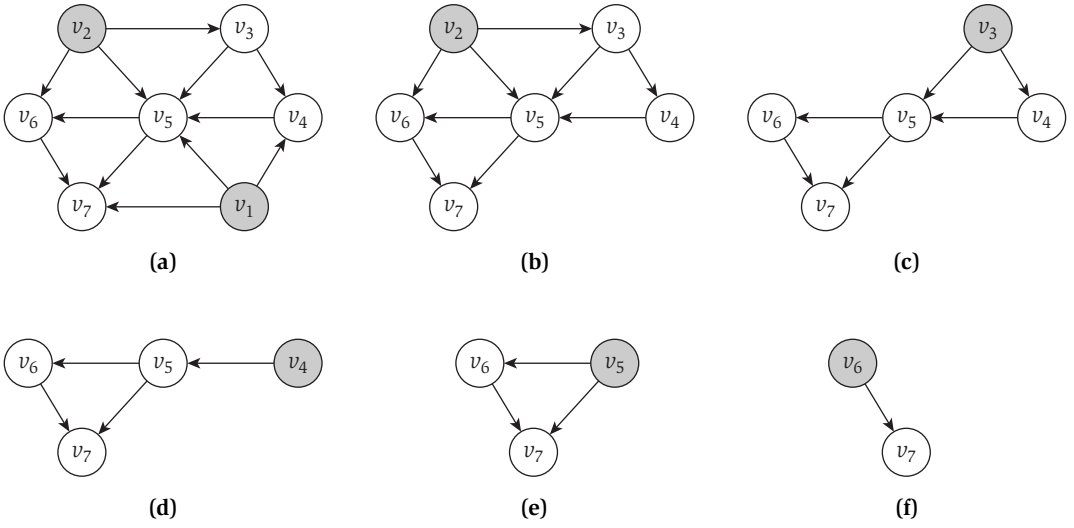

Figure 3.8 Starting from the graph in Figure 3.7, nodes are deleted one by one so as to be added to a topological ordering. The shaded nodes are those with no incoming edges; note that there is always at least one such edge at every stage of the algorithm's execution.

(3.19) guarantees, is that when we apply this algorithm to a DAG, there will always be at least one such node available to delete.

To bound the running time of this algorithm, we note that identifying a node v with no incoming edges, and deleting it from G, can be done in $O(n)$ time. Since the algorithm runs for n iterations, the total running time is $O(n^2)$.

This is not a bad running time; and if G is very dense, containing $\Theta(n^2)$ edges, then it is linear in the size of the input. But we may well want something better when the number of edges m is much less than n^2. In such a case, a running time of $O(m + n)$ could be a significant improvement over $\Theta(n^2)$.

In fact, we can achieve a running time of $O(m + n)$ using the same high-level algorithm—iteratively deleting nodes with no incoming edges. We simply have to be more efficient in finding these nodes, and we do this as follows.

We declare a node to be "active" if it has not yet been deleted by the algorithm, and we explicitly maintain two things:

(a) for each node w, the number of incoming edges that w has from active nodes; and

(b) the set S of all active nodes in G that have no incoming edges from other active nodes.

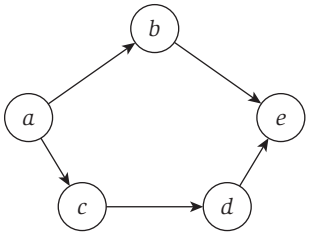

Figure 3.9 How many topological orderings does this graph have?

At the start, all nodes are active, so we can initialize (a) and (b) with a single pass through the nodes and edges. Then, each iteration consists of selecting a node v from the set S and deleting it. After deleting v, we go through all nodes w to which v had an edge, and subtract one from the number of active incoming edges that we are maintaining for w. If this causes the number of active incoming edges to w to drop to zero, then we add w to the set S. Proceeding in this way, we keep track of nodes that are eligible for deletion at all times, while spending constant work per edge over the course of the whole algorithm.

