9

Intractable Problems and Approximation Algorithms

We now introduce techniques for proving that *no* efficient algorithm exists for a given problem. The practical reader is probably squirming at the notion of proving anything, and will be particularly alarmed at the idea of investing time to prove that something does not exist. Why are you better off knowing that something you don't know how to do in fact can't be done at all?

The truth is that the theory of NP-completeness is an immensely useful tool for the algorithm designer, even though all it provides are negative results. The theory of NP-completeness enables the algorithm designer to focus her efforts more productively, by revealing that the search for an efficient algorithm for this particular problem is doomed to failure. When one *fails* to show a problem is hard, that suggests there may well be an efficient algorithm to solve it. Two of the war stories in this book described happy results springing from bogus claims of hardness.

The theory of NP-completeness also enables us to identify what properties make a particular problem hard. This provides direction for us to model it in different ways or exploit more benevolent characteristics of the problem. Developing a sense for which problems are hard is an important skill for algorithm designers, and only comes from hands-on experience with proving hardness.

The fundamental concept we will use is that of *reductions* between pairs of problems, showing that the problems are really equivalent. We illustrate this idea through a series of reductions, each of which either yields an efficient algorithm or an argument that no such algorithm can exist. We also provide brief introductions to (1) the complexity-theoretic aspects of NP-completeness, one of the most fundamental notions in Computer Science, and (2) the theory of approximation algorithms, which leads to heuristics that probably return something *close* to the optimal solution.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_9,

9.1 Problems and Reductions

We have encountered several problems in this book for which we couldn't find any efficient algorithm. The theory of NP-completeness provides the tools needed to show that all these problems are on some level really the same problem.

The key idea to demonstrating the hardness of a problem is that of a *reduction*, or translation, between two problems. The following allegory of NP-completeness may help explain the idea. A bunch of kids take turns fighting each other in the schoolyard to prove how "tough" they are. Adam beats up Bill, who then beats up Chuck. So who if any among them is "tough?" The truth is that there is no way to know without an external standard. If I told you that Chuck was in fact Chuck Norris, certified tough guy, you have to be impressed—both Adam and Bill are at least as tough as he is. On the other hand, suppose I tell you it is a kindergarten school yard. No one would call me tough, but even I can take out Adam. This proves that none of the three of them should be called be tough. In this allegory, each fight represents a reduction. Chuck Norris takes on the role of satisfiability—a certifiably hard problem. The part of an inefficient algorithm with a possible shot at redemption is played by me.

Reductions are operations that convert one problem into another. To describe them, we must be somewhat rigorous in our definitions. An algorithmic *problem* is a general question, with parameters for input and conditions on what constitutes a satisfactory answer or solution. An *instance* is a problem with the input parameters specified. The difference can be made clear by an example:

Problem: The Traveling Salesman Problem (TSP)
Input: A weighted graph G.
Output: Which tour $\{v_1, v_2, ..., v_n\}$ minimizes $\sum_{i=1}^{n-1} d[v_i, v_{i+1}] + d[v_n, v_1]$?

Any weighted graph defines an instance of TSP. Each particular *problem* has at least one minimum cost tour. The general traveling salesman *instance* asks for an algorithm to find the optimal tour for all possible instances.

9.1.1 The Key Idea

Now consider two algorithmic problems, called *Bandersnatch* and *Bo-billy*. Suppose that I gave you the following reduction/algorithm to solve the *Bandersnatch* problem:

Bandersnatch(G)
    Translate the input G to an instance Y of the Bo-billy problem.
    Call the subroutine Bo-billy on Y to solve this instance.
    Return the answer of Bo-billy(Y) as the answer to Bandersnatch(G).

This algorithm will *correctly* solve the Bandersnatch problem provided that the translation to Bo-billy always preserves the correctness of the answer. In other words, the translation has the property that for any instance of G,

$$\text{Bandersnatch}(G) = \text{Bo-billy}(Y)$$

A translation of instances from one type of problem to instances of another such that the answers are preserved is what is meant by a *reduction*.

Now suppose this reduction translates G to Y in $O(P(n))$ time. There are two possible implications:

- *If* my Bo-billy subroutine ran in $O(P'(n))$, this means I could solve the Bandersnatch problem in $O(P(n) + P'(n))$ by spending the time to translate the problem and then the time to execute the Bo-Billy subroutine.
- *If* I know that $\Omega(P'(n))$ is a lower bound on computing Bandersnatch, meaning there definitely exists no faster way to solve it, then $\Omega(P'(n) - P(n))$ *must* be a lower bound to compute Bo-billy. Why? If I could solve Bo-billy any faster, then I could violate my lower bound by solving Bandersnatch using the above reduction. This implies that there can be no way to solve Bo-billy any faster than claimed.

This first argument is Steve demonstrating the weakness of the entire schoolyard with a quick right to Adam's chin. The second highlights the Chuck Norris approach we will use to prove that problems are hard. Essentially, this reduction shows that Bo-billy is no easier than Bandersnatch. Therefore, if Bandersnatch is hard this means Bo-billy must also be hard.

We will illustrate this point by giving several problem reductions in this chapter.

> *Take-Home Lesson:* Reductions are a way to show that two problems are essentially identical. A fast algorithm (or the lack of one) for one of the problems implies a fast algorithm (or the lack of one) for the other.

9.1.2 Decision Problems

Reductions translate between problems so that their answers are identical in every problem instance. Problems differ in the *range* or *type* of possible answers. The traveling salesman problem returns a permutation of vertices as the answer, while other types of problems return numbers as answers, perhaps restricted to positive numbers or integers.

The simplest interesting class of problems have answers restricted to true and false. These are called *decision problems*. It proves convenient to reduce/translate answers between decision problems because both only allow true and false as possible answers.

Fortunately, most interesting optimization problems can be phrased as decision problems that capture the essence of the computation. For example, the traveling salesman decision problem could be defined as:

Problem: The Traveling Salesman Decision Problem
Input: A weighted graph G and integer k.
Output: Does there exist a TSP tour with cost $\leq k$?

The decision version captures the heart of the traveling salesman problem, in that if you had a fast algorithm for the decision problem, you could use it to do a binary search with different values of k and quickly hone in on the optimal solution. With a little more cleverness, you could reconstruct the actual tour permutation using a fast solution to the decision problem.

From now on we will generally talk about decision problems, because it proves easier and still captures the power of the theory.

9.2 Reductions for Algorithms

An engineer and an algorist are sitting in a kitchen. The algorist asks the engineer to boil some water, so the engineer gets up, picks up the kettle from the counter top, adds water from the sink, brings it to the burner, turns on the burner, waits for the whistling sound, and turns off the burner. Sometime later, the engineer asks the algorist to boil more water. She gets up, takes the kettle from the burner, moves it over to the counter top, and sits down. "Done." she says, "I have *reduced* the task to a solved problem."

This boiling water reduction illustrates an honorable way to generate new algorithms from old. If we can translate the input for a problem we *want to solve* into input for a problem we *know how to solve*, we can compose the translation and the solution into an algorithm for our problem.

In this section, we look at several reductions that lead to efficient algorithms. To solve problem a, we translate/reduce the a instance to an instance of b, then solve this instance using an efficient algorithm for problem b. The overall running time is the time needed to perform the reduction plus that solve the b instance.

9.2.1 Closest Pair

The *closest pair* problem asks to find the pair of numbers within a set that have the smallest difference between them. We can make it a decision problem by asking if this value is less than some threshold:

Input: A set S of n numbers, and threshold t.
Output: Is there a pair $s_i, s_j \in S$ such that $|s_i - s_j| \leq t$?

The closest pair is a simple application of sorting, since the closest pair must be neighbors after sorting. This gives the following algorithm:

CloseEnoughPair(S,t)
    Sort S.
    Is $\min_{1 \leq i < n} |s_i - s_{i+1}| \leq t$?

There are several things to note about this simple reduction.

1. The decision version captured what is interesting about the problem, meaning it is no easier than finding the actual closest pair.
2. The complexity of this algorithm depends upon the complexity of sorting. Use an $O(n \log n)$ algorithm to sort, and it takes $O(n \log n + n)$ to find the closest pair.
3. This reduction and the fact that there is an $\Omega(n \log n)$ lower bound on sorting *does not* prove that a close-enough pair must take $\Omega(n \log n)$ time in the worst case. Perhaps this is just a slow algorithm for a close-enough pair, and there is a faster one lurking somewhere?
4. On the other hand, *if* we knew that a close-enough pair required $\Omega(n \log n)$ time to solve in the worst case, this reduction would suffice to prove that sorting couldn't be solved any faster than $\Omega(n \log n)$ because that would imply a faster algorithm for a close-enough pair.

9.2.2 Longest Increasing Subsequence

In Chapter 8, we demonstrated how dynamic programming can be used to solve a variety of problems, including string edit distance (Section 8.2 (page 280)) and longest increasing subsequence (Section 8.3 (page 289)). To review,

Problem: Edit Distance
Input: Integer or character sequences S and T; penalty costs for each insertion (c_{ins}), deletion (c_{del}), and substitution (c_{del}).
Output: What is the minimum cost sequence of operations to transform S to T?

Problem: Longest Increasing Subsequence
Input: An integer or character sequence S.
Output: What is the longest sequence of integer positions $\{p_1, \ldots, p_m\}$ such that $p_i < p_{i+1}$ and $S_{p_i} < S_{p_{i+1}}$?

In fact, longest increasing subsequence (LIS) can be solved as a special case of edit distance:

```
Longest Increasing Subsequence(S)
    T = Sort(S)
    c_ins = c_del = 1
    c_sub = ∞
    Return (|S| - EditDistance(S,T,c_ins,c_del,c_del)/2)
```

Why does this work? By constructing the second sequence T as the elements of S sorted in increasing order, we ensure that any common subsequence must be an

increasing subsequence. If we are never allowed to do any substitutions (because $c_{sub} = \infty$), the optimal alignment of two sequences finds the longest common subsequence between them and removes everything else. Thus, transforming $\{3, 1, 2\}$ to $\{1, 2, 3\}$ costs two, namely inserting and deleting the unmatched 3. The length of S minus half this cost gives the length of the LIS.

What are the implications of this reduction? The reduction takes $O(n \log n)$ time. Because edit distance takes time $O(|S| \cdot |T|)$, this gives a quadratic algorithm to find the longest increasing subsequence of S, which is the same complexity as the the algorithm presented in Section 8.3 (page 289). In fact, there exists a faster $O(n \log n)$ algorithm for LIS using clever data structures, while edit distance is known to be quadratic in the worst case. Here, our reduction gives us a simple but not optimal polynomial-time algorithm.

