Chapter 11

Approximation Algorithms

Following our encounter with NP-completeness and the idea of computational intractability in general, we've been dealing with a fundamental question: How should we design algorithms for problems where polynomial time is probably an unattainable goal?

In this chapter, we focus on a new theme related to this question: *approximation algorithms*, which run in polynomial time and find solutions that are guaranteed to be close to optimal. There are two key words to notice in this definition: *close* and *guaranteed*. We will not be seeking the optimal solution, and as a result, it becomes feasible to aim for a polynomial running time. At the same time, we will be interested in proving that our algorithms find solutions that are guaranteed to be close to the optimum. There is something inherently tricky in trying to do this: In order to prove an approximation guarantee, we need to compare our solution with—and hence reason about—an optimal solution that is computationally very hard to find. This difficulty will be a recurring issue in the analysis of the algorithms in this chapter.

We will consider four general techniques for designing approximation algorithms. We start with *greedy algorithms*, analogous to the kind of algorithms we developed in Chapter 4. These algorithms will be simple and fast, as in Chapter 4, with the challenge being to find a greedy rule that leads to solutions provably close to optimal. The second general approach we pursue is the *pricing method*. This approach is motivated by an economic perspective; we will consider a price one has to pay to enforce each constraint of the problem. For example, in a graph problem, we can think of the nodes or edges of the graph sharing the cost of the solution in some equitable way. The pricing method is often referred to as the *primal-dual technique*, a term inherited from

the study of linear programming, which can also be used to motivate this approach. Our presentation of the pricing method here will not assume familiarity with linear programming. We will introduce linear programming through our third technique in this chapter, *linear programming and rounding*, in which one exploits the relationship between the computational feasibility of linear programming and the expressive power of its more difficult cousin, *integer programming*. Finally, we will describe a technique that can lead to extremely good approximations: using dynamic programming on a rounded version of the input.

11.1 Greedy Algorithms and Bounds on the Optimum: A Load Balancing Problem

As our first topic in this chapter, we consider a fundamental *Load Balancing Problem* that arises when multiple servers need to process a set of jobs or requests. We focus on a basic version of the problem in which all servers are identical, and each can be used to serve any of the requests. This simple problem is useful for illustrating some of the basic issues that one needs to deal with in designing and analyzing approximation algorithms, particularly the task of comparing an approximate solution with an optimum solution that we cannot compute efficiently. Moreover, we'll see that the general issue of load balancing is a problem with many facets, and we'll explore some of these in later sections.

The Problem

We formulate the Load Balancing Problem as follows. We are given a set of m machines $M_1, \ldots, M_m$ and a set of n jobs; each job j has a processing time t_j. We seek to assign each job to one of the machines so that the loads placed on all machines are as "balanced" as possible.

More concretely, in any assignment of jobs to machines, we can let $A(i)$ denote the set of jobs assigned to machine M_i; under this assignment, machine M_i needs to work for a total time of

$$T_i = \sum_{j \in A(i)} t_j,$$

and we declare this to be the *load* on machine M_i. We seek to minimize a quantity known as the *makespan*; it is simply the maximum load on any machine, $T = \max_i T_i$. Although we will not prove this, the scheduling problem of finding an assignment of minimum makespan is NP-hard.

Designing the Algorithm

We first consider a very simple greedy algorithm for the problem. The algorithm makes one pass through the jobs in any order; when it comes to job j, it assigns j to the machine whose load is smallest so far.

```
Greedy-Balance:
Start with no jobs assigned
Set T_i = 0 and A(i) = ∅ for all machines M_i
For j = 1, ..., n
  Let M_i be a machine that achieves the minimum min_k T_k
  Assign job j to machine M_i
  Set A(i) ← A(i) ∪ {j}
  Set T_i ← T_i + t_j
EndFor
```

For example, Figure 11.1 shows the result of running this greedy algorithm on a sequence of six jobs with sizes $2, 3, 4, 6, 2, 2$; the resulting makespan is 8, the "height" of the jobs on the first machine. Note that this is not the optimal solution; had the jobs arrived in a different order, so that the algorithm saw the sequence of sizes $6, 4, 3, 2, 2, 2$, then it would have produced an allocation with a makespan of 7.

Analyzing the Algorithm

Let T denote the makespan of the resulting assignment; we want to show that T is not much larger than the minimum possible makespan T^*. Of course, in trying to do this, we immediately encounter the basic problem mentioned above: We need to compare our solution to the optimal value T^*, even though we don't know what this value is and have no hope of computing it. For the

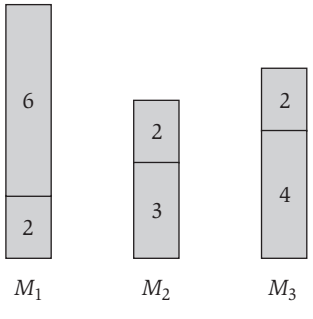

Figure 11.1 The result of running the greedy load balancing algorithm on three machines with job sizes $2, 3, 4, 6, 2, 2$.

analysis, therefore, we will need a *lower bound* on the optimum—a quantity with the guarantee that no matter how good the optimum is, it cannot be less than this bound.

There are many possible lower bounds on the optimum. One idea for a lower bound is based on considering the total processing time $\sum_j t_j$. One of the m machines must do at least a $1/m$ fraction of the total work, and so we have the following.

(11.1) *The optimal makespan is at least*

$$T^* \geq \frac{1}{m} \sum_j t_j.$$

There is a particular kind of case in which this lower bound is much too weak to be useful. Suppose we have one job that is extremely long relative to the sum of all processing times. In a sufficiently extreme version of this, the optimal solution will place this job on a machine by itself, and it will be the last one to finish. In such a case, our greedy algorithm would actually produce the optimal solution; but the lower bound in (11.1) isn't strong enough to establish this.

This suggests the following additional lower bound on T^*.

(11.2) *The optimal makespan is at least* $T^* \geq \max_j t_j$.

Now we are ready to evaluate the assignment obtained by our greedy algorithm.

(11.3) *Algorithm* `Greedy-Balance` *produces an assignment of jobs to machines with makespan* $T \leq 2T^*$.

Proof. Here is the overall plan for the proof. In analyzing an approximation algorithm, one compares the solution obtained to what one knows about the optimum—in this case, our lower bounds (11.1) and (11.2). We consider a machine M_i that attains the maximum load T in our assignment, and we ask: What was the last job j to be placed on M_i? If t_j is not too large relative to most of the other jobs, then we are not too far above the lower bound (11.1). And, if t_j is a very large job, then we can use (11.2). Figure 11.2 shows the structure of this argument.

Here is how we can make this precise. When we assigned job j to M_i, the machine M_i had the smallest load of any machine; this is the key property of our greedy algorithm. Its load just before this assignment was $T_i - t_j$, and since this was the smallest load at that moment, it follows that every machine

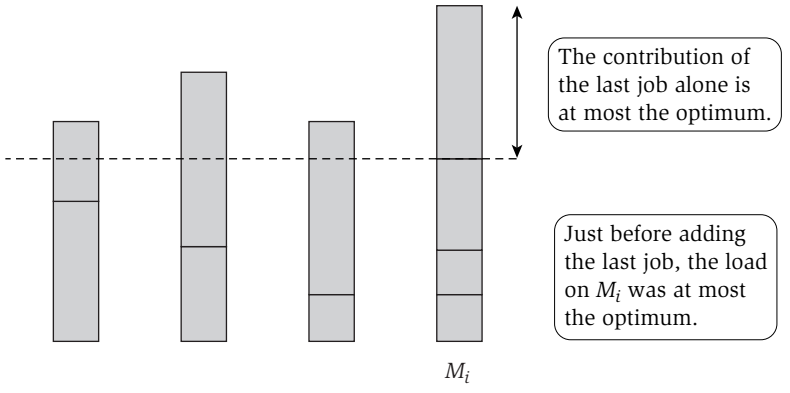

Figure 11.2 Accounting for the load on machine M_i in two parts: the last job to be added, and all the others.

had load at least $T_i - t_j$. Thus, adding up the loads of all machines, we have $\sum_k T_k \geq m(T_i - t_j)$, or equivalently,

$$T_i - t_j \leq \frac{1}{m} \sum_k T_k.$$

But the value $\sum_k T_k$ is just the total load of all jobs $\sum_j t_j$ (since every job is assigned to exactly one machine), and so the quantity on the right-hand side of this inequality is exactly our lower bound on the optimal value, from (11.1). Thus

$$T_i - t_j \leq T^*.$$

Now we account for the remaining part of the load on M_i, which is just the final job j. Here we simply use the other lower bound we have, (11.2), which says that $t_j \leq T^*$. Adding up these two inequalities, we see that

$$T_i = (T_i - t_j) + t_j \leq 2T^*.$$

Since our makespan T is equal to T_i, this is the result we want. ■

It is not hard to give an example in which the solution is indeed close to a factor of 2 away from optimal. Suppose we have m machines and $n = m(m-1) + 1$ jobs. The first $m(m-1) = n - 1$ jobs each require time $t_j = 1$. The last job is much larger; it requires time $t_n = m$. What does our greedy algorithm do with this sequence of jobs? It evenly balances the first $n - 1$ jobs, and then has to add the giant job n to one of them; the resulting makespan is $T = 2m - 1$.

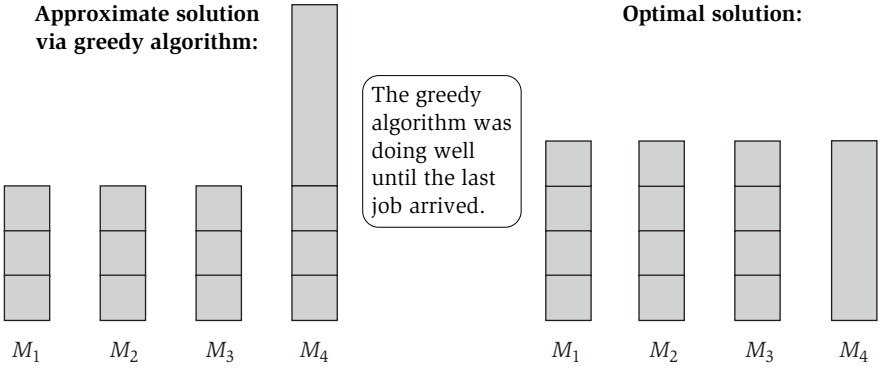

Figure 11.3 A bad example for the greedy balancing algorithm with $m = 4$.

What does the optimal solution look like in this example? It assigns the large job to one of the machines, say, M_1, and evenly spreads the remaining jobs over the other $m - 1$ machines. This results in a makespan of m. Thus the ratio between the greedy algorithm's solution and the optimal solution is $(2m - 1)/m = 2 - 1/m$, which is close to a factor of 2 when m is large.

See Figure 11.3 for a picture of this with $m = 4$; one has to admire the perversity of the construction, which misleads the greedy algorithm into perfectly balancing everything, only to mess everything up with the final giant item.

In fact, with a little care, one can improve the analysis in (11.3) to show that the greedy algorithm with m machines is within exactly this factor of $2 - 1/m$ on every instance; the example above is really as bad as possible.

Extensions: An Improved Approximation Algorithm

Now let's think about how we might develop a better approximation algorithm—in other words, one for which we are always guaranteed to be within a factor strictly smaller than 2 away from the optimum. To do this, it helps to think about the worst cases for our current approximation algorithm. Our earlier bad example had the following flavor: We spread everything out very evenly across the machines, and then one last, giant, unfortunate job arrived. Intuitively, it looks like it would help to get the largest jobs arranged nicely first, with the idea that later, small jobs can only do so much damage. And in fact, this idea does lead to a measurable improvement.

Thus we now analyze the variant of the greedy algorithm that first sorts the jobs in decreasing order of processing time and then proceeds as before.

We will prove that the resulting assignment has a makespan that is at most 1.5 times the optimum.

```
Sorted-Balance:
Start with no jobs assigned
Set T_i = 0 and A(i) = ∅ for all machines M_i
Sort jobs in decreasing order of processing times t_j
Assume that t_1 ≥ t_2 ≥ ... ≥ t_n
For j = 1, ..., n
  Let M_i be the machine that achieves the minimum min_k T_k
  Assign job j to machine M_i
  Set A(i) ← A(i) ∪ {j}
  Set T_i ← T_i + t_j
EndFor
```

The improvement comes from the following observation. If we have fewer than m jobs, then the greedy solution will clearly be optimal, since it puts each job on its own machine. And if we have more than m jobs, then we can use the following further lower bound on the optimum.

(11.4) *If there are more than m jobs, then $T^* \geq 2t_{m+1}$.*

Proof. Consider only the first $m+1$ jobs in the sorted order. They each take at least t_{m+1} time. There are $m+1$ jobs and only m machines, so there must be a machine that gets assigned two of these jobs. This machine will have processing time at least $2t_{m+1}$. ■

(11.5) *Algorithm* `Sorted-Balance` *produces an assignment of jobs to machines with makespan $T \leq \frac{3}{2}T^*$.*

Proof. The proof will be very similar to the analysis of the previous algorithm. As before, we will consider a machine M_i that has the maximum load. If M_i only holds a single job, then the schedule is optimal.

So let's assume that machine M_i has at least two jobs, and let t_j be the last job assigned to the machine. Note that $j \geq m+1$, since the algorithm will assign the first m jobs to m distinct machines. Thus $t_j \leq t_{m+1} \leq \frac{1}{2}T^*$, where the second inequality is (11.4).

We now proceed as in the proof of (11.3), with the following single change. At the end of that proof, we had inequalities $T_i - t_j \leq T^*$ and $t_j \leq T^*$, and we added them up to get the factor of 2. But in our case here, the second of these

inequalities is, in fact, $t_j \leq \frac{1}{2}T^*$; so adding the two inequalities gives us the bound

$$T_i \leq \frac{3}{2}T^*. \quad \blacksquare$$

11.2 The Center Selection Problem

Like the problem in the previous section, the Center Selection Problem, which we consider here, also relates to the general task of allocating work across multiple servers. The issue at the heart of Center Selection is where best to place the servers; in order to keep the formulation clean and simple, we will not incorporate the notion of load balancing into the problem. The Center Selection Problem also provides an example of a case in which the most natural greedy algorithm can result in an arbitrarily bad solution, but a slightly different greedy method is guaranteed to always result in a near-optimal solution.

The Problem

Consider the following scenario. We have a set S of n *sites*—say, n little towns in upstate New York. We want to select k *centers* for building large shopping malls. We expect that people in each of these n towns will shop at one of the malls, and so we want to select the sites of the k malls to be central.

Let us start by defining the input to our problem more formally. We are given an integer k, a set S of n sites (corresponding to the towns), and a distance function. When we consider instances where the sites are points in the plane, the distance function will be the standard Euclidean distance between points, and any point in the plane is an option for placing a center. The algorithm we develop, however, can be applied to more general notions of distance. In applications, distance sometimes means straight-line distance, but can also mean the travel time from point s to point z, or the driving distance (i.e., distance along roads), or even the cost of traveling. We will allow any distance function that satisfies the following natural properties.

- $dist(s, s) = 0$ for all $s \in S$
- the distance is symmetric: $dist(s, z) = dist(z, s)$ for all sites $s, z \in S$
- the triangle inequality: $dist(s, z) + dist(z, h) \geq dist(s, h)$

The first and third of these properties tend to be satisfied by essentially all natural notions of distance. Although there are applications with asymmetric distances, most cases of interest also satisfy the second property. Our greedy algorithm will apply to any distance function that satisfies these three properties, and it will depend on all three.

Next we have to clarify what we mean by the goal of wanting the centers to be "central." Let C be a set of centers. We assume that the people in a given town will shop at the closest mall. This suggests we define the distance of a site s to the centers as $dist(s, C) = \min_{c \in C} dist(s, c)$. We say that C forms an *r-cover* if each site is within distance at most r from one of the centers—that is, if $dist(s, C) \leq r$ for all sites $s \in S$. The minimum r for which C is an r-cover will be called the *covering radius* of C and will be denoted by $r(C)$. In other words, the covering radius of a set of centers C is the farthest that anyone needs to travel to get to his or her nearest center. Our goal will be to select a set C of k centers for which $r(C)$ is as small as possible.

Designing and Analyzing the Algorithm

Difficulties with a Simple Greedy Algorithm We now discuss greedy algorithms for this problem. As before, the meaning of "greedy" here is necessarily a little fuzzy; essentially, we consider algorithms that select sites one by one in a myopic fashion—that is, choosing each without explicitly considering where the remaining sites will go.

Probably the simplest greedy algorithm would work as follows. It would put the first center at the best possible location for a single center, then keep adding centers so as to reduce the covering radius, each time, by as much as possible. It turns out that this approach is a bit too simplistic to be effective: there are cases where it can lead to very bad solutions.

To see that this simple greedy approach can be really bad, consider an example with only two sites s and z, and $k = 2$. Assume that s and z are located in the plane, with distance equal to the standard Euclidean distance in the plane, and that any point in the plane is an option for placing a center. Let d be the distance between s and z. Then the best location for a single center c_1 is halfway between s and z, and the covering radius of this one center is $r(\{c_1\}) = d/2$. The greedy algorithm would start with c_1 as the first center. No matter where we add a second center, at least one of s or z will have the center c_1 as closest, and so the covering radius of the set of two centers will still be $d/2$. Note that the optimum solution with $k = 2$ is to select s and z themselves as the centers. This will lead to a covering radius of 0. A more complex example illustrating the same problem can be obtained by having two dense "clusters" of sites, one around s and one around z. Here our proposed greedy algorithm would start by opening a center halfway between the clusters, while the optimum solution would open a separate center for each cluster.

