# 35 Approximation Algorithms

Many problems of practical significance are NP-complete, yet they are too important to abandon merely because nobody knows how to find an optimal solution in polynomial time. Even if a problem is NP-complete, there may be hope. You have at least three options to get around NP-completeness. First, if the actual inputs are small, an algorithm with exponential running time might be fast enough. Second, you might be able to isolate important special cases that you can solve in polynomial time. Third, you can try to devise an approach to find a *near-optimal* solution in polynomial time (either in the worst case or the expected case). In practice, near-optimality is often good enough. We call an algorithm that returns near-optimal solutions an ***approximation algorithm***. This chapter presents polynomial-time approximation algorithms for several NP-complete problems.

## Performance ratios for approximation algorithms

Suppose that you are working on an optimization problem in which each potential solution has a positive cost, and you want to find a near-optimal solution. Depending on the problem, you could define an optimal solution as one with maximum possible cost or as one with minimum possible cost, which is to say that the problem might be either a maximization or a minimization problem.

We say that an algorithm for a problem has an ***approximation ratio*** of $\rho(n)$ if, for any input of size $n$, the cost $C$ of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost $C^*$ of an optimal solution:

$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \le \rho(n). \qquad (35.1)$$

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a ***$\rho(n)$-approximation algorithm***. The definitions of approximation ratio and $\rho(n)$-approximation algorithm apply to both minimization and maximization problems. For a maximization problem, $0 < C \le C^*$, and the ratio $C^*/C$ gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution. Similarly, for a minimization problem, $0 < C^* \le C$, and the ratio $C/C^*$ gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution. Because we assume that all solutions have positive cost, these ratios are always well defined. The approximation ratio of an approximation algorithm is never less than 1, since $C/C^* \le 1$ implies $C^*/C \ge 1$. Therefore, a 1-approximation algorithm[1] produces an optimal solution, and an approximation algorithm with a large approximation ratio may return a solution that is much worse than optimal.

For many problems, we know of polynomial-time approximation algorithms with small constant approximation ratios, although for other problems, the best known polynomial-time approximation algorithms have approximation ratios that grow as functions of the input size $n$. An example of such a problem is the set-cover problem presented in Section 35.3.

Some polynomial-time approximation algorithms can achieve increasingly better approximation ratios by using more and more computation time. For such problems, you can trade computation time for the quality of the approximation. An example is the subset-sum problem studied in Section 35.5. This situation is important enough to deserve a name of its own.

An ***approximation scheme*** for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\epsilon > 0$ such that for any fixed $\epsilon$, the scheme is a $(1 + \epsilon)$-approximation algorithm. We say that an approximation scheme is a ***polynomial-time approximation scheme*** if for any fixed $\epsilon > 0$, the scheme runs in time polynomial in the size $n$ of its input instance.

The running time of a polynomial-time approximation scheme can increase very rapidly as $\epsilon$ decreases. For example, the running time of a polynomial-time approximation scheme might be $O(n^{2/\epsilon})$. Ideally, if $\epsilon$ decreases by a constant factor, the running time to achieve the desired approximation should not increase by more than a constant factor (though not necessarily the same constant factor by which $\epsilon$ decreased).

We say that an approximation scheme is a ***fully polynomial-time approximation scheme*** if it is an approximation scheme and its running time is polynomial in both $1/\epsilon$ and the size $n$ of the input instance. For example, the scheme might have a running time of $O((1/\epsilon)^2 n^3)$. With such a scheme, any constant-factor decrease in $\epsilon$ comes with a corresponding constant-factor increase in the running time.

[1] When the approximation ratio is independent of $n$, we use the terms "approximation ratio of $\rho$" and "$\rho$-approximation algorithm," indicating no dependence on $n$.

## Chapter outline

The first four sections of this chapter present some examples of polynomial-time approximation algorithms for NP-complete problems, and the fifth section gives a fully polynomial-time approximation scheme. We begin in Section 35.1 with a study of the vertex-cover problem, an NP-complete minimization problem that has an approximation algorithm with an approximation ratio of 2. Section 35.2 looks at a version of the traveling-salesperson problem in which the cost function satisfies the triangle inequality and presents an approximation algorithm with an approximation ratio of 2. The section also shows that without the triangle inequality, for any constant $\rho \geq 1$, a $\rho$-approximation algorithm cannot exist unless P = NP. Section 35.3 applies a greedy method as an effective approximation algorithm for the set-covering problem, obtaining a covering whose cost is at worst a logarithmic factor larger than the optimal cost. Section 35.4 uses randomization and linear programming to develop two more approximation algorithms. The section first defines the optimization version of 3-CNF satisfiability and gives a simple randomized algorithm that produces a solution with an expected approximation ratio of $8/7$. Then Section 35.4 examines a weighted variant of the vertex-cover problem and exhibits how to use linear programming to develop a 2-approximation algorithm. Finally, Section 35.5 presents a fully polynomial-time approximation scheme for the subset-sum problem.

## 35.1 The vertex-cover problem

Section 34.5.2 defined the vertex-cover problem and proved it NP-complete. Recall that a ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v)$ is an edge of $G$, then either $u \in V'$ or $v \in V'$ (or both). The size of a vertex cover is the number of vertices in it.

The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an ***optimal vertex cover***. This problem is the optimization version of an NP-complete decision problem.

Even though nobody knows how to find an optimal vertex cover in a graph $G$ in polynomial time, there is an efficient algorithm to find a vertex cover that is near-optimal. The approximation algorithm APPROX-VERTEX-COVER on the facing page takes as input an undirected graph $G$ and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover.

Figure 35.1 illustrates how APPROX-VERTEX-COVER operates on an example graph. The variable $C$ contains the vertex cover being constructed. Line 1 initializes $C$ to the empty set. Line 2 sets $E'$ to be a copy of the edge set $G.E$ of the graph. The **while** loop of lines 3–6 repeatedly picks an edge $(u, v)$ from $E'$, adds

```
APPROX-VERTEX-COVER(G)
1  $C = \emptyset$
2  $E' = G.E$
3  while $E' \neq \emptyset$
4      let $(u, v)$ be an arbitrary edge of $E'$
5      $C = C \cup \{u, v\}$
6      remove from $E'$ edge $(u, v)$ and every edge incident on either $u$ or $v$
7  return $C$
```

its endpoints $u$ and $v$ into $C$, and deletes all edges in $E'$ that $u$ or $v$ covers. Finally, line 7 returns the vertex cover $C$. The running time of this algorithm is $O(V + E)$, using adjacency lists to represent $E'$.

***Theorem 35.1***
APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

***Proof*** We have already shown that APPROX-VERTEX-COVER runs in polynomial time.

The set $C$ of vertices that is returned by APPROX-VERTEX-COVER is a vertex cover, since the algorithm loops until every edge in $G.E$ has been covered by some vertex in $C$.

To see that APPROX-VERTEX-COVER returns a vertex cover that is at most twice the size of an optimal cover, let $A$ denote the set of edges that line 4 of APPROX-VERTEX-COVER picked. In order to cover the edges in $A$, any vertex cover—in particular, an optimal cover $C^*$—must include at least one endpoint of each edge in $A$. No two edges in $A$ share an endpoint, since once an edge is picked in line 4, all other edges that are incident on its endpoints are deleted from $E'$ in line 6. Thus, no two edges in $A$ are covered by the same vertex from $C^*$, meaning that for every vertex in $C^*$, there is at most one edge in $A$, giving the lower bound

$$|C^*| \geq |A| \tag{35.2}$$

on the size of an optimal vertex cover. Each execution of line 4 picks an edge for which neither of its endpoints is already in $C$, yielding an upper bound (an exact upper bound, in fact) on the size of the vertex cover returned:

$$|C| = 2|A| . \tag{35.3}$$

Combining equations (35.2) and (35.3) yields

$$\begin{aligned} |C| &= 2|A| \\ &\leq 2|C^*| , \end{aligned}$$

thereby proving the theorem. ■

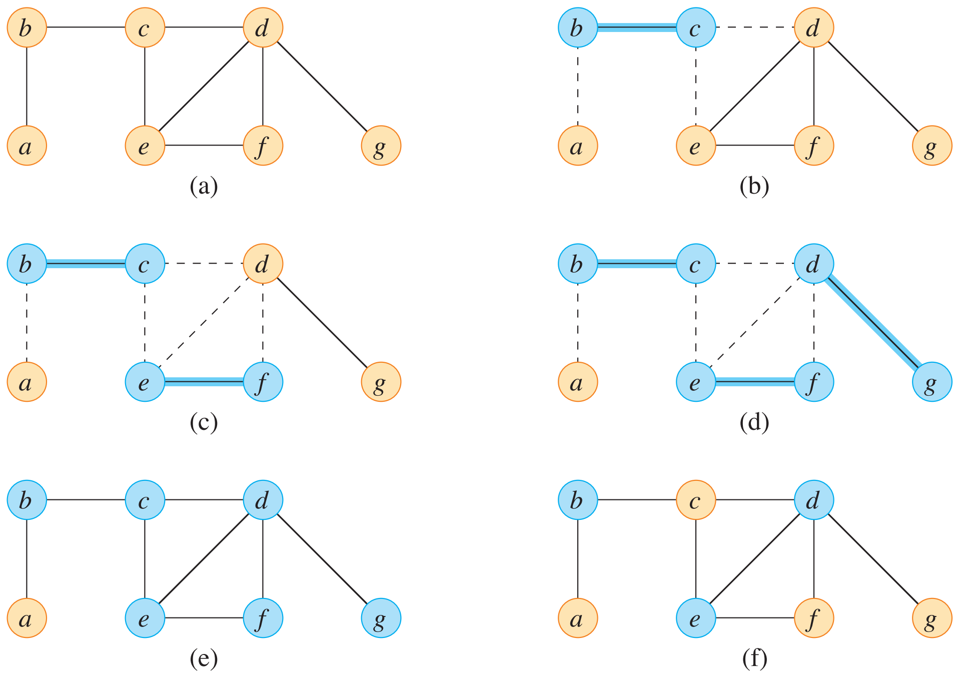

**Figure 35.1** The operation of APPROX-VERTEX-COVER. **(a)** The input graph $G$, which has 7 vertices and 8 edges. **(b)** The highlighted edge $(b, c)$ is the first edge chosen by APPROX-VERTEX-COVER. Vertices $b$ and $c$, in blue, are added to the set $C$ containing the vertex cover being created. Dashed edges $(a, b)$, $(c, e)$, and $(c, d)$ are removed since they are now covered by some vertex in $C$. **(c)** Edge $(e, f)$ is chosen, and vertices $e$ and $f$ are added to $C$. **(d)** Edge $(d, g)$ is chosen, and vertices $d$ and $g$ are added to $C$. **(e)** The set $C$, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices $b, c, d, e, f, g$. **(f)** The optimal vertex cover for this problem contains only three vertices: $b$, $d$, and $e$.

