Chapter 9

Coping with NP-completeness

You are the junior member of a seasoned project team. Your current task is to write code for solving a simple-looking problem involving graphs and numbers. What are you supposed to do?

If you are very lucky, your problem will be among the half-dozen problems concerning graphs with weights (shortest path, minimum spanning tree, maximum flow, etc.), that we have solved in this book. Even if this is the case, recognizing such a problem in its natural habitat—grungy and obscured by reality and context—requires practice and skill. It is more likely that you will need to reduce your problem to one of these lucky ones—or to solve it using dynamic programming or linear programming.

But chances are that nothing like this will happen. The world of search problems is a bleak landscape. There are a few spots of light—brilliant algorithmic ideas—each illuminating a small area around it (the problems that reduce to it; two of these areas, linear and dynamic programming, are in fact decently large). But the remaining vast expanse is pitch dark: **NP**-complete. What are you to do?

You can start by proving that your problem is actually **NP**-complete. Often a proof by generalization (recall the discussion on page 270 and Exercise 8.10) is all that you need; and sometimes a simple reduction from 3SAT or ZOE is not too difficult to find. This sounds like a theoretical exercise, but, if carried out successfully, it does bring some tangible rewards: now your status in the team has been elevated, you are no longer the kid who can't do, and you have become the noble knight with the impossible quest.

But, unfortunately, a problem does not go away when proved **NP**-complete. The real question is, *What do you do next?*

This is the subject of the present chapter and also the inspiration for some of the most important modern research on algorithms and complexity. **NP**-completeness is not a death certificate—it is only the beginning of a fascinating adventure.

Your problem's **NP**-completeness proof probably constructs graphs that are complicated and weird, very much unlike those that come up in your application. For example, even though SAT is **NP**-complete, satisfying assignments for HORN SAT (the instances of SAT that come up in logic programming) can be found efficiently (recall Section 5.3). Or, suppose the graphs that arise in your application are trees. In this case, many **NP**-complete problems,

such as INDEPENDENT SET, can be solved in linear time by dynamic programming (recall Section 6.7).

Unfortunately, this approach does not always work. For example, we know that 3SAT is **NP**-complete. And the INDEPENDENT SET problem, along with many other **NP**-complete problems, remains so even for planar graphs (graphs that can be drawn in the plane without crossing edges). Moreover, often you cannot neatly characterize the instances that come up in your application. Instead, you will have to rely on some form of *intelligent exponential search*—procedures such as *backtracking* and *branch and bound* which are exponential time in the worst-case, but, with the right design, could be very efficient on typical instances that come up in your application. We discuss these methods in Section 9.1.

Or you can develop an algorithm for your **NP**-complete optimization problem that falls short of the optimum *but never by too much*. For example, in Section 5.4 we saw that the greedy algorithm always produces a set cover that is no more than $\log n$ times the optimal set cover. An algorithm that achieves such a guarantee is called an *approximation algorithm*. As we will see in Section 9.2, such algorithms are known for many **NP**-complete optimization problems, and they are some of the most clever and sophisticated algorithms around. And the theory of **NP**-completeness can again be used as a guide in this endeavor, by showing that, for some problems, there are even severe limits to how well they can be approximated—unless of course $\mathbf{P} = \mathbf{NP}$.

Finally, there are *heuristics*, algorithms with no guarantees on either the running time or the degree of approximation. Heuristics rely on ingenuity, intuition, a good understanding of the application, meticulous experimentation, and often insights from physics or biology, to attack a problem. We see some common kinds in Section 9.3.

9.1 Intelligent exhaustive search

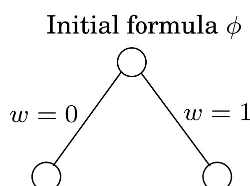
9.1.1 Backtracking

Backtracking is based on the observation that it is often possible to reject a solution by looking at just a small portion of it. For example, if an instance of SAT contains the clause $(x_1 \vee x_2)$, then all assignments with $x_1 = x_2 = 0$ (i.e., `false`) can be instantly eliminated. To put it differently, by quickly checking and discrediting this *partial assignment*, we are able to prune a quarter of the entire search space. A promising direction, but can it be systematically exploited?

