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.

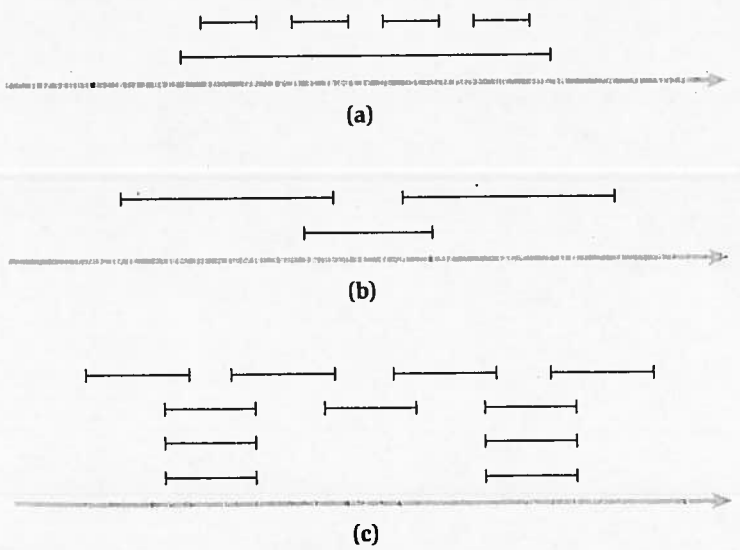


Figure 4.1 Some instances of the Interval Scheduling Problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the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

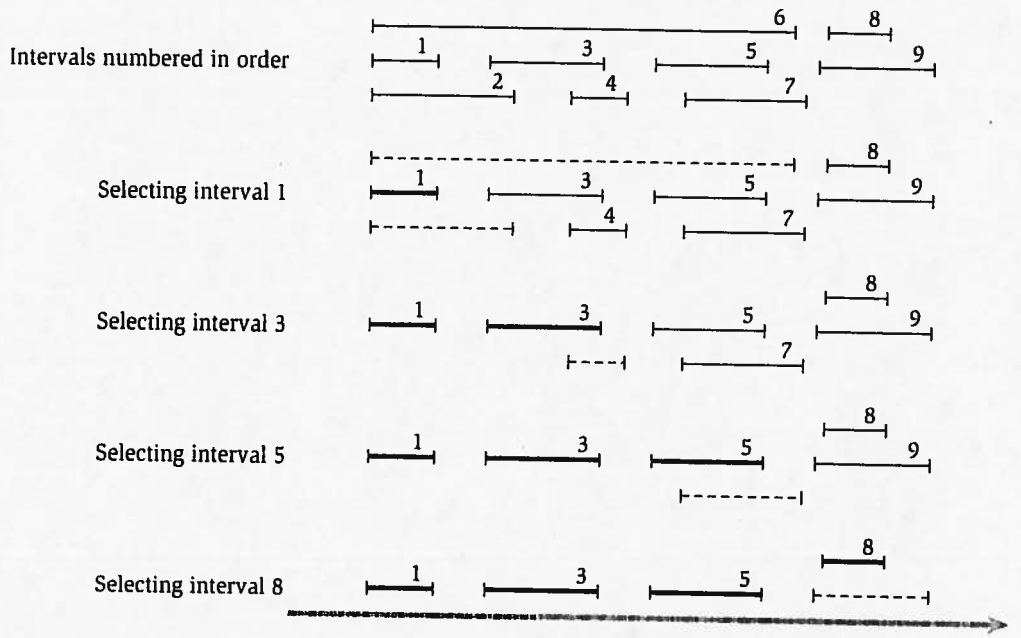


Figure 4.2 Sample run of the Interval Scheduling Algorithm. At each step the selected intervals are darker lines, and the intervals deleted at the corresponding step are indicated with dashed lines.

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.

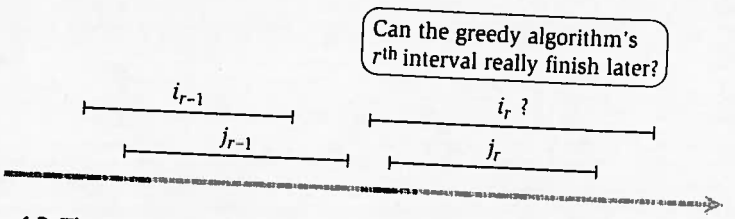


Figure 4.3 The inductive step in the proof that the greedy algorithm stays ahead.

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 $f(i_1) \leq f(j_1)$. 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 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 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 O: for each r, the r^{th} interval it selects finishes at least as soon as the r^{th} interval in 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 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 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.

