 9.1: MAX BISECTION is **NP**-complete.

Proof: We shall reduce MAX CUT to it. This is a special kind of reduction: We modify the given instance of MAX CUT *so that the extra constraint is easy to satisfy*, and thus the modified instance (of MAX BISECTION) has a solution if and only if the original instance (of MAX CUT) does. The trick here is very simple: *Add $|V|$ completely disconnected new nodes to G.* Since every cut of G can be made into a bisection by appropriately splitting the new nodes between S and $V - S$, the result follows. □

Actually, an even easier way to prove Lemma 9.1 would be to observe that the reduction in the proof of Theorem 9.5 constructs a graph in which the optimum cut *is always a bisection!* How about the minimization version of the bisection problem, called BISECTION WIDTH? It turns out that the extra requirement turns the polynomial problem MIN CUT into an **NP**-complete problem:

Theorem 9.6: BISECTION WIDTH is **NP**-complete.

Proof: Just observe that a graph $G = (V, E)$, where $|V| = 2n$ is an even number, has a bisection of size K or more if and only if the *complement* of G has a bisection of size $n^2 - K$. □

This sequence is instructive for another reason: It touches on the issue of *when a maximization problem is computationally equivalent to the corresponding minimization problem* (see Problem 9.5.14 for some other interesting examples and counterexamples).

We now turn to another genre of graph-theoretic problems. Although HAMILTON PATH was defined for directed graphs, we shall now deal with its undirected special case: Given an undirected graph, does it have a Hamilton path, that is, a path visiting each node exactly once?

Theorem 9.7: HAMILTON PATH is **NP**-complete.

Proof: We shall reduce 3SAT to HAMILTON PATH. We are given a formula ϕ in conjunctive normal form with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$, each with three literals. We shall construct a graph $R(\phi)$ that has a Hamilton path if and only if the formula is satisfiable.

In any reduction from 3SAT we must find ways to express in the domain of the target problem the basic elements of 3SAT; hopefully, everything else will fall in place. But what are the basic ingredients of 3SAT? In an instance of 3SAT we have first of all Boolean variables; the basic attribute of a variable is that it has

Figure 9-4. The choice gadget.

a *choice* between two values, **true** and **false**. We then have occurrences of these variables; the fundamental issue here is *consistency*, that is, all occurrences of x must have the same truth value, and all occurrences of $\neg x$ must have the opposite value. Finally, the occurrences are organized into clauses; it is the clauses that provide the *constraints* that must be satisfied in 3SAT. A typical reduction from 3SAT to a problem constructs an instance that contains parts that "flip-flop" to represent the *choice* by variables; parts of the instance that "propagate" the message of this choice to all occurrences of each variable, thus ensuring *consistency;* and parts that make sure the *constraint* is satisfied. The nature of these parts will vary wildly with the target problem, and ingenuity is sometimes required to take advantage of the intricacies of each specific problem to design the appropriate parts.

In the case of HAMILTON PATH, it is easy to conceive of a *choice* gadget (see Figure 9.4): This simple device will allow the Hamilton path, approaching this subgraph from above, to pick either the left or right parallel edge, thus committing to a truth value. (It will become clear soon that, despite the appearance of Figure 9.4, the graph constructed will have no actual parallel edges.) In this and the other figures in this proof we assume that the devices shown are connected with the rest of the graph only through their *endpoints*, denoted as full dots; there are no edges connecting other nodes of the device to the rest of the graph.

Consistency is ensured using the graph in Figure 9.5(a). Its key property is this: Suppose that this graph is a subgraph of a graph G, connected to the rest of G through its endpoints alone, and suppose that G has a Hamilton path *which does not start or end at a node of this subgraph.* Then this device must be traversed by the Hamilton path in one of two ways, shown in Figures 9.5(b) and (c). To prove this, one has to follow the path as it traverses the subgraph starting from one of the endpoints, and make sure that all deviations from the two ways will lead to a node being left out of the path. This observation

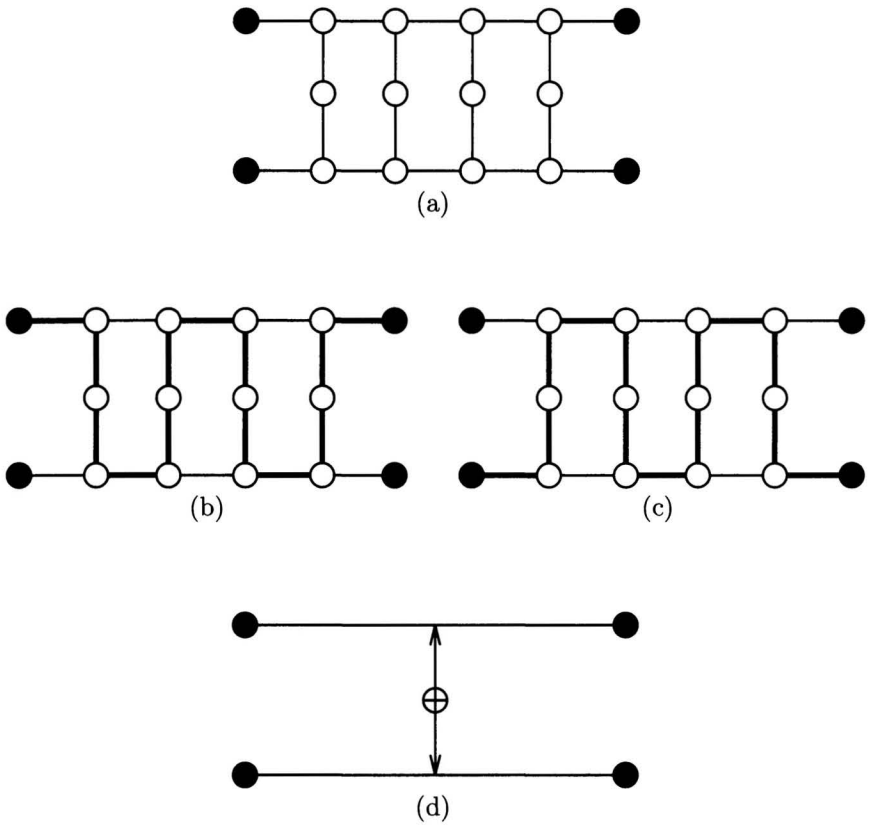

Figure 9-5. The consistency gadget.

establishes that this device behaves as two separate edges, having the property that *in each Hamilton path one of the edges is traversed, and the other is not.* That is, we can think of the device of Figure 9.5(a) as an "exclusive-or" gate connecting two otherwise independent edges of the graph (see Figure 9.5(d)).

How about clauses? How shall we translate the *constraint* imposed by them in the language of Hamilton paths? The device here is, once more, the triangle—one side for each literal in the clause, see Figure 9.6. This is how it works: Suppose that, using our choice and consistency devices, we have made sure that each side of the triangle is traversed by the Hamilton path if and only if the corresponding literal is **false**. Then it is immediate that at least one literal has to be **true**: Otherwise, all three edges of the triangle will be traversed, and hence the alleged "Hamilton path" is neither.

It is now straightforward to put all the pieces together (see Figure 9.7 for

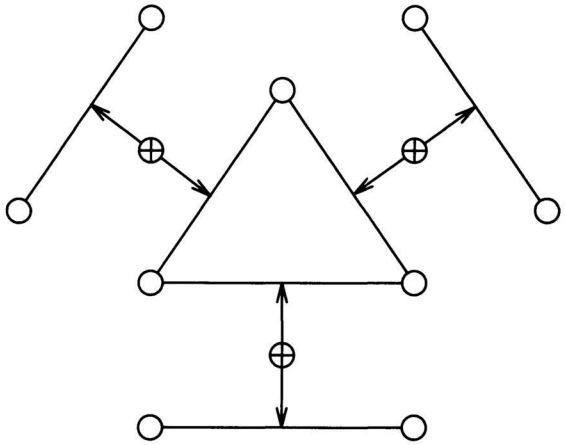

Figure 9-6. The constraint gadget.

an example). Our graph G has n copies of the choice gadget, one for each variable, connected in series (this is the right part of the figure; the first node of the chain is called 1). It also has m triangles, one for each clause, with an edge in the triangle identified with each literal in the clause (in the left part). If a side corresponds to literal x_i, it is connected with an "exclusive or" gadget with the **true** edge of the choice subgraph corresponding to x_i (so that it is traversed if that edge is not); a side corresponding to $\neg x_i$ is connected to the **false** side. (Each **true** or **false** edge may be connected by "exclusive ors" to several sides; these "exclusive ors" are arranged next to each other, as suggested in Figure 9.7.) Finally, all $3m$ nodes of the triangles, plus the last node of the chain of choice gadgets and a new node 3, are connected by all possible edges, creating a huge clique; a single node 2 is attached to node 3 (the last feature only facilitates our proof). This completes our construction of $R(\phi)$.