Knowing the Optimal Radius Helps In searching for an improved algorithm, we begin with a useful thought experiment. Suppose for a minute that someone told us what the optimum radius r is. Would this information help? That is, suppose we *know* that there is a set of k centers C^* with radius $r(C^*) \leq r$, and

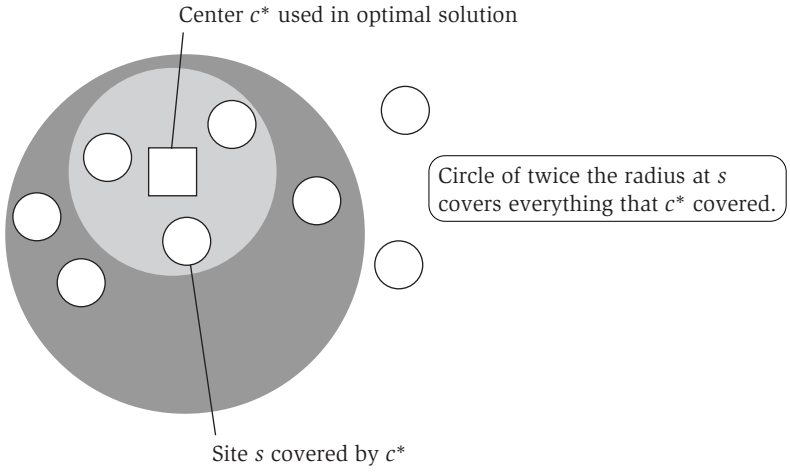

Figure 11.4 Everything covered at radius r by c^* is also covered at radius $2r$ by s.

our job is to find some set of k centers C whose covering radius is not much more than r. It turns out that finding a set of k centers with covering radius at most $2r$ can be done relatively easily.

Here is the idea: We can use the existence of this solution C^* in our algorithm even though we do not know what C^* is. Consider any site $s \in S$. There must be a center $c^* \in C^*$ that covers site s, and this center c^* is at distance at most r from s. Now our idea would be to take this site s as a center in our solution instead of c^*, as we have no idea what c^* is. We would like to make s cover all the sites that c^* covers in the unknown solution C^*. This is accomplished by expanding the radius from r to $2r$. All the sites that were at distance at most r from center c^* are at distance at most $2r$ from s (by the triangle inequality). See Figure 11.4 for a simple illustration of this argument.

S' will represent the sites that still need to be covered
Initialize $S' = S$
Let $C = \emptyset$
While $S' \neq \emptyset$
  Select any site $s \in S'$ and add s to C
  Delete all sites from S' that are at distance at most $2r$ from s
EndWhile
If $|C| \leq k$ then
  Return C as the selected set of sites
Else

```
    Claim (correctly) that there is no set of k centers with
       covering radius at most r
EndIf
```

Clearly, if this algorithm returns a set of at most k centers, then we have what we wanted.

(11.6) *Any set of centers C returned by the algorithm has covering radius $r(C) \leq 2r$.*

Next we argue that if the algorithm fails to return a set of centers, then its conclusion that no set can have covering radius at most r is indeed correct.

(11.7) *Suppose the algorithm selects more than k centers. Then, for any set C^* of size at most k, the covering radius is $r(C^*) > r$.*

Proof. Assume the opposite, that there is a set C^* of at most k centers with covering radius $r(C^*) \leq r$. Each center $c \in C$ selected by the greedy algorithm is one of the original sites in S, and the set C^* has covering radius at most r, so there must be a center $c^* \in C^*$ that is at most a distance of r from c—that is, $dist(c, c^*) \leq r$. Let us say that such a center c^* is *close* to c. We want to claim that no center c^* in the optimal solution C^* can be close to two different centers in the greedy solution C. If we can do this, we are done: each center $c \in C$ has a close optimal center $c^* \in C^*$, and each of these close optimal centers is distinct. This will imply that $|C^*| \geq |C|$, and since $|C| > k$, this will contradict our assumption that C^* contains at most k centers.

So we just need to show that no optimal center $c^* \in C$ can be close to each of two centers $c, c' \in C$. The reason for this is pictured in Figure 11.5. Each pair of centers $c, c' \in C$ is separated by a distance of more than $2r$, so if c^* were within a distance of at most r from each, then this would violate the triangle inequality, since $dist(c, c^*) + dist(c^*, c') \geq dist(c, c') > 2r$. ■

Eliminating the Assumption That We Know the Optimal Radius Now we return to the original question: How do we select a good set of k centers *without* knowing what the optimal covering radius might be?

It is worth discussing two different answers to this question. First, there are many cases in the design of approximation algorithms where it is conceptually useful to assume that you know the value achieved by an optimal solution. In such situations, you can often start with an algorithm designed under this assumption and convert it into one that achieves a comparable performance guarantee by simply trying out a range of "guesses" as to what the optimal

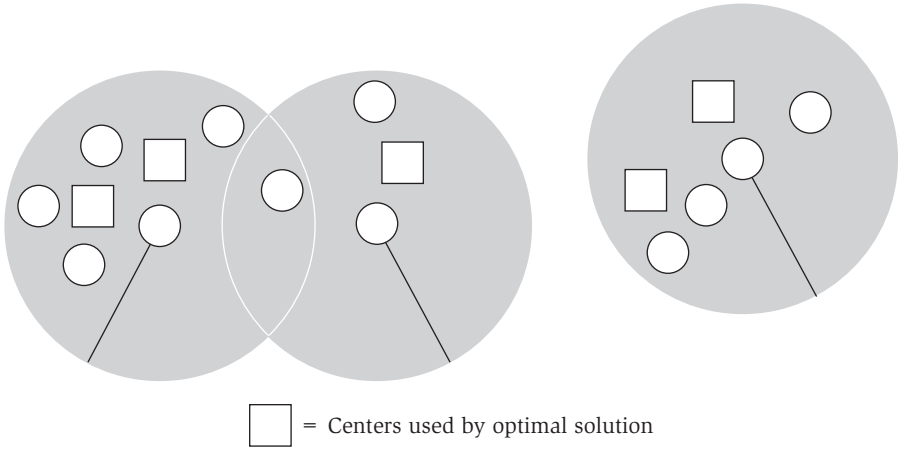

Figure 11.5 The crucial step in the analysis of the greedy algorithm that knows the optimal radius r. No center used by the optimal solution can lie in two different circles, so there must be at least as many optimal centers as there are centers chosen by the greedy algorithm.

value might be. Over the course of the algorithm, this sequence of guesses gets more and more accurate, until an approximate solution is reached.

For the Center Selection Problem, this could work as follows. We can start with some very weak initial guesses about the radius of the optimal solution: We know it is greater than 0, and it is at most the maximum distance r_{max} between any two sites. So we could begin by splitting the difference between these two and running the greedy algorithm we developed above with this value of $r = r_{max}/2$. One of two things will happen, according to the design of the algorithm: Either we find a set of k centers with covering radius at most $2r$, or we conclude that there is no solution with covering radius at most r. In the first case, we can afford to lower our guess on the radius of the optimal solution; in the second case, we need to raise it. This gives us the ability to perform a kind of binary search on the radius: in general, we will iteratively maintain values $r_0 < r_1$ so that we know the optimal radius is greater than r_0, but we have a solution of radius at most $2r_1$. From these values, we can run the above algorithm with radius $r = (r_0 + r_1)/2$; we will either conclude that the optimal solution has radius greater than $r > r_0$, or obtain a solution with radius at most $2r = (r_0 + r_1) < 2r_1$. Either way, we will have sharpened our estimates on one side or the other, just as binary search is supposed to do. We can stop when we have estimates r_0 and r_1 that are close to each other; at this point, our solution of radius $2r_1$ is close to being a 2-approximation to the optimal radius, since we know the optimal radius is greater than r_0 (and hence close to r_1).

A Greedy Algorithm That Works For the specific case of the Center Selection Problem, there is a surprising way to get around the assumption of knowing the radius, without resorting to the general technique described earlier. It turns out we can run essentially the same greedy algorithm developed earlier without knowing anything about the value of r.

The earlier greedy algorithm, armed with knowledge of r, repeatedly selects one of the original sites s as the next center, making sure that it is at least $2r$ away from all previously selected sites. To achieve essentially the same effect without knowing r, we can simply select the site s that is farthest away from all previously selected centers: If there is any site at least $2r$ away from all previously chosen centers, then this farthest site s must be one of them. Here is the resulting algorithm.

```
Assume k ≤ |S| (else define C = S)
Select any site s and let C = {s}
While |C| < k
   Select a site s ∈ S that maximizes dist(s, C)
   Add site s to C
EndWhile
Return C as the selected set of sites
```

(11.8) *This greedy algorithm returns a set C of k points such that $r(C) \leq 2r(C^*)$, where C^* is an optimal set of k points.*

Proof. Let $r = r(C^*)$ denote the minimum possible radius of a set of k centers. For the proof, we assume that we obtain a set of k centers C with $r(C) > 2r$, and from this we derive a contradiction.

So let s be a site that is more than $2r$ away from every center in C. Consider some intermediate iteration in the execution of the algorithm, where we have thus far selected a set of centers C'. Suppose we are adding the center c' in this iteration. We claim that c' is at least $2r$ away from all sites in C'. This follows as site s is more than $2r$ away from all sites in the larger set C, and we select a site c that is the farthest site from all previously selected centers. More formally, we have the following chain of inequalities:

$$dist(c', C') \geq dist(s, C') \geq dist(s, C) > 2r.$$

It follows that our greedy algorithm is a correct implementation of the first k iterations of the `while` loop of the previous algorithm, which knew the optimal radius r: In each iteration, we are adding a center at distance more than $2r$ from all previously selected centers. But the previous algorithm would

have $S' \neq \emptyset$ after selecting k centers, as it would have $s \in S'$, and so it would go on and select more than k centers and eventually conclude that k centers cannot have covering radius at most r. This contradicts our choice of r, and the contradiction proves that $r(C) \leq 2r$. ■

Note the surprising fact that our final greedy 2-approximation algorithm is a very simple modification of the first greedy algorithm that did not work. Perhaps the most important change is simply that our algorithm always selects sites as centers (i.e., every mall will be built in one of the little towns and not halfway between two of them).

11.3 Set Cover: A General Greedy Heuristic

In this section we will consider a very general problem that we also encountered in Chapter 8, the Set Cover Problem. A number of important algorithmic problems can be formulated as special cases of Set Cover, and hence an approximation algorithm for this problem will be widely applicable. We will see that it is possible to design a greedy algorithm here that produces solutions with a guaranteed approximation factor relative to the optimum, although this factor will be weaker than what we saw for the problems in Sections 11.1 and 11.2.

While the greedy algorithm we design for Set Cover will be very simple, the analysis will be more complex than what we encountered in the previous two sections. There we were able to get by with very simple bounds on the (unknown) optimum solution, while here the task of comparing to the optimum is more difficult, and we will need to use more sophisticated bounds. This aspect of the method can be viewed as our first example of the pricing method, which we will explore more fully in the next two sections.

The Problem

Recall from our discussion of NP-completeness that the Set Cover Problem is based on a set U of n elements and a list $S_1, \ldots, S_m$ of subsets of U; we say that a *set cover* is a collection of these sets whose union is equal to all of U.

In the version of the problem we consider here, each set S_i has an associated *weight* $w_i \geq 0$. The goal is to find a set cover $\mathcal{C}$ so that the total weight

$$\sum_{S_i \in \mathcal{C}} w_i$$

is minimized. Note that this problem is at least as hard as the decision version of Set Cover we encountered earlier; if we set all $w_i = 1$, then the minimum

weight of a set cover is at most k if and only if there is a collection of at most k sets that covers U.

Designing the Algorithm

We will develop and analyze a greedy algorithm for this problem. The algorithm will have the property that it builds the cover one set at a time; to choose its next set, it looks for one that seems to make the most progress toward the goal. What is a natural way to define "progress" in this setting? Desirable sets have two properties: They have small weight w_i, and they cover lots of elements. Neither of these properties alone, however, would be enough for designing a good approximation algorithm. Instead, it is natural to combine these two criteria into the single measure $w_i/|S_i|$—that is, by selecting S_i, we cover $|S_i|$ elements at a cost of w_i, and so this ratio gives the "cost per element covered," a very reasonable thing to use as a guide.

Of course, once some sets have already been selected, we are only concerned with how we are doing on the elements still left uncovered. So we will maintain the set R of remaining uncovered elements and choose the set S_i that minimizes $w_i/|S_i \cap R|$.

```
Greedy-Set-Cover:
Start with R = U and no sets selected
While R ≠ ∅
  Select set S_i that minimizes w_i/|S_i ∩ R|
  Delete set S_i from R
EndWhile
Return the selected sets
```

As an example of the behavior of this algorithm, consider what it would do on the instance in Figure 11.6. It would first choose the set containing the four nodes at the bottom (since this has the best weight-to-coverage ratio, 1/4). It then chooses the set containing the two nodes in the second row, and finally it chooses the sets containing the two individual nodes at the top. It thereby chooses a collection of sets of total weight 4. Because it myopically chooses the best option each time, this algorithm misses the fact that there's a way to cover everything using a weight of just $2 + 2\varepsilon$, by selecting the two sets that each cover a full column.

Analyzing the Algorithm

The sets selected by the algorithm clearly form a set cover. The question we want to address is: How much larger is the weight of this set cover than the weight w^* of an optimal set cover?

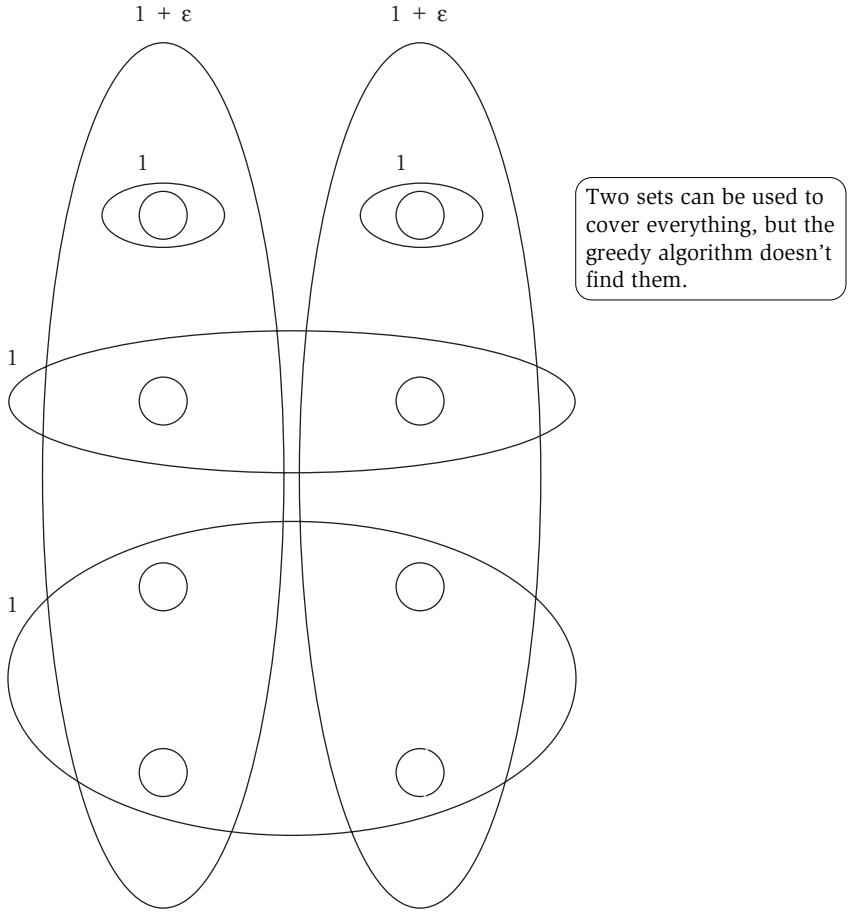

Figure 11.6 An instance of the Set Cover Problem where the weights of sets are either 1 or $1+\varepsilon$ for some small $\varepsilon > 0$. The greedy algorithm chooses sets of total weight 4, rather than the optimal solution of weight $2 + 2\varepsilon$.

As in Sections 11.1 and 11.2, our analysis will require a good lower bound on this optimum. In the case of the Load Balancing Problem, we used lower bounds that emerged naturally from the statement of the problem: the average load, and the maximum job size. The Set Cover Problem will turn out to be more subtle; "simple" lower bounds are not very useful, and instead we will use a lower bound that the greedy algorithm implicitly constructs as a by-product.

Recall the intuitive meaning of the ratio $w_i/|S_i \cap R|$ used by the algorithm; it is the "cost paid" for covering each new element. Let's record this cost paid for

element s in the quantity c_s. We add the following line to the code immediately after selecting the set S_i.

`Define` $c_s = w_i/|S_i \cap R|$ `for all` $s \in S_i \cap R$

The values c_s do not affect the behavior of the algorithm at all; we view them as a bookkeeping device to help in our comparison with the optimum w^*. As each set S_i is selected, its weight is distributed over the costs c_s of the elements that are newly covered. Thus these costs completely account for the total weight of the set cover, and so we have

(11.9) *If $\mathcal{C}$ is the set cover obtained by* `Greedy-Set-Cover`*, then* $\sum_{S_i \in \mathcal{C}} w_i = \sum_{s \in U} c_s$.

The key to the analysis is to ask how much total cost any single set S_k can account for—in other words, to give a bound on $\sum_{s \in S_k} c_s$ relative to the weight w_k of the set, even for sets not selected by the greedy algorithm. Giving an upper bound on the ratio

$$\frac{\sum_{s \in S_k} c_s}{w_k}$$

that holds for every set says, in effect, "To cover a lot of cost, you must use a lot of weight." We know that the optimum solution must cover the full cost $\sum_{s \in U} c_s$ via the sets it selects; so this type of bound will establish that it needs to use at least a certain amount of weight. This is a lower bound on the optimum, just as we need for the analysis.