Let us reflect on this proof. At first, you might wonder how you can possibly prove that the size of the vertex cover returned by APPROX-VERTEX-COVER is at most twice the size of an optimal vertex cover, when you don't even know the size of an optimal vertex cover. Instead of requiring that you know the exact size of an optimal vertex cover, you find a lower bound on the size. As Exercise 35.1-2 asks you to show, the set $A$ of edges that line 4 of APPROX-VERTEX-COVER selects is actually a maximal matching in the graph $G$. (A ***maximal matching*** is a matching to which no edges can be added and still have a matching.) The size of a maximal matching is, as we argued in the proof of Theorem 35.1, a lower bound on the size of an optimal vertex cover. The algorithm returns a vertex cover whose size is at most twice the size of the maximal matching $A$. The approximation ratio comes from relating the size of the solution returned to the lower bound. We will use this methodology in later sections as well.

**Exercises**

***35.1-1***
Give an example of a graph for which APPROX-VERTEX-COVER always yields a suboptimal solution.

***35.1-2***
Prove that the set of edges picked in line 4 of APPROX-VERTEX-COVER forms a maximal matching in the graph $G$.

★ ***35.1-3***
Consider the following heuristic to solve the vertex-cover problem. Repeatedly select a vertex of highest degree, and remove all of its incident edges. Give an example to show that this heuristic does not provide an approximation ratio of 2. (*Hint:* Try a bipartite graph with vertices of uniform degree on the left and vertices of varying degree on the right.)

***35.1-4***
Give an efficient greedy algorithm that finds an optimal vertex cover for a tree in linear time.

***35.1-5***
The proof of Theorem 34.12 on page 1084 illustrates that the vertex-cover problem and the NP-complete clique problem are complementary in the sense that an optimal vertex cover is the complement of a maximum-size clique in the complement graph. Does this relationship imply that there is a polynomial-time approximation algorithm with a constant approximation ratio for the clique problem? Justify your answer.

## 35.2 The traveling-salesperson problem

The input to the traveling-salesperson problem, introduced in Section 34.5.4, is a complete undirected graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$. The goal is to find a hamiltonian cycle (a tour) of $G$ with minimum cost. As an extension of our notation, let $c(A)$ denote the total cost of the edges in the subset $A \subseteq E$:

$$c(A) = \sum_{(u,v) \in A} c(u, v) .$$

In many practical situations, the least costly way to go from a place $u$ to a place $w$ is to go directly, with no intermediate steps. Put another way, cutting out an intermediate stop never increases the cost. Such a cost function $c$ satisfies the ***triangle inequality***: for all vertices $u, v, w \in V$,

$$c(u, w) \le c(u, v) + c(v, w) .$$

The triangle inequality seems as though it should naturally hold, and it is automatically satisfied in several applications. For example, if the vertices of the graph are points in the plane and the cost of traveling between two vertices is the ordinary euclidean distance between them, then the triangle inequality is satisfied. Furthermore, many cost functions other than euclidean distance satisfy the triangle inequality.

As Exercise 35.2-2 shows, the traveling-salesperson problem is NP-complete even if you require the cost function to satisfy the triangle inequality. Thus, you should not expect to find a polynomial-time algorithm for solving this problem exactly. Your time would be better spent looking for good approximation algorithms.

In Section 35.2.1, we examine a 2-approximation algorithm for the traveling-salesperson problem with the triangle inequality. In Section 35.2.2, we show that without the triangle inequality, a polynomial-time approximation algorithm with a constant approximation ratio does not exist unless P = NP.

## 35.2.1 The traveling-salesperson problem with the triangle inequality

Applying the methodology of the previous section, start by computing a structure —a minimum spanning tree—whose weight gives a lower bound on the length of an optimal traveling-salesperson tour. Then use the minimum spanning tree to create a tour whose cost is no more than twice that of the minimum spanning tree's weight, as long as the cost function satisfies the triangle inequality. The procedure APPROX-TSP-TOUR on the next page implements this approach, calling the minimum-spanning-tree algorithm MST-PRIM on page 596 as a subroutine. The parameter $G$ is a complete undirected graph, and the cost function $c$ satisfies the triangle inequality.

Recall from Section 12.1 that a preorder tree walk recursively visits every vertex in the tree, listing a vertex when it is first encountered, before visiting any of its children.

Figure 35.2 illustrates the operation of APPROX-TSP-TOUR. Part (a) of the figure shows a complete undirected graph, and part (b) shows the minimum spanning tree $T$ grown from root vertex $a$ by MST-PRIM. Part (c) shows how a preorder walk of $T$ visits the vertices, and part (d) displays the corresponding tour, which is the tour returned by APPROX-TSP-TOUR. Part (e) displays an optimal tour, which is about 23% shorter.

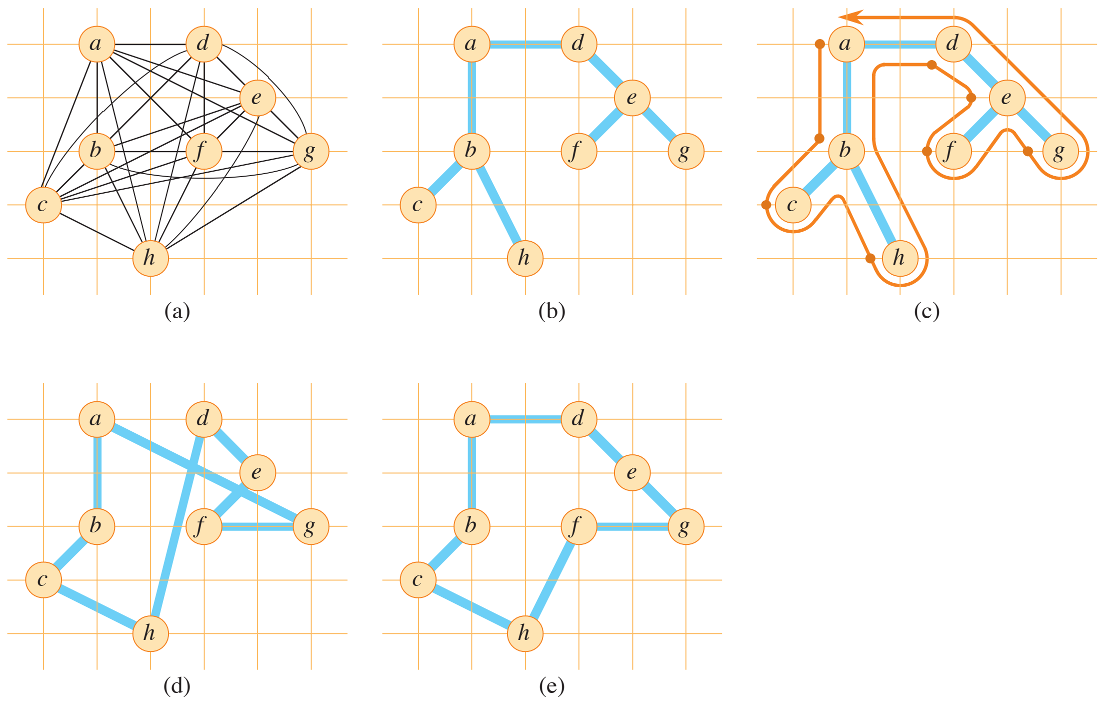

**Figure 35.2** The operation of APPROX-TSP-TOUR. **(a)** A complete undirected graph. Vertices lie on intersections of integer grid lines. For example, $f$ is one unit to the right and two units up from $h$. The cost function between two points is the ordinary euclidean distance. **(b)** A minimum spanning tree $T$ of the complete graph, as computed by MST-PRIM. Vertex $a$ is the root vertex. Only edges in the minimum spanning tree are shown. The vertices happen to be labeled in such a way that they are added to the main tree by MST-PRIM in alphabetical order. **(c)** A walk of $T$, starting at $a$. A full walk of the tree visits the vertices in the order $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. A preorder walk of $T$ lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering $a, b, c, h, d, e, f, g$. **(d)** A tour obtained by visiting the vertices in the order given by the preorder walk, which is the tour $H$ returned by APPROX-TSP-TOUR. Its total cost is approximately 19.074. **(e)** An optimal tour $H^*$ for the original complete graph. Its total cost is approximately 14.715.

```
APPROX-TSP-TOUR($G, c$)
1  select a vertex $r \in G.V$ to be a "root" vertex
2  compute a minimum spanning tree $T$ for $G$ from root $r$
       using MST-PRIM($G, c, r$)
3  let $H$ be a list of vertices, ordered according to when they are first visited
       in a preorder tree walk of $T$
4  return the hamiltonian cycle $H$
```

By Exercise 21.2-2, even with a simple implementation of MST-PRIM, the running time of APPROX-TSP-TOUR is $\Theta(V^2)$. We now show that if the cost function for an instance of the traveling-salesperson problem satisfies the triangle inequality, then APPROX-TSP-TOUR returns a tour whose cost is at most twice the cost of an optimal tour.