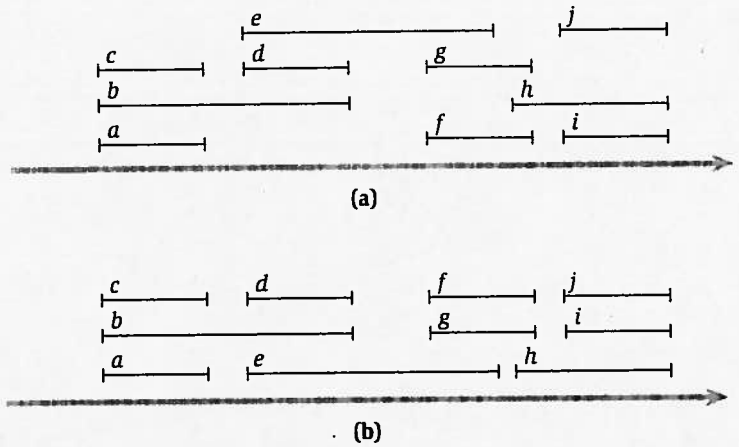


Figure 4.4 (a) An instance of the Interval Partitioning Problem with ten intervals (*a* through *j*). (b) A solution in which all intervals are scheduled using three resources: each row represents a set of intervals that can all be scheduled on a single resource.

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, ..., I_n denote the intervals in this order
For j = 1, 2, 3, ..., 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, ..., 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

brute-force search: although it's systematically working through the exponentially large set of possible solutions to the problem, it does this without ever examining them all explicitly. It is because of this careful balancing act that dynamic programming can be a tricky technique to get used to; it typically takes a reasonable amount of practice before one is fully comfortable with it.

With this in mind, we now turn to a first example of dynamic programming: the Weighted Interval Scheduling Problem that we defined back in Section 1.2. We are going to develop a dynamic programming algorithm for this problem in two stages: first as a recursive procedure that closely resembles brute-force search; and then, by reinterpreting this procedure, as an iterative algorithm that works by building up solutions to larger and larger subproblems.

6.1 Weighted Interval Scheduling: A Recursive Procedure

We have seen that a particular greedy algorithm produces an optimal solution to the Interval Scheduling Problem, where the goal is to accept as large a set of nonoverlapping intervals as possible. The Weighted Interval Scheduling Problem is a strictly more general version, in which each interval has a certain *value* (or *weight*), and we want to accept a set of maximum value.

Designing a Recursive Algorithm

Since the original Interval Scheduling Problem is simply the special case in which all values are equal to 1, we know already that most greedy algorithms will not solve this problem optimally. But even the algorithm that worked before (repeatedly choosing the interval that ends earliest) is no longer optimal in this more general setting, as the simple example in Figure 6.1 shows.

Indeed, no natural greedy algorithm is known for this problem, which is what motivates our switch to dynamic programming. As discussed above, we will begin our introduction to dynamic programming with a recursive type of algorithm for this problem, and then in the next section we'll move to a more iterative method that is closer to the style we use in the rest of this chapter.

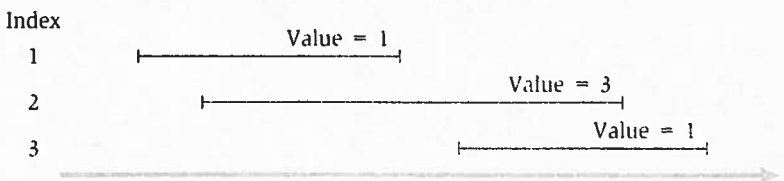


Figure 6.1 A simple instance of weighted interval scheduling.

We use the notation from our discussion of Interval Scheduling in Section 1.2. We have n requests labeled $1, \ldots, n$, with each request i specifying a start time s_i and a finish time f_i. Each interval i now also has a *value*, or *weight* v_i. Two intervals are *compatible* if they do not overlap. The goal of our current problem is to select a subset $S \subseteq \{1, \ldots, n\}$ of mutually compatible intervals, so as to maximize the sum of the values of the selected intervals, $\sum_{i \in S} v_i$.

Let's suppose that the requests are sorted in order of nondecreasing finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$. We'll say a request i comes *before* a request j if $i < j$. This will be the natural left-to-right order in which we'll consider intervals. To help in talking about this order, we define $p(j)$, for an interval j, to be the largest index $i < j$ such that intervals i and j are disjoint. In other words, i is the leftmost interval that ends before j begins. We define $p(j) = 0$ if no request $i < j$ is disjoint from j. An example of the definition of $p(j)$ is shown in Figure 6.2.