Solved Exercises

Solved Exercise 1

Consider the directed acyclic graph G in Figure 3.9. How many topological orderings does it have?

Solution Recall that a topological ordering of G is an ordering of the nodes as $v_1, v_2, \ldots, v_n$ so that all edges point "forward": for every edge (v_i, v_j), we have $i < j$.

So one way to answer this question would be to write down all $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$ possible orderings and check whether each is a topological ordering. But this would take a while.

Instead, we think about this as follows. As we saw in the text (or reasoning directly from the definition), the first node in a topological ordering must be one that has no edge coming into it. Analogously, the last node must be one that has no edge leaving it. Thus, in every topological ordering of G, the node a must come first and the node e must come last.

Now we have to figure how the nodes b, c, and d can be arranged in the middle of the ordering. The edge (c, d) enforces the requirement that c must come before d; but b can be placed anywhere relative to these two: before both, between c and d, or after both. This exhausts all the possibilities, and so we conclude that there are three possible topological orderings:

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

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

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

Solved Exercise 2

Some friends of yours are working on techniques for coordinating groups of mobile robots. Each robot has a radio transmitter that it uses to communicate

with a base station, and your friends find that if the robots get too close to one another, then there are problems with interference among the transmitters. So a natural problem arises: how to plan the motion of the robots in such a way that each robot gets to its intended destination, but in the process the robots don't come close enough together to cause interference problems.

We can model this problem abstractly as follows. Suppose that we have an undirected graph $G = (V, E)$, representing the floor plan of a building, and there are two robots initially located at nodes a and b in the graph. The robot at node a wants to travel to node c along a path in G, and the robot at node b wants to travel to node d. This is accomplished by means of a *schedule*: at each time step, the schedule specifies that one of the robots moves across a single edge, from one node to a neighboring node; at the end of the schedule, the robot from node a should be sitting on c, and the robot from b should be sitting on d.

A schedule is *interference-free* if there is no point at which the two robots occupy nodes that are at a distance $\leq r$ from one another in the graph, for a given parameter r. We'll assume that the two starting nodes a and b are at a distance greater than r, and so are the two ending nodes c and d.

Give a polynomial-time algorithm that decides whether there exists an interference-free schedule by which each robot can get to its destination.

Solution This is a problem of the following general flavor. We have a set of possible *configurations* for the robots, where we define a configuration to be a choice of location for each one. We are trying to get from a given starting configuration (a, b) to a given ending configuration (c, d), subject to constraints on how we can move between configurations (we can only change one robot's location to a neighboring node), and also subject to constraints on which configurations are "legal."

This problem can be tricky to think about if we view things at the level of the underlying graph G: for a given configuration of the robots—that is, the current location of each one—it's not clear what rule we should be using to decide how to move one of the robots next. So instead we apply an idea that can be very useful for situations in which we're trying to perform this type of search. We observe that our problem looks a lot like a path-finding problem, not in the original graph G but in the space of all possible configurations.

Let us define the following (larger) graph H. The node set of H is the set of all possible configurations of the robots; that is, H consists of all possible pairs of nodes in G. We join two nodes of H by an edge if they represent configurations that could be consecutive in a schedule; that is, (u, v) and (u', v') will be joined by an edge in H if one of the pairs u, u' or v, v' are equal, and the other pair corresponds to an edge in G.

We can already observe that paths in H from (a, b) to (c, d) correspond to schedules for the robots: such a path consists precisely of a sequence of configurations in which, at each step, one robot crosses a single edge in G. However, we have not yet encoded the notion that the schedule should be interference-free.

To do this, we simply delete from H all nodes that correspond to configurations in which there would be interference. Thus we define H' to be the graph obtained from H by deleting all nodes (u, v) for which the distance between u and v in G is at most r.

