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)
    $c_{ins} = c_{del} = 1$
    $c_{sub} = \infty$
    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

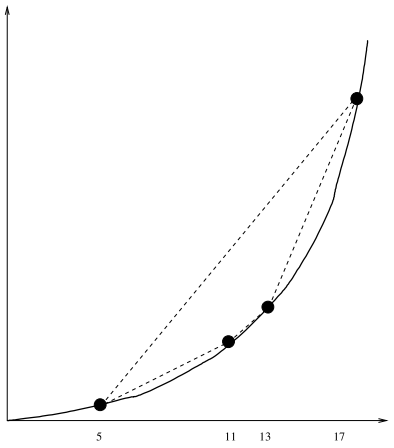

Figure 9.1: Reducing convex hull to sorting by mapping points to a parabola

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:

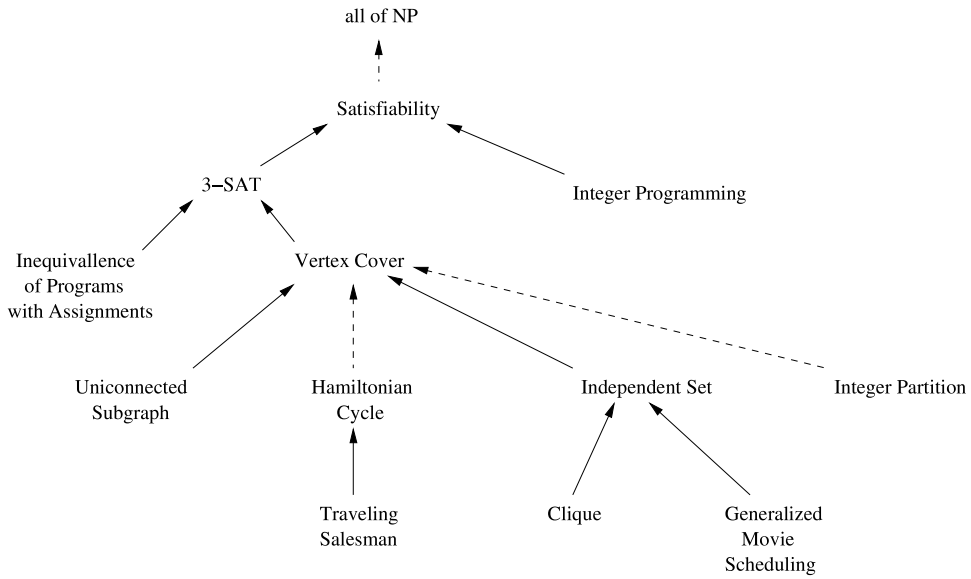

Figure 9.2: A portion of the reduction tree for NP-complete problems. Solid lines denote the reductions presented in this chapter

---

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*.

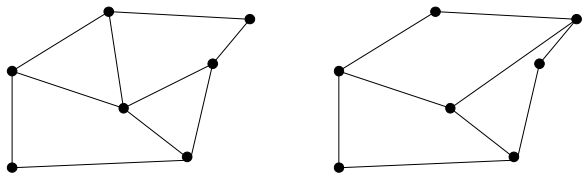

Figure 9.3: Graphs with (l) and without (r) Hamiltonian cycles

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

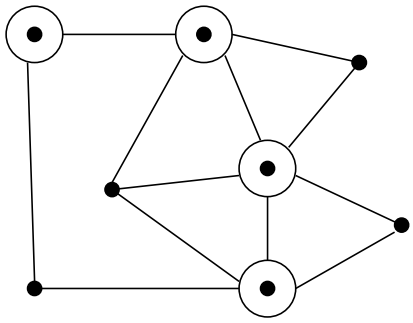

Figure 9.4: Circled vertices form a vertex cover, and the others form an independent set

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

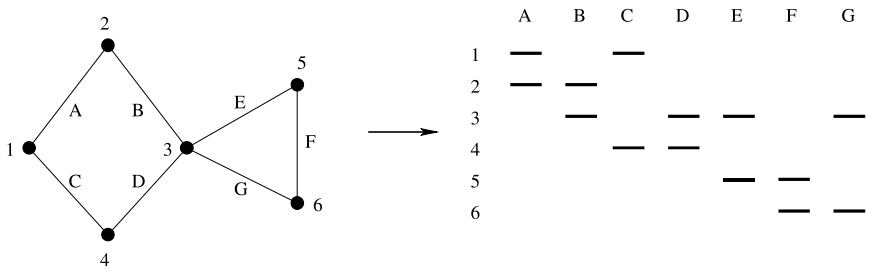

Figure 9.5: Reduction from independent set to generalized movie scheduling, with numbered vertices and lettered edges

---

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?

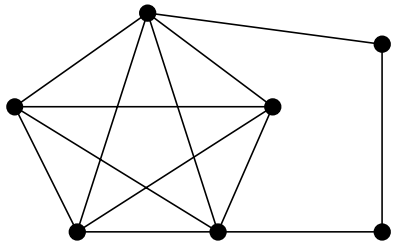

Figure 9.6: A small graph with a five-vertex clique

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 $z_1 =$ 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 $v_{i,1} =$ false, but this forces $v_{i,2} =$ false, and so on until finally $C_{i,n-2}$ cannot be satisfied. However, if any single literal $z_i =$ 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