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

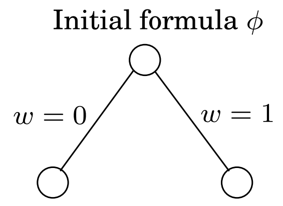
9.1.1 Backtracking

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?

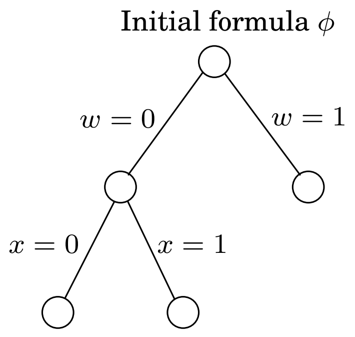
Here's how it is done. Consider the Boolean formula $\phi(w, x, y, z)$ specified by the set of clauses

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

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

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

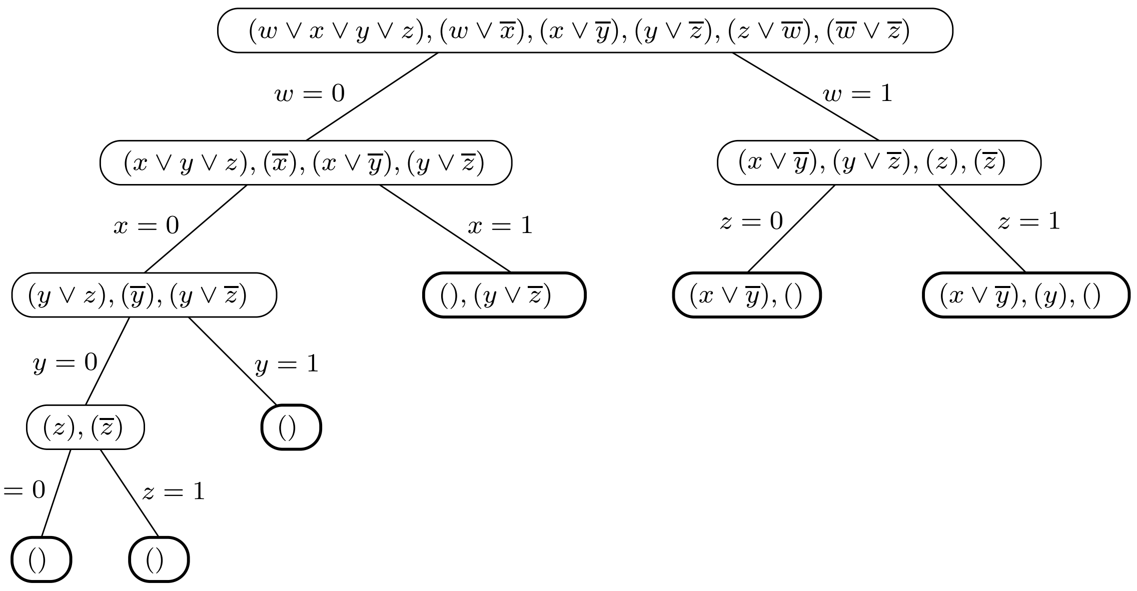
Likewise, $w = 0$ and $x = 1$ leaves

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

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.



$(w \vee x \vee y \vee z), (w \vee \overline{x}), (x \vee \overline{y}), (y \vee \overline{z}), (z \vee \overline{w}), (\overline{w} \vee \overline{z})$
$w = 0$
$w = 1$
$(x \vee y \vee z), (\overline{x}), (x \vee \overline{y}), (y \vee \overline{z})$
$(x \vee \overline{y}), (y \vee \overline{z}), (z), (\overline{z})$
$x = 0$
$x = 1$
$z = 0$
$z = 1$
$(y \vee z), (\overline{y}), (y \vee \overline{z})$
$(), (y \vee \overline{z})$
$(x \vee \overline{y}), ()$
$(x \vee \overline{y}), (y), ()$
$y = 0$
$y = 1$
$(z), (\overline{z})$
$()$
$z = 0$
$z = 1$
$()$
$()$

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 $\mathcal{S} = \{P_0\}$, the set of active subproblems
Repeat while $\mathcal{S}$ is nonempty:
  choose a subproblem $P \in \mathcal{S}$ and remove it from $\mathcal{S}$
  expand it into smaller subproblems $P_1, P_2, \ldots, P_k$
  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 $\mathcal{S}$
  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 $\mathcal{S} = \{P_0\}$, the set of active subproblems
bestsofar $= \infty$
Repeat while $\mathcal{S}$ is nonempty:
  choose a subproblem (partial solution) $P \in \mathcal{S}$ and remove it from $\mathcal{S}$
  expand it into smaller subproblems $P_1, P_2, \ldots, P_k$
  For each $P_i$:
    If $P_i$ is a complete solution:  update bestsofar
    else if lowerbound$(P_i) <$ bestsofar:  add $P_i$ to $\mathcal{S}$
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 $[a, S \cup \{x\}, x]$.

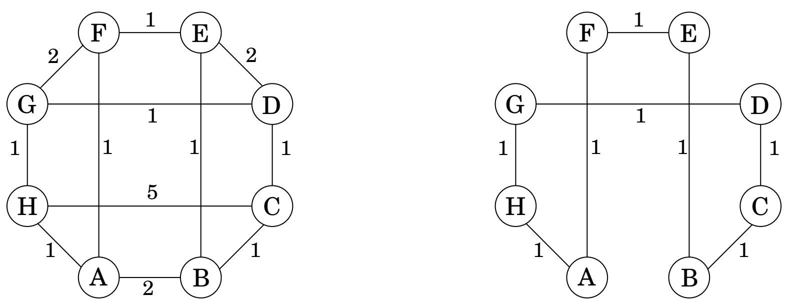
How can we lower-bound the cost of completing a partial tour $[a, S, b]$? Many sophisticated methods have been developed for this, but let's look at a rather simple one. The remainder of the tour consists of a path through $V - S$, plus edges from a and b to $V - S$. Therefore, its cost is at least the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