We claim that the graph has a Hamilton path if and only if ϕ has a satisfying truth assignment. Suppose that a Hamilton path exists. Its two ends must be the two nodes of degree one, 1 and 2, so we can assume that the path starts at node 1 (Figure 9.7). From there, it must traverse one of the two parallel edges of the choice gadget for the first variable. Furthermore, all exclusive ors must be traversed as in Figure 9.5(b) or (c). Therefore the path will continue on after traversing the exclusive ors, and thus the whole chain of choices will be traversed. This part of the Hamilton path defines a truth assignment, call it T. After this, the path will continue to traverse the triangles in some order, and end up at 2.

We claim that T satisfies ϕ. In proof, since all exclusive or gadgets are traversed as in Figure 9.5(b) or (c), they indeed behave like exclusive ors con-

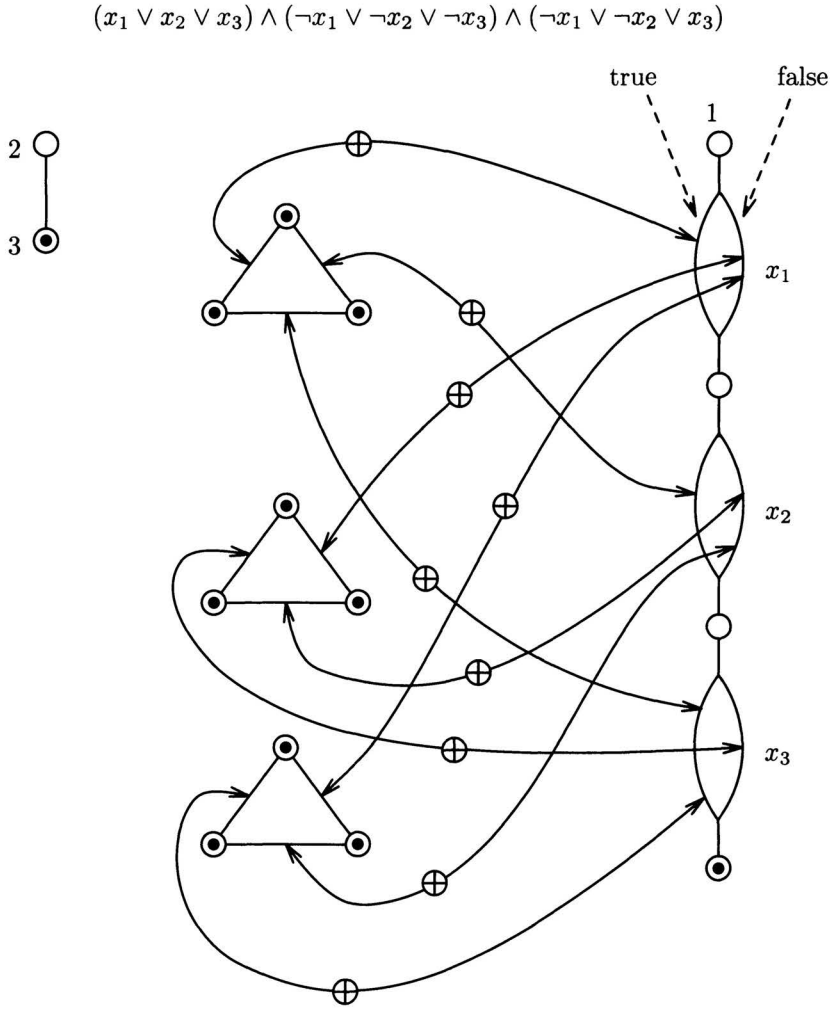

Figure 9-7. The reduction from 3SAT to HAMILTON PATH.

necting otherwise independent edges. So, the sides of triangles corresponding to literals are traversed if and only if the literals are **false**. It follows that there are no clauses that have all three literals **false**, and hence ϕ is satisfied.

Conversely, suppose that there is a truth assignment T that satisfies ϕ.

We shall exhibit a Hamilton path of $R(\phi)$. The Hamilton path starts at 1, and traverses the chain of choices, picking for each variable the edge corresponding to its truth value under T. Once this is done, the rest of the graph is a huge clique, with certain node-disjoint paths of length two or less that have to be traversed. Since all possible edges are present, it is easy to piece these paths together and complete the Hamilton path, so that it ends at nodes 3 and 2. □

Corollary: TSP (D) is **NP**-complete.

Proof: We shall reduce HAMILTON PATH to it. Given a graph G with n nodes, we shall design a distance matrix d_{ij} and a budget B such that there is a tour of length B or less if and only if G has a Hamilton path. There are n cities, one for each node of the graph. The distance between two cities i and j is 1 if there is an edge $[i, j]$ in G, and 2 otherwise. Finally, $B = n + 1$. The proof that this works is left to the reader. □

Suppose that we are asked to "color" the vertices of a given graph with k colors such that no two adjacent nodes have the same color. This classical problem is called k-COLORING. When $k = 2$ it is quite easy to solve (Problem 1.4.5). For $k = 3$ things change, as usual:

Theorem 9.8: 3-COLORING is **NP**-complete.

Proof: The proof is a simple reduction from NAESAT. We are given a set of clauses $C_1, \ldots, C_m$ each with three literals, involving the variables $x_1, \ldots, x_n$, and we are asked whether there is a truth assignment on the variables such that no clause has all literals **true**, or all literals **false**.

We shall construct a graph G, and argue that it can be colored with colors $\{0, 1, 2\}$ if and only if all clauses can take diverse values. Triangles play an important role again: A triangle forces us to use up all three colors on its nodes. Thus, our graph has for each variable x_i a triangle $[a, x_i, \neg x_i]$; all these triangles share a node a (it is the node at the top colored "2" in Figure 9.8).

Each clause C_i is also represented by a triangle, $[C_{i1}, C_{i2}, C_{i3}]$ (bottom of Figure 9.8). Finally, there is an edge connecting C_{ij} with the node that represents the jth literal of C_i. This completes the construction of the graph G (see Figure 9.8 for an example).

We claim that G can be colored with colors $\{0, 1, 2\}$ if and only if the given instance of NAESAT is satisfiable. In proof, suppose that the graph is indeed 3-colorable. We can assume, by changing color names if necessary, that node a takes the color 2, and so for each i one of the nodes x_i and $\neg x_i$ is colored 1 and the other 0. If x_i takes the color 1 we think that the variable is **true**, otherwise it is **false**. How can the clause triangles be colored? If all literals in a clause are **true**, then the corresponding triangle cannot be colored, since color 1 cannot be used; so the overall graph is not 3-colorable. Similarly if all literal are **false**. This completes the proof of one direction.

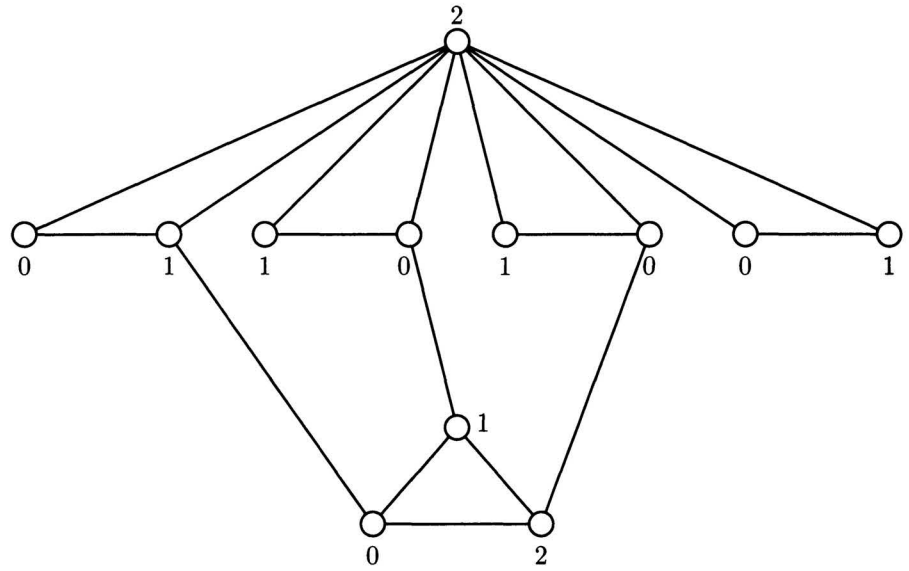

Figure 9-8. The reduction to 3-COLORING.