Now, given an instance of the Weighted Interval Scheduling Problem, let's consider an optimal solution $\mathcal{O}$, ignoring for now that we have no idea what it is. Here's something completely obvious that we can say about $\mathcal{O}$: either interval n (the last one) belongs to $\mathcal{O}$, or it doesn't. Suppose we explore both sides of this dichotomy a little further. If $n \in \mathcal{O}$, then clearly no interval indexed strictly between $p(n)$ and n can belong to $\mathcal{O}$, because by the definition of $p(n)$, we know that intervals $p(n) + 1, p(n) + 2, \ldots, n - 1$ all overlap interval n. Moreover, if $n \in \mathcal{O}$, then $\mathcal{O}$ must include an *optimal* solution to the problem consisting of requests $\{1, \ldots, p(n)\}$—for if it didn't, we could replace $\mathcal{O}$'s choice of requests from $\{1, \ldots, p(n)\}$ with a better one, with no danger of overlapping request n.

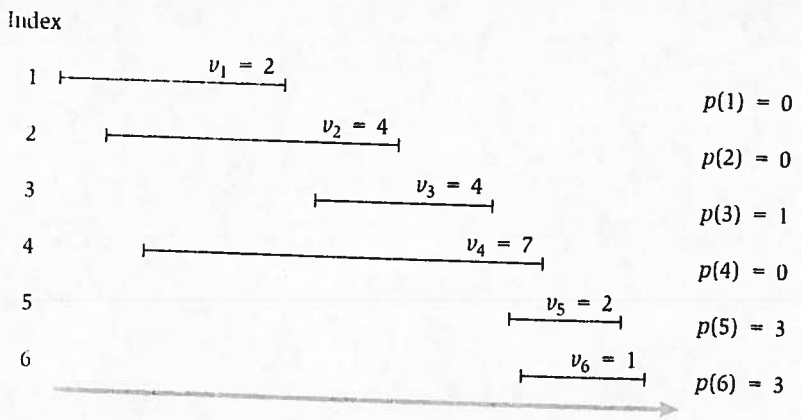


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

On the other hand, if $n \notin \mathcal{O}$, then $\mathcal{O}$ is simply equal to the optimal solution to the problem consisting of requests $\{1, \ldots, n-1\}$. This is by completely analogous reasoning: we're assuming that $\mathcal{O}$ does not include request n; so if it does not choose the optimal set of requests from $\{1, \ldots, n-1\}$, we could replace it with a better one.

All this suggests that finding the optimal solution on intervals $\{1, 2, \ldots, n\}$ involves looking at the optimal solutions of smaller problems of the form $\{1, 2, \ldots, j\}$. Thus, for any value of j between 1 and n, let $\mathcal{O}_j$ denote the optimal solution to the problem consisting of requests $\{1, \ldots, j\}$, and let $\text{OPT}(j)$ denote the value of this solution. (We define $\text{OPT}(0) = 0$, based on the convention that this is the optimum over an empty set of intervals.) The optimal solution we're seeking is precisely $\mathcal{O}_n$, with value $\text{OPT}(n)$. For the optimal solution $\mathcal{O}_j$ on $\{1, 2, \ldots, j\}$, our reasoning above (generalizing from the case in which $j = n$) says that either $j \in \mathcal{O}_j$, in which case $\text{OPT}(j) = v_j + \text{OPT}(p(j))$, or $j \notin \mathcal{O}_j$, in which case $\text{OPT}(j) = \text{OPT}(j-1)$. Since these are precisely the two possible choices ($j \in \mathcal{O}_j$ or $j \notin \mathcal{O}_j$), we can further say that

(6.1) $$\text{OPT}(j) = \max(v_j + \text{OPT}(p(j)), \text{OPT}(j-1)).$$

And how do we decide whether n belongs to the optimal solution $\mathcal{O}_j$? This too is easy: it belongs to the optimal solution if and only if the first of the options above is at least as good as the second; in other words,

(6.2) *Request j belongs to an optimal solution on the set $\{1, 2, \ldots, j\}$ if and only if*

$$v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1).$$

These facts form the first crucial component on which a dynamic programming solution is based: a recurrence equation that expresses the optimal solution (or its value) in terms of the optimal solutions to smaller subproblems.

Despite the simple reasoning that led to this point, (6.1) is already a significant development. It directly gives us a recursive algorithm to compute $\text{OPT}(n)$, assuming that we have already sorted the requests by finishing time and computed the values of $p(j)$ for each j.

```
Compute-Opt(j)
  If j=0 then
    Return 0
  Else
    Return max(v_j+Compute-Opt(p(j)), Compute-Opt(j-1))
  Endif
```

The correctness of the algorithm follows directly by induction on j:

(6.3) `Compute-Opt`(j) *correctly computes* $\text{OPT}(j)$ *for each* $j = 1, 2, \ldots, n$.