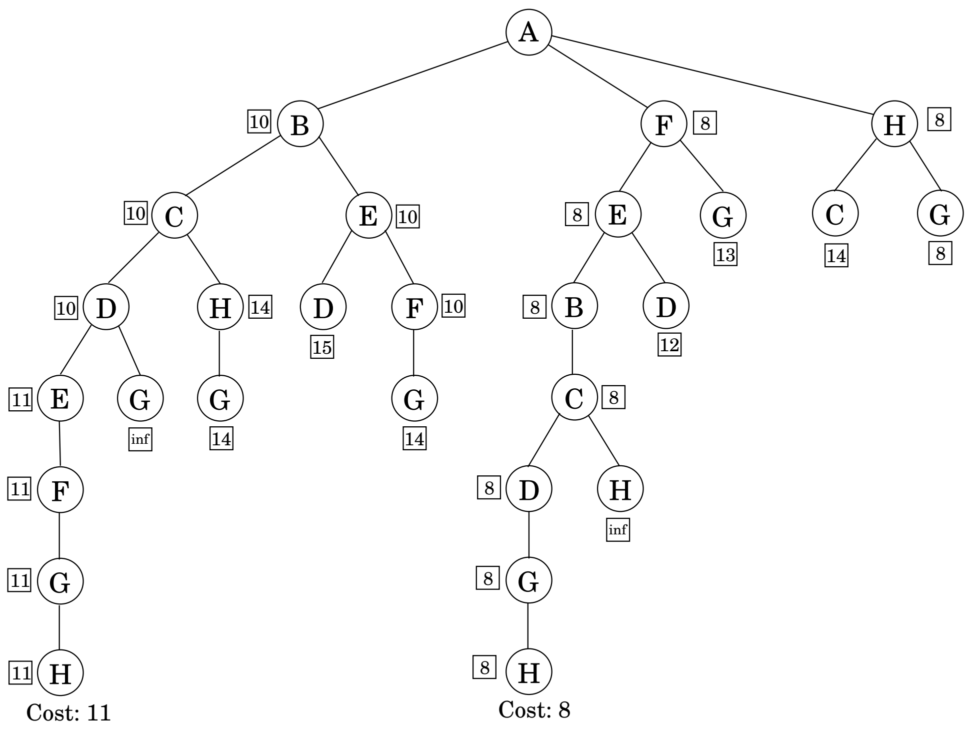
(Do you see why?) And this lower bound can be computed quickly by a minimum spanning tree algorithm. Figure 9.2 runs through an example: each node of the tree represents a partial tour (specifically, the path from the root to that node) that at some stage is considered by the branch-and-bound procedure. 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

Figure 9.3 A graph whose optimal vertex cover, shown shaded, is of size 8.

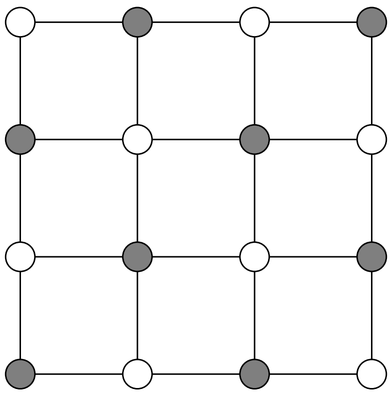
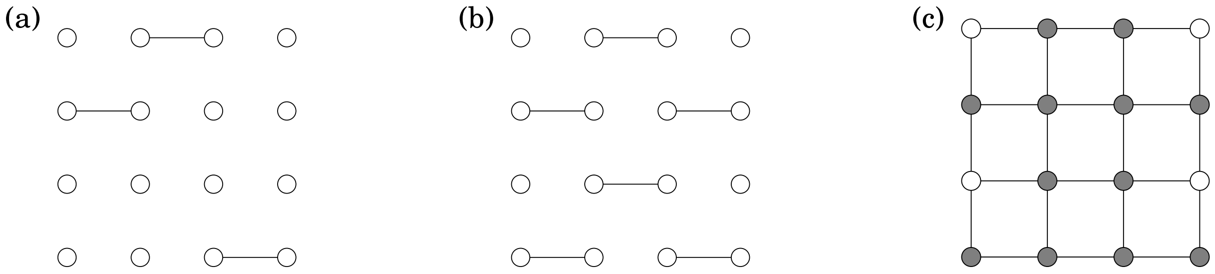


Figure 9.4 (a) A matching, (b) its completion to a maximal matching, and (c) the resulting vertex cover.



more edges can be added to it. Maximal matchings will help us find good vertex covers, and moreover, they are easy to generate: repeatedly pick edges that are disjoint from the ones chosen already, until this is no longer possible.

What is the relationship between matchings and vertex covers? Here is the crucial fact: any vertex cover of a graph G must be at least as large as the number of edges in any matching in G; that is, *any matching provides a lower bound on* OPT. This is simply because each edge of the matching must be covered by one of its endpoints in any vertex cover! Finding such a lower bound is a key step in designing an approximation algorithm, because we must compare the quality of the solution found by our algorithm to OPT, which is **NP**-complete to compute.

One more observation completes the design of our approximation algorithm: let S be a set that contains both endpoints of each edge in a maximal matching M. Then S must be a vertex cover—if it isn't, that is, if it doesn't touch some edge $e \in E$, then M could not possibly be maximal since we could still add e to it. But our cover S has $2|M|$ vertices. And from the previous paragraph we know that *any* vertex cover must have size at least $|M|$. So we're done.

Here's the algorithm for VERTEX COVER.

```
Find a maximal matching $M \subseteq E$
```

```
    Return $S = \{$all endpoints of edges in $M\}$
```

This simple procedure always returns a vertex cover whose size is at most twice optimal!

In summary, even though we have no way of finding the best vertex cover, we can easily find another structure, a maximal matching, with two key properties:

1. Its size gives us a lower bound on the optimal vertex cover.
2. It can be used to build a vertex cover, whose size can be related to that of the optimal cover using property 1.