For the other direction, suppose that a satisfying (in the NAESAT sense) truth assignment exists. We color node a by color 2, and the variable triangles in the way that reflects the truth assignment. And for any clause, we can color the clause triangle as follows: We pick two literals in it with opposite truth values (they exist, since the clause is satisfied) and color the vertices corresponding to them with the available color among $\{0, 1\}$ (0 if the literal is **true**, 1 if it is **false**); we then color the third node 2. □

9.4 SETS AND NUMBERS

We can generalize bipartite matching of Section 1.2 as follows: Suppose that we are given three sets B, G, and H (boys, girls, and *homes*), each containing n elements, and a ternary relation $T \subseteq B \times G \times H$. We are asked to find a set of n triples in T, no two of which have a component in common—that is, each boy is matched to a different girl, and each couple has a home of its own. We call this problem TRIPARTITE MATCHING.

Theorem 9.9: TRIPARTITE MATCHING is **NP**-complete.

Proof: We shall reduce 3SAT to TRIPARTITE MATCHING. The basic ingredient is a *combined gadget for both choice and consistency*, shown in Figure 9.9 (where triples of the relation R are shown as triangles). There is such a device

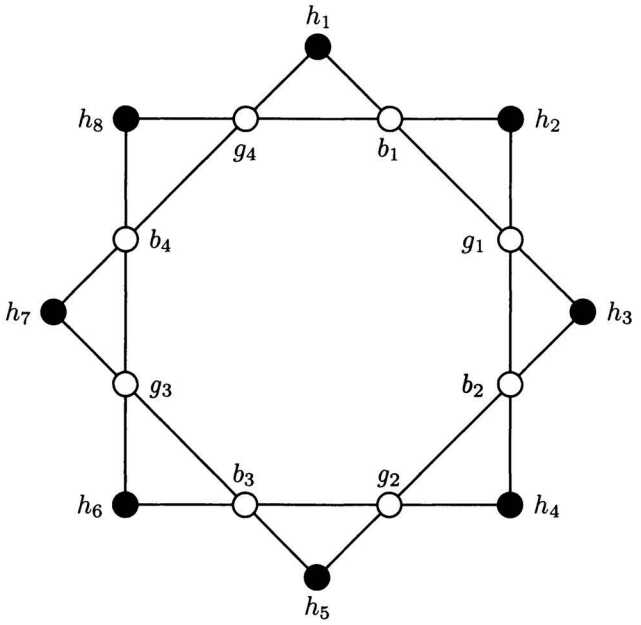

Figure 9-9. The choice-consistency gadget.

for each variable x of the formula. It involves k boys and k girls (forming a circle $2k$ long) and $2k$ homes, where k is either the number of occurrences of x in the formula, or the number of occurrences of $\neg x$, whichever is larger (in the figure, $k = 4$; we could have assumed that $k = 2$, recall Proposition 9.3). Each occurrence of x or $\neg x$ in ϕ is represented by one of the h_j's; however, if x and $\neg x$ have unequal number of occurrences, some of the-h_i's will correspond to no occurrence. Homes $h_{2i-1}, i = 1, \ldots, k$ represent occurrences of x, while $h_{2i}, i = 1, \ldots, k$ represent occurrences of $\neg x$. The k boys and k girls participate in no other triple of R other than those shown in the figure. Thus, if a matching exists, b_i is matched either to g_i and h_{2i}, or to g_{i-1} (g_k if $i = 1$) and h_{2i-1}, $i = 1, \ldots, k$. The first matching is taken to mean that $T(x) =$ **true**, the second that $T(x) =$ **false**. Notice that, indeed, this device ensures that variable x picks a truth value, and all of its occurrences have consistent values.

The clause constraint is represented as follows: For each clause c we have a boy and a girl, say b and g. The only triples to which b or g belong are three triples of the form (b, g, h), where h ranges over the three homes corresponding to the three occurrences of the literals in the clause c. The idea is that, if one of these three homes was left unoccupied when the variables were assigned truth values, this means that it corresponds to a **true** literal, and thus c is satisfied.

If all three literals in c are **false**, then b and g cannot be matched with a home.

This would complete the construction, except for one detail: Although the instance has the same number of boys and girls, *there are more homes than either.* If there are m clauses there are going to be $3m$ occurrences, which means that the number of homes, H, is at least $3m$ (for each variable we have at least as many homes as occurrences). On the other hand, there are $\frac{H}{2}$ boys in the choice-consistency gadgets, and $m \leq \frac{H}{3}$ more in the constraint part; so there are indeed fewer boys than homes. Suppose that the excess of homes over boys (and girls) is ℓ—a number easy to calculate from the given instance of 3SAT. We can take care of this problem very easily: We introduce ℓ more boys and ℓ more girls (thus the numbers of boys, girls, and homes are now equal). The ith such girl participates in $|H|$ triples, with the ith boy *and each home.* In other words, these last additions are ℓ "easy to please" couples, useful for completing any matching in which homes were left unoccupied.

We omit the formal proof that a tripartite matching exists if and only if the original Boolean expression was satisfiable. □

There are some other interesting problems involving sets, that we define next. In SET COVERING we are given a family $F = \{S_1, \ldots, S_n\}$ of subsets of a finite set U, and a budget B. We are asking for a set of B sets in F whose union is U. In SET PACKING we are also given a family of subsets of a set U, and a goal K; this time we are asked if there are K pairwise disjoint sets in the family. In a problem called EXACT COVER BY 3-SETS we are given a family $F = \{S_1, \ldots, S_n\}$ of subsets of a set U, such that $|U| = 3m$ for some integer m, and $|S_i| = 3$ for all i. We are asked if there are m sets in F that are disjoint and have U as their union.

We can show all these problems **NP**-complete by pointing out that they are all generalizations of TRIPARTITE MATCHING. This is quite immediate in the case of EXACT COVER BY 3-SETS; TRIPARTITE MATCHING is the special case in which U can be partitioned into three equal sets B, G, and H, such that each set in F contains one element from each. Then, it is easy to see that EXACT COVER BY 3-SETS is the special case of SET COVERING in which the universe has $3m$ elements, all sets in F have three elements, and the budget is m. Similarly for SET PACKING.

Corollary: EXACT COVER BY 3-SETS, SET COVERING, and SET PACKING are **NP**-complete. □

INTEGER PROGRAMMING asks whether a given system of linear inequalities, in n variables and with integer coefficients, has an integer solution. We have already seen more than a dozen reasons why this problem is **NP**-complete: All problems we have seen so far can be easily expressed in terms of linear inequalities over the integers. For example, SET COVERING can be expressed by the inequalities $Ax \geq \mathbf{1}$; $\sum_{i=1}^{n} x_i \leq B$; $0 \leq x_i \leq 1$, where each x_i is a $0 - 1$

variable which is one if and only if S_i is in the cover, A is the matrix whose rows are the bit vectors of the sets, $\mathbf{1}$ is the column vector with all entries 1, and B is the budget of the instance. Hence INTEGER PROGRAMMING is **NP**-complete (the hard part is showing that it is in **NP**; see the notes at the end of the chapter). In contrast, LINEAR PROGRAMMING, the same problem without the requirement that the solutions be integers, is in **P** (see the discussion in 9.5.34).

We shall next look at a very special case of INTEGER PROGRAMMING. The *knapsack problem* looks at the following situation. We must select some among a set of n items. Item i has *value* v_i, and *weight* w_i, both positive integers. There is a limit W to the total weight of the items we can pick. We wish to pick certain items (without repetitions) to maximize the total value, subject to the constraint that the total weight is at most G. That is, we are looking for a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} w_i \leq W$, and $\sum_{i \in S} v_i$ is as large as possible. In the recognition version, called KNAPSACK, we are also given a goal K, and we wish to find a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i \geq K$.