The full algorithm is then as follows. We construct the graph H', and then run the connectivity algorithm from the text to determine whether there is a path from (a, b) to (c, d). The correctness of the algorithm follows from the fact that paths in H' correspond to schedules, and the nodes in H' correspond precisely to the configurations in which there is no interference.

Finally, we need to consider the running time. Let n denote the number of nodes in G, and m denote the number of edges in G. We'll analyze the running time by doing three things: (1) bounding the size of H' (which will in general be larger than G), (2) bounding the time it takes to construct H', and (3) bounding the time it takes to search for a path from (a, b) to (c, d) in H.

1. First, then, let's consider the size of H'. H' has at most n^2 nodes, since its nodes correspond to pairs of nodes in G. Now, how many edges does H' have? A node (u, v) will have edges to (u', v) for each neighbor u' of u in G, and to (u, v') for each neighbor v' of v in G. A simple upper bound says that there can be at most n choices for (u', v), and at most n choices for (u, v'), so there are at most $2n$ edges incident to each node of H'. Summing over the (at most) n^2 nodes of H', we have $O(n^3)$ edges.

   (We can actually give a better bound of $O(mn)$ on the number of edges in H', by using the bound (3.9) we proved in Section 3.3 on the sum of the degrees in a graph. We'll leave this as a further exercise.)

2. Now we bound the time needed to construct H'. We first build H by enumerating all pairs of nodes in G in time $O(n^2)$, and constructing edges using the definition above in time $O(n)$ per node, for a total of $O(n^3)$. Now we need to figure out which nodes to delete from H so as to produce H'. We can do this as follows. For each node u in G, we run a breadth-first search from u and identify all nodes v within distance r of u. We list all these pairs (u, v) and delete them from H. Each breadth-first search in G takes time $O(m + n)$, and we're doing one from each node, so the total time for this part is $O(mn + n^2)$.

3. Now we have H', and so we just need to decide whether there is a path from (a, b) to (c, d). This can be done using the connectivity algorithm from the text in time that is linear in the number of nodes and edges of H'. Since H' has $O(n^2)$ nodes and $O(n^3)$ edges, this final step takes polynomial time as well.

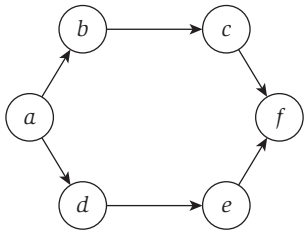

Figure 3.10 How many topological orderings does this graph have?

Exercises

1. Consider the directed acyclic graph G in Figure 3.10. How many topological orderings does it have?

2. Give an algorithm to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one. (It should not output all cycles in the graph, just one of them.) The running time of your algorithm should be $O(m + n)$ for a graph with n nodes and m edges.

3. The algorithm described in Section 3.6 for computing a topological ordering of a DAG repeatedly finds a node with no incoming edges and deletes it. This will eventually produce a topological ordering, provided that the input graph really is a DAG.

But suppose that we're given an arbitrary graph that may or may not be a DAG. Extend the topological ordering algorithm so that, given an input directed graph G, it outputs one of two things: (a) a topological ordering, thus establishing that G is a DAG; or (b) a cycle in G, thus establishing that G is not a DAG. The running time of your algorithm should be $O(m + n)$ for a directed graph with n nodes and m edges.

4. Inspired by the example of that great Cornellian, Vladimir Nabokov, some of your friends have become amateur lepidopterists (they study butterflies). Often when they return from a trip with specimens of butterflies, it is very difficult for them to tell how many distinct species they've caught—thanks to the fact that many species look very similar to one another.

One day they return with n butterflies, and they believe that each belongs to one of two different species, which we'll call A and B for purposes of this discussion. They'd like to divide the n specimens into two groups—those that belong to A and those that belong to B—but it's very hard for them to directly label any one specimen. So they decide to adopt the following approach.