Thus, this simple algorithm has an approximation ratio of $\alpha_{\mathcal{A}} \leq 2$. In fact, it is not hard to find examples on which it does make a 100% error; hence $\alpha_{\mathcal{A}} = 2$.

9.2.2 Clustering

We turn next to a *clustering* problem, in which we have some data (text documents, say, or images, or speech samples) that we want to divide into groups. It is often useful to define "distances" between these data points, numbers that capture how close or far they are from one another. Often the data are true points in some high-dimensional space and the distances are the usual Euclidean distance; in other cases, the distances are the result of some "similarity tests" to which we have subjected the data points. Assume that we have such distances and that they satisfy the usual *metric* properties:

1. $d(x, y) \geq 0$ for all x, y.
2. $d(x, y) = 0$ if and only if $x = y$.
3. $d(x, y) = d(y, x)$.
4. (Triangle inequality) $d(x, y) \leq d(x, z) + d(z, y)$.

We would like to partition the data points into groups that are compact in the sense of having small diameter.

k-CLUSTER

Input: Points $X = \{x_1, \ldots, x_n\}$ with underlying distance metric $d(\cdot, \cdot)$; integer k.

Output: A partition of the points into k clusters $C_1, \ldots, C_k$.

Goal: Minimize the diameter of the clusters,

$$\max_j \max_{x_a, x_b \in C_j} d(x_a, x_b).$$

One way to visualize this task is to imagine n points in space, which are to be covered by k spheres of equal size. What is the smallest possible diameter of the spheres? Figure 9.5 shows an example.

This problem is **NP**-hard, but has a very simple approximation algorithm. The idea is to pick k of the data points as *cluster centers* and to then assign each of the remaining points to

Figure 9.5 Some data points and the optimal $k = 4$ clusters.

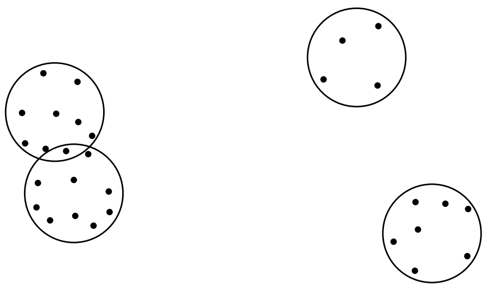
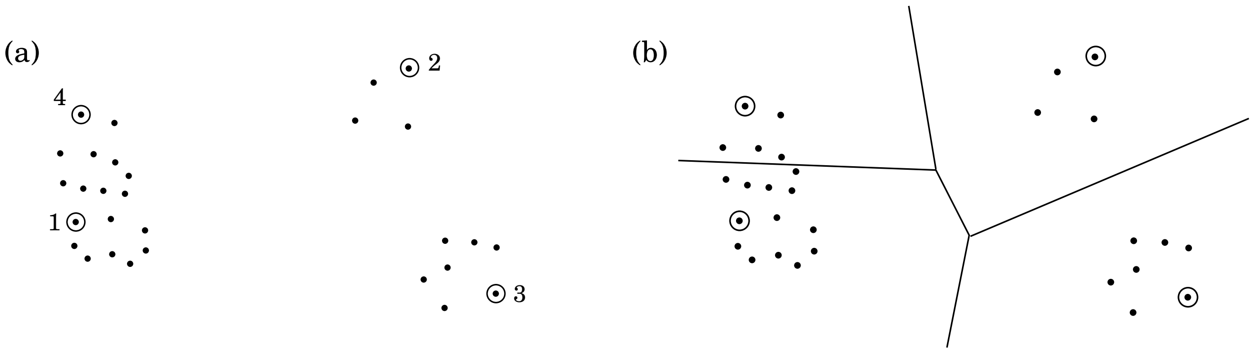


Figure 9.6 (a) Four centers chosen by farthest-first traversal. (b) The resulting clusters.



the center closest to it, thus creating k clusters. The centers are picked one at a time, using an intuitive rule: always pick the next center to be as far as possible from the centers chosen so far (see Figure 9.6).

```
Pick any point $\mu_1 \in X$ as the first cluster center
for $i = 2$ to $k$:
  Let $\mu_i$ be the point in $X$ that is farthest from $\mu_1, \ldots, \mu_{i-1}$
  (i.e., that maximizes $\min_{j<i} d(\cdot, \mu_j)$)
Create $k$ clusters:  $C_i = \{$all $x \in X$ whose closest center is $\mu_i\}$
```

It's clear that this algorithm returns a valid partition. What's more, the resulting diameter is guaranteed to be at most twice optimal.

Here's the argument. Let $x \in X$ be the point farthest from $\mu_1, \ldots, \mu_k$ (in other words the next center we would have chosen, if we wanted $k + 1$ of them), and let r be its distance to its closest center. Then every point in X must be within distance r of its cluster center. By the triangle inequality, this means that every cluster has diameter at most $2r$.

But how does r relate to the diameter of the optimal clustering? Well, we have identified $k + 1$ points $\{\mu_1, \mu_2, \ldots, \mu_k, x\}$ that are all at a distance at least r from each other (why?). *Any* partition into k clusters must put two of these points in the same cluster and must therefore have diameter at least r.

This algorithm has a certain high-level similarity to our scheme for VERTEX COVER. Instead of a maximal matching, we use a different easily computable structure—a set of k points that cover all of X within some radius r, while at the same time being mutually separated by a distance of at least r. This structure is used both to generate a clustering and to give a lower bound on the optimal clustering.

We know of no better approximation algorithm for this problem.