***Theorem 35.2***
When the triangle inequality holds, APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesperson problem.

***Proof*** We have already seen that APPROX-TSP-TOUR runs in polynomial time.

Let $H^*$ denote an optimal tour for the given set of vertices. Deleting any edge from a tour yields a spanning tree, and each edge cost is nonnegative. Therefore, the weight of the minimum spanning tree $T$ computed in line 2 of APPROX-TSP-TOUR provides a lower bound on the cost of an optimal tour:

$$c(T) \le c(H^*) . \tag{35.4}$$

A ***full walk*** of $T$ lists the vertices when they are first visited and also whenever they are returned to after a visit to a subtree. Let's call this full walk $W$. The full walk of our example gives the order

$$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a .$$

Since the full walk traverses every edge of $T$ exactly twice, by extending the definition of the cost $c$ in the natural manner to handle multisets of edges, we have

$$c(W) = 2c(T) . \tag{35.5}$$

Inequality (35.4) and equation (35.5) imply that

$$c(W) \le 2c(H^*) , \tag{35.6}$$

and so the cost of $W$ is within a factor of 2 of the cost of an optimal tour.

Of course, the full walk $W$ is not a tour, since it visits some vertices more than once. By the triangle inequality, however, deleting a visit to any vertex from $W$ does not increase the cost. (When a vertex $v$ is deleted from $W$ between visits to $u$ and $w$, the resulting ordering specifies going directly from $u$ to $w$.) Repeatedly apply this operation on each visit to a vertex after the first time it's visited in $W$, so that $W$ is left with only the first visit to each vertex. In our example, this process leaves the ordering

$$a, b, c, h, d, e, f, g .$$

This ordering is the same as that obtained by a preorder walk of the tree $T$. Let $H$ be the cycle corresponding to this preorder walk. It is a hamiltonian cycle, since every vertex is visited exactly once, and in fact it is the cycle computed by APPROX-TSP-TOUR. Since $H$ is obtained by deleting vertices from the full walk $W$, we have

$$c(H) \le c(W) . \tag{35.7}$$

Combining inequalities (35.6) and (35.7) gives $c(H) \le 2c(H^*)$, which completes the proof. ∎

Despite the small approximation ratio provided by Theorem 35.2, APPROX-TSP-TOUR is usually not the best practical choice for this problem. There are other approximation algorithms that typically perform much better in practice. (See the references at the end of this chapter.)

### 35.2.2 The general traveling-salesperson problem

When the cost function $c$ does not satisfy the triangle inequality, there is no way to find good approximate tours in polynomial time unless $\text{P} = \text{NP}$.

***Theorem 35.3***
If $\text{P} \ne \text{NP}$, then for any constant $\rho \ge 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general traveling-salesperson problem.

***Proof*** The proof is by contradiction. Suppose to the contrary that for some number $\rho \ge 1$, there is a polynomial-time approximation algorithm $A$ with approximation ratio $\rho$. Without loss of generality, assume that $\rho$ is an integer, by rounding it up if necessary. We will show how to use $A$ to solve instances of the hamiltonian-cycle problem (defined in Section 34.2) in polynomial time. Since Theorem 34.13 on page 1085 says that the hamiltonian-cycle problem is NP-complete, Theorem 34.4 on page 1063 implies that if it has a polynomial-time algorithm, then $\text{P} = \text{NP}$.

Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem. We will show how to determine efficiently whether $G$ contains a hamiltonian cycle by making use of the hypothesized approximation algorithm $A$. Convert $G$ into an instance of the traveling-salesperson problem as follows. Let $G' = (V, E')$ be the complete graph on $V$, that is,

$$E' = \{(u, v) : u, v \in V \text{ and } u \ne v\} .$$

Assign an integer cost to each edge in $E'$ as follows:

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E , \\ \rho\,|V| + 1 & \text{otherwise} . \end{cases}$$

Given a representation of $G$, it takes time polynomial in $|V|$ and $|E|$ to create representations of $G'$ and $c$.

Now consider the traveling-salesperson problem $(G', c)$. If the original graph $G$ has a hamiltonian cycle $H$, then the cost function $c$ assigns to each edge of $H$ a cost of 1, and so $(G', c)$ contains a tour of cost $|V|$. On the other hand, if $G$ does not contain a hamiltonian cycle, then any tour of $G'$ must use some edge not in $E$. But any tour that uses an edge not in $E$ has a cost of at least

$$\begin{aligned}(\rho\,|V| + 1) + (|V| - 1) &= \rho\,|V| + |V| \\ &> \rho\,|V| \ .\end{aligned}$$

Because edges not in $G$ are so costly, there is a gap of at least $\rho\,|V|$ between the cost of a tour that is a hamiltonian cycle in $G$ (cost $|V|$) and the cost of any other tour (cost at least $\rho\,|V| + |V|$). Therefore, the cost of a tour that is not a hamiltonian cycle in $G$ is at least a factor of $\rho + 1$ greater than the cost of a tour that is a hamiltonian cycle in $G$.

What happens upon applying the approximation algorithm $A$ to the traveling-salesperson problem $(G', c)$? Because $A$ is guaranteed to return a tour of cost no more than $\rho$ times the cost of an optimal tour, if $G$ contains a hamiltonian cycle, then $A$ must return it. If $G$ has no hamiltonian cycle, then $A$ returns a tour of cost more than $\rho\,|V|$. Therefore, using $A$ solves the hamiltonian-cycle problem in polynomial time. ■

The proof of Theorem 35.3 serves as an example of a general technique to prove that no good approximation algorithm exists for a particular problem. Given an NP-hard decision problem $X$, produce in polynomial time a minimization problem $Y$ such that "yes" instances of $X$ correspond to instances of $Y$ with value at most $k$ (for some $k$), but that "no" instances of $X$ correspond to instances of $Y$ with value greater than $\rho k$. This technique shows that, unless P = NP, there is no polynomial-time $\rho$-approximation algorithm for problem $Y$.

## Exercises

***35.2-1***

Let $G = (V, E)$ be a complete undirected graph containing at least 3 vertices, and let $c$ be a cost function that satisfies the triangle inequality. Prove that $c(u, v) \geq 0$ for all $u, v \in V$.

***35.2-2***

Show how in polynomial time to transform one instance of the traveling-salesperson problem into another instance whose cost function satisfies the triangle inequality. The two instances must have the same set of optimal tours. Explain why such a polynomial-time transformation does not contradict Theorem 35.3, assuming that P $\neq$ NP.

***35.2-3***

Consider the following ***closest-point heuristic*** for building an approximate traveling-salesperson tour whose cost function satisfies the triangle inequality. Begin with a trivial cycle consisting of a single arbitrarily chosen vertex. At each step, identify the vertex $u$ that is not on the cycle but whose distance to any vertex on the cycle is minimum. Suppose that the vertex on the cycle that is nearest $u$ is vertex $v$. Extend the cycle to include $u$ by inserting $u$ just after $v$. Repeat until all vertices are on the cycle. Prove that this heuristic returns a tour whose total cost is not more than twice the cost of an optimal tour.

***35.2-4***

A solution to the ***bottleneck traveling-salesperson problem*** is the hamiltonian cycle that minimizes the cost of the most costly edge in the cycle. Assuming that the cost function satisfies the triangle inequality, show that there exists a polynomial-time approximation algorithm with approximation ratio 3 for this problem. (*Hint:* Show recursively how to visit all the nodes in a bottleneck spanning tree, as discussed in Problem 21-4 on page 601, exactly once by taking a full walk of the tree and skipping nodes, but without skipping more than two consecutive intermediate nodes. Show that the costliest edge in a bottleneck spanning tree has a cost bounded from above by the cost of the costliest edge in a bottleneck hamiltonian cycle.)

***35.2-5***

Suppose that the vertices for an instance of the traveling-salesperson problem are points in the plane and that the cost $c(u, v)$ is the euclidean distance between points $u$ and $v$. Show that an optimal tour never crosses itself.

***35.2-6***

Adapt the proof of Theorem 35.3 to show that for any constant $c \geq 0$, there is no polynomial-time approximation algorithm with approximation ratio $|V|^c$ for the general traveling-salesperson problem.

## 35.3 The set-covering problem

The set-covering problem is an optimization problem that models many problems that require resources to be allocated. Its corresponding decision problem generalizes the NP-complete vertex-cover problem and is therefore also NP-hard. The approximation algorithm developed to handle the vertex-cover problem doesn't apply here, however. Instead, this section investigates a simple greedy heuristic with a logarithmic approximation ratio. That is, as the size of the instance gets larger, the size of the approximate solution may grow, relative to the size of an optimal solution. Because the logarithm function grows rather slowly, however, this approximation algorithm may nonetheless give useful results.

