Chapter 4

Greedy Algorithms

In *Wall Street*, that iconic movie of the 1980s, Michael Douglas gets up in front of a room full of stockholders and proclaims, "Greed . . . is good. Greed is right. Greed works." In this chapter, we'll be taking a much more understated perspective as we investigate the pros and cons of short-sighted greed in the design of algorithms. Indeed, our aim is to approach a number of different computational problems with a recurring set of questions: Is greed good? Does greed work?

It is hard, if not impossible, to define precisely what is meant by a *greedy algorithm*. An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion. One can often design many different greedy algorithms for the same problem, each one locally, incrementally optimizing some different measure on its way to a solution.

When a greedy algorithm succeeds in solving a nontrivial problem optimally, it typically implies something interesting and useful about the structure of the problem itself; there is a local decision rule that one can use to construct optimal solutions. And as we'll see later, in Chapter 11, the same is true of problems in which a greedy algorithm can produce a solution that is guaranteed to be *close* to optimal, even if it does not achieve the precise optimum. These are the kinds of issues we'll be dealing with in this chapter. It's easy to invent greedy algorithms for almost any problem; finding cases in which they work well, and proving that they work well, is the interesting challenge.

The first two sections of this chapter will develop two basic methods for proving that a greedy algorithm produces an optimal solution to a problem. One can view the first approach as establishing that *the greedy algorithm stays ahead*. By this we mean that if one measures the greedy algorithm's progress in a step-by-step fashion, one sees that it does better than any other algorithm at each step; it then follows that it produces an optimal solution. The second approach is known as an *exchange argument*, and it is more general: one considers any possible solution to the problem and gradually transforms it into the solution found by the greedy algorithm without hurting its quality. Again, it will follow that the greedy algorithm must have found a solution that is at least as good as any other solution.

Following our introduction of these two styles of analysis, we focus on several of the most well-known applications of greedy algorithms: *shortest paths in a graph*, the *Minimum Spanning Tree Problem*, and the construction of *Huffman codes* for performing data compression. They each provide nice examples of our analysis techniques. We also explore an interesting relationship between minimum spanning trees and the long-studied problem of *clustering*. Finally, we consider a more complex application, the *Minimum-Cost Arborescence Problem*, which further extends our notion of what a greedy algorithm is.

4.1 Interval Scheduling: The Greedy Algorithm Stays Ahead

Let's recall the Interval Scheduling Problem, which was the first of the five representative problems we considered in Chapter 1. We have a set of requests $\{1, 2, \ldots, n\}$; the i^{th} request corresponds to an interval of time starting at $s(i)$ and finishing at $f(i)$. (Note that we are slightly changing the notation from Section 1.2, where we used s_i rather than $s(i)$ and f_i rather than $f(i)$. This change of notation will make things easier to talk about in the proofs.) We'll say that a subset of the requests is *compatible* if no two of them overlap in time, and our goal is to accept as large a compatible subset as possible. Compatible sets of maximum size will be called *optimal*.

Designing a Greedy Algorithm

Using the Interval Scheduling Problem, we can make our discussion of greedy algorithms much more concrete. The basic idea in a greedy algorithm for interval scheduling is to use a simple rule to select a first request i_1. Once a request i_1 is accepted, we reject all requests that are not compatible with i_1. We then select the next request i_2 to be accepted, and again reject all requests that are not compatible with i_2. We continue in this fashion until we run out of requests. The challenge in designing a good greedy algorithm is in deciding which simple rule to use for the selection—and there are many natural rules for this problem that do not give good solutions.

Let's try to think of some of the most natural rules and see how they work.

- The most obvious rule might be to always select the available request that starts earliest—that is, the one with minimal start time $s(i)$. This way our resource starts being used as quickly as possible.

  This method does not yield an optimal solution. If the earliest request i is for a very long interval, then by accepting request i we may have to reject a lot of requests for shorter time intervals. Since our goal is to satisfy as many requests as possible, we will end up with a suboptimal solution. In a really bad case—say, when the finish time $f(i)$ is the maximum among all requests—the accepted request i keeps our resource occupied for the whole time. In this case our greedy method would accept a single request, while the optimal solution could accept many. Such a situation is depicted in Figure 4.1(a).

- This might suggest that we should start out by accepting the request that requires the smallest interval of time—namely, the request for which $f(i) - s(i)$ is as small as possible. As it turns out, this is a somewhat better rule than the previous one, but it still can produce a suboptimal schedule. For example, in Figure 4.1(b), accepting the short interval in the middle would prevent us from accepting the other two, which form an optimal solution.

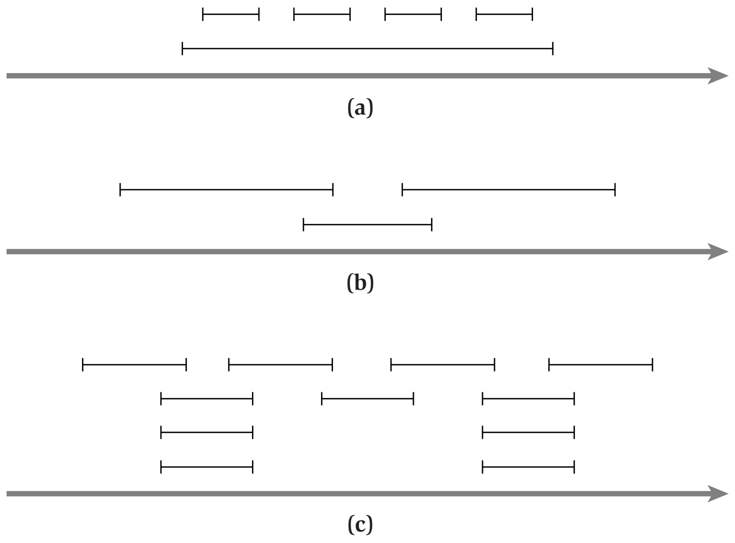

Figure 4.1 Some instances of the Interval Scheduling Problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the interval with the fewest conflicts.

- In the previous greedy rule, our problem was that the second request competes with both the first and the third—that is, accepting this request made us reject two other requests. We could design a greedy algorithm that is based on this idea: for each request, we count the number of other requests that are not compatible, and accept the request that has the fewest number of noncompatible requests. (In other words, we select the interval with the fewest "conflicts.") This greedy choice would lead to the optimum solution in the previous example. In fact, it is quite a bit harder to design a bad example for this rule; but it can be done, and we've drawn an example in Figure 4.1(c). The unique optimal solution in this example is to accept the four requests in the top row. The greedy method suggested here accepts the middle request in the second row and thereby ensures a solution of size no greater than three.

A greedy rule that does lead to the optimal solution is based on a fourth idea: we should accept first the request that finishes first, that is, the request i for which $f(i)$ is as small as possible. This is also quite a natural idea: we ensure that our resource becomes free as soon as possible while still satisfying one request. In this way we can maximize the time left to satisfy other requests.

Let us state the algorithm a bit more formally. We will use R to denote the set of requests that we have neither accepted nor rejected yet, and use A to denote the set of accepted requests. For an example of how the algorithm runs, see Figure 4.2.

```
Initially let $R$ be the set of all requests, and let $A$ be empty
While $R$ is not yet empty
  Choose a request $i \in R$ that has the smallest finishing time
  Add request $i$ to $A$
  Delete all requests from $R$ that are not compatible with request $i$
EndWhile
Return the set $A$ as the set of accepted requests
```

Analyzing the Algorithm

While this greedy method is quite natural, it is certainly not obvious that it returns an optimal set of intervals. Indeed, it would only be sensible to reserve judgment on its optimality: the ideas that led to the previous nonoptimal versions of the greedy method also seemed promising at first.

As a start, we can immediately declare that the intervals in the set A returned by the algorithm are all compatible.

(4.1) *A is a compatible set of requests.*

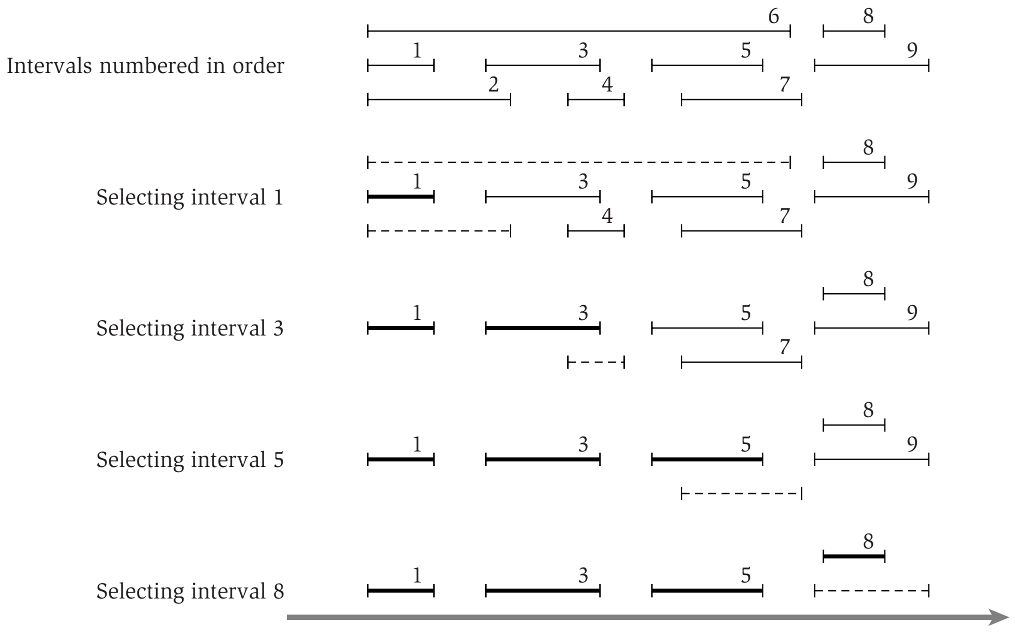

Figure 4.2 Sample run of the Interval Scheduling Algorithm. At each step the selected intervals are darker lines, and the intervals deleted at the corresponding step are indicated with dashed lines.

What we need to show is that this solution is optimal. So, for purposes of comparison, let $\mathcal{O}$ be an optimal set of intervals. Ideally one might want to show that $A = \mathcal{O}$, but this is too much to ask: there may be many optimal solutions, and at best A is equal to a single one of them. So instead we will simply show that $|A| = |\mathcal{O}|$, that is, that A contains the same number of intervals as $\mathcal{O}$ and hence is also an optimal solution.

The idea underlying the proof, as we suggested initially, will be to find a sense in which our greedy algorithm "stays ahead" of this solution $\mathcal{O}$. We will compare the partial solutions that the greedy algorithm constructs to initial segments of the solution $\mathcal{O}$, and show that the greedy algorithm is doing better in a step-by-step fashion.

We introduce some notation to help with this proof. Let $i_1, \ldots, i_k$ be the set of requests in A in the order they were added to A. Note that $|A| = k$. Similarly, let the set of requests in $\mathcal{O}$ be denoted by $j_1, \ldots, j_m$. Our goal is to prove that $k = m$. Assume that the requests in $\mathcal{O}$ are also ordered in the natural left-to-right order of the corresponding intervals, that is, in the order of the start and finish points. Note that the requests in $\mathcal{O}$ are compatible, which implies that the start points have the same order as the finish points.

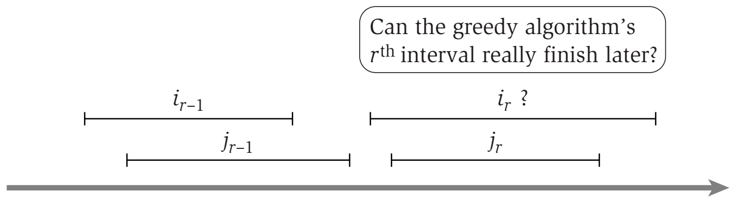
Can the greedy algorithm's r^{th} interval really finish later?
i_{r-1}
i_r ?
j_{r-1}
j_r

Figure 4.3 The inductive step in the proof that the greedy algorithm stays ahead.

Our intuition for the greedy method came from wanting our resource to become free again as soon as possible after satisfying the first request. And indeed, our greedy rule guarantees that $f(i_1) \le f(j_1)$. This is the sense in which we want to show that our greedy rule "stays ahead"—that each of its intervals finishes at least as soon as the corresponding interval in the set $\mathcal{O}$. Thus we now prove that for each $r \ge 1$, the r^{th} accepted request in the algorithm's schedule finishes no later than the r^{th} request in the optimal schedule.

(4.2) *For all indices $r \le k$ we have $f(i_r) \le f(j_r)$.*

Proof. We will prove this statement by induction. For $r = 1$ the statement is clearly true: the algorithm starts by selecting the request i_1 with minimum finish time.

Now let $r > 1$. We will assume as our induction hypothesis that the statement is true for $r - 1$, and we will try to prove it for r. As shown in Figure 4.3, the induction hypothesis lets us assume that $f(i_{r-1}) \le f(j_{r-1})$. In order for the algorithm's r^{th} interval not to finish earlier as well, it would need to "fall behind" as shown. But there's a simple reason why this could not happen: rather than choose a later-finishing interval, the greedy algorithm always has the option (at worst) of choosing j_r and thus fulfilling the induction step.

We can make this argument precise as follows. We know (since $\mathcal{O}$ consists of compatible intervals) that $f(j_{r-1}) \le s(j_r)$. Combining this with the induction hypothesis $f(i_{r-1}) \le f(j_{r-1})$, we get $f(i_{r-1}) \le s(j_r)$. Thus the interval j_r is in the set R of available intervals at the time when the greedy algorithm selects i_r. The greedy algorithm selects the available interval with *smallest* finish time; since interval j_r is one of these available intervals, we have $f(i_r) \le f(j_r)$. This completes the induction step. ■

Thus we have formalized the sense in which the greedy algorithm is remaining ahead of $\mathcal{O}$: for each r, the r^{th} interval it selects finishes at least as soon as the r^{th} interval in $\mathcal{O}$. We now see why this implies the optimality of the greedy algorithm's set A.

(4.3) *The greedy algorithm returns an optimal set A.*

Proof. We will prove the statement by contradiction. If A is not optimal, then an optimal set $\mathcal{O}$ must have more requests, that is, we must have $m > k$. Applying (4.2) with $r = k$, we get that $f(i_k) \leq f(j_k)$. Since $m > k$, there is a request j_{k+1} in $\mathcal{O}$. This request starts after request j_k ends, and hence after i_k ends. So after deleting all requests that are not compatible with requests $i_1, \ldots, i_k$, the set of possible requests R still contains j_{k+1}. But the greedy algorithm stops with request i_k, and it is only supposed to stop when R is empty—a contradiction. ∎

Implementation and Running Time We can make our algorithm run in time $O(n \log n)$ as follows. We begin by sorting the n requests in order of finishing time and labeling them in this order; that is, we will assume that $f(i) \leq f(j)$ when $i < j$. This takes time $O(n \log n)$. In an additional $O(n)$ time, we construct an array $S[1 \ldots n]$ with the property that $S[i]$ contains the value $s(i)$.

We now select requests by processing the intervals in order of increasing $f(i)$. We always select the first interval; we then iterate through the intervals in order until reaching the first interval j for which $s(j) \geq f(1)$; we then select this one as well. More generally, if the most recent interval we've selected ends at time f, we continue iterating through subsequent intervals until we reach the first j for which $s(j) \geq f$. In this way, we implement the greedy algorithm analyzed above in one pass through the intervals, spending constant time per interval. Thus this part of the algorithm takes time $O(n)$.

Extensions

The Interval Scheduling Problem we considered here is a quite simple scheduling problem. There are many further complications that could arise in practical settings. The following point out issues that we will see later in the book in various forms.

- In defining the problem, we assumed that all requests were known to the scheduling algorithm when it was choosing the compatible subset. It would also be natural, of course, to think about the version of the problem in which the scheduler needs to make decisions about accepting or rejecting certain requests before knowing about the full set of requests. Customers (requestors) may well be impatient, and they may give up and leave if the scheduler waits too long to gather information about all other requests. An active area of research is concerned with such *on-line* algorithms, which must make decisions as time proceeds, without knowledge of future input.

- Our goal was to maximize the number of satisfied requests. But we could picture a situation in which each request has a different value to us. For example, each request i could also have a value v_i (the amount gained by satisfying request i), and the goal would be to maximize our income: the sum of the values of all satisfied requests. This leads to the *Weighted Interval Scheduling Problem*, the second of the representative problems we described in Chapter 1.

There are many other variants and combinations that can arise. We now discuss one of these further variants in more detail, since it forms another case in which a greedy algorithm can be used to produce an optimal solution.

A Related Problem: Scheduling All Intervals

The Problem In the Interval Scheduling Problem, there is a single resource and many requests in the form of time intervals, so we must choose which requests to accept and which to reject. A related problem arises if we have many identical resources available and we wish to schedule *all* the requests using as few resources as possible. Because the goal here is to partition all intervals across multiple resources, we will refer to this as the *Interval Partitioning* Problem.[1]

For example, suppose that each request corresponds to a lecture that needs to be scheduled in a classroom for a particular interval of time. We wish to satisfy all these requests, using as few classrooms as possible. The classrooms at our disposal are thus the multiple resources, and the basic constraint is that any two lectures that overlap in time must be scheduled in different classrooms. Equivalently, the interval requests could be jobs that need to be processed for a specific period of time, and the resources are machines capable of handling these jobs. Much later in the book, in Chapter 10, we will see a different application of this problem in which the intervals are routing requests that need to be allocated bandwidth on a fiber-optic cable.

As an illustration of the problem, consider the sample instance in Figure 4.4(a). The requests in this example can all be scheduled using three resources; this is indicated in Figure 4.4(b), where the requests are rearranged into three rows, each containing a set of nonoverlapping intervals. In general, one can imagine a solution using k resources as a rearrangement of the requests into k rows of nonoverlapping intervals: the first row contains all the intervals

[1] The problem is also referred to as the *Interval Coloring Problem*; the terminology arises from thinking of the different resources as having distinct colors—all the intervals assigned to a particular resource are given the corresponding color.

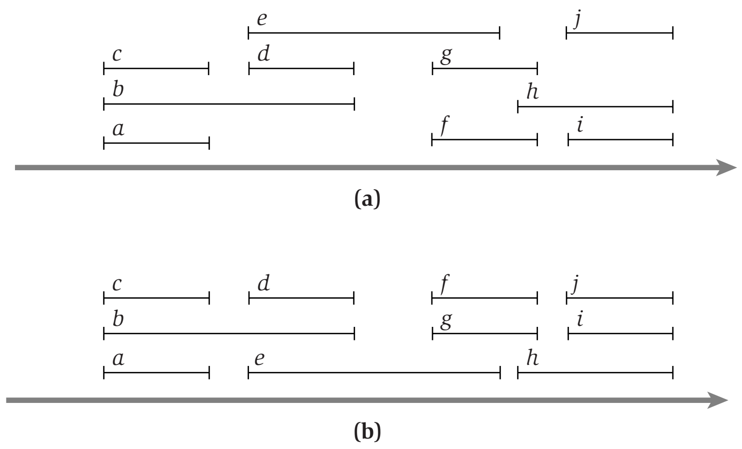

Figure 4.4 (a) An instance of the Interval Partitioning Problem with ten intervals (*a* through *j*). (b) A solution in which all intervals are scheduled using three resources: each row represents a set of intervals that can all be scheduled on a single resource.

assigned to the first resource, the second row contains all those assigned to the second resource, and so forth.

Now, is there any hope of using just two resources in this sample instance? Clearly the answer is no. We need at least three resources since, for example, intervals a, b, and c all pass over a common point on the time-line, and hence they all need to be scheduled on different resources. In fact, one can make this last argument in general for any instance of Interval Partitioning. Suppose we define the *depth* of a set of intervals to be the maximum number that pass over any single point on the time-line. Then we claim

(4.4) *In any instance of Interval Partitioning, the number of resources needed is at least the depth of the set of intervals.*

Proof. Suppose a set of intervals has depth d, and let $I_1, \ldots, I_d$ all pass over a common point on the time-line. Then each of these intervals must be scheduled on a different resource, so the whole instance needs at least d resources. ■

We now consider two questions, which turn out to be closely related. First, can we design an efficient algorithm that schedules all intervals using the minimum possible number of resources? Second, is there always a schedule using a number of resources that is *equal* to the depth? In effect, a positive answer to this second question would say that the only obstacles to partitioning intervals are purely local—a set of intervals all piled over the same point. It's not immediately clear that there couldn't exist other, "long-range" obstacles that push the number of required resources even higher.

We now design a simple greedy algorithm that schedules all intervals using a number of resources equal to the depth. This immediately implies the optimality of the algorithm: in view of (4.4), no solution could use a number of resources that is smaller than the depth. The analysis of our algorithm will therefore illustrate another general approach to proving optimality: one finds a simple, "structural" bound asserting that every possible solution must have at least a certain value, and then one shows that the algorithm under consideration always achieves this bound.

Designing the Algorithm Let d be the depth of the set of intervals; we show how to assign a *label* to each interval, where the labels come from the set of numbers $\{1, 2, \ldots, d\}$, and the assignment has the property that overlapping intervals are labeled with different numbers. This gives the desired solution, since we can interpret each number as the name of a resource, and the label of each interval as the name of the resource to which it is assigned.

The algorithm we use for this is a simple one-pass greedy strategy that orders intervals by their starting times. We go through the intervals in this order, and try to assign to each interval we encounter a label that hasn't already been assigned to any previous interval that overlaps it. Specifically, we have the following description.

```
Sort the intervals by their start times, breaking ties arbitrarily
Let $I_1, I_2, \ldots, I_n$ denote the intervals in this order
For $j = 1, 2, 3, \ldots, n$
  For each interval $I_i$ that precedes $I_j$ in sorted order and overlaps it
     Exclude the label of $I_i$ from consideration for $I_j$
  Endfor
  If there is any label from $\{1, 2, \ldots, d\}$ that has not been excluded then
    Assign a nonexcluded label to $I_j$
  Else
    Leave $I_j$ unlabeled
  Endif
Endfor
```

Analyzing the Algorithm We claim the following.

(4.5) *If we use the greedy algorithm above, every interval will be assigned a label, and no two overlapping intervals will receive the same label.*

Proof. First let's argue that no interval ends up unlabeled. Consider one of the intervals I_j, and suppose there are t intervals earlier in the sorted order that overlap it. These t intervals, together with I_j, form a set of $t + 1$ intervals that all pass over a common point on the time-line (namely, the start time of

I_j), and so $t + 1 \leq d$. Thus $t \leq d - 1$. It follows that at least one of the d labels is not excluded by this set of t intervals, and so there is a label that can be assigned to I_j.

Next we claim that no two overlapping intervals are assigned the same label. Indeed, consider any two intervals I and I' that overlap, and suppose I precedes I' in the sorted order. Then when I' is considered by the algorithm, I is in the set of intervals whose labels are excluded from consideration; consequently, the algorithm will not assign to I' the label that it used for I. ■

The algorithm and its analysis are very simple. Essentially, if you have d labels at your disposal, then as you sweep through the intervals from left to right, assigning an available label to each interval you encounter, you can never reach a point where all the labels are currently in use.

Since our algorithm is using d labels, we can use (4.4) to conclude that it is, in fact, always using the minimum possible number of labels. We sum this up as follows.

(4.6) *The greedy algorithm above schedules every interval on a resource, using a number of resources equal to the depth of the set of intervals. This is the optimal number of resources needed.*

4.2 Scheduling to Minimize Lateness: An Exchange Argument

We now discuss a scheduling problem related to the one with which we began the chapter. Despite the similarities in the problem formulation and in the greedy algorithm to solve it, the proof that this algorithm is optimal will require a more sophisticated kind of analysis.

The Problem

Consider again a situation in which we have a single resource and a set of n requests to use the resource for an interval of time. Assume that the resource is available starting at time s. In contrast to the previous problem, however, each request is now more flexible. Instead of a start time and finish time, the request i has a deadline d_i, and it requires a contiguous time interval of length t_i, but it is willing to be scheduled at any time before the deadline. Each accepted request must be assigned an interval of time of length t_i, and different requests must be assigned nonoverlapping intervals.

There are many objective functions we might seek to optimize when faced with this situation, and some are computationally much more difficult than

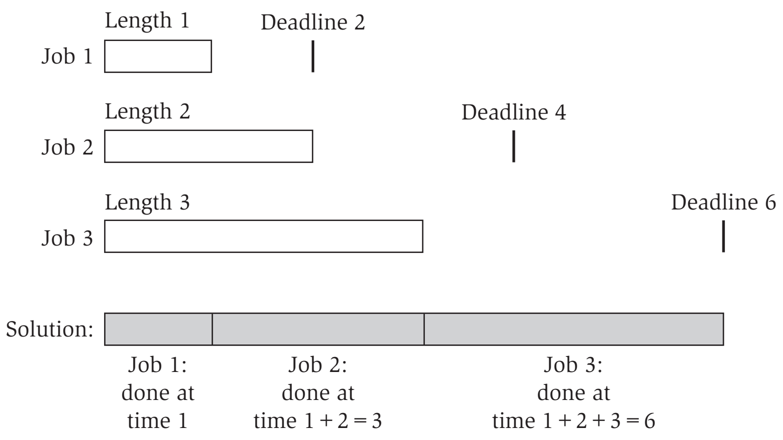

Figure 4.5 A sample instance of scheduling to minimize lateness.

others. Here we consider a very natural goal that can be optimized by a greedy algorithm. Suppose that we plan to satisfy each request, but we are allowed to let certain requests run late. Thus, beginning at our overall start time s, we will assign each request i an interval of time of length t_i; let us denote this interval by $[s(i), f(i)]$, with $f(i) = s(i) + t_i$. Unlike the previous problem, then, the algorithm must actually determine a start time (and hence a finish time) for each interval.