Our analysis will use the *harmonic function*

$$H(n) = \sum_{i=1}^{n} \frac{1}{i}.$$

To understand its asymptotic size as a function of n, we can interpret it as a sum approximating the area under the curve $y = 1/x$. Figure 11.7 shows how it is naturally bounded above by $1 + \int_1^n \frac{1}{x}\,dx = 1 + \ln n$, and bounded below by $\int_1^{n+1} \frac{1}{x}\,dx = \ln(n+1)$. Thus we see that $H(n) = \Theta(\ln n)$.

Here is the key to establishing a bound on the performance of the algorithm.

(11.10) *For every set S_k, the sum $\sum_{s \in S_k} c_s$ is at most $H(|S_k|) \cdot w_k$.*

Proof. To simplify the notation, we will assume that the elements of S_k are the first $d = |S_k|$ elements of the set U; that is, $S_k = \{s_1, \ldots, s_d\}$. Furthermore, let us assume that these elements are labeled in the order in which they are assigned a cost c_{s_j} by the greedy algorithm (with ties broken arbitrarily). There

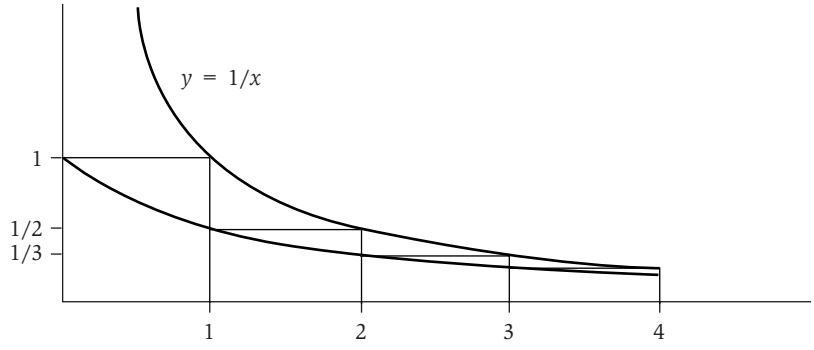

Figure 11.7 Upper and lower bounds for the Harmonic Function $H(n)$.

is no loss of generality in doing this, since it simply involves a renaming of the elements in U.

Now consider the iteration in which element s_j is covered by the greedy algorithm, for some $j \leq d$. At the start of this iteration, $s_j, s_{j+1}, \ldots, s_d \in R$ by our labeling of the elements. This implies that $|S_k \cap R|$ is at least $d - j + 1$, and so the average cost of the set S_k is at most

$$\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}.$$

Note that this is not necessarily an equality, since s_j may be covered in the same iteration as some of the other elements $s_{j'}$ for $j' < j$. In this iteration, the greedy algorithm selected a set S_i of minimum average cost; so this set S_i has average cost at most that of S_k. It is the average cost of S_i that gets assigned to s_j, and so we have

$$c_{s_j} = \frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}.$$

We now simply add up these inequalities for all elements $s \in S_k$:

$$\sum_{s \in S_k} c_s = \sum_{j=1}^{d} c_{s_j} \leq \sum_{j=1}^{d} \frac{w_k}{d - j + 1} = \frac{w_k}{d} + \frac{w_k}{d-1} + \ldots + \frac{w_k}{1} = H(d) \cdot w_k. \quad \blacksquare$$

We now complete our plan to use the bound in (11.10) for comparing the greedy algorithm's set cover to the optimal one. Letting $d^* = \max_i |S_i|$ denote the maximum size of any set, we have the following approximation result.

(11.11) *The set cover $\mathcal{C}$ selected by* `Greedy-Set-Cover` *has weight at most $H(d^*)$ times the optimal weight w^*.*

Proof. Let $\mathcal{C}^*$ denote the optimum set cover, so that $w^* = \sum_{S_i \in \mathcal{C}^*} w_i$. For each of the sets in $\mathcal{C}^*$, (11.10) implies

$$w_i \geq \frac{1}{H(d^*)} \sum_{s \in S_i} c_s.$$

Because these sets form a set cover, we have

$$\sum_{S_i \in \mathcal{C}^*} \sum_{s \in S_i} c_s \geq \sum_{s \in U} c_s.$$

Combining these with (11.9), we obtain the desired bound:

$$w^* = \sum_{S_i \in \mathcal{C}^*} w_i \geq \sum_{S_i \in \mathcal{C}^*} \frac{1}{H(d^*)} \sum_{s \in S_i} c_s \geq \frac{1}{H(d^*)} \sum_{s \in U} c_s = \frac{1}{H(d^*)} \sum_{S_i \in \mathcal{C}} w_i. \quad \blacksquare$$

Asymptotically, then, the bound in (11.11) says that the greedy algorithm finds a solution within a factor $O(\log d^*)$ of optimal. Since the maximum set size d^* can be a constant fraction of the total number of elements n, this is a worst-case upper bound of $O(\log n)$. However, expressing the bound in terms of d^* shows us that we're doing much better if the largest set is small.

It's interesting to note that this bound is essentially the best one possible, since there are instances where the greedy algorithm can do this badly. To see how such instances arise, consider again the example in Figure 11.6. Now suppose we generalize this so that the underlying set of elements U consists of two tall columns with $n/2$ elements each. There are still two sets, each of weight $1 + \varepsilon$, for some small $\varepsilon > 0$, that cover the columns separately. We also create $\Theta(\log n)$ sets that generalize the structure of the other sets in the figure: there is a set that covers the bottommost $n/2$ nodes, another that covers the next $n/4$, another that covers the next $n/8$, and so forth. Each of these sets will have weight 1.

Now the greedy algorithm will choose the sets of size $n/2, n/4, n/8, \ldots$, in the process producing a solution of weight $\Omega(\log n)$. Choosing the two sets that cover the columns separately, on the other hand, yields the optimal solution, with weight $2 + 2\varepsilon$. Through more complicated constructions, one can strengthen this to produce instances where the greedy algorithm incurs a weight that is very close to $H(n)$ times the optimal weight. And in fact, by much more complicated means, it has been shown that no polynomial-time approximation algorithm can achieve an approximation bound much better than $H(n)$ times optimal, unless $P = NP$.

11.4 The Pricing Method: Vertex Cover

We now turn to our second general technique for designing approximation algorithms, the *pricing method*. We will introduce this technique by considering a version of the Vertex Cover Problem. As we saw in Chapter 8, Vertex Cover is in fact a special case of Set Cover, and so we will begin this section by considering the extent to which one can use reductions in the design of approximation algorithms. Following this, we will develop an algorithm with a better approximation guarantee than the general bound that we obtained for Set Cover in the previous section.

The Problem

Recall that a *vertex cover* in a graph $G = (V, E)$ is a set $S \subseteq V$ so that each edge has at least one end in S. In the version of the problem we consider here, each vertex $i \in V$ has a *weight* $w_i \geq 0$, with the weight of a set S of vertices denoted $w(S) = \sum_{i \in S} w_i$. We would like to find a vertex cover S for which $w(S)$ is minimum. When all weights are equal to 1, deciding if there is a vertex cover of weight at most k is the standard decision version of Vertex Cover.

Approximations via Reductions? Before we work on developing an algorithm, we pause to discuss an interesting issue that arises: Vertex Cover is easily reducible to Set Cover, and we have just seen an approximation algorithm for Set Cover. What does this imply about the approximability of Vertex Cover? A discussion of this question brings out some of the subtle ways in which approximation results interact with polynomial-time reductions.

First consider the special case in which all weights are equal to 1—that is, we are looking for a vertex cover of minimum size. We will call this the *unweighted case*. Recall that we showed Set Cover to be NP-complete using a reduction from the decision version of unweighted Vertex Cover. That is,

$$\text{Vertex Cover} \leq_p \text{Set Cover}$$

This reduction says, "If we had a polynomial-time algorithm that solves the Set Cover Problem, then we could use this algorithm to solve the Vertex Cover Problem in polynomial time." We now have a polynomial-time algorithm for the Set Cover Problem that approximates the solution. Does this imply that we can use it to formulate an approximation algorithm for Vertex Cover?

(11.12) *One can use the Set Cover approximation algorithm to give an $H(d)$-approximation algorithm for the weighted Vertex Cover Problem, where d is the maximum degree of the graph.*

Proof. The proof is based on the reduction that showed Vertex Cover $\leq_p$ Set Cover, which also extends to the weighted case. Consider an instance of the weighted Vertex Cover Problem, specified by a graph $G = (V, E)$. We define an

instance of Set Cover as follows. The underlying set U is equal to E. For each node i, we define a set S_i consisting of all edges incident to node i and give this set weight w_i. Collections of sets that cover U now correspond precisely to vertex covers. Note that the maximum size of any S_i is precisely the maximum degree d.

Hence we can use the approximation algorithm for Set Cover to find a vertex cover whose weight is within a factor of $H(d)$ of minimum. ■

This $H(d)$-approximation is quite good when d is small; but it gets worse as d gets larger, approaching a bound that is logarithmic in the number of vertices. In the following, we will develop a stronger approximation algorithm that comes within a factor of 2 of optimal.

Before turning to the 2-approximation algorithm, we make the following further observation: One has to be very careful when trying to use reductions for designing approximation algorithms. It worked in (11.12), but we made sure to go through an argument for why it worked; it is not the case that every polynomial-time reduction leads to a comparable implication for approximation algorithms.

Here is a cautionary example. We used Independent Set to prove that the Vertex Cover Problem is NP-complete. Specifically, we proved

$$\text{Independent Set} \leq_P \text{Vertex Cover},$$

which states that "if we had a polynomial-time algorithm that solves the Vertex Cover Problem, then we could use this algorithm to solve the Independent Set Problem in polynomial time." Can we use an approximation algorithm for the minimum-size vertex cover to design a comparably good approximation algorithm for the maximum-size independent set?

The answer is no. Recall that a set I of vertices is independent if and only if its complement $S = V - I$ is a vertex cover. Given a minimum-size vertex cover S^*, we obtain a maximum-size independent set by taking the complement $I^* = V - S$. Now suppose we use an approximation algorithm for the Vertex Cover Problem to get an approximately minimum vertex cover S. The complement $I = V - S$ is indeed an independent set—there's no problem there. The trouble is when we try to determine our approximation factor for the Independent Set Problem; I can be very far from optimal. Suppose, for example, that the optimal vertex cover S^* and the optimal independent set I^* both have size $|V|/2$. If we invoke a 2-approximation algorithm for the Vertex Cover Problem, we may perfectly well get back the set $S = V$. But, in this case, our "approximately maximum independent set" $I = V - S$ has no elements.

Designing the Algorithm: The Pricing Method

Even though (11.12) gave us an approximation algorithm with a provable guarantee, we will be able to do better. Our approach forms a nice illustration of the *pricing method* for designing approximation algorithms.

The Pricing Method to Minimize Cost The pricing method (also known as the *primal-dual method*) is motivated by an economic perspective. For the case of the Vertex Cover Problem, we will think of the weights on the nodes as *costs*, and we will think of each edge as having to pay for its "share" of the cost of the vertex cover we find. We have actually just seen an analysis of this sort, in the greedy algorithm for Set Cover from Section 11.3; it too can be thought of as a pricing algorithm. The greedy algorithm for Set Cover defined values c_s, the cost the algorithm paid for covering element s. We can think of c_s as the element s's "share" of the cost. Statement (11.9) shows that it is very natural to think of the values c_s as cost-shares, as the sum of the cost-shares $\sum_{s \in U} c_s$ is the cost of the set cover $\mathcal{C}$ returned by the algorithm, $\sum_{S_i \in \mathcal{C}} w_i$. The key to proving that the algorithm is an $H(d^*)$-approximation algorithm was a certain approximate "fairness" property for the cost-shares: (11.10) shows that the elements in a set S_k are charged by at most an $H(|S_k|)$ factor more than the cost of covering them by the set S_k.

In this section, we'll develop the pricing technique through another application, Vertex Cover. Again, we will think of the weight w_i of the vertex i as the cost for using i in the cover. We will think of each edge e as a separate "agent" who is willing to "pay" something to the node that covers it. The algorithm will not only find a vertex cover S, but also determine prices $p_e \geq 0$ for each edge $e \in E$, so that if each edge $e \in E$ pays the price p_e, this will in total approximately cover the cost of S. These prices p_e are the analogues of c_s from the Set Cover Algorithm.

Thinking of the edges as agents suggests some natural fairness rules for prices, analogous to the property proved by (11.10). First of all, selecting a vertex i covers all edges incident to i, so it would be "unfair" to charge these incident edges in total more than the cost of vertex i. We call prices p_e *fair* if, for each vertex i, the edges adjacent to i do not have to pay more than the cost of the vertex: $\sum_{e=(i,j)} p_e \leq w_i$. Note that the property proved by (11.10) for Set Cover is an approximate fairness condition, while in the Vertex Cover algorithm we'll actually use the exact fairness defined here. A useful fact about fair prices is that they provide a lower bound on the cost of any solution.

(11.13) *For any vertex cover S^*, and any nonnegative and fair prices p_e, we have $\sum_{e \in E} p_e \leq w(S^*)$.*

Proof. Consider a vertex cover S^*. By the definition of fairness, we have $\sum_{e=(i,j)} p_e \leq w_i$ for all nodes $i \in S^*$. Adding these inequalities over all nodes in S^*, we get

$$\sum_{i \in S^*} \sum_{e=(i,j)} p_e \leq \sum_{i \in S^*} w_i = w(S^*).$$

Now the expression on the left-hand side is a sum of terms, each of which is some edge price p_e. Since S^* is a vertex cover, each edge e contributes at least one term p_e to the left-hand side. It may contribute more than one copy of p_e to this sum, since it may be covered from both ends by S^*; but the prices are nonnegative, and so the sum on the left-hand side is at least as large as the sum of all prices p_e. That is,

$$\sum_{e \in E} p_e \leq \sum_{i \in S^*} \sum_{e=(i,j)} p_e.$$

Combining this with the previous inequality, we get

$$\sum_{e \in E} p_e \leq w(S^*),$$

as desired. ■

The Algorithm The goal of the approximation algorithm will be to find a vertex cover and to set prices at the same time. We can think of the algorithm as being greedy in how it sets the prices. It then uses these prices to drive the way it selects nodes for the vertex cover.

We say that a node i is *tight* (or "paid for") if $\sum_{e=(i,j)} p_e = w_i$.

```
Vertex-Cover-Approx(G, w):
  Set p_e = 0 for all e ∈ E
  While there is an edge e = (i, j) such that neither i nor j is tight
     Select such an edge e
     Increase p_e without violating fairness
  EndWhile
  Let S be the set of all tight nodes
  Return S
```

For example, consider the execution of this algorithm on the instance in Figure 11.8. Initially, no node is tight; the algorithm decides to select the edge (a, b). It can raise the price paid by (a, b) up to 3, at which point the node b becomes tight and it stops. The algorithm then selects the edge (a, d). It can only raise this price up to 1, since at this point the node a becomes tight (due to the fact that the weight of a is 4, and it is already incident to an edge that is

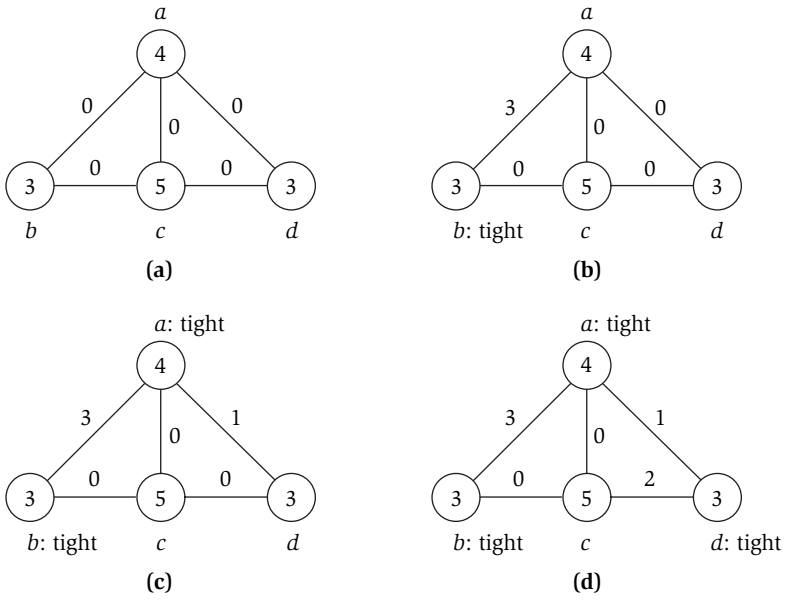

Figure 11.8 Parts (a)–(d) depict the steps in an execution of the pricing algorithm on an instance of the weighted Vertex Cover Problem. The numbers inside the nodes indicate their weights; the numbers annotating the edges indicate the prices they pay as the algorithm proceeds.

paying 3). Finally, the algorithm selects the edge (c, d). It can raise the price paid by (c, d) up to 2, at which point d becomes tight. We now have a situation where all edges have at least one tight end, so the algorithm terminates. The tight nodes are a, b, and d; so this is the resulting vertex cover. (Note that this is not the minimum-weight vertex cover; that would be obtained by selecting a and c.)

Analyzing the Algorithm

At first sight, one may have the sense that the vertex cover S is fully paid for by the prices: all nodes in S are tight, and hence the edges adjacent to the node i in S can pay for the cost of i. But the point is that an edge e can be adjacent to more than one node in the vertex cover (i.e., if both ends of e are in the vertex cover), and hence e may have to pay for more than one node. This is the case, for example, with the edges (a, b) and (a, d) at the end of the execution in Figure 11.8.