9.2.3 Least Common Multiple

The *least common multiple* and *greatest common divisor* problems arise often in working with integers. We say b *divides* a ($b|a$) if there exists an integer d that $a = bd$. Then:

Problem: Least Common Multiple (lcm)
Input: Two integers x and y.
Output: Return the smallest integer m such that m is a multiple of x and m is also a multiple of y.

Problem: Greatest Common Divisor (gcd)
Input: Two integers x and y.
Output: Return the largest integer d such that d divides x and d divides y.

For example, $lcm(24, 36) = 72$ and $gcd(24, 36) = 12$. Both problems can be solved easily after reducing x and y to their prime factorizations, but no efficient algorithm is known for factoring integers (see Section 13.8 (page 420)). Fortunately, Euclid's algorithm gives an efficient way to solve greatest common divisor without factoring. It is a recursive algorithm that rests on two observations. First,

if $b|a$, then $gcd(a, b) = b$.

This should be pretty clear. if b divides a, then $a = bk$ for some integer k, and thus $gcd(bk, b) = b$. Second,

If $a = bt + r$ for integers t and r, then $gcd(a, b) = gcd(b, r)$.

Since $x \cdot y$ is a multiple of both x and y, $lcm(x, y) \leq xy$. The only way that there can be a smaller common multiple is if there is some nontrivial factor shared between x and y. This observation, coupled with Euclid's algorithm, provides an efficient way to compute least common multiple, namely

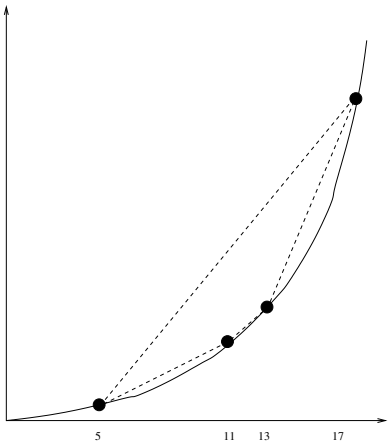


Figure 9.1: Reducing convex hull to sorting by mapping points to a parabola

LeastCommonMultiple(x,y)
Return $(xy/gcd(x, y))$.

This reduction gives us a nice way to reuse Euclid's efforts on another problem.

9.2.4 Convex Hull (*)

Our final example of a reduction from an "easy" problem (i.e. , one that can be solved in polynomial time) goes from finding convex hulls to sorting numbers. A polygon is *convex* if the straight line segment drawn between any two points inside the polygon P must lie completely within the polygon. This is the case when P contains no notches or *concavities*, so convex polygons are nicely shaped. The convex hull provides a very useful way to provide structure to a point set. Applications are presented in Section 17.2 (page 568).

Problem: Convex Hull
Input: A set S of n points in the plane.
Output: Find the smallest convex polygon containing all the points of S.

We now show how to transform from sorting to convex hull. This means we must translate each number to a point. We do so by mapping x to (x, x^2). Why? It means each integer is mapped to a point on the parabola $y = x^2$. Since this parabola is convex, every point must be on the convex hull. Furthermore, since neighboring points on the convex hull have neighboring x values, the convex hull returns the points sorted by the x-coordinate—i.e., the original numbers. Creating and reading off the points takes $O(n)$ time:

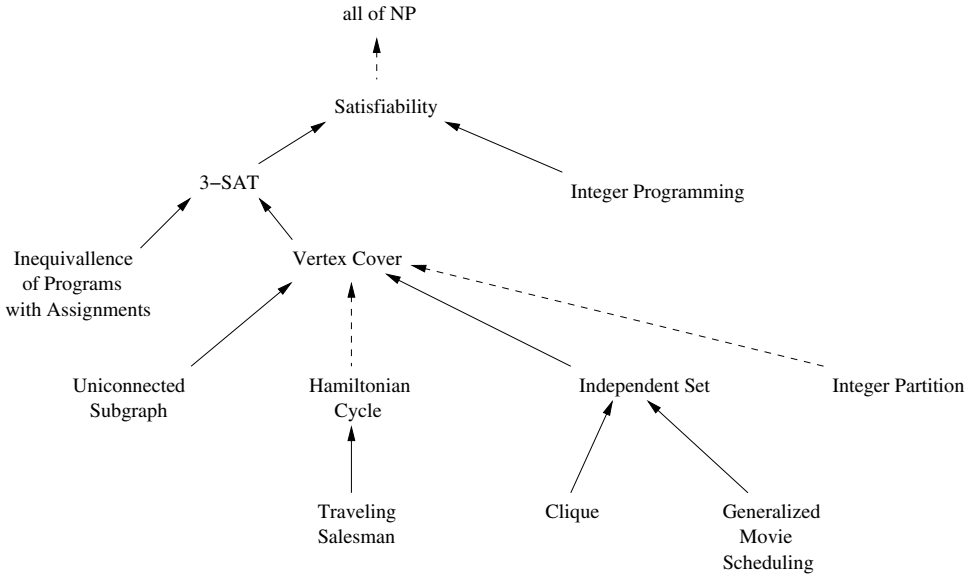


Figure 9.2: A portion of the reduction tree for NP-complete problems. Solid lines denote the reductions presented in this chapter

---

Sort(S)
    For each $i \in S$, create point (i, i^2).
    Call subroutine convex-hull on this point set.
    From the leftmost point in the hull,
        read off the points from left to right.

What does this mean? Recall the sorting lower bound of $\Omega(n \lg n)$. If we could compute convex hull in better than $n \lg n$, this reduction implies that we could sort faster than $\Omega(n \lg n)$, which violates our lower bound. Thus, convex hull must take $\Omega(n \lg n)$ as well! Observe that any $O(n \lg n)$ convex hull algorithm also gives us a complicated but correct $O(n \lg n)$ sorting algorithm as well.

9.3 Elementary Hardness Reductions

The reductions in the previous section demonstrate transformations between pairs of problems for which efficient algorithms exist. However, we are mainly concerned with using reductions to prove hardness, by showing that a fast algorithm for *Bandersnatch* would imply one that cannot exist for *Bo-billy*.

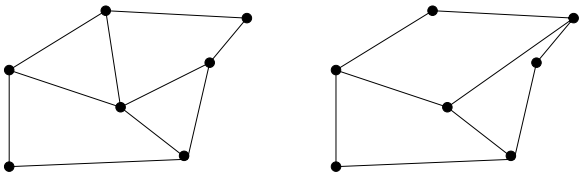


Figure 9.3: Graphs with (l) and without (r) Hamiltonian cycles

For now, I want you to trust me when I say that *Hamiltonian cycle* and *vertex cover* are hard problems. The entire picture (presented in Figure 9.2) will become clear by the end of the chapter.

9.3.1 Hamiltonian Cycle

The Hamiltonian cycle problem is one of the most famous in graph theory. It seeks a tour that visits each vertex of a given graph exactly once. It has a long history and many applications, as discussed in Section 16.5. Formally, it is defined as:

Problem: Hamiltonian Cycle
Input: An unweighted graph G.
Output: Does there exist a simple tour that visits each vertex of G without repetition?

Hamiltonian cycle has some obvious similarity to the traveling salesman problem. Both problems seek a tour that visits each vertex exactly once. There are also differences between the two problems. TSP works on weighted graphs, while Hamiltonian cycle works on unweighted graphs. The following reduction from Hamiltonian cycle to traveling salesman shows that the similarities are greater than the differences:

```
HamiltonianCycle(G = (V, E))
    Construct a complete weighted graph G' = (V', E') where V' = V.
    n = |V|
    for i = 1 to n do
        for j = 1 to n do
            if (i, j) ∈ E then w(i, j) = 1 else w(i, j) = 2
    Return the answer to Traveling-Salesman-Decision-Problem(G', n).
```

The actual reduction is quite simple, with the translation from unweighted to weighted graph easily performed in $O(n^2)$ time. Further, this translation is designed to ensure that the answers of the two problems will be identical. If the graph G has a Hamiltonian cycle $\{v_1, \ldots, v_n\}$, then this exact same tour will correspond to n edges in E', each with weight 1. This gives a TSP tour in G' of weight exactly

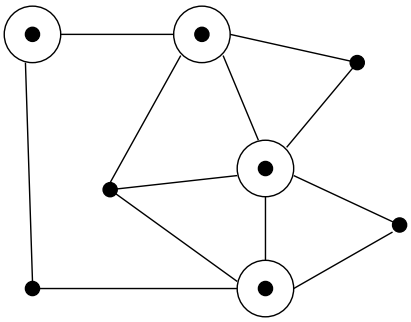


Figure 9.4: Circled vertices form a vertex cover, and the others form an independent set

n. If G does not have a Hamiltonian cycle, then there can be no such TSP tour in G' because the only way to get a tour of cost n in G would be to use only edges of weight 1, which implies a Hamiltonian cycle in G.

This reduction is both efficient and truth preserving. A fast algorithm for TSP would imply a fast algorithm for Hamiltonian cycle, while a hardness proof for Hamiltonian cycle would imply that TSP is hard. Since the latter is the case, this reduction shows that TSP is hard, at least as hard as the Hamiltonian cycle.

9.3.2 Independent Set and Vertex Cover

The vertex cover problem, discussed more thoroughly in Section 16.3 (page 530), asks for a small set of vertices that contacts each edge in a graph. More formally:

Problem: Vertex Cover
Input: A graph $G = (V, E)$ and integer $k \leq |V|$.
Output: Is there a subset S of at most k vertices such that every $e \in E$ contains at least one vertex in S?

It is trivial to find *a* vertex cover of a graph, namely the cover that consists of all the vertices. More tricky is to cover the edges using as small a set of vertices as possible. For the graph in Figure 9.4, four of the eight vertices are sufficient to cover.

A set of vertices S of graph G is *independent* if there are no edges (x, y) where both $x \in S$ and $y \in S$. This means there are no edges between any two vertices in independent set. As discussed in Section 16.2 (page 528), independent set arises in facility location problems. The maximum independent set decision problem is formally defined:

Problem: Independent Set
Input: A graph G and integer $k \leq |V|$.
Output: Does there exist an independent set of k vertices in G?

Both vertex cover and independent set are problems that revolve around finding special subsets of vertices: the first with representatives of every edge, the second with no edges. If S is the vertex cover of G, the remaining vertices $S - V$ must form an independent set, for if an edge had both vertices in $S - V$, then S could not have been a vertex cover. This gives us a reduction between the two problems:

VertexCover(G, k)
    $G' = G$
    $k' = |V| - k$
    Return the answer to IndependentSet(G', k')

Again, a simple reduction shows that the two problems are identical. Notice how this translation occurs without any knowledge of the answer. We transform the *input*, not the solution. This reduction shows that the hardness of vertex cover implies that independent set must also be hard. It is easy to reverse the roles of the two problems in this particular reduction, thus proving that both problems are equally hard.

Stop and Think: Hardness of General Movie Scheduling

Problem: Prove that the *general* movie scheduling problem is NP-complete, with a reduction from independent set.

