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.

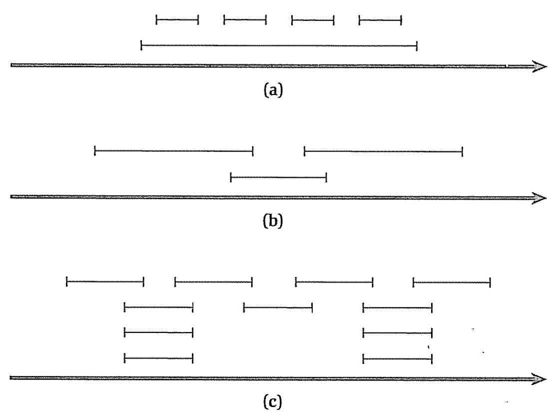

Figure 4.1 Some instances of the Interval Scheduling Problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the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

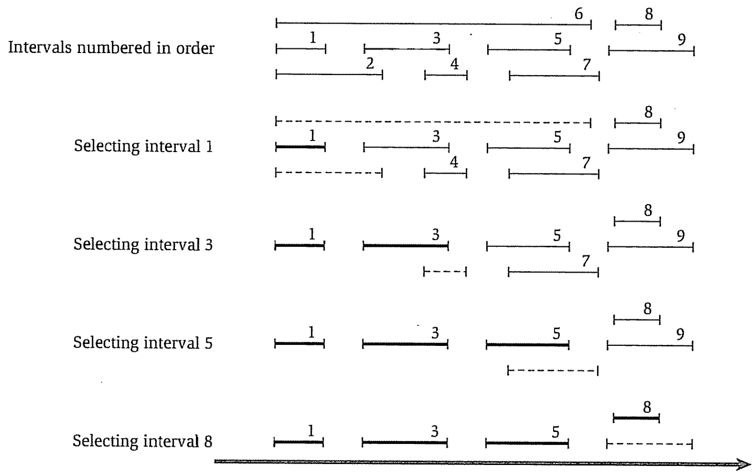

Figure 4.2 Sample run of the Interval Scheduling Algorithm. At each step the selected intervals are darker lines, and the intervals deleted at the corresponding step are indicated with dashed lines.

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.

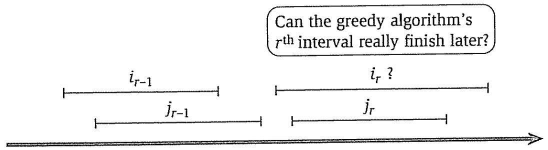

Figure 4.3 The inductive step in the proof that the greedy algorithm stays ahead.

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 $f(i_k) \le f(j_k)$. 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 $f(i) \le f(j)$ 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 $s(j) \ge f(1)$; 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 $s(j) \ge f$. 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.