We say that a request i is *late* if it misses the deadline, that is, if $f(i) > d_i$. The *lateness* of such a request i is defined to be $l_i = f(i) - d_i$. We will say that $l_i = 0$ if request i is not late. The goal in our new optimization problem will be to schedule all requests, using nonoverlapping intervals, so as to minimize the *maximum lateness*, $L = \max_i l_i$. This problem arises naturally when scheduling jobs that need to use a single machine, and so we will refer to our requests as *jobs*.

Figure 4.5 shows a sample instance of this problem, consisting of three jobs: the first has length $t_1 = 1$ and deadline $d_1 = 2$; the second has $t_2 = 2$ and $d_2 = 4$; and the third has $t_3 = 3$ and $d_3 = 6$. It is not hard to check that scheduling the jobs in the order $1, 2, 3$ incurs a maximum lateness of 0.

Designing the Algorithm

What would a greedy algorithm for this problem look like? There are several natural greedy approaches in which we look at the data (t_i, d_i) about the jobs and use this to order them according to some simple rule.

- One approach would be to schedule the jobs in order of increasing length t_i, so as to get the short jobs out of the way quickly. This immediately looks too simplistic, since it completely ignores the deadlines of the jobs. And indeed, consider a two-job instance where the first job has $t_1 = 1$ and $d_1 = 100$, while the second job has $t_2 = 10$ and $d_2 = 10$. Then the second job has to be started right away if we want to achieve lateness $L = 0$, and scheduling the second job first is indeed the optimal solution.

- The previous example suggests that we should be concerned about jobs whose available *slack time* $d_i - t_i$ is very small—they're the ones that need to be started with minimal delay. So a more natural greedy algorithm would be to sort jobs in order of increasing slack $d_i - t_i$.

  Unfortunately, this greedy rule fails as well. Consider a two-job instance where the first job has $t_1 = 1$ and $d_1 = 2$, while the second job has $t_2 = 10$ and $d_2 = 10$. Sorting by increasing slack would place the second job first in the schedule, and the first job would incur a lateness of 9. (It finishes at time 11, nine units beyond its deadline.) On the other hand, if we schedule the first job first, then it finishes on time and the second job incurs a lateness of only 1.

There is, however, an equally basic greedy algorithm that always produces an optimal solution. We simply sort the jobs in increasing order of their deadlines d_i, and schedule them in this order. (This rule is often called *Earliest Deadline First*.) There is an intuitive basis to this rule: we should make sure that jobs with earlier deadlines get completed earlier. At the same time, it's a little hard to believe that this algorithm always produces optimal solutions—specifically because it never looks at the lengths of the jobs. Earlier we were skeptical of the approach that sorted by length on the grounds that it threw away half the input data (i.e., the deadlines); but now we're considering a solution that throws away the other half of the data. Nevertheless, Earliest Deadline First does produce optimal solutions, and we will now prove this.

First we specify some notation that will be useful in talking about the algorithm. By renaming the jobs if necessary, we can assume that the jobs are labeled in the order of their deadlines, that is, we have

$$d_1 \leq \ldots \leq d_n.$$

We will simply schedule all jobs in this order. Again, let s be the start time for all jobs. Job 1 will start at time $s = s(1)$ and end at time $f(1) = s(1) + t_1$; Job 2 will start at time $s(2) = f(1)$ and end at time $f(2) = s(2) + t_2$; and so forth. We will use f to denote the finishing time of the last scheduled job. We write this algorithm here.

```
Order the jobs in order of their deadlines
Assume for simplicity of notation that $d_1 \leq \ldots \leq d_n$
Initially, $f = s$
```

```
Consider the jobs $i = 1, \ldots, n$ in this order
   Assign job $i$ to the time interval from $s(i) = f$ to $f(i) = f + t_i$
   Let $f = f + t_i$
End
Return the set of scheduled intervals $[s(i), f(i)]$ for $i = 1, \ldots, n$
```

Analyzing the Algorithm

To reason about the optimality of the algorithm, we first observe that the schedule it produces has no "gaps"—times when the machine is not working yet there are jobs left. The time that passes during a gap will be called *idle time:* there is work to be done, yet for some reason the machine is sitting idle. Not only does the schedule A produced by our algorithm have no idle time; it is also very easy to see that there is an optimal schedule with this property. We do not write down a proof for this.

(4.7) *There is an optimal schedule with no idle time.*

Now, how can we prove that our schedule A is optimal, that is, its maximum lateness L is as small as possible? As in previous analyses, we will start by considering an optimal schedule $\mathcal{O}$. Our plan here is to gradually modify $\mathcal{O}$, preserving its optimality at each step, but eventually transforming it into a schedule that is identical to the schedule A found by the greedy algorithm. We refer to this type of analysis as an *exchange argument*, and we will see that it is a powerful way to think about greedy algorithms in general.

We first try characterizing schedules in the following way. We say that a schedule A' has an *inversion* if a job i with deadline d_i is scheduled before another job j with earlier deadline $d_j < d_i$. Notice that, by definition, the schedule A produced by our algorithm has no inversions. If there are jobs with identical deadlines then there can be many different schedules with no inversions. However, we can show that all these schedules have the same maximum lateness L.

(4.8) *All schedules with no inversions and no idle time have the same maximum lateness.*

Proof. If two different schedules have neither inversions nor idle time, then they might not produce exactly the same order of jobs, but they can only differ in the order in which jobs with identical deadlines are scheduled. Consider such a deadline d. In both schedules, the jobs with deadline d are all scheduled consecutively (after all jobs with earlier deadlines and before all jobs with later deadlines). Among the jobs with deadline d, the last one has the greatest lateness, and this lateness does not depend on the order of the jobs. ■

The main step in showing the optimality of our algorithm is to establish that there is an optimal schedule that has no inversions and no idle time. To do this, we will start with any optimal schedule having no idle time; we will then convert it into a schedule with no inversions without increasing its maximum lateness. Thus the resulting scheduling after this conversion will be optimal as well.

(4.9) *There is an optimal schedule that has no inversions and no idle time.*

Proof. By (4.7), there is an optimal schedule $\mathcal{O}$ with no idle time. The proof will consist of a sequence of statements. The first of these is simple to establish.

(a) *If $\mathcal{O}$ has an inversion, then there is a pair of jobs i and j such that j is scheduled immediately after i and has $d_j < d_i$.*

Indeed, consider an inversion in which a job a is scheduled sometime before a job b, and $d_a > d_b$. If we advance in the scheduled order of jobs from a to b one at a time, there has to come a point at which the deadline we see decreases for the first time. This corresponds to a pair of consecutive jobs that form an inversion.

Now suppose $\mathcal{O}$ has at least one inversion, and by (a), let i and j be a pair of inverted requests that are consecutive in the scheduled order. We will decrease the number of inversions in $\mathcal{O}$ by swapping the requests i and j in the schedule $\mathcal{O}$. The pair (i, j) formed an inversion in $\mathcal{O}$, this inversion is eliminated by the swap, and no new inversions are created. Thus we have

(b) *After swapping i and j we get a schedule with one less inversion.*

The hardest part of this proof is to argue that the inverted schedule is also optimal.

(c) *The new swapped schedule has a maximum lateness no larger than that of $\mathcal{O}$.*

It is clear that if we can prove (c), then we are done. The initial schedule $\mathcal{O}$ can have at most $\binom{n}{2}$ inversions (if all pairs are inverted), and hence after at most $\binom{n}{2}$ swaps we get an optimal schedule with no inversions.

So we now conclude by proving (c), showing that by swapping a pair of consecutive, inverted jobs, we do not increase the maximum lateness L of the schedule. ∎

Proof of (c). We invent some notation to describe the schedule $\mathcal{O}$: assume that each request r is scheduled for the time interval $[s(r), f(r)]$ and has lateness l'_r. Let $L' = \max_r l'_r$ denote the maximum lateness of this schedule.

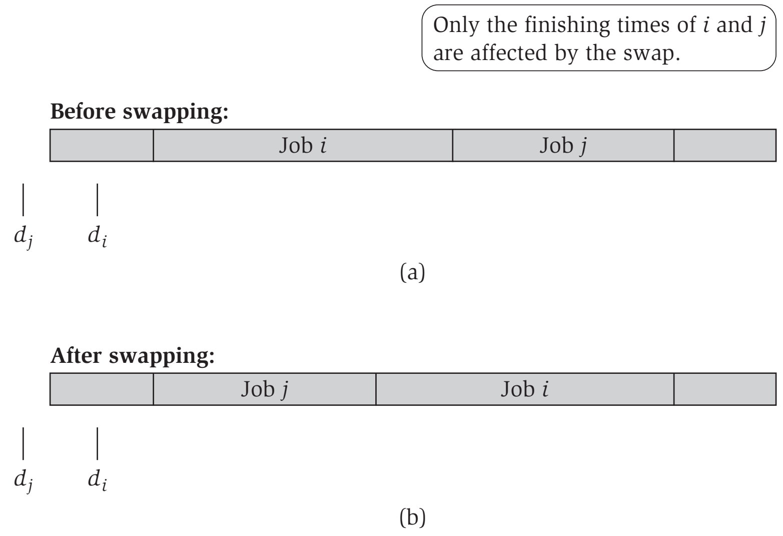

Figure 4.6 The effect of swapping two consecutive, inverted jobs.

Let $\overline{\mathcal{O}}$ denote the swapped schedule; we will use $\bar{s}(r)$, $\bar{f}(r)$, $\bar{l}_r$, and $\bar{L}$ to denote the corresponding quantities in the swapped schedule.

Now recall our two adjacent, inverted jobs i and j. The situation is roughly as pictured in Figure 4.6. The finishing time of j before the swap is exactly equal to the finishing time of i after the swap. Thus all jobs other than jobs i and j finish at the same time in the two schedules. Moreover, job j will get finished earlier in the new schedule, and hence the swap does not increase the lateness of job j.

Thus the only thing to worry about is job i: its lateness may have been increased, and what if this actually raises the maximum lateness of the whole schedule? After the swap, job i finishes at time $f(j)$, when job j was finished in the schedule $\mathcal{O}$. If job i is late in this new schedule, its lateness is $\bar{l}_i = \bar{f}(i) - d_i = f(j) - d_i$. But the crucial point is that i cannot be *more late* in the schedule $\overline{\mathcal{O}}$ than j was in the schedule $\mathcal{O}$. Specifically, our assumption $d_i > d_j$ implies that

$$\bar{l}_i = f(j) - d_i < f(j) - d_j = l'_j.$$

Since the lateness of the schedule $\mathcal{O}$ was $L' \geq l'_j > \bar{l}_i$, this shows that the swap does not increase the maximum lateness of the schedule. ■

The optimality of our greedy algorithm now follows immediately.

(4.10) *The schedule A produced by the greedy algorithm has optimal maximum lateness L.*

Proof. Statement (4.9) proves that an optimal schedule with no inversions exists. Now by (4.8) all schedules with no inversions have the same maximum lateness, and so the schedule obtained by the greedy algorithm is optimal. ■

Extensions

There are many possible generalizations of this scheduling problem. For example, we assumed that all jobs were available to start at the common start time s. A natural, but harder, version of this problem would contain requests i that, in addition to the deadline d_i and the requested time t_i, would also have an earliest possible starting time r_i. This earliest possible starting time is usually referred to as the *release time*. Problems with release times arise naturally in scheduling problems where requests can take the form: Can I reserve the room for a two-hour lecture, sometime between 1 P.M. and 5 P.M.? Our proof that the greedy algorithm finds an optimal solution relied crucially on the fact that all jobs were available at the common start time s. (Do you see where?) Unfortunately, as we will see later in the book, in Chapter 8, this more general version of the problem is much more difficult to solve optimally.

4.3 Optimal Caching: A More Complex Exchange Argument

We now consider a problem that involves processing a sequence of requests of a different form, and we develop an algorithm whose analysis requires a more subtle use of the exchange argument. The problem is that of *cache maintenance*.

The Problem

To motivate caching, consider the following situation. You're working on a long research paper, and your draconian library will only allow you to have eight books checked out at once. You know that you'll probably need more than this over the course of working on the paper, but at any point in time, you'd like to have ready access to the eight books that are most relevant at that time. How should you decide which books to check out, and when should you return some in exchange for others, to minimize the number of times you have to exchange a book at the library?

This is precisely the problem that arises when dealing with a *memory hierarchy*: There is a small amount of data that can be accessed very quickly, and a large amount of data that requires more time to access; and you must decide which pieces of data to have close at hand.

Memory hierarchies have been a ubiquitous feature of computers since very early in their history. To begin with, data in the main memory of a processor can be accessed much more quickly than the data on its hard disk; but the disk has much more storage capacity. Thus, it is important to keep the most regularly used pieces of data in main memory, and go to disk as infrequently as possible. The same phenomenon, qualitatively, occurs with on-chip caches in modern processors. These can be accessed in a few cycles, and so data can be retrieved from cache much more quickly than it can be retrieved from main memory. This is another level of hierarchy: small caches have faster access time than main memory, which in turn is smaller and faster to access than disk. And one can see extensions of this hierarchy in many other settings. When one uses a Web browser, the disk often acts as a cache for frequently visited Web pages, since going to disk is still much faster than downloading something over the Internet.

Caching is a general term for the process of storing a small amount of data in a fast memory so as to reduce the amount of time spent interacting with a slow memory. In the previous examples, the on-chip cache reduces the need to fetch data from main memory, the main memory acts as a cache for the disk, and the disk acts as a cache for the Internet. (Much as your desk acts as a cache for the campus library, and the assorted facts you're able to remember without looking them up constitute a cache for the books on your desk.)

For caching to be as effective as possible, it should generally be the case that when you go to access a piece of data, it is already in the cache. To achieve this, a *cache maintenance* algorithm determines what to keep in the cache and what to evict from the cache when new data needs to be brought in.

Of course, as the caching problem arises in different settings, it involves various different considerations based on the underlying technology. For our purposes here, though, we take an abstract view of the problem that underlies most of these settings. We consider a set U of n pieces of data stored in *main memory*. We also have a faster memory, the *cache*, that can hold $k < n$ pieces of data at any one time. We will assume that the cache initially holds some set of k items. A sequence of data items $D = d_1, d_2, \ldots, d_m$ drawn from U is presented to us—this is the sequence of memory references we must process—and in processing them we must decide at all times which k items to keep in the cache. When item d_i is presented, we can access it very quickly if it is already in the cache; otherwise, we are required to bring it from main memory into the cache and, if the cache is full, to *evict* some other piece of data that is currently in the cache to make room for d_i. This is called a *cache miss*, and we want to have as few of these as possible.

Thus, on a particular sequence of memory references, a cache maintenance algorithm determines an *eviction schedule*—specifying which items should be evicted from the cache at which points in the sequence—and this determines the contents of the cache and the number of misses over time. Let's consider an example of this process.

- Suppose we have three items $\{a, b, c\}$, the cache size is $k = 2$, and we are presented with the sequence

  $$a, b, c, b, c, a, b.$$

  Suppose that the cache initially contains the items a and b. Then on the third item in the sequence, we could evict a so as to bring in c; and on the sixth item we could evict c so as to bring in a; we thereby incur two cache misses over the whole sequence. After thinking about it, one concludes that any eviction schedule for this sequence must include at least two cache misses.

Under real operating conditions, cache maintenance algorithms must process memory references $d_1, d_2, \ldots$ without knowledge of what's coming in the future; but for purposes of evaluating the quality of these algorithms, systems researchers very early on sought to understand the nature of the optimal solution to the caching problem. Given a full sequence S of memory references, what is the eviction schedule that incurs as few cache misses as possible?

Designing and Analyzing the Algorithm

In the 1960s, Les Belady showed that the following simple rule will always incur the minimum number of misses:

```
When $d_i$ needs to be brought into the cache,
  evict the item that is needed the farthest into the future
```

We will call this the *Farthest-in-Future Algorithm*. When it is time to evict something, we look at the next time that each item in the cache will be referenced, and choose the one for which this is as late as possible.

This is a very natural algorithm. At the same time, the fact that it is optimal on all sequences is somewhat more subtle than it first appears. Why evict the item that is needed farthest in the future, as opposed, for example, to the one that will be used least frequently in the future? Moreover, consider a sequence like

$$a, b, c, d, a, d, e, a, d, b, c$$

with $k = 3$ and items $\{a, b, c\}$ initially in the cache. The Farthest-in-Future rule will produce a schedule S that evicts c on the fourth step and b on the seventh step. But there are other eviction schedules that are just as good. Consider the schedule S' that evicts b on the fourth step and c on the seventh step, incurring the same number of misses. So in fact it's easy to find cases where schedules produced by rules other than Farthest-in-Future are also optimal; and given this flexibility, why might a deviation from Farthest-in-Future early on not yield an actual savings farther along in the sequence? For example, on the seventh step in our example, the schedule S' is actually evicting an item (c) that is needed *farther* into the future than the item evicted at this point by Farthest-in-Future, since Farthest-in-Future gave up c earlier on.

These are some of the kinds of things one should worry about before concluding that Farthest-in-Future really is optimal. In thinking about the example above, we quickly appreciate that it doesn't really matter whether b or c is evicted at the fourth step, since the other one should be evicted at the seventh step; so given a schedule where b is evicted first, we can swap the choices of b and c without changing the cost. This reasoning—swapping one decision for another—forms the first outline of an *exchange argument* that proves the optimality of Farthest-in-Future.

Before delving into this analysis, let's clear up one important issue. All the cache maintenance algorithms we've been considering so far produce schedules that only bring an item d into the cache in a step i if there is a request to d in step i, and d is not already in the cache. Let us call such a schedule *reduced*—it does the minimal amount of work necessary in a given step. But in general one could imagine an algorithm that produced schedules that are not reduced, by bringing in items in steps when they are not requested. We now show that for every nonreduced schedule, there is an equally good reduced schedule.

Let S be a schedule that may not be reduced. We define a new schedule $\overline{S}$—the *reduction* of S—as follows. In any step i where S brings in an item d that has not been requested, our construction of $\overline{S}$ "pretends" to do this but actually leaves d in main memory. It only really brings d into the cache in the next step j after this in which d is requested. In this way, the cache miss incurred by $\overline{S}$ in step j can be charged to the earlier cache operation performed by S in step i, when it brought in d. Hence we have the following fact.

(4.11) *$\overline{S}$ is a reduced schedule that brings in at most as many items as the schedule S.*

Note that for any reduced schedule, the number of items that are brought in is exactly the number of misses.

Proving the Optimalthy of Farthest-in-Future We now proceed with the exchange argument showing that Farthest-in-Future is optimal. Consider an arbitrary sequence D of memory references; let S_{FF} denote the schedule produced by Farthest-in-Future, and let S^* denote a schedule that incurs the minimum possible number of misses. We will now gradually "transform" the schedule S^* into the schedule S_{FF}, one eviction decision at a time, without increasing the number of misses.

Here is the basic fact we use to perform one step in the transformation.

(4.12) *Let S be a reduced schedule that makes the same eviction decisions as S_{FF} through the first j items in the sequence, for a number j. Then there is a reduced schedule S' that makes the same eviction decisions as S_{FF} through the first $j+1$ items, and incurs no more misses than S does.*

Proof. Consider the $(j+1)^{\text{st}}$ request, to item $d = d_{j+1}$. Since S and S_{FF} have agreed up to this point, they have the same cache contents. If d is in the cache for both, then no eviction decision is necessary (both schedules are reduced), and so S in fact agrees with S_{FF} through step $j+1$, and we can set $S' = S$. Similarly, if d needs to be brought into the cache, but S and S_{FF} both evict the same item to make room for d, then we can again set $S' = S$.

So the interesting case arises when d needs to be brought into the cache, and to do this S evicts item f while S_{FF} evicts item $e \neq f$. Here S and S_{FF} do not already agree through step $j+1$ since S has e in cache while S_{FF} has f in cache. Hence we must actually do something nontrivial to construct S'.

As a first step, we should have S' evict e rather than f. Now we need to further ensure that S' incurs no more misses than S. An easy way to do this would be to have S' agree with S for the remainder of the sequence; but this is no longer possible, since S and S' have slightly different caches from this point onward. So instead we'll have S' try to get its cache back to the same state as S as quickly as possible, while not incurring unnecessary misses. Once the caches are the same, we can finish the construction of S' by just having it behave like S.

Specifically, from request $j+2$ onward, S' behaves exactly like S until one of the following things happens for the first time.

(i) There is a request to an item $g \neq e, f$ that is not in the cache of S, and S evicts e to make room for it. Since S' and S only differ on e and f, it must be that g is not in the cache of S' either; so we can have S' evict f, and now the caches of S and S' are the same. We can then have S' behave exactly like S for the rest of the sequence.

(ii) There is a request to f, and S evicts an item e'. If $e' = e$, then we're all set: S' can simply access f from the cache, and after this step the caches of S and S' will be the same. If $e' \neq e$, then we have S' evict e' as well, and bring in e from main memory; this too results in S and S' having the same caches. However, we must be careful here, since S' is no longer a reduced schedule: it brought in e when it wasn't immediately needed. So to finish this part of the construction, we further transform S' to its reduction $\overline{S'}$ using (4.11); this doesn't increase the number of items brought in by S', and it still agrees with S_{FF} through step $j+1$.

Hence, in both these cases, we have a new reduced schedule S' that agrees with S_{FF} through the first $j+1$ items and incurs no more misses than S does. And crucially—here is where we use the defining property of the Farthest-in-Future Algorithm—one of these two cases will arise *before* there is a reference to e. This is because in step $j+1$, Farthest-in-Future evicted the item (e) that would be needed farthest in the future; so before there could be a request to e, there would have to be a request to f, and then case (ii) above would apply. ■

Using this result, it is easy to complete the proof of optimality. We begin with an optimal schedule S^*, and use (4.12) to construct a schedule S_1 that agrees with S_{FF} through the first step. We continue applying (4.12) inductively for $j = 1, 2, 3, \ldots, m$, producing schedules S_j that agree with S_{FF} through the first j steps. Each schedule incurs no more misses than the previous one; and by definition $S_m = S_{FF}$, since it agrees with it through the whole sequence. Thus we have

(4.13) *S_{FF} incurs no more misses than any other schedule S^* and hence is optimal.*

Extensions: Caching under Real Operating Conditions

As mentioned in the previous subsection, Belady's optimal algorithm provides a benchmark for caching performance; but in applications, one generally must make eviction decisions on the fly without knowledge of future requests. Experimentally, the best caching algorithms under this requirement seem to be variants of the *Least-Recently-Used* (LRU) Principle, which proposes evicting the item from the cache that was referenced *longest ago*.

If one thinks about it, this is just Belady's Algorithm with the direction of time reversed—longest in the past rather than farthest in the future. It is effective because applications generally exhibit *locality of reference*: a running program will generally keep accessing the things it has just been accessing. (It is easy to invent pathological exceptions to this principle, but these are relatively rare in practice.) Thus one wants to keep the more recently referenced items in the cache.

Long after the adoption of LRU in practice, Sleator and Tarjan showed that one could actually provide some theoretical analysis of the performance of LRU, bounding the number of misses it incurs relative to Farthest-in-Future. We will discuss this analysis, as well as the analysis of a randomized variant on LRU, when we return to the caching problem in Chapter 13.

4.4 Shortest Paths in a Graph

Some of the basic algorithms for graphs are based on greedy design principles. Here we apply a greedy algorithm to the problem of finding shortest paths, and in the next section we look at the construction of minimum-cost spanning trees.

The Problem

As we've seen, graphs are often used to model networks in which one travels from one point to another—traversing a sequence of highways through interchanges, or traversing a sequence of communication links through intermediate routers. As a result, a basic algorithmic problem is to determine the shortest path between nodes in a graph. We may ask this as a point-to-point question: Given nodes u and v, what is the shortest u-v path? Or we may ask for more information: Given a *start node* s, what is the shortest path from s to each other node?

The concrete setup of the shortest paths problem is as follows. We are given a directed graph $G = (V, E)$, with a designated start node s. We assume that s has a path to every other node in G. Each edge e has a length $\ell_e \geq 0$, indicating the time (or distance, or cost) it takes to traverse e. For a path P, the *length of* P—denoted $\ell(P)$—is the sum of the lengths of all edges in P. Our goal is to determine the shortest path from s to every other node in the graph. We should mention that although the problem is specified for a directed graph, we can handle the case of an undirected graph by simply replacing each undirected edge $e = (u, v)$ of length ℓ_e by two directed edges (u, v) and (v, u), each of length ℓ_e.

Designing the Algorithm

In 1959, Edsger Dijkstra proposed a very simple greedy algorithm to solve the single-source shortest-paths problem. We begin by describing an algorithm that just determines the *length* of the shortest path from s to each other node in the graph; it is then easy to produce the paths as well. The algorithm maintains a set S of vertices u for which we have determined a shortest-path distance $d(u)$ from s; this is the "explored" part of the graph. Initially $S = \{s\}$, and $d(s) = 0$. Now, for each node $v \in V - S$, we determine the shortest path that can be constructed by traveling along a path through the explored part S to some $u \in S$, followed by the single edge (u, v). That is, we consider the quantity

$d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$. We choose the node $v \in V - S$ for which this quantity is minimized, add v to S, and define $d(v)$ to be the value $d'(v)$.

```
Dijkstra's Algorithm ($G, \ell$)
Let $S$ be the set of explored nodes
    For each $u \in S$, we store a distance $d(u)$
Initially $S = \{s\}$ and $d(s) = 0$
While $S \neq V$
    Select a node $v \notin S$ with at least one edge from $S$ for which
       $d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$ is as small as possible
    Add $v$ to $S$ and define $d(v) = d'(v)$
EndWhile
```

It is simple to produce the s-u paths corresponding to the distances found by Dijkstra's Algorithm. As each node v is added to the set S, we simply record the edge (u, v) on which it achieved the value $\min_{e=(u,v):u \in S} d(u) + \ell_e$. The path P_v is implicitly represented by these edges: if (u, v) is the edge we have stored for v, then P_v is just (recursively) the path P_u followed by the single edge (u, v). In other words, to construct P_v, we simply start at v; follow the edge we have stored for v in the reverse direction to u; then follow the edge we have stored for u in the reverse direction to its predecessor; and so on until we reach s. Note that s must be reached, since our backward walk from v visits nodes that were added to S earlier and earlier.