Problem: General Movie Scheduling Decision Problem
Input: A set I of n *sets of* intervals on the line, integer k.
Output: Can a subset of at least k mutually nonoverlapping interval *sets* which can be selected from I?

Solution: Recall the movie scheduling problem, discussed in Section 1.2 (page 9). Each possible movie project came with a single time interval during which filming took place. We sought the largest possible subset of movie projects such that no two conflicting projects (i.e. , both requiring the actor at the same time) were selected.

The general problem allows movie projects to have discontinuous schedules. For example, Project A running from January-March and May-June does not intersect Project B running in April and August, but *does* collide with Project C running from June-July.

If we are going to prove general movie scheduling hard from independent set, what is Bandersnatch and what is Bo-billy? We need to show how to translate *all* independent set problems into instances of movie scheduling—i.e. , sets of disjointed line intervals.

What is the correspondence between the two problems? Both problems involve selecting the largest subsets possible—of vertices and movies, respectively. This

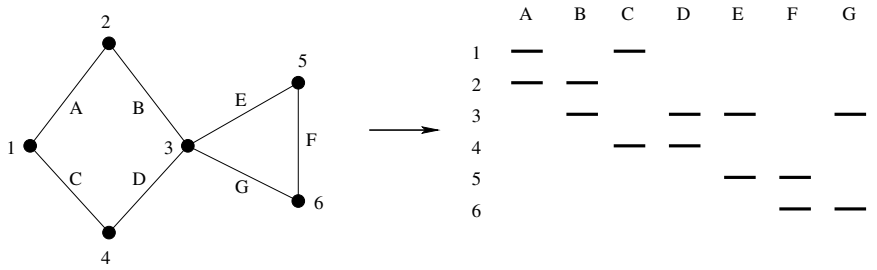


Figure 9.5: Reduction from independent set to generalized movie scheduling, with numbered vertices and lettered edges

---

suggests we must translate vertices into movies. Further, both require the selected elements not to interfere, by sharing an edge or overlapping an interval, respectively.

IndependentSet(G, k)
    $I = \emptyset$
    For the ith edge (x, y), $1 \leq i \leq m$
        Add interval $[i, i + 0.5]$ for movie X to I
        Add interval $[i, i + 0.5]$ for movie y to I
    Return the answer to GeneralMovieScheduling(I, k)

My construction is as follows. Create an interval on the line for each of the m edges of the graph. The movie associated with each vertex will contain the intervals for the edges adjacent with it, as shown in Figure 9.5.

Each pair of vertices sharing an edge (forbidden to be in independent set) defines a pair of movies sharing a time interval (forbidden to be in the actor's schedule). Thus, the largest satisfying subsets for both problems are the same, and a fast algorithm for solving general movie scheduling gives us a fast algorithm for solving independent set. Thus, general movie scheduling must be hard as hard as independent set, and hence NP-complete. ∎

9.3.3 Clique

A social clique is a group of mutual friends who all hang around together. A graph-theoretic *clique* is a complete subgraph where each vertex pair has an edge between them. Cliques are the densest possible subgraphs:

Problem: Maximum Clique
Input: A graph $G = (V, E)$ and integer $k \leq |V|$.
Output: Does the graph contain a clique of k vertices; i.e. , is there a subset $S \subset V$, where $|S| \leq k$, such that every pair of vertices in S defines an edge of G?

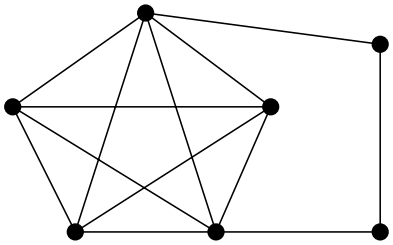


Figure 9.6: A small graph with a five-vertex clique

The graph in Figure 9.6 contains a clique of five vertices. Within the friendship graph, we would expect to see large cliques corresponding to workplaces, neighborhoods, religious organizations, and schools. Applications of cliques are further discussed in Section 16.1 (page 525).

In the independent set problem, we looked for a subset S with no edges between two vertices of S. This contrasts with clique, where we insist that there always be an edge between two vertices. A reduction between these problems follows by reversing the roles of edges and nonedges—an operation known as *complementing* the graph:

IndependentSet(G, k)
    Construct a graph $G' = (V', E')$ where $V' = V$, and
        For all (i, j) not in E, add (i, j) to E'
    Return the answer to Clique(G', k)

These last two reductions provide a chain linking of three different problems. The hardness of clique is implied by the hardness of independent set, which is implied by the hardness of vertex cover. By constructing reductions in a chain, we link together pairs of problems in implications of hardness. Our work is done as soon as all these chains begin with a single problem that is accepted as hard. Satisfiability is the problem that will serve as the first link in this chain.

9.4 Satisfiability

To demonstrate the hardness of all problems using reductions, we must start with a single problem that is absolutely, certifiably, undeniably hard. The mother of all NP-complete problems is a logic problem named *satisfiability*:

Problem: Satisfiability
Input: A set of Boolean variables V and a set of clauses C over V.
Output: Does there exist a satisfying truth assignment for C—i.e. , a way to set the variables $v_1, \ldots, v_n$ true or false so that each clause contains at least one true literal?

This can be made clear with two examples. Suppose that $C = \{\{v_1, \overline{v}_2\}, \{\overline{v}_1, v_2\}\}$ over the Boolean variables $V = \{v_1, v_2\}$. We use $\overline{v}_i$ to denote the complement of the variable v_i, so we get credit for satisfying a particular clause containing v_i if $v_i =$ true, or a clause containing $\overline{v}_i$ if $v_i =$ false. Therefore, satisfying a particular set of clauses involves making a series of n true or false decisions, trying to find the right truth assignment to satisfy all of them.

These example clauses $C = \{\{v_1, \overline{v}_2\}, \{\overline{v}_1, v_2\}\}$ can be satisfied by either setting $v_1 = v_2 =$ true or $v_1 = v_2 =$ false. However, consider the set of clauses $C = \{\{v_1, v_2\}, \{v_1, \overline{v}_2\}, \{\overline{v}_1\}\}$. Here there can be no satisfying assignment, because v_1 must be false to satisfy the third clause, which means that v_2 must be false to satisfy the second clause, which then leaves the first clause unsatisfiable. Although you try, and you try, and you try, and you try, you can't get no satisfaction.

For a combination of social and technical reasons, it is well accepted that satisfiability is a hard problem; one for which no worst-case polynomial-time algorithm exists. Literally every top-notch algorithm expert in the world (and countless lesser lights) have directly or indirectly tried to come up with a fast algorithm to test whether a given set of clauses is satisfiable. All have failed. Furthermore, many strange and impossible-to-believe things in the field of computational complexity have been shown to be true if there exists a fast satisfiability algorithm. Satisfiability is a hard problem, and we should feel comfortable accepting this. See Section 14.10 (page 472) for more on the satisfiability problem and its applications.

9.4.1 3-Satisfiability

Satisfiability's role as the first NP-complete problem implies that the problem is hard to solve in the worst case. But certain special-case instances of the problem are not necessarily so tough. Suppose that each clause contains exactly one literal. We must appropriately set that literal to satisfy such a clause. We can repeat this argument for every clause in the problem instance. Thus only when we have two clauses that directly contradict each other, such as $C = \{\{v_1\}, \{\overline{v}_1\}\}$, will the set not be satisfiable.

Since clause sets with only one literal per clause are easy to satisfy, we are interested in slightly larger classes. How many literals per clause do you need to turn the problem from polynomial to hard? This transition occurs when each clause contains three literals, i.e.

Problem: 3-Satisfiability (3-SAT)
Input: A collection of clauses C where each clause contains exactly 3 literals, over a set of Boolean variables V.
Output: Is there a truth assignment to V such that each clause is satisfied?

Since this is a restricted case of satisfiability, the hardness of 3-SAT implies that satisfiability is hard. The converse isn't true, since the hardness of general satisfiability might depend upon having long clauses. We can show the hardness

of 3-SAT using a reduction that translates every instance of satisfiability into an instance of 3-SAT without changing whether it is satisfiable.

This reduction transforms each clause independently based on its *length*, by adding new clauses and Boolean variables along the way. Suppose clause C_i contained k literals:

- $k = 1$, meaning that $C_i = \{z_1\}$ – We create two new variables v_1, v_2 and four new 3-literal clauses: $\{v_1, v_2, z_1\}$, $\{v_1, \overline{v}_2, z_1\}$, $\{\overline{v}_1, v_2, z_1\}$, $\{\overline{v}_1, \overline{v}_2, z_1\}$. Note that the only way that all four of these clauses can be simultaneously satisfied is if $z_1 = \text{true}$, which also means the original C_i will be satisfied.
- $k = 2$, meaning that $C_i = \{z_1, z_2\}$ – We create one new variable v_1 and two new clauses: $\{v_1, z_1, z_2\}$, $\{\overline{v}_1, z_1, z_2\}$. Again, the only way to satisfy both of these clauses is to have at least one of z_1 and z_2 be true, thus satisfying c_i.
- $k = 3$, meaning that $C_i = \{z_1, z_2, z_3\}$ – We copy C_i into the 3-SAT instance unchanged: $\{z_1, z_2, z_3\}$.
- $k > 3$, meaning that $C_i = \{z_1, z_2, ..., z_n\}$ – We create $n-3$ new variables and $n-2$ new clauses in a chain, where for $2 \leq j \leq n-3$, $C_{i,j} = \{v_{i,j-1}, z_{j+1}, \overline{v}_{i,j}\}$, $C_{i,1} = \{z_1, z_2, \overline{v}_{i,1}\}$, and $C_{i,n-2} = \{v_{i,n-3}, z_{n-1}, z_n\}$.

The most complicated case here is that of the large clauses. If none of the original literals in C_i are true, then there are not enough new variables to be able to satisfy all of the new subclauses. You can satisfy $C_{i,1}$ by setting $v_{i,1} = \text{false}$, but this forces $v_{i,2} = \text{false}$, and so on until finally $C_{i,n-2}$ cannot be satisfied. However, if any single literal $z_i = \text{true}$, then we have $n-3$ free variables and $n-3$ remaining 3-clauses, so we can satisfy each of them.

This transform takes $O(m+n)$ time if there were n clauses and m total literals in the SAT instance. Since any SAT solution also satisfies the 3-SAT instance and any 3-SAT solution describes how to set the variables giving a SAT solution, the transformed problem is equivalent to the original.

Note that a slight modification to this construction would serve to prove that 4-SAT, 5-SAT, or any $(k \geq 3)$-SAT is also NP-complete. However, this construction breaks down if we try to use it for 2-SAT, since there is no way to stuff anything into the chain of clauses. It turns out that a depth-first search on an appropriate graph can be used to give a linear-time algorithm for 2-SAT, as discussed in Section 14.10 (page 472).