For each pair of specimens i and j, they study them carefully side by side. If they're confident enough in their judgment, then they label the pair (i, j) either "same" (meaning they believe them both to come from the same species) or "different" (meaning they believe them to come from different species). They also have the option of rendering no judgment on a given pair, in which case we'll call the pair *ambiguous*.

So now they have the collection of n specimens, as well as a collection of m judgments (either "same" or "different") for the pairs that were not declared to be ambiguous. They'd like to know if this data is consistent with the idea that each butterfly is from one of species A or B. So more concretely, we'll declare the m judgments to be *consistent* if it is possible to label each specimen either A or B in such a way that for each pair (i, j) labeled "same," it is the case that i and j have the same label; and for each pair (i, j) labeled "different," it is the case that i and j have different labels. They're in the middle of tediously working out whether their judgments are consistent, when one of them realizes that you probably have an algorithm that would answer this question right away.

Give an algorithm with running time $O(m + n)$ that determines whether the m judgments are consistent.

5. A binary tree is a rooted tree in which each node has at most two children. Show by induction that in any binary tree the number of nodes with two children is exactly one less than the number of leaves.

6. We have a connected graph $G = (V, E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same tree T. Prove that $G = T$. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

7. Some friends of yours work on wireless networks, and they're currently studying the properties of a network of n mobile devices. As the devices move around (actually, as their human owners move around), they define a graph at any point in time as follows: there is a node representing each of the n devices, and there is an edge between device i and device j if the physical locations of i and j are no more than 500 meters apart. (If so, we say that i and j are "in range" of each other.)

They'd like it to be the case that the network of devices is connected at all times, and so they've constrained the motion of the devices to satisfy

the following property: at all times, each device i is within 500 meters of at least $n/2$ of the other devices. (We'll assume n is an even number.) What they'd like to know is: Does this property by itself guarantee that the network will remain connected?

Here's a concrete way to formulate the question as a claim about graphs.

> *Claim: Let G be a graph on n nodes, where n is an even number. If every node of G has degree at least $n/2$, then G is connected.*

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

8. A number of stories in the press about the structure of the Internet and the Web have focused on some version of the following question: How far apart are typical nodes in these networks? If you read these stories carefully, you find that many of them are confused about the difference between the *diameter* of a network and the *average distance* in a network; they often jump back and forth between these concepts as though they're the same thing.

As in the text, we say that the *distance* between two nodes u and v in a graph $G = (V, E)$ is the minimum number of edges in a path joining them; we'll denote this by $dist(u, v)$. We say that the *diameter* of G is the maximum distance between any pair of nodes; and we'll denote this quantity by $diam(G)$.

Let's define a related quantity, which we'll call the *average pairwise distance* in G (denoted $apd(G)$). We define $apd(G)$ to be the average, over all $\binom{n}{2}$ sets of two distinct nodes u and v, of the distance between u and v. That is,

$$apd(G) = \left[\sum_{\{u,v\} \subseteq V} dist(u, v) \right] / \binom{n}{2}.$$

Here's a simple example to convince yourself that there are graphs G for which $diam(G) \neq apd(G)$. Let G be a graph with three nodes u, v, w, and with the two edges $\{u, v\}$ and $\{v, w\}$. Then

$$diam(G) = dist(u, w) = 2,$$

while

$$apd(G) = [dist(u, v) + dist(u, w) + dist(v, w)]/3 = 4/3.$$

Of course, these two numbers aren't all *that* far apart in the case of this three-node graph, and so it's natural to ask whether there's always a close relation between them. Here's a claim that tries to make this precise.

Claim: There exists a positive natural number c so that for all connected graphs G, it is the case that

$$\frac{diam(G)}{apd(G)} \leq c.$$

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

9. There's a natural intuition that two nodes that are far apart in a communication network—separated by many hops—have a more tenuous connection than two nodes that are close together. There are a number of algorithmic results that are based to some extent on different ways of making this notion precise. Here's one that involves the susceptibility of paths to the deletion of nodes.