To get a better sense of what the algorithm is doing, consider the snapshot of its execution depicted in Figure 4.7. At the point the picture is drawn, two iterations have been performed: the first added node u, and the second added node v. In the iteration that is about to be performed, the node x will be added because it achieves the smallest value of $d'(x)$; thanks to the edge (u, x), we have $d'(x) = d(u) + l_{ux} = 2$. Note that attempting to add y or z to the set S at this point would lead to an incorrect value for their shortest-path distances; ultimately, they will be added because of their edges from x.

Analyzing the Algorithm

We see in this example that Dijkstra's Algorithm is doing the right thing and avoiding recurring pitfalls: growing the set S by the wrong node can lead to an overestimate of the shortest-path distance to that node. The question becomes: Is it always true that when Dijkstra's Algorithm adds a node v, we get the true shortest-path distance to v?

We now answer this by proving the correctness of the algorithm, showing that the paths P_u really are shortest paths. Dijkstra's Algorithm is greedy in

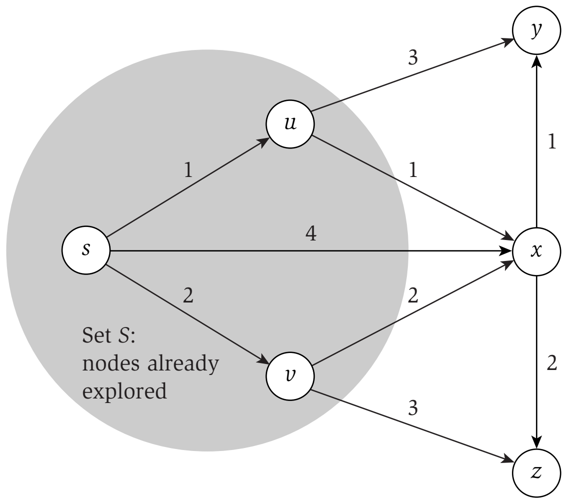

Figure 4.7 A snapshot of the execution of Dijkstra's Algorithm. The next node that will be added to the set S is x, due to the path through u.

the sense that we always form the shortest new s-v path we can make from a path in S followed by a single edge. We prove its correctness using a variant of our first style of analysis: we show that it "stays ahead" of all other solutions by establishing, inductively, that each time it selects a path to a node v, that path is shorter than every other possible path to v.

(4.14) *Consider the set S at any point in the algorithm's execution. For each $u \in S$, the path P_u is a shortest s-u path.*

Note that this fact immediately establishes the correctness of Dijkstra's Algorithm, since we can apply it when the algorithm terminates, at which point S includes all nodes.

Proof. We prove this by induction on the size of S. The case $|S| = 1$ is easy, since then we have $S = \{s\}$ and $d(s) = 0$. Suppose the claim holds when $|S| = k$ for some value of $k \geq 1$; we now grow S to size $k + 1$ by adding the node v. Let (u, v) be the final edge on our s-v path P_v.

By induction hypothesis, P_u is the shortest s-u path for each $u \in S$. Now consider any other s-v path P; we wish to show that it is at least as long as P_v. In order to reach v, this path P must leave the set S *somewhere*; let y be the first node on P that is not in S, and let $x \in S$ be the node just before y.

The situation is now as depicted in Figure 4.8, and the crux of the proof is very simple: P cannot be shorter than P_v because it is already at least as

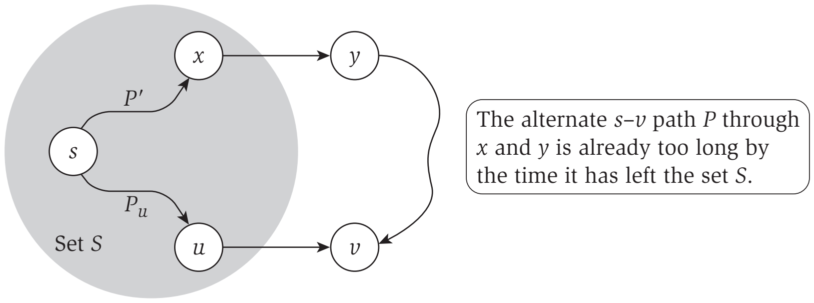
x
y
P'
s
P_u
Set S
u
v
The alternate s–v path P through x and y is already too long by the time it has left the set S.

Figure 4.8 The shortest path P_v and an alternate s-v path P through the node y.

long as P_v by the time it has left the set S. Indeed, in iteration $k+1$, Dijkstra's Algorithm must have considered adding node y to the set S via the edge (x, y) and rejected this option in favor of adding v. This means that there is no path from s to y through x that is shorter than P_v. But the subpath of P up to y is such a path, and so this subpath is at least as long as P_v. Since edge lengths are nonnegative, the full path P is at least as long as P_v as well.

This is a complete proof; one can also spell out the argument in the previous paragraph using the following inequalities. Let P' be the subpath of P from s to x. Since $x \in S$, we know by the induction hypothesis that P_x is a shortest s-x path (of length $d(x)$), and so $\ell(P') \geq \ell(P_x) = d(x)$. Thus the subpath of P out to node y has length $\ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq d'(y)$, and the full path P is at least as long as this subpath. Finally, since Dijkstra's Algorithm selected v in this iteration, we know that $d'(y) \geq d'(v) = \ell(P_v)$. Combining these inequalities shows that $\ell(P) \geq \ell(P') + \ell(x, y) \geq \ell(P_v)$. ∎

Here are two observations about Dijkstra's Algorithm and its analysis. First, the algorithm does not always find shortest paths if some of the edges can have negative lengths. (Do you see where the proof breaks?) Many shortest-path applications involve negative edge lengths, and a more complex algorithm—due to Bellman and Ford—is required for this case. We will see this algorithm when we consider the topic of dynamic programming.

The second observation is that Dijkstra's Algorithm is, in a sense, even simpler than we've described here. Dijkstra's Algorithm is really a "continuous" version of the standard breadth-first search algorithm for traversing a graph, and it can be motivated by the following physical intuition. Suppose the edges of G formed a system of pipes filled with water, joined together at the nodes; each edge e has length ℓ_e and a fixed cross-sectional area. Now suppose an extra droplet of water falls at node s and starts a wave from s. As the wave expands out of node s at a constant speed, the expanding sphere of wavefront reaches nodes in increasing order of their distance from s. It is easy to believe (and also true) that the path taken by the wavefront to get to any node v is a shortest path. Indeed, it is easy to see that this is exactly the path to v found by Dijkstra's Algorithm, and that the nodes are discovered by the expanding water in the same order that they are discovered by Dijkstra's Algorithm.

Implementation and Running Time To conclude our discussion of Dijkstra's Algorithm, we consider its running time. There are $n-1$ iterations of the `While` loop for a graph with n nodes, as each iteration adds a new node v to S. Selecting the correct node v efficiently is a more subtle issue. One's first impression is that each iteration would have to consider each node $v \notin S$, and go through all the edges between S and v to determine the minimum $\min_{e=(u,v):u\in S} d(u) + \ell_e$, so that we can select the node v for which this minimum is smallest. For a graph with m edges, computing all these minima can take $O(m)$ time, so this would lead to an implementation that runs in $O(mn)$ time.

We can do considerably better if we use the right data structures. First, we will explicitly maintain the values of the minima $d'(v) = \min_{e=(u,v):u\in S} d(u) + \ell_e$ for each node $v \in V - S$, rather than recomputing them in each iteration. We can further improve the efficiency by keeping the nodes $V - S$ in a priority queue with $d'(v)$ as their keys. Priority queues were discussed in Chapter 2; they are data structures designed to maintain a set of n elements, each with a key. A priority queue can efficiently insert elements, delete elements, change an element's key, and extract the element with the minimum key. We will need the third and fourth of the above operations: `ChangeKey` and `ExtractMin`.

How do we implement Dijkstra's Algorithm using a priority queue? We put the nodes V in a priority queue with $d'(v)$ as the key for $v \in V$. To select the node v that should be added to the set S, we need the `ExtractMin` operation. To see how to update the keys, consider an iteration in which node v is added to S, and let $w \notin S$ be a node that remains in the priority queue. What do we have to do to update the value of $d'(w)$? If (v, w) is not an edge, then we don't have to do anything: the set of edges considered in the minimum $\min_{e=(u,w):u\in S} d(u) + \ell_e$ is exactly the same before and after adding v to S. If $e' = (v, w) \in E$, on the other hand, then the new value for the key is $\min(d'(w), d(v) + \ell_{e'})$. If $d'(w) > d(v) + \ell_{e'}$ then we need to use the `ChangeKey` operation to decrease the key of node w appropriately. This `ChangeKey` operation can occur at most once per edge, when the tail of the edge e' is added to S. In summary, we have the following result.

(4.15) *Using a priority queue, Dijkstra's Algorithm can be implemented on a graph with n nodes and m edges to run in $O(m)$ time, plus the time for n* `ExtractMin` *and m* `ChangeKey` *operations.*

Using the heap-based priority queue implementation discussed in Chapter 2, each priority queue operation can be made to run in $O(\log n)$ time. Thus the overall time for the implementation is $O(m \log n)$.

4.5 The Minimum Spanning Tree Problem

We now apply an exchange argument in the context of a second fundamental problem on graphs: the Minimum Spanning Tree Problem.

The Problem

Suppose we have a set of locations $V = \{v_1, v_2, \ldots, v_n\}$, and we want to build a communication network on top of them. The network should be connected—there should be a path between every pair of nodes—but subject to this requirement, we wish to build it as cheaply as possible.

For certain pairs (v_i, v_j), we may build a direct link between v_i and v_j for a certain cost $c(v_i, v_j) > 0$. Thus we can represent the set of possible links that may be built using a graph $G = (V, E)$, with a positive *cost* c_e associated with each edge $e = (v_i, v_j)$. The problem is to find a subset of the edges $T \subseteq E$ so that the graph (V, T) is connected, and the total cost $\sum_{e \in T} c_e$ is as small as possible. (We will assume that the full graph G is connected; otherwise, no solution is possible.)

Here is a basic observation.

(4.16) *Let T be a minimum-cost solution to the network design problem defined above. Then (V, T) is a tree.*

Proof. By definition, (V, T) must be connected; we show that it also will contain no cycles. Indeed, suppose it contained a cycle C, and let e be any edge on C. We claim that $(V, T - \{e\})$ is still connected, since any path that previously used the edge e can now go "the long way" around the remainder of the cycle C instead. It follows that $(V, T - \{e\})$ is also a valid solution to the problem, and it is cheaper—a contradiction. ∎

If we allow some edges to have 0 cost (that is, we assume only that the costs c_e are nonnegative), then a minimum-cost solution to the network design problem may have extra edges—edges that have 0 cost and could optionally be deleted. But even in this case, there is always a minimum-cost solution that is a tree. Starting from any optimal solution, we could keep deleting edges on cycles until we had a tree; with nonnegative edges, the cost would not increase during this process.

We will call a subset $T \subseteq E$ a *spanning tree* of G if (V, T) is a tree. Statement (4.16) says that the goal of our network design problem can be rephrased as that of finding the cheapest spanning tree of the graph; for this reason, it is generally called the *Minimum Spanning Tree Problem*. Unless G is a very simple graph, it will have exponentially many different spanning trees, whose structures may look very different from one another. So it is not at all clear how to efficiently find the cheapest tree from among all these options.

Designing Algorithms

As with the previous problems we've seen, it is easy to come up with a number of natural greedy algorithms for the problem. But curiously, and fortunately, this is a case where *many* of the first greedy algorithms one tries turn out to be correct: they each solve the problem optimally. We will review a few of these algorithms now and then discover, via a nice pair of exchange arguments, some of the underlying reasons for this plethora of simple, optimal algorithms.

Here are three greedy algorithms, each of which correctly finds a minimum spanning tree.

- One simple algorithm starts without any edges at all and builds a spanning tree by successively inserting edges from E in order of increasing cost. As we move through the edges in this order, we insert each edge e as long as it does not create a cycle when added to the edges we've already inserted. If, on the other hand, inserting e would result in a cycle, then we simply discard e and continue. This approach is called *Kruskal's Algorithm*.
- Another simple greedy algorithm can be designed by analogy with Dijkstra's Algorithm for paths, although, in fact, it is even simpler to specify than Dijkstra's Algorithm. We start with a root node s and try to greedily grow a tree from s outward. At each step, we simply add the node that can be attached as cheaply as possibly to the partial tree we already have.

  More concretely, we maintain a set $S \subseteq V$ on which a spanning tree has been constructed so far. Initially, $S = \{s\}$. In each iteration, we grow S by one node, adding the node v that minimizes the "attachment cost" $\min_{e=(u,v):u \in S} c_e$, and including the edge $e = (u, v)$ that achieves this minimum in the spanning tree. This approach is called *Prim's Algorithm.*
- Finally, we can design a greedy algorithm by running sort of a "backward" version of Kruskal's Algorithm. Specifically, we start with the full graph (V, E) and begin deleting edges in order of decreasing cost. As we get to each edge e (starting from the most expensive), we delete it as

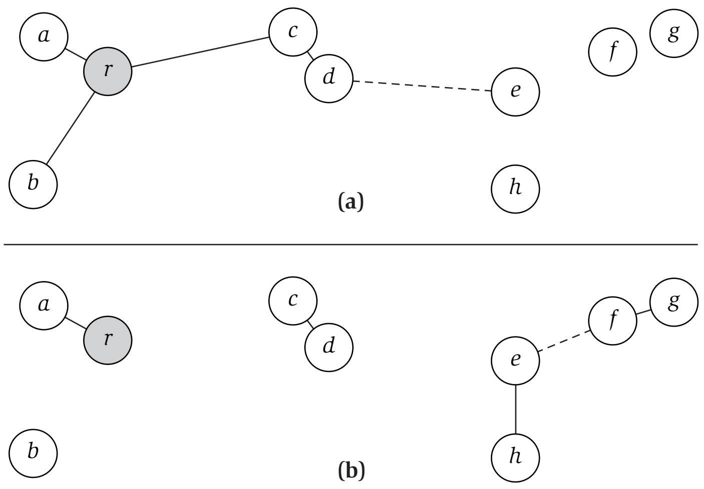

Figure 4.9 Sample run of the Minimum Spanning Tree Algorithms of (a) Prim and (b) Kruskal, on the same input. The first 4 edges added to the spanning tree are indicated by solid lines; the next edge to be added is a dashed line.

long as doing so would not actually disconnect the graph we currently have. For want of a better name, this approach is generally called the *Reverse-Delete Algorithm* (as far as we can tell, it's never been named after a specific person).

For example, Figure 4.9 shows the first four edges added by Prim's and Kruskal's Algorithms respectively, on a geometric instance of the Minimum Spanning Tree Problem in which the cost of each edge is proportional to the geometric distance in the plane.

The fact that each of these algorithms is guaranteed to produce an optimal solution suggests a certain "robustness" to the Minimum Spanning Tree Problem—there are many ways to get to the answer. Next we explore some of the underlying reasons why so many different algorithms produce minimum-cost spanning trees.

Analyzing the Algorithms

All these algorithms work by repeatedly inserting or deleting edges from a partial solution. So, to analyze them, it would be useful to have in hand some basic facts saying when it is "safe" to include an edge in the minimum spanning tree, and, correspondingly, when it is safe to eliminate an edge on the grounds that it couldn't possibly be in the minimum spanning tree. For purposes of the analysis, we will make the simplifying assumption that all edge costs are distinct from one another (i.e., no two are equal). This assumption makes it easier to express the arguments that follow, and we will show later in this section how this assumption can be easily eliminated.

When Is It Safe to Include an Edge in the Minimum Spanning Tree? The crucial fact about edge insertion is the following statement, which we will refer to as the *Cut Property*.

(4.17) *Assume that all edge costs are distinct. Let S be any subset of nodes that is neither empty nor equal to all of V, and let edge $e = (v, w)$ be the minimum-cost edge with one end in S and the other in $V - S$. Then every minimum spanning tree contains the edge e.*

Proof. Let T be a spanning tree that does not contain e; we need to show that T does not have the minimum possible cost. We'll do this using an exchange argument: we'll identify an edge e' in T that is more expensive than e, and with the property exchanging e for e' results in another spanning tree. This resulting spanning tree will then be cheaper than T, as desired.

The crux is therefore to find an edge that can be successfully exchanged with e. Recall that the ends of e are v and w. T is a spanning tree, so there must be a path P in T from v to w. Starting at v, suppose we follow the nodes of P in sequence; there is a first node w' on P that is in $V - S$. Let $v' \in S$ be the node just before w' on P, and let $e' = (v', w')$ be the edge joining them. Thus, e' is an edge of T with one end in S and the other in $V - S$. See Figure 4.10 for the situation at this stage in the proof.

If we exchange e for e', we get a set of edges $T' = T - \{e'\} \cup \{e\}$. We claim that T' is a spanning tree. Clearly (V, T') is connected, since (V, T) is connected, and any path in (V, T) that used the edge $e' = (v', w')$ can now be "rerouted" in (V, T') to follow the portion of P from v' to v, then the edge e, and then the portion of P from w to w'. To see that (V, T') is also acyclic, note that the only cycle in $(V, T' \cup \{e'\})$ is the one composed of e and the path P, and this cycle is not present in (V, T') due to the deletion of e'.

We noted above that the edge e' has one end in S and the other in $V - S$. But e is the cheapest edge with this property, and so $c_e < c_{e'}$. (The inequality is strict since no two edges have the same cost.) Thus the total cost of T' is less than that of T, as desired. ∎

The proof of (4.17) is a bit more subtle than it may first appear. To appreciate this subtlety, consider the following shorter but incorrect argument for (4.17). Let T be a spanning tree that does not contain e. Since T is a spanning tree, it must contain an edge f with one end in S and the other in $V - S$. Since e is the cheapest edge with this property, we have $c_e < c_f$, and hence $T - \{f\} \cup \{e\}$ is a spanning tree that is cheaper than T.

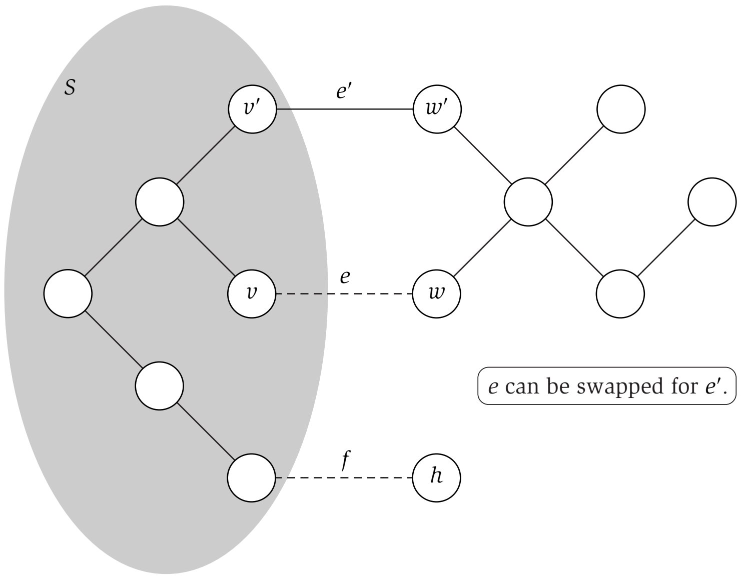

Figure 4.10 Swapping the edge e for the edge e' in the spanning tree T, as described in the proof of (4.17).

The problem with this argument is not in the claim that f exists, or that $T - \{f\} \cup \{e\}$ is cheaper than T. The difficulty is that $T - \{f\} \cup \{e\}$ may not be a spanning tree, as shown by the example of the edge f in Figure 4.10. The point is that we can't prove (4.17) by simply picking *any* edge in T that crosses from S to $V - S$; some care must be taken to find the right one.

The Optimality of Kruskal's and Prim's Algorithms We can now easily prove the optimality of both Kruskal's Algorithm and Prim's Algorithm. The point is that both algorithms only include an edge when it is justified by the Cut Property (4.17).

(4.18) *Kruskal's Algorithm produces a minimum spanning tree of G.*

Proof. Consider any edge $e = (v, w)$ added by Kruskal's Algorithm, and let S be the set of all nodes to which v has a path at the moment just before e is added. Clearly $v \in S$, but $w \notin S$, since adding e does not create a cycle. Moreover, no edge from S to $V - S$ has been encountered yet, since any such edge could have been added without creating a cycle, and hence would have been added by Kruskal's Algorithm. Thus e is the cheapest edge with one end in S and the other in $V - S$, and so by (4.17) it belongs to every minimum spanning tree.

So if we can show that the output (V, T) of Kruskal's Algorithm is in fact a spanning tree of G, then we will be done. Clearly (V, T) contains no cycles, since the algorithm is explicitly designed to avoid creating cycles. Further, if (V, T) were not connected, then there would exist a nonempty subset of nodes S (not equal to all of V) such that there is no edge from S to $V - S$. But this contradicts the behavior of the algorithm: we know that since G is connected, there is at least one edge between S and $V - S$, and the algorithm will add the first of these that it encounters. ■

(4.19) *Prim's Algorithm produces a minimum spanning tree of G.*

Proof. For Prim's Algorithm, it is also very easy to show that it only adds edges belonging to every minimum spanning tree. Indeed, in each iteration of the algorithm, there is a set $S \subseteq V$ on which a partial spanning tree has been constructed, and a node v and edge e are added that minimize the quantity $\min_{e=(u,v):u \in S} c_e$. By definition, e is the cheapest edge with one end in S and the other end in $V - S$, and so by the Cut Property (4.17) it is in every minimum spanning tree.

It is also straightforward to show that Prim's Algorithm produces a spanning tree of G, and hence it produces a minimum spanning tree. ■

When Can We Guarantee an Edge Is Not in the Minimum Spanning Tree? The crucial fact about edge deletion is the following statement, which we will refer to as the *Cycle Property*.

(4.20) *Assume that all edge costs are distinct. Let C be any cycle in G, and let edge $e = (v, w)$ be the most expensive edge belonging to C. Then e does not belong to any minimum spanning tree of G.*

Proof. Let T be a spanning tree that contains e; we need to show that T does not have the minimum possible cost. By analogy with the proof of the Cut Property (4.17), we'll do this with an exchange argument, swapping e for a cheaper edge in such a way that we still have a spanning tree.

So again the question is: How do we find a cheaper edge that can be exchanged in this way with e? Let's begin by deleting e from T; this partitions the nodes into two components: S, containing node v; and $V - S$, containing node w. Now, the edge we use in place of e should have one end in S and the other in $V - S$, so as to stitch the tree back together.

We can find such an edge by following the cycle C. The edges of C other than e form, by definition, a path P with one end at v and the other at w. If we follow P from v to w, we begin in S and end up in $V - S$, so there is some

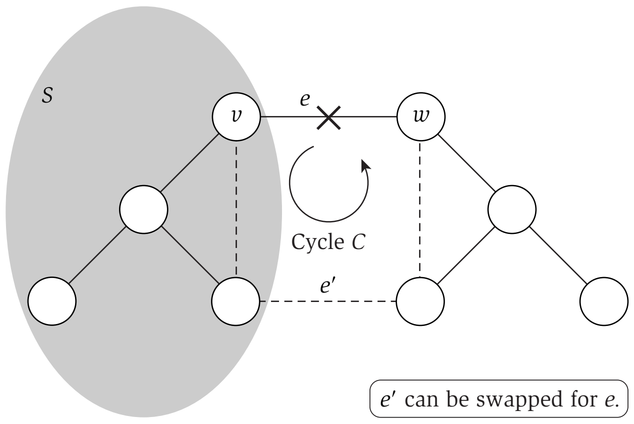

Figure 4.11 Swapping the edge e' for the edge e in the spanning tree T, as described in the proof of (4.20).

edge e' on P that crosses from S to $V - S$. See Figure 4.11 for an illustration of this.

Now consider the set of edges $T' = T - \{e\} \cup \{e'\}$. Arguing just as in the proof of the Cut Property (4.17), the graph (V, T') is connected and has no cycles, so T' is a spanning tree of G. Moreover, since e is the most expensive edge on the cycle C, and e' belongs to C, it must be that e' is cheaper than e, and hence T' is cheaper than T, as desired. ■

The Optimality of the Reverse-Delete Algorithm Now that we have the Cycle Property (4.20), it is easy to prove that the Reverse-Delete Algorithm produces a minimum spanning tree. The basic idea is analogous to the optimality proofs for the previous two algorithms: Reverse-Delete only adds an edge when it is justified by (4.20).

(4.21) *The Reverse-Delete Algorithm produces a minimum spanning tree of G.*

Proof. Consider any edge $e = (v, w)$ removed by Reverse-Delete. At the time that e is removed, it lies on a cycle C; and since it is the first edge encountered by the algorithm in decreasing order of edge costs, it must be the most expensive edge on C. Thus by (4.20), e does not belong to any minimum spanning tree.