An instance $(X, \mathcal{F})$ of the ***set-covering problem*** consists of a finite set $X$ and a family $\mathcal{F}$ of subsets of $X$, such that every element of $X$ belongs to at least one subset in $\mathcal{F}$:

$$X = \bigcup_{S \in \mathcal{F}} S .$$

We say that a subfamily $\mathcal{C} \subseteq \mathcal{F}$ ***covers*** a set of elements $U$ if

$$U \subseteq \bigcup_{S \in \mathcal{C}} S .$$

The problem is to find a minimum-size subfamily $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of $X$:

$$X = \bigcup_{S \in \mathcal{C}} S .$$

Figure 35.3 illustrates the set-covering problem. The size of $\mathcal{C}$ is the number of sets it contains, rather than the number of individual elements in these sets, since every subfamily $\mathcal{C}$ that covers $X$ must contain all $|X|$ individual elements. In Figure 35.3, the minimum set cover has size 3.

The set-covering problem abstracts many commonly arising combinatorial problems. As a simple example, suppose that $X$ represents a set of skills that are needed to solve a problem and that you have a given set of people available to work on the problem. You wish to form a committee, containing as few people as possible, such that for every requisite skill in $X$, at least one member of the committee has that skill. The decision version of the set-covering problem asks whether a covering exists with size at most $k$, where $k$ is an additional parameter specified in the problem instance. The decision version of the problem is NP-complete, as Exercise 35.3-2 asks you to show.

## A greedy approximation algorithm

The greedy method in the procedure GREEDY-SET-COVER on the facing page works by picking, at each stage, the set $S$ that covers the greatest number of remaining elements that are uncovered. In the example of Figure 35.3, GREEDY-SET-COVER adds to $\mathcal{C}$, in order, the sets $S_1$, $S_4$, and $S_5$, followed by either $S_3$ or $S_6$.

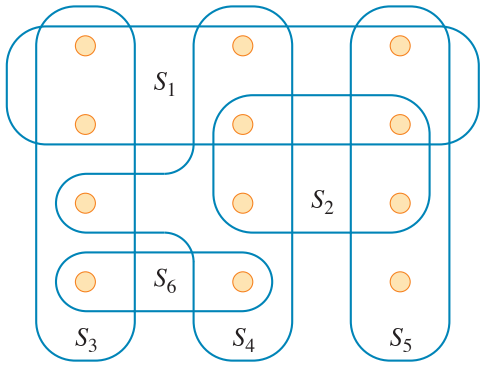

**Figure 35.3** An instance $(X, \mathcal{F})$ of the set-covering problem, where $X$ consists of the 12 tan points and $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. Each set $S_i \in \mathcal{F}$ is outlined in blue. A minimum-size set cover is $\mathcal{C} = \{S_3, S_4, S_5\}$, with size 3. The greedy algorithm produces a cover of size 4 by selecting either the sets $S_1$, $S_4$, $S_5$, and $S_3$ or the sets $S_1$, $S_4$, $S_5$, and $S_6$, in order.

```
GREEDY-SET-COVER$(X, \mathcal{F})$
1  $U_0 = X$
2  $\mathcal{C} = \emptyset$
3  $i = 0$
4  while $U_i \neq \emptyset$
5      select $S \in \mathcal{F}$ that maximizes $|S \cap U_i|$
6      $U_{i+1} = U_i - S$
7      $\mathcal{C} = \mathcal{C} \cup \{S\}$
8      $i = i + 1$
9  return $\mathcal{C}$
```

The greedy algorithm works as follows. At the start of each iteration, $U_i$ is a subset of $X$ containing the remaining uncovered elements, with the initial subset $U_0$ containing all the elements in $X$. The set $\mathcal{C}$ contains the subfamily being constructed. Line 5 is the greedy decision-making step, choosing a subset $S$ that covers as many uncovered elements as possible (breaking ties arbitrarily). After $S$ is selected, line 6 updates the set of remaining uncovered elements, denoting it by $U_{i+1}$, and line 7 places $S$ into $\mathcal{C}$. When the algorithm terminates, $\mathcal{C}$ is a subfamily of $\mathcal{F}$ that covers $X$.

## Analysis

We now show that the greedy algorithm returns a set cover that is not too much larger than an optimal set cover.

***Theorem 35.4***
The procedure GREEDY-SET-COVER run on a set $X$ and family of subsets $\mathcal{F}$ is a polynomial-time $O(\lg X)$-approximation algorithm.

***Proof*** Let's first show that the algorithm runs in time that is polynomial in $|X|$ and $|\mathcal{F}|$. The number of iterations of the loop in lines 4–7 is bounded above by $\min\{|X|, |\mathcal{F}|\} = O(|X| + |\mathcal{F}|)$. The loop body can be implemented to run in $O(|X| \cdot |\mathcal{F}|)$ time. Thus the algorithm runs in $O(|X| \cdot |\mathcal{F}| \cdot (|X| + |\mathcal{F}|))$ time, which is polynomial in the input size. (Exercise 35.3-3 asks for a linear-time algorithm.)

To prove the approximation bound, let $\mathcal{C}^*$ be an optimal set cover for the original instance $(X, \mathcal{F})$, and let $k = |\mathcal{C}^*|$. Since $\mathcal{C}^*$ is also a set cover of each subset $U_i$ of $X$ constructed by the algorithm, we know that any subset $U_i$ constructed by the algorithm can be covered by $k$ sets. Therefore, if $(U_i, \mathcal{F})$ is an instance of the set-covering problem, its optimal set cover has size at most $k$.

If an optimal set cover for an instance $(U_i, \mathcal{F})$ has size at most $k$, at least one of the sets in $\mathcal{C}$ covers at least $|U_i|/k$ new elements. Thus, line 5 of GREEDY-SET-COVER, which chooses a set with the maximum number of uncovered elements, must choose a set in which the number of newly covered elements is at least $|U_i|/k$. These elements are removed when constructing $U_{i+1}$, giving

$$\begin{aligned} |U_{i+1}| &\le |U_i| - |U_i|/k \\ &= |U_i|(1 - 1/k) . \end{aligned} \tag{35.8}$$

Iterating inequality (35.8) gives

$$\begin{aligned} |U_0| &= |X| , \\ |U_1| &\le |U_0|(1 - 1/k) , \\ |U_2| &\le |U_1|(1 - 1/k) = |U|(1 - 1/k)^2 , \end{aligned}$$

and in general

$$|U_i| \le |U_0|(1 - 1/k)^i = |X|(1 - 1/k)^i . \tag{35.9}$$

The algorithm stops when $U_i = \emptyset$, which means that $|U_i| < 1$. Thus an upper bound on the number of iterations of the algorithm is the smallest value of $i$ for which $|U_i| < 1$.

Since $1 + x \le e^x$ for all real $x$ (see inequality (3.14) on page 66), by letting $x = -1/k$, we have $1 - 1/k \le e^{-1/k}$, so that $(1 - 1/k)^k \le (e^{-1/k})^k = 1/e$. Denoting the number $i$ of iterations by $ck$ for some nonnegative integer $c$, we want $c$ such that

$$|X|(1 - 1/k)^{ck} \le |X|e^{-c} < 1 . \tag{35.10}$$

Multiplying both sides by $e^c$ and then taking the natural logarithm of both sides gives $c \ge \ln|X|$, so we can choose for $c$ any integer that is at least $\ln|X|$. We choose $c = \lceil \ln |X| \rceil$. Since $i = ck$ is an upper bound on the number of iterations, which equals the size of $\mathcal{C}$, and $k = |\mathcal{C}^*|$, we have $|\mathcal{C}| \le i = ck = c\,|\mathcal{C}^*| = |\mathcal{C}^*|\,\lceil \ln |X| \rceil$, and the theorem follows. ■

**Exercises**

***35.3-1***
Consider each of the following words as a set of letters: {`arid`, `dash`, `drain`, `heard`, `lost`, `nose`, `shun`, `slate`, `snare`, `thread`}. Show which set cover GREEDY-SET-COVER produces when you break ties in favor of the word that appears first in the dictionary.

***35.3-2***
Show that the decision version of the set-covering problem is NP-complete by reducing the vertex-cover problem to it.

***35.3-3***
Show how to implement GREEDY-SET-COVER to run in $O\left(\sum_{S \in \mathcal{F}} |S|\right)$ time.

***35.3-4***
The proof of Theorem 35.4 says that when GREEDY-SET-COVER, run on the instance $(X, \mathcal{F})$, returns the subfamily $\mathcal{C}$, then $|\mathcal{C}| \le |\mathcal{C}^*|\,\lceil \ln X \rceil$. Show that the following weaker bound is trivially true:

$$|\mathcal{C}| \le |\mathcal{C}^*| \max\{|S| : S \in \mathcal{F}\} .$$

***35.3-5***
GREEDY-SET-COVER can return a number of different solutions, depending on how it breaks ties in line 5. Give a procedure BAD-SET-COVER-INSTANCE$(n)$ that returns an $n$-element instance of the set-covering problem for which, depending on how line 5 breaks ties, GREEDY-SET-COVER can return a number of different solutions that is exponential in $n$.

## 35.4 Randomization and linear programming

This section studies two useful techniques for designing approximation algorithms: randomization and linear programming. It starts with a simple randomized algorithm for an optimization version of 3-CNF satisfiability, and then it shows how to design an approximation algorithm for a weighted version of the vertex-cover problem based on linear programming. This section only scratches the surface of these two powerful techniques. The chapter notes give references for further study of these areas.

## A randomized approximation algorithm for MAX-3-CNF satisfiability

Just as some randomized algorithms compute exact solutions, some randomized algorithms compute approximate solutions. We say that a randomized algorithm for a problem has an ***approximation ratio*** of $\rho(n)$ if, for any input of size $n$, the *expected* cost $C$ of the solution produced by the randomized algorithm is within a factor of $\rho(n)$ of the cost $C^*$ of an optimal solution:

$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \le \rho(n). \tag{35.11}$$

We call a randomized algorithm that achieves an approximation ratio of $\rho(n)$ a ***randomized $\rho(n)$-approximation algorithm.*** In other words, a randomized approximation algorithm is like a deterministic approximation algorithm, except that the approximation ratio is for an expected cost.