Suppose that an n-node undirected graph $G = (V, E)$ contains two nodes s and t such that the distance between s and t is strictly greater than $n/2$. Show that there must exist some node v, not equal to either s or t, such that deleting v from G destroys all s-t paths. (In other words, the graph obtained from G by deleting v contains no path from s to t.) Give an algorithm with running time $O(m + n)$ to find such a node v.

10. A number of art museums around the country have been featuring work by an artist named Mark Lombardi (1951–2000), consisting of a set of intricately rendered graphs. Building on a great deal of research, these graphs encode the relationships among people involved in major political scandals over the past several decades: the nodes correspond to participants, and each edge indicates some type of relationship between a pair of participants. And so, if you peer closely enough at the drawings, you can trace out ominous-looking paths from a high-ranking U.S. government official, to a former business partner, to a bank in Switzerland, to a shadowy arms dealer.

Such pictures form striking examples of *social networks*, which, as we discussed in Section 3.1, have nodes representing people and organizations, and edges representing relationships of various kinds. And the short paths that abound in these networks have attracted considerable attention recently, as people ponder what they mean. In the case of Mark Lombardi's graphs, they hint at the short set of steps that can carry you from the reputable to the disreputable.

Of course, a single, spurious short path between nodes v and w in such a network may be more coincidental than anything else; a large number of short paths between v and w can be much more convincing. So in addition to the problem of computing a single shortest v-w path in a graph G, social networks researchers have looked at the problem of determining the *number* of shortest v-w paths.

This turns out to be a problem that can be solved efficiently. Suppose we are given an undirected graph $G = (V, E)$, and we identify two nodes v and w in G. Give an algorithm that computes the number of shortest v-w paths in G. (The algorithm should not list all the paths; just the number suffices.) The running time of your algorithm should be $O(m + n)$ for a graph with n nodes and m edges.

11. You're helping some security analysts monitor a collection of networked computers, tracking the spread of an online virus. There are n computers in the system, labeled $C_1, C_2, \ldots, C_n$, and as input you're given a collection of *trace data* indicating the times at which pairs of computers communicated. Thus the data is a sequence of ordered triples (C_i, C_j, t_k); such a triple indicates that C_i and C_j exchanged bits at time t_k. There are m triples total.

We'll assume that the triples are presented to you in sorted order of time. For purposes of simplicity, we'll assume that each pair of computers communicates at most once during the interval you're observing.

The security analysts you're working with would like to be able to answer questions of the following form: If the virus was inserted into computer C_a at time x, could it possibly have infected computer C_b by time y? The mechanics of infection are simple: if an infected computer C_i communicates with an uninfected computer C_j at time t_k (in other words, if one of the triples (C_i, C_j, t_k) or (C_j, C_i, t_k) appears in the trace data), then computer C_j becomes infected as well, starting at time t_k. Infection can thus spread from one machine to another across a *sequence* of communications, provided that no step in this sequence involves a move backward in time. Thus, for example, if C_i is infected by time t_k, and the trace data contains triples (C_i, C_j, t_k) and (C_j, C_q, t_r), where $t_k \leq t_r$, then C_q will become infected via C_j. (Note that it is okay for t_k to be equal to t_r; this would mean that C_j had open connections to both C_i and C_q at the same time, and so a virus could move from C_i to C_q.)

For example, suppose $n = 4$, the trace data consists of the triples

$$(C_1, C_2, 4), \quad (C_2, C_4, 8), \quad (C_3, C_4, 8), \quad (C_1, C_4, 12),$$

and the virus was inserted into computer C_1 at time 2. Then C_3 would be infected at time 8 by a sequence of three steps: first C_2 becomes infected at time 4, then C_4 gets the virus from C_2 at time 8, and then C_3 gets the virus from C_4 at time 8. On the other hand, if the trace data were

$$(C_2, C_3, 8), \quad (C_1, C_4, 12), \quad (C_1, C_2, 14),$$

and again the virus was inserted into computer C_1 at time 2, then C_3 would not become infected during the period of observation: although C_2 becomes infected at time 14, we see that C_3 only communicates with C_2 *before* C_2 was infected. There is no sequence of communications moving forward in time by which the virus could get from C_1 to C_3 in this second example.