So if we show that the output (V, T) of Reverse-Delete is a spanning tree of G, we will be done. Clearly (V, T) is connected, since the algorithm never removes an edge when this will disconnect the graph. Now, suppose by way of contradiction that (V, T) contains a cycle C. Consider the most expensive edge e on C, which would be the first one encountered by the algorithm. This edge should have been removed, since its removal would not have disconnected the graph, and this contradicts the behavior of Reverse-Delete. ■

While we will not explore this further here, the combination of the Cut Property (4.17) and the Cycle Property (4.20) implies that something even more general is going on. *Any* algorithm that builds a spanning tree by repeatedly including edges when justified by the Cut Property and deleting edges when justified by the Cycle Property—in any order at all—will end up with a minimum spanning tree. This principle allows one to design natural greedy algorithms for this problem beyond the three we have considered here, and it provides an explanation for why so many greedy algorithms produce optimal solutions for this problem.

Eliminating the Assumption that All Edge Costs Are Distinct Thus far, we have assumed that all edge costs are distinct, and this assumption has made the analysis cleaner in a number of places. Now, suppose we are given an instance of the Minimum Spanning Tree Problem in which certain edges have the same cost – how can we conclude that the algorithms we have been discussing still provide optimal solutions?

There turns out to be an easy way to do this: we simply take the instance and perturb all edge costs by different, extremely small numbers, so that they all become distinct. Now, any two costs that differed originally will still have the same relative order, since the perturbations are so small; and since all of our algorithms are based on just comparing edge costs, the perturbations effectively serve simply as "tie-breakers" to resolve comparisons among costs that used to be equal.

Moreover, we claim that any minimum spanning tree T for the new, perturbed instance must have also been a minimum spanning tree for the original instance. To see this, we note that if T cost more than some tree T^* in the original instance, then for small enough perturbations, the change in the cost of T cannot be enough to make it better than T^* under the new costs. Thus, if we run any of our minimum spanning tree algorithms, using the perturbed costs for comparing edges, we will produce a minimum spanning tree T that is also optimal for the original instance.

Implementing Prim's Algorithm

We next discuss how to implement the algorithms we have been considering so as to obtain good running-time bounds. We will see that both Prim's and Kruskal's Algorithms can be implemented, with the right choice of data structures, to run in $O(m \log n)$ time. We will see how to do this for Prim's Algorithm here, and defer discussing the implementation of Kruskal's Algorithm to the next section. Obtaining a running time close to this for the Reverse-Delete Algorithm is difficult, so we do not focus on Reverse-Delete in this discussion.

For Prim's Algorithm, while the proof of correctness was quite different from the proof for Dijkstra's Algorithm for the Shortest-Path Algorithm, the implementations of Prim and Dijkstra are almost identical. By analogy with Dijkstra's Algorithm, we need to be able to decide which node v to add next to the growing set S, by maintaining the attachment costs $a(v) = \min_{e=(u,v):u \in S} c_e$ for each node $v \in V - S$. As before, we keep the nodes in a priority queue with these attachment costs $a(v)$ as the keys; we select a node with an `ExtractMin` operation, and update the attachment costs using `ChangeKey` operations. There are $n - 1$ iterations in which we perform `ExtractMin`, and we perform `ChangeKey` at most once for each edge. Thus we have

(4.22) *Using a priority queue, Prim's Algorithm can be implemented on a graph with n nodes and m edges to run in $O(m)$ time, plus the time for n* `ExtractMin`*, and m* `ChangeKey` *operations.*

As with Dijkstra's Algorithm, if we use a heap-based priority queue we can implement both `ExtractMin` and `ChangeKey` in $O(\log n)$ time, and so get an overall running time of $O(m \log n)$.

Extensions

The minimum spanning tree problem emerged as a particular formulation of a broader *network design* goal—finding a good way to connect a set of sites by installing edges between them. A minimum spanning tree optimizes a particular goal, achieving connectedness with minimum total edge cost. But there are a range of further goals one might consider as well.

We may, for example, be concerned about point-to-point distances in the spanning tree we build, and be willing to reduce these even if we pay more for the set of edges. This raises new issues, since it is not hard to construct examples where the minimum spanning tree does not minimize point-to-point distances, suggesting some tension between these goals.

Alternately, we may care more about the *congestion* on the edges. Given traffic that needs to be routed between pairs of nodes, one could seek a spanning tree in which no single edge carries more than a certain amount of this traffic. Here too, it is easy to find cases in which the minimum spanning tree ends up concentrating a lot of traffic on a single edge.

More generally, it is reasonable to ask whether a spanning tree is even the right kind of solution to our network design problem. A tree has the property that destroying any one edge disconnects it, which means that trees are not at all robust against failures. One could instead make resilience an explicit goal, for example seeking the cheapest connected network on the set of sites that remains connected after the deletion of any one edge.

All of these extensions lead to problems that are computationally much harder than the basic Minimum Spanning Tree problem, though due to their importance in practice there has been research on good heuristics for them.

4.6 Implementing Kruskal's Algorithm: The Union-Find Data Structure

One of the most basic graph problems is to find the set of connected components. In Chapter 3 we discussed linear-time algorithms using BFS or DFS for finding the connected components of a graph.

In this section, we consider the scenario in which a graph evolves through the addition of edges. That is, the graph has a fixed population of nodes, but it grows over time by having edges appear between certain pairs of nodes. Our goal is to maintain the set of connected components of such a graph throughout this evolution process. When an edge is added to the graph, we don't want to have to recompute the connected components from scratch. Rather, we will develop a data structure that we call the `Union-Find` structure, which will store a representation of the components in a way that supports rapid searching and updating.

This is exactly the data structure needed to implement Kruskal's Algorithm efficiently. As each edge $e = (v, w)$ is considered, we need to efficiently find the identities of the connected components containing v and w. If these components are different, then there is no path from v and w, and hence edge e should be included; but if the components are the same, then there is a v-w path on the edges already included, and so e should be omitted. In the event that e is included, the data structure should also support the efficient merging of the components of v and w into a single new component.

The Problem

The `Union-Find` data structure allows us to maintain disjoint sets (such as the components of a graph) in the following sense. Given a node u, the operation `Find`(u) will return the name of the set containing u. This operation can be used to test if two nodes u and v are in the same set, by simply checking if `Find`(u) = `Find`(v). The data structure will also implement an operation `Union`(A, B) to take two sets A and B and merge them to a single set.

These operations can be used to maintain connected components of an evolving graph $G = (V, E)$ as edges are added. The sets will be the connected components of the graph. For a node u, the operation `Find`(u) will return the name of the component containing u. If we add an edge (u, v) to the graph, then we first test if u and v are already in the same connected component (by testing if `Find`(u) = `Find`(v)). If they are not, then `Union`(`Find`(u),`Find`(v)) can be used to merge the two components into one. It is important to note that the `Union-Find` data structure can only be used to maintain components of a graph as we *add* edges; it is not designed to handle the effects of edge deletion, which may result in a single component being "split" into two.

To summarize, the `Union-Find` data structure will support three operations.

- `MakeUnionFind`(S) for a set S will return a `Union-Find` data structure on set S where all elements are in separate sets. This corresponds, for example, to the connected components of a graph with no edges. Our goal will be to implement `MakeUnionFind` in time $O(n)$ where $n = |S|$.
- For an element $u \in S$, the operation `Find`(u) will return the name of the set containing u. Our goal will be to implement `Find`(u) in $O(\log n)$ time. Some implementations that we discuss will in fact take only $O(1)$ time for this operation.
- For two sets A and B, the operation `Union`(A, B) will change the data structure by merging the sets A and B into a single set. Our goal will be to implement `Union` in $O(\log n)$ time.

Let's briefly discuss what we mean by the *name* of a set—for example, as returned by the `Find` operation. There is a fair amount of flexibility in defining the names of the sets; they should simply be consistent in the sense that `Find`(v) and `Find`(w) should return the same name if v and w belong to the same set, and different names otherwise. In our implementations, we will name each set using one of the elements it contains.

A Simple Data Structure for Union-Find

Maybe the simplest possible way to implement a `Union-Find` data structure is to maintain an array `Component` that contains the name of the set currently containing each element. Let S be a set, and assume it has n elements denoted $\{1, \ldots, n\}$. We will set up an array `Component` of size n, where `Component`$[s]$ is the name of the set containing s. To implement `MakeUnionFind`(S), we set up the array and initialize it to `Component`$[s] = s$ for all $s \in S$. This implementation makes `Find`(v) easy: it is a simple lookup and takes only $O(1)$ time. However, `Union`(A, B) for two sets A and B can take as long as $O(n)$ time, as we have to update the values of `Component`$[s]$ for all elements in sets A and B.

To improve this bound, we will do a few simple optimizations. First, it is useful to explicitly maintain the list of elements in each set, so we don't have to look through the whole array to find the elements that need updating. Further, we save some time by choosing the name for the union to be the name of one of the sets, say, set A: this way we only have to update the values `Component`[s] for $s \in B$, but not for any $s \in A$. Of course, if set B is large, this idea by itself doesn't help very much. Thus we add one further optimization. When set B is big, we may want to keep its name and change `Component`[s] for all $s \in A$ instead. More generally, we can maintain an additional array `size` of length n, where `size`[A] is the size of set A, and when a `Union`(A, B) operation is performed, we use the name of the larger set for the union. This way, fewer elements need to have their `Component` values updated.

Even with these optimizations, the worst case for a `Union` operation is still $O(n)$ time; this happens if we take the union of two large sets A and B, each containing a constant fraction of all the elements. However, such bad cases for `Union` cannot happen very often, as the resulting set $A \cup B$ is even bigger. How can we make this statement more precise? Instead of bounding the worst-case running time of a single `Union` operation, we can bound the total (or average) running time of a sequence of k `Union` operations.

(4.23) *Consider the array implementation of the* `Union-Find` *data structure for some set S of size n, where unions keep the name of the larger set. The* `Find` *operation takes $O(1)$ time,* `MakeUnionFind`(S) *takes $O(n)$ time, and any sequence of k* `Union` *operations takes at most $O(k \log k)$ time.*

Proof. The claims about the `MakeUnionFind` and `Find` operations are easy to verify. Now consider a sequence of k `Union` operations. The only part of a `Union` operation that takes more than $O(1)$ time is updating the array `Component`. Instead of bounding the time spent on one `Union` operation, we will bound the total time spent updating `Component`[v] for an element v throughout the sequence of k operations.

Recall that we start the data structure from a state when all n elements are in their own separate sets. A single `Union` operation can consider at most two of these original one-element sets, so after any sequence of k `Union` operations, all but at most $2k$ elements of S have been completely untouched. Now consider a particular element v. As v's set is involved in a sequence of `Union` operations, its size grows. It may be that in some of these `Union`s, the value of `Component`[v] is updated, and in others it is not. But our convention is that the union uses the name of the larger set, so in every update to `Component`[v] the size of the set containing v at least doubles. The size of v's set starts out at 1, and the maximum possible size it can reach is $2k$ (since we argued above that all but at most $2k$ elements are untouched by `Union` operations). Thus `Component`[v] gets updated at most $\log_2(2k)$ times throughout the process. Moreover, at most $2k$ elements are involved in any `Union` operations at all, so we get a bound of $O(k \log k)$ for the time spent updating `Component` values in a sequence of k `Union` operations. ■

While this bound on the average running time for a sequence of k operations is good enough in many applications, including implementing Kruskal's Algorithm, we will try to do better and reduce the *worst-case* time required. We'll do this at the expense of raising the time required for the `Find` operation to $O(\log n)$.

A Better Data Structure for Union-Find

The data structure for this alternate implementation uses pointers. Each node $v \in S$ will be contained in a record with an associated pointer to the name of the set that contains v. As before, we will use the elements of the set S as possible set names, naming each set after one of its elements. For the `MakeUnionFind`(S) operation, we initialize a record for each element $v \in S$ with a pointer that points to itself (or is defined as a `null` pointer), to indicate that v is in its own set.

Consider a `Union` operation for two sets A and B, and assume that the name we used for set A is a node $v \in A$, while set B is named after node $u \in B$. The idea is to have either u or v be the name of the combined set; assume we select v as the name. To indicate that we took the union of the two sets, and that the name of the union set is v, we simply update u's pointer to point to v. We do not update the pointers at the other nodes of set B.

As a result, for elements $w \in B$ other than u, the name of the set they belong to must be computed by following a sequence of pointers, first leading them to the "old name" u and then via the pointer from u to the "new name" v. See Figure 4.12 for what such a representation looks like. For example, the two sets in Figure 4.12 could be the outcome of the following sequence of `Union` operations: `Union`(w, u), `Union`(s, u), `Union`(t, v), `Union`(z, v), `Union`(i, x), `Union`(y, j), `Union`(x, j), and `Union`(u, v).

This pointer-based data structure implements `Union` in $O(1)$ time: all we have to do is to update one pointer. But a `Find` operation is no longer constant time, as we have to follow a sequence of pointers through a history of old names the set had, in order to get to the current name. How long can a `Find`(u) operation take? The number of steps needed is exactly the number of times the set containing node u had to change its name, that is, the number of times the `Component`$[u]$ array position would have been updated in our previous array representation. This can be as large as $O(n)$ if we are not careful with choosing set names. To reduce the time required for a `Find` operation, we will use the same optimization we used before: keep the name of the larger set as the name of the union. The sequence of `Union`s that produced the data

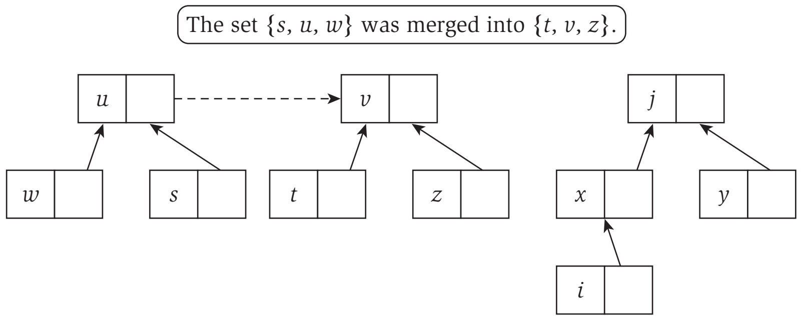

Figure 4.12 A `Union-Find` data structure using pointers. The data structure has only two sets at the moment, named after nodes v and j. The dashed arrow from u to v is the result of the last `Union` operation. To answer a `Find` query, we follow the arrows until we get to a node that has no outgoing arrow. For example, answering the query `Find`(i) would involve following the arrows i to x, and then x to j.

structure in Figure 4.12 followed this convention. To implement this choice efficiently, we will maintain an additional field with the nodes: the size of the corresponding set.

(4.24) *Consider the above pointer-based implementation of the* `Union-Find` *data structure for some set S of size n, where unions keep the name of the larger set. A* `Union` *operation takes* $O(1)$ *time,* `MakeUnionFind`(S) *takes* $O(n)$ *time, and a* `Find` *operation takes* $O(\log n)$ *time.*

Proof. The statements about `Union` and `MakeUnionFind` are easy to verify. The time to evaluate `Find`(v) for a node v is the number of times the set containing node v changes its name during the process. By the convention that the union keeps the name of the larger set, it follows that every time the name of the set containing node v changes, the size of this set at least doubles. Since the set containing v starts at size 1 and is never larger than n, its size can double at most $\log_2 n$ times, and so there can be at most $\log_2 n$ name changes. ■

Further Improvements

Next we will briefly discuss a natural optimization in the pointer-based `Union-Find` data structure that has the effect of speeding up the `Find` operations. Strictly speaking, this improvement will not be necessary for our purposes in this book: for all the applications of `Union-Find` data structures that we consider, the $O(\log n)$ time per operation is good enough in the sense that further improvement in the time for operations would not translate to improvements in the overall running time of the algorithms where we use them. (The Union-Find operations will not be the only computational bottleneck in the running time of these algorithms.)

To motivate the improved version of the data structure, let us first discuss a bad case for the running time of the pointer-based Union-Find data structure. First we build up a structure where one of the Find operations takes about $\log n$ time. To do this, we can repeatedly take Unions of equal-sized sets. Assume v is a node for which the Find(v) operation takes about $\log n$ time. Now we can issue Find(v) repeatedly, and it takes $\log n$ for each such call. Having to follow the same sequence of $\log n$ pointers every time for finding the name of the set containing v is quite redundant: after the first request for Find(v), we already "know" the name x of the set containing v, and we also know that all other nodes that we touched during our path from v to the current name also are all contained in the set x. So in the improved implementation, we will *compress* the path we followed after every Find operation by resetting all pointers along the path to point to the current name of the set. No information is lost by doing this, and it makes subsequent Find operations run more quickly. See Figure 4.13 for a Union-Find data structure and the result of Find(v) using path compression.

Now consider the running time of the operations in the resulting implementation. As before, a Union operation takes $O(1)$ time and MakeUnion-Find(S) takes $O(n)$ time to set up a data structure for a set of size n. How did the time required for a Find(v) operation change? Some Find operations can still take up to $\log n$ time; and for some Find operations we actually increase

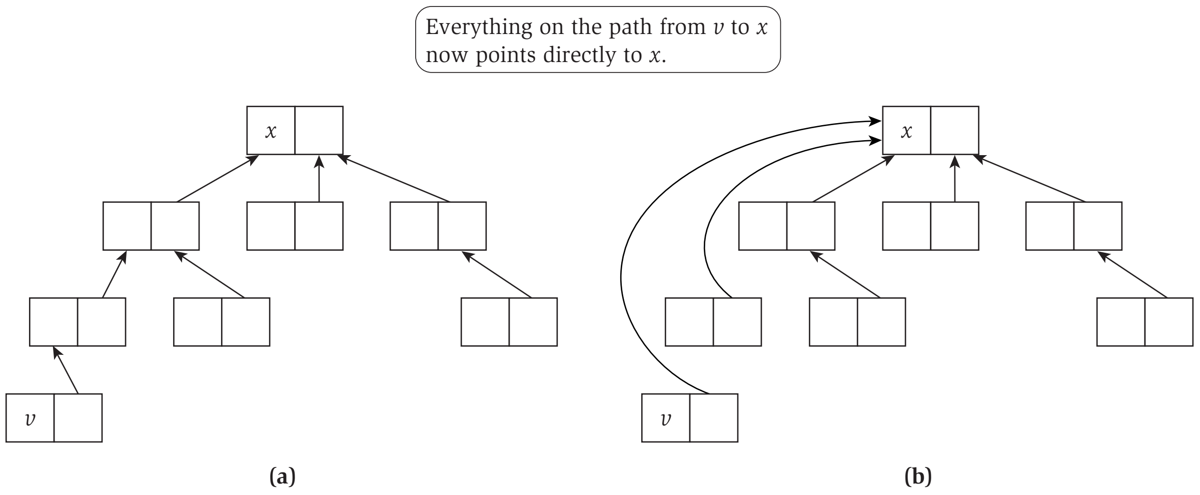
Everything on the path from v to x now points directly to x.
x
v
x
v
(a)
(b)

Figure 4.13 (a) An instance of a Union-Find data structure; and (b) the result of the operation Find(v) on this structure, using path compression.

the time, since after finding the name x of the set containing v, we have to go back through the same path of pointers from v to x, and reset each of these pointers to point to x directly. But this additional work can at most double the time required, and so does not change the fact that a `Find` takes at most $O(\log n)$ time. The real gain from compression is in making subsequent calls to `Find` cheaper, and this can be made precise by the same type of argument we used in (4.23): bounding the total time for a sequence of n `Find` operations, rather than the worst-case time for any one of them. Although we do not go into the details here, a sequence of n `Find` operations employing compression requires an amount of time that is extremely close to linear in n; the actual upper bound is $O(n\alpha(n))$, where $\alpha(n)$ is an extremely slow-growing function of n called the *inverse Ackermann function*. (In particular, $\alpha(n) \leq 4$ for any value of n that could be encountered in practice.)

Implementing Kruskal's Algorithm

Now we'll use the `Union-Find` data structure to implement Kruskal's Algorithm. First we need to sort the edges by cost. This takes time $O(m \log m)$. Since we have at most one edge between any pair of nodes, we have $m \leq n^2$ and hence this running time is also $O(m \log n)$.

After the sorting operation, we use the `Union-Find` data structure to maintain the connected components of (V, T) as edges are added. As each edge $e = (v, w)$ is considered, we compute `Find`(u) and `Find`(v) and test if they are equal to see if v and w belong to different components. We use `Union`(`Find`(u),`Find`(v)) to merge the two components, if the algorithm decides to include edge e in the tree T.

We are doing a total of at most $2m$ `Find` and $n - 1$ `Union` operations over the course of Kruskal's Algorithm. We can use either (4.23) for the array-based implementation of `Union-Find`, or (4.24) for the pointer-based implementation, to conclude that this is a total of $O(m \log n)$ time. (While more efficient implementations of the `Union-Find` data structure are possible, this would not help the running time of Kruskal's Algorithm, which has an unavoidable $O(m \log n)$ term due to the initial sorting of the edges by cost.)

To sum up, we have

(4.25) *Kruskal's Algorithm can be implemented on a graph with n nodes and m edges to run in $O(m \log n)$ time.*

4.7 Clustering

We motivated the construction of minimum spanning trees through the problem of finding a low-cost network connecting a set of sites. But minimum spanning trees arise in a range of different settings, several of which appear on the surface to be quite different from one another. An appealing example is the role that minimum spanning trees play in the area of *clustering*.

The Problem

Clustering arises whenever one has a collection of objects—say, a set of photographs, documents, or microorganisms—that one is trying to classify or organize into coherent groups. Faced with such a situation, it is natural to look first for measures of how similar or dissimilar each pair of objects is. One common approach is to define a *distance function* on the objects, with the interpretation that objects at a larger distance from one another are less similar to each other. For points in the physical world, distance may actually be related to their physical distance; but in many applications, distance takes on a much more abstract meaning. For example, we could define the distance between two species to be the number of years since they diverged in the course of evolution; we could define the distance between two images in a video stream as the number of corresponding pixels at which their intensity values differ by at least some threshold.

Now, given a distance function on the objects, the clustering problem seeks to divide them into groups so that, intuitively, objects within the same group are “close,” and objects in different groups are “far apart.” Starting from this vague set of goals, the field of clustering branches into a vast number of technically different approaches, each seeking to formalize this general notion of what a good set of groups might look like.

Clusterings of Maximum Spacing Minimum spanning trees play a role in one of the most basic formalizations, which we describe here. Suppose we are given a set U of n objects, labeled $p_1, p_2, \ldots, p_n$. For each pair, p_i and p_j, we have a numerical distance $d(p_i, p_j)$. We require only that $d(p_i, p_i) = 0$; that $d(p_i, p_j) > 0$ for distinct p_i and p_j; and that distances are symmetric: $d(p_i, p_j) = d(p_j, p_i)$.

Suppose we are seeking to divide the objects in U into k groups, for a given parameter k. We say that a *k-clustering* of U is a partition of U into k nonempty sets $C_1, C_2, \ldots, C_k$. We define the *spacing* of a k-clustering to be the minimum distance between any pair of points lying in different clusters. Given that we want points in different clusters to be far apart from one another, a natural goal is to seek the k-clustering with the maximum possible spacing.

The question now becomes the following. There are exponentially many different k-clusterings of a set U; how can we efficiently find the one that has maximum spacing?

Designing the Algorithm

To find a clustering of maximum spacing, we consider growing a graph on the vertex set U. The connected components will be the clusters, and we will try to bring nearby points together into the same cluster as rapidly as possible. (This way, they don't end up as points in different clusters that are very close together.) Thus we start by drawing an edge between the closest pair of points. We then draw an edge between the next closest pair of points. We continue adding edges between pairs of points, in order of increasing distance $d(p_i, p_j)$. In this way, we are growing a graph H on U edge by edge, with connected components corresponding to clusters. Notice that we are only interested in the connected components of the graph H, not the full set of edges; so if we are about to add the edge (p_i, p_j) and find that p_i and p_j already belong to the same cluster, we will refrain from adding the edge—it's not necessary, because it won't change the set of components. In this way, our graph-growing process will never create a cycle; so H will actually be a union of trees. Each time we add an edge that spans two distinct components, it is as though we have merged the two corresponding clusters. In the clustering literature, the iterative merging of clusters in this way is often termed *single-link clustering*, a special case of *hierarchical agglomerative clustering*. (*Agglomerative* here means that we combine clusters; *single-link* means that we do so as soon as a single link joins them together.) See Figure 4.14 for an example of an instance with $k = 3$ clusters where this algorithm partitions the points into an intuitively natural grouping.

What is the connection to minimum spanning trees? It's very simple: although our graph-growing procedure was motivated by this cluster-merging idea, our procedure is precisely Kruskal's Minimum Spanning Tree Algorithm. We are doing exactly what Kruskal's Algorithm would do if given a graph G on U in which there was an edge of cost $d(p_i, p_j)$ between each pair of nodes (p_i, p_j). The only difference is that we seek a k-clustering, so we stop the procedure once we obtain k connected components.

In other words, we are running Kruskal's Algorithm but stopping it just before it adds its last $k - 1$ edges. This is equivalent to taking the full minimum spanning tree T (as Kruskal's Algorithm would have produced it), deleting the $k - 1$ most expensive edges (the ones that we never actually added), and defining the k-clustering to be the resulting connected components $C_1, C_2, \ldots, C_k$. Thus, iteratively merging clusters is equivalent to computing a minimum spanning tree and deleting the most expensive edges.