A particular instance of 3-CNF satisfiability, as defined in Section 34.4, may or may not be satisfiable. In order to be satisfiable, there must exist an assignment of the variables so that every clause evaluates to 1. If an instance is not satisfiable, you might instead want to know how "close" to satisfiable it is, that is, find an assignment of the variables that satisfies as many clauses as possible. We call the resulting maximization problem ***MAX-3-CNF satisfiability***. The input to MAX-3-CNF satisfiability is the same as for 3-CNF satisfiability, and the goal is to return an assignment of the variables that maximizes the number of clauses evaluating to 1. You might be surprised that randomly setting each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ yields a randomized 8/7-approximation algorithm, but we're about to see why. Recall that the definition of 3-CNF satisfiability from Section 34.4 requires each clause to consist of exactly three distinct literals. We now further assume that no clause contains both a variable and its negation. Exercise 35.4-1 asks you to remove this last assumption.

***Theorem 35.5***
Given an instance of MAX-3-CNF satisfiability with $n$ variables $x_1, x_2, \ldots, x_n$ and $m$ clauses, the randomized algorithm that independently sets each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ is a randomized 8/7-approximation algorithm.

***Proof*** Suppose that each variable is independently set to 1 with probability $1/2$ and to 0 with probability $1/2$. Define, for $i = 1, 2, \ldots, m$, the indicator random variable

$$Y_i = \mathrm{I}\{\text{clause } i \text{ is satisfied}\} ,$$

so that $Y_i = 1$ as long as at least one of the literals in the $i$th clause is set to 1. Since no literal appears more than once in the same clause, and since we assume that no variable and its negation appear in the same clause, the settings of the three literals in each clause are independent. A clause is not satisfied only if all three of its literals are set to 0, and so $\Pr\{\text{clause } i \text{ is not satisfied}\} = (1/2)^3 = 1/8$. Thus, we have $\Pr\{\text{clause } i \text{ is satisfied}\} = 1 - 1/8 = 7/8$, and Lemma 5.1 on page 130 gives $\mathrm{E}[Y_i] = 7/8$. Let $Y$ be the number of satisfied clauses overall, so that $Y = Y_1 + Y_2 + \cdots + Y_m$. Then, we have

$$\begin{aligned}
\mathrm{E}[Y] &= \mathrm{E}\left[\sum_{i=1}^{m} Y_i\right] \\
&= \sum_{i=1}^{m} \mathrm{E}[Y_i] \qquad \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{m} 7/8 \\
&= 7m/8 .
\end{aligned}$$

Since $m$ is an upper bound on the number of satisfied clauses, the approximation ratio is at most $m/(7m/8) = 8/7$. ■

## Approximating weighted vertex cover using linear programming

The ***minimum-weight vertex-cover problem*** takes as input an undirected graph $G = (V, E)$ in which each vertex $v \in V$ has an associated positive weight $w(v)$. The weight $w(V')$ of a vertex cover $V' \subseteq V$ is the sum of the weights of its vertices: $w(V') = \sum_{v \in V'} w(v)$. The goal is to find a vertex cover of minimum weight.

The approximation algorithm for unweighted vertex cover from Section 35.1 won't work here, because the solution it returns could be far from optimal for the weighted problem. Instead, we'll first compute a lower bound on the weight of the minimum-weight vertex cover, by using a linear program. Then we'll "round" this solution and use it to obtain a vertex cover.

Start by associating a variable $x(v)$ with each vertex $v \in V$, and require that $x(v)$ equals either 0 or 1 for each $v \in V$. The vertex cover includes $v$ if and only if $x(v) = 1$. Then the constraint that for any edge $(u, v)$, at least one of $u$ and $v$ must belong to the vertex cover can be expressed as $x(u) + x(v) \geq 1$. This view gives rise to the following ***0-1 integer program*** for finding a minimum-weight vertex cover:

$$\text{minimize} \quad \sum_{v \in V} w(v)\, x(v) \tag{35.12}$$

subject to

$$x(u) + x(v) \geq 1 \quad \text{for each } (u, v) \in E \tag{35.13}$$
$$x(v) \in \{0, 1\} \quad \text{for each } v \in V\,. \tag{35.14}$$

In the special case in which all the weights $w(v)$ equal 1, this formulation is the optimization version of the NP-hard vertex-cover problem. Let's remove the constraint that $x(v) \in \{0, 1\}$ and replace it by $0 \leq x(v) \leq 1$, resulting in the following linear program:

$$\text{minimize} \quad \sum_{v \in V} w(v)\, x(v) \tag{35.15}$$

subject to

$$x(u) + x(v) \geq 1 \quad \text{for each } (u, v) \in E \tag{35.16}$$
$$x(v) \leq 1 \quad \text{for each } v \in V \tag{35.17}$$
$$x(v) \geq 0 \quad \text{for each } v \in V\,. \tag{35.18}$$

We refer to this linear program as the ***linear-programming relaxation***. Any feasible solution to the 0-1 integer program in lines (35.12)–(35.14) is also a feasible solution to its linear-programming relaxation in lines (35.15)–(35.18). Therefore, the value of an optimal solution to the linear-programming relaxation provides a lower bound on the value of an optimal solution to the 0-1 integer program, and hence a lower bound on the optimal weight in the minimum-weight vertex-cover problem.

The procedure APPROX-MIN-WEIGHT-VC on the facing page starts with a solution to the linear-programming relaxation and uses it to construct an approximate solution to the minimum-weight vertex-cover problem. The procedure works as follows. Line 1 initializes the vertex cover to be empty. Line 2 formulates the linear-programming relaxation in lines (35.15)–(35.18) and then solves this linear program. An optimal solution gives each vertex $v$ an associated value $\bar{x}(v)$, where $0 \leq \bar{x}(v) \leq 1$. The procedure uses this value to guide the choice of which vertices to add to the vertex cover $C$ in lines 3–5: the vertex cover $C$ includes vertex $v$ if and only if $\bar{x}(v) \geq 1/2$. In effect, the procedure "rounds" each fractional variable in the solution to the linear-programming relaxation to either 0 or 1 in order to obtain a solution to the 0-1 integer program in lines (35.12)–(35.14). Finally, line 6 returns the vertex cover $C$.

***Theorem 35.6***

Algorithm APPROX-MIN-WEIGHT-VC is a polynomial-time2-approximation algorithm for the minimum-weight vertex-cover problem.

```
APPROX-MIN-WEIGHT-VC$(G, w)$
1  $C = \emptyset$
2  compute $\bar{x}$, an optimal solution to the linear-programming relaxation
       in lines (35.15)–(35.18)
3  for each vertex $v \in V$
4      if $\bar{x}(v) \geq 1/2$
5          $C = C \cup \{v\}$
6  return $C$
```

***Proof*** Because there is a polynomial-time algorithm to solve the linear program in line 2, and because the **for** loop of lines 3–5 runs in polynomial time, APPROX-MIN-WEIGHT-VC is a polynomial-time algorithm.

It remains to show that APPROX-MIN-WEIGHT-VC is a 2-approximation algorithm. Let $C^*$ be an optimal solution to the minimum-weight vertex-cover problem, and let $z^*$ be the value of an optimal solution to the linear-programming relaxation in lines (35.15)–(35.18). Since an optimal vertex cover is a feasible solution to the linear-programming relaxation, $z^*$ must be a lower bound on $w(C^*)$, that is,

$$z^* \leq w(C^*) . \tag{35.19}$$

Next, we claim that rounding the fractional values of the variables $\bar{x}(v)$ in lines 3–5 produces a set $C$ that is a vertex cover and satisfies $w(C) \leq 2z^*$. To see that $C$ is a vertex cover, consider any edge $(u, v) \in E$. By constraint (35.16), we know that $x(u) + x(v) \geq 1$, which implies that at least one of $\bar{x}(u)$ and $\bar{x}(v)$ is at least $1/2$. Therefore, at least one of $u$ and $v$ is included in the vertex cover, and so every edge is covered.

Now we consider the weight of the cover. We have

$$\begin{aligned} z^* &= \sum_{v \in V} w(v)\,\bar{x}(v) \\ &\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v)\,\bar{x}(v) \\ &\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} \\ &= \sum_{v \in C} w(v) \cdot \frac{1}{2} \\ &= \frac{1}{2} \sum_{v \in C} w(v) \\ &= \frac{1}{2} w(C) . \end{aligned} \tag{35.20}$$

Combining inequalities (35.19) and (35.20) gives

$$w(C) \le 2z^* \le 2w(C^*) ,$$

and hence APPROX-MIN-WEIGHT-VC is a 2-approximation algorithm. ■

**Exercises**

***35.4-1***
Show that even if a clause is allowed to contain both a variable and its negation, randomly setting each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ still yields a randomized $8/7$-approximation algorithm.

***35.4-2***
The ***MAX-CNF satisfiability problem*** is like the MAX-3-CNF satisfiability problem, except that it does not restrict each clause to have exactly three literals. Give a randomized 2-approximation algorithm for the MAX-CNF satisfiability problem.

***35.4-3***
In the MAX-CUT problem, the input is an unweighted undirected graph $G = (V, E)$. We define a cut $(S, V - S)$ as in Chapter 21 and the ***weight*** of a cut as the number of edges crossing the cut. The goal is to find a cut of maximum weight. Suppose that each vertex $v$ is randomly and independently placed into $S$ with probability $1/2$ and into $V - S$ with probability $1/2$. Show that this algorithm is a randomized 2-approximation algorithm.