However, notice that if we take edges for which both ends happened to show up in the vertex cover, and we charge them their price twice, then we're exactly paying for the vertex cover. (In the example, the cost of the vertex

cover is the cost of nodes a, b, and d, which is 10. We can account for this cost exactly by charging (a, b) and (a, d) twice, and (c, d) once.) Now, it's true that this is unfair to some edges, but the amount of unfairness can be bounded: Each edge gets charged its price at most two times (once for each end).

We now make this argument precise, as follows.

(11.14) *The set S and prices p returned by the algorithm satisfy the inequality $w(S) \leq 2 \sum_{e \in E} p_e$.*

Proof. All nodes in S are tight, so we have $\sum_{e=(i,j)} p_e = w_i$ for all $i \in S$. Adding over all nodes in S we get

$$w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} p_e.$$

An edge $e = (i, j)$ can be included in the sum on the right-hand side at most twice (if both i and j are in S), and so we get

$$w(S) = \sum_{i \in S} \sum_{e=(i,j)} p_e \leq 2 \sum_{e \in E} p_e,$$

as claimed. ■

Finally, this factor of 2 carries into an argument that yields the approximation guarantee.

(11.15) *The set S returned by the algorithm is a vertex cover, and its cost is at most twice the minimum cost of any vertex cover.*

Proof. First note that S is indeed a vertex cover. Suppose, by contradiction, that S does not cover edge $e = (i, j)$. This implies that neither i nor j is tight, and this contradicts the fact that the `While` loop of the algorithm terminated.

To get the claimed approximation bound, we simply put together statement (11.14) with (11.13). Let p be the prices set by the algorithm, and let S^* be an optimal vertex cover. By (11.14) we have $2 \sum_{e \in E} p_e \geq w(S)$, and by (11.13) we have $\sum_{e \in E} p_e \leq w(S^*)$.

In other words, the sum of the edge prices is a lower bound on the weight of *any* vertex cover, and twice the sum of the edge prices is an upper bound on the weight of our vertex cover:

$$w(S) \leq 2 \sum_{e \in E} p_e \leq 2w(S^*). \quad ■$$

11.5 Maximization via the Pricing Method: The Disjoint Paths Problem

We now continue the theme of pricing algorithms with a fundamental problem that arises in network routing: the *Disjoint Paths Problem*. We'll start out by developing a greedy algorithm for this problem and then show an improved algorithm based on pricing.

The Problem

To set up the problem, it helps to recall one of the first applications we saw for the Maximum-Flow Problem: finding disjoint paths in graphs, which we discussed in Chapter 7. There we were looking for edge-disjoint paths all starting at a node s and ending at a node t. How crucial is it to the tractability of this problem that all paths have to start and end at the same node? Using the technique from Section 7.7, one can extend this to find disjoint paths where we are given a set of start nodes S and a set of terminals T, and the goal is to find edge-disjoint paths where paths may start at any node in S and end at any node in T.

Here, however, we will look at a case where each path to be routed has its own designated starting node and ending node. Specifically, we consider the following *Maximum Disjoint Paths Problem*. We are given a directed graph G, together with k pairs of nodes $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ and an integer capacity c. We think of each pair (s_i, t_i) as a *routing request*, which asks for a path from s_i to t_i. A solution to this instance consists of a subset of the requests we will satisfy, $I \subseteq \{1, \ldots, k\}$, together with paths that satisfy them while not overloading any one edge: a path P_i for $i \in I$ so that P_i goes from s_i to t_i, and each edge is used by at most c paths. The problem is to find a solution with $|I|$ as large as possible—that is, to satisfy as many requests as possible. Note that the capacity c controls how much "sharing" of edges we allow; when $c = 1$, we are requiring the paths to be fully edge-disjoint, while larger c allows some overlap among the paths.

We have seen in Exercise 39 in Chapter 8 that it is NP-complete to determine whether all k routing requests can be satisfied when the paths are required to be node-disjoint. It is not hard to show that the edge-disjoint version of the problem (corresponding to the case with $c = 1$) is also NP-complete.

Thus it turns out to have been crucial for the application of efficient network flow algorithms that the endpoints of the paths not be explicitly paired up as they are in Maximum Disjoint Paths. To develop this point a little further, suppose we attempted to reduce Maximum Disjoint Paths to a network flow problem by defining the set of sources to be $S = \{s_1, s_2, \ldots, s_k\}$, defining the

set of sinks to be $T = \{t_1, t_2, \ldots, t_k\}$, setting each edge capacity to be c, and looking for the maximum possible number of disjoint paths starting in S and ending in T. Why wouldn't this work? The problem is that there's no way to tell the flow algorithm that a path starting at $s_i \in S$ *must* end at $t_i \in T$; the algorithm guarantees only that this path will end at *some* node in T. As a result, the paths that come out of the flow algorithm may well not constitute a solution to the instance of Maximum Disjoint Paths, since they might not link a source s_i to its corresponding endpoint t_i.

Disjoint paths problems, where we need to find paths connecting designated pairs of terminal nodes, are very common in networking applications. Just think about paths on the Internet that carry streaming media or Web data, or paths through the phone network carrying voice traffic.[1] Paths sharing edges can interfere with each other, and too many paths sharing a single edge will cause problems in most applications. The maximum allowable amount of sharing will differ from application to application. Requiring the paths to be disjoint is the strongest constraint, eliminating all interference between paths. We'll see, however, that in cases where some sharing is allowed (even just two paths to an edge), better approximation algorithms are possible.

Designing and Analyzing a Greedy Algorithm

We first consider a very simple algorithm for the case when the capacity $c = 1$: that is, when the paths need to be edge-disjoint. The algorithm is essentially greedy, except that it exhibits a preference for short paths. We will show that this simple algorithm is an $O(\sqrt{m})$-approximation algorithm, where $m = |E|$ is the number of edges in G. This may sound like a rather large factor of approximation, and it is, but there is a strong sense in which it is essentially the best we can do. The Maximum Disjoint Paths Problem is not only NP-complete, but it is also hard to approximate: It has been shown that unless $\mathcal{P} = \mathcal{NP}$, it is impossible for any polynomial-time algorithm to achieve an approximation bound significantly better than $O(\sqrt{m})$ in arbitrary directed graphs.

After developing the greedy algorithm, we will consider a slightly more sophisticated pricing algorithm for the capacitated version. It is interesting

[1] A researcher from the telecommunications industry once gave the following explanation for the distinction between Maximum Disjoint Paths and network flow, and the broken reduction in the previous paragraph. On Mother's Day, traditionally the busiest day of the year for telephone calls, the phone company must solve an enormous disjoint paths problem: ensuring that each source individual s_i is connected by a path through the voice network to his or her mother t_i. Network flow algorithms, finding disjoint paths between a set S and a set T, on the other hand, will ensure only that each person gets their call through to *somebody's* mother.

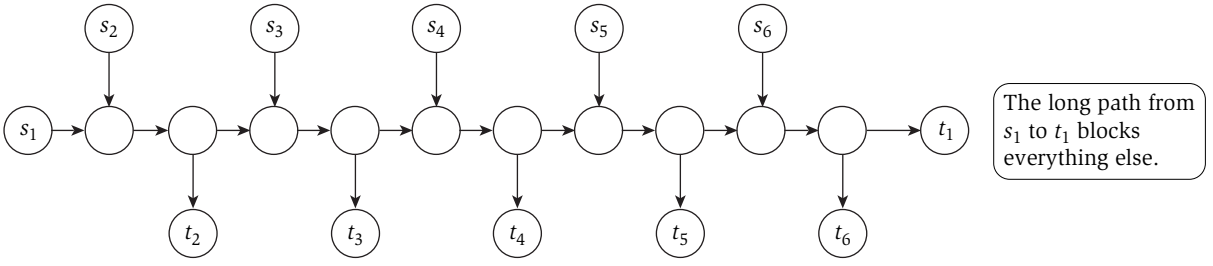

Figure 11.9 A case in which it's crucial that a greedy algorithm for selecting disjoint paths favors short paths over long ones.

to note that the pricing algorithm does much better than the simple greedy algorithm, even when the capacity c is only slightly more than 1.

```
Greedy-Disjoint-Paths:
Set I = ∅
Until no new path can be found
  Let P_i be the shortest path (if one exists) that is edge-disjoint
      from previously selected paths, and connects some (s_i, t_i) pair
      that is not yet connected
  Add i to I and select path P_i to connect s_i to t_i
EndUntil
```

Analyzing the Algorithm The algorithm clearly selects edge-disjoint paths. Assuming the graph G is connected, it must select at least one path. But how does the number of paths selected compare with the maximum possible? A kind of situation we need to worry about is shown in Figure 11.9: One of the paths, from s_1 to t_1, is very long, so if we select it first, we eliminate up to $\Omega(m)$ other paths.

We now show that the greedy algorithm's preference for short paths not only avoids the problem in this example, but in general it limits the number of other paths that a selected path can interfere with.

(11.16) *The algorithm* `Greedy-Disjoint-Paths` *is a* $(2\sqrt{m}+1)$*-approximation algorithm for the Maximum Disjoint Paths Problem.*

Proof. Consider an optimal solution: Let I^* be the set of pairs for which a path was selected in this optimum solution, and let P_i^* for $i \in I^*$ be the selected paths. Let I denote the set of pairs returned by the algorithm, and let P_i for $i \in I$ be the corresponding paths. We need to bound $|I^*|$ in terms of $|I|$. The key to the analysis is to make a distinction between short and long paths and to consider

them separately. We will call a path *long* if it has at least $\sqrt{m}$ edges, and we will call it *short* otherwise. Let I_s^* denote the set of indices in I^* so that the corresponding path P_i^* is short, and let I_s denote the set of indices in I so that the corresponding path P_i is short.

The graph G has m edges, and each long path uses at least $\sqrt{m}$ edges, so there can be at most $\sqrt{m}$ long paths in I^*.

Now consider the short paths in I^*. In order for I^* to be much larger than I, there would have to be many pairs that are connected in I^* but not in I. Thus let us consider pairs that are connected by the optimum using a short path, but are not connected by the greedy algorithm. Since the path P_i^* connecting s_i and t_i in the optimal solution I^* is short, the greedy algorithm would have selected this path, if it had been available, before selecting any long paths. But the greedy algorithm did not connect s_i and t_i at all, and hence one of the edges e along the path P_i^* must occur in a path P_j that was selected earlier by the greedy algorithm. We will say that edge e *blocks* the path P_i^*.

Now the lengths of the paths selected by the greedy algorithm are monotone increasing, since each iteration has fewer options for choosing paths. The path P_j was selected before considering P_i^* and hence it must be shorter: $|P_j| \leq |P_i^*| \leq \sqrt{m}$. So path P_j is short. Since the paths used by the optimum are edge-disjoint, each edge in a path P_j can block at most one path P_i^*. It follows that each short path P_j blocks at most $\sqrt{m}$ paths in the optimal solution, and so we get the bound

$$|I_s^* - I| \leq \sum_{j \in I_s} |P_j| \leq |I_s|\sqrt{m}.$$

We use this to produce a bound on the overall size of the optimal solution. To do this, we view I^* as consisting of three kinds of paths, following the analysis thus far:

- long paths, of which there are at most $\sqrt{m}$;
- paths that are also in I; and
- short paths that are not in I, which we have just bounded by $|I_s|\sqrt{m}$.

Putting this all together, and using the fact that $|I| \geq 1$ whenever at least one set of terminal pairs can be connected, we get the claimed bound:

$$|I^*| \leq \sqrt{m} + |I| + |I_s^* - I| \leq \sqrt{m} + |I| + \sqrt{m}|I_s| \leq (2\sqrt{m} + 1)|I|. \quad \blacksquare$$

This provides an approximation algorithm for the case when the selected paths have to be disjoint. As we mentioned earlier, the approximation bound of $O(\sqrt{m})$ is rather weak, but unless $\mathcal{P} = \mathcal{NP}$, it is essentially the best possible for the case of disjoint paths in arbitrary directed graphs.

Designing and Analyzing a Pricing Algorithm

Not letting any two paths use the same edge is quite extreme; in most applications one can allow a few paths to share an edge. We will now develop an analogous algorithm, based on the pricing method, for the case where $c > 1$ paths may share any edge. In the disjoint case just considered, we viewed all edges as equal and preferred short paths. We can think of this as a simple kind of pricing algorithm: the paths have to pay for using up the edges, and each edge has a unit cost. Here we will consider a pricing scheme in which edges are viewed as more expensive if they have been used already, and hence have less capacity left over. This will encourage the algorithm to "spread out" its paths, rather than piling them up on any single edge. We will refer to the cost of an edge e as its *length* ℓ_e, and define the *length* of a path to be the sum of the lengths of the edges it contains: $\ell(P) = \sum_{e \in P} \ell_e$. We will use a multiplicative parameter β to increase the length of an edge each time an additional path uses it.

```
Greedy-Paths-with-Capacity:
Set I = ∅
Set edge length ℓ_e = 1 for all e ∈ E
Until no new path can be found
  Let P_i be the shortest path (if one exists) so that adding P_i to
      the selected set of paths does not use any edge more than c
      times, and P_i connects some (s_i, t_i) pair not yet connected
  Add i to I and select path P_i to connect s_i to t_i
  Multiply the length of all edges along P_i by β
EndUntil
```

Analyzing the Algorithm For the analysis we will focus on the simplest case, when at most two paths may use the same edge—that is, when $c = 2$. We'll see that, for this case, setting $\beta = m^{1/3}$ will give the best approximation result for this algorithm. Unlike the disjoint paths case (when $c = 1$), it is not known whether the approximation bounds we obtain here for $c > 1$ are close to the best possible for polynomial-time algorithms in general, assuming $\mathcal{P} \neq \mathcal{NP}$.

The key to the analysis in the disjoint case was to distinguish "short" and "long" paths. For the case when $c = 2$, we will consider a path P_i selected by the algorithm to be *short* if the length is less than β^2. Let I_s denote the set of short paths selected by the algorithm.

Next we want to compare the number of paths selected with the maximum possible. Let I^* be an optimal solution and P_i^* be the set of paths used in this solution. As before, the key to the analysis is to consider the edges that block

the selection of paths in I^*. Long paths can block a lot of other paths, so for now we will focus on the short paths in I_s. As we try to continue following what we did in the disjoint case, we immediately run into a difficulty, however. In that case, the length of a path in I^* was simply the number of edges it contained; but here, the lengths are changing as the algorithm runs, and so it is not clear how to define the length of a path in I^* for purposes of the analysis. In other words, for the analysis, when should we measure this length? (At the beginning of the execution? At the end?)

It turns out that the crucial moment in the algorithm, for purposes of our analysis, is the first point at which there are no short paths left to choose. Let $\bar{\ell}$ be the length function at this point in the execution of the algorithm; we'll use $\bar{\ell}$ to measure the length of paths in I^*. For a path P, we use $\bar{\ell}(P)$ to denote its length, $\sum_{e \in P} \bar{\ell}_e$. We consider a path P_i^* in the optimal solution I^* *short* if $\bar{\ell}(P_i^*) < \beta^2$, and *long* otherwise. Let I_s^* denote the set of short paths in I^*. The first step is to show that there are no short paths connecting pairs that are not connected by the approximation algorithm.

(11.17) *Consider a source-sink pair $i \in I^*$ that is not connected by the approximation algorithm; that is, $i \notin I$. Then $\bar{\ell}(P_i^*) \geq \beta^2$.*

Proof. As long as short paths are being selected, we do not have to worry about explicitly enforcing the requirement that each edge be used by at most $c = 2$ paths: any edge e considered for selection by a third path would already have length $\ell_e = \beta^2$, and hence be long.

Consider the state of the algorithm with length $\bar{\ell}$. By the argument in the previous paragraph, we can imagine the algorithm having run up to this point without caring about the limit of c; it just selected a short path whenever it could find one. Since the endpoints s_i, t_i of P_i^* are not connected by the greedy algorithm, and since there are no short paths left when the length function reaches $\bar{\ell}$, it must be the case that path P_i^* has length at least β^2 as measured by $\bar{\ell}$. ■

The analysis in the disjoint case used the fact that there are only m edges to limit the number of long paths. Here we consider length $\bar{\ell}$, rather than the number of edges, as the quantity that is being consumed by paths. Hence, to be able to reason about this, we will need a bound on the total length in the graph $\sum_e \bar{\ell}_e$. The sum of the lengths over all edges $\sum_e \ell_e$ starts out at m (length 1 for each edge). Adding a short path to the solution I_s can increase the length by at most β^3, as the selected path has length at most β^2, and the lengths of the edges are increased by a β factor along the path. This gives us a useful comparison between the number of short paths selected and the total length.

(11.18) *The set I_s of short paths selected by the approximation algorithm, and the lengths $\bar{\ell}$, satisfy the relation $\sum_e \bar{\ell}_e \leq \beta^3 |I_s| + m$.*

Finally, we prove an approximation bound for this algorithm. We will find that even though we have simply increased the number of paths allowed on each edge from 1 to 2, the approximation guarantee drops by a significant amount that essentially incorporates this change into the exponent: from $O(m^{1/2})$ down to $O(m^{1/3})$.