Analyzing the Algorithm

Have we achieved our goal of producing clusters that are as spaced apart as possible? The following claim shows that we have.

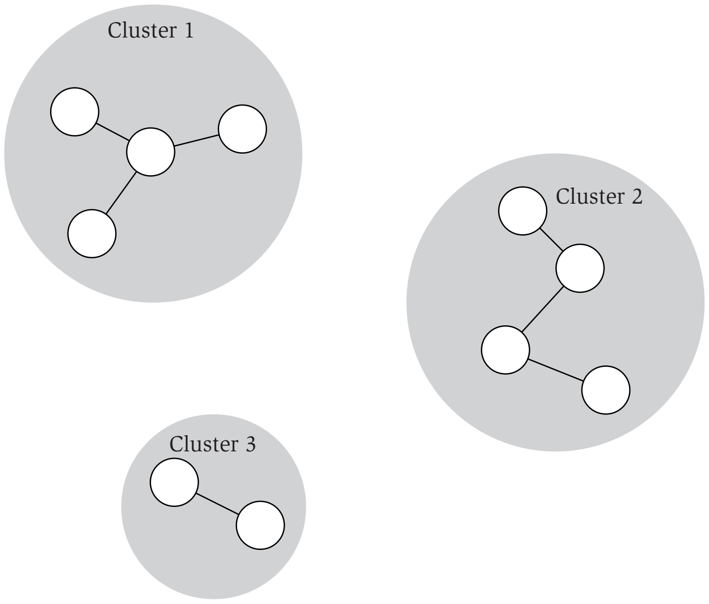

Figure 4.14 An example of single-linkage clustering with $k = 3$ clusters. The clusters are formed by adding edges between points in order of increasing distance.

(4.26) *The components $C_1, C_2, \ldots, C_k$ formed by deleting the $k - 1$ most expensive edges of the minimum spanning tree T constitute a k-clustering of maximum spacing.*

Proof. Let $\mathcal{C}$ denote the clustering $C_1, C_2, \ldots, C_k$. The spacing of $\mathcal{C}$ is precisely the length d^* of the $(k-1)^{\text{st}}$ most expensive edge in the minimum spanning tree; this is the length of the edge that Kruskal's Algorithm would have added next, at the moment we stopped it.

Now consider some other k-clustering $\mathcal{C}'$, which partitions U into nonempty sets $C'_1, C'_2, \ldots, C'_k$. We must show that the spacing of $\mathcal{C}'$ is at most d^*.

Since the two clusterings $\mathcal{C}$ and $\mathcal{C}'$ are not the same, it must be that one of our clusters C_r is not a subset of any of the k sets C'_s in $\mathcal{C}'$. Hence there are points $p_i, p_j \in C_r$ that belong to different clusters in $\mathcal{C}'$—say, $p_i \in C'_s$ and $p_j \in C'_t \neq C'_s$.

Now consider the picture in Figure 4.15. Since p_i and p_j belong to the same component C_r, it must be that Kruskal's Algorithm added all the edges of a p_i-p_j path P before we stopped it. In particular, this means that each edge on

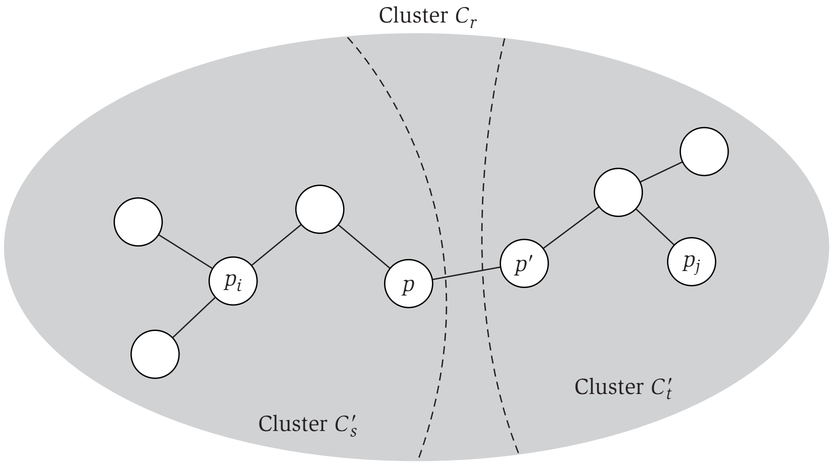

Figure 4.15 An illustration of the proof of (4.26), showing that the spacing of any other clustering can be no larger than that of the clustering found by the single-linkage algorithm.

P has length at most d^*. Now, we know that $p_i \in C'_s$ but $p_j \notin C'_s$; so let p' be the first node on P that does not belong to C'_s, and let p be the node on P that comes just before p'. We have just argued that $d(p, p') \le d^*$, since the edge (p, p') was added by Kruskal's Algorithm. But p and p' belong to different sets in the clustering $\mathcal{C}'$, and hence the spacing of $\mathcal{C}'$ is at most $d(p, p') \le d^*$. This completes the proof. ∎

4.8 Huffman Codes and Data Compression

In the Shortest-Path and Minimum Spanning Tree Problems, we've seen how greedy algorithms can be used to commit to certain parts of a solution (edges in a graph, in these cases), based entirely on relatively short-sighted considerations. We now consider a problem in which this style of "committing" is carried out in an even looser sense: a greedy rule is used, essentially, to shrink the size of the problem instance, so that an equivalent smaller problem can then be solved by recursion. The greedy operation here is proved to be "safe," in the sense that solving the smaller instance still leads to an optimal solution for the original instance, but the global consequences of the initial greedy decision do not become fully apparent until the full recursion is complete.

The problem itself is one of the basic questions in the area of *data compression*, an area that forms part of the foundations for digital communication.

The Problem

Encoding Symbols Using Bits Since computers ultimately operate on sequences of *bits* (i.e., sequences consisting only of the symbols 0 and 1), one needs encoding schemes that take text written in richer alphabets (such as the alphabets underpinning human languages) and converts this text into long strings of bits.

The simplest way to do this would be to use a fixed number of bits for each symbol in the alphabet, and then just concatenate the bit strings for each symbol to form the text. To take a basic example, suppose we wanted to encode the 26 letters of English, plus the space (to separate words) and five punctuation characters: comma, period, question mark, exclamation point, and apostrophe. This would give us 32 symbols in total to be encoded. Now, you can form 2^b different sequences out of b bits, and so if we use 5 bits per symbol, then we can encode $2^5 = 32$ symbols—just enough for our purposes. So, for example, we could let the bit string 00000 represent a, the bit string 00001 represent b, and so forth up to 11111, which could represent the apostrophe. Note that the mapping of bit strings to symbols is arbitrary; the point is simply that five bits per symbol is sufficient. In fact, encoding schemes like ASCII work precisely this way, except that they use a larger number of bits per symbol so as to handle larger character sets, including capital letters, parentheses, and all those other special symbols you see on a typewriter or computer keyboard.

Let's think about our bare-bones example with just 32 symbols. Is there anything more we could ask for from an encoding scheme? We couldn't ask to encode each symbol using just four bits, since 2^4 is only 16—not enough for the number of symbols we have. Nevertheless, it's not clear that over large stretches of text, we really need to be spending an *average* of five bits per symbol. If we think about it, the letters in most human alphabets do not get used equally frequently. In English, for example, the letters e, t, a, o, i, and n get used much more frequently than q, j, x, and z (by more than an order of magnitude). So it's really a tremendous waste to translate them all into the same number of bits; instead we could use a small number of bits for the frequent letters, and a larger number of bits for the less frequent ones, and hope to end up using fewer than five bits per letter when we average over a long string of typical text.

This issue of reducing the average number of bits per letter is a fundamental problem in the area of *data compression*. When large files need to be shipped across communication networks, or stored on hard disks, it's important to represent them as compactly as possible, subject to the requirement that a subsequent reader of the file should be able to correctly reconstruct it. A huge amount of research is devoted to the design of *compression algorithms* that can take files as input and reduce their space through efficient encoding schemes.

We now describe one of the fundamental ways of formulating this issue, building up to the question of how we might construct the *optimal* way to take advantage of the nonuniform frequencies of the letters. In one sense, such an optimal solution is a very appealing answer to the problem of compressing data: it squeezes all the available gains out of nonuniformities in the frequencies. At the end of the section, we will discuss how one can make further progress in compression, taking advantage of features other than nonuniform frequencies.

Variable-Length Encoding Schemes Before the Internet, before the digital computer, before the radio and telephone, there was the telegraph. Communicating by telegraph was a lot faster than the contemporary alternatives of hand-delivering messages by railroad or on horseback. But telegraphs were only capable of transmitting pulses down a wire, and so if you wanted to send a message, you needed a way to encode the text of your message as a sequence of pulses.

To deal with this issue, the pioneer of telegraphic communication, Samuel Morse, developed *Morse code*, translating each letter into a sequence of *dots* (short pulses) and *dashes* (long pulses). For our purposes, we can think of dots and dashes as zeros and ones, and so this is simply a mapping of symbols into bit strings, just as in ASCII. Morse understood the point that one could communicate more efficiently by encoding frequent letters with short strings, and so this is the approach he took. (He consulted local printing presses to get frequency estimates for the letters in English.) Thus, Morse code maps e to 0 (a single dot), t to 1 (a single dash), a to 01 (dot-dash), and in general maps more frequent letters to shorter bit strings.

In fact, Morse code uses such short strings for the letters that the encoding of words becomes ambiguous. For example, just using what we know about the encoding of e, t, and a, we see that the string 0101 could correspond to any of the sequences of letters *eta*, *aa*, *etet*, or *aet*. (There are other possibilities as well, involving other letters.) To deal with this ambiguity, Morse code transmissions involve short pauses between letters (so the encoding of *aa* would actually be dot-dash-pause-dot-dash-pause). This is a reasonable solution—using very short bit strings and then introducing pauses—but it means that we haven't actually encoded the letters using just 0 and 1; we've actually encoded it using a three-letter alphabet of 0, 1, and "pause." Thus, if we really needed to encode everything using only the bits 0 and 1, there would need to be some further encoding in which the pause got mapped to bits.

Prefix Codes The ambiguity problem in Morse code arises because there exist pairs of letters where the bit string that encodes one letter is a *prefix* of the bit string that encodes another. To eliminate this problem, and hence to obtain an encoding scheme that has a well-defined interpretation for every sequence of bits, it is enough to map letters to bit strings in such a way that no encoding is a prefix of any other.

Concretely, we say that a *prefix code* for a set S of letters is a function γ that maps each letter $x \in S$ to some sequence of zeros and ones, in such a way that for distinct $x, y \in S$, the sequence $\gamma(x)$ is not a prefix of the sequence $\gamma(y)$.

Now suppose we have a text consisting of a sequence of letters $x_1x_2x_3 \cdots x_n$. We can convert this to a sequence of bits by simply encoding each letter as a bit sequence using γ and then concatenating all these bit sequences together: $\gamma(x_1)\gamma(x_2)\cdots\gamma(x_n)$. If we then hand this message to a recipient who knows the function γ, they will be able to reconstruct the text according to the following rule.

- Scan the bit sequence from left to right.
- As soon as you've seen enough bits to match the encoding of some letter, output this as the first letter of the text. This must be the correct first letter, since no shorter or longer prefix of the bit sequence could encode any other letter.
- Now delete the corresponding set of bits from the front of the message and iterate.

In this way, the recipient can produce the correct set of letters without our having to resort to artificial devices like pauses to separate the letters.

For example, suppose we are trying to encode the set of five letters $S = \{a, b, c, d, e\}$. The encoding γ_1 specified by

$$\begin{aligned}
\gamma_1(a) &= 11 \\
\gamma_1(b) &= 01 \\
\gamma_1(c) &= 001 \\
\gamma_1(d) &= 10 \\
\gamma_1(e) &= 000
\end{aligned}$$

is a prefix code, since we can check that no encoding is a prefix of any other. Now, for example, the string *cecab* would be encoded as 0010000011101. A recipient of this message, knowing γ_1, would begin reading from left to right. Neither 0 nor 00 encodes a letter, but 001 does, so the recipient concludes that the first letter is c. This is a safe decision, since no longer sequence of bits beginning with 001 could encode a different letter. The recipient now iterates on the rest of the message, 0000011101; next they will conclude that the second letter is e, encoded as 000.

Optimal Prefix Codes We've been doing all this because some letters are more frequent than others, and we want to take advantage of the fact that more frequent letters can have shorter encodings. To make this objective precise, we now introduce some notation to express the frequencies of letters.

Suppose that for each letter $x \in S$, there is a frequency f_x, representing the fraction of letters in the text that are equal to x. In other words, assuming there are n letters total, nf_x of these letters are equal to x. We notice that the frequencies sum to 1; that is, $\sum_{x \in S} f_x = 1$.

Now, if we use a prefix code γ to encode the given text, what is the total length of our encoding? This is simply the sum, over all letters $x \in S$, of the number of times x occurs times the length of the bit string $\gamma(x)$ used to encode x. Using $|\gamma(x)|$ to denote the length $\gamma(x)$, we can write this as

$$\text{encoding length} = \sum_{x \in S} nf_x|\gamma(x)| = n \sum_{x \in S} f_x|\gamma(x)|.$$

Dropping the leading coefficient of n from the final expression gives us $\sum_{x \in S} f_x|\gamma(x)|$, the *average* number of bits required per letter. We denote this quantity by $\text{ABL}(\gamma)$.

To continue the earlier example, suppose we have a text with the letters $S = \{a, b, c, d, e\}$, and their frequencies are as follows:

$$f_a = .32, \quad f_b = .25, \quad f_c = .20, \quad f_d = .18, \quad f_e = .05.$$

Then the average number of bits per letter using the prefix code γ_1 defined previously is

$$.32 \cdot 2 + .25 \cdot 2 + .20 \cdot 3 + .18 \cdot 2 + .05 \cdot 3 = 2.25.$$

It is interesting to compare this to the average number of bits per letter using a fixed-length encoding. (Note that a fixed-length encoding is a prefix code: if all letters have encodings of the same length, then clearly no encoding can be a prefix of any other.) With a set S of five letters, we would need three bits per letter for a fixed-length encoding, since two bits could only encode four letters. Thus, using the code γ_1 reduces the bits per letter from 3 to 2.25, a savings of 25 percent.

And, in fact, γ_1 is not the best we can do in this example. Consider the prefix code γ_2 given by

$$\begin{aligned}\gamma_2(a) &= 11\\ \gamma_2(b) &= 10\\ \gamma_2(c) &= 01\\ \gamma_2(d) &= 001\\ \gamma_2(e) &= 000\end{aligned}$$

The average number of bits per letter using γ_2 is

$$.32 \cdot 2 + .25 \cdot 2 + .20 \cdot 2 + .18 \cdot 3 + .05 \cdot 3 = 2.23.$$

So now it is natural to state the underlying question. Given an alphabet and a set of frequencies for the letters, we would like to produce a prefix code that is as efficient as possible—namely, a prefix code that minimizes the average number of bits per letter $\text{ABL}(\gamma) = \sum_{x \in S} f_x |\gamma(x)|$. We will call such a prefix code *optimal*.

Designing the Algorithm

The search space for this problem is fairly complicated; it includes all possible ways of mapping letters to bit strings, subject to the defining property of prefix codes. For alphabets consisting of an extremely small number of letters, it is feasible to search this space by brute force, but this rapidly becomes infeasible.

We now describe a greedy method to construct an optimal prefix code very efficiently. As a first step, it is useful to develop a tree-based means of representing prefix codes that exposes their structure more clearly than simply the lists of function values we used in our previous examples.

Representing Prefix Codes Using Binary Trees Suppose we take a rooted tree T in which each node that is not a leaf has at most two children; we call such a tree a *binary tree*. Further suppose that the number of leaves is equal to the size of the alphabet S, and we label each leaf with a distinct letter in S.

Such a labeled binary tree T naturally describes a prefix code, as follows. For each letter $x \in S$, we follow the path from the root to the leaf labeled x; each time the path goes from a node to its left child, we write down a 0, and each time the path goes from a node to its right child, we write down a 1. We take the resulting string of bits as the encoding of x.

Now we observe

(4.27) *The encoding of S constructed from T is a prefix code.*

Proof. In order for the encoding of x to be a prefix of the encoding of y, the path from the root to x would have to be a prefix of the path from the root to y. But this is the same as saying that x would lie on the path from the root to y, which isn't possible if x is a leaf. ∎

This relationship between binary trees and prefix codes works in the other direction as well. Given a prefix code γ, we can build a binary tree recursively as follows. We start with a root; all letters $x \in S$ whose encodings begin with a 0 will be leaves in the left subtree of the root, and all letters $y \in S$ whose encodings begin with a 1 will be leaves in the right subtree of the root. We now build these two subtrees recursively using this rule.

For example, the labeled tree in Figure 4.16(a) corresponds to the prefix code γ_0 specified by

$$\begin{aligned} \gamma_0(a) &= 1 \\ \gamma_0(b) &= 011 \\ \gamma_0(c) &= 010 \\ \gamma_0(d) &= 001 \\ \gamma_0(e) &= 000 \end{aligned}$$

To see this, note that the leaf labeled a is obtained by simply taking the right-hand edge out of the root (resulting in an encoding of 1); the leaf labeled e is obtained by taking three successive left-hand edges starting from the root; and analogous explanations apply for b, c, and d. By similar reasoning, one can see that the labeled tree in Figure 4.16(b) corresponds to the prefix code γ_1 defined earlier, and the labeled tree in Figure 4.16(c) corresponds to the prefix code γ_2 defined earlier. Note also that the binary trees for the two prefix codes γ_1 and γ_2 are identical in structure; only the labeling of the leaves is different. The tree for γ_0, on the other hand, has a different structure.

Thus the search for an optimal prefix code can be viewed as the search for a binary tree T, together with a labeling of the leaves of T, that minimizes the average number of bits per letter. Moreover, this average quantity has a natural interpretation in the terms of the structure of T: the length of the encoding of a letter $x \in S$ is simply the length of the path from the root to the leaf labeled x. We will refer to the length of this path as the *depth* of the leaf, and we will denote the depth of a leaf v in T simply by $\text{depth}_T(v)$. (As two bits of notational convenience, we will drop the subscript T when it is clear from context, and we will often use a letter $x \in S$ to also denote the leaf that is labeled by it.) Thus we are seeking the labeled tree that minimizes the weighted average of the depths of all leaves, where the average is weighted by the frequencies of the letters that label the leaves: $\sum_{x \in S} f_x \cdot \text{depth}_T(x)$. We will use $\text{ABL}(T)$ to denote this quantity.

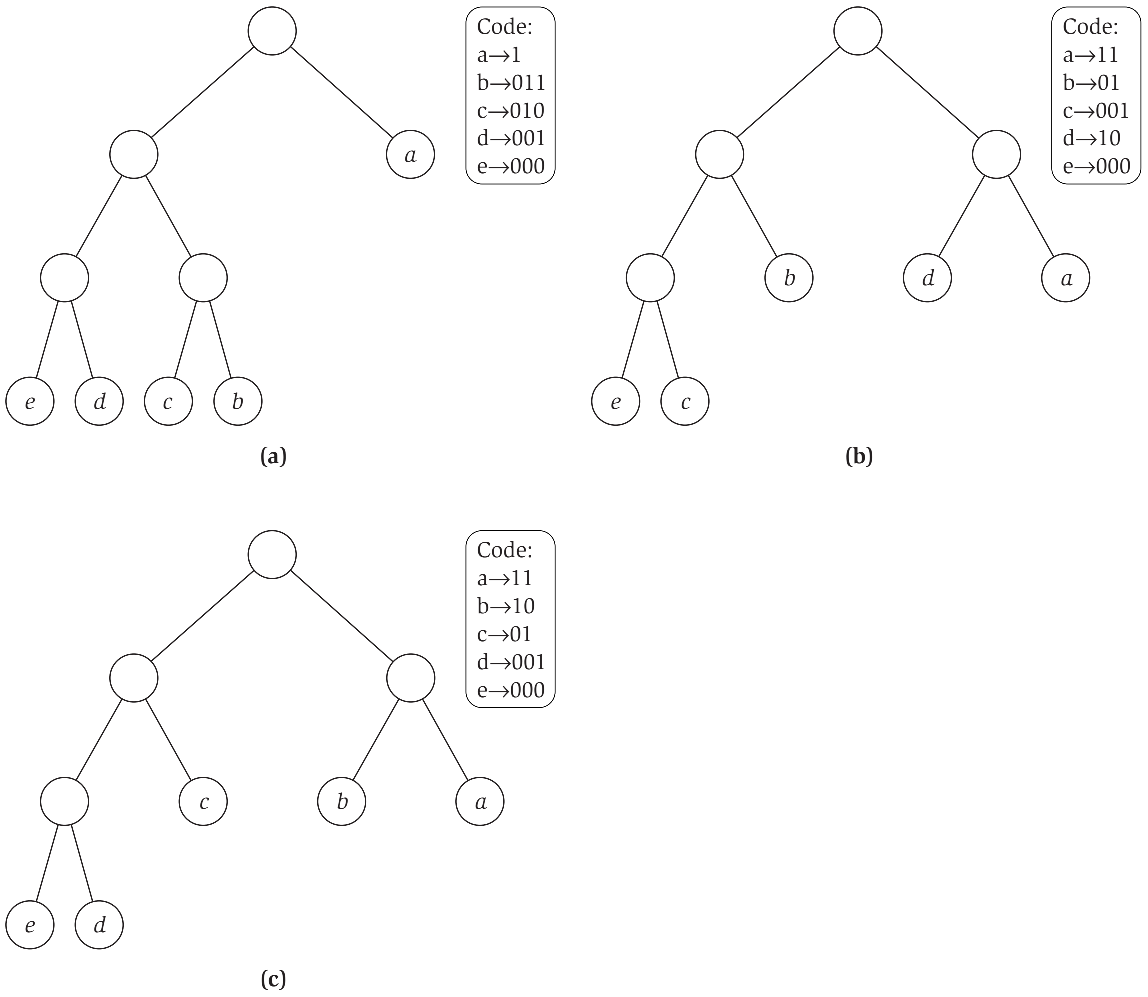

Figure 4.16 Parts (a), (b), and (c) of the figure depict three different prefix codes for the alphabet $S = \{a, b, c, d, e\}$.

As a first step in considering algorithms for this problem, let's note a simple fact about the optimal tree. For this fact, we need a definition: we say that a binary tree is *full* if each node that is not a leaf has two children. (In other words, there are no nodes with exactly one child.) Note that all three binary trees in Figure 4.16 are full.

(4.28) *The binary tree corresponding to the optimal prefix code is full.*

Proof. This is easy to prove using an exchange argument. Let T denote the binary tree corresponding to the optimal prefix code, and suppose it contains a node u with exactly one child v. Now convert T into a tree T' by replacing node u with v.

To be precise, we need to distinguish two cases. If u was the root of the tree, we simply delete node u and use v as the root. If u is not the root, let w be the parent of u in T. Now we delete node u and make v be a child of w in place of u. This change decreases the number of bits needed to encode any leaf in the subtree rooted at node u, and it does not affect other leaves. So the prefix code corresponding to T' has a smaller average number of bits per letter than the prefix code for T, contradicting the optimality of T. ∎

A First Attempt: The Top-Down Approach Intuitively, our goal is to produce a labeled binary tree in which the leaves are as close to the root as possible. This is what will give us a small average leaf depth.

A natural way to do this would be to try building a tree from the top down by "packing" the leaves as tightly as possible. So suppose we try to split the alphabet S into two sets S_1 and S_2, such that the total frequency of the letters in each set is exactly $\frac{1}{2}$. If such a perfect split is not possible, then we can try for a split that is as nearly balanced as possible. We then recursively construct prefix codes for S_1 and S_2 independently, and make these the two subtrees of the root. (In terms of bit strings, this would mean sticking a 0 in front of the encodings we produce for S_1, and sticking a 1 in front of the encodings we produce for S_2.)

It is not entirely clear how we should concretely define this "nearly balanced" split of the alphabet, but there are ways to make this precise. The resulting encoding schemes are called *Shannon-Fano* codes, named after Claude Shannon and Robert Fano, two of the major early figures in the area of *information theory*, which deals with representing and encoding digital information. These types of prefix codes can be fairly good in practice, but for our present purposes they represent a kind of dead end: no version of this top-down splitting strategy is guaranteed to always produce an optimal prefix code. Consider again our example with the five-letter alphabet $S = \{a, b, c, d, e\}$ and frequencies

$$f_a = .32, \quad f_b = .25, \quad f_c = .20, \quad f_d = .18, \quad f_e = .05.$$

There is a unique way to split the alphabet into two sets of equal frequency: $\{a, d\}$ and $\{b, c, e\}$. For $\{a, d\}$, we can use a single bit to encode each. For $\{b, c, e\}$, we need to continue recursively, and again there is a unique way to split the set into two subsets of equal frequency. The resulting code corresponds to the code γ_1, given by the labeled tree in Figure 4.16(b); and we've already seen that γ_1 is not as efficient as the prefix code γ_2 corresponding to the labeled tree in Figure 4.16(c).

Shannon and Fano knew that their approach did not always yield the optimal prefix code, but they didn't see how to compute the optimal code without brute-force search. The problem was solved a few years later by David Huffman, at the time a graduate student who learned about the question in a class taught by Fano.

We now describe the ideas leading up to the greedy approach that Huffman discovered for producing optimal prefix codes.

What If We Knew the Tree Structure of the Optimal Prefix Code? A technique that is often helpful in searching for an efficient algorithm is to assume, as a thought experiment, that one knows something partial about the optimal solution, and then to see how one would make use of this partial knowledge in finding the complete solution. (Later, in Chapter 6, we will see in fact that this technique is a main underpinning of the *dynamic programming* approach to designing algorithms.)