$\rightarrow$	0	0	0	1	0	1	1	0	0	0	0	0
	1	1	0	0	1	0	0	0	0	0	0	0
$\rightarrow$	1	0	1	0	0	0	0	1	0	0	0	0
$\rightarrow$	0	1	0	0	0	0	0	0	1	0	0	1
	0	0	1	1	1	0	0	0	0	0	0	0
$\rightarrow$	0	0	0	0	1	0	0	0	0	1	1	0
$+$	1	0	1	1	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1

Figure 9.10. Reduction to KNAPSACK.

Theorem 9.10: KNAPSACK is **NP**-complete.

Proof: This is another case in which restricting the problem facilitates the **NP**-completeness proof. We are going to look at the special case of KNAPSACK in which $v_i = w_i$ for all i, and $K = W$. That is, we are given a set of n integers $w_1, \ldots, w_n$, and another integer K, and we wish to find out if a subset of the given integers adds up to exactly K. This simple numerical problem turns out to be **NP**-complete.

We shall reduce EXACT COVER BY 3-SETS to it. We are given an instance $\{S_1, S_2, \ldots, S_n\}$ of EXACT COVER BY 3-SETS, where we are asking whether there are disjoint sets among the given ones that cover the set $U = \{1, 2, \ldots, 3m\}$. Think of the given sets as bit vectors in $\{0,1\}^{3m}$. Such vectors can also be thought as binary integers, and set union now resembles integer addition (see Figure 9.10). Our goal is to find a subset of these integers that add up to

$K = 2^n - 1$ (the all-ones vector, corresponding to the universe). The reduction seems complete!

But there is a "bug" in this reduction: Binary integer addition is different from set union in that it has *carry*. For example, $3 + 5 + 7 = 15$ in bit-vector form is $0011 + 0101 + 0111 = 1111$; but the corresponding sets $\{3, 4\}$,$\{2, 4\}$, and $\{2, 3, 4\}$ are not disjoint, neither is their union $\{1, 2, 3, 4\}$. There is a simple and clever way around this problem: Think of these vectors as integers not in base 2, *but in base* $n + 1$. That is, set S_i becomes integer $w_i = \sum_{j \in S_i} (n+1)^{3m-j}$. Since now there can be no carry in any addition of up to n of these numbers, it is straightforward to argue that there is a set of these integers that adds up to $K = \sum_{j=0}^{3m-1} (n+1)^j$ if and only if there is an exact cover among $\{S_1, S_2, \ldots, S_n\}$. □

Pseudopolynomial Algorithms and Strong NP-completeness

In view of Theorem 9.10, the following result seems rather intriguing:

Proposition 9.4: Any instance of KNAPSACK can be solved in $\mathcal{O}(nW)$ time, where n is the number of items and W is the weight limit.

Proof: Define $V(w, i)$ to be the largest value attainable by selecting some among the i first items *so that their total weight is exactly* w. It is easy to see that the nW entries of the $V(w, i)$ table can be computed in order of increasing i, and with a constant number of operations per entry, as follows:

$$V(w, i+1) = \max\{V(w, i), v_{i+1} + V(w - w_{i+1}, i)\}$$

To start, $V(w, 0) = 0$ for all w. Finally, the given instance of KNAPSACK is a "yes" instance if and only if the table contains an entry greater than or equal to the goal K. □

Naturally, Proposition 9.4 does not establish that **P** = **NP** (so, keep on reading this book!). This is not a polynomial algorithm because its time bound nW is *not a polynomial function of the input:* The length of the input is something like $n \log W$. We have seen this pattern before in our first attempt at an algorithm for MAX FLOW in Section 1.2, when the time required was again polynomial in the integers appearing in the input (instead of their logarithms, which is always the correct measure). Such "pseudopolynomial" algorithms are a source not only of confusion, but of genuinely positive results (see Chapter 13 on approximation algorithms).

In relation to pseudopolynomial algorithms, it is interesting to make the following important distinction between KNAPSACK and the other problems that we showed **NP**-complete in this chapter—SAT, MAX CUT, TSP (D), CLIQUE, TRIPARTITE MATCHING, HAMILTON PATH, and many others. All these latter problems were shown **NP**-complete via reductions that constructed only *polynomially small integers.* For problems such as CLIQUE and SAT, in which integers

are only used as node names and variable indices, this is immediate. But even for TSP (D), in which one would expect numbers to play an important role as intercity distances, we only needed distances no larger than *two* to establish **NP**-completeness (recall the proof of the Corollary to Theorem 9.7)†. In contrast, in our **NP**-completeness proof for KNAPSACK we had to create *exponentially large integers* in our reduction.

If a problem remains **NP**-complete even if any instance of length n is restricted to contain integers of size at most $p(n)$, a polynomial, then we say that the problem is *strongly* **NP**-*complete*. All **NP**-complete problems that we have seen so far in this chapter, with the single exception of KNAPSACK, are strongly **NP**-complete. It is no coincidence then that, of all these problems, only KNAPSACK can be solved by a pseudopolynomial algorithm: It should be clear that strongly **NP**-complete problems *have no pseudopolynomial algorithms*, unless of course $\mathbf{P} = \mathbf{NP}$ (see Problem 9.5.31).

We end this chapter with a last interesting example: A problem which involves numbers and bearing a certain similarity to KNAPSACK, but turns out to be strongly **NP**-complete.

BIN PACKING: We are given N positive integers $a_1, a_2, \ldots, a_N$ (the *items*), and two more integers C (the *capacity*) and B (the *number of bins*). We are asked whether these numbers can be partitioned into B subsets, each of which has total sum at most C.

Theorem 9.11: BIN PACKING is **NP**-complete.

Proof: We shall reduce TRIPARTITE MATCHING to it. We are given a set of boys $B = \{b_1, b_2, \ldots, b_n\}$, a set of girls $G = \{g_1, g_2, \ldots, g_n\}$, a set of homes $H = \{h_1, h_2, \ldots, h_n\}$, and a set of triples $T = \{t_1, \ldots, t_m\} \subseteq B \times G \times H$; we are asked whether there is a set of n triples in T, such that each boy, girl, and home is contained in one of the n triples.

The instance of BIN PACKING that we construct has $N = 4m$ items—one for each triple, and one for each occurrence of a boy, girl, or home to a triple. The items corresponding to the occurrences of b_1, for example, will be denoted by $b_1[1], b_1[2], \ldots, b_1[N(b_1)]$, where $N(b_1)$ is the number of occurrences of b_1 in the triples; similarly for the other boys, the girls, and the homes. The items corresponding to triples will be denoted simply t_j.

The sizes of these items are shown in Figure 9.11. M is a very large number, say $100n$. Notice that one of the occurrences of each boy, girl, and home (arbitrarily the first) has different size than the rest; it is this occurrence that will participate in the matching. The capacity C of each bin is $40M^4+15$—

† To put it otherwise, these problems would remain **NP**-complete if numbers in their instances were represented *in unary*—even such wasteful representation would increase the size of the instance by a only polynomial amount, and thus the reduction would still be a valid one.

just enough to fit a triple and one occurrence of each of its three members *as long as either all three or none of the three are a first occurrence.* There are m bins, as many as triples.

Item	Size
first occurrence of a boy $b_i[1]$	$10M^4 + iM + 1$
other occurrences of a boy $b_i[q], q > 1$	$11M^4 + iM + 1$
first occurrence of a girl $g_j[1]$	$10M^4 + jM^2 + 2$
other occurrences of a girl $g_j[q], q > 1$	$11M^4 + jM^2 + 2$
first occurrence of a home $h_k[1]$	$10M^4 + kM^3 + 4$
other occurrences of a home $h_k[q], q > 1$	$8M^4 + kM^3 + 4$
triple $(b_i, g_j, h_k) \in T$	$10M^4 + 8 - iM - jM^2 - kM^3$

Figure 9.11. The items in BIN PACKING.