(11.19) *The algorithm* `Greedy-Paths-with-Capacity`*, using $\beta = m^{1/3}$, is a $(4m^{1/3} + 1)$-approximation algorithm in the case when the capacity $c = 2$.*

Proof. We first bound $|I^* - I|$. By (11.17), we have $\bar{\ell}(P_i^*) \geq \beta^2$ for all $i \in I^* - I$. Summing over all paths in $I^* - I$, we get

$$\sum_{i \in I^* - I} \bar{\ell}(P_i^*) \geq \beta^2 |I^* - I|.$$

On the other hand, each edge is used by at most two paths in the solution I^*, so we have

$$\sum_{i \in I^* - I} \bar{\ell}(P_i^*) \leq \sum_{e \in E} 2\bar{\ell}_e.$$

Combining these bounds with (11.18) we get

$$\beta^2 |I^*| \leq \beta^2 |I^* - I| + \beta^2 |I| \leq \sum_{i \in I^* - I} \bar{\ell}(P_i^*) + \beta^2 |I|$$

$$\leq \sum_{e \in E} 2\bar{\ell}_e + \beta^2 |I| \leq 2(\beta^3 |I| + m) + \beta^2 |I|.$$

Finally, dividing through by β^2, using $|I| \geq 1$ and setting $\beta = m^{1/3}$, we get that $|I^*| \leq (4m^{1/3} + 1)|I|$. ■

The same algorithm also works for the capacitated Disjoint Path Problem with any capacity $c > 0$. If we choose $\beta = m^{1/(c+1)}$, then the algorithm is a $(2cm^{1/(c+1)} + 1)$-approximation algorithm. To extend the analysis, one has to consider paths to be short if their length is at most β^c.

(11.20) *The algorithm* `Greedy-Paths-with-Capacity`*, using $\beta = m^{1/c+1}$, is a $(2cm^{1/(c+1)} + 1)$-approximation algorithm when the the edge capacities are c.*

11.6 Linear Programming and Rounding: An Application to Vertex Cover

We will start by introducing a powerful technique from operations research: *linear programming*. Linear programming is the subject of entire courses, and

we will not attempt to provide any kind of comprehensive overview of it here. In this section, we will introduce some of the basic ideas underlying linear programming and show how these can be used to approximate NP-hard optimization problems.

Recall that in Section 11.4 we developed a 2-approximation algorithm for the weighted Vertex Cover Problem. As a first application for the linear programming technique, we'll give here a different 2-approximation algorithm that is conceptually much simpler (though slower in running time).

Linear Programming as a General Technique

Our 2-approximation algorithm for the weighted version of Vertex Cover will be based on linear programming. We describe linear programming here not just to give the approximation algorithm, but also to illustrate its power as a very general technique.

So what is linear programming? To answer this, it helps to first recall, from linear algebra, the problem of simultaneous linear equations. Using matrix-vector notation, we have a vector x of unknown real numbers, a given matrix A, and a given vector b; and we want to solve the equation $Ax = b$. Gaussian elimination is a well-known efficient algorithm for this problem.

The basic Linear Programming Problem can be viewed as a more complex version of this, with inequalities in place of equations. Specifically, consider the problem of determining a vector x that satisfies $Ax \geq b$. By this notation, we mean that each coordinate of the vector Ax should be greater than or equal to the corresponding coordinate of the vector b. Such systems of inequalities define regions in space. For example, suppose $x = (x_1, x_2)$ is a two-dimensional vector, and we have the four inequalities

$$x_1 \geq 0, x_2 \geq 0$$
$$x_1 + 2x_2 \geq 6$$
$$2x_1 + x_2 \geq 6$$

Then the set of solutions is the region in the plane shown in Figure 11.10.

Given a region defined by $Ax \geq b$, linear programming seeks to minimize a linear combination of the coordinates of x, over all x belonging to the region. Such a linear combination can be written $c^t x$, where c is a vector of coefficients, and $c^t x$ denotes the inner product of two vectors. Thus our standard form for Linear Programming, as an optimization problem, will be the following.

Given an $m \times n$ matrix A, and vectors $b \in R^m$ and $c \in R^n$, find a vector $x \in R^n$ to solve the following optimization problem:

$$\min(c^t x \text{ such that } x \geq 0; Ax \geq b).$$

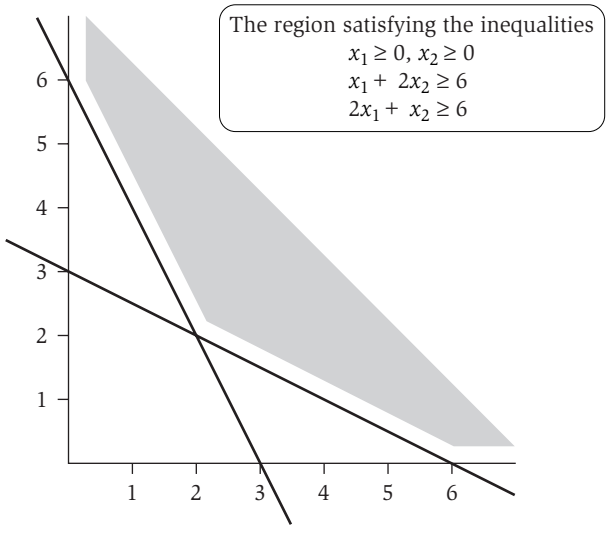

Figure 11.10 The feasible region of a simple linear program.

$c^t x$ is often called the *objective function* of the linear program, and $Ax \geq b$ is called the set of *constraints*. For example, suppose we define the vector c to be $(1.5, 1)$ in the example in Figure 11.10; in other words, we are seeking to minimize the quantity $1.5x_1 + x_2$ over the region defined by the inequalities. The solution to this would be to choose the point $x = (2, 2)$, where the two slanting lines cross; this yields a value of $c^t x = 5$, and one can check that there is no way to get a smaller value.

We can phrase Linear Programming as a decision problem in the following way.

> *Given a matrix A, vectors b and c, and a bound γ, does there exist x so that $x \geq 0$, $Ax \geq b$, and $c^t x \leq \gamma$?*

To avoid issues related to how we represent real numbers, we will assume that the coordinates of the vectors and matrices involved are integers.

The Computational Complexity of Linear Programming The decision version of Linear Programming is in $\mathcal{NP}$. This is intuitively very believable—we just have to exhibit a vector x satisfying the desired properties. The one concern is that even if all the input numbers are integers, such a vector x may not have integer coordinates, and it may in fact require very large precision to specify: How do we know that we'll be able to read and manipulate it in polynomial time? But, in fact, one can show that if there is a solution, then there is one that is rational and needs only a polynomial number of bits to write down; so this is not a problem.

Linear Programming was also known to be in co-$\mathcal{NP}$ for a long time, though this is not as easy to see. Students who have taken a linear programming course may notice that this fact follows from linear programming duality.[2]

For a long time, indeed, Linear Programming was the most famous example of a problem in both $\mathcal{NP}$ and co-$\mathcal{NP}$ that was not known to have a polynomial-time solution. Then, in 1981, Leonid Khachiyan, who at the time was a young researcher in the Soviet Union, gave a polynomial-time algorithm for the problem. After some initial concern in the U.S. popular press that this discovery might turn out to be a *Sputnik*-like event in the Cold War (it didn't), researchers settled down to understand exactly what Khachiyan had done. His initial algorithm, while polynomial-time, was in fact quite slow and impractical; but since then practical polynomial-time algorithms—so-called *interior point methods*—have also been developed following the work of Narendra Karmarkar in 1984.

Linear programming is an interesting example for another reason as well. The most widely used algorithm for this problem is the *simplex method*. It works very well in practice and is competitive with polynomial-time interior methods on real-world problems. Yet its worst-case running time is known to be exponential; it is simply that this exponential behavior shows up in practice only very rarely. For all these reasons, linear programming has been a very useful and important example for thinking about the limits of polynomial time as a formal definition of efficiency.

For our purposes here, though, the point is that linear programming problems can be solved in polynomial time, and very efficient algorithms exist in practice. You can learn a lot more about all this in courses on linear programming. The question we ask here is this: How can linear programming help us when we want to solve combinatorial problems such as Vertex Cover?

Vertex Cover as an Integer Program

Recall that a *vertex cover* in a graph $G = (V, E)$ is a set $S \subseteq V$ so that each edge has at least one end in S. In the weighted Vertex Cover Problem, each vertex $i \in V$ has a *weight* $w_i \geq 0$, with the weight of a set S of vertices denoted $w(S) = \sum_{i \in S} w_i$. We would like to find a vertex cover S for which $w(S)$ is minimum.

[2] Those of you who are familiar with duality may also notice that the *pricing method* of the previous sections is motivated by linear programming duality: the prices are exactly the variables in the dual linear program (which explains why pricing algorithms are often referred to as *primal-dual algorithms*).

We now try to formulate a linear program that is in close correspondence with the Vertex Cover Problem. Thus we consider a graph $G = (V, E)$ with a weight $w_i \geq 0$ on each node i. Linear programming is based on the use of vectors of variables. In our case, we will have a *decision variable* x_i for each node $i \in V$ to model the choice of whether to include node i in the vertex cover; $x_i = 0$ will indicate that node i is not in the vertex cover, and $x_i = 1$ will indicate that node i is in the vertex cover. We can create a single n-dimensional vector x in which the i^{th} coordinate corresponds to the i^{th} decision variable x_i.

We use linear inequalities to encode the requirement that the selected nodes form a vertex cover; we use the objective function to encode the goal of minimizing the total weight. For each edge $(i, j) \in E$, it must have one end in the vertex cover, and we write this as the inequality $x_i + x_j \geq 1$. Finally, to express the minimization problem, we write the set of node weights as an n-dimensional vector w, with the i^{th} coordinate corresponding to w_i; we then seek to minimize $w^t x$. In summary, we have formulated the Vertex Cover Problem as follows.

$$\begin{aligned} \text{(VC.IP)} \qquad \text{Min} \quad & \sum_{i \in V} w_i x_i \\ \text{s.t.} \quad & x_i + x_j \geq 1 \quad (i, j) \in E \\ & x_i \in \{0, 1\} \quad i \in V. \end{aligned}$$

We claim that the vertex covers of G are in one-to-one correspondence with the solutions x to this system of linear inequalities in which all coordinates are equal to 0 or 1.

(11.21) *S is a vertex cover in G if and only if the vector x, defined as $x_i = 1$ for $i \in S$, and $x_i = 0$ for $i \notin S$, satisfies the constraints in (VC.IP). Further, we have $w(S) = w^t x$.*

We can put this system into the matrix form we used for linear programming, as follows. We define a matrix A whose columns correspond to the nodes in V and whose rows correspond to the edges in E; entry $A[e, i] = 1$ if node i is an end of the edge e, and 0 otherwise. (Note that each row has exactly two nonzero entries.) If we use $\vec{1}$ to denote the vector with all coordinates equal to 1, and $\vec{0}$ to denote the vector with all coordinates equal to 0, then the system of inequalities above can be written as

$$\begin{aligned} Ax &\geq \vec{1} \\ \vec{1} \geq x &\geq \vec{0}. \end{aligned}$$

But keep in mind that this is not just an instance of the Linear Programming Problem: We have crucially required that all coordinates in the solution be either 0 or 1. So our formulation suggests that we should solve the problem

$$\min(w^t x \text{ subject to } \vec{1} \geq x \geq \vec{0}, Ax \geq \vec{1},\ x \text{ has integer coordinates}).$$

This is an instance of the Linear Programming Problem in which we require the coordinates of x to take integer values; without this extra constraint, the coordinates of x could be arbitrary real numbers. We call this problem *Integer Programming*, as we are looking for integer-valued solutions to a linear program.

Integer Programming is considerably harder than Linear Programming; indeed, our discussion really constitutes a reduction from Vertex Cover to the decision version of Integer Programming. In other words, we have proved

(11.22) Vertex Cover $\leq_P$ Integer Programming.

To show the NP-completeness of Integer Programming, we would still have to establish that the decision version is in $\mathcal{NP}$. There is a complication here, as with Linear Programming, since we need to establish that there is always a solution x that can be written using a polynomial number of bits. But this can indeed be proven. Of course, for our purposes, the integer program we are dealing with is explicitly constrained to have solutions in which each coordinate is either 0 or 1. Thus it is clearly in $\mathcal{NP}$, and our reduction from Vertex Cover establishes that even this special case is NP-complete.

Using Linear Programming for Vertex Cover

We have yet to resolve whether our foray into linear and integer programming will turn out to be useful or simply a dead end. Trying to solve the integer programming problem (VC.IP) optimally is clearly not the right way to go, as this is NP-hard.

The way to make progress is to exploit the fact that Linear Programming is not as hard as Integer Programming. Suppose we take (VC.IP) and modify it, dropping the requirement that each $x_i \in \{0, 1\}$ and reverting to the constraint that each x_i is an arbitrary real number between 0 and 1. This gives us an instance of the Linear Programming Problem that we could call (VC.LP), and we can solve it in polynomial time: We can find a set of values $\{x_i^*\}$ between 0 and 1 so that $x_i^* + x_j^* \geq 1$ for each edge (i, j), and $\sum_i w_i x_i^*$ is minimized. Let x^* denote this vector, and $w_{LP} = w^t x^*$ denote the value of the objective function.

We note the following basic fact.

(11.23) *Let S^* denote a vertex cover of minimum weight. Then $w_{LP} \leq w(S^*)$.*

Proof. Vertex covers of G correspond to integer solutions of (VC.IP), so the minimum of $\min(w^t x : \vec{1} \geq x \geq 0, Ax \geq 1)$ over all integer x vectors is exactly the minimum-weight vertex cover. To get the minimum of the linear program (VC.LP), we allow x to take arbitrary real-number values—that is, we minimize over many more choices of x—and so the minimum of (VC.LP) is no larger than that of (VC.IP). ■

Note that (11.23) is one of the crucial ingredients we need for an approximation algorithm: a good lower bound on the optimum, in the form of the efficiently computable quantity w_{LP}.

However, w_{LP} can definitely be smaller than $w(S^*)$. For example, if the graph G is a triangle and all weights are 1, then the minimum vertex cover has a weight of 2. But, in a linear programming solution, we can set $x_i = \frac{1}{2}$ for all three vertices, and so get a linear programming solution of weight only $\frac{3}{2}$. As a more general example, consider a graph on n nodes in which each pair of nodes is connected by an edge. Again, all weights are 1. Then the minimum vertex cover has weight $n - 1$, but we can find a linear programming solution of value $n/2$ by setting $x_i = \frac{1}{2}$ for all vertices i.

So the question is: How can solving this linear program help us actually *find* a near-optimal vertex cover? The idea is to work with the values x_i^* and to infer a vertex cover S from them. It is natural that if $x_i^* = 1$ for some node i, then we should put it in the vertex cover S; and if $x_i^* = 0$, then we should leave it out of S. But what should we do with fractional values in between? What should we do if $x_i^* = .4$ or $x_i^* = .5$? The natural approach here is to *round*. Given a fractional solution $\{x_i^*\}$, we define $S = \{i \in V : x_i^* \geq \frac{1}{2}\}$—that is, we round values at least $\frac{1}{2}$ up, and those below $\frac{1}{2}$ down.

(11.24) *The set S defined in this way is a vertex cover, and $w(S) \leq w_{LP}$.*

Proof. First we argue that S is a vertex cover. Consider an edge $e = (i, j)$. We claim that at least one of i and j must be in S. Recall that one of our inequalities is $x_i + x_j \geq 1$. So in any solution x^* that satisfies this inequality, either $x_i^* \geq \frac{1}{2}$ or $x_j^* \geq \frac{1}{2}$. Thus at least one of these two will be rounded up, and i or j will be placed in S.

Now we consider the weight $w(S)$ of this vertex cover. The set S only has vertices with $x_i^* \geq \frac{1}{2}$; thus the linear program "paid" at least $\frac{1}{2} w_i$ for node i, and we only pay w_i: at most twice as much. More formally, we have the following chain of inequalities.

$$w_{LP} w^t x^* = \sum_i w_i x_i^* \geq \sum_{i \in S} w_i x_i^* \geq \frac{1}{2} \sum_{i \in S} w_i = \frac{1}{2} w(S). \quad ■$$

Thus we have a produced a vertex cover S of weight at most $2w_{LP}$. The lower bound in (11.23) showed that the optimal vertex cover has weight at least w_{LP}, and so we have the following result.

(11.25) *The algorithm produces a vertex cover S of at most twice the minimum possible weight.*

* 11.7 Load Balancing Revisited: A More Advanced LP Application

In this section we consider a more general load balancing problem. We will develop an approximation algorithm using the same general outline as the 2-approximation we just designed for Vertex Cover: We solve a corresponding linear program, and then round the solution. However, the algorithm and its analysis here will be significantly more complex than what was needed for Vertex Cover. It turns out that the instance of the Linear Programming Problem we need to solve is, in fact, a flow problem. Using this fact, we will be able to develop a much deeper understanding of what the fractional solutions to the linear program look like, and we will use this understanding in order to round them. For this problem, the only known constant-factor approximation algorithm is based on rounding this linear programming solution.

The Problem

The problem we consider in this section is a significant, but natural, generalization of the Load Balancing Problem with which we began our study of approximation algorithms. There, as here, we have a set J of n jobs, and a set M of m machines, and the goal is to assign each job to a machine so that the maximum load on any machine will be as small as possible. In the simple Load Balancing Problem we considered earlier, each job j can be assigned to any machine i. Here, on the other hand, we will restrict the set of machines that each job may consider; that is, for each job there is just a subset of machines to which it can be assigned. This restriction arises naturally in a number of applications: for example, we may be seeking to balance load while maintaining the property that each job is assigned to a physically nearby machine, or to a machine with an appropriate authorization to process the job.

More formally, each job j has a fixed given size $t_j \geq 0$ and a set of machines $M_j \subseteq M$ that it may be assigned to. The sets M_j can be completely arbitrary. We call an assignment of jobs to machines *feasible* if each job j is assigned to a machine $i \in M_j$. The goal is still to minimize the maximum load on any machine: Using $J_i \subseteq J$ to denote the jobs assigned to a machine $i \in M$ in a feasible assignment, and using $L_i = \sum_{j \in J_i} t_j$ to denote the resulting load,

we seek to minimize $\max_i L_i$. This is the definition of the *Generalized Load Balancing Problem*.