9.2.3 TSP

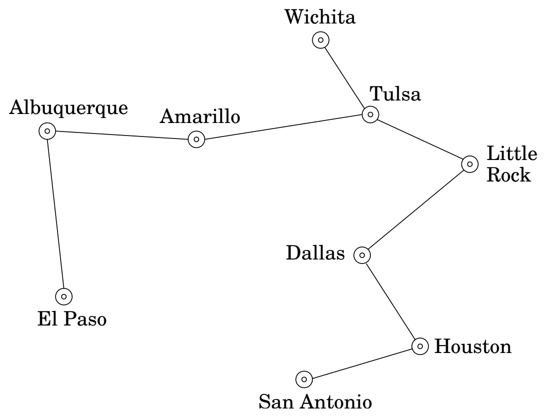
The triangle inequality played a crucial role in making the k-CLUSTER problem approximable. It also helps with the TRAVELING SALESMAN PROBLEM: if the distances between cities satisfy the metric properties, then there is an algorithm that outputs a tour of length at most 1.5 times optimal. We'll now look at a slightly weaker result that achieves a factor of 2.

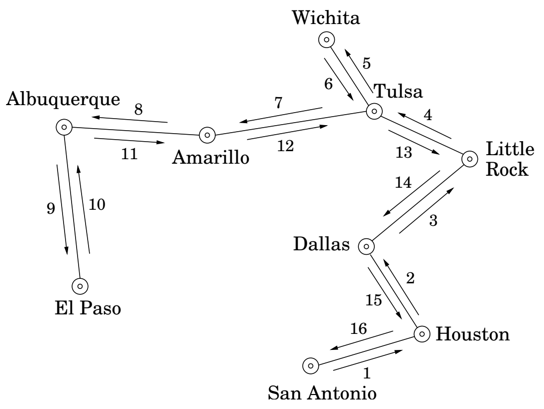
Continuing with the thought processes of our previous two approximation algorithms, we can ask whether there is some structure that is easy to compute and that is plausibly related to the best traveling salesman tour (as well as providing a good lower bound on OPT). A little thought and experimentation reveals the answer to be the *minimum spanning tree*.

Let's understand this relation. Removing any edge from a traveling salesman tour leaves a path through all the vertices, which is a spanning tree. Therefore,

$$\text{TSP cost} \;\geq\; \text{cost of this path} \;\geq\; \text{MST cost}.$$

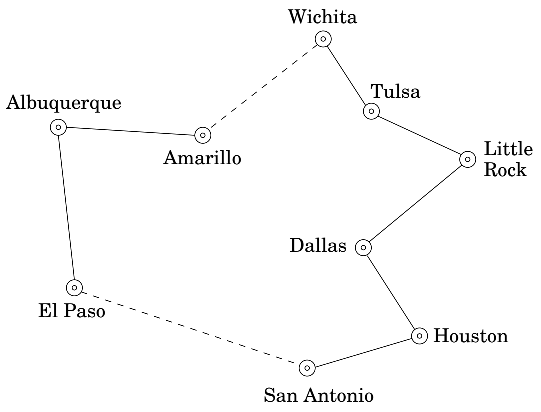
Now, we somehow need to use the MST to build a traveling salesman tour. If we can use each edge *twice*, then by following the shape of the MST we end up with a tour that visits all the cities, some of them more than once. Here's an example, with the MST on the left and the resulting tour on the right (the numbers show the order in which the edges are taken).





Therefore, this tour has a length at most twice the MST cost, which as we've already seen is at most twice the TSP cost.

This is the result we wanted, but we aren't quite done because our tour visits some cities multiple times and is therefore not legal. To fix the problem, the tour should simply skip any city it is about to revisit, and instead move directly to the next *new* city in its list:



By the triangle inequality, these bypasses can only make the overall tour shorter.

General TSP

But what if we are interested in instances of TSP that do not satisfy the triangle inequality? It turns out that this is a *much* harder problem to approximate.

Here is why: Recall that on page 274 we gave a polynomial-time reduction which given any graph G and integer any $C > 0$ produces an instance $I(G, C)$ of the TSP such that:

(i) If G has a Rudrata path, then $\text{OPT}(I(G, C)) = n$, the number of vertices in G.

(ii) If G has no Rudrata path, then $\text{OPT}(I(G, C)) \geq n + C$.

This means that even an approximate solution to TSP would enable us to solve RUDRATA PATH! Let's work out the details.

Consider an approximation algorithm $\mathcal{A}$ for TSP and let $\alpha_{\mathcal{A}}$ denote its approximation ratio. From any instance G of RUDRATA PATH, we will create an instance $I(G, C)$ of TSP using the specific constant $C = n\alpha_{\mathcal{A}}$. What happens when algorithm $\mathcal{A}$ is run on this TSP instance? In case (i), it must output a tour of length at most $\alpha_{\mathcal{A}}\text{OPT}(I(G, C)) = n\alpha_{\mathcal{A}}$, whereas in case (ii) it must output a tour of length at least $\text{OPT}(I(G, C)) > n\alpha_{\mathcal{A}}$. Thus we can figure out whether G has a Rudrata path! Here is the resulting procedure:

```
Given any graph $G$:
   compute $I(G,C)$ (with $C = n \cdot \alpha_{\mathcal{A}}$) and run algorithm $\mathcal{A}$ on it
   if the resulting tour has length $\leq n\alpha_{\mathcal{A}}$:
      conclude that $G$ has a Rudrata path
   else:  conclude that $G$ has no Rudrata path
```

This tells us whether or not G has a Rudrata path; by calling the procedure a polynomial number of times, we can find the actual path (Exercise 8.2).

We've shown that if TSP has a polynomial-time approximation algorithm, then there is a polynomial algorithm for the **NP**-complete RUDRATA PATH problem. So, unless **P** = **NP**, there cannot exist an efficient approximation algorithm for the TSP.

9.2.4 Knapsack

Our last approximation algorithm is for a maximization problem and has a very impressive guarantee: given any $\epsilon > 0$, it will return a solution of value at least $(1-\epsilon)$ times the optimal value, in time that scales only polynomially in the input size and in $1/\epsilon$.

The problem is KNAPSACK, which we first encountered in Chapter 6. There are n items, with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ (all positive integers), and the goal is to pick the most valuable combination of items subject to the constraint that their total weight is at most W.