Suppose that there is a way to fit these items into m bins. Notice immediately that the sum of all items is mC (the total capacity of all bins), and thus all bins must be exactly full. Consider one bin. It must contain four items (proof: all item sizes are between $\frac{1}{5}$ and $\frac{1}{3}$ of the bin capacity). Since the sum of the items modulo M must be 15 ($C \bmod M = 15$), and there is only one way of creating 15 by choosing four numbers (with repetitions allowed) out of 1, 2, 4, and 8 (these are the residues of all item sizes, see Figure 9.11), the bin must contain a triple that contributes 8 mod M, say (b_i, g_j, h_k), and occurrences of a boy $b_{i'}$, a girl $g_{j'}$, and a home $h_{k'}$, contributing 1, 2, and 4 mod 15, respectively. Since the sum modulo M^2 must be 15 as well, we must have $(i' - i) \cdot M + 15 = 15 \bmod M^2$, and thus $i = i'$. Similarly, taking the sum modulo M^3 we get $j = j'$, and modulo M^4 we get $k = k'$. Thus, each bin contains a triple $t = (b_i, g_j, h_k)$, together with one occurrence of b_i, one of g_j, and one of h_k. Furthermore, either all three occurrences are first occurrences, or none of them are—otherwise $40M^4$ cannot be achieved. Hence, there are n bins that contain only first occurrences; *the n triples in these bins form a tripartite matching.*

Conversely, if a tripartite matching exists, we can fit all items into the m bins by matching each triple with occurrences of its members, making sure that the triples in the matching get first occurrences of all three members. The proof is complete. □

Notice that the numbers constructed in this reduction are polynomially large —$\mathcal{O}(|x|^4)$, where x is the original instance of TRIPARTITE MATCHING. Hence, BIN PACKING is strongly **NP**-complete: Any pseudopolynomial algo-

rithm for BIN PACKING would yield a polynomial algorithm for TRIPARTITE MATCHING, implying $\mathbf{P} = \mathbf{NP}$.

BIN PACKING is a useful point of departure for reductions to problems in which numbers appear to play a central role, but which, unlike KNAPSACK (at least as far as we know), are strongly **NP**-complete.

9.5 NOTES, REFERENCES, AND PROBLEMS

9.5.1 Many of the **NP**-completeness results in this chapter were proved in

- R. M. Karp "Reducibility among combinatorial problems," pp. 85–103 in *Complexity of Computer Computations*, edited by J. W. Thatcher and R. E. Miller, Plenum Press, New York, 1972,

a most influential paper in which the true scope of **NP**-completeness (as well as its importance for combinatorial optimization) was revealed. One can find there early proofs of Theorems 9.3, 9.7, 9.8, and 9.9, as well as the **NP**-completeness results in Problems 9.5.7 and 9.5.12. The definitive work on **NP**-completeness is

- M. R. Garey and D. S. Johnson *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, San Francisco, 1979.

This book contains a list of several hundreds of **NP**-complete problems, *circa* 1979, and is still a most useful reference on the subject (and a good book to leaf through during spells of **NP**-completeness prover's block). David Johnson's continuing commentary on **NP**-completeness, and complexity in general, is a good supplementary reference:

- D. S. Johnson "The NP-completeness column: An on-going guide," *J. of Algorithms, 4*, 1981, and hence.

Proposition 9.1 on the characterization of **NP** in terms of "certificates" was implicit in the work of Jack Edmonds (see the references in 1.4.7).

9.5.2 Problem: Give a direct reduction from SAT to 3SAT. (That is, given a clause with more than three literals, show how to rewrite it as an equivalent set of three-literal clauses, perhaps using additional auxiliary variables.)

9.5.3 Problem: Show that the version of 3SAT in which we require that *exactly one* literal in each clause be true (instead of at least one), called ONE-IN-THREE SAT, is **NP**-complete. (In fact, it remains **NP**-complete even if all literals are positive; see the Corollary to Theorem 9.9 on EXACT COVER BY 3-SETS.)

9.5.4 Problem: (a) Show that the special case of SAT in which each variable can only appear *twice* is in **P**.

In fact, something stronger is true:

(b) Show that the restriction of 3SAT (exactly three literals per clause, no repetitions) in which each variable can appear at most three times, is in **P**. (Consider the bipartite graph with clauses as boys and variables as girls, and edges denoting appearances. Use Problem 9.5.25 to show that it always has a matching.)

9.5.5 Problem: Show that the version of 3SAT in which each clause contains appearances of three *distinct* variables is also **NP**-complete. What is the minimum number of occurrences of variables for which the result holds?

9.5.6 Problem: Show that the special case of SAT in which each clause is *either Horn or has two literals* is **NP**-complete. (In other words, the polynomial special cases of SAT do not mix well.)

9.5.7 Problem: In the DOMINATING SET problem we are given a *directed* graph $G = (V, E)$ and an integer K. We are asking whether there is a set D of K or fewer nodes such that for each $v \notin D$ there is a $u \in D$ with $(u, v) \in E$. Show that DOMINATING SET is **NP**-complete. (Start from NODE COVER, obviously a very similar problem, and make a simple local replacement.)

9.5.8 Problem: A *tournament* is a directed graph such that for all nodes $u \neq v$ exactly one of the edges (u, v) and (u, v) is present. (Why is it called this?)

(a) Show that *every tournament with n nodes has a dominating set of size* $\log n$. (Show that in any tournament there is a player who beats at least half of her opponents; add this player to the dominating set. What is left to dominate?)

(b) Show that, if the DOMINATING SET problem remains **NP**-complete even if the graph is a tournament, then $\mathbf{NP} \subseteq \mathbf{TIME}(n^{k \cdot \log n})$.

The DOMINATING SET problem for tournaments is one of a few natural problems that exhibit a strange sort of "limited nondeterminism;" see

- N. Megiddo and U. Vishkin "On finding a minimum dominating set in a tournament," *Theor. Comp. Sci, 61*, pp. 307–316, 1988, and
- C. H. Papadimitriou and M. Yannakakis "On limited nondeterminism and the complexity of computing the V-C dimension," *Proc. of the 1993 Symposium on Structure in Complexity Theory.*

9.5.9 Problem: (a) Show that the INDEPENDENT SET problem remains **NP**-complete even if the graph is planar. (Show how to replace a crossing of edges so that the basic features of the problem are preserved. Alternatively, see Problem 9.5.16.)

(b) Which of these problems, closely related to INDEPENDENT SET, also remain **NP**-complete in the planar special case? (i) NODE COVER, (ii) CLIQUE, (iii) DOMINATING SET.

9.5.10 Problem: Let $G = (V, E)$ be a directed graph. A *kernel* of G is a subset K of V such that (1) for any two $u, v \in K$ $(u, v) \notin E$, and (2) for any $v \notin K$ there is a $u \in K$ such that $(u, v) \in E$. In other words, a kernel is an independent dominating set.

(a) Show that it is **NP**-complete to tell whether a graph has a kernel. (Two nodes with both arcs between them and no other incoming arcs can act as a flip-flop.)

(b) Show that a strongly connected directed graph with no odd cycles is bipartite (its nodes can be partitioned into two sets, with no edges within the sets), and so it has at least two kernels.

(c) Show that it is **NP**-complete to tell whether a strongly connected directed graph with no odd cycles has a third kernel (besides the two discovered in (b) above).

(d) Based on (b) above show that any directed graph with no odd cycles has a kernel.

(e) Modify the search algorithm in Section 1.1 to decide in polynomial time whether a given directed graph has no odd cycles.

(f) Show that any *symmetric* graph without self-loops (that is, any undirected graph) has a kernel.

(g) Show that the MINIMUM UNDIRECTED KERNEL problem, telling whether an undirected graph has a kernel with at most B nodes, is **NP**-complete. (Similar reduction as in (a).)

9.5.11 Problem: Show that MAX BISECTION remains **NP**-complete even if the given graph is connected.

9.5.12 Problem: In the STEINER TREE problem we are given distances $d_{ij} \geq 0$ between n cities, and a set $M \subseteq \{1, 2, \ldots, n\}$ of *mandatory cities*. Find the shortest possible connected graph that contains the mandatory cities. Show that STEINER TREE is **NP**-complete.

- M. R. Garey, R. L. Graham, and D. S. Johnson "The complexity of computing Steiner minimal trees," *SIAM J. Applied Math., 34,*, pp. 477–495, 1977

show that the EUCLIDEAN STEINER TREE problem (the mandatory nodes are points in the plane, the distances are the Euclidean metric, and all points in the plane are potential non-mandatory nodes; this is the original fascinating problem proposed by Georg Steiner, of which the graph-theoretic version STEINER TREE is a rather uninteresting generalization) is **NP**-complete.