In addition to containing our initial Load Balancing Problem as a special case (setting $M_j = M$ for all jobs j), Generalized Load Balancing includes the Bipartite Perfect Matching Problem as another special case. Indeed, given a bipartite graph with the same number of nodes on each side, we can view the nodes on the left as jobs and the nodes on the right as machines; we define $t_j = 1$ for all jobs j, and define M_j to be the set of machine nodes i such that there is an edge $(i, j) \in E$. There is an assignment of maximum load 1 if and only if there is a perfect matching in the bipartite graph. (Thus, network flow techniques can be used to find the optimum load in this special case.) The fact that Generalized Load Balancing includes both these problems as special cases gives some indication of the challenge in designing an algorithm for it.

Designing and Analyzing the Algorithm

We now develop an approximation algorithm based on linear programming for the Generalized Load Balancing Problem. The basic plan is the same one we saw in the previous section: we'll first formulate the problem as an equivalent linear program where the variables have to take specific discrete values; we'll then relax this to a linear program by dropping this requirement on the values of the variables; and then we'll use the resulting fractional assignment to obtain an actual assignment that is close to optimal. We'll need to be more careful than in the case of the Vertex Cover Problem in rounding the solution to produce the actual assignment.

Integer and Linear Programming Formulations First we formulate the Generalized Load Balancing Problem as a linear program with restrictions on the variable values. We use variables x_{ij} corresponding to each pair (i, j) of machine $i \in M$ and job $j \in J$. Setting $x_{ij} = 0$ will indicate that job j is not assigned to machine i, while setting $x_{ij} = t_j$ will indicate that all the load t_j of job j is assigned to machine i. We can think of x as a single vector with mn coordinates.

We use linear inequalities to encode the requirement that each job is assigned to a machine: For each job j we require that $\sum_i x_{ij} = t_j$. The load of a machine i can then be expressed as $L_i = \sum_j x_{ij}$. We require that $x_{ij} = 0$ whenever $i \notin M_j$. We will use the objective function to encode the goal of finding an assignment that minimizes the maximum load. To do this, we will need one more variable, L, that will correspond to the load. We use the inequalities $\sum_j x_{ij} \leq L$ for all machines i. In summary, we have formulated the following problem.

$$\begin{aligned}
\text{(GL.IP)} \qquad & \min L \\
& \sum_i x_{ij} = t_j \quad \text{for all } j \in J \\
& \sum_j x_{ij} \leq L \quad \text{for all } i \in M \\
& x_{ij} \in \{0, t_j\} \quad \text{for all } j \in J, i \in M_j. \\
& x_{ij} = 0 \quad \text{for all } j \in J, i \notin M_j.
\end{aligned}$$

First we claim that the feasible assignments are in one-to-one correspondence with the solutions x satisfying the above constraints, and, in an optimal solution to (GL.IP), L is the load of the corresponding assignment.

(11.26) *An assignment of jobs to machines has load at most L if and only if the vector x, defined by setting $x_{ij} = t_j$ whenever job j is assigned to machine i, and $x_{ij} = 0$ otherwise, satisfies the constraints in (GL.IP), with L set to the maximum load of the assignment.*

Next we will consider the corresponding linear program obtained by replacing the requirement that each $x_{ij} \in \{0, t_j\}$ by the weaker requirement that $x_{ij} \geq 0$ for all $j \in J$ and $i \in M_j$. Let (GL.LP) denote the resulting linear program. It would also be natural to add $x_{ij} \leq t_j$. We do not add these inequalities explicitly, as they are implied by the nonnegativity and the equation $\sum_i x_{ij} = t_j$ that is required for each job j.

We immediately see that if there is an assignment with load at most L, then (GL.LP) must have a solution with value at most L. Or, in the contrapositive,

(11.27) *If the optimum value of (GL.LP) is L, then the optimal load is at least $L^* \geq L$.*

We can use linear programming to obtain such a solution (x, L) in polynomial time. Our goal will then be to use x to create an assignment. Recall that the Generalized Load Balancing Problem is NP-hard, and hence we cannot expect to solve it exactly in polynomial time. Instead, we will find an assignment with load at most two times the minimum possible. To be able to do this, we will also need the simple lower bound (11.2), which we used already in the original Load Balancing Problem.

(11.28) *The optimal load is at least $L^* \geq \max_j t_j$.*

Rounding the Solution When There Are No Cycles The basic idea is to round the x_{ij} values to 0 or t_j. However, we cannot use the simple idea of just rounding large values up and small values down. The problem is that the linear programming solution may assign small fractions of a job j to each of

the m machines, and hence for some jobs there may be no large x_{ij} values. The algorithm we develop will be a rounding of x in the weak sense that each job j will be assigned to a machine i with $x_{ij} > 0$, but we may have to round a few really small values up. This weak rounding already ensures that the assignment is feasible, in the sense that we do not assign any job j to a machine i not in M_j (because if $i \notin M_j$, then we have $x_{ij} = 0$).

The key is to understand what the structure of the fractional solution is like and to show that while a few jobs may be spread out to many machines, this cannot happen to too many jobs. To this end, we'll consider the following bipartite graph $G(x) = (V(x), E(x))$: The nodes are $V(x) = M \cup J$, the set of jobs and the set of machines, and there is an edge $(i, j) \in E(x)$ if and only if $x_{ij} > 0$.

We'll show that, given any solution for (GL.LP), we can obtain a new solution x with the same load L, such that $G(x)$ has no cycles. This is the crucial step, as we show that a solution x with no cycles can be used to obtain an assignment with load at most $L + L^*$.

(11.29) *Given a solution (x, L) of (GL.LP) such that the graph $G(x)$ has no cycles, we can use this solution x to obtain a feasible assignment of jobs to machines with load at most $L + L^*$ in $O(mn)$ time.*

Proof. Since the graph $G(x)$ has no cycles, each of its connected components is a tree. We can produce the assignment by considering each component separately. Thus, consider one of the components, which is a tree whose nodes correspond to jobs and machines, as shown in Figure 11.11.

First, root the tree at an arbitrary node. Now consider a job j. If the node corresponding to job j is a leaf of the tree, let machine node i be its parent. Since j has degree 1 in the tree $G(x)$, machine i is the only machine that has been assigned any part of job j, and hence we must have that $x_{ij} = t_j$. Our assignment will assign such a job j to its only neighbor i. For a job j whose corresponding node is not a leaf in $G(x)$, we assign j to an arbitrary child of the corresponding node in the rooted tree.

The method can clearly be implemented in $O(mn)$ time (including the time to set up the graph $G(x)$). It defines a feasible assignment, as the linear program (GL.LP) required that $x_{ij} = 0$ whenever $i \notin M_j$. To finish the proof, we need to show that the load is at most $L + L^*$. Let i be any machine, and let J_i be the set of jobs assigned to machine i. The jobs assigned to machine i form a subset of the neighbors of i in $G(x)$: the set J_i contains those children of node i that are leaves, plus possibly the parent $p(i)$ of node i. To bound the load, we consider the parent $p(i)$ separately. For all other jobs $j \neq p(i)$ assigned to i, we have $x_{ij} = t_j$, and hence we can bound the load using the solution x, as follows.

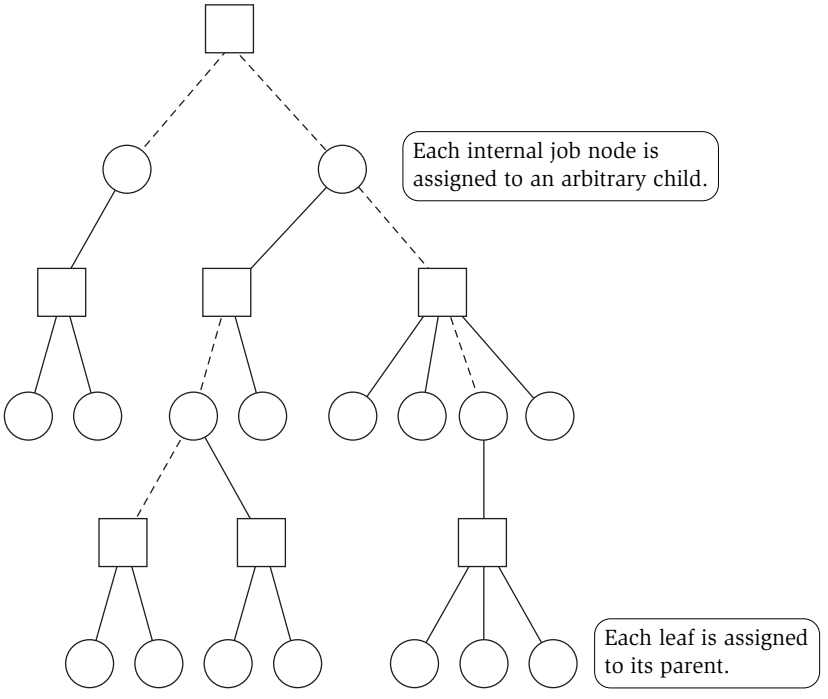

Figure 11.11 An example of a graph $G(x)$ with no cycles, where the squares are machines and the circles are jobs. The solid lines show the resulting assignment of jobs to machines.

$$\sum_{j \in J_i, j \neq p(i)} t_j \leq \sum_{j \in J} x_{ij} \leq L,$$

using the inequality bounding the load in (GL.LP). For the parent $j = p(i)$ of node i, we use $t_j \leq L^*$ by (11.28). Adding the two inequalities, we get that $\sum_{j \in J_i} p_{ij} \leq L + L^*$, as claimed. ■

Now, by (11.27), we know that $L \leq L^*$, so a solution whose load is bounded by $L + L^*$ is also bounded by $2L^*$—in other words, twice the optimum. Thus we have the following consequence of (11.29).

(11.30) *Given a solution (x, L) of (GL.LP) such that the graph $G(x)$ has no cycles, then we can use this solution x to obtain a feasible assignment of jobs to machines with load at most twice the optimum in $O(mn)$ time.*

Eliminating Cycles from the Linear Programming Solution To wrap up our approximation algorithm, then, we just need to show how to convert

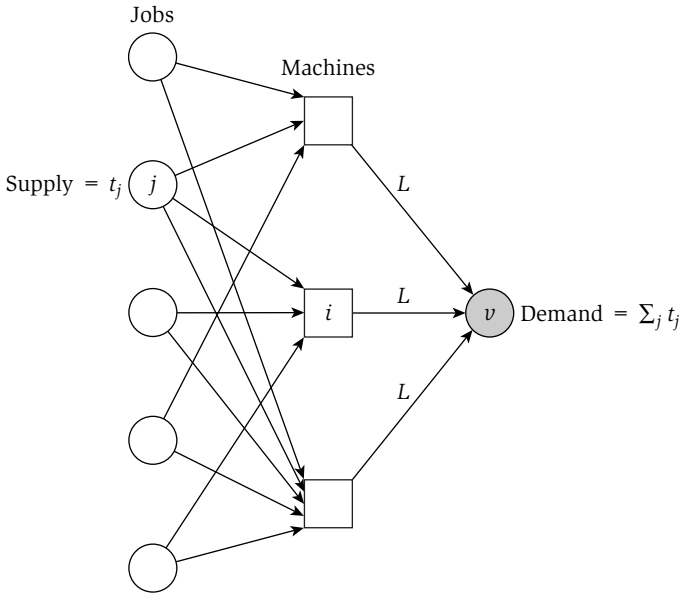

Figure 11.12 The network flow computation used to find a solution to (GL.LP). Edges between the jobs and machines have infinite capacity.

an arbitrary solution of (GL.LP) into a solution x with no cycles in $G(x)$. In the process, we will also show how to obtain a solution to the linear program (GL.LP) using flow computations. More precisely, given a fixed load value L, we show how to use a flow computation to decide if (GL.LP) has a solution with value at most L. For this construction, consider the following directed graph $G = (V, E)$ shown in Figure 11.12. The set of vertices of the flow graph G will be $V = M \cup J \cup \{v\}$, where v is a new node. The nodes $j \in J$ will be sources with supply t_j, and the only demand node is the new sink v, which has demand $\sum_j t_j$. We'll think of the flow in this network as "load" flowing from jobs to the sink v via the machines. We add an edge (j, i) with infinite capacity from job j to machine i if and only if $i \in M_j$. Finally, we add an edge (i, v) for each machine node i with capacity L.

(11.31) *The solutions of this flow problem with capacity L are in one-to-one correspondence with solutions of (GL.LP) with value L, where x_{ij} is the flow value along edge (i, j), and the flow value on edge (i, t) is the load $\sum_j x_{ij}$ on machine i.*

This statement allows us to solve (GL.LP) using flow computations and a binary search for the optimal value L: we try successive values of L until we find the smallest one for which there is a feasible flow.

Here we'll use the understanding we gained of (GL.LP) from the equivalent flow formulation to modify a solution x to eliminate all cycles from $G(x)$. In terms of the flow we have just defined, $G(x)$ is the undirected graph obtained from G by ignoring the directions of the edges, deleting the sink v and all adjacent edges, and also deleting all edges from J to M that do not carry flow. We'll eliminate all cycles in $G(x)$ in a sequence of at most mn steps, where the goal of a single step is to eliminate at least one edge from $G(x)$ without increasing the load L or introducing any new edges.

(11.32) *Let (x, L) be any solution to (GL.LP) and C be a cycle in $G(x)$. In time linear in the length of the cycle, we can modify the solution x to eliminate at least one edge from $G(x)$ without increasing the load or introducing any new edges.*

Proof. Consider the cycle C in $G(x)$. Recall that $G(x)$ corresponds to the set of edges that carry flow in the solution x. We will modify the solution by *augmenting* the flow along the cycle C, using essentially the procedure `augment` from Section 7.1. The augmentation along a cycle will not change the balance between incoming and outgoing flow at any node; rather, it will eliminate one backward edge from the residual graph, and hence an edge from $G(x)$. Assume that the nodes along the cycle are $i_1, j_1, i_2, j_2, \ldots, i_k, j_k$, where i_ℓ is a machine node and j_ℓ is a job node. We'll modify the solution by decreasing the flow along all edges (j_ℓ, i_ℓ) and increasing the flow on the edges $(j_\ell, i_{\ell+1})$ for all $\ell = 1, \ldots, k$ (where $k+1$ is used to denote 1), by the same amount δ. This change will not affect the flow conservation constraints. By setting $\delta = \min_{\ell=1}^{k} x_{i_\ell j_\ell}$, we ensure that the flow remains feasible and the edge obtaining the minimum is deleted from $G(x)$. ■

We can use the algorithm contained in the proof of (11.32) repeatedly to eliminate all cycles from $G(x)$. Initially, $G(x)$ may have mn edges, so after at most $O(mn)$ iterations, the resulting solution (x, L) will have no cycles in $G(x)$. At this point, we can use (11.30) to obtain a feasible assignment with at most twice the optimal load. We summarize the result by the following statement.

(11.33) *Given an instance of the Generalized Load Balancing Problem, we can find, in polynomial time, a feasible assignment with load at most twice the minimum possible.*

11.8 Arbitrarily Good Approximations: The Knapsack Problem

Often, when you talk to someone faced with an NP-hard optimization problem, they're hoping you can give them something that will produce a solution within, say, 1 percent of the optimum, or at least within a small percentage of optimal. Viewed from this perspective, the approximation algorithms we've seen thus far come across as quite weak: solutions within a factor of 2 of the minimum for Center Selection and Vertex Cover (i.e., 100 percent more than optimal). The Set Cover Algorithm in Section 10.3 is even worse: Its cost is not even within a fixed constant factor of the minimum possible!

Here is an important point underlying this state of affairs: NP-complete problems, as you well know, are all equivalent with respect to polynomial-time solvability; but assuming $\mathcal{P} \neq \mathcal{NP}$, they differ considerably in the extent to which their solutions can be efficiently approximated. In some cases, it is actually possible to prove limits on approximability. For example, if $\mathcal{P} \neq \mathcal{NP}$, then the guarantee provided by our Center Selection Algorithm is the best possible for any polynomial-time algorithm. Similarly, the guarantee provided by the Set Cover Algorithm, however bad it may seem, is very close to the best possible, unless $\mathcal{P} = \mathcal{NP}$. For other problems, such as the Vertex Cover Problem, the approximation algorithm we gave is essentially the best known, but it is an open question whether there could be polynomial-time algorithms with better guarantees. We will not discuss the topic of lower bounds on approximability in this book; while some lower bounds of this type are not so difficult to prove (such as for Center Selection), many are extremely technical.

The Problem

In this section, we discuss an NP-complete problem for which it is possible to design a polynomial-time algorithm providing a very strong approximation. We will consider a slightly more general version of the Knapsack (or Subset Sum) Problem. Suppose you have n items that you consider packing in a knapsack. Each item $i = 1, \ldots, n$ has two integer parameters: a weight w_i and a value v_i. Given a knapsack capacity W, the goal of the Knapsack Problem is to find a subset S of items of maximum value subject to the restriction that the total weight of the set should not exceed W. In other words, we wish to maximize $\sum_{i \in S} v_i$ subject to the condition $\sum_{i \in S} w_i \leq W$.

How strong an approximation can we hope for? Our algorithm will take as input the weights and values defining the problem and will also take an extra parameter ϵ, the desired precision. It will find a subset S whose total weight does not exceed W, with value $\sum_{i \in S} v_i$ at most a $(1 + \epsilon)$ factor below the maximum possible. The algorithm will run in polynomial time for any

fixed choice of $\epsilon > 0$; however, the dependence on ϵ will not be polynomial. We call such an algorithm a *polynomial-time approximation scheme*.