For the current problem, it is useful to ask: What if someone gave us the binary tree T^* that corresponded to an optimal prefix code, but not the labeling of the leaves? To complete the solution, we would need to figure out which letter should label which leaf of T^*, and then we'd have our code. How hard is this?

In fact, this is quite easy. We begin by formulating the following basic fact.

(4.29) *Suppose that u and v are leaves of T^*, such that depth(u) < depth(v). Further, suppose that in a labeling of T^* corresponding to an optimal prefix code, leaf u is labeled with $y \in S$ and leaf v is labeled with $z \in S$. Then $f_y \geq f_z$.*

Proof. This has a quick proof using an exchange argument. If $f_y < f_z$, then consider the code obtained by exchanging the labels at the nodes u and v. In the expression for the average number of bits per letter, $\text{ABL}(T^*) = \sum_{x \in S} f_x \operatorname{depth}(x)$, the effect of this exchange is as follows: the multiplier on f_y increases (from depth(u) to depth(v)), and the multiplier on f_z decreases by the same amount (from depth(v) to depth(u)).

Thus the change to the overall sum is $(\operatorname{depth}(v) - \operatorname{depth}(u))(f_y - f_z)$. If $f_y < f_z$, this change is a negative number, contradicting the supposed optimality of the prefix code that we had before the exchange. ∎

We can see the idea behind (4.29) in Figure 4.16(b): a quick way to see that the code here is not optimal is to notice that it can be improved by exchanging the positions of the labels c and d. Having a lower-frequency letter at a strictly smaller depth than some other higher-frequency letter is precisely what (4.29) rules out for an optimal solution.

Statement (4.29) gives us the following intuitively natural, and optimal, way to label the tree T^* if someone should give it to us. We first take all leaves of depth 1 (if there are any) and label them with the highest-frequency letters in any order. We then take all leaves of depth 2 (if there are any) and label them with the next-highest-frequency letters in any order. We continue through the leaves in order of increasing depth, assigning letters in order of decreasing frequency. The point is that this can't lead to a suboptimal labeling of T^*, since any supposedly better labeling would be susceptible to the exchange in (4.29). It is also crucial to note that, among the labels we assign to a block of leaves all at the same depth, it doesn't matter which label we assign to which leaf. Since the depths are all the same, the corresponding multipliers in the expression $\sum_{x \in S} f_x |\gamma(x)|$ are the same, and so the choice of assignment among leaves of the same depth doesn't affect the average number of bits per letter.

But how is all this helping us? We don't have the structure of the optimal tree T^*, and since there are exponentially many possible trees (in the size of the alphabet), we aren't going to be able to perform a brute-force search over all of them.

In fact, our reasoning about T^* becomes very useful if we think not about the very beginning of this labeling process, with the leaves of minimum depth, but about the very end, with the leaves of maximum depth—the ones that receive the letters with lowest frequency. Specifically, consider a leaf v in T^* whose depth is as large as possible. Leaf v has a parent u, and by (4.28) T^* is a full binary tree, so u has another child w. We refer to v and w as *siblings*, since they have a common parent. Now, we have

(4.30) *w is a leaf of T^*.*

Proof. If w were not a leaf, there would be some leaf w' in the subtree below it. But then w' would have a depth greater than that of v, contradicting our assumption that v is a leaf of maximum depth in T^*. ∎

So v and w are sibling leaves that are as deep as possible in T^*. Thus our level-by-level process of labeling T^*, as justified by (4.29), will get to the level containing v and w last. The leaves at this level will get the lowest-frequency letters. Since we have already argued that the order in which we assign these letters to the leaves within this level doesn't matter, there is an optimal labeling in which v and w get the two lowest-frequency letters of all.

We sum this up in the following claim.

(4.31) *There is an optimal prefix code, with corresponding tree T^*, in which the two lowest-frequency letters are assigned to leaves that are siblings in T^*.*

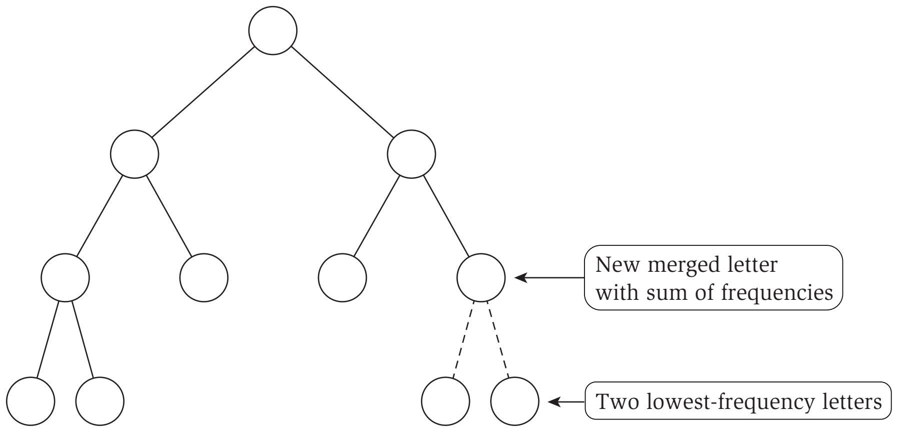
New merged letter with sum of frequencies
Two lowest-frequency letters

Figure 4.17 There is an optimal solution in which the two lowest-frequency letters label sibling leaves; deleting them and labeling their parent with a new letter having the combined frequency yields an instance with a smaller alphabet.

An Algorithm to Construct an Optimal Prefix Code Suppose that y^* and z^* are the two lowest-frequency letters in S. (We can break ties in the frequencies arbitrarily.) Statement (4.31) is important because it tells us something about where y^* and z^* go in the optimal solution; it says that it is safe to "lock them together" in thinking about the solution, because we know they end up as sibling leaves below a common parent. In effect, this common parent acts like a "meta-letter" whose frequency is the sum of the frequencies of y^* and z^*.

This directly suggests an algorithm: we replace y^* and z^* with this meta-letter, obtaining an alphabet that is one letter smaller. We recursively find a prefix code for the smaller alphabet, and then "open up" the meta-letter back into y^* and z^* to obtain a prefix code for S. This recursive strategy is depicted in Figure 4.17.

A concrete description of the algorithm is as follows.

```
To construct a prefix code for an alphabet $S$, with given frequencies:
  If $S$ has two letters then
    Encode one letter using 0 and the other letter using 1
  Else
    Let $y^*$ and $z^*$ be the two lowest-frequency letters
    Form a new alphabet $S'$ by deleting $y^*$ and $z^*$ and
      replacing them with a new letter $\omega$ of frequency $f_{y^*} + f_{z^*}$
    Recursively construct a prefix code $\gamma'$ for $S'$, with tree $T'$
    Define a prefix code for $S$ as follows:
      Start with $T'$
      Take the leaf labeled $\omega$ and add two children below it
        labeled $y^*$ and $z^*$
   Endif
```

We refer to this as *Huffman's Algorithm*, and the prefix code that it produces for a given alphabet is accordingly referred to as a *Huffman code*. In general, it is clear that this algorithm always terminates, since it simply invokes a recursive call on an alphabet that is one letter smaller. Moreover, using (4.31), it will not be difficult to prove that the algorithm in fact produces an optimal prefix code. Before doing this, however, we pause to note some further observations about the algorithm.

First let's consider the behavior of the algorithm on our sample instance with $S = \{a, b, c, d, e\}$ and frequencies

$$f_a = .32, \quad f_b = .25, \quad f_c = .20, \quad f_d = .18, \quad f_e = .05.$$

The algorithm would first merge d and e into a single letter—let's denote it (de)—of frequency $.18 + .05 = .23$. We now have an instance of the problem on the four letters $S' = \{a, b, c, (de)\}$. The two lowest-frequency letters in S' are c and (de), so in the next step we merge these into the single letter (cde) of frequency $.20 + .23 = .43$. This gives us the three-letter alphabet $\{a, b, (cde)\}$. Next we merge a and b, and this gives us a two-letter alphabet, at which point we invoke the base case of the recursion. If we unfold the result back through the recursive calls, we get the tree pictured in Figure 4.16(c).

It is interesting to note how the greedy rule underlying Huffman's Algorithm—the merging of the two lowest-frequency letters—fits into the structure of the algorithm as a whole. Essentially, at the time we merge these two letters, we don't know exactly how they will fit into the overall code. Rather, we simply commit to having them be children of the same parent, and this is enough to produce a new, equivalent problem with one less letter.

Moreover, the algorithm forms a natural contrast with the earlier approach that led to suboptimal Shannon-Fano codes. That approach was based on a top-down strategy that worried first and foremost about the top-level split in the binary tree—namely, the two subtrees directly below the root. Huffman's Algorithm, on the other hand, follows a bottom-up approach: it focuses on the leaves representing the two lowest-frequency letters, and then continues by recursion.

Analyzing the Algorithm

The Optimality of the Algorithm We first prove the optimality of Huffman's Algorithm. Since the algorithm operates recursively, invoking itself on smaller and smaller alphabets, it is natural to try establishing optimality by induction on the size of the alphabet. Clearly it is optimal for all two-letter alphabets (since it uses only one bit per letter). So suppose by induction that it is optimal for all alphabets of size $k-1$, and consider an input instance consisting of an alphabet S of size k.

Let's quickly recap the behavior of the algorithm on this instance. The algorithm merges the two lowest-frequency letters $y^*, z^* \in S$ into a single letter ω, calls itself recursively on the smaller alphabet S' (in which y^* and z^* are replaced by ω), and by induction produces an optimal prefix code for S', represented by a labeled binary tree T'. It then extends this into a tree T for S, by attaching leaves labeled y^* and z^* as children of the node in T' labeled ω.

There is a close relationship between $\text{ABL}(T)$ and $\text{ABL}(T')$. (Note that the former quantity is the average number of bits used to encode letters in S, while the latter quantity is the average number of bits used to encode letters in S'.)

(4.32) $\text{ABL}(T') = \text{ABL}(T) - f_\omega$.

Proof. The depth of each letter x other than y^*, z^* is the same in both T and T'. Also, the depths of y^* and z^* in T are each one greater than the depth of ω in T'. Using this, plus the fact that $f_\omega = f_{y^*} + f_{z^*}$, we have

$$\begin{aligned}
\text{ABL}(T) &= \sum_{x\in S} f_x \cdot \text{depth}_T(x) \\
&= f_{y^*} \cdot \text{depth}_T(y^*) + f_{z^*} \cdot \text{depth}_T(z^*) + \sum_{x\neq y^*,z^*} f_x \cdot \text{depth}_T(x) \\
&= (f_{y^*} + f_{z^*}) \cdot (1 + \text{depth}_{T'}(\omega)) + \sum_{x\neq y^*,z^*} f_x \cdot \text{depth}_{T'}(x) \\
&= f_\omega \cdot (1 + \text{depth}_{T'}(\omega)) + \sum_{x\neq y^*,z^*} f_x \cdot \text{depth}_{T'}(x) \\
&= f_\omega + f_\omega \cdot \text{depth}_{T'}(\omega) + \sum_{x\neq y^*,z^*} f_x \cdot \text{depth}_{T'}(x) \\
&= f_\omega + \sum_{x\in S'} f_x \cdot \text{depth}_{T'}(x) \\
&= f_\omega + \text{ABL}(T'). \quad \blacksquare
\end{aligned}$$

Using this, we now prove optimality as follows.

(4.33) *The Huffman code for a given alphabet achieves the minimum average number of bits per letter of any prefix code.*

Proof. Suppose by way of contradiction that the tree T produced by our greedy algorithm is not optimal. This means that there is some labeled binary tree Z

such that $\text{ABL}(Z) < \text{ABL}(T)$; and by (4.31), there is such a tree Z in which the leaves representing y^* and z^* are siblings.

It is now easy to get a contradiction, as follows. If we delete the leaves labeled y^* and z^* from Z, and label their former parent with ω, we get a tree Z' that defines a prefix code for S'. In the same way that T is obtained from T', the tree Z is obtained from Z' by adding leaves for y^* and z^* below ω; thus the identity in (4.32) applies to Z and Z' as well: $\text{ABL}(Z') = \text{ABL}(Z) - f_\omega$.

But we have assumed that $\text{ABL}(Z) < \text{ABL}(T)$; subtracting f_ω from both sides of this inequality we get $\text{ABL}(Z') < \text{ABL}(T')$, which contradicts the optimality of T' as a prefix code for S'. ■

Implementation and Running Time It is clear that Huffman's Algorithm can be made to run in polynomial time in k, the number of letters in the alphabet. The recursive calls of the algorithm define a sequence of $k - 1$ iterations over smaller and smaller alphabets, and each iteration except the last consists simply of identifying the two lowest-frequency letters and merging them into a single letter that has the combined frequency. Even without being careful about the implementation, identifying the lowest-frequency letters can be done in a single scan of the alphabet, in time $O(k)$, and so summing this over the $k - 1$ iterations gives $O(k^2)$ time.

But in fact Huffman's Algorithm is an ideal setting in which to use a priority queue. Recall that a priority queue maintains a set of k elements, each with a numerical key, and it allows for the insertion of new elements and the extraction of the element with the minimum key. Thus we can maintain the alphabet S in a priority queue, using each letter's frequency as its key. In each iteration we just extract the minimum twice (this gives us the two lowest-frequency letters), and then we insert a new letter whose key is the sum of these two minimum frequencies. Our priority queue now contains a representation of the alphabet that we need for the next iteration.

Using an implementation of priority queues via heaps, as in Chapter 2, we can make each insertion and extraction of the minimum run in time $O(\log k)$; hence, each iteration—which performs just three of these operations—takes time $O(\log k)$. Summing over all k iterations, we get a total running time of $O(k \log k)$.

Extensions

The structure of optimal prefix codes, which has been our focus here, stands as a fundamental result in the area of data compression. But it is important to understand that this optimality result does not by any means imply that we have found the best way to compress data under all circumstances.

What more could we want beyond an optimal prefix code? First, consider an application in which we are transmitting black-and-white images: each image is a 1,000-by-1,000 array of pixels, and each pixel takes one of the two values *black* or *white*. Further, suppose that a typical image is almost entirely white: roughly 1,000 of the million pixels are black, and the rest are white. Now, if we wanted to compress such an image, the whole approach of prefix codes has very little to say: we have a text of length one million over the two-letter alphabet $\{black, white\}$. As a result, the text is already encoded using one bit per letter—the lowest possible in our framework.

It is clear, though, that such images should be highly compressible. Intuitively, one ought to be able to use a "fraction of a bit" for each white pixel, since they are so overwhelmingly frequent, at the cost of using multiple bits for each black pixel. (In an extreme version, sending a list of (x, y) coordinates for each black pixel would be an improvement over sending the image as a text with a million bits.) The challenge here is to define an encoding scheme where the notion of using fractions of bits is well-defined. There are results in the area of data compression, however, that do just this; *arithmetic coding* and a range of other techniques have been developed to handle settings like this.

A second drawback of prefix codes, as defined here, is that they cannot *adapt* to changes in the text. Again let's consider a simple example. Suppose we are trying to encode the output of a program that produces a long sequence of letters from the set $\{a, b, c, d\}$. Further suppose that for the first half of this sequence, the letters a and b occur equally frequently, while c and d do not occur at all; but in the second half of this sequence, the letters c and d occur equally frequently, while a and b do not occur at all. In the framework developed in this section, we are trying to compress a text over the four-letter alphabet $\{a, b, c, d\}$, and all letters are equally frequent. Thus each would be encoded with two bits.

But what's really happening in this example is that the frequency remains stable for half the text, and then it changes radically. So one could get away with just one bit per letter, plus a bit of extra overhead, as follows.

- Begin with an encoding in which the bit 0 represents a and the bit 1 represents b.
- Halfway into the sequence, insert some kind of instruction that says, "We're changing the encoding now. From now on, the bit 0 represents c and the bit 1 represents d."
- Use this new encoding for the rest of the sequence.

The point is that investing a small amount of space to describe a new encoding can pay off many times over if it reduces the average number of bits per letter over a long run of text that follows. Such approaches, which change the encoding in midstream, are called *adaptive* compression schemes, and for many kinds of data they lead to significant improvements over the static method we've considered here.

These issues suggest some of the directions in which work on data compression has proceeded. In many of these cases, there is a trade-off between the power of the compression technique and its computational cost. In particular, many of the improvements to Huffman codes just described come with a corresponding increase in the computational effort needed both to produce the compressed version of the data and also to decompress it and restore the original text. Finding the right balance among these trade-offs is a topic of active research.

* 4.9 Minimum-Cost Arborescences: A Multi-Phase Greedy Algorithm

As we've seen more and more examples of greedy algorithms, we've come to appreciate that there can be considerable diversity in the way they operate. Many greedy algorithms make some sort of an initial "ordering" decision on the input, and then process everything in a one-pass fashion. Others make more incremental decisions—still local and opportunistic, but without a global "plan" in advance. In this section, we consider a problem that stresses our intuitive view of greedy algorithms still further.

The Problem

The problem is to compute a minimum-cost *arborescence* of a directed graph. This is essentially an analogue of the Minimum Spanning Tree Problem for directed, rather than undirected, graphs; we will see that the move to directed graphs introduces significant new complications. At the same time, the style of the algorithm has a strongly greedy flavor, since it still constructs a solution according to a local, myopic rule.

We begin with the basic definitions. Let $G = (V, E)$ be a directed graph in which we've distinguished one node $r \in V$ as a *root*. An *arborescence* (with respect to r) is essentially a directed spanning tree rooted at r. Specifically, it is a subgraph $T = (V, F)$ such that T is a spanning tree of G if we ignore the direction of edges; and there is a path in T from r to each other node $v \in V$ if we take the direction of edges into account. Figure 4.18 gives an example of two different arborescences in the same directed graph.

There is a useful equivalent way to characterize arborescences, and this is as follows.

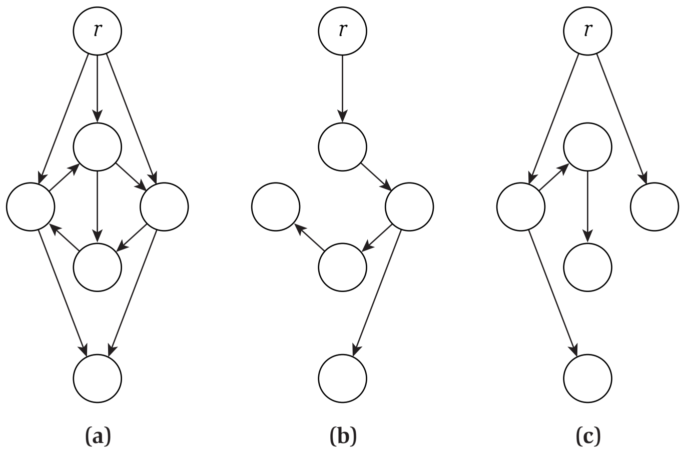

Figure 4.18 A directed graph can have many different arborescences. Parts (b) and (c) depict two different aborescences, both rooted at node r, for the graph in part (a).

(4.34) *A subgraph $T = (V, F)$ of G is an arborescence with respect to root r if and only if T has no cycles, and for each node $v \neq r$, there is exactly one edge in F that enters v.*

Proof. If T is an arborescence with root r, then indeed every other node v has exactly one edge entering it: this is simply the last edge on the unique r-v path.

Conversely, suppose T has no cycles, and each node $v \neq r$ has exactly one entering edge. In order to establish that T is an arborescence, we need only show that there is a directed path from r to each other node v. Here is how to construct such a path. We start at v and repeatedly follow edges in the backward direction. Since T has no cycles, we can never return to a node we've previously visited, and thus this process must terminate. But r is the only node without incoming edges, and so the process must in fact terminate by reaching r; the sequence of nodes thus visited yields a path (in the reverse direction) from r to v. ■

It is easy to see that, just as every connected graph has a spanning tree, a directed graph has an arborescence rooted at r provided that r can reach every node. Indeed, in this case, the edges in a breadth-first search tree rooted at r will form an arborescence.

(4.35) *A directed graph G has an arborescence rooted at r if and only if there is a directed path from r to each other node.*

The basic problem we consider here is the following. We are given a directed graph $G = (V, E)$, with a distinguished root node r and with a non-negative cost $c_e \geq 0$ on each edge, and we wish to compute an arborescence rooted at r of minimum total cost. (We will refer to this as an *optimal* arborescence.) We will assume throughout that G at least has an arborescence rooted at r; by (4.35), this can be easily checked at the outset.

Designing the Algorithm

Given the relationship between arborescences and trees, the minimum-cost arborescence problem certainly has a strong initial resemblance to the Minimum Spanning Tree Problem for undirected graphs. Thus it's natural to start by asking whether the ideas we developed for that problem can be carried over directly to this setting. For example, must the minimum-cost arborescence contain the cheapest edge in the whole graph? Can we safely delete the most expensive edge on a cycle, confident that it cannot be in the optimal arborescence?

Clearly the cheapest edge e in G will not belong to the optimal arborescence if e enters the root, since the arborescence we're seeking is not supposed to have any edges entering the root. But even if the cheapest edge in G belongs to *some* arborescence rooted at r, it need not belong to the optimal one, as the example of Figure 4.19 shows. Indeed, including the edge of cost 1 in Figure 4.19 would prevent us from including the edge of cost 2 out of the root r (since there can only be one entering edge per node); and this in turn would force us to incur an unacceptable cost of 10 when we included one of

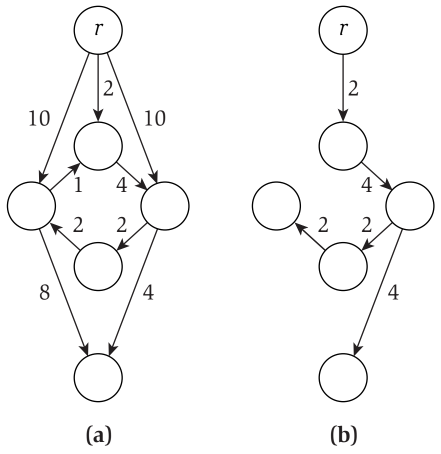

Figure 4.19 (a) A directed graph with costs on its edges, and (b) an optimal arborescence rooted at r for this graph.

the other edges out of r. This kind of argument never clouded our thinking in the Minimum Spanning Tree Problem, where it was always safe to plunge ahead and include the cheapest edge; it suggests that finding the optimal arborescence may be a significantly more complicated task. (It's worth noticing that the optimal arborescence in Figure 4.19 also includes the most expensive edge on a cycle; with a different construction, one can even cause the optimal arborescence to include the most expensive edge in the whole graph.)

Despite this, it is possible to design a greedy type of algorithm for this problem; it's just that our myopic rule for choosing edges has to be a little more sophisticated. First let's consider a little more carefully what goes wrong with the general strategy of including the cheapest edges. Here's a particular version of this strategy: for each node $v \neq r$, select the cheapest edge entering v (breaking ties arbitrarily), and let F^* be this set of $n-1$ edges. Now consider the subgraph (V, F^*). Since we know that the optimal arborescence needs to have exactly one edge entering each node $v \neq r$, and (V, F^*) represents the cheapest possible way of making these choices, we have the following fact.

(4.36) *If (V, F^*) is an arborescence, then it is a minimum-cost arborescence.*

So the difficulty is that (V, F^*) may not be an arborescence. In this case, (4.34) implies that (V, F^*) must contain a cycle C, which does not include the root. We now must decide how to proceed in this situation.

To make matters somewhat clearer, we begin with the following observation. Every arborescence contains exactly one edge entering each node $v \neq r$; so if we pick some node v and subtract a uniform quantity from the cost of every edge entering v, then the total cost of every arborescence changes by exactly the same amount. This means, essentially, that the actual cost of the cheapest edge entering v is not important; what matters is the cost of all other edges entering v *relative* to this. Thus let y_v denote the minimum cost of any edge entering v. For each edge $e = (u, v)$, with cost $c_e \geq 0$, we define its *modified cost* c'_e to be $c_e - y_v$. Note that since $c_e \geq y_v$, all the modified costs are still nonnegative. More crucially, our discussion motivates the following fact.

(4.37) *T is an optimal arborescence in G subject to costs $\{c_e\}$ if and only if it is an optimal arborescence subject to the modified costs $\{c'_e\}$.*

Proof. Consider an arbitrary arborescence T. The difference between its cost with costs $\{c_e\}$ and $\{c'_e\}$ is exactly $\sum_{v \neq r} y_v$—that is,

$$\sum_{e \in T} c_e - \sum_{e \in T} c'_e = \sum_{v \neq r} y_v.$$

This is because an arborescence has exactly one edge entering each node v in the sum. Since the difference between the two costs is independent of the choice of the arborescence T, we see that T has minimum cost subject to $\{c_e\}$ if and only if it has minimum cost subject to $\{c'_e\}$. ∎

We now consider the problem in terms of the costs $\{c'_e\}$. All the edges in our set F^* have cost 0 under these modified costs; and so if (V, F^*) contains a cycle C, we know that all edges in C have cost 0. This suggests that we can afford to use as many edges from C as we want (consistent with producing an arborescence), since including edges from C doesn't raise the cost.

Thus our algorithm continues as follows. We contract C into a single *supernode*, obtaining a smaller graph $G' = (V', E')$. Here, V' contains the nodes of $V - C$, plus a single node c^* representing C. We transform each edge $e \in E$ to an edge $e' \in E'$ by replacing each end of e that belongs to C with the new node c^*. This can result in G' having parallel edges (i.e., edges with the same ends), which is fine; however, we delete self-loops from E'—edges that have both ends equal to c^*. We recursively find an optimal arborescence in this smaller graph G', subject to the costs $\{c'_e\}$. The arborescence returned by this recursive call can be converted into an arborescence of G by including all but one edge on the cycle C.