9.5 Creative Reductions

Since both satisfiability and 3-SAT are known to be hard, we can use either of them in reductions. Usually 3-SAT is the better choice, because it is simpler to work with. What follows are a pair of more complicated reductions, designed to serve as examples and also increase our repertoire of known hard problems. Many

reductions are quite intricate, because we are essentially programming one problem in the language of a significantly different problem.

One perpetual point of confusion is getting the direction of the reduction right. Recall that we must transform *every* instance of a known NP-complete problem into an instance of the problem we are interested in. If we perform the reduction the other way, all we get is a slow way to solve the problem of interest, by using a subroutine that takes exponential time. This always is confusing at first, for this direction of reduction seems backwards. Make sure you understand the direction of reduction now, and think back to this whenever you get confused.

9.5.1 Integer Programming

As discussed in Section 13.6 (page 411), integer programming is a fundamental combinatorial optimization problem. It is best thought of as linear programming with the variables restricted to take only integer (instead of real) values.

Problem: Integer Programming
Input: A set of integer variables V, a set of inequalities over V, a maximization function $f(V)$, and an integer B.
Output: Does there exist an assignment of integers to V such that all inequalities are true and $f(V) \geq B$?

Consider the following two examples. Suppose

$$v_1 \geq 1, \quad v_2 \geq 0$$

$$v_1 + v_2 \leq 3$$

$$f(v) : 2v_2, \quad B = 3$$

A solution to this would be $v_1 = 1$, $v_2 = 2$. Not all problems have realizable solutions, however. For the following problem:

$$v_1 \geq 1, \quad v_2 \geq 0$$

$$v_1 + v_2 \leq 3$$

$$f(v) : 2v_2, \quad B = 5$$

The maximum possible value of $f(v)$ given the constraints is $2 \times 2 = 4$, so there can be no solution to the associated decision problem.

We show that integer programming is hard using a reduction from 3-SAT. For this particular reduction, general satisfiability would work just as well, although usually 3-SAT makes reductions easier.

In which direction must the reduction go? We want to prove integer programming is hard, and know that 3-SAT is hard. If I could solve 3-SAT using integer

programming and integer programming were easy, this would mean that satisfiability would be easy. Now the direction should be clear; we must translate 3-SAT into integer programming.

What should the translation look like? Every satisfiability instance contains Boolean (true/false) variables and clauses. Every integer programming instance contains integer variables (values restricted to 0,1,2,...) and constraints. A reasonable idea is to make the integer variables correspond to Boolean variables and use constraints to serve the same role as the clauses do in the original problem.

Our translated integer programming problem will have twice as many variables as the SAT instance—one for each variable and one for its complement. For each variable v_i in the set problem, we will add the following constraints:

- We restrict each integer programming variable V_i to values of either 0 or 1, but adding constraints $1 \geq V_i \geq 0$ and $1 \geq \overline{V}_i \geq 0$. Thus coupled with integrality, they correspond to values of true and false.

- We ensure that exactly one of the two integer programming variables associated with a given SAT variable is true, by adding constraints so that $1 \geq V_i + \overline{V}_i \geq 1$.

For each 3-SAT clause $C_i = \{z_1, z_2, z_3\}$, construct a constraint: $V_1 + V_2 + V_3 \geq 1$. To satisfy this constraint, at least one of the literals per clause must be set to 1, thus corresponding to a true literal. Satisfying this constraint is therefore equivalent to satisfying the clause.

The maximization function and bound prove relatively unimportant, since we have already encoded the entire 3-SAT instance. By using $f(v) = V_1$ and $B = 0$, we ensure that they will not interfere with any variable assignment satisfying all the inequalities. Clearly, this reduction can be done in polynomial time. To establish that this reduction preserves the answer, we must verify two things:

- *Any SAT solution gives a solution to the IP problem* – In any SAT solution, a true literal corresponds to a 1 in the integer program, since the clause is satisfied. Therefore, the sum in each clause inequality is ≥ 1.

- *Any IP solution gives a solution to the original SAT problem* – All variables must be set to either 0 or 1 in any solution to this integer programming instance. If $V_i = 1$, then set literal $z_i =$ true. If $V_i = 0$, then set literal $z_i =$ false. This is a legal assignment which must also satisfy all the clauses.

The reduction works both ways, so integer programming must be hard. Notice the following properties, which hold true in general for NP-completeness proofs:

1. This reduction preserved the structure of the problem. It did not *solve* the problem, just put it into a different format.

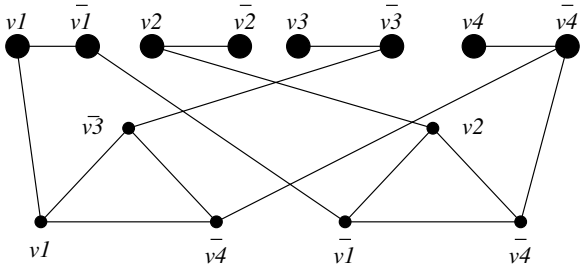


Figure 9.7: Reducing satisfiability instance $\{\{v_1, \bar{v_3}, \bar{v_4}\}, \{\bar{v_1}, v_2, \bar{v_4}\}\}$ to vertex cover

2. The possible IP instances that can result from this transformation are only a small subset of all possible IP instances. However, since some of them are hard, the general problem must be hard.

3. The transformation captures the essence of *why* IP is hard. It has nothing to do with having big coefficients or big ranges of variables, because restricting them to 0/1 is enough. It has nothing to do with having inequalities with large numbers of variables. Integer programming is hard because satisfying a set of constraints is hard. A careful study of the properties needed for a reduction can tell us a lot about the problem.

9.5.2 Vertex Cover

Algorithmic graph theory proves to be a fertile ground for hard problems. The prototypical NP-complete graph problem is vertex cover, previously defined in Section 9.3.2 (page 325) as follows:

Problem: Vertex Cover
Input: A graph $G = (V, E)$ and integer $k \leq |V|$.
Output: Is there a subset S of at most k vertices such that every $e \in E$ has at least one vertex in S?

Demonstrating the hardness of vertex cover proves more difficult than the previous reductions we have seen, because the structure of the two relevant problems is very different. A reduction from 3-satisfiability to vertex cover has to construct a graph G and bound k from the variables and clauses of the satisfiability instance.

First, we translate the variables of the 3-SAT problem. For each Boolean variable v_i, we create two vertices v_i and $\bar{v_i}$ connected by an edge. At least n vertices will be needed to cover all these edges, since no two of the edges will share a vertex.

Second, we translate the clauses of the 3-SAT problem. For each of the c clauses, we create three new vertices, one for each literal in each clause. The three vertices of each clause will be connected so as to form c triangles. At least two vertices per

triangle must be included in any vertex cover of these triangles, for a total of $2c$ cover vertices.

Finally, we will connect these two sets of components together. Each literal in the vertex "gadgets" is connected to vertices in the clause gadgets (triangles) that share the same literal. From a 3-SAT instance with n variables and c clauses, this constructs a graph with $2n + 3c$ vertices. The complete reduction for the 3-SAT problem $\{\{v_1, \bar{v_3}, \bar{v_4}\}, \{\bar{v_1}, v_2, \bar{v_4}\}\}$ is shown in Figure 9.7.

This graph has been designed to have a vertex cover of size $n + 2c$ if and only if the original expression is satisfiable. By the earlier analysis, every vertex cover must have at least $n + 2c$ vertices, since adding the connecting edges to G cannot shrink the size of the vertex cover to less than that of the disconnected vertex and clause gadgets. To show that our reduction is correct, we must demonstrate that:

- *Every satisfying truth assignment gives a vertex cover* – Given a satisfying truth assignment for the clauses, select the n vertices from the vertex gadgets that correspond to true literals to be members of the vertex cover. Since this defines a satisfying truth assignment, a true literal from each clause must cover at least one of the three cross edges connecting each triangle vertex to a vertex gadget. Therefore, by selecting the other two vertices of each clause triangle, we also pick up all remaining cross edges.

- *Every vertex cover gives a satisfying truth assignment* – In any vertex cover C of size $n + 2c$, exactly n of the vertices must belong to the vertex gadgets. Let these first stage vertices define the truth assignment, while the remaining $2c$ cover vertices must be distributed at two per clause gadget. Otherwise a clause gadget edge must go uncovered. These clause gadget vertices can cover only two of the three connecting cross edges per clause. Therefore, if C gives a vertex cover, at least one cross edge per clause must be covered, meaning that the corresponding truth assignment satisfies all clauses.

This proof of the hardness of vertex cover, chained with the clique and independent set reductions of Section 9.3.2 (page 325), gives us a library of hard graph problems that we can use to make future hardness proofs easier.

Take-Home Lesson: A small set of NP-complete problems (3-SAT, vertex cover, integer partition, and Hamiltonian cycle) suffice to prove the hardness of most other hard problems.

9.6 The Art of Proving Hardness

Proving that problems are hard is a skill. But once you get the hang of it, reductions can be surprisingly straightforward and pleasurable to do. Indeed, the dirty little secret of NP-completeness proofs is that they are usually easier to create than explain, in much the same way that it can be easier to rewrite old code than understand and modify it.

It takes experience to judge whether a problem is likely to be hard. Perhaps the quickest way to gain this experience is through careful study of the catalog. Slightly changing the wording of a problem can make the difference between it being polynomial or NP-complete. Finding the shortest path in a graph is easy, while finding the longest path in a graph is hard. Constructing a tour that visits all the edges once in a graph is easy (Eulerian cycle), while constructing a tour that visits all the vertices once is hard (Hamiltonian cycle).

The first thing to do when you suspect a problem might be NP-complete is look in Garey and Johnson's book *Computers and Intractability* [GJ79], which contains a list of several hundred problems known to be NP-complete. Likely you will find the problem you are interested in.

Otherwise I offer the following advice to those seeking to prove the hardness of a given problem:

- *Make your source problem as simple (i.e. , restricted) as possible.*

  Never try to use the general traveling salesman problem (TSP) as a source problem. Better, use Hamiltonian cycle (i.e. , TSP) where all the weights are 1 or ∞. Even better, use Hamiltonian path instead of cycle, so you don't have to worry about closing up the cycle. Best of all, use Hamiltonian path on directed planar graphs where each vertex has total degree 3. All of these problems are equally hard, but the more you can restrict the problem that you are reducing, the less work your reduction has to do.

  As another example, never try to use full satisfiability to prove hardness. Start with 3-satisfiability. In fact, you don't even have to use full 3-satisfiability. Instead, you can use *planar 3-satisfiability*, where there exists a way to draw the clauses as a graph in the plane such that you can connect all instances of the same literal together without edges crossing. This property tends to be useful in proving the hardness of geometric problems. All these problems are equally hard, and hence NP-completeness reductions using any of them are equally convincing.