Earlier we saw a dynamic programming solution to this problem with running time $O(nW)$. Using a similar technique, a running time of $O(nV)$ can also be achieved, where V is the sum of the values. Neither of these running times is polynomial, because W and V can be very large, exponential in the size of the input.

Let's consider the $O(nV)$ algorithm. In the bad case when V is large, what if we simply scale down all the values in some way? For instance, if

$$v_1 = 117{,}586{,}003,\ v_2 = 738{,}493{,}291,\ v_3 = 238{,}827{,}453,$$

we could simply knock off some precision and instead use 117, 738, and 238. This doesn't change the problem all that much and will make the algorithm much, much faster!

Now for the details. Along with the input, the user is assumed to have specified some approximation factor $\epsilon > 0$.

```
Discard any item with weight $> W$
Let $v_{\max} = \max_i v_i$
Rescale values $\widehat{v}_i = \lfloor v_i \cdot \frac{n}{\epsilon v_{\max}} \rfloor$
Run the dynamic programming algorithm with values $\{\widehat{v}_i\}$
Output the resulting choice of items
```

Let's see why this works. First of all, since the rescaled values $\widehat{v}_i$ are all at most n/ϵ, the dynamic program is efficient, running in time $O(n^3/\epsilon)$.

Now suppose the optimal solution to the original problem is to pick some subset of items S, with total value K^*. The rescaled value of this same assignment is

$$\sum_{i\in S} \widehat{v}_i \;=\; \sum_{i\in S} \left\lfloor v_i \cdot \frac{n}{\epsilon v_{\max}} \right\rfloor \;\geq\; \sum_{i\in S} \left(v_i \cdot \frac{n}{\epsilon v_{\max}} - 1 \right) \;\geq\; K^* \cdot \frac{n}{\epsilon v_{\max}} - n.$$

Therefore, the optimal assignment for the shrunken problem, call it $\widehat{S}$, has a rescaled value of at least this much. In terms of the original values, assignment $\widehat{S}$ has a value of at least

$$\sum_{i\in \widehat{S}} v_i \;\geq\; \sum_{i\in \widehat{S}} \widehat{v}_i \cdot \frac{\epsilon v_{\max}}{n} \;\geq\; \left(K^* \cdot \frac{n}{\epsilon v_{\max}} - n \right) \cdot \frac{\epsilon v_{\max}}{n} \;=\; K^* - \epsilon v_{\max} \;\geq\; K^*(1-\epsilon).$$

9.2.5 The approximability hierarchy

Given any **NP**-complete optimization problem, we seek the best approximation algorithm possible. Failing this, we try to prove *lower bounds* on the approximation ratios that are achievable in polynomial time (we just carried out such a proof for the general TSP). All told, **NP**-complete optimization problems are classified as follows:

- Those for which, like the TSP, no finite approximation ratio is possible.
- Those for which an approximation ratio is possible, but there are limits to how small this can be. VERTEX COVER, k-CLUSTER, and the TSP with triangle inequality belong here. (For these problems we have not established limits to their approximability, but these limits do exist, and their proofs constitute some of the most sophisticated results in this field.)
- Down below we have a more fortunate class of **NP**-complete problems for which approximability has no limits, and polynomial approximation algorithms with error ratios arbitrarily close to zero exist. KNAPSACK resides here.
- Finally, there is another class of problems, between the first two given here, for which the approximation ratio is about $\log n$. SET COVER is an example.

(A humbling reminder: All this is contingent upon the assumption $\mathbf{P} \neq \mathbf{NP}$. Failing this, this hierarchy collapses down to **P**, and all **NP**-complete optimization problems can be solved exactly in polynomial time.)

A final point on approximation algorithms: often these algorithms, or their variants, perform much better on typical instances than their worst-case approximation ratio would have you believe.

9.3 Local search heuristics

Our next strategy for coping with **NP**-completeness is inspired by evolution (which is, after all, the world's best-tested optimization procedure)—by its incremental process of introducing small mutations, trying them out, and keeping them if they work well. This paradigm is called *local search* and can be applied to any optimization task. Here's how it looks for a minimization problem.

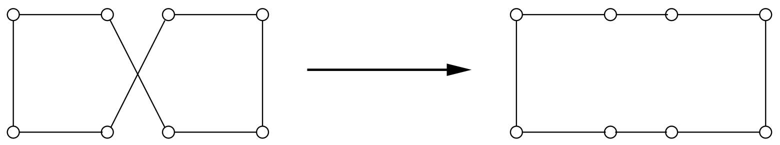
```
let $s$ be any initial solution
while there is some solution $s'$ in the neighborhood of $s$
   for which cost($s'$) < cost($s$):  replace $s$ by $s'$
return $s$
```

On each iteration, the current solution is replaced by a better one close to it, in its *neighborhood*. This neighborhood structure is something we impose upon the problem and is the central design decision in local search. As an illustration, let's revisit the traveling salesman problem.

9.3.1 Traveling salesman, once more

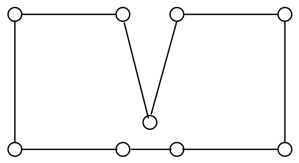
Assume we have all interpoint distances between n cities, giving a search space of $(n-1)!$ different tours. What is a good notion of neighborhood?

The most obvious notion is to consider two tours as being close if they differ in just a few edges. They can't differ in just one edge (do you see why?), so we will consider differences of two edges. We define the *2-change* neighborhood of tour s as being the set of tours that can be obtained by removing two edges of s and then putting in two other edges. Here's an example of a local move:

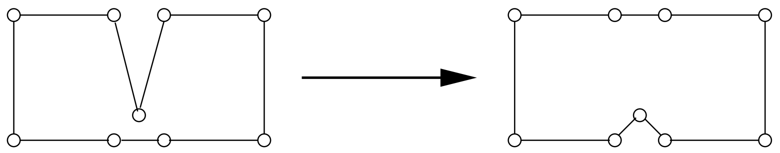


We now have a well-defined local search procedure. How does it measure up under our two standard criteria for algorithms—what is its overall running time, and does it always return the best solution?