9.5.13 Problem: MINIMUM SPANNING TREE is basically the STEINER TREE problem with all nodes mandatory. We are given cities and nonnegative distances, and we are asked to construct the shortest graph that connects all cities. Show that the optimum graph will have no cycles (hence the name). Show that the *greedy algorithm* (add to the graph the shortest distance not yet considered, unless it is superfluous—presumably because its endpoints are already connected) solves this problem in polynomial time.

9.5.14 Problem: Consider the following pairs of minimization-maximization problems in weighted graphs:

(a) MINIMUM SPANNING TREE and MAXIMUM SPANNING TREE (we seek the *heaviest* connecting tree).

(b) SHORTEST PATH (recall Problem 1.4.15) and LONGEST PATH (we seek the longest path with no repeating nodes; this is sometimes called the TAXICAB RIPOFF problem).

(c) MIN CUT between s and t, and MAX CUT between s and t.

(d) MAX WEIGHT COMPLETE MATCHING (in a bipartite graph with edge weights), and MIN WEIGHT COMPLETE MATCHING.

(e) TSP, and the version in which the *longest* tour must be found.

Which of these pairs are polynomially equivalent and which are not? Why?

9.5.15 Problem: (a) Show that the HAMILTON CYCLE problem (we are now seeking a cycle that visits all nodes once) is **NP**-complete. (Modify the proof of Theorem 9.7 a little.)

(b) Show that HAMILTON CYCLE remains **NP**-complete even if the graphs are restricted to be planar, bipartite, and cubic (all degrees three). (Start with the previous reduction, and patiently remove all violations of the three conditions. For planarity,

notice that only exclusive-or gadgets intersect; find a way to have two of them "cross over" with no harm to their function. Alternatively, see Problem 9.5.16.)

(c) A *grid graph* is a finite, induced subgraph of the infinite two-dimensional grid. That is, the nodes of the graph are pairs of integers, and $[(x, y), (x', y')]$ is an edge if and only if $|x - x'| + |y - y'| = 1$. Show that the Hamilton cycle problem for grid graphs is **NP**-complete. (Start with the construction in (b) and embed the graph in the grid. Simulate nodes with small squares, and edges with "tentacles," long strips of width two.)

(d) Conclude from (c) that the *Euclidean* special case of the TSP (the distances are actual Euclidean distances between cities on the plane) is **NP**-complete. (There is a slight problem with defining the Euclidean TSP: To what precision do we need to calculate square roots?)

Part (c) is from

- A. Itai, C. H. Papadimitriou, J. L. Szwarcfiter "Hamilton paths in grid graphs," *SIAM J. Comp., 11*, 3, pp. 676–686, 1982.

Part (d) was first proved in

- C. H. Papadimitriou "The Euclidean traveling salesman problem is **NP**-complete," *Theor. Comp. Sci 4*, pp. 237–244, 1977, and independently in
- M. R. Garey, R. L. Graham, and D. S. Johnson "Some NP-complete geometric problems," in *Proc. 8th Annual ACM Symp. on the Theory of Computing*, pp. 10–22, 1976.

Also, Theorems 9.2 and 9.5 on MAX2SAT and MAX CUT are from

- M. R. Garey, D. S. Johnson, and L. J. Stockmeyer "Some simplified NP-complete graph problems," *Theor. Comp. Sci., 1*, pp. 237-267, 1976.

It is open whether HAMILTON CYCLE is **NP**-complete for *solid* grid graphs, that is, grid graphs without "holes."

9.5.16 In many applications of graph algorithms, such as in vehicle routing and integrated circuit design, the underlying graph is always *planar*, that is, it can be drawn on the plane with no edge crossings. It is of interest to determine which **NP**-complete graph problems retain their complexity when restricted to planar graphs. In this regard, a special case of SAT is especially interesting: The *occurrence graph* of an instance of SAT is the graph that has as nodes all variables and clauses, and has an edge from a variable to a clause if the variable (or its negation) appears in the clause. We say that an instance of SAT is *planar* if its occurrence graph is planar. It was shown in

- D. Lichtenstein "Planar formulae and their uses," *SIAM J. Comp., 11*, pp. 329–393, 1982.

that this special case, called PLANAR SAT, is **NP**-complete even if all clauses have at most three literals, and each variable appears at most five times.

Problem: Use this result to show that INDEPENDENT SET, NODE COVER, and HAMILTON PATH remain **NP**-complete even if the underlying graphs are planar.

How about CLIQUE?

It is open whether BISECTION WIDTH (with unit weights on the edges) is **NP**-complete for planar graphs.

9.5.17 Problem: The *line graph* of a graph $G = (V, E)$ is a graph $L(G) = (E, H)$, where $[e, e'] \in H$ if and only if e and e' are adjacent. Show that HAMILTON PATH is **NP**-complete for line graphs (despite the obvious connection to Eulerian graphs).

9.5.18 Problem: The CYCLE COVER problem is this: Given a directed graph, is there a set of node-disjoint cycles that covers all nodes?

(a) Show that the CYCLE COVER problem can be solved in polynomial time (it is a disguise of MATCHING).

(b) Suppose now that we do not allow cycles of length two in our cycle cover. Show that the problem now becomes **NP**-complete. (Reduction from 3SAT. For a similar proof see Theorem 18.3.)

(c) Show that there is an integer k such that the following problem is **NP**-complete: Given an undirected graph, find a cycle cover without cycles of length k or less. (Modify the proof that HAMILTON CYCLE is **NP**-complete.) What is the smallest value of k for which you can prove **NP**-completeness?

(d) Show that the directed Hamilton path problem is polynomial when the directed graph is *acyclic*. Extend to the case in which there are no cycles with fewer than $\frac{n}{2}$ nodes.

9.5.19 Problem: We are given a directed graph with weights w on the edges, two nodes s and t, and an integer B. We wish to find two node-disjoint paths from s to t, both of length at most B. Show that the problem is **NP**-complete. (Reduction from 3SAT. The problem is polynomial when we wish to minimize the sum of the lengths of the two disjoint paths.)

9.5.20 Problem: The CROSSWORD PUZZLE problem is this: We are given an integer n, a subset $B \subseteq \{1, \ldots, n\}^2$ of *black squares*, and a finite *dictionary* $D \subseteq \Sigma^*$. We are asked whether there is a mapping F from $\{1, \ldots, n\}^2 - B$ to Σ such that all *maximal* words of the form $(F(i,j), F(i,j+1), \ldots, F(i,j+k))$ and $(F(i,j), F(i+1,j), \ldots, F(i+k,j))$, are in D. Show that CROSSWORD PUZZLE is **NP**-complete. (It remains **NP**-complete even if $B = \emptyset$.)

9.5.21 Problem: The ZIGSAW PUZZLE problem is the following: We are given a set of polygonal pieces (say, in terms of the sequence of the integer coordinates of the vertices of each). We are asked if there is a way to arrange these pieces on the plane so that (a) no two of them overlap, and (b) their union is a square. Show that ZIGSAW PUZZLE is **NP**-complete. (Start from a version of the TILING problem, Problem 20.2.10, in which some tiles may remain unused. Then simulate tiles by square pieces, with small *square indentations* on each side. The position of the indentations reflects the sort of the tile and forces horizontal and vertical compatibility. Add appropriate small pieces, enough to fill these indentations.)

9.5.22 Problem: Let G be a directed graph. The *transitive closure* of G, denoted

G^*, is the graph with the same set of nodes as G, but which has an edge (u, v) whenever there is a path from u to v in G.

(a) Show that the transitive closure of a graph can be computed in polynomial time. Can you compute it in $\mathcal{O}(n^3)$ time? (Recall Example 8.2. Even faster algorithms are possible.)

(b) The *transitive reduction* of G, $R(G)$, is the graph H with the fewest edges such that $H^* = G^*$. Show that computing the transitive reduction of G is *equivalent* to computing the transitive closure, and hence can be done in $\mathcal{O}(n^3)$ time. (This is from

- A. V. Aho, M. R. Garey, and J. D. Ullman "The transitive reduction of a directed graph," *SIAM J. Comp., 1*, pp. 131–137, 1972.)

(c) Define now the *strong transitive reduction* of G to be the *subgraph* H of G with the fewest edges such that $H^* = G^*$. Show that telling whether the strong transitive reduction of G has K or fewer edges is **NP**-complete. (What is the strong transitive reduction of a strongly connected graph?)