- *Make your target problem as hard as possible.*

  Don't be afraid to add extra constraints or freedoms to make your target problem more general. Perhaps your undirected graph problem can be generalized into a directed graph problem and can hence be easier to prove hard. Once you have a proof of hardness for the general problem, you can then go back and try to simplify the target.

- *Select the right source problem for the right reason.*

  Selecting the right source problem makes a big difference in how difficult it is to prove a problem hard. This is the first and easiest place to go wrong, although theoretically any NP-complete problem works as well as any other. When trying to prove that a problem is hard, some people fish around through

lists of dozens of problems, looking for the best fit. These people are amateurs; odds are they never will recognize the problem they are looking for when they see it.

I use four (and only four) problems as candidates for my hard source problem. Limiting them to four means that I can know a lot about each of these problems, such as which variants of the problems are hard and which are not. My favorite source problems are:

- *3-SAT*: The old reliable. When none of the three problems below seem appropriate, I go back to the original source.
- *Integer partition*: This is the one and only choice for problems whose hardness seems to require using large numbers.
- *Vertex cover*: This is the answer for any graph problem whose hardness depends upon *selection*. Chromatic number, clique, and independent set all involve trying to select the correct subset of vertices or edges.
- *Hamiltonian path*: This is my choice for any graph problem whose hardness depends upon *ordering*. If you are trying to route or schedule something, Hamiltonian path is likely your lever into the problem.

- *Amplify the penalties for making the undesired selection.*

  Many people are too timid in their thinking about hardness proofs. You are trying to translate one problem into another, while keeping the problems as close to their original identities as as possible. The easiest way to do this is to be bold with your penalties; to punish anyone for trying to deviate from your intended solution. Your thinking should be, "if you select this element, then you must pick up this huge set that prevents you from finding an optimal solution." The sharper the consequences for doing what is undesired, the easier it is to prove the equivalence of the problems.

- *Think strategically at a high level, then build gadgets to enforce tactics.*

  You should be asking yourself the following types of questions: "How can I force that A or B is chosen but not both?" "How can I force that A is taken before B?" "How can I clean up the things I did not select?" Once you have an idea of what you want your gadgets to do, you can worry about how to actually craft them.

- *When you get stuck, alternate between looking for an algorithm or a reduction.*

  Sometimes the reason you cannot prove hardness is that there exists an efficient algorithm to solve your problem! Techniques such as dynamic programming or reducing problems to powerful but polynomial-time graph problems, such as matching or network flow, can yield surprising algorithms. Whenever you can't prove hardness, it pays to alter your opinion occasionally to keep yourself honest.

9.7 War Story: Hard Against the Clock

My class's attention span was running down like sand through an hourglass. Eyes were starting to glaze, even in the front row. Breathing had become soft and regular in the middle of the room. Heads were tilted back and eyes shut in the back.

There were twenty minutes left to go in my lecture on NP-completeness, and I couldn't really blame them. They had already seen several reductions like the ones presented here. But NP-completeness reductions are easier to create than to understand or explain. They had to watch one being created to appreciate how things worked.

I reached for my trusty copy of Garey and Johnson's book [GJ79], which contains a list of over four hundred different known NP-complete problems in an appendix in the back.

"Enough of this!" I announced loudly enough to startle those in the back row. "NP-completeness proofs are sufficiently routine that we can construct them on demand. I need a volunteer with a finger. Can anyone help me?"

A few students in the front held up their hands. A few students in the back held up their fingers. I opted for one from the front row.

"Select a problem at random from the back of this book. I can prove the hardness of any of these problems in the now seventeen minutes remaining in this class. Stick your finger in and read me a problem."

I had definitely gotten their attention. But I could have done that by offering to juggle chain-saws. Now I had to deliver results without cutting myself into ribbons.

The student picked out a problem. "OK, prove that *Inequivalence of Programs with Assignments* is hard," she said.

"Huh? I've never heard of that problem before. What is it? Read me the entire description of the problem so I can write it on the board." The problem was as follows:

Problem: Inequivalence of Programs with Assignments
Input: A finite set X of variables, two programs P_1 and P_2, each a sequence of assignments of the form

$$x_0 \leftarrow \text{ if } (x_1 = x_2) \text{ then } x_3 \text{ else } x_4$$

where the x_i are in X; and a value set V.
Output: Is there an initial assignment of a value from V to each variable in X such that the two programs yield different final values for some variable in X?

I looked at my watch. Fifteen minutes to go. But now everything was on the table. I was faced with a language problem. The input was two programs with variables, and I had to test to see whether they always do the same thing.

"First things first. We need to select a source problem for our reduction. Do we start with integer partition? 3-satisfiability? Vertex cover or Hamiltonian path?"

Since I had an audience, I tried thinking out loud. "Our target is not a graph problem or a numerical problem, so let's start thinking about the old reliable: 3-

satisfiability. There seem to be some similarities. 3-SAT has variables. This thing has variables. To be more like 3-SAT, we could try limiting the variables in this problem so they only take on two values—i.e. , $V = \{\text{true}, \text{false}\}$. Yes. That seems convenient."

My watch said fourteen minutes left. "So, class, which way does the reduction go? 3-SAT to language or language to 3-SAT?"

The front row correctly murmured, "3-SAT to language."

"Right. So we have to translate our set of clauses into two programs. How can we do that? We can try to split the clauses into two sets and write separate programs for each of them. But how do we split them? I don't see any natural way do it, because eliminating any single clause from the problem might suddenly make an unsatisfiable formula satisfiable, thus completely changing the answer. Instead, let's try something else. We can translate all the clauses into one program, and then make the second program be trivial. For example, the second program might ignore the input and always output either only true or only false. This sounds better. *Much* better."

I was still talking out loud to myself, which wasn't that unusual. But I had people listening to me, which was.

"Now, how can we turn a set of clauses into a program? We want to know whether the set of clauses can be satisfied, or if there is an assignment of the variables such that it is true. Suppose we constructed a program to evaluate whether $c_1 = (x_1, \overline{x}_2, x_3)$ is satisfied."

It took me a few minutes worth of scratching before I found the right program to simulate a clause. I assumed that I had access to constants for true and false:

$c_1 =$ if $(x_1 = \text{true})$ then true else false
$c_1 =$ if $(x_2 = \text{false})$ then true else c_1
$c_1 =$ if $(x_3 = \text{true})$ then true else c_1

"Great. Now I have a way to evaluate the truth of each clause. I can do the same thing to evaluate whether all the clauses are satisfied."

$sat =$ if $(c_1 = \text{true})$ then true else false
$sat =$ if $(c_2 = \text{true})$ then sat else false
$\vdots$
$sat =$ if $(c_n = \text{true})$ then sat else false

Now the back of the classroom was getting excited. They were starting to see a ray of hope that they would get to leave on time. There were two minutes left in class.

"Great. So now we have a program that can evaluate to be true if and only if there is a way to assign the variables to satisfy the set of clauses. We need a second program to finish the job. What about $sat = \text{false}$? Yes, that is all we need. Our language problem asks whether the two programs always output the same

thing, regardless of the possible variable assignments. If the clauses are satisfiable, that means that there must be an assignment of the variables such that the long program would output true. Testing whether the programs are equivalent is exactly the same as asking if the clauses are satisfiable."

I lifted my arms in triumph. "And so, the problem is neat, sweet, and NP-complete." I got the last word out just before the bell rang.

9.8 War Story: And Then I Failed

This exercise of picking a random NP-complete problem from the 400+ problems in Garey and Johnson's book and proving hardness on demand was so much fun that I have repeated it each time I have taught the algorithms course. Sure enough, I got it eight times in a row. But just as Joe DiMaggio's 56-game hitting streak came to an end, and Google will eventually have a losing quarter financially, the time came for me to get my comeuppance.

The class had voted to see a reduction from the graph theory section of the catalog, and a randomly selected student picked number 30. Problem GT30 turned out to be:

Problem: Uniconnected Subgraph
Input: Directed graph $G = (V, A)$, positive integer $k \leq |A|$.
Output: Is there a subset of arcs $A' \in A$ with $|A'| \geq k$ such that $G' = (V, A')$ has at most one directed path between any pair of vertices?

"It is a selection problem," I realized as soon as the problem was revealed. After all, we had to select the largest possible subset of arcs so that there were no pair of vertices with multiple paths between them. This meant that vertex cover was the problem of choice.

I worked through how the two problems stacked up. Both sought subsets, although vertex cover wanted subsets of vertices and uniconnected subgraph wanted subsets of edges. Vertex cover wanted the smallest possible subset, while unidirected subgraph wanted the largest possible subset. My source problem had undirected edges while my target had directed arcs, so somehow I would have to add edge direction into the reduction.

I had to do something to direct the edges of the vertex cover graph. I could try to replace each undirected edge (x, y) with a single arc, say from y to x. But quite different directed graphs would result depending upon which direction I selected. Finding the "right" orientation of edges might be a hard problem, certainly too hard to use in the translation phase of the reduction.

I realized I could direct the edges so the resulting graph was a DAG. But then, so what? DAGs certainly can have many different directed paths between pairs of vertices.

Alternately, I could try to replace each undirected edge (x, y) with *two* arcs, from x to y and y to x. Now there was no need to chose the right arcs for my

reduction, but the graph certainly got complicated. I couldn't see how to force things to prevent vertex pairs from having unwanted multiple paths between them.

Meanwhile, the clock was running and I knew it. A sense of panic set in during the last ten minutes of the class, and I realized I wasn't going to get it this time.

There is no feeling worse for a professor than botching up a lecture. You stand up there flailing away, knowing (1) that the students don't understand what you are saying, but (2) they do understand that you also don't understand what you are saying. The bell rang and the students left the room with faces either sympathetic or smirking.

I promised them a solution for the next class, but somehow I kept getting stuck in the same place each time I thought about it. I even tried to cheat and look up the proof in a journal. But the reference that Garey and Johnson cited was a 30-year old unpublished technical report. It wasn't on the web or in our library.

I dreaded the idea of returning to give my next class, the last lecture of the semester. But the night before class the answer came to me in a dream. *"Split the edges,"* the voice said. I awoke with a start and looked at the clock. It was 3:00 AM.

I sat up in bed and scratched out the proof. Suppose I replace each undirected edge (x, y) with a gadget consisting of a new central vertex v_{xy} with arcs going from it to x and y, respectively. This is nice. Now, which vertices are capable of having multiple paths between them? The new vertices had only outgoing edges, so they can only serve as the source of multiple paths. The old vertices had only incoming edges. There was at most one way to get from one of the new source vertices to any of the original vertices of the vertex cover graph, so these could not result in multiple paths.