Proof. By definition $\text{OPT}(0) = 0$. Now, take some $j > 0$, and suppose by way of induction that `Compute-Opt`(i) correctly computes $\text{OPT}(i)$ for all $i < j$. By the induction hypothesis, we know that `Compute-Opt`$(p(j)) = \text{OPT}(p(j))$ and `Compute-Opt`$(j-1) = \text{OPT}(j-1)$; and hence from (6.1) it follows that

$$\begin{aligned}\text{OPT}(j) &= \max(v_j + \texttt{Compute-Opt}(p(j)), \texttt{Compute-Opt}(j-1)) \\ &= \texttt{Compute-Opt}(j). \quad \blacksquare\end{aligned}$$

Unfortunately, if we really implemented the algorithm `Compute-Opt` as just written, it would take exponential time to run in the worst case. For example, see Figure 6.3 for the tree of calls issued for the instance of Figure 6.2: the tree widens very quickly due to the recursive branching. To take a more extreme example, on a nicely layered instance like the one in Figure 6.4, where $p(j) = j - 2$ for each $j = 2, 3, 4, \ldots, n$, we see that `Compute-Opt`(j) generates separate recursive calls on problems of sizes $j-1$ and $j-2$. In other words, the total number of calls made to `Compute-Opt` on this instance will grow

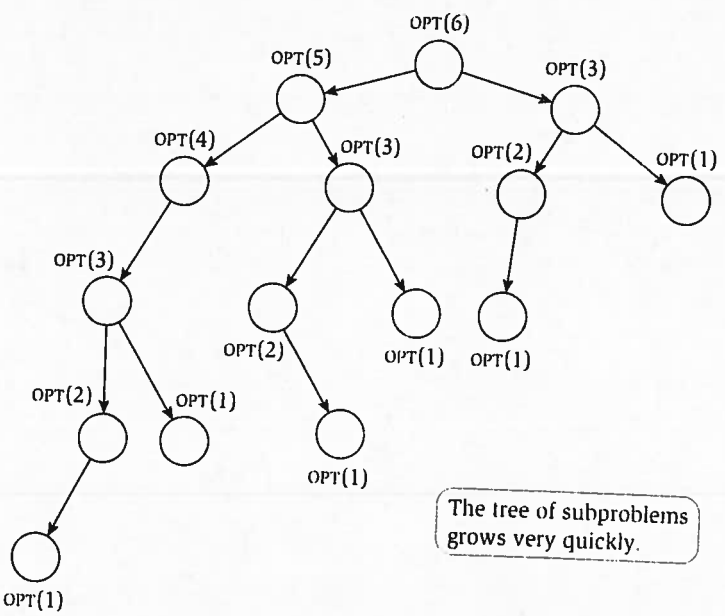


Figure 6.3 The tree of subproblems called by `Compute-Opt` on the problem instance of Figure 6.2.

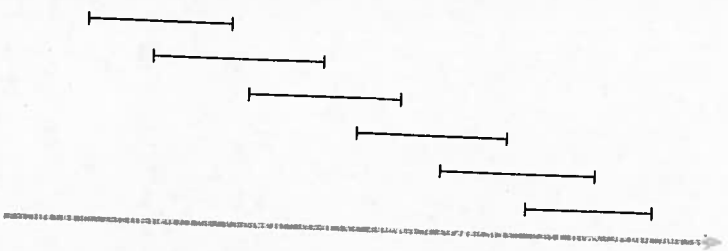


Figure 6.4 An instance of weighted interval scheduling on which the simple Compute-Opt recursion will take exponential time. The values of all intervals in this instance are 1.

like the Fibonacci numbers, which increase exponentially. Thus we have not achieved a polynomial-time solution.

Memoizing the Recursion

In fact, though, we're not so far from having a polynomial-time algorithm. A fundamental observation, which forms the second crucial component of a dynamic programming solution, is that our recursive algorithm Compute-Opt is really only solving $n+1$ different subproblems: Compute-Opt(0), Compute-Opt(1), ..., Compute-Opt(n). The fact that it runs in exponential time as written is simply due to the spectacular redundancy in the number of times it issues each of these calls.

How could we eliminate all this redundancy? We could store the value of Compute-Opt in a globally accessible place the first time we compute it and then simply use this precomputed value in place of all future recursive calls. This technique of saving values that have already been computed is referred to as *memoization*.

