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.

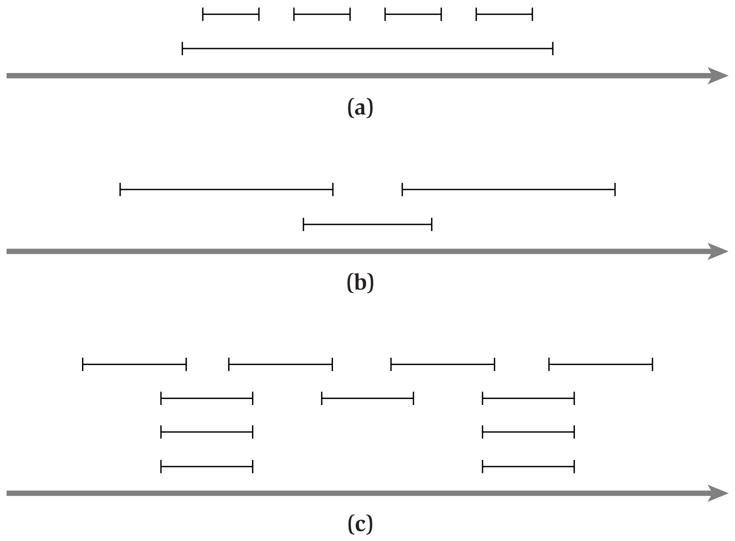

Figure 4.1 Some instances of the Interval Scheduling Problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the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

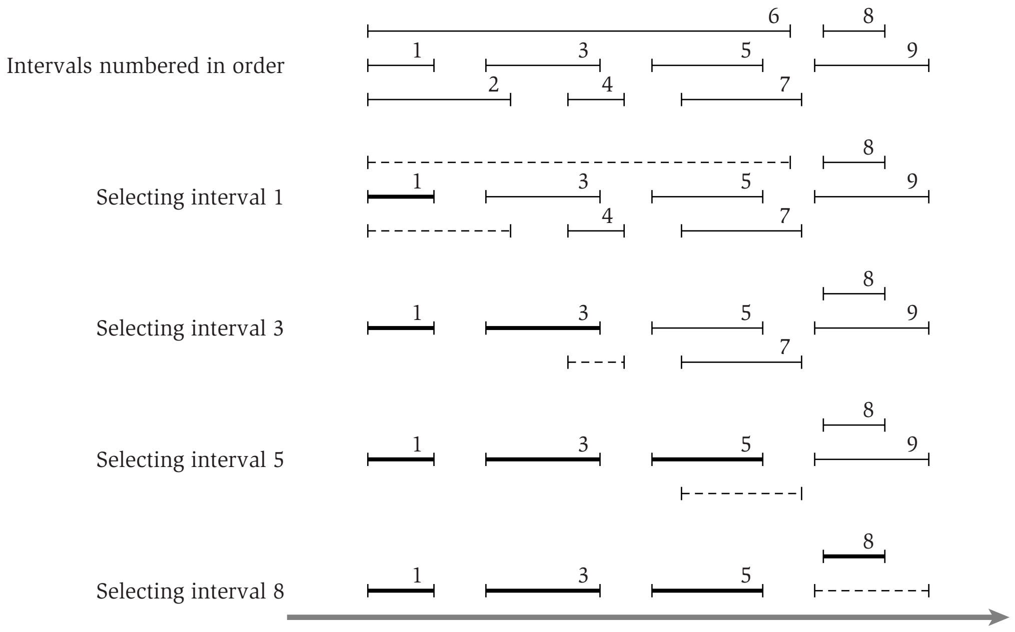

Figure 4.2 Sample run of the Interval Scheduling Algorithm. At each step the selected intervals are darker lines, and the intervals deleted at the corresponding step are indicated with dashed lines.

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.

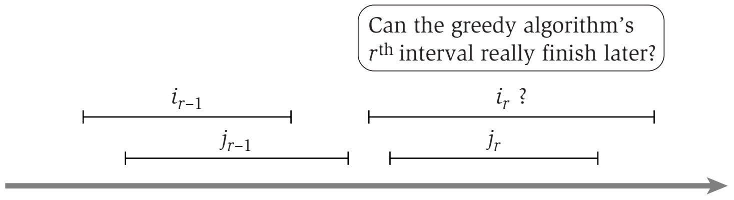
Can the greedy algorithm's r^{th} interval really finish later?
i_{r-1}
i_r ?
j_{r-1}
j_r

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 $f(i_1) \leq f(j_1)$. 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 $r \geq 1$, 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 $r \leq k$ we have $f(i_r) \leq f(j_r)$.*

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 $f(i_{r-1}) \leq f(j_{r-1})$. 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 $f(j_{r-1}) \leq s(j_r)$. Combining this with the induction hypothesis $f(i_{r-1}) \leq f(j_{r-1})$, we get $f(i_{r-1}) \leq s(j_r)$. 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 $f(i_r) \leq f(j_r)$. 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 *online* 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.