But now add a sink node s with edges from all original vertices. There were exactly two paths from each new source vertex to this sink—one through each of the two original vertices it was adjacent to. One of these had to be broken to create a uniconnected subgraph. How could we break it? We could pick one of these two vertices to disconnect from the sink by deleting either arc (x, s) or (y, s) for new vertex v_{xy}. We maximize the size of our subgraph by finding the smallest number of arcs to delete. We must delete the outgoing arc from at least one of the two vertices defining each original edge. *This is exactly the same as finding the vertex cover in this graph!* The reduction is illustrated in Figure 9.8.

Presenting this proof in class provided some personal vindication, but more to the point validates the principles I teach for proving hardness. Observe that the reduction really wasn't all that difficult after all: just split the edges and add a sink node. NP-completeness reductions are often surprisingly simple once you look at them the right way.

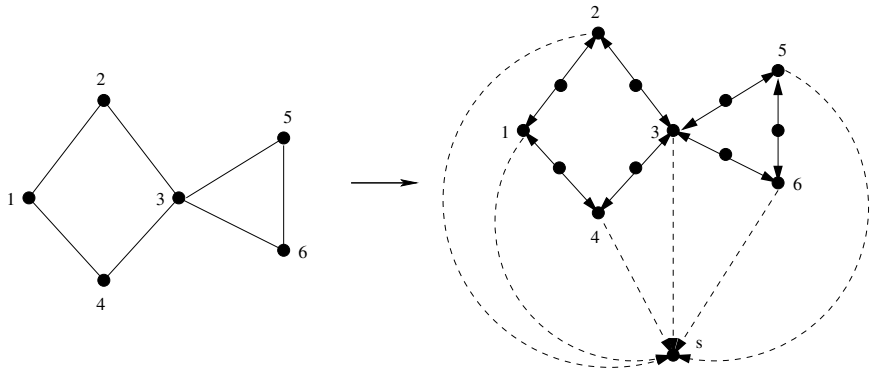


Figure 9.8: Reducing vertex cover to unidirected subgraph, by dividing edges and adding a sink node

9.9 P vs. NP

The theory of NP-completeness rests on a foundation of rigorous but subtle definitions from automata and formal language theory. This terminology is typically confusing to or misused by beginners who lack a mastery of these foundations. It is not really essential to the practical aspects of designing and applying reductions. That said, the question "Is P=NP?" is the most profound open problem in computer science, so any educated algorist should have some idea what the stakes are.

9.9.1 Verification vs. Discovery

The primary question in P vs NP is whether *verification* is really an easier task than initial *discovery*. Suppose that while taking an exam you "happen" to notice the answer of the student next to you. Are you now better off? You wouldn't dare to turn it in without checking, since an able student such as yourself could answer the question correctly if you took enough time to solve it. The issue is whether you can really verify the answer faster than you could find it from scratch.

For the NP-complete decision problems we have studied here, the answer *seems* obvious:

- Can you verify that a graph has a TSP tour of at most k weight given the order of vertices on the tour? Sure. Just add up the weights of the edges on the tour and show it is at most k. That is easier than finding the tour, *isn't it*?

- Can you verify that a given truth assignment represents a solution to a given satisfiability problem? Sure. Just check each clause and make sure it contains

at least one true literal from the given truth assignment. That is easier than finding the satisfying assignment, *isn't it*?

- Can you verify that a graph G has a vertex cover of at most k vertices if given the subset S of at most k vertices forming such a cover? Sure. Just traverse each edge (u, v) of G, and check that either u or v is in S. That is easier than finding the vertex cover, *isn't it*?

At first glance, this seems obvious. The given solutions can be verified in linear time for all three of these problems, while no algorithm faster than brute force search is known to find the solutions for any of them. The catch is that we have no rigorous lower bound *proof* that prevents the existence of fast algorithms to solve these problems. Perhaps there are in fact polynomial algorithms (say $O(n^7)$) that we have just been too blind to see yet.

9.9.2 The Classes P and NP

Every well-defined algorithmic problem must have an asymptotically fastest-possible algorithm solving it, as measured in the Big-Oh, worst-case sense of fastest.

We can think of the class P as an exclusive club for algorithmic problems that a problem can only join after demonstrating that there exists a polynomial-time algorithm to solve it. Shortest path, minimum spanning tree, and the original movie scheduling problem are all members in good standing of this class P. The P stands for *polynomial-time*.

A less-exclusive club welcomes all the algorithmic problems whose solutions can be *verified* in polynomial-time. As shown above, this club contains traveling salesman, satisfiability, and vertex cover, none of which currently have the credentials to join P. However, all the members of P get a free pass into this less exclusive club. If you can solve the problem from scratch in polynomial time, you certainly can verify a solution that fast: just solve it from scratch and see if the solution you get is as good as the one you were given.

We call this less-exclusive club NP. You can think of this as standing for *not-necessarily polynomial-time*.[1]

The $1,000,000 question is whether there are in fact problems in NP that cannot be members of P. If no such problem exists, the classes must be the same and $P = NP$. If even one such a problem exists, the two classes are different and $P \neq NP$. The opinion of most algorists and complexity theorists is that the classes differ, meaning $P \neq NP$, but a much stronger proof than "I can't find a fast enough algorithm" is needed.

[1] In fact, it stands for *nondeterministic polynomial-time*. This is in the sense of nondeterministic automata, if you happen to know about such things.

9.9.3 Why is Satisfiability the Mother of All Hard Problems?

An enormous tree of NP-completeness reductions has been established that entirely rests on the hardness of satisfiability. The portion of this tree demonstrated and/or stated in this chapter (and proven elsewhere) is shown in Figure 9.2.

This may look like a delicate affair. What would it mean if someone *does* find a polynomial-time algorithm for satisfiability? A fast algorithms for any given NP-complete problem (say traveling salesman) implies fast algorithm for all the problems on the path in the reduction tree between TSP and satisfiability (Hamiltonian cycle, vertex cover, and 3-SAT). But a fast algorithm for satisfiability doesn't immediately yield us anything because the reduction path from SAT to SAT is empty.

Fear not. There exists an extraordinary super-reduction (called Cook's theorem) reducing *all* the problems in NP to satisfiability. Thus, if you prove that satisfiability (or equivalently any single NP-complete problem) is in P, then *all* other problems in NP follow and $P = NP$. Since essentially every problem mentioned in this book is in NP, this would be an enormously powerful and surprising result.

Cook's theorem proves that satisfiability is as hard as any problem in NP. Furthermore, it proves that every NP-complete problem is as hard as any other. Any domino falling (i.e. , a polynomial-time algorithm for any NP-complete problem) knocks them all down. Our inability to find a fast algorithm for any of these problems is a strong reason for believing that they are all truly hard, and probably $P \neq NP$.

9.9.4 NP-hard vs. NP-complete?

The final technicality we will discuss is the difference between a problem being NP-hard and NP-complete. I tend to be somewhat loose with my terminology, but there is a subtle (usually irrelevant) distinction between the two concepts.

We say that a problem is *NP-hard* if, like satisfiability, it is at least as hard as any problem in NP. We say that a problem is *NP-complete* if it is NP-hard, and also in NP itself. Because NP is such a large class of problems, most NP-hard problems you encounter will actually be complete, and the issue can always be settled by giving a (usually simple) verification strategy for the problem. All the NP-hard problems we have encountered in this book are also NP-complete.

That said, there are problems that appear to be NP-hard yet not in NP. These problems might be *even harder* than NP-complete! Two-player games such as chess provide examples of problems that are not in NP. Imagine sitting down to play chess with some know-it-all, who is playing white. He pushes his central pawn up two squares to start the game, and announces *checkmate*. The only obvious way to show he is right would be to construct the full tree of all your possible moves with his irrefutable replies and demonstrate that you, in fact, cannot win from the current position. This full tree will have a number of nodes exponential in its height, which is the number of moves before you lose playing your most spirited possible defense.

Clearly this tree cannot be constructed and analyzed in polynomial time, so the problem is not in NP.

9.10 Dealing with NP-complete Problems

For the practical person, demonstrating that a problem is NP-complete is never the end of the line. Presumably, there was a reason why you wanted to solve it in the first place. That application will not go away when told that there is no polynomial-time algorithm. You still seek a program that solves the problem of interest. All you know is that you won't find one that quickly solves the problem to optimality in the worst case. You still have three options:

- *Algorithms fast in the average case* – Examples of such algorithms include backtracking algorithms with substantial pruning.

- *Heuristics* – Heuristic methods like simulated annealing or greedy approaches can be used to quickly find a solution with no guarantee that it will be the best one.

- *Approximation algorithms* – The theory of NP-completeness only stipulates that it is hard to get close to the answer. With clever, problem-specific heuristics, we can probably get *close* to the optimal answer on all possible instances.

Approximation algorithms return solutions with a guarantee attached, namely that the optimal solution can never be much better than this given solution. Thus you can never go too far wrong when using an approximation algorithm. No matter what your input instance is and how lucky you are, you are doomed to do all right. Furthermore, approximation algorithms realizing probably good bounds are often conceptually simple, very fast, and easy to program.

One thing that is usually not clear, however, is how well the solution from an approximation algorithm compares to what you might get from a heuristic that gives you no guarantees. The answer could be worse or it could be better. Leaving your money in a bank savings account guarantees you 3% interest without risk. Still, you likely will do much better investing your money in stocks than leaving it in the bank, even though performance is not guaranteed.

One way to get the best of approximation algorithms and heuristics is to run both of them on the given problem instance and pick the solution giving the better result. This way, you get a solution that comes with a guarantee and a second chance to do even better. When it comes to heuristics for hard problems, sometimes you can have it both ways.

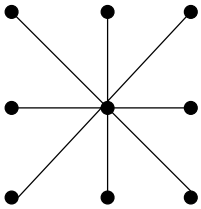


Figure 9.9: Neglecting to pick the center vertex leads to a terrible vertex cover

9.10.1 Approximating Vertex Cover

As we have seen before, finding the minimum vertex cover of a graph is NP-complete. However, a very simple procedure can efficiently find a cover that is at most twice as large as the optimal cover:

VertexCover($G = (V, E)$)
    While ($E \neq \emptyset$) do:
        Select an arbitrary edge $(u, v) \leq E$
        Add both u and v to the vertex cover
        Delete all edges from E that are incident to either u or v.

It should be apparent that this procedure always produces a vertex cover, since each edge is only deleted after an incident vertex has been added to the cover. More interesting is the claim that any vertex cover must use at least half as many vertices as this one. Why? Consider only the $\leq n/2$ edges selected by the algorithm that constitute a matching in the graph. No two of these edges can share a vertex. Therefore, any cover of just these edges must include at least one vertex per edge, which makes it at least half the size of this greedy cover.