We implement the above strategy in the more "intelligent" procedure M-Compute-Opt. This procedure will make use of an array $M[0 \ldots n]$; $M[j]$ will start with the value "empty," but will hold the value of Compute-Opt(j) as soon as it is first determined. To determine OPT(n), we invoke M-Compute-Opt(n).

```
M-Compute-Opt(j)
  If j = 0 then
    Return 0
  Else if M[j] is not empty then
    Return M[j]
  Else
```

```
  Define M[j] = max(v_j+M-Compute-Opt(p(j)), M-Compute-Opt(j − 1))
   Return M[j]
 Endif
```

Analyzing the Memoized Version

Clearly, this looks very similar to our previous implementation of the algorithm; however, memoization has brought the running time way down.

(6.4) *The running time of* `M-Compute-Opt`(n) *is* $O(n)$ *(assuming the input intervals are sorted by their finish times).*

Proof. The time spent in a single call to `M-Compute-Opt` is $O(1)$, excluding the time spent in recursive calls it generates. So the running time is bounded by a constant times the number of calls ever issued to `M-Compute-Opt`. Since the implementation itself gives no explicit upper bound on this number of calls, we try to find a bound by looking for a good measure of "progress."

The most useful progress measure here is the number of entries in M that are not "empty." Initially this number is 0; but each time the procedure invokes the recurrence, issuing two recursive calls to `M-Compute-Opt`, it fills in a new entry, and hence increases the number of filled-in entries by 1. Since M has only $n + 1$ entries, it follows that there can be at most $O(n)$ calls to `M-Compute-Opt`, and hence the running time of `M-Compute-Opt`(n) is $O(n)$, as desired. ■

Computing a Solution in Addition to Its Value

So far we have simply computed the *value* of an optimal solution; presumably we want a full optimal set of intervals as well. It would be easy to extend `M-Compute-Opt` so as to keep track of an optimal solution in addition to its value: we could maintain an additional array S so that $S[i]$ contains an optimal set of intervals among $\{1, 2, \ldots, i\}$. Naively enhancing the code to maintain the solutions in the array S, however, would blow up the running time by an additional factor of $O(n)$: while a position in the M array can be updated in $O(1)$ time, writing down a set in the S array takes $O(n)$ time. We can avoid this $O(n)$ blow-up by not explicitly maintaining S, but rather by recovering the optimal solution from values saved in the array M after the optimum value has been computed.

We know from (6.2) that j belongs to an optimal solution for the set of intervals $\{1, \ldots, j\}$ if and only if $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j - 1)$. Using this observation, we get the following simple procedure, which "traces back" through the array M to find the set of intervals in an optimal solution.

```
Find-Solution(j)
  If j = 0 then
    Output nothing
  Else
    If v_j + M[p(j)] ≥ M[j − 1] then
      Output j together with the result of Find-Solution(p(j))
    Else
      Output the result of Find-Solution(j − 1)
    Endif
  Endif
```

Since Find-Solution calls itself recursively only on strictly smaller values, it makes a total of $O(n)$ recursive calls; and since it spends constant time per call, we have

(6.5) *Given the array M of the optimal values of the sub-problems,* Find-Solution *returns an optimal solution in $O(n)$ time.*

6.2 Principles of Dynamic Programming: Memoization or Iteration over Subproblems

We now use the algorithm for the Weighted Interval Scheduling Problem developed in the previous section to summarize the basic principles of dynamic programming, and also to offer a different perspective that will be fundamental to the rest of the chapter: iterating over subproblems, rather than computing solutions recursively.

In the previous section, we developed a polynomial-time solution to the Weighted Interval Scheduling Problem by first designing an exponential-time recursive algorithm and then converting it (by memoization) to an efficient recursive algorithm that consulted a global array M of optimal solutions to subproblems. To really understand what is going on here, however, it helps to formulate an essentially equivalent version of the algorithm. It is this new formulation that most explicitly captures the essence of the dynamic programming technique, and it will serve as a general template for the algorithms we develop in later sections.

Designing the Algorithm

The key to the efficient algorithm is really the array M. It encodes the notion that we are using the value of optimal solutions to the subproblems on intervals $\{1, 2, \ldots, j\}$ for each j, and it uses (6.1) to define the value of $M[j]$ based on

```
Find-Solution(j)
  If j = 0 then
    Output nothing
  Else
    If v_j + M[p(j)] ≥ M[j − 1] then
      Output j together with the result of Find-Solution(p(j))
    Else
      Output the result of Find-Solution(j − 1)
    Endif
  Endif
```

Since Find-Solution calls itself recursively only on strictly smaller values, it makes a total of $O(n)$ recursive calls; and since it spends constant time per call, we have

(6.5) *Given the array M of the optimal values of the sub-problems,* Find-Solution *returns an optimal solution in $O(n)$ time.*

6.2 Principles of Dynamic Programming: Memoization or Iteration over Subproblems

We now use the algorithm for the Weighted Interval Scheduling Problem developed in the previous section to summarize the basic principles of dynamic programming, and also to offer a different perspective that will be fundamental to the rest of the chapter: iterating over subproblems, rather than computing solutions recursively.