9.5.23 Problem: (a) We are given two graphs G and H, and we are asked whether there is a subgraph of G isomorphic to H. Show that this problem is **NP**-complete.

(b) Show that the problem remains **NP**-complete even if the graph H is restricted to be a tree with the same number of nodes as G.

(c) Show that the problem remains **NP**-complete even if the tree is restricted to have diameter six. (Show first that it is **NP**-complete to tell if a graph with $3m$ nodes has a subgraph which consists of m node-disjoint paths of length two.)

Part (c) and several generalizations are from

- C. H. Papadimitriou and M. Yannakakis "The complexity of restricted spanning tree problems," *J.ACM, 29*, 2, pp. 285-309, 1982.

9.5.24 Problem: We are given a graph G and we are asked if it has as a subgraph a tree T with as many nodes as G (a spanning tree of G, that is), such that the set of *leaves* of T (nodes of degree one) is

(a) of cardinality equal to a given number K.
(b) of cardinality less than a given number K.
(c) of cardinality greater than a given number K.
(d) equal to a given set of nodes L.
(e) a subset of a given set of nodes L.
(f) a superset of a given set of nodes L.

Consider also the variants in which all nodes of T are required to have degree

(g) at most two.
(h) at most a given number K.
(j) equal to an odd number.

Which of these versions are **NP**-complete, and which are polynomial?

9.5.25 Problem: (a) Suppose that in a bipartite graph each set B of boys is adjacent to a set $g(B)$ of girls with $|g(B)| \geq |B|$. Show that the bipartite graph has a matching.

(Start from the Max-Flow Min-Cut Theorem and exploit the relationship between the two problems.)

(b) Show that the INDEPENDENT SET problem for bipartite graphs can be solved in polynomial time.

9.5.26 Interval graphs. Let $G = (V, E)$ be an undirected graph. We say that G is an *interval graph* if there is a path P (a graph with nodes $1, \ldots, m$ for some $m > 1$, and edges of the form $[i, i+1], i = 1, \ldots, m-1$) and a mapping f from V to the set of subpaths (connected subgraphs) of P such that $[v, u] \in E$ if and only if $f(u)$ and $f(v)$ have a node in common.

(a) Show that all trees are interval graphs.

(b) Show that the following problems can be solved in polynomial time for interval graphs: CLIQUE, COLORING, and INDEPENDENT SET.

9.5.27 Chordal graphs. Let $G = (V, E)$ be an undirected graph. We say that G is *chordal* if each cycle $[v_1, v_2, \ldots, v_k, v_1]$ with $k > 3$ distinct nodes has a *chord*, that is, an edge $[v_i, v_j]$ with $j \neq i \pm 1 \bmod k$.

(a) Show that interval graphs (recall the previous problem) are chordal.

A *perfect elimination sequence* of a graph G is a permutation $(v_1, v_2, \ldots, v_n)$ of V such that for all $i \leq n$ the following is true: If $[v_i, v_j], [v_i, v_{j'}] \in E$ and $j, j' > i$, then $[v_j, v_{j'}] \in E$. That is, there is a way of deleting all nodes of the graph, one node at a time, so that the neighborhood of each deleted node is a clique in the remaining graph.

(b) Let A be a symmetric matrix. The *sparsity graph* of A, $G(A)$, has the rows of A as nodes, and an edge from i to j if and only if A_{ij} is non-zero. We say that A has a *fill-in-free elimination* if there is a permutation of rows and columns of A such that the resulting matrix can be made upper diagonal using Gaussian elimination in such a way that *all zero entries of A remain zero*, regardless of the precise values of the nonzero elements.

Finally, we say that G has a *tree model* if there is a tree T and a mapping f from V to the set of subtrees (connected subgraphs) of T such that $[v, u] \in E$ if and only if $f(u)$ and $f(v)$ have a node in common.

(c) Show that the following are equivalent for a graph G:
(i) G is chordal.
(ii) G has a perfect elimination sequence.
(iii) G has a tree model.

(Notice that the equivalence of (i) and (iii) provides a proof of part (a) above. To show that (ii) implies (i), consider a chordless cycle and the first node in it to be deleted. To show that (iii) implies (ii), consider all nodes u of G such that $f(u)$ contains a leaf of T. That (i) implies (iii) is tedious.)

(d) Show that the following problems can be solved in polynomial time for chordal graphs: CLIQUE, COLORING, and INDEPENDENT SET.

(e) The clique number $\omega(G)$ of a graph G is the number of nodes in its largest clique.

The chromatic number $\chi(G)$ is the minimum number of colors that are required to color the nodes of the graph so that no two adjacent nodes have the same color. Show that for all G $\chi(G) \geq \omega(G)$.

A graph G is called *perfect* if for all of its induced subgraphs G' $\chi(G') = \omega(G')$.

(f) Show that interval graphs, chordal graphs, and bipartite graphs are perfect.

For much more on perfect graphs, their special cases, and their positive algorithmic properties, see

- M. C. Golumbic *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

9.5.28 Problem: Show that 3-COLORING remains **NP**-complete even when the graph is planar. (Again, we must replace crossings with suitable graphs. This **NP**-completeness result is remarkable because telling whether any graph can be colored with two colors is easy (why?), and coloring a planar graph with four colors is always possible, see

- K. Appel and W. Haken "Every planar map is 4-colorable," *Illinois J. of Math., 21*, pp. 429–490 and 491–567, 1977.)

9.5.29 Disjoint paths in graphs. In the problem DISJOINT PATHS we are given a directed graph G, and a set of pairs of nodes $\{(s_1, t_1), \ldots, (s_k, t_k)\}$. We are asked whether there are *node-disjoint paths* from s_i to t_i, $i = 1, \ldots, k$.

(a) Show that DISJOINT PATHS is **NP**-complete. (From 3SAT. The graph you construct has for each variable two endpoints and two separate parallel paths between them, and for each clause two endpoints connected by three paths that correspond to the literals in the clause. These two kinds of paths should intersect in the appropriate way.)

(b) Show that DISJOINT PATHS is **NP**-complete even if the graph is planar. (You may wish to start from PLANAR SAT (see 9.5.16 above). Or, introduce new endpoints at each crossing.)

(c) Show that DISJOINT PATHS is **NP**-complete even if the graph is undirected. These results were first shown in

- J. F. Lynch "The equivalence of theorem proving and the interconnection problem," *ACM SIGDA Newsletter, 5*, 3, pp. 31–36, 1975.

(d) Show that the special case of the DISJOINT PATHS problem, in which all sources coincide ($s_1 = s_2 = \ldots = s_k$), is in **P**.

Let $k > 1$, and define k DISJOINT PATHS to be the special case of the problem in which there are exactly k pairs.

(e) Show that the 2 DISJOINT PATHS problem is **NP**-complete. (This is a clever **NP**-completeness proof due to

- S. Fortune, J. E. Hopcroft, and J. Wyllie "The directed subgraph homeomorphism problem," *Theor. Comp. Sci., 10*, pp. 111–121, 1980.)

(f) Let H be a directed graph with k edges. The DISJOINT H-PATHS problem is the special case of k DISJOINT PATHS, in which the s_i's and t_i's are required to coincide

as the heads and tails of the edges of H. For example, if H is the directed graph with two nodes $1, 2$ and two edges $(1, 2), (2, 1)$, the DISJOINT H-PATHS problem is this: Given a directed graph and two nodes a and b, is there a simple cycle (with no repetitions of nodes) involving a and b? Using the results in (d) and (e) above show that the DISJOINT H-PATHS problem is always **NP**-complete, unless H is a tree of depth one (see part (d) above), in which case it is in **P**.

The undirected special case of k DISJOINT PATHS is in **P** *for all* k. See

- N. Robertson and P. D. Seymour "Graph minors XIII: The disjoint paths problem," thirteenth part of a series of twenty papers, of which nineteen appear in *J. Combinatorial Theory, Ser. B, 35*, 1983, and thereafter; the second paper in the series appeared in the *J. of Algorithms, 7*, 1986.

This powerful and surprising result is just one step along the way of proving perhaps the most important and sweeping result in algorithmic graph theory to-date, namely that *all graph-theoretic properties that are closed under minors are in* **P**. We say that a graph-theoretic property is closed under minors if, whenever G has the property, then so does any graph that results from G by (a) deleting a node, and (b) collapsing two adjacent nodes. See the sequence of papers referenced above.