You may ask: How could such a strong kind of approximation algorithm be possible in polynomial time when the Knapsack Problem is NP-hard? With integer values, if we get close enough to the optimum value, we must reach the optimum itself! The catch is in the nonpolynomial dependence on the desired precision: for any fixed choice of ϵ, such as $\epsilon = .5$, $\epsilon = .2$, or even $\epsilon = .01$, the algorithm runs in polynomial time, but as we change ϵ to smaller and smaller values, the running time gets larger. By the time we make ϵ small enough to make sure we get the optimum value, it is no longer a polynomial-time algorithm.

Designing the Algorithm

In Section 6.4 we considered algorithms for the Subset Sum Problem, the special case of the Knapsack Problem when $v_i = w_i$ for all items i. We gave a dynamic programming algorithm for this special case that ran in $O(nW)$ time assuming the weights are integers. This algorithm naturally extends to the more general Knapsack Problem (see the end of Section 6.4 for this extension). The algorithm given in Section 6.4 works well when the weights are small (even if the values may be big). It is also possible to extend our dynamic programming algorithm for the case when the values are small, even if the weights may be big. At the end of this section, we give a dynamic programming algorithm for that case running in time $O(n^2v^*)$, where $v^* = \max_i v_i$. Note that this algorithm does not run in polynomial time: It is only pseudo-polynomial, because of its dependence on the size of the values v_i. Indeed, since we proved this problem to be NP-complete in Chapter 8, we don't expect to be able to find a polynomial-time algorithm.

Algorithms that depend on the values in a pseudo-polynomial way can often be used to design polynomial-time approximation schemes, and the algorithm we develop here is a very clean example of the basic strategy. In particular, we will use the dynamic programming algorithm with running time $O(n^2v^*)$ to design a polynomial-time approximation scheme; the idea is as follows. If the values are small integers, then v^* is small and the problem can be solved in polynomial time already. On the other hand, if the values are large, then we do not have to deal with them exactly, as we only want an approximately optimum solution. We will use a rounding parameter b (whose value we'll set later) and will consider the values rounded to an integer multiple of b. We will use our dynamic programming algorithm to solve the problem with the rounded values. More precisely, for each item i, let its rounded value be $\tilde{v}_i = \lceil v_i/b \rceil b$. Note that the rounded and the original value are quite close to each other.

(11.34) *For each item i we have $v_i \leq \tilde{v}_i \leq v_i + b$.*

What did we gain by the rounding? If the values were big to start with, we did not make them any smaller. However, the rounded values are all integer multiples of a common value b. So, instead of solving the problem with the rounded values $\tilde{v}_i$, we can change the units; we can divide all values by b and get an equivalent problem. Let $\hat{v}_i = \tilde{v}_i/b = \lceil v_i/b \rceil$ for $i = 1, \ldots, n$.

(11.35) *The Knapsack Problem with values $\tilde{v}_i$ and the scaled problem with values $\hat{v}_i$ have the same set of optimum solutions, the optimum values differ exactly by a factor of b, and the scaled values are integral.*

Now we are ready to state our approximation algorithm. We will assume that all items have weight at most W (as items with weight $w_i > W$ are not in any solution, and hence can be deleted). We also assume for simplicity that ϵ^{-1} is an integer.

```
Knapsack-Approx(ε):
  Set b = (ε/(2n)) max_i v_i
  Solve the Knapsack Problem with values v̂_i (equivalently ṽ_i)
  Return the set S of items found
```

Analyzing the Algorithm

First note that the solution found is at least feasible; that is, $\sum_{i \in S} w_i \leq W$. This is true as we have rounded only the values and not the weights. This is why we need the new dynamic programming algorithm described at the end of this section.

(11.36) *The set of items S returned by the algorithm has total weight at most W, that is $\sum_{i \in S} w_i \leq W$.*

Next we'll prove that this algorithm runs in polynomial time.

(11.37) *The algorithm* `Knapsack-Approx` *runs in polynomial time for any fixed $\epsilon > 0$.*

Proof. Setting b and rounding item values can clearly be done in polynomial time. The time-consuming part of this algorithm is the dynamic programming to solve the rounded problem. Recall that for problems with integer values, the dynamic programming algorithm we use runs in time $O(n^2 v^*)$, where $v^* = \max_i v_i$.

Now we are applying this algorithms for an instance in which each item i has weight w_i and value $\hat{v}_i$. To determine the running time, we need to

determine $\max_i \hat{v}_i$. The item j with maximum value $v_j = \max_i v_i$ also has maximum value in the rounded problem, so $\max_i \hat{v}_i = \hat{v}_j = \lceil v_j/b \rceil = n\epsilon^{-1}$. Hence the overall running time of the algorithm is $O(n^3\epsilon^{-1})$. Note that this is polynomial time for any fixed $\epsilon > 0$ as claimed; but the dependence on the desired precision ϵ is not polynomial, as the running time includes ϵ^{-1} rather than $\log \epsilon^{-1}$. ■

Finally, we need to consider the key issue: How good is the solution obtained by this algorithm? Statement (11.34) shows that the values $\tilde{v}_i$ we used are close to the real values v_i, and this suggests that the solution obtained may not be far from optimal.

(11.38) *If S is the solution found by the* `Knapsack-Approx` *algorithm, and S^* is any other solution satisfying $\sum_{i\in S^*} w_i \le W$, then we have $(1+\epsilon)\sum_{i\in S} v_i \ge \sum_{i\in S^*} v_i$.*

Proof. Let S^* be any set satisfying $\sum_{i\in S^*} w_i \le W$. Our algorithm finds the optimal solution with values $\tilde{v}_i$, so we know that

$$\sum_{i\in S} \tilde{v}_i \ge \sum_{i\in S^*} \tilde{v}_i.$$

The rounded values $\tilde{v}_i$ and the real values v_i are quite close by (11.34), so we get the following chain of inequalities.

$$\sum_{i\in S^*} v_i \le \sum_{i\in S^*} \tilde{v}_i \le \sum_{i\in S} \tilde{v}_i \le \sum_{i\in S} (v_i + b) \le nb + \sum_{i\in S} v_i,$$

showing that the value $\sum_{i\in S} v_i$ of the solution we obtained is at most nb smaller than the maximum value possible. We wanted to obtain a relative error showing that the value obtained, $\sum_{i\in S} v_i$, is at most a $(1+\epsilon)$ factor less than the maximum possible, so we need to compare nb to the value $\sum_{i\in S} v_i$.

Let j be the item with largest value; by our choice of b, we have $v_j = 2\epsilon^{-1}nb$ and $v_j = \tilde{v}_j$. By our assumption that each item alone fits in the knapsack ($w_i \le W$ for all i), we have $\sum_{i\in S} \tilde{v}_i \ge \tilde{v}_j = 2\epsilon^{-1}nb$. Finally, the chain of inequalities above says $\sum_{i\in S} v_i \ge \sum_{i\in S} \tilde{v}_i - nb$, and thus $\sum_{i\in S} v_i \ge (2\epsilon^{-1} - 1)nb$. Hence $nb \le \epsilon \sum_{i\in S} v_i$ for $\epsilon \le 1$, and so

$$\sum_{i\in S^*} v_i \le \sum_{i\in S} v_i + nb \le (1+\epsilon) \sum_{i\in S} v_i. \quad ■$$

The New Dynamic Programming Algorithm for the Knapsack Problem

To solve a problem by dynamic programming, we have to define a polynomial set of subproblems. The dynamic programming algorithm we defined when we studied the Knapsack Problem earlier uses subproblems of the form $\text{OPT}(i, w)$: the subproblem of finding the maximum value of any solution using a subset of the items $1, \dots, i$ and a knapsack of weight w. When the weights are large, this is a large set of problems. We need a set of subproblems that work well when the values are reasonably small; this suggests that we should use subproblems associated with values, not weights. We define our subproblems as follows. The subproblem is defined by i and a target value V, and $\overline{\text{OPT}}(i, V)$ is the smallest knapsack weight W so that one can obtain a solution using a subset of items $\{1, \dots, i\}$ with value at least V. We will have a subproblem for all $i = 0, \dots, n$ and values $V = 0, \dots, \sum_{j=1}^{i} v_j$. If v^* denotes $\max_i v_i$, then we see that the largest V can get in a subproblem is $\sum_{j=1}^{n} v_j \leq nv^*$. Thus, assuming the values are integral, there are at most $O(n^2 v^*)$ subproblems. None of these subproblems is precisely the original instance of Knapsack, but if we have the values of all subproblems $\overline{\text{OPT}}(n, V)$ for $V = 0, \dots, \sum_i v_i$, then the value of the original problem can be obtained easily: it is the largest value V such that $\overline{\text{OPT}}(n, V) \leq W$.

It is not hard to give a recurrence for solving these subproblems. By analogy with the dynamic programming algorithm for Subset Sum, we consider cases depending on whether or not the last item n is included in the optimal solution $\mathcal{O}$.

- If $n \notin \mathcal{O}$, then $\overline{\text{OPT}}(n, V) = \overline{\text{OPT}}(n-1, V)$.
- If $n \in \mathcal{O}$ is the only item in $\mathcal{O}$, then $\overline{\text{OPT}}(n, V) = w_n$.
- If $n \in \mathcal{O}$ is not the only item in $\mathcal{O}$, then $\overline{\text{OPT}}(n, V) = w_n + \overline{\text{OPT}}(n-1, V - v_n)$.

These last two options can be summarized more compactly as

- If $n \in \mathcal{O}$, then $\overline{\text{OPT}}(n, V) = w_n + \overline{\text{OPT}}(n-1, \max(0, V - v_n))$.

This implies the following analogue of the recurrence (6.8) from Chapter 6.

(11.39) *If* $V > \sum_{i=1}^{n-1} v_i$, *then* $\overline{\text{OPT}}(n, V) = w_n + \overline{\text{OPT}}(n-1, V - v_n)$. *Otherwise*

$$\overline{\text{OPT}}(n, V) = \min(\overline{\text{OPT}}(n-1, V), w_n + \overline{\text{OPT}}(n-1, \max(0, V - v_n))).$$

We can then write down an analogous dynamic programming algorithm.

```
Knapsack(n):
  Array M[0 ... n, 0 ... V]
```

```
For i = 0, ..., n
  M[i, 0] = 0
Endfor
For i = 1, 2, ..., n
  For V = 1, ..., ∑_{j=1}^{i} v_j
    If V > ∑_{j=1}^{i-1} v_j then
      M[i, V] = w_i + M[i − 1, V]
    Else
      M[i, V] = min(M[i − 1, V], w_i + M[i − 1, max(0, V − v_i)])
    Endif
  Endfor
Endfor
Return the maximum value V such that M[n, V] ≤ W
```

(11.40) `Knapsack`(n) *takes* $O(n^2v^*)$ *time and correctly computes the optimal values of the subproblems.*

As was done before, we can trace back through the table M containing the optimal values of the subproblems, to find an optimal solution.

Solved Exercises

Solved Exercise 1

Recall the Shortest-First greedy algorithm for the Interval Scheduling Problem: Given a set of intervals, we repeatedly pick the shortest interval I, delete all the other intervals I' that intersect I, and iterate.

In Chapter 4, we saw that this algorithm does *not* always produce a maximum-size set of nonoverlapping intervals. However, it turns out to have the following interesting approximation guarantee. If s^* is the maximum size of a set of nonoverlapping intervals, and s is the size of the set produced by the Shortest-First Algorithm, then $s \geq \frac{1}{2}s^*$ (that is, Shortest-First is a 2-approximation).

Prove this fact.

Solution Let's first recall the example in Figure 4.1 from Chapter 4, which showed that Shortest-First does not necessarily find an optimal set of intervals. The difficulty is clear: We may select a short interval j while eliminating two longer flanking intervals i and i'. So we have done only half as well as the optimum.

The question is to show that *Shortest-First* could never do worse than this. The issues here are somewhat similar to what came up in the analysis of the

greedy algorithm for the Maximum Disjoint Paths Problem in Section 11.5: Each interval we select may "block" some of the intervals in an optimal solution, and we want to argue that by always selecting the shortest possible interval, these blocking effects are not too severe. In the case of disjoint paths, we analyzed the overlaps among paths essentially edge by edge, since the underlying graph there had an arbitrary structure. Here we can benefit from the highly restricted structure of intervals on a line so as to obtain a stronger bound.

In order for Shortest-First to do less than half as well as the optimum, there would have to be a large optimal solution that overlaps with a much smaller solution chosen by Shortest-First. Intuitively, it seems that the only way this could happen would be to have one of the intervals i in the optimal solution nested completely inside one of the intervals j chosen by Shortest-First. This in turn would contradict the behavior of Shortest-First: Why didn't it choose this shorter interval i that's nested inside j?

Let's see if we can make this argument precise. Let A denote the set of intervals chosen by Shortest-First, and let $\mathcal{O}$ denote an optimal set of intervals. For each interval $j \in A$, consider the set of intervals in $\mathcal{O}$ that it conflicts with. We claim

(11.41) *Each interval $j \in A$ conflicts with at most two intervals in $\mathcal{O}$.*

Proof. Assume by way of contradiction that there is an interval in $j \in A$ that conflicts with at least three intervals in $i_1, i_2, i_3 \in O$. These three intervals do not conflict with one another, as they are part of a single solution $\mathcal{O}$, so they are ordered sequentially in time. Suppose they are ordered with i_1 first, then i_2, and then i_3. Since interval j conflicts with both i_1 and i_3, the interval i_2 in between must be shorter than j and fit completely inside it. Moreover, since i_2 was never selected by Shortest-First, it must have been available as an option when Shortest-First selected interval j. This is a contradiction, since i_2 is shorter than j. ■

The Shortest-First Algorithm only terminates when every unselected interval conflicts with one of the intervals it selected. So, in particular, each interval in $\mathcal{O}$ is either included in A, or conflicts with an interval in A.

Now we use the following accounting scheme to bound the number of intervals in $\mathcal{O}$. For each $i \in \mathcal{O}$, we have some interval $j \in A$ "pay" for i, as follows. If i is also in A, then i will pay for itself. Otherwise, we arbitrarily choose an interval $j \in A$ that conflicts with i and have j pay for i. As we just argued, every interval in $\mathcal{O}$ conflicts with some interval in A, so all intervals in $\mathcal{O}$ will be paid for under this scheme. But by (11.41), each interval $j \in A$ conflicts with at most two intervals in $\mathcal{O}$, and so it will only pay for at most

two intervals. Thus, all intervals in $\mathcal{O}$ are paid for by intervals in A, and in this process each interval in A pays at most twice. If follows that A must have at least half as many intervals as $\mathcal{O}$.

Exercises

1. Suppose you're acting as a consultant for the Port Authority of a small Pacific Rim nation. They're currently doing a multi-billion-dollar business per year, and their revenue is constrained almost entirely by the rate at which they can unload ships that arrive in the port.

Here's a basic sort of problem they face. A ship arrives, with n containers of weight $w_1, w_2, \ldots, w_n$. Standing on the dock is a set of trucks, each of which can hold K units of weight. (You can assume that K and each w_i is an integer.) You can stack multiple containers in each truck, subject to the weight restriction of K; the goal is to minimize the number of trucks that are needed in order to carry all the containers. This problem is NP-complete (you don't have to prove this).

A greedy algorithm you might use for this is the following. Start with an empty truck, and begin piling containers $1, 2, 3, \ldots$ into it until you get to a container that would overflow the weight limit. Now declare this truck "loaded" and send it off; then continue the process with a fresh truck. This algorithm, by considering trucks one at a time, may not achieve the most efficient way to pack the full set of containers into an available collection of trucks.

(a) Give an example of a set of weights, and a value of K, where this algorithm does not use the minimum possible number of trucks.

(b) Show, however, that the number of trucks used by this algorithm is within a factor of 2 of the minimum possible number, for any set of weights and any value of K.

2. At a lecture in a computational biology conference one of us attended a few years ago, a well-known protein chemist talked about the idea of building a "representative set" for a large collection of protein molecules whose properties we don't understand. The idea would be to intensively study the proteins in the representative set and thereby learn (by inference) about all the proteins in the full collection.

To be useful, the representative set must have two properties.

- It should be relatively small, so that it will not be too expensive to study it.

- Every protein in the full collection should be "similar" to some protein in the representative set. (In this way, it truly provides some information about all the proteins.)

More concretely, there is a large set P of proteins. We define similarity on proteins by a *distance function* d: Given two proteins p and q, it returns a number $d(p, q) \geq 0$. In fact, the function $d(\cdot, \cdot)$ most typically used is the *sequence alignment* measure, which we looked at when we studied dynamic programming in Chapter 6. We'll assume this is the distance being used here. There is a predefined distance cut-off Δ that's specified as part of the input to the problem; two proteins p and q are deemed to be "similar" to one another if and only if $d(p, q) \leq \Delta$.

We say that a subset of P is a *representative set* if, for every protein p, there is a protein q in the subset that is similar to it—that is, for which $d(p, q) \leq \Delta$. Our goal is to find a representative set that is as small as possible.

(a) Give a polynomial-time algorithm that approximates the minimum representative set to within a factor of $O(\log n)$. Specifically, your algorithm should have the following property: If the minimum possible size of a representative set is s^*, your algorithm should return a representative set of size at most $O(s^* \log n)$.

(b) Note the close similarity between this problem and the Center Selection Problem—a problem for which we considered approximation algorithms in Section 11.2. Why doesn't the algorithm described there solve the current problem?