***35.4-4***
Show that the constraints in line (35.17) are redundant in the sense that removing them from the linear-programming relaxation in lines (35.15)–(35.18) yields a linear program for which any optimal solution $x$ must satisfy $x(v) \le 1$ for each $v \in V$.

## 35.5 The subset-sum problem

Recall from Section 34.5.5 that an instance of the subset-sum problem is given by a pair $(S, t)$, where $S$ is a set $\{x_1, x_2, \ldots, x_n\}$ of positive integers and $t$ is a positive integer. This decision problem asks whether there exists a subset of $S$ that adds up exactly to the target value $t$. As we saw in Section 34.5.5, this problem is NP-complete.

The optimization problem associated with this decision problem arises in practical applications. The optimization problem seeks a subset of $\{x_1, x_2, \ldots, x_n\}$

whose sum is as large as possible but not larger than $t$. For example, consider a truck that can carry no more than $t$ pounds, which is to be loaded with up to $n$ different boxes, the $i$th of which weighs $x_i$ pounds. How heavy a load can the truck take without exceeding the $t$-pound weight limit?

We start this section with an exponential-time algorithm to compute the optimal value for this optimization problem. Then we show how to modify the algorithm so that it becomes a fully polynomial-time approximation scheme. (Recall that a fully polynomial-time approximation scheme has a running time that is polynomial in $1/\epsilon$ as well as in the size of the input.)

## An exponential-time exact algorithm

Suppose that you compute, for each subset $S'$ of $S$, the sum of the elements in $S'$, and then you select, among the subsets whose sum does not exceed $t$, the one whose sum is closest to $t$. This algorithm returns the optimal solution, but it might take exponential time. To implement this algorithm, you can use an iterative procedure that, in iteration $i$, computes the sums of all subsets of $\{x_1, x_2, \ldots, x_i\}$, using as a starting point the sums of all subsets of $\{x_1, x_2, \ldots, x_{i-1}\}$. In doing so, you would realize that once a particular subset $S'$ has a sum exceeding $t$, there is no reason to maintain it, since no superset of $S'$ can be an optimal solution. Let's see how to implement this strategy.

The procedure EXACT-SUBSET-SUM takes an input set $S = \{x_1, x_2, \ldots, x_n\}$, the size $n = |S|$, and a target value $t$. This procedure iteratively computes $L_i$, the list of sums of all subsets of $\{x_1, \ldots, x_i\}$ that do not exceed $t$, and then it returns the maximum value in $L_n$.

If $L$ is a list of positive integers and $x$ is another positive integer, then let $L + x$ denote the list of integers derived from $L$ by increasing each element of $L$ by $x$. For example, if $L = \langle 1, 2, 3, 5, 9 \rangle$, then $L + 2 = \langle 3, 4, 5, 7, 11 \rangle$. This notation extends to sets, so that

$$S + x = \{s + x : s \in S\} .$$

```
EXACT-SUBSET-SUM(S, n, t)
1  $L_0 = \langle 0 \rangle$
2  for $i = 1$ to $n$
3      $L_i$ = MERGE-LISTS($L_{i-1}, L_{i-1} + x_i$)
4      remove from $L_i$ every element that is greater than $t$
5  return the largest element in $L_n$
```

EXACT-SUBSET-SUM invokes an auxiliary procedure MERGE-LISTS$(L, L')$, which returns the sorted list that is the merge of its two sorted input lists $L$ and $L'$, with duplicate values removed. Like the MERGE procedure we used in merge sort on page 36, MERGE-LISTS runs in $O(|L| + |L'|)$ time. We omit the pseudocode for MERGE-LISTS.

To see how EXACT-SUBSET-SUM works, let $P_i$ denote the set of values obtained by selecting each (possibly empty) subset of $\{x_1, x_2, \ldots, x_i\}$ and summing its members. For example, if $S = \{1, 4, 5\}$, then

$$\begin{aligned} P_1 &= \{0, 1\} , \\ P_2 &= \{0, 1, 4, 5\} , \\ P_3 &= \{0, 1, 4, 5, 6, 9, 10\} . \end{aligned}$$

Given the identity

$$P_i = P_{i-1} \cup (P_{i-1} + x_i) , \tag{35.21}$$

you can prove by induction on $i$ (see Exercise 35.5-1) that the list $L_i$ is a sorted list containing every element of $P_i$ whose value is not more than $t$. Since the length of $L_i$ can be as much as $2^i$, EXACT-SUBSET-SUM is an exponential-time algorithm in general, although it is a polynomial-time algorithm in the special cases in which $t$ is polynomial in $|S|$ or all the numbers in $S$ are bounded by a polynomial in $|S|$.

## A fully polynomial-time approximation scheme

The key to devising a fully polynomial-time approximation scheme for the subset-sum problem is to "trim" each list $L_i$ after it is created. Here's the idea behind trimming: if two values in $L$ are close to each other, then since the goal is just an approximate solution, there is no need to maintain both of them explicitly. More precisely, use a trimming parameter $\delta$ such that $0 < \delta < 1$. When ***trimming*** a list $L$ by $\delta$, remove as many elements from $L$ as possible, in such a way that if $L'$ is the result of trimming $L$, then for every element $y$ that was removed from $L$, some element $z$ still in $L'$ approximates $y$. For $z$ to approximate $y$, it must be no greater than $y$ and also within a factor of $1 + \delta$ of $y$, so that

$$\frac{y}{1+\delta} \le z \le y . \tag{35.22}$$

You can think of such a $z$ as "representing" $y$ in the new list $L'$. Each removed element $y$ is represented by a remaining element $z$ satisfying inequality (35.22). For example, suppose that $\delta = 0.1$ and

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle .$$

Then trimming $L$ results in

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle ,$$

where the deleted value 11 is represented by 10, the deleted values 21 and 22 are represented by 20, and the deleted value 24 is represented by 23. Because every element of the trimmed version of the list is also an element of the original version of the list, trimming can dramatically decrease the number of elements kept while keeping a close (and slightly smaller) representative value in the list for each deleted element.

The procedure TRIM trims list $L = \langle y_1, y_2, \ldots, y_m \rangle$ in $\Theta(m)$ time, given $L$ and the trimming parameter $\delta$. It assumes that $L$ is sorted into monotonically increasing order. The output of the procedure is a trimmed, sorted list. The procedure scans the elements of $L$ in monotonically increasing order. A number is appended onto the returned list $L'$ only if it is the first element of $L$ or if it cannot be represented by the most recent number placed into $L'$.

```
TRIM($L, \delta$)
1  let $m$ be the length of $L$
2  $L' = \langle y_1 \rangle$
3  $last = y_1$
4  for $i = 2$ to $m$
5      if $y_i > last \cdot (1 + \delta)$        // $y_i \geq last$ because $L$ is sorted
6          append $y_i$ onto the end of $L'$
7          $last = y_i$
8  return $L'$
```

Given the procedure TRIM, the procedure APPROX-SUBSET-SUM on the following page implements the approximation scheme. This procedure takes as input a set $S = \{x_1, x_2, \ldots, x_n\}$ of $n$ integers (in arbitrary order), the size $n = |S|$, the target integer $t$, and an approximation parameter $\epsilon$, where

$$0 < \epsilon < 1 . \tag{35.23}$$

It returns a value $z^*$ whose value is within a factor of $1 + \epsilon$ of the optimal solution.

The APPROX-SUBSET-SUM procedure works as follows. Line 1 initializes the list $L_0$ to be the list containing just the element 0. The **for** loop in lines 2–5 computes $L_i$ as a sorted list containing a suitably trimmed version of the set $P_i$, with all elements larger than $t$ removed. Since the procedure creates $L_i$ from $L_{i-1}$, it must ensure that the repeated trimming doesn't introduce too much compounded inaccuracy. That's why instead of the trimming parameter being $\epsilon$ in the call to TRIM, it has the smaller value $\epsilon/2n$. We'll soon see that APPROX-SUBSET-SUM returns a correct approximation if one exists.

```
APPROX-SUBSET-SUM($S, n, t, \epsilon$)
1  $L_0 = \langle 0 \rangle$
2  for $i = 1$ to $n$
3      $L_i = $ MERGE-LISTS($L_{i-1}, L_{i-1} + x_i$)
4      $L_i = $ TRIM($L_i, \epsilon/2n$)
5      remove from $L_i$ every element that is greater than $t$
6  let $z^*$ be the largest value in $L_n$
7  return $z^*$
```

As an example, suppose that APPROX-SUBSET-SUM is given

$$S = \langle 104, 102, 201, 101 \rangle$$

with $t = 308$ and $\epsilon = 0.40$. The trimming parameter $\delta$ is $\epsilon/2n = 0.40/8 = 0.05$. The procedure computes the following values on the indicated lines:

line 1: $L_0 = \langle 0 \rangle$ ,

line 3: $L_1 = \langle 0, 104 \rangle$ ,
line 4: $L_1 = \langle 0, 104 \rangle$ ,
line 5: $L_1 = \langle 0, 104 \rangle$ ,

line 3: $L_2 = \langle 0, 102, 104, 206 \rangle$ ,
line 4: $L_2 = \langle 0, 102, 206 \rangle$ ,
line 5: $L_2 = \langle 0, 102, 206 \rangle$ ,

line 3: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$ ,
line 4: $L_3 = \langle 0, 102, 201, 303, 407 \rangle$ ,
line 5: $L_3 = \langle 0, 102, 201, 303 \rangle$ ,

line 3: $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$ ,
line 4: $L_4 = \langle 0, 101, 201, 302, 404 \rangle$ ,
line 5: $L_4 = \langle 0, 101, 201, 302 \rangle$ .

