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:

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

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 ϵ, 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 ϵ decreases. For example, the running time of a polynomial-time approximation scheme might be $O(n^{2/\epsilon})$. Ideally, if ϵ 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 ϵ 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 ϵ 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 ρ" and "ρ-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 ρ-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 = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {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. ■

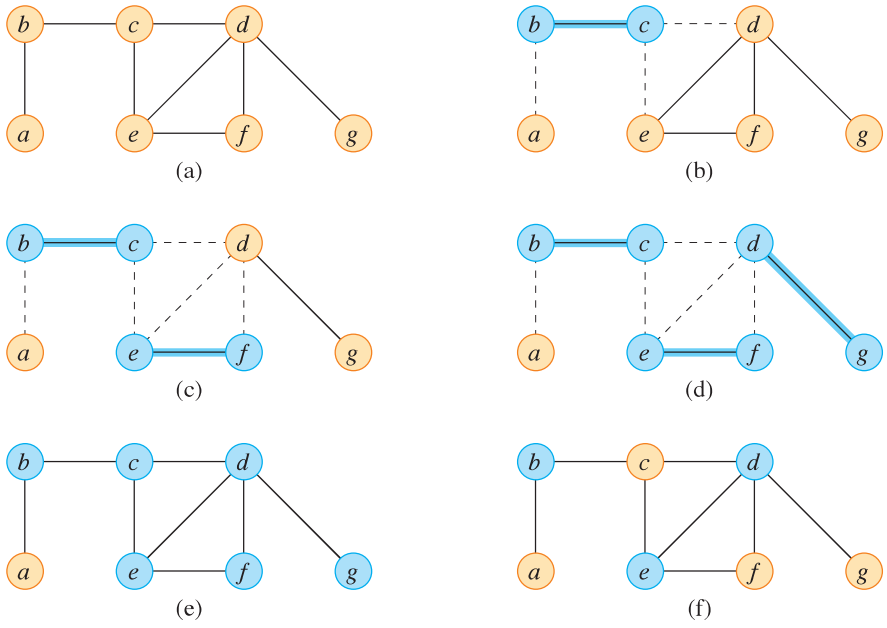

Figure 35.1 The operation of APPROX-VERTEX-COVER. **(a)** The input graph G, which has 7 vertices and 8 edges. **(b)** The highlighted edge (b, c) is the first edge chosen by APPROX-VERTEX-COVER. Vertices b and c, in blue, are added to the set C containing the vertex cover being created. Dashed edges (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 $u, v, w \in V$,

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

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.

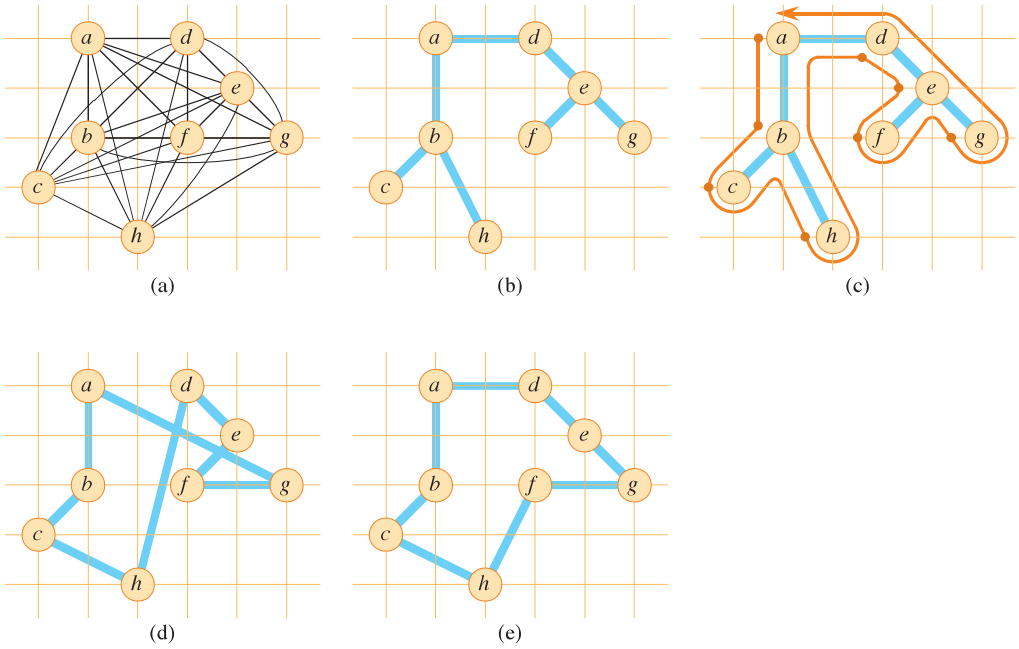

Figure 35.2 The operation of APPROX-TSP-TOUR. **(a)** A complete undirected graph. Vertices lie on intersections of integer grid lines. For example, f is one unit to the right and two units up from h. The cost function between two points is the ordinary euclidean distance. **(b)** A minimum spanning 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 ev-

ery 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 P = NP.

Theorem 35.3
If P $\neq$ NP, then for any constant $\rho \ge 1$, there is no polynomial-time approximation algorithm with approximation ratio ρ 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 ρ. Without loss of generality, assume that ρ 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 P = 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,

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

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

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