In the previous section, we developed a polynomial-time solution to the Weighted Interval Scheduling Problem by first designing an exponential-time recursive algorithm and then converting it (by memoization) to an efficient recursive algorithm that consulted a global array M of optimal solutions to subproblems. To really understand what is going on here, however, it helps to formulate an essentially equivalent version of the algorithm. It is this new formulation that most explicitly captures the essence of the dynamic programming technique, and it will serve as a general template for the algorithms we develop in later sections.

Designing the Algorithm

The key to the efficient algorithm is really the array M. It encodes the notion that we are using the value of optimal solutions to the subproblems on intervals $\{1, 2, \ldots, j\}$ for each j, and it uses (6.1) to define the value of $M[j]$ based on

values that come earlier in the array. Once we have the array M, the problem is solved: $M[n]$ contains the value of the optimal solution on the full instance, and `Find-Solution` can be used to trace back through M efficiently and return an optimal solution itself.

The point to realize, then, is that we can directly compute the entries in M by an iterative algorithm, rather than using memoized recursion. We just start with $M[0] = 0$ and keep incrementing j; each time we need to determine a value $M[j]$, the answer is provided by (6.1). The algorithm looks as follows.

```
Iterative-Compute-Opt
  M[0] = 0
  For j = 1, 2, ..., n
    M[j] = max(v_j + M[p(j)], M[j − 1])
  Endfor
```

Analyzing the Algorithm

By exact analogy with the proof of (6.3), we can prove by induction on j that this algorithm writes $\text{OPT}(j)$ in array entry $M[j]$; (6.1) provides the induction step. Also, as before, we can pass the filled-in array M to `Find-Solution` to get an optimal solution in addition to the value. Finally, the running time of `Iterative-Compute-Opt` is clearly $O(n)$, since it explicitly runs for n iterations and spends constant time in each.

An example of the execution of `Iterative-Compute-Opt` is depicted in Figure 6.5. In each iteration, the algorithm fills in one additional entry of the array M, by comparing the value of $v_j + M[p(j)]$ to the value of $M[j-1]$.

A Basic Outline of Dynamic Programming

This, then, provides a second efficient algorithm to solve the Weighted Interval Scheduling Problem. The two approaches clearly have a great deal of conceptual overlap, since they both grow from the insight contained in the recurrence (6.1). For the remainder of the chapter, we will develop dynamic programming algorithms using the second type of approach—iterative building up of subproblems—because the algorithms are often simpler to express this way. But in each case that we consider, there is an equivalent way to formulate the algorithm as a memoized recursion.

Most crucially, the bulk of our discussion about the particular problem of selecting intervals can be cast more generally as a rough template for designing dynamic programming algorithms. To set about developing an algorithm based on dynamic programming, one needs a collection of subproblems derived from the original problem that satisfies a few basic properties.