The procedure returns $z^* = 302$ as its answer, which is well within $\epsilon = 40\%$ of the optimal answer $307 = 104 + 102 + 101$. In fact, it is within 2%.

***Theorem 35.7***
APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme for the subset-sum problem.

***Proof*** The operations of trimming $L_i$ in line 4 and removing from $L_i$ every element that is greater than $t$ maintain the property that every element of $L_i$ is also a member of $P_i$. Therefore, the value $z^*$ returned in line 7 is indeed the sum of some subset of $S$, that is, $z^* \in P_n$. Let $y^* \in P_n$ denote an optimal solution to the subset-sum problem, so that it is the greatest value in $P_n$ that is less than or equal to $t$. Because line 5 ensures that $z^* \le t$, we know that $z^* \le y^*$. By inequality (35.1), we need to show that $y^*/z^* \le 1 + \epsilon$. We must also show that the running time of this algorithm is polynomial in both $1/\epsilon$ and the size of the input.

As Exercise 35.5-2 asks you to show, for every element $y$ in $P_i$ that is at most $t$, there exists an element $z \in L_i$ such that

$$\frac{y}{(1 + \epsilon/2n)^i} \le z \le y . \tag{35.24}$$

Inequality (35.24) must hold for $y^* \in P_n$, and therefore there exists an element $z \in L_n$ such that

$$\frac{y^*}{(1 + \epsilon/2n)^n} \le z \le y^* ,$$

and thus

$$\frac{y^*}{z} \le \left(1 + \frac{\epsilon}{2n}\right)^n . \tag{35.25}$$

Since there exists an element $z \in L_n$ fulfilling inequality (35.25), the inequality must hold for $z^*$, which is the largest value in $L_n$, which is to say

$$\frac{y^*}{z^*} \le \left(1 + \frac{\epsilon}{2n}\right)^n . \tag{35.26}$$

Now we show that $y^*/z^* \le 1+\epsilon$. We do so by showing that $(1+\epsilon/2n)^n \le 1+\epsilon$. First, inequality (35.23), $0 < \epsilon < 1$, implies that

$$(\epsilon/2)^2 \le \epsilon/2 \tag{35.27}$$

Next, from equation (3.16) on page 66, we have $\lim_{n\to\infty}(1 + \epsilon/2n)^n = e^{\epsilon/2}$. Exercise 35.5-3 asks you to show that

$$\frac{d}{dn}\left(1 + \frac{\epsilon}{2n}\right)^n > 0 . \tag{35.28}$$

Therefore, the function $(1 + \epsilon/2n)^n$ increases with $n$ as it approaches its limit of $e^{\epsilon/2}$, and we have

$$\begin{aligned}\left(1 + \frac{\epsilon}{2n}\right)^n &\le e^{\epsilon/2} \\ &\le 1 + \epsilon/2 + (\epsilon/2)^2 \quad \text{(by inequality (3.15) on page 66)} \\ &\le 1 + \epsilon \quad \text{(by inequality (35.27))} .\end{aligned} \tag{35.29}$$

Combining inequalities (35.26) and (35.29) completes the analysis of the approximation ratio.

To show that APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme, we derive a bound on the length of $L_i$. After trimming, successive elements $z$ and $z'$ of $L_i$ must have the relationship $z'/z > 1+\epsilon/2n$. That is, they must differ by a factor of at least $1 + \epsilon/2n$. Each list, therefore, contains the value 0, possibly the value 1, and up to $\lfloor \log_{1+\epsilon/2n} t \rfloor$ additional values. The number of elements in each list $L_i$ is at most

$$\begin{aligned}
\log_{1+\epsilon/2n} t + 2 &= \frac{\ln t}{\ln(1+\epsilon/2n)} + 2 \\
&\le \frac{2n(1+\epsilon/2n)\ln t}{\epsilon} + 2 && \text{(by inequality (3.23) on page 67)} \\
&< \frac{3n \ln t}{\epsilon} + 2 && \text{(by inequality (35.23), } 0 < \epsilon < 1) .
\end{aligned}$$

This bound is polynomial in the size of the input—which is the number of bits $\lg t$ needed to represent $t$ plus the number of bits needed to represent the set $S$, which in turn is polynomial in $n$—and in $1/\epsilon$. Since the running time of APPROX-SUBSET-SUM is polynomial in the lengths of the lists $L_i$, we conclude that APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme. ■

## Exercises

***35.5-1***

Prove equation (35.21). Then show that after executing line 4 of EXACT-SUBSET-SUM, $L_i$ is a sorted list containing every element of $P_i$ whose value is not more than $t$.

***35.5-2***

Using induction on $i$, prove inequality (35.24).

***35.5-3***

Prove inequality (35.28).

***35.5-4***

How can you modify the approximation scheme presented in this section to find a good approximation to the smallest value not less than $t$ that is a sum of some subset of the given input list?

***35.5-5***

Modify the APPROX-SUBSET-SUM procedure to also return the subset of $S$ that sums to the value $z^*$.

## Problems

### *35-1 Bin packing*

You are given a set of $n$ objects, where the size $s_i$ of the $i$th object satisfies $0 < s_i < 1$. Your goal is to pack all the objects into the minimum number of unit-size bins. Each bin can hold any subset of the objects whose total size does not exceed 1.

***a.*** Prove that the problem of determining the minimum number of bins required is NP-hard. (*Hint:* Reduce from the subset-sum problem.)

The ***first-fit*** heuristic takes each object in turn and places it into the first bin that can accommodate it, as follows. It maintains an ordered list of bins. Let $b$ denote the number of bins in the list, where $b$ increases over the course of the algorithm, and let $\langle B_1, \ldots, B_b \rangle$ be the list of bins. Initially $b = 0$ and the list is empty. The algorithm takes each object $i$ in turn and places it in the lowest-numbered bin that can still accommodate it. If no bin can accommodate object $i$, then $b$ is incremented and a new bin $B_b$ is opened, containing object $i$. Let $S = \sum_{i=1}^{n} s_i$.

***b.*** Argue that the optimal number of bins required is at least $\lceil S \rceil$.

***c.*** Argue that the first-fit heuristic leaves at most one bin at most half full.

***d.*** Prove that the number of bins used by the first-fit heuristic never exceeds $\lceil 2S \rceil$.

***e.*** Prove an approximation ratio of 2 for the first-fit heuristic.

***f.*** Give an efficient implementation of the first-fit heuristic, and analyze its running time.

### *35-2 Approximating the size of a maximum clique*

Let $G = (V, E)$ be an undirected graph. For any $k \geq 1$, define $G^{(k)}$ to be the undirected graph $(V^{(k)}, E^{(k)})$, where $V^{(k)}$ is the set of all ordered $k$-tuples of vertices from $V$ and $E^{(k)}$ is defined so that $(v_1, v_2, \ldots, v_k)$ is adjacent to $(w_1, w_2, \ldots, w_k)$ if and only if for $i = 1, 2, \ldots, k$, either vertex $v_i$ is adjacent to $w_i$ in $G$, or else $v_i = w_i$.

***a.*** Prove that the size of the maximum clique in $G^{(k)}$ is equal to the $k$th power of the size of the maximum clique in $G$.

***b.*** Argue that if there is an approximation algorithm that has a constant approximation ratio for finding a maximum-size clique, then there is a polynomial-time approximation scheme for the problem.

***35-3 Weighted set-covering problem***

Suppose that sets have weights in the set-covering problem, so that each set $S_i$ in the family $\mathcal{F}$ has an associated weight $w_i$. The weight of a cover $\mathcal{C}$ is $\sum_{S_i \in \mathcal{C}} w_i$. The goal is wish to determine a minimum-weight cover. (Section 35.3 handles the case in which $w_i = 1$ for all $i$.)

Show how to generalize the greedy set-covering heuristic in a natural manner to provide an approximate solution for any instance of the weighted set-covering problem. Letting $d$ be the maximum size of any set $S_i$, show that your heuristic has an approximation ratio of $H(d) = \sum_{i=1}^{d} 1/i$.

***35-4 Maximum matching***

Recall that for an undirected graph $G$, a matching is a set of edges such that no two edges in the set are incident on the same vertex. Section 25.1 showed how to find a maximum matching in a bipartite graph, that is, a matching such that no other matching in $G$ contains more edges. This problem examines matchings in undirected graphs that are not required to be bipartite.

***a.*** Show that a maximal matching need not be a maximum matching by exhibiting an undirected graph $G$ and a maximal matching $M$ in $G$ that is not a maximum matching. (*Hint:* You can find such a graph with only four vertices.)

***b.*** Consider a connected, undirected graph $G = (V, E)$. Give an $O(E)$-time greedy algorithm to find a maximal matching in $G$.

This problem concentrates on a polynomial-time approximation algorithm for maximum matching. Whereas the fastest known algorithm for maximum matching takes superlinear (but polynomial) time, the approximation algorithm here will run in linear time. You will show that the linear-time greedy algorithm for maximal matching in part (b) is a 2-approximation algorithm for maximum matching.

***c.*** Show that the size of a maximum matching in $G$ is a lower bound on the size of any vertex cover for $G$.

***d.*** Consider a maximal matching $M$ in $G = (V, E)$. Let $T = \{v \in V : \text{some edge in } M \text{ is incident on } v\}$. What can you say about the subgraph of $G$ induced by the vertices of $G$ that are not in $T$?

***e.*** Conclude from part (d) that $2\,|M|$ is the size of a vertex cover for $G$.

***f.*** Using parts (c) and (e), prove that the greedy algorithm in part (b) is a 2-approximation algorithm for maximum matching.