In summary, here is the full algorithm.

```
For each node $v \neq r$
   Let $y_v$ be the minimum cost of an edge entering node $v$
   Modify the costs of all edges $e$ entering $v$ to $c'_e = c_e - y_v$
Choose one 0-cost edge entering each $v \neq r$, obtaining a set $F^*$
If $F^*$ forms an arborescence, then return it
Else there is a directed cycle $C \subseteq F^*$
   Contract $C$ to a single supernode, yielding a graph $G' = (V', E')$
   Recursively find an optimal arborescence $(V', F')$ in $G'$
               with costs $\{c'_e\}$
   Extend $(V', F')$ to an arborescence $(V, F)$ in $G$
               by adding all but one edge of $C$
```

Analyzing the Algorithm

It is easy to implement this algorithm so that it runs in polynomial time. But does it lead to an optimal arborescence? Before concluding that it does, we need to worry about the following point: not every arborescence in G corresponds to an arborescence in the contracted graph G'. Could we perhaps "miss" the true optimal arborescence in G by focusing on G'? What is true is the following.

The arborescences of G' are in one-to-one correspondence with arborescences of G *that have exactly one edge entering the cycle* C; and these corresponding arborescences have the same cost with respect to $\{c'_e\}$, since C consists of 0-cost edges. (We say that an edge $e = (u, v)$ enters C if v belongs to C but u does not.) So to prove that our algorithm finds an optimal arborescence in G, we must prove that G has an optimal arborescence with exactly one edge entering C. We do this now.

(4.38) *Let C be a cycle in G consisting of edges of cost 0, such that $r \notin C$. Then there is an optimal arborescence rooted at r that has exactly one edge entering C.*

Proof. Consider an optimal arborescence T in G. Since r has a path in T to every node, there is at least one edge of T that enters C. If T enters C exactly once, then we are done. Otherwise, suppose that T enters C more than once. We show how to modify it to obtain an arborescence of no greater cost that enters C exactly once.

Let $e = (a, b)$ be an edge entering C that lies on as short a path as possible from r; this means in particular that no edges on the path from r to a can enter C. We delete all edges of T that enter C, except for the edge e. We add in all edges of C except for the one edge that enters b, the head of edge e. Let T' denote the resulting subgraph of G.

We claim that T' is also an arborescence. This will establish the result, since the cost of T' is clearly no greater than that of T: the only edges of T' that do not also belong to T have cost 0. So why is T' an arborescence? Observe that T' has exactly one edge entering each node $v \neq r$, and no edge entering r. So T' has exactly $n - 1$ edges; hence if we can show there is an r-v path in T' for each v, then T' must be connected in an undirected sense, and hence a tree. Thus it would satisfy our initial definition of an arborescence.

So consider any node $v \neq r$; we must show there is an r-v path in T'. If $v \in C$, we can use the fact that the path in T from r to e has been preserved in the construction of T'; thus we can reach v by first reaching e and then following the edges of the cycle C. Now suppose that $v \notin C$, and let P denote the r-v path in T. If P did not touch C, then it still exists in T'. Otherwise, let w be the last node in $P \cap C$, and let P' be the subpath of P from w to v. Observe that all the edges in P' still exist in T'. We have already argued that w is reachable from r in T', since it belongs to C. Concatenating this path to w with the subpath P' gives us a path to v as well. ■

We can now put all the pieces together to argue that our algorithm is correct.

(4.39) *The algorithm finds an optimal arborescence rooted at r in G.*

Proof. The proof is by induction on the number of nodes in G. If the edges of F form an arborescence, then the algorithm returns an optimal arborescence by (4.36). Otherwise, we consider the problem with the modified costs $\{c'_e\}$, which is equivalent by (4.37). After contracting a 0-cost cycle C to obtain a smaller graph G', the algorithm produces an optimal arborescence in G' by the inductive hypothesis. Finally, by (4.38), there is an optimal arborescence in G that corresponds to the optimal arborescence computed for G'. ■

Solved Exercises

Solved Exercise 1

Suppose that three of your friends, inspired by repeated viewings of the horror-movie phenomenon *The Blair Witch Project*, have decided to hike the Appalachian Trail this summer. They want to hike as much as possible per day but, for obvious reasons, not after dark. On a map they've identified a large set of good *stopping points* for camping, and they're considering the following system for deciding when to stop for the day. Each time they come to a potential stopping point, they determine whether they can make it to the next one before nightfall. If they can make it, then they keep hiking; otherwise, they stop.

Despite many significant drawbacks, they claim this system does have one good feature. "Given that we're only hiking in the daylight," they claim, "it minimizes the number of camping stops we have to make."

Is this true? The proposed system is a greedy algorithm, and we wish to determine whether it minimizes the number of stops needed.

To make this question precise, let's make the following set of simplifying assumptions. We'll model the Appalachian Trail as a long line segment of length L, and assume that your friends can hike d miles per day (independent of terrain, weather conditions, and so forth). We'll assume that the potential stopping points are located at distances $x_1, x_2, \ldots, x_n$ from the start of the trail. We'll also assume (very generously) that your friends are always correct when they estimate whether they can make it to the next stopping point before nightfall.

We'll say that a set of stopping points is *valid* if the distance between each adjacent pair is at most d, the first is at distance at most d from the start of the trail, and the last is at distance at most d from the end of the trail. Thus a set of stopping points is valid if one could camp only at these places and still make it across the whole trail. We'll assume, naturally, that the full set of n stopping points is valid; otherwise, there would be no way to make it the whole way.

We can now state the question as follows. Is your friends' greedy algorithm—hiking as long as possible each day—*optimal*, in the sense that it finds a valid set whose size is as small as possible?

Solution Often a greedy algorithm looks correct when you first encounter it, so before succumbing too deeply to its intuitive appeal, it's useful to ask: why might it not work? What should we be worried about?

There's a natural concern with this algorithm: Might it not help to stop early on some day, so as to get better synchronized with camping opportunities on future days? But if you think about it, you start to wonder whether this could really happen. Could there really be an alternate solution that intentionally lags behind the greedy solution, and then puts on a burst of speed and passes the greedy solution? How could it pass it, given that the greedy solution travels as far as possible each day?

This last consideration starts to look like the outline of an argument based on the "staying ahead" principle from Section 4.1. Perhaps we can show that as long as the greedy camping strategy is ahead on a given day, no other solution can catch up and overtake it the next day.

We now turn this into a proof showing the algorithm is indeed optimal, identifying a natural sense in which the stopping points it chooses "stay ahead" of any other legal set of stopping points. Although we are following the style of proof from Section 4.1, it's worth noting an interesting contrast with the Interval Scheduling Problem: there we needed to prove that a greedy algorithm maximized a quantity of interest, whereas here we seek to minimize a certain quantity.

Let $R = \{x_{p_1}, \ldots, x_{p_k}\}$ denote the set of stopping points chosen by the greedy algorithm, and suppose by way of contradiction that there is a smaller valid set of stopping points; let's call this smaller set $S = \{x_{q_1}, \ldots, x_{q_m}\}$, with $m < k$.

To obtain a contradiction, we first show that the stopping point reached by the greedy algorithm on each day j is farther than the stopping point reached under the alternate solution. That is,

(4.40) *For each $j = 1, 2, \ldots, m$, we have $x_{p_j} \geq x_{q_j}$.*

Proof. We prove this by induction on j. The case $j = 1$ follows directly from the definition of the greedy algorithm: your friends travel as long as possible on the first day before stopping. Now let $j > 1$ and assume that the claim is true for all $i < j$. Then

$$x_{q_j} - x_{q_{j-1}} \leq d,$$

since S is a valid set of stopping points, and

$$x_{q_j} - x_{p_{j-1}} \leq x_{q_j} - x_{q_{j-1}}$$

since $x_{p_{j-1}} \geq x_{q_{j-1}}$ by the induction hypothesis. Combining these two inequalities, we have

$$x_{q_j} - x_{p_{j-1}} \leq d.$$

This means that your friends have the option of hiking all the way from $x_{p_{j-1}}$ to x_{q_j} in one day; and hence the location x_{p_j} at which they finally stop can only be farther along than x_{q_j}. (Note the similarity with the corresponding proof for the Interval Scheduling Problem: here too the greedy algorithm is staying ahead because, at each step, the choice made by the alternate solution is one of its valid options.) ■

Statement (4.40) implies in particular that $x_{q_m} \leq x_{p_m}$. Now, if $m < k$, then we must have $x_{p_m} < L - d$, for otherwise your friends would never have needed to stop at the location $x_{p_{m+1}}$. Combining these two inequalities, we have concluded that $x_{q_m} < L - d$; but this contradicts the assumption that S is a valid set of stopping points.

Consequently, we cannot have $m < k$, and so we have proved that the greedy algorithm produces a valid set of stopping points of minimum possible size.

Solved Exercise 2

Your friends are starting a security company that needs to obtain licenses for n different pieces of cryptographic software. Due to regulations, they can only obtain these licenses at the rate of at most one per month.

Each license is currently selling for a price of \$100. However, they are all becoming more expensive according to exponential growth curves: in particular, the cost of license j increases by a factor of $r_j > 1$ each month, where r_j is a given parameter. This means that if license j is purchased t months from now, it will cost $100 \cdot r_j^t$. We will assume that all the price growth rates are distinct; that is, $r_i \neq r_j$ for licenses $i \neq j$ (even though they start at the same price of \$100).

The question is: Given that the company can only buy at most one license a month, in which order should it buy the licenses so that the total amount of money it spends is as small as possible?

Give an algorithm that takes the n rates of price growth $r_1, r_2, \ldots, r_n$, and computes an order in which to buy the licenses so that the total amount of money spent is minimized. The running time of your algorithm should be polynomial in n.

Solution Two natural guesses for a good sequence would be to sort the r_i in decreasing order, or to sort them in increasing order. Faced with alternatives like this, it's perfectly reasonable to work out a small example and see if the example eliminates at least one of them. Here we could try $r_1 = 2$, $r_2 = 3$, and $r_3 = 4$. Buying the licenses in increasing order results in a total cost of

$$100(2 + 3^2 + 4^3) = 7{,}500,$$

while buying them in decreasing order results in a total cost of

$$100(4 + 3^2 + 2^3) = 2{,}100.$$

This tells us that increasing order is not the way to go. (On the other hand, it doesn't tell us immediately that decreasing order is the right answer, but our goal was just to eliminate one of the two options.)

Let's try proving that sorting the r_i in decreasing order in fact always gives the optimal solution. When a greedy algorithm works for problems like this, in which we put a set of things in an optimal order, we've seen in the text that it's often effective to try proving correctness using an exchange argument.

To do this here, let's suppose that there is an optimal solution O that differs from our solution S. (In other words, S consists of the licenses sorted in decreasing order.) So this optimal solution O must contain an inversion—that is, there must exist two neighboring months t and $t+1$ such that the price increase rate of the license bought in month t (let us denote it by r_t) is less than that bought in month $t+1$ (similarly, we use r_{t+1} to denote this). That is, we have $r_t < r_{t+1}$.

We claim that by exchanging these two purchases, we can strictly improve our optimal solution, which contradicts the assumption that O was optimal. Therefore if we succeed in showing this, we will successfully show that our algorithm is indeed the correct one.

Notice that if we swap these two purchases, the rest of the purchases are identically priced. In O, the amount paid during the two months involved in the swap is $100(r_t^t + r_{t+1}^{t+1})$. On the other hand, if we swapped these two purchases, we would pay $100(r_{t+1}^t + r_t^{t+1})$. Since the constant 100 is common to both expressions, we want to show that the second term is less than the first one. So we want to show that

$$\begin{aligned} &r_{t+1}^t + r_t^{t+1} < r_t^t + r_{t+1}^{t+1} \\ &r_t^{t+1} - r_t^t < r_{t+1}^{t+1} - r_{t+1}^t \\ &r_t^t(r_t - 1) < r_{t+1}^t(r_{t+1} - 1). \end{aligned}$$

But this last inequality is true simply because $r_i > 1$ for all i and since $r_t < r_{t+1}$.

This concludes the proof of correctness. The running time of the algorithm is $O(n \log n)$, since the sorting takes that much time and the rest (outputting) is linear. So the overall running time is $O(n \log n)$.

Note: It's interesting to note that things become much less straightforward if we vary this question even a little. Suppose that instead of buying licenses whose prices increase, you're trying to sell off equipment whose cost is depreciating. Item i depreciates at a factor of $r_i < 1$ per month, starting from \$100, so if you sell it t months from now you will receive $100 \cdot r_i^t$. (In other words, the exponential rates are now less than 1, instead of greater than 1.) If you can only sell one item per month, what is the optimal order in which to sell them? Here, it turns out that there are cases in which the optimal solution doesn't put the rates in either increasing or decreasing order (as in the input $\frac{3}{4}, \frac{1}{2}, \frac{1}{100}$).

Solved Exercise 3

Suppose you are given a connected graph G, with edge costs that you may assume are all distinct. G has n vertices and m edges. A particular edge e of G is specified. Give an algorithm with running time $O(m + n)$ to decide whether e is contained in a minimum spanning tree of G.

Solution From the text, we know of two rules by which we can conclude whether an edge e belongs to a minimum spanning tree: the Cut Property (4.17) says that e is in every minimum spanning tree when it is the cheapest edge crossing from some set S to the complement $V - S$; and the Cycle Property (4.20) says that e is in no minimum spanning tree if it is the most expensive edge on some cycle C. Let's see if we can make use of these two rules as part of an algorithm that solves this problem in linear time.

Both the Cut and Cycle Properties are essentially talking about how e relates to the set of edges that are *cheaper* than e. The Cut Property can be viewed as asking: Is there some set $S \subseteq V$ so that in order to get from S to $V - S$ without using e, we need to use an edge that is more expensive than e? And if we think about the cycle C in the statement of the Cycle Property, going the "long way" around C (avoiding e) can be viewed as an alternate route between the endpoints of e that only uses cheaper edges.

Putting these two observations together suggests that we should try proving the following statement.

(4.41) *Edge $e = (v, w)$ does not belong to a minimum spanning tree of G if and only if v and w can be joined by a path consisting entirely of edges that are cheaper than e.*

Proof. First suppose that P is a v-w path consisting entirely of edges cheaper than e. If we add e to P, we get a cycle on which e is the most expensive edge. Thus, by the Cycle Property, e does not belong to a minimum spanning tree of G.

On the other hand, suppose that v and w cannot be joined by a path consisting entirely of edges cheaper than e. We will now identify a set S for which e is the cheapest edge with one end in S and the other in $V - S$; if we can do this, the Cut Property will imply that e belongs to every minimum spanning tree. Our set S will be the set of all nodes that are reachable from v using a path consisting only of edges that are cheaper than e. By our assumption, we have $w \in V - S$. Also, by the definition of S, there cannot be an edge $f = (x, y)$ that is cheaper than e, and for which one end x lies in S and the other end y lies in $V - S$. Indeed, if there were such an edge f, then since the node x is reachable from v using only edges cheaper than e, the node y would be reachable as well. Hence e is the cheapest edge with one end in S and the other in $V - S$, as desired, and so we are done. ∎

Given this fact, our algorithm is now simply the following. We form a graph G' by deleting from G all edges of weight greater than c_e (as well as deleting e itself). We then use one of the connectivity algorithms from Chapter 3 to determine whether there is a path from v to w in G'. Statement (4.41) says that e belongs to a minimum spanning tree if and only if there is no such path.

The running time of this algorithm is $O(m + n)$ to build G', and $O(m + n)$ to test for a path from v to w.

Exercises

1. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

   Let G be an arbitrary connected, undirected graph with a distinct cost $c(e)$ on every edge e. Suppose e^* is the cheapest edge in G; that is, $c(e^*) < c(e)$ for every *edge $e \neq e^*$. Then there is a minimum spanning tree T of G that contains the edge e^*.*

2. For each of the following two statements, decide whether it is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

(a) Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph G, with edge costs that are all positive and distinct. Let T be a minimum spanning tree for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2, thereby creating a new instance of the problem with the same graph but different costs.

True or false? T must still be a minimum spanning tree for this new instance.

(b) Suppose we are given an instance of the Shortest s-t Path Problem on a directed graph G. We assume that all edge costs are positive and distinct. Let P be a minimum-cost s-t path for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2, thereby creating a new instance of the problem with the same graph but different costs.

True or false? P must still be a minimum-cost s-t path for this new instance.

3. You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package i has a weight w_i. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking. Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed.

Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Your proof should follow the type of analysis we used for the Interval Scheduling Problem: it should establish the optimality of this greedy packing algorithm by identifying a measure under which it "stays ahead" of all other solutions.

4. Some of your friends have gotten into the burgeoning field of *time-series data mining*, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges—what's being bought—are one source of data with a natural ordering in time. Given a long sequence S of such events, your friends want an efficient way to detect certain "patterns" in them—for example, they may want to know if the four events

```
buy Yahoo, buy eBay, buy Yahoo, buy Oracle
```

occur in this sequence S, in order but not necessarily consecutively.

They begin with a collection of possible *events* (e.g., the possible transactions) and a sequence S of n of these events. A given event may occur multiple times in S (e.g., `Yahoo` stock may be bought many times in a single sequence S). We will say that a sequence S' is a *subsequence* of S if there is a way to delete certain of the events from S so that the remaining events, in order, are equal to the sequence S'. So, for example, the sequence of four events above is a subsequence of the sequence

```
buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo,
buy Oracle
```

Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of S. So this is the problem they pose to you: Give an algorithm that takes two sequences of events—S' of length m and S of length n, each possibly containing an event more than once—and decides in time $O(m + n)$ whether S' is a subsequence of S.

5. Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal, using as few base stations as possible.

6. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of junior-high-school-age campers. One of his plans is the following mini-triathalon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming . . . and so on.)

Each contestant has a projected *swimming time* (the expected time it will take him or her to complete the 20 laps), a projected *biking time* (the expected time it will take him or her to complete the 10 miles of bicycling), and a projected *running time* (the time it will take him or her to complete the 3 miles of running). Your friend wants to decide on a *schedule* for the triathalon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathalon, assuming they each spend exactly their projected swimming, biking, and running times on the three parts. (Again, note that participants can bike and run simultaneously, but at most one person can be in the pool at any time.) What's the best order for sending people out, if one wants the whole competition to be over as early as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible.

7. The wildly popular Spanish-language search engine El Goog needs to do a serious amount of computation every time it recompiles its index. Fortunately, the company has at its disposal a single large supercomputer, together with an essentially unlimited supply of high-end PCs.

They've broken the overall computation into n distinct jobs, labeled $J_1, J_2, \ldots, J_n$, which can be performed completely independently of one another. Each job consists of two stages: first it needs to be *preprocessed* on the supercomputer, and then it needs to be *finished* on one of the PCs. Let's say that job J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC.

Since there are at least n PCs available on the premises, the finishing of the jobs can be performed fully in parallel—all the jobs can be processed at the same time. However, the supercomputer can only work on a single job at a time, so the system managers need to work out an order in which to feed the jobs to the supercomputer. As soon as the first job in order is done on the supercomputer, it can be handed off to a PC for finishing; at that point in time a second job can be fed to the supercomputer; when the second job is done on the supercomputer, it can proceed to a PC regardless of whether or not the first job is done (since the PCs work in parallel); and so on.

Let's say that a *schedule* is an ordering of the jobs for the supercomputer, and the *completion time* of the schedule is the earliest time at which all jobs will have finished processing on the PCs. This is an important quantity to minimize, since it determines how rapidly El Goog can generate a new index.

Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible.

8. Suppose you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

9. One of the basic motivations behind the Minimum Spanning Tree Problem is the goal of designing a spanning network for a set of nodes with minimum *total* cost. Here we explore another type of objective: designing a spanning network for which the *most expensive* edge is as cheap as possible.

Specifically, let $G = (V, E)$ be a connected graph with n vertices, m edges, and positive edge costs that you may assume are all distinct. Let $T = (V, E')$ be a spanning tree of G; we define the *bottleneck edge* of T to be the edge of T with the greatest cost.

A spanning tree T of G is a *minimum-bottleneck spanning tree* if there is no spanning tree T' of G with a cheaper bottleneck edge.

(a) Is every minimum-bottleneck tree of G a minimum spanning tree of G? Prove or give a counterexample.

(b) Is every minimum spanning tree of G a minimum-bottleneck tree of G? Prove or give a counterexample.

10. Let $G = (V, E)$ be an (undirected) graph with costs $c_e \geq 0$ on the edges $e \in E$. Assume you are given a minimum-cost spanning tree T in G. Now assume that a new edge is added to G, connecting two nodes $v, w \in V$ with cost c.

(a) Give an efficient algorithm to test if T remains the minimum-cost spanning tree with the new edge added to G (but not to the tree T). Make your algorithm run in time $O(|E|)$. Can you do it in $O(|V|)$ time? Please note any assumptions you make about what data structure is used to represent the tree T and the graph G.

(b) Suppose T is no longer the minimum-cost spanning tree. Give a linear-time algorithm (time $O(|E|)$) to update the tree T to the new minimum-cost spanning tree.

11. Suppose you are given a connected graph $G = (V, E)$, with a cost c_e on each edge e. In an earlier problem, we saw that when all edge costs are distinct, G has a unique minimum spanning tree. However, G may have many minimum spanning trees when the edge costs are not all distinct. Here we formulate the question: Can Kruskal's Algorithm be made to find all the minimum spanning trees of G?

Recall that Kruskal's Algorithm sorted the edges in order of increasing cost, then greedily processed edges one by one, adding an edge e as long as it did not form a cycle. When some edges have the same cost, the phrase "in order of increasing cost" has to be specified a little more carefully: we'll say that an ordering of the edges is *valid* if the corresponding sequence of edge costs is nondecreasing. We'll say that a *valid execution* of Kruskal's Algorithm is one that begins with a valid ordering of the edges of G.

For any graph G, and any minimum spanning tree T of G, is there a valid execution of Kruskal's Algorithm on G that produces T as output? Give a proof or a counterexample.

12. Suppose you have n video streams that need to be sent, one after another, over a communication link. Stream i consists of a total of b_i bits that need to be sent, at a constant rate, over a period of t_i seconds. You cannot send two streams at the same time, so you need to determine a *schedule* for the streams: an order in which to send them. Whichever order you choose, there cannot be any delays between the end of one stream and the start of the next. Suppose your schedule starts at time 0 (and therefore ends at time $\sum_{i=1}^{n} t_i$, whichever order you choose). We assume that all the values b_i and t_i are positive integers.

Now, because you're just one user, the link does not want you taking up too much bandwidth, so it imposes the following constraint, using a fixed parameter r:

($*$) For each natural number $t > 0$, the total number of bits you send over the time interval from 0 to t cannot exceed rt.

Note that this constraint is only imposed for time intervals that start at 0, *not* for time intervals that start at any other value.

We say that a schedule is *valid* if it satisfies the constraint ($*$) imposed by the link.

The Problem. Given a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, determine whether there exists a valid schedule.

Example. Suppose we have $n = 3$ streams, with

$$(b_1, t_1) = (2000, 1), \quad (b_2, t_2) = (6000, 2), \quad (b_3, t_3) = (2000, 1),$$

and suppose the link's parameter is $r = 5000$. Then the schedule that runs the streams in the order $1, 2, 3$, is valid, since the constraint $(*)$ is satisfied:

$t = 1$: the whole first stream has been sent, and $2000 < 5000 \cdot 1$
$t = 2$: half of the second stream has also been sent,
and $2000 + 3000 < 5000 \cdot 2$
Similar calculations hold for $t = 3$ and $t = 4$.

(a) Consider the following claim:

Claim: There exists a valid schedule if and only if each stream i satisfies $b_i \leq rt_i$.

Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

(b) Give an algorithm that takes a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, and determines whether there exists a valid schedule. The running time of your algorithm should be polynomial in n.

13. A small business—say, a photocopying service with a single large machine—faces the following scheduling problem. Each morning they get a set of jobs from customers. They want to do the jobs on their single machine in an order that keeps their customers happiest. Customer i's job will take t_i time to complete. Given a schedule (i.e., an ordering of the jobs), let C_i denote the finishing time of job i. For example, if job j is the first to be done, we would have $C_j = t_j$; and if job j is done right after job i, we would have $C_j = C_i + t_j$. Each customer i also has a given weight w_i that represents his or her importance to the business. The happiness of customer i is expected to be dependent on the finishing time of i's job. So the company decides that they want to order the jobs to minimize the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$.

Design an efficient algorithm to solve this problem. That is, you are given a set of n jobs with a processing time t_i and a weight w_i for each job. You want to order the jobs so as to minimize the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$.

Example. Suppose there are two jobs: the first takes time $t_1 = 1$ and has weight $w_1 = 10$, while the second job takes time $t_2 = 3$ and has weight

$w_2 = 2$. Then doing job 1 first would yield a weighted completion time of $10 \cdot 1 + 2 \cdot 4 = 18$, while doing the second job first would yield the larger weighted completion time of $10 \cdot 4 + 2 \cdot 3 = 46$.

14. You're working with a group of security consultants who are helping to monitor a large computer system. There's particular interest in keeping track of processes that are labeled "sensitive." Each such process has a designated start time and finish time, and it runs continuously between these times; the consultants have a list of the planned start and finish times of all sensitive processes that will be run that day.