3. Suppose you are given a set of positive integers $A = \{a_1, a_2, \ldots, a_n\}$ and a positive integer B. A subset $S \subseteq A$ is called *feasible* if the sum of the numbers in S does not exceed B:

$$\sum_{a_i \in S} a_i \leq B.$$

The sum of the numbers in S will be called the *total sum* of S.

You would like to select a feasible subset S of A whose total sum is as large as possible.

Example. If $A = \{8, 2, 4\}$ and $B = 11$, then the optimal solution is the subset $S = \{8, 2\}$.

(a) Here is an algorithm for this problem.

Initially $S = \phi$
Define $T = 0$
For $i = 1, 2, \ldots, n$

```
  If T + a_i ≤ B then
    S ← S ∪ {a_i}
    T ← T + a_i
  Endif
Endfor
```

Give an instance in which the total sum of the set S returned by this algorithm is less than half the total sum of some other feasible subset of A.

(b) Give a polynomial-time approximation algorithm for this problem with the following guarantee: It returns a feasible set $S \subseteq A$ whose total sum is at least half as large as the maximum total sum of any feasible set $S' \subseteq A$. Your algorithm should have a running time of at most $O(n \log n)$.

4. Consider an optimization version of the Hitting Set Problem defined as follows. We are given a set $A = \{a_1, \ldots, a_n\}$ and a collection $B_1, B_2, \ldots, B_m$ of subsets of A. Also, each element $a_i \in A$ has a *weight* $w_i \geq 0$. The problem is to find a hitting set $H \subseteq A$ such that the total weight of the elements in H, that is, $\sum_{a_i \in H} w_i$, is as small as possible. (As in Exercise 5 in Chapter 8, we say that H is a hitting set if $H \cap B_i$ is not empty for each i.) Let $b = \max_i |B_i|$ denote the maximum size of any of the sets $B_1, B_2, \ldots, B_m$. Give a polynomial-time approximation algorithm for this problem that finds a hitting set whose total weight is at most b times the minimum possible.

5. You are asked to consult for a business where clients bring in jobs each day for processing. Each job has a processing time t_i that is known when the job arrives. The company has a set of ten machines, and each job can be processed on any of these ten machines.

At the moment the business is running the simple Greedy-Balance Algorithm we discussed in Section 11.1. They have been told that this may not be the best approximation algorithm possible, and they are wondering if they should be afraid of bad performance. However, they are reluctant to change the scheduling as they really like the simplicity of the current algorithm: jobs can be assigned to machines as soon as they arrive, without having to defer the decision until later jobs arrive.

In particular, they have heard that this algorithm can produce solutions with makespan as much as twice the minimum possible; but their experience with the algorithm has been quite good: They have been running it each day for the last month, and they have not observed it to produce a makespan more than 20 percent above the average load, $\frac{1}{10} \sum_i t_i$.

To try understanding why they don't seem to be encountering this factor-of-two behavior, you ask a bit about the kind of jobs and loads they see. You find out that the sizes of jobs range between 1 and 50, that is, $1 \leq t_i \leq 50$ for all jobs i; and the total load $\sum_i t_i$ is quite high each day: it is always at least 3,000.

Prove that on the type of inputs the company sees, the Greedy-Balance Algorithm will always find a solution whose makespan is at most 20 percent above the average load.

6. Recall that in the basic Load Balancing Problem from Section 11.1, we're interested in placing jobs on machines so as to minimize the *makespan*—the maximum load on any one machine. In a number of applications, it is natural to consider cases in which you have access to machines with different amounts of processing power, so that a given job may complete more quickly on one of your machines than on another. The question then becomes: How should you allocate jobs to machines in these more heterogeneous systems?

Here's a basic model that exposes these issues. Suppose you have a system that consists of m *slow* machines and k *fast* machines. The fast machines can perform twice as much work per unit time as the slow machines. Now you're given a set of n jobs; job i takes time t_i to process on a slow machine and time $\frac{1}{2}t_i$ to process on a fast machine. You want to assign each job to a machine; as before, the goal is to minimize the makespan—that is the maximum, over all machines, of the total processing time of jobs assigned to that machine.

Give a polynomial-time algorithm that produces an assignment of jobs to machines with a makespan that is at most three times the optimum.

7. You're consulting for an e-commerce site that receives a large number of visitors each day. For each visitor i, where $i \in \{1, 2, \ldots, n\}$, the site has assigned a value v_i, representing the expected revenue that can be obtained from this customer.

Each visitor i is shown one of m possible ads $A_1, A_2, \ldots, A_m$ as they enter the site. The site wants a selection of one ad for each customer so that *each* ad is seen, overall, by a set of customers of reasonably large total weight. Thus, given a selection of one ad for each customer, we will define the *spread* of this selection to be the minimum, over $j = 1, 2, \ldots, m$, of the total weight of all customers who were shown ad A_j.

Example Suppose there are six customers with values $3, 4, 12, 2, 4, 6$, and there are $m = 3$ ads. Then, in this instance, one could achieve a spread of

9 by showing ad A_1 to customers 1, 2, 4, ad A_2 to customer 3, and ad A_3 to customers 5 and 6.

The ultimate goal is to find a selection of an ad for each customer that maximizes the spread. Unfortunately, this optimization problem is NP-hard (you don't have to prove this). So instead, we will try to approximate it.

(a) Give a polynomial-time algorithm that approximates the maximum spread to within a factor of 2. That is, if the maximum spread is s, then your algorithm should produce a selection of one ad for each customer that has spread at least $s/2$. In designing your algorithm, you may assume that no single customer has a value that is significantly above the average; specifically, if $\overline{v} = \sum_{i=1}^{n} v_i$ denotes the total value of all customers, then you may assume that no single customer has a value exceeding $\overline{v}/(2m)$.

(b) Give an example of an instance on which the algorithm you designed in part (a) does not find an optimal solution (that is, one of maximum spread). Say what the optimal solution is in your sample instance, and what your algorithm finds.

8. Some friends of yours are working with a system that performs real-time scheduling of jobs on multiple servers, and they've come to you for help in getting around an unfortunate piece of legacy code that can't be changed.

Here's the situation. When a batch of jobs arrives, the system allocates them to servers using the simple Greedy-Balance Algorithm from Section 11.1, which provides an approximation to within a factor of 2. In the decade and a half since this part of the system was written, the hardware has gotten faster to the point where, on the instances that the system needs to deal with, your friends find that it's generally possible to compute an optimal solution.

The difficulty is that the people in charge of the system's internals won't let them change the portion of the software that implements the Greedy-Balance Algorithm so as to replace it with one that finds the optimal solution. (Basically, this portion of the code has to interact with so many other parts of the system that it's not worth the risk of something going wrong if it's replaced.)

After grumbling about this for a while, your friends come up with an alternate idea. Suppose they could write a little piece of code that takes the description of the jobs, computes an optimal solution (since they're able to do this on the instances that arise in practice), and then feeds the jobs to the Greedy-Balance Algorithm *in an order that will cause it to allocate them optimally*. In other words, they're hoping to be able to

reorder the input in such a way that when Greedy-Balance encounters the input in this order, it produces an optimal solution.

So their question to you is simply the following: Is this always possible? Their conjecture is,

> *For every instance of the load balancing problem from Section 11.1, there exists an order of the jobs so that when Greedy-Balance processes the jobs in this order, it produces an assignment of jobs to machines with the minimum possible makespan.*

Decide whether you think this conjecture is true or false, and give either a proof or a counterexample.

9. Consider the following maximization version of the 3-Dimensional Matching Problem. Given disjoint sets X, Y, and Z, and given a set $T \subseteq X \times Y \times Z$ of ordered triples, a subset $M \subseteq T$ is a *3-dimensional matching* if each element of $X \cup Y \cup Z$ is contained in at most one of these triples. The *Maximum 3-Dimensional Matching Problem* is to find a 3-dimensional matching M of maximum size. (The size of the matching, as usual, is the number of triples it contains. You may assume $|X| = |Y| = |Z|$ if you want.)

Give a polynomial-time algorithm that finds a 3-dimensional matching of size at least $\frac{1}{3}$ times the maximum possible size.

10. Suppose you are given an $n \times n$ *grid graph* G, as in Figure 11.13.

Associated with each node v is a *weight* $w(v)$, which is a nonnegative integer. You may assume that the weights of all nodes are distinct. Your

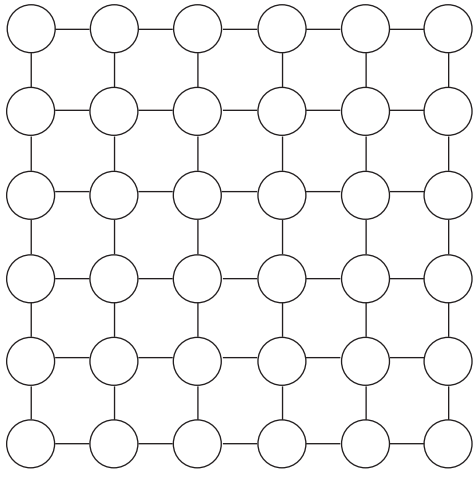

Figure 11.13 A grid graph.

goal is to choose an independent set S of nodes of the grid, so that the sum of the weights of the nodes in S is as large as possible. (The sum of the weights of the nodes in S will be called its *total weight*.)

Consider the following greedy algorithm for this problem.

```
The "heaviest-first" greedy algorithm:
  Start with S equal to the empty set
  While some node remains in G
    Pick a node v_i of maximum weight
    Add v_i to S
    Delete v_i and its neighbors from G
  Endwhile
  Return S
```

(a) Let S be the independent set returned by the "heaviest-first" greedy algorithm, and let T be any other independent set in G. Show that, for each node $v \in T$, either $v \in S$, or there is a node $v' \in S$ so that $w(v) \leq w(v')$ and (v, v') is an edge of G.

(b) Show that the "heaviest-first" greedy algorithm returns an independent set of total weight at least $\frac{1}{4}$ times the maximum total weight of any independent set in the grid graph G.

11. Recall that in the Knapsack Problem, we have n items, each with a weight w_i and a value v_i. We also have a weight bound W, and the problem is to select a set of items S of highest possible value subject to the condition that the total weight does not exceed W—that is, $\sum_{i \in S} w_i \leq W$. Here's one way to look at the approximation algorithm that we designed in this chapter. If we are told there exists a subset $\mathcal{O}$ whose total weight is $\sum_{i \in \mathcal{O}} w_i \leq W$ and whose total value is $\sum_{i \in \mathcal{O}} v_i = V$ for some V, then our approximation algorithm can find a set $\mathcal{A}$ with total weight $\sum_{i \in \mathcal{A}} w_i \leq W$ and total value at least $\sum_{i \in \mathcal{A}} v_i \geq V/(1+\epsilon)$. Thus the algorithm approximates the best value, while keeping the weights strictly under W. (Of course, returning the set $\mathcal{O}$ is always a valid solution, but since the problem is NP-hard, we don't expect to always be able to find $\mathcal{O}$ itself; the approximation bound of $1+\epsilon$ means that other sets $\mathcal{A}$, with slightly less value, can be valid answers as well.)

Now, as is well known, you can always pack a little bit more for a trip just by "sitting on your suitcase"—in other words, by slightly overflowing the allowed weight limit. This too suggests a way of formalizing the approximation question for the Knapsack Problem, but it's the following, different, formalization.

Suppose, as before, that you're given n items with weights and values, as well as parameters W and V; and you're told that there is a subset $\mathcal{O}$ whose total weight is $\sum_{i\in\mathcal{O}} w_i \leq W$ and whose total value is $\sum_{i\in\mathcal{O}} v_i = V$ for some V. For a given fixed $\epsilon > 0$, design a polynomial-time algorithm that finds a subset of items $\mathcal{A}$ such that $\sum_{i\in\mathcal{A}} w_i \leq (1+\epsilon)W$ and $\sum_{i\in\mathcal{A}} v_i \geq V$. In other words, you want $\mathcal{A}$ to achieve at least as high a total value as the given bound V, but you're allowed to exceed the weight limit W by a factor of $1+\epsilon$.

Example. Suppose you're given four items, with weights and values as follows:

$$(w_1, v_1) = (5, 3), (w_2, v_2) = (4, 6)$$

$$(w_3, v_3) = (1, 4), (w_4, v_4) = (6, 11)$$

You're also given $W = 10$ and $V = 13$ (since, indeed, the subset consisting of the first three items has total weight at most 10 and has value 13). Finally, you're given $\epsilon = .1$. This means you need to find (via your approximation algorithm) a subset of weight at most $(1 + .1) * 10 = 11$ and value at least 13. One valid solution would be the subset consisting of the first and fourth items, with value $14 \geq 13$. (Note that this is a case where you're able to achieve a value strictly greater than V, since you're allowed to slightly overfill the knapsack.)

12. Consider the following problem. There is a set U of n nodes, which we can think of as users (e.g., these are locations that need to access a service, such as a Web server). You would like to place servers at multiple locations. Suppose you are given a set S possible sites that would be willing to act as locations for the servers. For each site $s \in S$, there is a fee $f_s \geq 0$ for placing a server at that location. Your goal will be to approximately minimize the cost while providing the service to each of the customers. So far this is very much like the Set Cover Problem: The places s are sets, the weight of set s is f_s, and we want to select a collection of sets that covers all users. There is one extra complication: Users $u \in U$ can be served from multiple sites, but there is an associated cost d_{us} for serving user u from site s. When the value d_{us} is very high, we do not want to serve user u from site s; and in general the service cost d_{us} serves as an incentive to serve customers from "nearby" servers whenever possible.

So here is the question, which we call the Facility Location Problem: Given the sets U and S, and costs f and d, you need to select a subset $A \subseteq S$ at which to place servers (at a cost of $\sum_{s\in A} f_s$), and assign each user u to the active server where it is cheapest to be served, $\min_{s\in A} d_{us}$. The goal

is to minimize the overall cost $\sum_{s \in A} f_s + \sum_{u \in U} \min_{s \in A} d_{us}$. Give an $H(n)$-approximation for this problem.

(Note that if all service costs d_{us} are 0 or infinity, then this problem is exactly the Set Cover Problem: f_s is the cost of the set named s, and d_{us} is 0 if node u is in set s, and infinity otherwise.)

Notes and Further Reading

The design of approximation algorithms for NP-hard problems is an active area of research, and it is the focus of a book of surveys edited by Hochbaum (1996) and a text by Vazirani (2001).

The greedy algorithm for load balancing and its analysis is due to Graham (1966, 1969); in fact, he proved that when the jobs are first sorted in descending order of size, the greedy algorithm achieves an assignment within a factor $\frac{4}{3}$ of optimal. (In the text, we give a simpler proof for the weaker bound of $\frac{3}{2}$.) Using more complicated algorithms, even stronger approximation guarantees can be proved for this problem (Hochbaum and Shmoys 1987; Hall 1996). The techniques used for these stronger load balancing approximation algorithms are also closely related to the method described in the text for designing arbitrarily good approximations for the Knapsack Problem.

The approximation algorithm for the Center Selection Problem follows the approach of Hochbaum and Shmoys (1985) and Dyer and Frieze (1985). Other geometric location problems of this flavor are discussed by Bern and Eppstein (1996) and in the book of surveys edited by Drezner (1995).

The greedy algorithm for Set Cover and its analysis are due independently to Johnson (1974), Lovász (1975), and Chvatal (1979). Further results for the Set Cover Problem are discussed in the survey by Hochbaum (1996).

As mentioned in the text, the pricing method for designing approximation algorithms is also referred to as the *primal-dual method* and can be motivated using linear programming. This latter perspective is the subject of the survey by Goemans and Williamson (1996). The pricing algorithm to approximate the Weighted Vertex Cover Problem is due to Bar-Yehuda and Even (1981).

The greedy algorithm for the disjoint paths problem is due to Kleinberg and Tardos (1995); the pricing-based approximation algorithm for the case when multiple paths can share an edge is due to Awerbuch, Azar, and Plotkin (1993). Algorithms have been developed for many other variants of the Disjoint Paths Problem; see the book of surveys edited by Korte et al. (1990) for a discussion of cases that can be solved optimally in polynomial time, and Plotkin (1995) and Kleinberg (1996) for surveys of work on approximation.

The linear programming rounding algorithm for the Weighted Vertex Cover Problem is due to Hochbaum (1982). The rounding algorithm for Generalized Load Balancing is due to Lenstra, Shmoys, and Tardos (1990); see the survey by Hall (1996) for other results in this vein. As discussed in the text, these two results illustrate a widely used method for designing approximation algorithms: One sets up an integer programming formulation for the problem, transforms it to a related (but not equivalent) linear programming problem, and then rounds the resulting solution. Vazirani (2001) discusses many further applications of this technique.

Local search and randomization are two other powerful techniques for designing approximation algorithms; we discuss these connections in the next two chapters.

One topic that we do not cover in this book is *inapproximability*. Just as one can prove that a given NP-hard problem can be approximated to within a certain factor in polynomial time, one can also sometimes establish lower bounds, showing that if the problem could be approximated to within better than some factor c in polynomial time, then it could be solved optimally, thereby proving $\mathcal{P} = \mathcal{NP}$. There is a growing body of work that establishes such limits to approximability for many NP-hard problems. In certain cases, these positive and negative results have lined up perfectly to produce an *approximation threshold*, establishing for certain problems that there is a polynomial-time approximation algorithm to within some factor c, and it is impossible to do better unless $\mathcal{P} = \mathcal{NP}$. Some of the early results on inapproximability were not very difficult to prove, but more recent work has introduced powerful techniques that become quite intricate. This topic is covered in the survey by Arora and Lund (1996).

Notes on the Exercises Exercises 4 and 12 are based on results of Dorit Hochbaum. Exercise 11 is based on results of Sartaj Sahni, Oscar Ibarra, and Chul Kim, and of Dorit Hochbaum and David Shmoys.