There are several interesting things to notice about this algorithm:

- *Although the procedure is simple, it is not stupid* – Many seemingly smarter heuristics can give a far worse performance in the worst case. For example, why not modify the above procedure to select only one of the two vertices for the cover instead of both. After all, the selected edge will be equally well covered by only one vertex. However, consider the star-shaped graph of Figure 9.9. This heuristic will produce a two-vertex cover, while the single-vertex heuristic can return a cover as large as $n - 1$ vertices, should we get unlucky and repeatedly select the leaf instead of the center as the cover vertex we retain.

- *Greedy isn't always the answer* – Perhaps the most natural heuristic for vertex cover would repeatedly select and delete the vertex of highest remaining degree for the vertex cover. After all, this vertex will cover the largest number

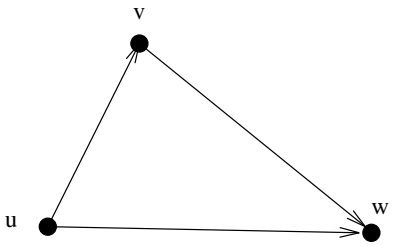


Figure 9.10: The triangle inequality ($d(u,w) \leq d(u,v) + d(v,w)$) typically holds in geometric and weighted graph problems

of possible edges. However, in the case of ties or near ties, this heuristic can go seriously astray. In the worst case, it can yield a cover that is $\Theta(\lg n)$ times optimal.

- *Making a heuristic more complicated does not necessarily make it better* – It is easy to complicate heuristics by adding more special cases or details. For example, the procedure above does not specify which edge should be selected next. It might seem reasonable to next select the edge whose endpoints have the highest degree. However, this does not improve the worst-case bound and just makes it more difficult to analyze.

- *A postprocessing cleanup step can't hurt* – The flip side of designing simple heuristics is that they can often be modified to yield better-in-practice solutions without weakening the approximation bound. For example, a postprocessing step that deletes any unnecessary vertex from the cover can only improve things in practice, even though it won't help the worst-case bound.

The important property of approximation algorithms is relating the size of the solution produced directly to a lower bound on the optimal solution. Instead of thinking about how well we might do, we must think about the worst case—i.e. , how badly we might perform.

9.10.2 The Euclidean Traveling Salesman

In most natural applications of the traveling salesman problem, direct routes are inherently shorter than indirect routes. For example, if a graph's edge weights were the straight-line distances between pairs of cities, the shortest path from x to y will always be "as the crow flies."

The edge weights induced by Euclidean geometry satisfy the triangle inequality, namely that $d(u,w) \leq d(u,v) + d(v,w)$ for all triples of vertices u, v, and w. The general reasonableness of this condition is demonstrated in Figure 9.10. The cost of airfare is an example of a distance function that *violates* the triangle inequality,

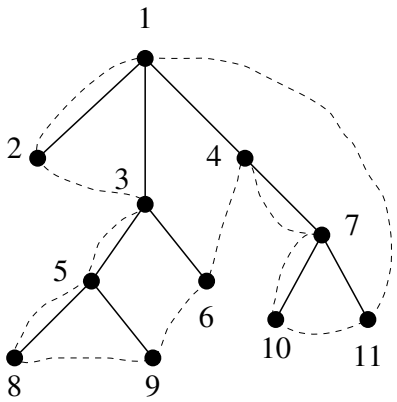


Figure 9.11: A depth-first traversal of a spanning tree, with the shortcut tour

since it is sometimes cheaper to fly through an intermediate city than to fly to the destination directly. TSP remains hard when the distances are Euclidean distances in the plane.

We can approximate the optimal traveling salesman tour using minimum spanning trees or graphs that obey the triangle inequality. First, observe that the weight of a minimum spanning tree is a lower bound on the cost of the optimal tour. Why? Deleting any edge from a tour leaves a path, the total weight of which must be no greater than that of the original tour. This path has no cycles, and hence is a tree, which means its weight is at least that of the minimum spanning tree. Thus the weight of the minimum spanning tree gives a lower bound on the optimal tour.

Consider now what happens in performing a depth-first traversal of a spanning tree. We will visit each edge twice, once going down the tree when discovering the edge and once going up after exploring the entire subtree. For example, in the depth-first search of Figure 9.11, we visit the vertices in order $1-2-1-3-5-8-5-9-5-3-6-3-1-4-7-10-7-11-7-4-1$, thus using every tree edge exactly twice. This tour repeats each edge of the minimum spanning tree twice, and hence costs at most twice the optimal tour.

However, vertices will be repeated on this depth-first search tour. To remove the extra vertices, we can take a shortest path to the next unvisited vertex at each step. The shortcut tour for the tree above is $1-2-3-5-8-9-6-4-7-10-11-1$. Because we have replaced a chain of edges by a single direct edge, the triangle inequality ensures that the tour can only get shorter. Thus, this shortcut tour is also within weight and twice that of optimal. Better, more complicated approximation algorithms for Euclidean TSP exist, as described in Section 16.4 (page 533). No approximation algorithms are known for TSPs that do not satisfy the triangle inequality.

9.10.3 Maximum Acyclic Subgraph

Directed acyclic graphs (DAGs) are easier to work with than general digraphs. Sometimes it is useful to simplify a given graph by deleting a small set of edges or vertices that suffice to break all cycles. Such *feedback set* problems are discussed in Section 16.11 (page 559).

Here we consider an interesting problem in this class where seek to retain as many edges as possible while breaking all directed cycles:

Problem: Maximum Directed Acyclic Subgraph
Input: A directed graph $G = (V, E)$.
Output: Find the largest possible subset $E' \in E$ such that $G' = (V, E')$ is acyclic.

In fact, there is a very simple algorithm that guarantees you a solution with at least half as many edges as optimum. I encourage you to try to find it now before peeking.

Construct *any* permutation of the vertices, and interpret it as a left-right ordering akin to topological sorting. Now some of the edges will point from left to right, while the remainder point from right to left.

One of these two edge subsets must be at least as large as the other. This means it contains at least half the edges. Furthermore each of these two edge subsets must be acyclic for the same reason only DAGs can be topologically sorted—you cannot form a cycle by repeatedly moving in one direction. Thus, the larger edge subset must be acyclic and contain at least half the edges of the optimal solution!

This approximation algorithm *is* simple to the point of almost being stupid. But note that heuristics can make it perform better in practice without losing this guarantee. Perhaps we can try many random permutations, and pick the best. Or we can try to exchange pairs of vertices in the permutations retaining those swaps, which throw more edges onto the bigger side.

9.10.4 Set Cover

The previous sections may encourage a false belief that every problem can be approximated to within a factor of two. Indeed, several catalog problems such as maximum clique cannot be approximated to *any* interesting factor.

Set cover occupies a middle ground between these extremes, having a factor-$\Theta(\lg n)$ approximation algorithm. Set cover is a more general version of the vertex cover problem. As defined in Section 18.1 (page 621),

Problem: Set Cover
Input: A collection of subsets $S = \{S_1, \ldots, S_m\}$ of the universal set $U = \{1, \ldots, n\}$.
Output: What is the smallest subset T of S whose union equals the universal set—i.e. , $\cup_{i=1}^{|T|} T_i = U$?

milestone class	6	5		4					3			2	1	0
uncovered elements	64	51	40	30	25	22	19	16	13	10	7	4	2	1
selected subset size	13	11	10	5	3	3	3	3	3	3	3	2	1	1

Figure 9.12: The coverage process for greedy on a particular instance of set cover

The natural heuristic is greedy. Repeatedly select the subset that covers the largest collection of thus-far uncovered elements until everything is covered. In pseudocode,

```
SetCover(S)
    While (U ≠ ∅) do:
        Identify the subset S_i with the largest intersection with U
        Select S_i for the set cover
        U = U − S_i
```

One consequence of this selection process is that the number of freshly-covered elements defines a nonincreasing sequence as the algorithm proceeds. Why? If not, greedy would have picked the more powerful subset earlier if it, in fact, existed.

Thus we can view this heuristic as reducing the number of uncovered elements from n down to zero by progressively smaller amounts. A trace of such an execution is shown in Figure 9.12.

An important milestone in such a trace occurs each time the number of remaining uncovered elements reduces past a power of two. Clearly there can be at most $\lceil \lg n \rceil$ such events.

Let w_i denote the number of subsets that were selected by the heuristic to cover elements between milestones $2^{i+1} - 1$ and 2^i. Define the width w to be the maximum w_i, where $0 \leq i \leq \lg_2 n$. In the example of Figure 9.12, the maximum width is given by the five subsets needed to go from $2^5 - 1$ down to 2^4.

Since there are at most $\lg n$ such milestones, the solution produced by the greedy heuristic must contain at most $w \cdot \lg n$ subsets. But I claim that the optimal solution must contain *at least* w subsets, so the heuristic solution is no worse than $\lg n$ times optimal.

Why? Consider the average number of new elements covered as we move between milestones $2^{i+1} - 1$ and 2^i. These 2^i elements require w_i subsets, so the average coverage is $\mu_i = 2^i / w_i$. More to the point, the last/smallest of these subsets covers at most μ_i subsets. Thus, *no subset exists in S that can cover more than μ_i of the remaining 2^i elements.* So, to finish the job, we need at least $2^i / \mu_i = w_i$ subsets.

The somewhat surprising thing is that there do exist set cover instances where this heuristic takes $\Omega(\lg n)$ times optimal. The logarithmic factor is a property of the problem/heuristic, not an artifact of weak analysis.

Take-Home Lesson: Approximation algorithms guarantee answers that are always close to the optimal solution. They can provide a practical approach to dealing with NP-complete problems.

Chapter Notes

The notion of NP-completeness was first developed by Cook [Coo71]. Satisfiability really is a $1,000,000 problem, and the Clay Mathematical Institute has offered such a prize to any person who resolves the P=NP question. See *http://www.claymath.org/* for more on the problem and the prize.

Karp [Kar72] showed the importance of Cook's result by providing reductions from satisfiability to more than 20 important algorithmic problems. I recommend Karp's paper for its sheer beauty and economy—he reduces each reduction to three line descriptions showing the problem equivalence. Together, these provided the tools to resolve the complexity of literally hundreds of important problems where no efficient algorithms were known.

The best introduction to the theory of NP-completeness remains Garey and Johnson's book *Computers and Intractability*. It introduces the general theory, including an accessible proof of Cook's theorem [Coo71] that satisfiability is as hard as anything in NP. They also provide an essential reference catalog of more than 300 NP-complete problems, which is a great resource for learning what is known about the most interesting hard problems. The reductions claimed, but missing from this chapter can be found in Garey and Johnson, or textbooks such as [CLRS01].

A few catalog problems exist in a limbo state where it is not known whether the problem has a fast algorithm or is NP-complete. The most prominent of these are graph isomorphism (see Section 16.9 (page 550)) and integer factorization (see Section 13.8 (page 420)). That this limbo list is so short is quite a tribute to the state of the art in algorithm design and the power of NP-completeness. For almost every important problem we either have a fast algorithm or a good solid reason for why one doesn't exist.