Design an algorithm that answers questions of this type: given a collection of trace data, the algorithm should decide whether a virus introduced at computer C_a at time x could have infected computer C_b by time y. The algorithm should run in time $O(m + n)$.

12. You're helping a group of ethnographers analyze some oral history data they've collected by interviewing members of a village to learn about the lives of people who've lived there over the past two hundred years.

From these interviews, they've learned about a set of n people (all of them now deceased), whom we'll denote $P_1, P_2, \ldots, P_n$. They've also collected facts about when these people lived relative to one another. Each fact has one of the following two forms:

- For some i and j, person P_i died before person P_j was born; or
- for some i and j, the life spans of P_i and P_j overlapped at least partially.

Naturally, they're not sure that all these facts are correct; memories are not so good, and a lot of this was passed down by word of mouth. So what they'd like you to determine is whether the data they've collected is at least internally consistent, in the sense that there could have existed a set of people for which all the facts they've learned simultaneously hold.

Give an efficient algorithm to do this: either it should produce proposed dates of birth and death for each of the n people so that all the facts hold true, or it should report (correctly) that no such dates can exist—that is, the facts collected by the ethnographers are not internally consistent.

Notes and Further Reading

The theory of graphs is a large topic, encompassing both algorithmic and non-algorithmic issues. It is generally considered to have begun with a paper by

Euler (1736), grown through interest in graph representations of maps and chemical compounds in the nineteenth century, and emerged as a systematic area of study in the twentieth century, first as a branch of mathematics and later also through its applications to computer science. The books by Berge (1976), Bollobas (1998), and Diestel (2000) provide substantial further coverage of graph theory. Recently, extensive data has become available for studying large networks that arise in the physical, biological, and social sciences, and there has been interest in understanding properties of networks that span all these different domains. The books by Barabasi (2002) and Watts (2002) discuss this emerging area of research, with presentations aimed at a general audience.

The basic graph traversal techniques covered in this chapter have numerous applications. We will see a number of these in subsequent chapters, and we refer the reader to the book by Tarjan (1983) for further results.

Notes on the Exercises Exercise 12 is based on a result of Martin Golumbic and Ron Shamir.

This page intentionally left blank

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.

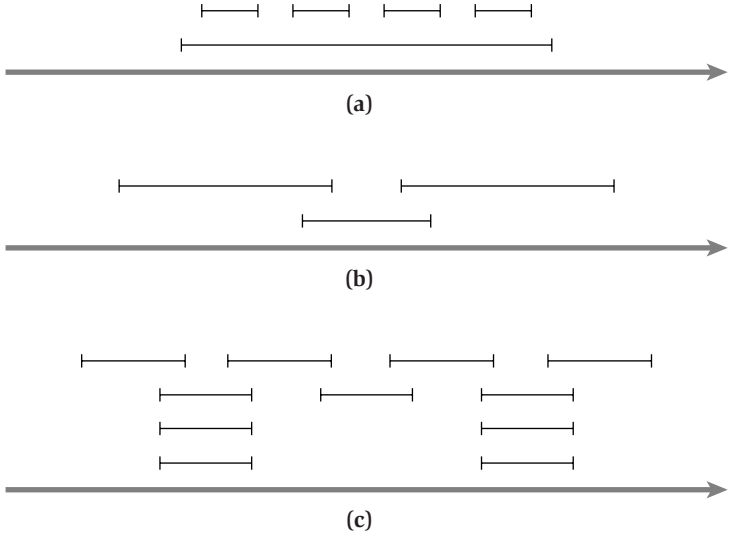

Figure 4.1 Some instances of the Interval Scheduling Problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the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