Embarrassingly, neither of these questions has a satisfactory answer. Each iteration is certainly fast, because a tour has only $O(n^2)$ neighbors. However, it is not clear how many iterations will be needed: whether for instance, there might be an exponential number of them. Likewise, all we can easily say about the final tour is that it is *locally optimal*—that is, it is superior to the tours in its immediate neighborhood. There might be better solutions further away. For instance, the following picture shows a possible final answer that is clearly suboptimal; the range of local moves is simply too limited to improve upon it.



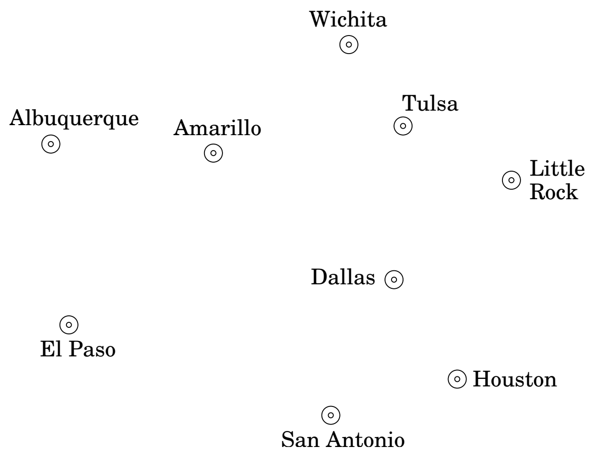
To overcome this, we may try a more generous neighborhood, for instance *3-change*, consisting of tours that differ on up to three edges. And indeed, the preceding bad case gets fixed:



But there is a downside, in that the size of a neighborhood becomes $O(n^3)$, making each iteration more expensive. Moreover, there may still be suboptimal local minima, although fewer than before. To avoid these, we would have to go up to *4-change*, or higher. In this manner, efficiency and quality often turn out to be competing considerations in a local search. Efficiency demands neighborhoods that can be searched quickly, but smaller neighborhoods can increase the abundance of low-quality local optima. The appropriate compromise is typically determined by experimentation.

Figure 9.7 (a) Nine American cities. (b) Local search, starting at a random tour, and using 3-change. The traveling salesman tour is found after three moves.

(a)



(b)

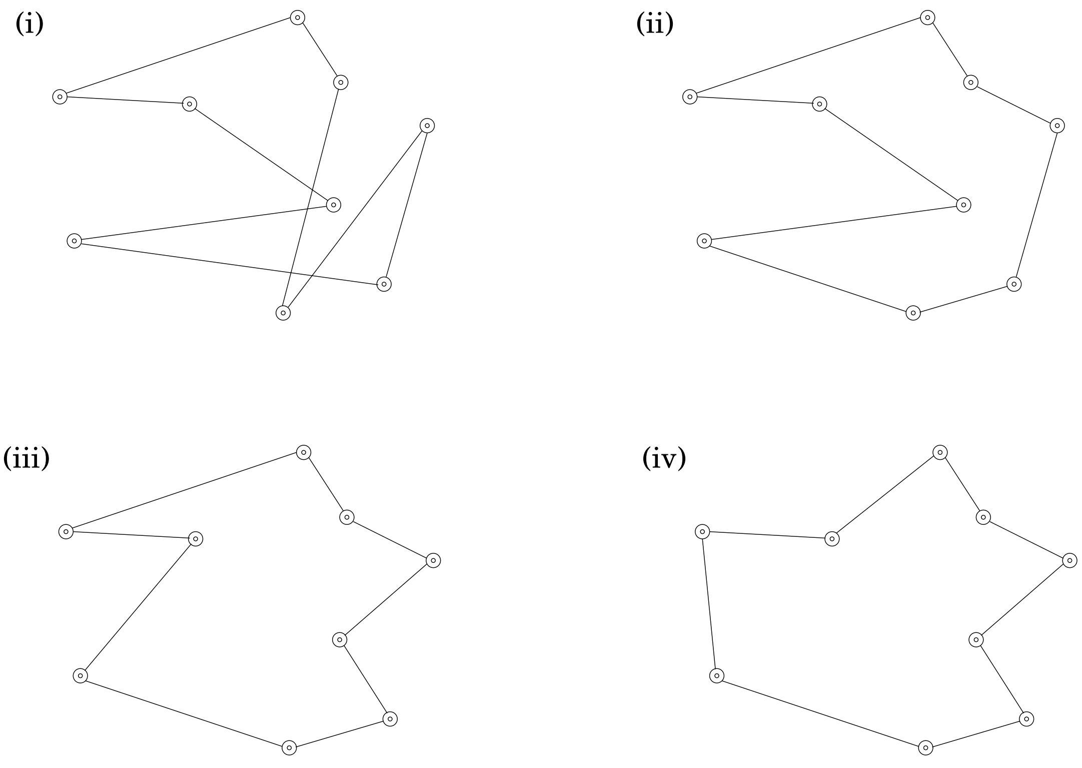


Figure 9.8 Local search.

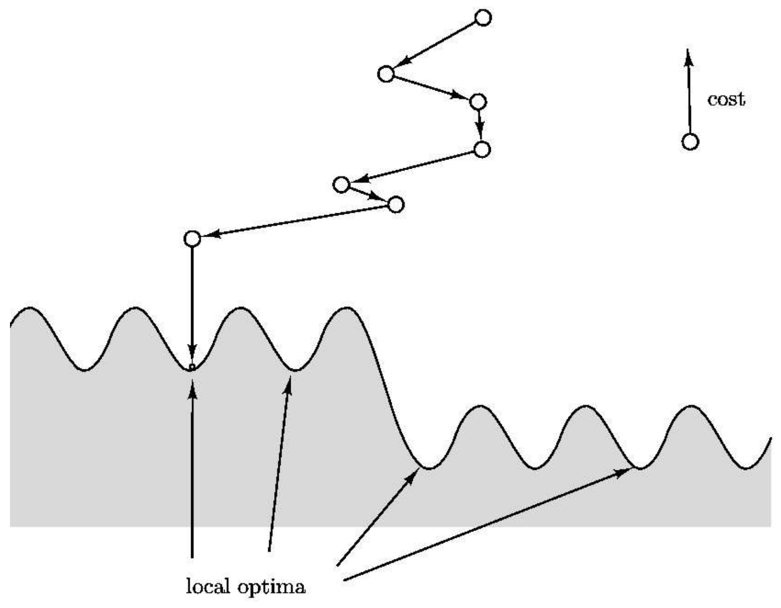


Figure 9.7 shows a specific example of local search at work. Figure 9.8 is a more abstract, stylized depiction of local search. The solutions crowd the unshaded area, and cost decreases when we move downward. Starting from an initial solution, the algorithm moves downhill until a local optimum is reached.