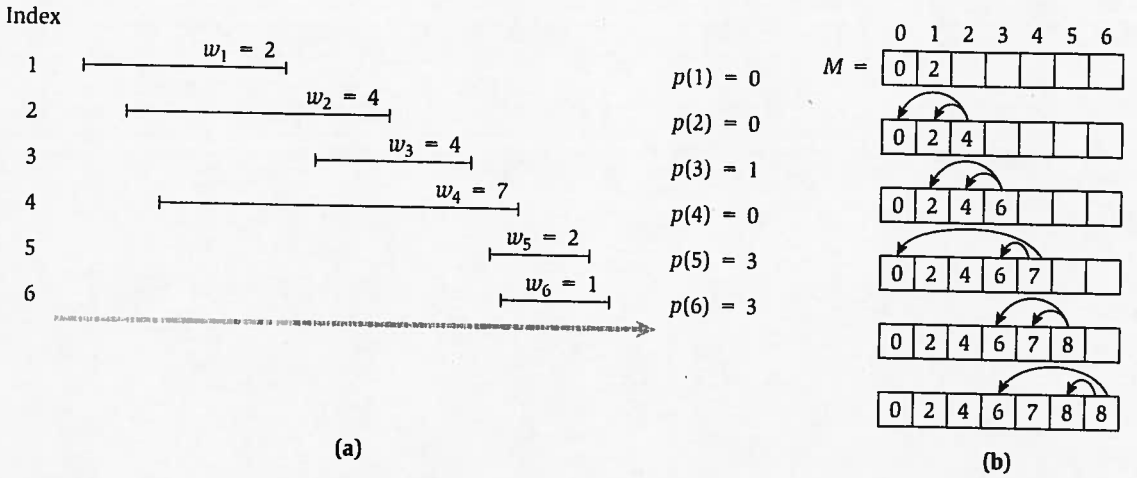


Figure 6.5 Part (b) shows the iterations of `Iterative-Compute-Opt` on the sample instance of Weighted Interval Scheduling depicted in part (a).

(i) There are only a polynomial number of subproblems.

(ii) The solution to the original problem can be easily computed from the solutions to the subproblems. (For example, the original problem may actually *be* one of the subproblems.)

(iii) There is a natural ordering on subproblems from "smallest" to "largest," together with an easy-to-compute recurrence (as in (6.1) and (6.2)) that allows one to determine the solution to a subproblem from the solutions to some number of smaller subproblems.

Naturally, these are informal guidelines. In particular, the notion of "smaller" in part (iii) will depend on the type of recurrence one has.

We will see that it is sometimes easier to start the process of designing such an algorithm by formulating a set of subproblems that looks natural, and then figuring out a recurrence that links them together; but often (as happened in the case of weighted interval scheduling), it can be useful to first define a recurrence by reasoning about the structure of an optimal solution, and then determine which subproblems will be necessary to unwind the recurrence. This chicken-and-egg relationship between subproblems and recurrences is a subtle issue underlying dynamic programming. It's never clear that a collection of subproblems will be useful until one finds a recurrence linking them together; but it can be difficult to think about recurrences in the absence of the "smaller" subproblems that they build on. In subsequent sections, we will develop further practice in managing this design trade-off.

In general: If our prior model (before running the test) is that $\Pr(E) \geq 2^i/(2^i+1)$ and if the test returns that the identity is correct (event B), then

$$\Pr(E \mid B) \geq \frac{\frac{2^i}{2^i+1}}{\frac{2^i}{2^i+1} + \frac{1}{2}\frac{1}{2^i+1}} = \frac{2^{i+1}}{2^{i+1}+1} = 1 - \frac{1}{2^i+1}.$$

Thus, if all 100 calls to the matrix identity test return that the identity is correct, our confidence in the correctness of this identity is at least $1 - 1/(2^{100}+1)$.

1.4. Application: A Randomized Min-Cut Algorithm

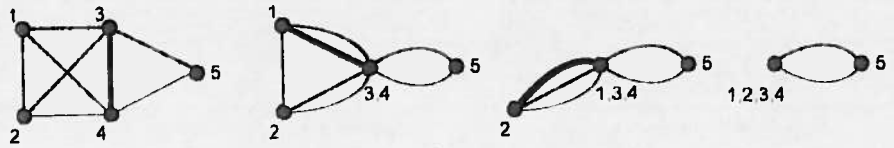
A *cut-set* in a graph is a set of edges whose removal breaks the graph into two or more connected components. Given a graph $G = (V, E)$ with n vertices, the minimum cut – or *min-cut* – problem is to find a minimum cardinality cut-set in G. Minimum cut problems arise in many contexts, including the study of network reliability. In the case where nodes correspond to machines in the network and edges correspond to connections between machines, the min-cut is the smallest number of edges that can fail before some pair of machines cannot communicate. Minimum cuts also arise in clustering problems. For example, if nodes represent Web pages (or any documents in a hypertext-based system) and two nodes have an edge between them if the corresponding nodes have a hyperlink between them, then small cuts divide the graph into clusters of documents with few links between clusters. Documents in different clusters are likely to be unrelated.

We shall proceed by making use of the definitions and techniques presented so far in order to analyze a simple randomized algorithm for the min-cut problem. The main operation in the algorithm is *edge contraction*. In contracting an edge $\{u, v\}$ we merge the two vertices u and v into one vertex, eliminate all edges connecting u and v, and retain all other edges in the graph. The new graph may have parallel edges but no self-loops. Examples appear in Figure 1.1, where in each step the dark edge is being contracted.