The war story problem on unidirected subgraph was originally proven hard in [Mah76].

9.11 Exercises

Transformations and Satisfiability

9-1. *[2]* Give the 3-SAT formula that results from applying the reduction of SAT to 3-SAT for the formula:

$$(x + y + \overline{z} + w + u + \overline{v}) \cdot (\overline{x} + \overline{y} + z + \overline{w} + u + v) \cdot (x + \overline{y} + \overline{z} + w + u + \overline{v}) \cdot (x + \overline{y})$$

9-2. *[3]* Draw the graph that results from the reduction of 3-SAT to vertex cover for the expression

$$(x + \overline{y} + z) \cdot (\overline{x} + y + \overline{z}) \cdot (\overline{x} + y + z) \cdot (x + \overline{y} + \overline{x})$$

9-3. *[4]* Suppose we are given a subroutine which can solve the traveling salesman decision problem of page 318 in, say, linear time. Give an efficient algorithm to find the actual TSP tour by making a polynomial number of calls to this subroutine.

9-4. *[7]* Implement a translator that translates satisfiability instances into equivalent 3-SAT instances.

9-5. *[7]* Design and implement a backtracking algorithm to test whether a set of formulae are satisfiable. What criteria can you use to prune this search?

9-6. *[8]* Implement the vertex cover to satisfiability reduction, and run the resulting clauses through a satisfiability testing code. Does this seem like a practical way to compute things?

Basic Reductions

9-7. *[4]* An instance of the *set cover* problem consists of a set X of n elements, a family F of subsets of X, and an integer k. The question is, does there exist k subsets from F whose union is X?

For example, if $X = \{1, 2, 3, 4\}$ and $F = \{\{1, 2\}, \{2, 3\}, \{4\}, \{2, 4\}\}$, there does not exist a solution for $k = 2$, but there does for $k = 3$ (for example, $\{1, 2\}, \{2, 3\}, \{4\}$).

Prove that set cover is NP-complete with a reduction from vertex cover.

9-8. *[4]* The *baseball card collector problem* is as follows. Given packets $P_1, \ldots, P_m$, each of which contains a subset of this year's baseball cards, is it possible to collect all the year's cards by buying $\leq k$ packets?

For example, if the players are $\{Aaron, Mays, Ruth, Skiena\}$ and the packets are

$$\{\{Aaron, Mays\}, \{Mays, Ruth\}, \{Skiena\}, \{Mays, Skiena\}\},$$

there does not exist a solution for $k = 2$, but there does for $k = 3$, such as

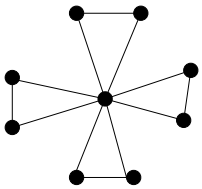
$$\{Aaron, Mays\}, \{Mays, Ruth\}, \{Skiena\}$$

Prove that the baseball card collector problem is NP-hard using a reduction from vertex cover.

9-9. *[4]* The *low-degree spanning tree problem* is as follows. Given a graph G and an integer k, does G contain a spanning tree such that all vertices in the tree have degree *at most* k (obviously, only tree edges count towards the degree)? For example, in the following graph, there is no spanning tree such that all vertices have a degree less than three.

(a) Prove that the low-degree spanning tree problem is NP-hard with a reduction from Hamiltonian *path*.

(b) Now consider the *high-degree spanning tree problem*, which is as follows. Given a graph G and an integer k, does G contain a spanning tree whose highest degree vertex is *at least* k? In the previous example, there exists a spanning tree with a highest degree of 8. Give an efficient algorithm to solve the high-degree spanning tree problem, and an analysis of its time complexity.



9-10. *[4]* Show that the following problem is NP-complete:

Problem: Dense subgraph
Input: A graph G, and integers k and y.
Output: Does G contain a subgraph with exactly k vertices and at least y edges?

9-11. *[4]* Show that the following problem is NP-complete:

Problem: Clique, no-clique
Input: An undirected graph $G = (V, E)$ and an integer k.
Output: Does G contain both a clique of size k and an independent set of size k.

9-12. *[5]* An *Eulerian cycle* is a tour that visits every edge in a graph exactly once. An *Eulerian subgraph* is a subset of the edges and vertices of a graph that has an Eulerian cycle. Prove that the problem of finding the number of edges in the largest Eulerian subgraph of a graph is NP-hard. (Hint: the Hamiltonian circuit problem is NP-hard even if each vertex in the graph is incident upon exactly three edges.)

Creative Reductions

9-13. *[5]* Prove that the following problem is NP-complete:

Problem: Hitting Set
Input: A collection C of subsets of a set S, positive integer k.
Output: Does S contain a subset S' such that $|S'| \leq k$ and each subset in C contains at least one element from S'?

9-14. *[5]* Prove that the following problem is NP-complete:

Problem: Knapsack
Input: A set S of n items, such that the ith item has value v_i and weight w_i. Two positive integers: weight limit W and value requirement V.
Output: Does there exist a subset $S' \in S$ such that $\sum_{i \in S'} w_i \leq W$ and $\sum_{i \in S'} v_i \geq V$? (Hint: start from integer partition.)

9-15. *[5]* Prove that the following problem is NP-complete:

Problem: Hamiltonian Path
Input: A graph G, and vertices s and t.

Output: Does G contain a path which starts from s, ends at t, and visits all vertices without visiting any vertex more than once? (Hint: start from Hamiltonian cycle.)

9-16. *[5]* Prove that the following problem is NP-complete:

Problem: Longest Path
Input: A graph G and positive integer k.
Output: Does G contain a path that visits at least k different vertices without visiting any vertex more than once?

9-17. *[6]* Prove that the following problem is NP-complete:

Problem: Dominating Set
Input: A graph $G = (V, E)$ and positive integer k.
Output: Is there a subset $V' \in V$ such that $|V'| \leq k$ where for each vertex $x \in V$ either $x \in V'$ or there exists an edge (x, y), where $y \in V'$.

9-18. *[7]* Prove that the vertex cover problem (does there exist a subset S of k vertices in a graph G such that every edge in G is incident upon at least one vertex in S?) remains NP-complete even when all the vertices in the graph are restricted to have even degrees.

9-19. *[7]* Prove that the following problem is NP-complete:

Problem: Set Packing
Input: A collection C of subsets of a set S, positive integer k.
Output: Does S contain at least k disjoint subsets (i.e. , such that none of these subsets have any elements in common?)

9-20. *[7]* Prove that the following problem is NP-complete:

Problem: Feedback Vertex Set
Input: A directed graph $G = (V, A)$ and positive integer k.
Output: Is there a subset $V' \in V$ such that $|V'| \leq k$, such that deleting the vertices of V' from G leaves a DAG?

Algorithms for Special Cases

9-21. *[5]* A Hamiltonian path P is a path that visits each vertex exactly once. The problem of testing whether a graph G contains a Hamiltonian path is NP-complete. There does not have to be an edge in G from the ending vertex to the starting vertex of P, unlike in the Hamiltonian cycle problem.

Give an $O(n + m)$-time algorithm to test whether a directed acyclic graph G (a DAG) contains a Hamiltonian path. (Hint: think about topological sorting and DFS.)

9-22. *[8]* The 2-SAT problem is, given a Boolean formula in 2-conjunctive normal form (CNF), to decide whether the formula is satisfiable. 2-SAT is like 3-SAT, except

that each clause can have only two literals. For example, the following formula is in 2-CNF:

$(x_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_3)$

Give a polynomial-time algorithm to solve 2-SAT.

P=NP?

9-23. *[4]* Show that the following problems are in NP:

- Does graph G have a simple path (i.e. , with no vertex repeated) of length k?
- Is integer n composite (i.e. , not prime)?
- Does graph G have a vertex cover of size k?

9-24. *[7]* It was a long open question whether the decision problem "Is integer n a composite number, in other words, not prime?" can be computed in time polynomial in the size of its input. Why doesn't the following algorithm suffice to prove it is in P, since it runs in $O(n)$ time?

```
PrimalityTesting(n)
    composite := false
    for i := 2 to n − 1 do
        if (n mod i) = 0 then
            composite := true
```

Approximation Algorithms

9-25. *[4]* In the *maximum-satisfiability problem*, we seek a truth assignment that satisfies as many clauses as possible. Give an heuristic that always satisfies at least half as many clauses as the optimal solution.

9-26. *[5]* Consider the following heuristic for vertex cover. Construct a DFS tree of the graph, and delete all the leaves from this tree. What remains must be a vertex cover of the graph. Prove that the size of this cover is at most twice as large as optimal.

9-27. *[5]* The *maximum cut* problem for a graph $G = (V, E)$ seeks to partition the vertices V into disjoint sets A and B so as to maximize the number of edges $(a, b) \in E$ such that $a \in A$ and $b \in B$. Consider the following heuristic for max cut. First assign v_1 to A and v_2 to B. For each remaining vertex, assign it to the side that adds the most edges to the cut. Prove that this cut is at least half as large as the optimal cut.

9-28. *[5]* In the *bin-packing problem*, we are given n items with weights $w_1, w_2, ..., w_n$, respectively. Our goal is to find the smallest number of bins that will hold the n objects, where each bin has capacity of at most one kilogram.

The *first-fit heuristic* considers the objects in the order in which they are given. For each object, place it into first bin that has room for it. If no such bin exists, start a new bin. Prove that this heuristic uses at most twice as many bins as the optimal solution.

9-29. *[5]* For the first-fit heuristic described just above, give an example where the packing it fits uses at least 5/3 times as many bins as optimal.

9-30. *[5]* A *vertex coloring* of graph $G = (V, E)$ is an assignment of colors to vertices of V such that each edge (x, y) implies that vertices x and y are assigned different colors. Give an algorithm for vertex coloring G using at most $\Delta + 1$ colors, where Δ is the maximum vertex degree of G.

Programming Challenges

These programming challenge problems with robot judging are available at *http://www.programming-challenges.com* or *http://online-judge.uva.es*.

Geometry

9-1. "The Monocycle" – Programming Challenges 111202, UVA Judge 10047.

9-2. "Dog and Gopher" – Programming Challenges 10310, UVA Judge 111301.

9-3. "Chocolate Chip Cookies" – Programming Challenges 111304, UVA Judge 10136.

9-4. "Birthday Cake" – Programming Challenges 111305, UVA Judge 10167.

Computational Geometry

9-5. "Closest Pair Problem" – Programming Challenges 111402, UVA Judge 10245.

9-6. "Chainsaw Massacre" – Programming Challenges 111403, UVA Judge 10043.

9-7. "Hotter Colder" – Programming Challenges 111404, UVA Judge 10084.

9-8. "Useless Tile Packers" – Programming Challenges 111405, UVA Judge 10065.

Note: These are not particularly relevant to NP-completeness, but are added for completeness.