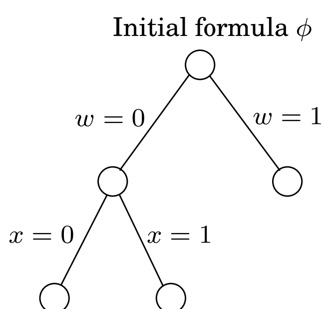
Here's how it is done. Consider the Boolean formula $\phi(w, x, y, z)$ specified by the set of clauses

$$(w \vee x \vee y \vee z),\ (w \vee \overline{x}),\ (x \vee \overline{y}),\ (y \vee \overline{z}),\ (z \vee \overline{w}),\ (\overline{w} \vee \overline{z}).$$

We will incrementally grow a tree of partial solutions. We start by branching on any one variable, say w:



Plugging $w = 0$ and $w = 1$ into ϕ, we find that no clause is immediately violated and thus neither of these two partial assignments can be eliminated outright. So we need to keep branching. We can expand either of the two available nodes, and on any variable of our choice. Let's try this one:



This time, we are in luck. The partial assignment $w = 0, x = 1$ violates the clause $(w \vee \overline{x})$ and can be terminated, thereby pruning a good chunk of the search space. We backtrack out of this cul-de-sac and continue our explorations at one of the two remaining active nodes.

In this manner, backtracking explores the space of assignments, growing the tree only at nodes where there is uncertainty about the outcome, and stopping if at any stage a satisfying assignment is encountered.

In the case of Boolean satisfiability, each node of the search tree can be described either by a partial assignment or by the clauses that remain when those values are plugged into the original formula. For instance, if $w = 0$ and $x = 0$ then any clause with $\overline{w}$ or $\overline{x}$ is instantly satisfied and any literal w or x is not satisfied and can be removed. What's left is

$$(y \vee z), (\overline{y}), (y \vee \overline{z}).$$

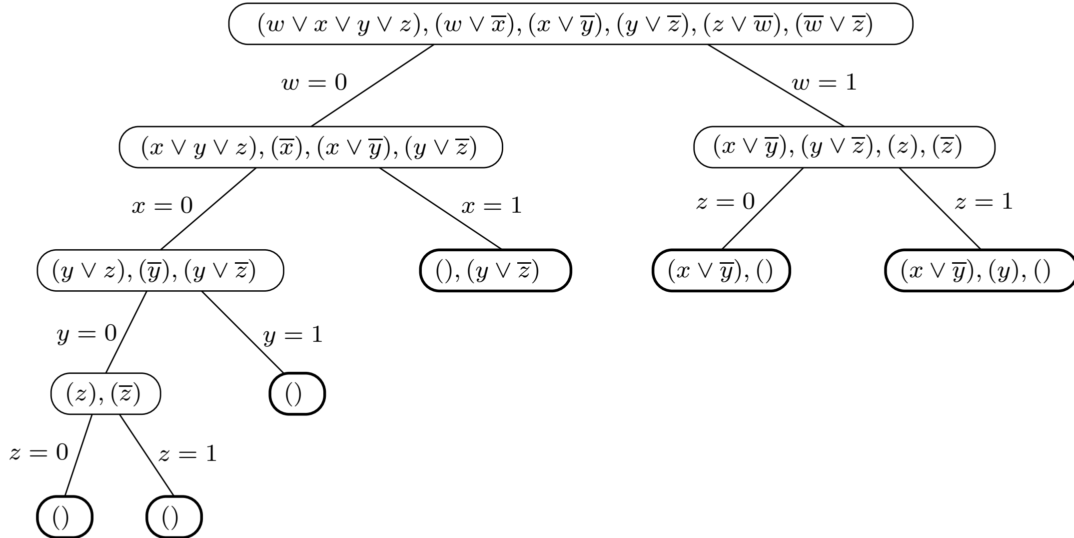
Likewise, $w = 0$ and $x = 1$ leaves

$$(), (y \vee \overline{z}),$$

with the "empty clause" ( ) ruling out satisfiability. Thus the nodes of the search tree, representing partial assignments, are themselves SAT *subproblems*.

This alternative representation is helpful for making the two decisions that repeatedly arise: which subproblem to expand next, and which branching variable to use. Since the benefit of backtracking lies in its ability to eliminate portions of the search space, and since this happens only when an empty clause is encountered, it makes sense to choose the subproblem that contains the *smallest* clause and to then branch on a variable in that clause. If this clause

Figure 9.1 Backtracking reveals that ϕ is not satisfiable.



happens to be a singleton, then at least one of the resulting branches will be terminated. (If there is a tie in choosing subproblems, one reasonable policy is to pick the one lowest in the tree, in the hope that it is close to a satisfying assignment.) See Figure 9.1 for the conclusion of our earlier example.