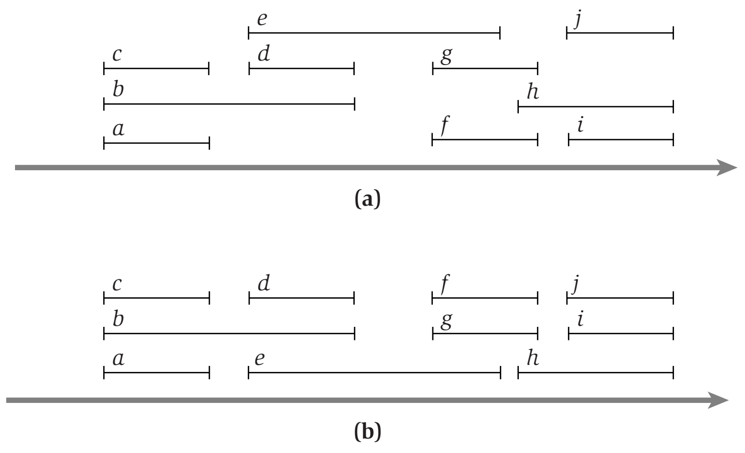

Figure 4.4 (a) An instance of the Interval Partitioning Problem with ten intervals (*a* through *j*). (b) A solution in which all intervals are scheduled using three resources: each row represents a set of intervals that can all be scheduled on a single resource.

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

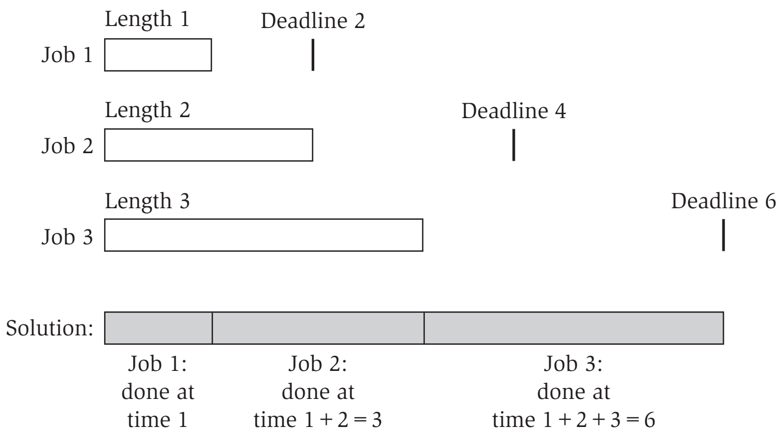

Figure 4.5 A sample instance of scheduling to minimize lateness.

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

$$d_1 \le \ldots \le d_n.$$

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 $d_1 \le \ldots \le d_n$
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.

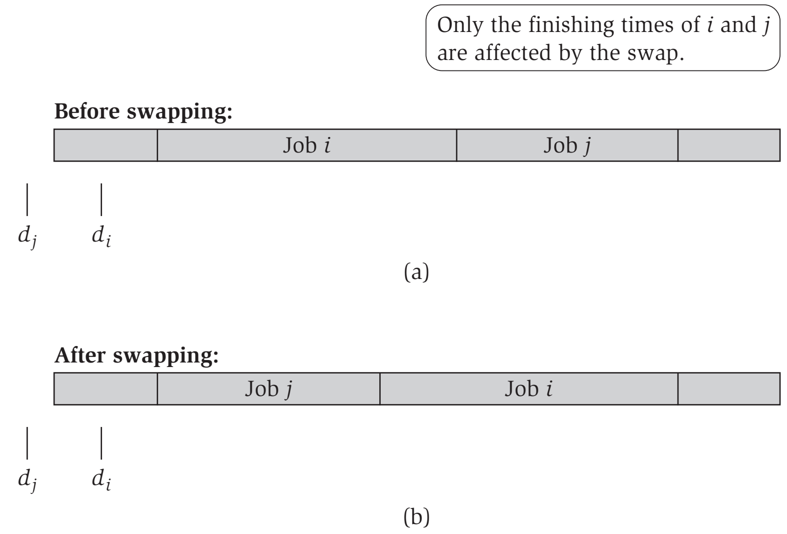

Figure 4.6 The effect of swapping two consecutive, inverted jobs.

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, "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.