In general, the search space might be riddled with local optima, and some of them may be of very poor quality. The hope is that with a judicious choice of neighborhood structure, most local optima will be reasonable. Whether this is the reality or merely misplaced faith, it is an empirical fact that local search algorithms are the top performers on a broad range of optimization problems. Let's look at another such example.

9.3.2 Graph partitioning

The problem of graph partitioning arises in a diversity of applications, from circuit layout to program analysis to image segmentation. We saw a special case of it, BALANCED CUT, in Chapter 8.

GRAPH PARTITIONING

Input: An undirected graph $G = (V, E)$ with nonnegative edge weights; a real number $\alpha \in (0, 1/2]$.

Output: A partition of the vertices into two groups A and B, each of size at least $\alpha|V|$.

Goal: Minimize the capacity of the cut (A, B).

Figure 9.9 An instance of GRAPH PARTITIONING, with the optimal partition for $\alpha = 1/2$. Vertices on one side of the cut are shaded.

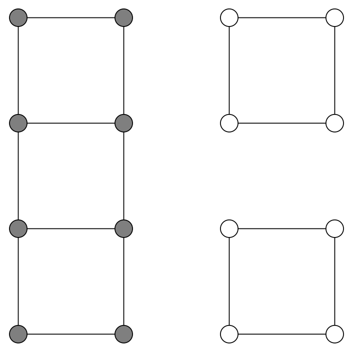
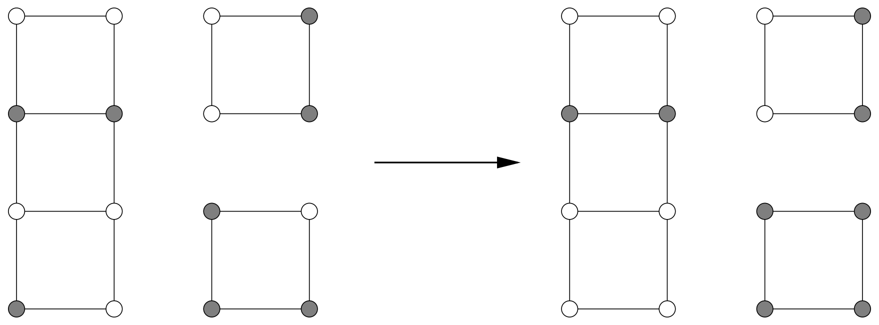
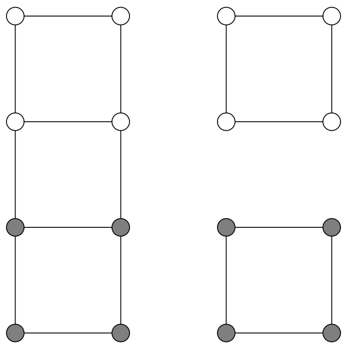


Figure 9.9 shows an example in which the graph has 16 nodes, all edge weights are 0 or 1, and the optimal solution has cost 0. Removing the restriction on the sizes of A and B would give the MINIMUM CUT problem, which we know to be efficiently solvable using flow techniques. The present variant, however, is **NP**-hard. In designing a local search algorithm, it will be a big convenience to focus on the special case $\alpha = 1/2$, in which A and B are forced to contain exactly half the vertices. The apparent loss of generality is purely cosmetic, as GRAPH PARTITIONING reduces to this particular case.

We need to decide upon a neighborhood structure for our problem, and there is one obvious way to do this. Let (A, B), with $|A| = |B|$, be a candidate solution; we will define its neighbors to be all solutions obtainable by swapping one pair of vertices across the cut, that is, all solutions of the form $(A - \{a\} + \{b\}, B - \{b\} + \{a\})$ where $a \in A$ and $b \in B$. Here's an example of a local move:



We now have a reasonable local search procedure, and we could just stop here. But there is still a lot of room for improvement in terms of the *quality* of the solutions produced. The search space includes some local optima that are quite far from the global solution. Here's one which has cost 2.



What can be done about such suboptimal solutions? We could expand the neighborhood size to allow two swaps at a time, but this particular bad instance would still stubbornly resist. Instead, let's look at some other generic schemes for improving local search procedures.