(g) Suppose that H is the directed graph consisting of a single node and *two self-loops*. Then the DISJOINT H-PATHS problem is this: Given a directed graph and a node a, are there two disjoint cycles through a? By (f) above, it is **NP**-complete. Show that it can be solved in polynomial time for *planar* directed graphs.

In fact, it turns out that the DISJOINT H-PATHS problem can be solved in polynomial time for planar graphs *for any* H; see

- A. Schrijver "Finding k disjoint paths in a directed planar graph," Centrum voor Wiskunde en Informatica Report BS-R9206, 1992.

9.5.30 The BANDWIDTH MINIMIZATION problem is this: We are given an undirected graph $G = (V, E)$ and an integer B. We are asked whether there is a permutation $(v_1, v_2, \ldots, v_n)$ of V such that $[v_i, v_j] \in E$ implies $|i - j| < B$. This problem is **NP**-complete, see

- C. H. Papadimitriou "The NP-completeness of the bandwidth minimization problem," *Computing, 16*, pp. 263–270, 1976, and
- M. R. Garey, R. L. Graham, D. S. Johnson, and D. E. Knuth "Complexity results for bandwidth minimization," *SIAM J. Appl. Math., 34*, pp. 477–495, 1978.

In fact, the latter paper shows that BANDWIDTH MINIMIZATION remains **NP**-complete *even if* G *is a tree*.

Problem: (a) Show that *all other graph-theoretic problems* we have seen in this chapter, including this section, can be solved in polynomial time if the given graph is a tree.

(b) Show that the BANDWIDTH MINIMIZATION problem with B fixed to any bounded integer can be solved in polynomial time. (This follows of course from the result on minor-closed properties discussed in 9.5.29 above; but a simple dynamic programming algorithm is also possible.)

9.5.31 Pseudopolynomial algorithms and strong NP-completeness. There is a difficulty in formalizing the concept of pseudopolynomial algorithms and strong **NP**-completeness discussed in Section 9.4. The difficulty is this: Strictly speaking, inputs are dry, uninterpreted strings operated on by Turing machines. The fact that certain parts of the input encode binary integers should be completely transparent to our treatment of algorithms and their complexity. How can we speak about the size of integers in the input without harming our convenient abstraction and representation-independence?

Suppose that each string $x \in \Sigma^*$ has, except for its length $|x|$, another integer value NUM(x) associated with it. All we know about this integer is that, for all x, NUM(x) can be computed in polynomial time from x, and that $|x| \leq \text{NUM}(x) \leq 2^{|x|}$. We say that a Turing machine operates in *pseudopolynomial time* if for all inputs x the number of steps of the machine is $p(\text{NUM}(x))$ for some polynomial $p(n)$. We say that a language L is *strongly* **NP**-*complete* if there is a polynomial $q(n)$ such that the following language is **NP**-complete: $L_{q(n)} = \{x \in L : \text{NUM}(x) \leq q(|x|)\}$.

Problem: (a) Find a plausible definition of NUM(x) for KNAPSACK. Show that the algorithm in Proposition 9.4 is pseudopolynomial.

(b) Find a plausible definition of NUM(x) for BIN PACKING. Show that BIN PACKING is strongly **NP**-complete.

(c) Show that, if there is a pseudopolynomial algorithm for a strongly **NP**-complete problem, then **P** = **NP**.

Naturally, the pseudopolynomiality vs. strong **NP**-completeness information we obtain is only as good as our definition of NUM(x); and in any problem one can come up with completely implausible and far-fetched definitions of NUM(x). The point is, *we can always come up with a good one* (namely, "the largest integer encoded in the input"), for which the dichotomy of Part (c) above is both meaningful and informative.

9.5.32 Problem: Let $k \geq 2$ be a fixed integer. The k-PARTITION problem is the following special case of BIN PACKING (recall Theorem 9.11): We are given $n = km$ integers $a_1, \ldots, a_n$, adding up to mC, and such that $\frac{C}{k+1} < a_i < \frac{C}{k-1}$ for all i. That is, the numbers are such that their sum fits exactly in m bins, but no $k+1$ of them fit into one bin, neither can any $k-1$ of them fill a bin. We are asked whether we can find a partition of these numbers into m groups of k, such that the sum in each group is precisely C.

(a) Show that 2-PARTITION is in **P**.

(b) Show that 4-PARTITION is **NP**-complete. (The proof of Theorem 9.11 basically establishes this.)

(c) Show that 3-PARTITION is **NP**-complete. (This requires a clever reduction of 4-PARTITION to 3-PARTITION.)

Part (c), the **NP**-completeness of 3-PARTITION, is from

- M. R. Garey and D. S. Johnson "Complexity results for multiprocessor scheduling with resource constraints," *SIAM J. Comp., 4*, pp. 397–411, 1975.

9.5.33 (a) Show that the following problem is **NP**-complete: Given n integers adding

up to $2K$, is there a subset that adds up to exactly K? This is known as the PARTITION problem, not to be confused with the k-PARTITION problem above. (Start from KNAPSACK, and add appropriate new items.)

(b) INTEGER KNAPSACK is this problem: We are given n integers and a goal K. We are asked whether we can choose *zero, one,* or *more* copies of each number so that the resulting multiset of integers adds up to K. Show that this problem is **NP**-complete. (Modify an instance of the ordinary KNAPSACK problem so that each item can be used at most once.)

9.5.34 Linear and integer programming. INTEGER PROGRAMMING is the problem of deciding whether a given system of linear equations has a nonnegative integer solution. It is of course **NP**-complete, as just about *all* **NP**-complete problems easily reduce to it... Actually, the difficult part here is showing that it is in **NP**; but it can be done, see

- C. H. Papadimitriou "On the complexity of integer programming", *J.ACM, 28*, 2, pp. 765-769, 1981.

In fact, in the paper above it is also shown that there is a pseudopolynomial algorithm for INTEGER PROGRAMMING when the number of equations is bounded by a constant, thus generalizing Proposition 9.4. (Naturally, the general INTEGER PROGRAMMING problem is strongly **NP**-complete.)

A different but equivalent formulation of INTEGER PROGRAMMING is in terms of a system of *inequalities* instead of equalities, and variables unrestricted in sign. For this form we have a more dramatic result: When the number of variables is bounded by a constant, there is a *polynomial-time* algorithm for the problem, based on the important *basis reduction technique*; see

- A. K. Lenstra, H. W. Lenstra, and L. Lovász "Factoring polynomials with rational coefficients," *Math. Ann, 261*, pp. 515–534, 1982, and
- M. Grötschel, L. Lovász, and A. Schrijver *Geometric Algorithms and Combinatorial Optimization,* Springer, Berlin, 1988.

In contrast, linear programming (the version in which we are allowed to have fractional solutions), is much easier: Despite the fact that the classical, empirically successful, and influential *simplex method*, see

- G. B. Dantzig *Linear Programming and Extensions*, Princeton Univ. Press, Princeton, N.J., 1963,

is exponential in the worst-case, polynomial-time algorithms have been discovered. The first polynomial algorithm for linear programming was the *ellipsoid method*

- L. G. Khachiyan "A polynomial algorithm for linear programming," *Dokl. Akad. Nauk SSSR, 244*, pp. 1093–1096, 1979. English Translation *Soviet Math. Doklad 20*, pp. 191–194, 1979;

while a more recent algorithm seems to be much more promising in practice:

- N. Karmarkar "A new polynomial-time algorithm for linear programming," *Combinatorica, 4*, pp. 373–395, 1984.

See also the books

- A. Schrijver *Theory of Linear and Integer Programming,* Wiley, New York, 1986, and
- C. H. Papadimitriou and K. Steiglitz *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

Problem: (a) Show that any instance of SAT can be expressed very easily as an instance of INTEGER PROGRAMMING with inequalities. Conclude that INTEGER PROGRAMMING is **NP**-complete even if the inequalities are known to have a fractional solution. (Start with an instance of SAT with at least two distinct literals per clause.)

(b) Express the existence of an integer flow of value K in a network with integer capacities as a set of linear inequalities.

(c) Is the MAX FLOW problem a special case of linear, or of integer programming? (On the surface it appears to be a special case of integer programming, since integer flows are required; but a little thought shows that the optimum solution will always be integer anyway—assuming all capacities are. So, the integrality constraint is superfluous.)