As a simple first step, they've written a program called `status_check` that, when invoked, runs for a few seconds and records various pieces of logging information about all the sensitive processes running on the system at that moment. (We'll model each invocation of `status_check` as lasting for only this single point in time.) What they'd like to do is to run `status_check` as few times as possible during the day, but enough that for each sensitive process P, `status_check` is invoked at least once during the execution of process P.

(a) Give an efficient algorithm that, given the start and finish times of all the sensitive processes, finds as small a set of times as possible at which to invoke `status_check`, subject to the requirement that `status_check` is invoked at least once during each sensitive process P.

(b) While you were designing your algorithm, the security consultants were engaging in a little back-of-the-envelope reasoning. "Suppose we can find a set of k sensitive processes with the property that no two are ever running at the same time. Then clearly your algorithm will need to invoke `status_check` at least k times: no one invocation of `status_check` can handle more than one of these processes."

This is true, of course, and after some further discussion, you all begin wondering whether something stronger is true as well, a kind of converse to the above argument. Suppose that k^* is the largest value of k such that one can find a set of k sensitive processes with no two ever running at the same time. Is it the case that there must be a set of k^* times at which you can run `status_check` so that some invocation occurs during the execution of each sensitive process? (In other words, the kind of argument in the previous paragraph is really the only thing forcing you to need a lot of invocations of `status_check`.) Decide whether you think this claim is true or false, and give a proof or a counterexample.

15. The manager of a large student union on campus comes to you with the following problem. She's in charge of a group of n students, each of whom is scheduled to work one *shift* during the week. There are different jobs associated with these shifts (tending the main desk, helping with package delivery, rebooting cranky information kiosks, etc.), but we can view each shift as a single contiguous interval of time. There can be multiple shifts going on at once.

She's trying to choose a subset of these n students to form a *supervising committee* that she can meet with once a week. She considers such a committee to be *complete* if, for every student not on the committee, that student's shift overlaps (at least partially) the shift of some student who is on the committee. In this way, each student's performance can be observed by at least one person who's serving on the committee.

Give an efficient algorithm that takes the schedule of n shifts and produces a complete supervising committee containing as few students as possible.

Example. Suppose $n = 3$, and the shifts are

Monday 4 P.M.–Monday 8 P.M.,
Monday 6 P.M.–Monday 10 P.M.,
Monday 9 P.M.–Monday 11 P.M..

Then the smallest complete supervising committee would consist of just the second student, since the second shift overlaps both the first and the third.

16. Some security consultants working in the financial domain are currently advising a client who is investigating a potential money-laundering scheme. The investigation thus far has indicated that n suspicious transactions took place in recent days, each involving money transferred into a single account. Unfortunately, the sketchy nature of the evidence to date means that they don't know the identity of the account, the amounts of the transactions, or the exact times at which the transactions took place. What they do have is an *approximate time-stamp* for each transaction; the evidence indicates that transaction i took place at time $t_i \pm e_i$, for some "margin of error" e_i. (In other words, it took place sometime between $t_i - e_i$ and $t_i + e_i$.) Note that different transactions may have different margins of error.

In the last day or so, they've come across a bank account that (for other reasons we don't need to go into here) they suspect might be the one involved in the crime. There are n recent *events* involving the account, which took place at times $x_1, x_2, \ldots, x_n$. To see whether it's plausible that this really is the account they're looking for, they're wondering whether it's possible to associate each of the account's n events with a distinct one of the n suspicious transactions in such a way that, if the account event at time x_i is associated with the suspicious transaction that occurred approximately at time t_j, then $|t_j - x_i| \le e_j$. (In other words, they want to know if the activity on the account lines up with the suspicious transactions to within the margin of error; the tricky part here is that they don't know which account event to associate with which suspicious transaction.)

Give an efficient algorithm that takes the given data and decides whether such an association exists. If possible, you should make the running time be at most $O(n^2)$.

17. Consider the following variation on the Interval Scheduling Problem. You have a processor that can operate 24 hours a day, every day. People submit requests to run *daily jobs* on the processor. Each such job comes with a *start time* and an *end time*; if the job is accepted to run on the processor, it must run continuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the Interval Scheduling Problem.)

Given a list of n such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n. You may assume for simplicity that no two jobs have the same start or end times.

Example. Consider the following four jobs, specified by *(start-time, end-time)* pairs.

(6 P.M., 6 A.M.), (9 P.M., 4 A.M.), (3 A.M., 2 P.M.), (1 P.M., 7 P.M.).

The optimal solution would be to pick the two jobs (9 P.M., 4 A.M.) and (1 P.M., 7 P.M.), which can be scheduled without overlapping.

18. Your friends are planning an expedition to a small town deep in the Canadian north next winter break. They've researched all the travel options and have drawn up a directed graph whose nodes represent intermediate destinations and edges represent the roads between them.

In the course of this, they've also learned that extreme weather causes roads in this part of the world to become quite slow in the winter and may cause large travel delays. They've found an excellent travel Web site that can accurately predict how fast they'll be able to travel along the roads; however, the speed of travel depends on the time of year. More precisely, the Web site answers queries of the following form: given an edge $e = (v, w)$ connecting two sites v and w, and given a proposed starting time t from location v, the site will return a value $f_e(t)$, the predicted arrival time at w. The Web site guarantees that $f_e(t) \geq t$ for all edges e and all times t (you can't travel backward in time), and that $f_e(t)$ is a monotone increasing function of t (that is, you do not arrive earlier by starting later). Other than that, the functions $f_e(t)$ may be arbitrary. For example, in areas where the travel time does not vary with the season, we would have $f_e(t) = t + \ell_e$, where ℓ_e is the time needed to travel from the beginning to the end of edge e.

Your friends want to use the Web site to determine the fastest way to travel through the directed graph from their starting point to their intended destination. (You should assume that they start at time 0, and that all predictions made by the Web site are completely correct.) Give a polynomial-time algorithm to do this, where we treat a single query to the Web site (based on a specific edge e and a time t) as taking a single computational step.

19. A group of network designers at the communications company CluNet find themselves facing the following problem. They have a connected graph $G = (V, E)$, in which the nodes represent sites that want to communicate. Each edge e is a communication link, with a given available bandwidth b_e.

For each pair of nodes $u, v \in V$, they want to select a single u-v path P on which this pair will communicate. The *bottleneck rate* $b(P)$ of this path P is the minimum bandwidth of any edge it contains; that is, $b(P) = \min_{e \in P} b_e$. The *best achievable bottleneck rate* for the pair u, v in G is simply the maximum, over all u-v paths P in G, of the value $b(P)$.

It's getting to be very complicated to keep track of a path for each pair of nodes, and so one of the network designers makes a bold suggestion: Maybe one can find a spanning tree T of G so that for *every* pair of nodes u, v, the unique u-v path in the tree actually attains the best achievable bottleneck rate for u, v in G. (In other words, even if you could choose any u-v path in the whole graph, you couldn't do better than the u-v path in T.)

This idea is roundly heckled in the offices of CluNet for a few days, and there's a natural reason for the skepticism: each pair of nodes might want a very different-looking path to maximize its bottleneck rate; why should there be a single tree that simultaneously makes everybody happy? But after some failed attempts to rule out the idea, people begin to suspect it could be possible.

Show that such a tree exists, and give an efficient algorithm to find one. That is, give an algorithm constructing a spanning tree T in which, for each $u, v \in V$, the bottleneck rate of the u-v path in T is equal to the best achievable bottleneck rate for the pair u, v in G.

20. Every September, somewhere in a far-away mountainous part of the world, the county highway crews get together and decide which roads to keep clear through the coming winter. There are n towns in this county, and the road system can be viewed as a (connected) graph $G = (V, E)$ on this set of towns, each edge representing a road joining two of them. In the winter, people are high enough up in the mountains that they stop worrying about the *length* of roads and start worrying about their *altitude*—this is really what determines how difficult the trip will be.

So each road—each edge e in the graph—is annotated with a number a_e that gives the altitude of the highest point on the road. We'll assume that no two edges have exactly the same altitude value a_e. The *height* of a path P in the graph is then the maximum of a_e over all edges e on P. Finally, a path between towns i and j is declared to be *winter-optimal* if it achieves the minimum possible height over all paths from i to j.

The highway crews are going to select a set $E' \subseteq E$ of the roads to keep clear through the winter; the rest will be left unmaintained and kept off limits to travelers. They all agree that whichever subset of roads E' they decide to keep clear, it should have the property that (V, E') is a connected subgraph; and more strongly, for every pair of towns i and j, the height of the winter-optimal path in (V, E') should be no greater than it is in the full graph $G = (V, E)$. We'll say that (V, E') is a *minimum-altitude connected subgraph* if it has this property.

Given that they're going to maintain this key property, however, they otherwise want to keep as few roads clear as possible. One year, they hit upon the following conjecture:

The minimum spanning tree of G, with respect to the edge weights a_e, is a minimum-altitude connected subgraph.

(In an earlier problem, we claimed that there is a unique minimum spanning tree when the edge weights are distinct. Thus, thanks to the assumption that all a_e are distinct, it is okay for us to speak of *the* minimum spanning tree.)

Initially, this conjecture is somewhat counterintuitive, since the minimum spanning tree is trying to minimize the *sum* of the values a_e, while the goal of minimizing altitude seems to be asking for a fairly different thing. But lacking an argument to the contrary, they begin considering an even bolder second conjecture:

A subgraph (V, E') is a minimum-altitude connected subgraph if and only if it contains the edges of the minimum spanning tree.

Note that this second conjecture would immediately imply the first one, since a minimum spanning tree contains its own edges.

So here's the question.

(a) Is the first conjecture true, for all choices of G and distinct altitudes a_e? Give a proof or a counterexample with explanation.

(b) Is the second conjecture true, for all choices of G and distinct altitudes a_e? Give a proof or a counterexample with explanation.

21. Let us say that a graph $G = (V, E)$ is a *near-tree* if it is connected and has at most $n + 8$ edges, where $n = |V|$. Give an algorithm with running time $O(n)$ that takes a near-tree G with costs on its edges, and returns a minimum spanning tree of G. You may assume that all the edge costs are distinct.

22. Consider the Minimum Spanning Tree Problem on an undirected graph $G = (V, E)$, with a cost $c_e \geq 0$ on each edge, where the costs may not all be different. If the costs are not all distinct, there can in general be many distinct minimum-cost solutions. Suppose we are given a spanning tree $T \subseteq E$ with the guarantee that for every $e \in T$, e belongs to *some* minimum-cost spanning tree in G. Can we conclude that T itself must be a minimum-cost spanning tree in G? Give a proof or a counterexample with explanation.

23. Recall the problem of computing a minimum-cost arborescence in a directed graph $G = (V, E)$, with a cost $c_e \geq 0$ on each edge. Here we will consider the case in which G is a directed acyclic graph—that is, it contains no directed cycles.

As in general directed graphs, there can be many distinct minimum-cost solutions. Suppose we are given a directed acyclic graph $G = (V, E)$, and an arborescence $A \subseteq E$ with the guarantee that for every $e \in A$, e belongs to *some* minimum-cost arborescence in G. Can we conclude that A itself must be a minimum-cost arborescence in G? Give a proof or a counterexample with explanation.

24. Timing circuits are a crucial component of VLSI chips. Here's a simple model of such a timing circuit. Consider a complete balanced binary tree with n leaves, where n is a power of two. Each edge e of the tree has an associated length ℓ_e, which is a positive number. The *distance* from the root to a given leaf is the sum of the lengths of all the edges on the path from the root to the leaf.

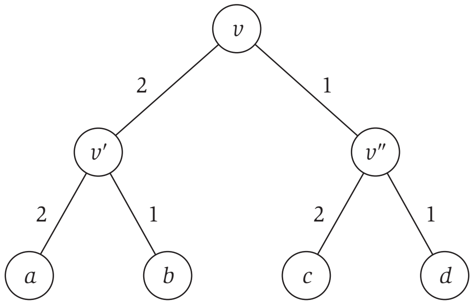

Figure 4.20 An instance of the zero-skew problem, described in Exercise 23.

The root generates a *clock signal* which is propagated along the edges to the leaves. We'll assume that the time it takes for the signal to reach a given leaf is proportional to the distance from the root to the leaf.

Now, if all leaves do not have the same distance from the root, then the signal will not reach the leaves at the same time, and this is a big problem. We want the leaves to be completely synchronized, and all to receive the signal at the same time. To make this happen, we will have to *increase* the lengths of certain edges, so that all root-to-leaf paths have the same length (we're not able to shrink edge lengths). If we achieve this, then the tree (with its new edge lengths) will be said to have *zero skew*. Our goal is to achieve zero skew in a way that keeps the sum of all the edge lengths as small as possible.

Give an algorithm that increases the lengths of certain edges so that the resulting tree has zero skew and the total edge length is as small as possible.

Example. Consider the tree in Figure 4.20, in which letters name the nodes and numbers indicate the edge lengths.

The unique optimal solution for this instance would be to take the three length-1 edges and increase each of their lengths to 2. The resulting tree has zero skew, and the total edge length is 12, the smallest possible.

25. Suppose we are given a set of points $P = \{p_1, p_2, \ldots, p_n\}$, together with a distance function d on the set P; d is simply a function on pairs of points in P with the properties that $d(p_i, p_j) = d(p_j, p_i) > 0$ if $i \neq j$, and that $d(p_i, p_i) = 0$ for each i.

We define a *hierarchical metric* on P to be any distance function τ that can be constructed as follows. We build a rooted tree T with n leaves, and we associate with each node v of T (both leaves and internal nodes) a *height* h_v. These heights must satisfy the properties that $h(v) = 0$ for each leaf v, and if u is the parent of v in T, then $h(u) \geq h(v)$. We place each point in P at a distinct leaf in T. Now, for any pair of points p_i and p_j, their distance $\tau(p_i, p_j)$ is defined as follows. We determine the least common ancestor v in T of the leaves containing p_i and p_j, and define $\tau(p_i, p_j) = h_v$.

We say that a hierarchical metric τ is *consistent* with our distance function d if, for all pairs i, j, we have $\tau(p_i, p_j) \leq d(p_i, p_j)$.

Give a polynomial-time algorithm that takes the distance function d and produces a hierarchical metric τ with the following properties.

(i) τ is consistent with d, and

(ii) if τ' is any other hierarchical metric consistent with d, then $\tau'(p_i, p_j) \leq \tau(p_i, p_j)$ for each pair of points p_i and p_j.

26. One of the first things you learn in calculus is how to minimize a differentiable function such as $y = ax^2 + bx + c$, where $a > 0$. The Minimum Spanning Tree Problem, on the other hand, is a minimization problem of a very different flavor: there are now just a finite number of possibilities for how the minimum might be achieved—rather than a continuum of possibilities—and we are interested in how to perform the computation without having to exhaust this (huge) finite number of possibilities.

One can ask what happens when these two minimization issues are brought together, and the following question is an example of this. Suppose we have a connected graph $G = (V, E)$. Each edge e now has a *time-varying edge cost* given by a function $f_e : \mathbf{R} \rightarrow \mathbf{R}$. Thus, at time t, it has cost $f_e(t)$. We'll assume that all these functions are positive over their entire range. Observe that the set of edges constituting the minimum spanning tree of G may change over time. Also, of course, the cost of the minimum spanning tree of G becomes a function of the time t; we'll denote this function $c_G(t)$. A natural problem then becomes: find a value of t at which $c_G(t)$ is minimized.

Suppose each function f_e is a polynomial of degree 2: $f_e(t) = a_e t^2 + b_e t + c_e$, where $a_e > 0$. Give an algorithm that takes the graph G and the values $\{(a_e, b_e, c_e) : e \in E\}$ and returns a value of the time t at which the minimum spanning tree has minimum cost. Your algorithm should run in time polynomial in the number of nodes and edges of the graph G. You may assume that arithmetic operations on the numbers $\{(a_e, b_e, c_e)\}$ can be done in constant time per operation.

27. In trying to understand the combinatorial structure of spanning trees, we can consider the space of *all* possible spanning trees of a given graph and study the properties of this space. This is a strategy that has been applied to many similar problems as well.

Here is one way to do this. Let G be a connected graph, and T and T' two different spanning trees of G. We say that T and T' are *neighbors* if T contains exactly one edge that is not in T', and T' contains exactly one edge that is not in T.

Now, from any graph G, we can build a (large) graph $\mathcal{H}$ as follows. The nodes of $\mathcal{H}$ are the spanning trees of G, and there is an edge between two nodes of $\mathcal{H}$ if the corresponding spanning trees are neighbors.

Is it true that, for any connected graph G, the resulting graph $\mathcal{H}$ is connected? Give a proof that $\mathcal{H}$ is always connected, or provide an example (with explanation) of a connected graph G for which $\mathcal{H}$ is not connected.

28. Suppose you're a consultant for the networking company CluNet, and they have the following problem. The network that they're currently working on is modeled by a connected graph $G = (V, E)$ with n nodes. Each edge e is a fiber-optic cable that is owned by one of two companies—creatively named X and Y—and leased to CluNet.

Their plan is to choose a spanning tree T of G and upgrade the links corresponding to the edges of T. Their business relations people have already concluded an agreement with companies X and Y stipulating a number k so that in the tree T that is chosen, k of the edges will be owned by X and $n - k - 1$ of the edges will be owned by Y.

CluNet management now faces the following problem. It is not at all clear to them whether there even *exists* a spanning tree T meeting these conditions, or how to find one if it exists. So this is the problem they put to you: Give a polynomial-time algorithm that takes G, with each edge labeled X or Y, and either (i) returns a spanning tree with exactly k edges labeled X, or (ii) reports correctly that no such tree exists.

29. Given a list of n natural numbers $d_1, d_2, \ldots, d_n$, show how to decide in polynomial time whether there exists an undirected graph $G = (V, E)$ whose node degrees are precisely the numbers $d_1, d_2, \ldots, d_n$. (That is, if $V = \{v_1, v_2, \ldots, v_n\}$, then the degree of v_i should be exactly d_i.) G should not contain multiple edges between the same pair of nodes, or "loop" edges with both endpoints equal to the same node.

30. Let $G = (V, E)$ be a graph with n nodes in which each pair of nodes is joined by an edge. There is a positive weight w_{ij} on each edge (i, j); and we will assume these weights satisfy the *triangle inequality* $w_{ik} \le w_{ij} + w_{jk}$. For a subset $V' \subseteq V$, we will use $G[V']$ to denote the subgraph (with edge weights) induced on the nodes in V'.

We are given a set $X \subseteq V$ of k *terminals* that must be connected by edges. We say that a *Steiner tree* on X is a set Z so that $X \subseteq Z \subseteq V$, together with a spanning subtree T of $G[Z]$. The *weight* of the Steiner tree is the weight of the tree T.

Show that the problem of finding a minimum-weight Steiner tree on X can be solved in time $O(n^{O(k)})$.

31. Let's go back to the original motivation for the Minimum Spanning Tree Problem. We are given a connected, undirected graph $G = (V, E)$ with positive edge lengths $\{\ell_e\}$, and we want to find a spanning subgraph of it. Now suppose we are willing to settle for a subgraph $H = (V, F)$ that is "denser" than a tree, and we are interested in guaranteeing that, for each pair of vertices $u, v \in V$, the length of the shortest u-v path in H is not much longer than the length of the shortest u-v path in G. By the *length* of a path P here, we mean the sum of ℓ_e over all edges e in P.

Here's a variant of Kruskal's Algorithm designed to produce such a subgraph.

- First we sort all the edges in order of increasing length. (You may assume all edge lengths are distinct.)
- We then construct a subgraph $H = (V, F)$ by considering each edge in order.
- When we come to edge $e = (u, v)$, we add e to the subgraph H if there is currently no u-v path in H. (This is what Kruskal's Algorithm would do as well.) On the other hand, if there is a u-v path in H, we let d_{uv} denote the length of the shortest such path; again, length is with respect to the values $\{\ell_e\}$. We add e to H if $3\ell_e < d_{uv}$.

In other words, we add an edge even when u and v are already in the same connected component, provided that the addition of the edge reduces their shortest-path distance by a sufficient amount.

Let $H = (V, F)$ be the subgraph of G returned by the algorithm.

(a) Prove that for every pair of nodes $u, v \in V$, the length of the shortest u-v path in H is at most three times the length of the shortest u-v path in G.

(b) Despite its ability to approximately preserve shortest-path distances, the subgraph H produced by the algorithm cannot be too dense. Let $f(n)$ denote the maximum number of edges that can possibly be produced as the output of this algorithm, over all n-node input graphs with edge lengths. Prove that

$$\lim_{n\to\infty} \frac{f(n)}{n^2} = 0.$$

32. Consider a directed graph $G = (V, E)$ with a root $r \in V$ and nonnegative costs on the edges. In this problem we consider variants of the minimum-cost arborescence algorithm.

(a) The algorithm discussed in Section 4.9 works as follows. We modify the costs, consider the subgraph of zero-cost edges, look for a directed cycle in this subgraph, and contract it (if one exists). Argue briefly that instead of looking for cycles, we can instead identify and contract strong components of this subgraph.

(b) In the course of the algorithm, we defined y_v to be the minimum cost of an edge entering v, and we modified the costs of all edges e entering node v to be $c'_e = c_e - y_v$. Suppose we instead use the following modified cost: $c''_e = \max(0, c_e - 2y_v)$. This new change is likely to turn more edges to 0 cost. Suppose now we find an arborescence T of 0 cost. Prove that this T has cost at most twice the cost of the minimum-cost arborescence in the original graph.

(c) Assume you do not find an arborescence of 0 cost. Contract all 0-cost strong components and recursively apply the same procedure on the resulting graph until an arborescence is found. Prove that this T has cost at most twice the cost of the minimum-cost arborescence in the original graph.

33. Suppose you are given a directed graph $G = (V, E)$ in which each edge has a cost of either 0 or 1. Also suppose that G has a node r such that there is a path from r to every other node in G. You are also given an integer k. Give a polynomial-time algorithm that either constructs an arborescence rooted at r of cost *exactly* k, or reports (correctly) that no such arborescence exists.

Notes and Further Reading

Due to their conceptual cleanness and intuitive appeal, greedy algorithms have a long history and many applications throughout computer science. In this chapter we focused on cases in which greedy algorithms find the optimal solution. Greedy algorithms are also often used as simple heuristics even when they are not guaranteed to find the optimal solution. In Chapter 11 we will discuss greedy algorithms that find near-optimal approximate solutions.

As discussed in Chapter 1, Interval Scheduling can be viewed as a special case of the Independent Set Problem on a graph that represents the overlaps among a collection of intervals. Graphs arising this way are called *interval graphs*, and they have been extensively studied; see, for example, the book by Golumbic (1980). Not just Independent Set but many hard computational problems become much more tractable when restricted to the special case of interval graphs.

Interval Scheduling and the problem of scheduling to minimize the maximum lateness are two of a range of basic scheduling problems for which a simple greedy algorithm can be shown to produce an optimal solution. A wealth of related problems can be found in the survey by Lawler, Lenstra, Rinnooy Kan, and Shmoys (1993).

The optimal algorithm for caching and its analysis are due to Belady (1966). As we mentioned in the text, under real operating conditions caching algorithms must make eviction decisions in real time without knowledge of future requests. We will discuss such caching strategies in Chapter 13.

The algorithm for shortest paths in a graph with nonnegative edge lengths is due to Dijkstra (1959). Surveys of approaches to the Minimum Spanning Tree Problem, together with historical background, can be found in the reviews by Graham and Hell (1985) and Nesetril (1997).

The single-link algorithm is one of the most widely used approaches to the general problem of clustering; the books by Anderberg (1973), Duda, Hart, and Stork (2001), and Jain and Dubes (1981) survey a variety of clustering techniques.

The algorithm for optimal prefix codes is due to Huffman (1952); the earlier approaches mentioned in the text appear in the books by Fano (1949) and Shannon and Weaver (1949). General overviews of the area of data compression can be found in the book by Bell, Cleary, and Witten (1990) and the survey by Lelewer and Hirschberg (1987). More generally, this topic belongs to the area of *information theory*, which is concerned with the representation and encoding of digital information. One of the founding works in this field is the book by Shannon and Weaver (1949), and the more recent textbook by Cover and Thomas (1991) provides detailed coverage of the subject.

The algorithm for finding minimum-cost arborescences is generally credited to Chu and Liu (1965) and to Edmonds (1967) independently. As discussed in the chapter, this multi-phase approach stretches our notion of what constitutes a greedy algorithm. It is also important from the perspective of linear programming, since in that context it can be viewed as a fundamental application of the *pricing method*, or the *primal-dual* technique, for designing algorithms. The book by Nemhauser and Wolsey (1988) develops these connections to linear programming. We will discuss this method in Chapter 11 in the context of approximation algorithms.

More generally, as we discussed at the outset of the chapter, it is hard to find a precise definition of what constitutes a greedy algorithm. In the search for such a definition, it is not even clear that one can apply the analogue of U.S. Supreme Court Justice Potter Stewart's famous test for obscenity—"I know it when I see it"—since one finds disagreements within the research community on what constitutes the boundary, even intuitively, between greedy and nongreedy algorithms. There has been research aimed at formalizing classes of greedy algorithms: the theory of *matroids* is one very influential example (Edmonds 1971; Lawler 2001); and the paper of Borodin, Nielsen, and Rackoff (2002) formalizes notions of greedy and "greedy-type" algorithms, as well as providing a comparison to other formal work on this question.

Notes on the Exercises Exercise 24 is based on results of M. Edahiro, T. Chao, Y. Hsu, J. Ho, K. Boese, and A. Kahng; Exercise 31 is based on a result of Ingo Althofer, Gautam Das, David Dobkin, and Deborah Joseph.