### *35-5 Parallel machine scheduling*

In the ***parallel-machine-scheduling problem***, the input has two parts: $n$ jobs, $J_1, J_2, \ldots, J_n$, where each job $J_k$ has an associated nonnegative processing time of $p_k$, and $m$ identical machines, $M_1, M_2, \ldots, M_m$. Any job can run on any machine. A ***schedule*** specifies, for each job $J_k$, the machine on which it runs and the time period during which it runs. Each job $J_k$ must run on some machine $M_i$ for $p_k$ consecutive time units, and during that time period no other job may run on $M_i$. Let $C_k$ denote the ***completion time*** of job $J_k$, that is, the time at which job $J_k$ completes processing. Given a schedule, define $C_{\max} = \max\{C_j : 1 \le j \le n\}$ to be the ***makespan*** of the schedule. The goal is to find a schedule whose makespan is minimum.

For example, consider an input with two machines $M_1$ and $M_2$, and four jobs $J_1$, $J_2$, $J_3$, and $J_4$ with $p_1 = 2$, $p_2 = 12$, $p_3 = 4$, and $p_4 = 5$. Then one possible schedule runs, on machine $M_1$, job $J_1$ followed by job $J_2$, and on machine $M_2$, job $J_4$ followed by job $J_3$. For this schedule, $C_1 = 2$, $C_2 = 14$, $C_3 = 9$, $C_4 = 5$, and $C_{\max} = 14$. An optimal schedule runs job $J_2$ on machine $M_1$ and jobs $J_1$, $J_3$, and $J_4$ on machine $M_2$. For this schedule, we have $C_1 = 2$, $C_2 = 12$, $C_3 = 6$, and $C_4 = 11$, and so $C_{\max} = 12$.

Given the input to a parallel-machine-scheduling problem, let $C^*_{\max}$ denote the makespan of an optimal schedule.

***a.*** Show that the optimal makespan is at least as large as the greatest processing time, that is,

$$C^*_{\max} \ge \max\{p_k : 1 \le k \le n\} .$$

***b.*** Show that the optimal makespan is at least as large as the average machine load, that is,

$$C^*_{\max} \ge \frac{1}{m}\sum_{k=1}^{n} p_k .$$

Consider the following greedy algorithm for parallel machine scheduling: whenever a machine is idle, schedule any job that has not yet been scheduled.

***c.*** Write pseudocode to implement this greedy algorithm. What is the running time of your algorithm?

***d.*** For the schedule returned by the greedy algorithm, show that

$$C_{\max} \le \frac{1}{m}\sum_{k=1}^{n} p_k + \max\{p_k : 1 \le k \le n\} .$$

Conclude that this algorithm is a polynomial-time 2-approximation algorithm.

***35-6 Approximating a maximum spanning tree***

Let $G = (V, E)$ be an undirected graph with distinct edge weights $w(u, v)$ on each edge $(u, v) \in E$. For each vertex $v \in V$, denote by $\max(v)$ the maximum-weight edge incident on that vertex. Let $S_G = \{\max(v) : v \in V\}$ be the set of maximum-weight edges incident on each vertex, and let $T_G$ be the maximum-weight spanning tree of $G$, that is, the spanning tree of maximum total weight. For any subset of edges $E' \subseteq E$, define $w(E') = \sum_{(u,v) \in E'} w(u, v)$.

***a.*** Give an example of a graph with at least 4 vertices for which $S_G = T_G$.

***b.*** Give an example of a graph with at least 4 vertices for which $S_G \neq T_G$.

***c.*** Prove that $S_G \subseteq T_G$ for any graph $G$.

***d.*** Prove that $w(S_G) \geq w(T_G)/2$ for any graph $G$.

***e.*** Give an $O(V + E)$-time algorithm to compute a 2-approximation to the maximum spanning tree.

***35-7 An approximation algorithm for the 0-1 knapsack problem***

Recall the knapsack problem from Section 15.2. The input includes $n$ items, where the $i$th item is worth $v_i$ dollars and weighs $w_i$ pounds. The input also includes the capacity of a knapsack, which is $W$ pounds. Here, we add the further assumptions that each weight $w_i$ is at most $W$ and that the items are indexed in monotonically decreasing order of their values: $v_1 \geq v_2 \geq \cdots \geq v_n$.

In the 0-1 knapsack problem, the goal is to find a subset of the items whose total weight is at most $W$ and whose total value is maximum. The fractional knapsack problem is like the 0-1 knapsack problem, except that a fraction of each item may be put into the knapsack, rather than either all or none of each item. If a fraction $x_i$ of item $i$ goes into the knapsack, where $0 \leq x_i \leq 1$, it contributes $x_i w_i$ to the weight of the knapsack and adds value $x_i v_i$. The goal of this problem is to develop a polynomial-time 2-approximation algorithm for the 0-1 knapsack problem.

In order to design a polynomial-time algorithm, let's consider restricted instances of the 0-1 knapsack problem. Given an instance $I$ of the knapsack problem, form restricted instances $I_j$, for $j = 1, 2, \ldots, n$, by removing items $1, 2, \ldots, j-1$ and requiring the solution to include item $j$ (all of item $j$ in both the fractional and 0-1 knapsack problems). No items are removed in instance $I_1$. For instance $I_j$, let $P_j$ denote an optimal solution to the 0-1 problem and $Q_j$ denote an optimal solution to the fractional problem.

***a.*** Argue that an optimal solution to instance $I$ of the 0-1 knapsack problem is one of $\{P_1, P_2, \ldots, P_n\}$.

***b.*** Prove that to find an optimal solution $Q_j$ to the fractional problem for instance $I_j$, you can include item $j$ and then use the greedy algorithm in which each step takes as much as possible of the unchosen item with the maximum value per pound $v_i/w_i$ in the set $\{j+1, j+2, \ldots, n\}$.

***c.*** Prove that there is always an optimal solution $Q_j$ to the fractional problem for instance $I_j$ that includes at most one item fractionally. That is, for all items except possibly one, either all of the item or none of the item goes into the knapsack.

***d.*** Given an optimal solution $Q_j$ to the fractional problem for instance $I_j$, form solution $R_j$ from $Q_j$ by deleting any fractional items from $Q_j$. Let $v(S)$ denote the total value of items taken in a solution $S$. Prove that $v(R_j) \geq v(Q_j)/2 \geq v(P_j)/2$.

***e.*** Give a polynomial-time algorithm that returns a maximum-value solution from the set $\{R_1, R_2, \ldots, R_n\}$, and prove that your algorithm is a polynomial-time 2-approximation algorithm for the 0-1 knapsack problem.

## Chapter notes

Although methods that do not necessarily compute exact solutions have been known for thousands of years (for example, methods to approximate the value of $\pi$), the notion of an approximation algorithm is much more recent. Hochbaum [221] credits Garey, Graham, and Ullman [175] and Johnson [236] with formalizing the concept of a polynomial-time approximation algorithm. The first such algorithm is often credited to Graham [197].

Since this early work, thousands of approximation algorithms have been designed for a wide range of problems, and there is a wealth of literature on this field. Texts by Ausiello et al. [29], Hochbaum [221], Vazirani [446], and Williamson and Shmoys [459] deal exclusively with approximation algorithms, as do surveys by Shmoys [409] and Klein and Young [256]. Several other texts, such as Garey and Johnson [176] and Papadimitriou and Steiglitz [353], have significant coverage of approximation algorithms as well. Books edited by Lawler, Lenstra, Rinnooy Kan, and Shmoys [277] and by Gutin and Punnen [204] provide extensive treatments of approximation algorithms and heuristics for the traveling-salesperson problem.

Papadimitriou and Steiglitz attribute the algorithm APPROX-VERTEX-COVER to F. Gavril and M. Yannakakis. The vertex-cover problem has been studied extensively (Hochbaum [221] lists 16 different approximation algorithms for this problem), but all the approximation ratios are at least $2 - o(1)$.

The algorithm APPROX-TSP-TOUR appears in a paper by Rosenkrantz, Stearns, and Lewis [384]. Christofides improved on this algorithm and gave a 3/2-approximation algorithm for the traveling-salesperson problem with the triangle inequality. Arora [23] and Mitchell [330] have shown that if the points lie in the euclidean plane, there is a polynomial-time approximation scheme. Theorem 35.3 is due to Sahni and Gonzalez [392].

The algorithm APPROX-SUBSET-SUM and its analysis are loosely modeled after related approximation algorithms for the knapsack and subset-sum problems by Ibarra and Kim [234].

Problem 35-7 is a combinatorial version of a more general result on approximating knapsack-type integer programs by Bienstock and McClosky [55].

The randomized algorithm for MAX-3-CNF satisfiability is implicit in the work of Johnson [236]. The weighted vertex-cover algorithm is by Hochbaum [220]. Section 35.4 only touches on the power of randomization and linear programming in the design of approximation algorithms. A combination of these two ideas yields a technique called "randomized rounding," which formulates a problem as an integer linear program, solves the linear-programming relaxation, and interprets the variables in the solution as probabilities. These probabilities then help guide the solution of the original problem. This technique was first used by Raghavan and Thompson [374], and it has had many subsequent uses. (See Motwani, Naor, and Raghavan [335] for a survey.) Several other notable ideas in the field of approximation algorithms include the primal-dual method (see Goemans and Williamson [184] for a survey), finding sparse cuts for use in divide-and-conquer algorithms [288], and the use of semidefinite programming [183].

As mentioned in the chapter notes for Chapter 34, results in probabilistically checkable proofs have led to lower bounds on the approximability of many problems, including several in this chapter. In addition to the references there, the chapter by Arora and Lund [26] contains a good description of the relationship between probabilistically checkable proofs and the hardness of approximating various problems.