More abstractly, a backtracking algorithm requires a *test* that looks at a subproblem and quickly declares one of three outcomes:

1. Failure: the subproblem has no solution.
2. Success: a solution to the subproblem is found.
3. Uncertainty.

In the case of SAT, this test declares failure if there is an empty clause, success if there are no clauses, and uncertainty otherwise. The backtracking procedure then has the following format.

```
Start with some problem P_0
Let S = {P_0}, the set of active subproblems
Repeat while S is nonempty:
  choose a subproblem P ∈ S and remove it from S
  expand it into smaller subproblems P_1, P_2, ..., P_k
  For each P_i:
    If test(P_i) succeeds:  halt and announce this solution
    If test(P_i) fails:  discard P_i
```

```
    Otherwise:  add P_i to S
  Announce that there is no solution
```

For SAT, the `choose` procedure picks a clause, and `expand` picks a variable within that clause. We have already discussed some reasonable ways of making such choices.

With the right `test`, `expand`, and `choose` routines, backtracking can be remarkably effective in practice. The backtracking algorithm we showed for SAT is the basis of many successful satisfiability programs. Another sign of quality is this: if presented with a 2SAT instance, it will always find a satisfying assignment, if one exists, in polynomial time (Exercise 9.1)!

9.1.2 Branch-and-bound

The same principle can be generalized from search problems such as SAT to optimization problems. For concreteness, let's say we have a minimization problem; maximization will follow the same pattern.

As before, we will deal with partial solutions, each of which represents a *subproblem*, namely, what is the (cost of the) best way to complete this solution? And as before, we need a basis for eliminating partial solutions, since there is no other source of efficiency in our method. To reject a subproblem, we must be certain that its cost exceeds that of some other solution we have already encountered. But its exact cost is unknown to us and is generally not efficiently computable. So instead we use a quick *lower bound* on this cost.

```
Start with some problem P_0
Let S = {P_0}, the set of active subproblems
bestsofar = ∞
Repeat while S is nonempty:
  choose a subproblem (partial solution) P ∈ S and remove it from S
  expand it into smaller subproblems P_1, P_2, ..., P_k
  For each P_i:
    If P_i is a complete solution:  update bestsofar
    else if lowerbound(P_i) < bestsofar:  add P_i to S
return bestsofar
```

Let's see how this works for the traveling salesman problem on a graph $G = (V, E)$ with edge lengths $d_e > 0$. A partial solution is a simple path $a \rightsquigarrow b$ passing through some vertices $S \subseteq V$, where S includes the endpoints a and b. We can denote such a partial solution by the tuple $[a, S, b]$—in fact, a will be fixed throughout the algorithm. The corresponding subproblem is to find the best completion of the tour, that is, the cheapest complementary path $b \rightsquigarrow a$ with intermediate nodes $V - S$. Notice that the initial problem is of the form $[a, \{a\}, a]$ for any $a \in V$ of our choosing.

At each step of the branch-and-bound algorithm, we extend a particular partial solution $[a, S, b]$ by a single edge (b, x), where $x \in V - S$. There can be up to $|V - S|$ ways to do this, and each of these branches leads to a subproblem of the form $[a, S \cup \{x\}, x]$.

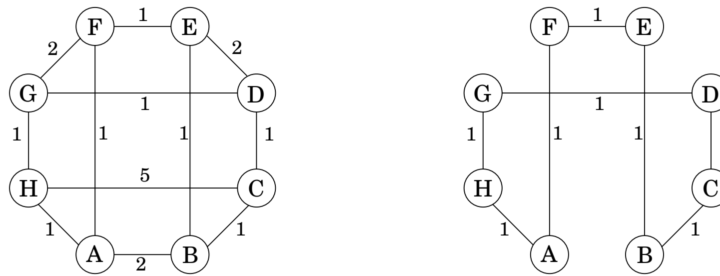
How can we lower-bound the cost of completing a partial tour $[a, S, b]$? Many sophisticated methods have been developed for this, but let's look at a rather simple one. The remainder of the tour consists of a path through $V - S$, plus edges from a and b to $V - S$. Therefore, its cost is at least the sum of the following:

1. The lightest edge from a to $V - S$.
2. The lightest edge from b to $V - S$.
3. The minimum spanning tree of $V - S$.

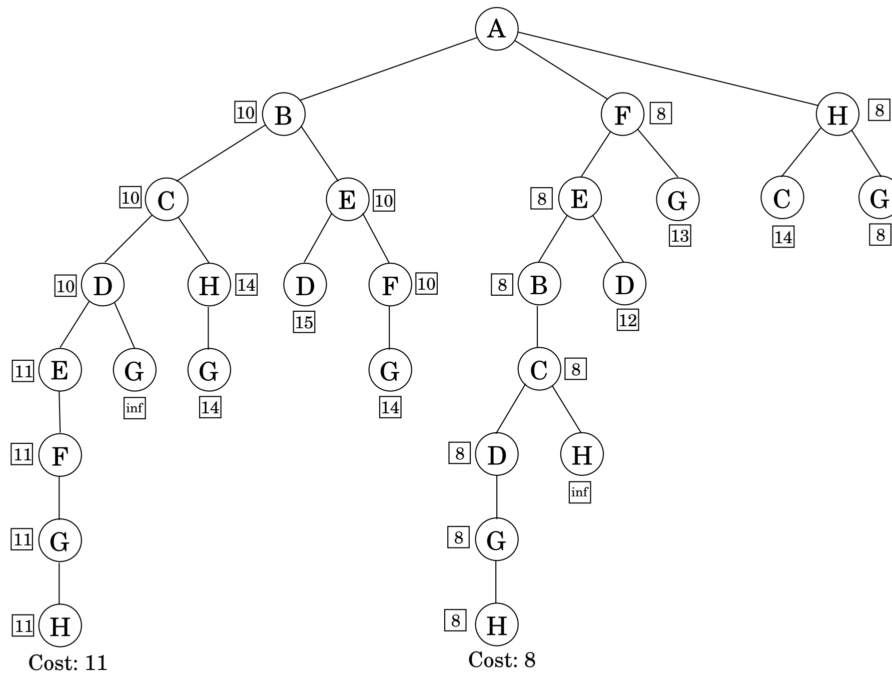
(Do you see why?) And this lower bound can be computed quickly by a minimum spanning tree algorithm. Figure 9.2 runs through an example: each node of the tree represents a partial tour (specifically, the path from the root to that node) that at some stage is considered by the branch-and-bound procedure. Notice how just 28 partial solutions are considered, instead of the $7! = 5{,}040$ that would arise in a brute-force search.

Figure 9.2 (a) A graph and its optimal traveling salesman tour. (b) The branch-and-bound search tree, explored left to right. Boxed numbers indicate lower bounds on cost.

(a)



(b)



9.2 Approximation algorithms

In an optimization problem we are given an instance I and are asked to find the optimum solution—the one with the maximum gain if we have a maximization problem like INDEPENDENT SET, or the minimum cost if we are dealing with a minimization problem such as the TSP. For every instance I, let us denote by $\text{OPT}(I)$ the value (benefit or cost) of the optimum solution. It makes the math a little simpler (and is not too far from the truth) to *assume that* $\text{OPT}(I)$ *is always a positive integer.*

We have already seen an example of a (famous) approximation algorithm in Section 5.4: the greedy scheme for SET COVER. For any instance I of size n, we showed that this greedy algorithm is guaranteed to quickly find a set cover of cardinality at most $\text{OPT}(I) \log n$. This $\log n$ factor is known as the approximation guarantee of the algorithm.

More generally, consider any minimization problem. Suppose now that we have an algorithm $\mathcal{A}$ for our problem which, given an instance I, returns a solution with value $\mathcal{A}(I)$. The *approximation ratio* of algorithm $\mathcal{A}$ is defined to be

$$\alpha_{\mathcal{A}} = \max_{I} \frac{\mathcal{A}(I)}{\text{OPT}(I)}.$$

In other words, $\alpha_{\mathcal{A}}$ measures by the factor by which the output of algorithm $\mathcal{A}$ exceeds the optimal solution, on the worst-case input. The approximation ratio can also be defined for maximization problems, such as INDEPENDENT SET, in the same way—except that to get a number larger than 1 we take the reciprocal.

So, when faced with an **NP**-complete optimization problem, a reasonable goal is to look for an approximation algorithm $\mathcal{A}$ whose $\alpha_{\mathcal{A}}$ is as small as possible. But this kind of guarantee might seem a little puzzling: How can we come close to the optimum if we cannot determine the optimum? Let's look at a simple example.

9.2.1 Vertex cover

We already know the VERTEX COVER problem is **NP**-hard.

VERTEX COVER

Input: An undirected graph $G = (V, E)$.

Output: A subset of the vertices $S \subseteq V$ that touches every edge.

Goal: Minimize $|S|$.

See Figure 9.3 for an example.

Since VERTEX COVER is a special case of SET COVER, we know from Chapter 5 that it can be approximated within a factor of $O(\log n)$ by the greedy algorithm: repeatedly delete the vertex of highest degree and include it in the vertex cover. And there are graphs on which the greedy algorithm returns a vertex cover that is indeed $\log n$ times the optimum.

A better approximation algorithm for VERTEX COVER is based on the notion of a *matching*, a subset of edges that have no vertices in common (Figure 9.4). A matching is *maximal* if no