9.3.3 Dealing with local optima

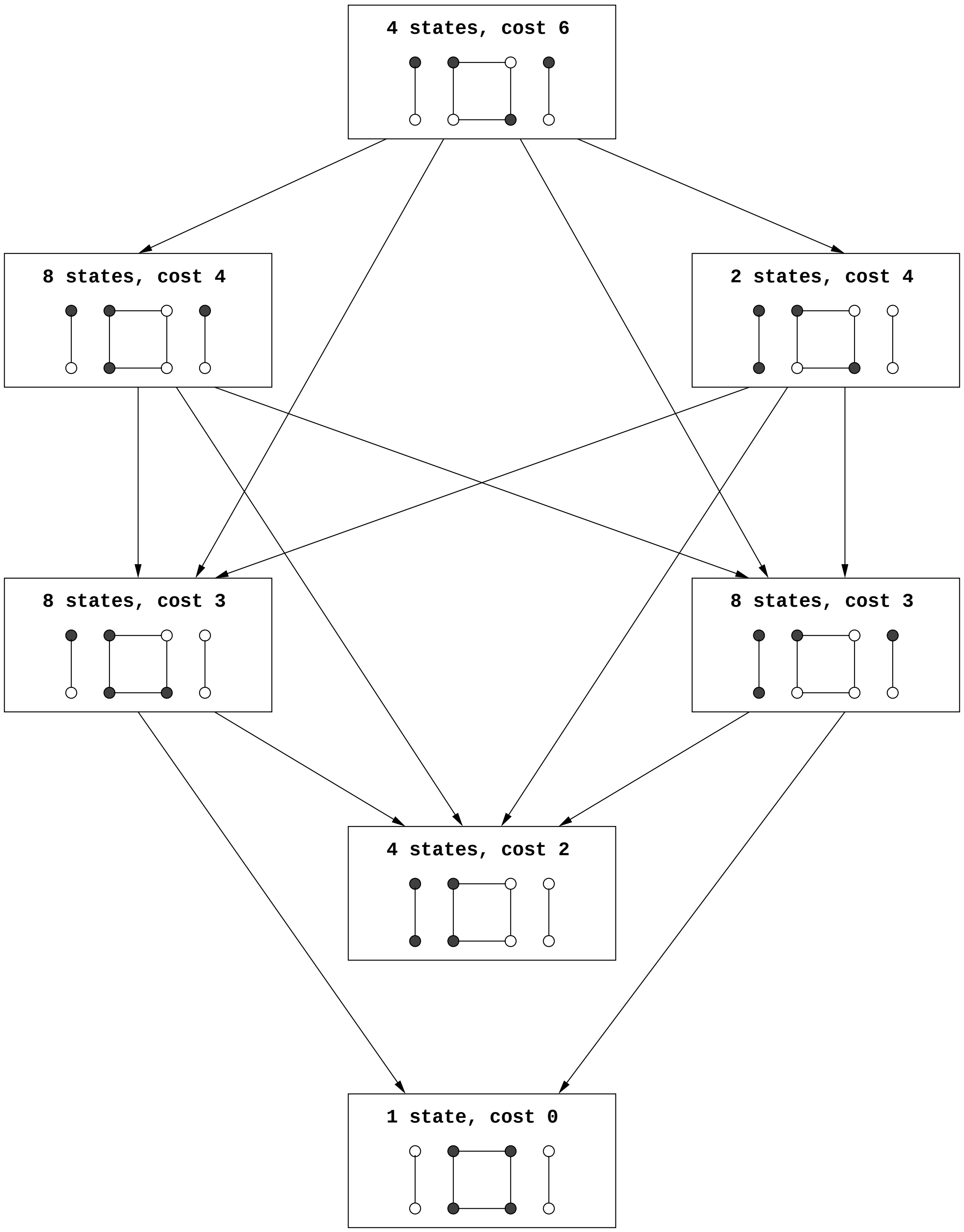
Randomization and restarts

Randomization can be an invaluable ally in local search. It is typically used in two ways: to pick a random initial solution, for instance a random graph partition; and to choose a local move when several are available.

When there are many local optima, randomization is a way of making sure that there is at least some probability of getting to the right one. The local search can then be repeated several times, with a different random seed on each invocation, and the best solution returned. If the probability of reaching a good local optimum on any given run is p, then within $O(1/p)$ runs such a solution is likely to be found (recall Exercise 1.34).

Figure 9.10 shows a small instance of graph partitioning, along with the search space of solutions. There are a total of $\binom{8}{4} = 70$ possible states, but since each of them has an identical twin in which the left and right sides of the cut are flipped, in effect there are just 35 solutions. In the figure, these are organized into seven groups for readability. There are five local optima, of which four are bad, with cost 2, and one is good, with cost 0. If local search is started at a random solution, and at each step a random neighbor of lower cost is selected, then the search is at most four times as likely to wind up in a bad solution than a good one. Thus only a small handful of repetitions is needed.

Figure 9.10 The search space for a graph with eight nodes. The space contains 35 solutions, which have been partitioned into seven groups for clarity. An example of each is shown. There are five local optima.



Simulated annealing

In the example of Figure 9.10, each run of local search has a reasonable chance of finding the global optimum. This isn't always true. As the problem size grows, the ratio of bad to good local optima often increases, sometimes to the point of being exponentially large. In such cases, simply repeating the local search a few times is ineffective.

A different avenue of attack is to occasionally allow moves that actually increase the cost, in the hope that they will pull the search out of dead ends. This would be very useful at the bad local optima of Figure 9.10, for instance. The method of *simulated annealing* redefines the local search by introducing the notion of a *temperature* T.

```
let $s$ be any starting solution
repeat
   randomly choose a solution $s'$ in the neighborhood of $s$
   if $\Delta = \text{cost}(s') - \text{cost}(s)$ is negative:
      replace $s$ by $s'$
   else:
      replace $s$ by $s'$ with probability $e^{-\Delta/T}$.
```

If T is zero, this is identical to our previous local search. But if T is large, then moves that increase the cost are occasionally accepted. What value of T should be used?

The trick is to start with T large and then gradually reduce it to zero. Thus initially, the local search can wander around quite freely, with only a mild preference for low-cost solutions. As time goes on, this preference becomes stronger, and the system mostly sticks to the lower-cost region of the search space, with occasional excursions out of it to escape local optima. Eventually, when the temperature drops further, the system converges on a solution. Figure 9.11 shows this process schematically.

Simulated annealing is inspired by the physics of crystallization. When a substance is to be crystallized, it starts in liquid state, with its particles in relatively unconstrained motion. Then it is slowly cooled, and as this happens, the particles gradually move into more regular configurations. This regularity becomes more and more pronounced until finally a crystal lattice is formed.

The benefits of simulated annealing come at a significant cost: because of the changing temperature and the initial freedom of movement, many more local moves are needed until convergence. Moreover, it is quite an art to choose a good timetable by which to decrease the temperature, called an *annealing schedule*. But in many cases where the quality of solutions improves significantly, the tradeoff is worthwhile.