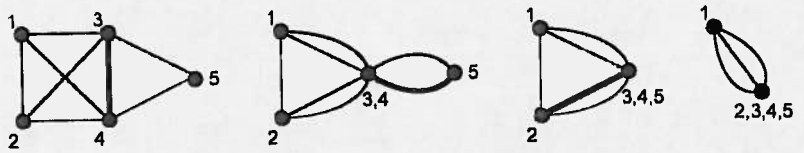
The algorithm consists of $n-2$ iterations. In each iteration, the algorithm picks an edge from the existing edges in the graph and contracts that edge. There are many possible ways one could choose the edge at each step. Our randomized algorithm chooses the edge uniformly at random from the remaining edges.

Each iteration reduces the number of vertices in the graph by one. After $n-2$ iterations, the graph consists of two vertices. The algorithm outputs the set of edges connecting the two remaining vertices.

It is easy to verify that any cut-set of a graph in an intermediate iteration of the algorithm is also a cut-set of the original graph. On the other hand, not every cut-set of the original graph is a cut-set of a graph in an intermediate iteration, since some edges of the cut-set may have been contracted in previous iterations. As a result, the output of the algorithm is always a cut-set of the original graph but not necessarily the minimum cardinality cut-set (see Figure 1.1).



(a) A successful run of min-cut.



(b) An unsuccessful run of min-cut.

Figure 1.1: An example of two executions of min-cut in a graph with minimum cut-set of size 2.

We now establish a lower bound on the probability that the algorithm returns a correct output.

Theorem 1.8: *The algorithm outputs a min-cut set with probability at least* $2/n(n-1)$.

Proof: Let k be the size of the min-cut set of G. The graph may have several cut-sets of minimum size. We compute the probability of finding one specific such set C.

Since C is a cut-set in the graph, removal of the set C partitions the set of vertices into two sets, S and $V - S$, such that there are no edges connecting vertices in S to vertices in $V - S$. Assume that, throughout an execution of the algorithm, we contract only edges that connect two vertices in S or two vertices in $V - S$, but not edges in C. In that case, all the edges eliminated throughout the execution will be edges connecting vertices in S or vertices in $V - S$, and after $n - 2$ iterations the algorithm returns a graph with two vertices connected by the edges in C. We may therefore conclude that, if the algorithm never chooses an edge of C in its $n - 2$ iterations, then the algorithm returns C as the minimum cut-set.

This argument gives some intuition for why we choose the edge at each iteration uniformly at random from the remaining existing edges. If the size of the cut C is small and if the algorithm chooses the edge uniformly at each step, then the probability that the algorithm chooses an edge of C is small – at least when the number of edges remaining is large compared to C.

Let E_i be the event that the edge contracted in iteration i is not in C, and let $F_i = \bigcap_{j=1}^{i} E_j$ be the event that no edge of C was contracted in the first i iterations. We need to compute $\Pr(F_{n-2})$.

We start by computing $\Pr(E_1) = \Pr(F_1)$. Since the minimum cut-set has k edges, all vertices in the graph must have degree k or larger. If each vertex is adjacent to at least k edges, then the graph must have at least $nk/2$ edges. The first contracted edge is chosen uniformly at random from the set of all edges. Since there are at least $nk/2$ edges in the graph and since C has k edges, the probability that we do not choose an edge of C in the first iteration is given by

$$\Pr(E_1) = \Pr(F_1) \geq 1 - \frac{2k}{nk} = 1 - \frac{2}{n}.$$

Let us suppose that the first contraction did not eliminate an edge of C. In other words, we condition on the event F_1. Then, after the first iteration, we are left with an $(n-1)$-node graph with minimum cut-set of size k. Again, the degree of each vertex in the graph must be at least k, and the graph must have at least $k(n-1)/2$ edges. Thus,

$$\Pr(E_2 \mid F_1) \geq 1 - \frac{k}{k(n-1)/2} = 1 - \frac{2}{n-1}.$$

Similarly,

$$\Pr(E_i \mid F_{i-1}) \geq 1 - \frac{k}{k(n-i+1)/2} = 1 - \frac{2}{n-i+1}.$$

To compute $\Pr(F_{n-2})$, we use

$$\begin{aligned}
\Pr(F_{n-2}) &= \Pr(E_{n-2} \cap F_{n-3}) = \Pr(E_{n-2} \mid F_{n-3}) \cdot \Pr(F_{n-3}) \\
&= \Pr(E_{n-2} \mid F_{n-3}) \cdot \Pr(E_{n-3} \mid F_{n-4}) \cdots \Pr(E_2 \mid F_1) \cdot \Pr(F_1) \\
&\geq \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) = \prod_{i=1}^{n-2} \left(\frac{n-i-1}{n-i+1}\right) \\
&= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right) \cdots \left(\frac{4}{6}\right)\left(\frac{3}{5}\right)\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) \\
&= \frac{2}{n(n-1)}.
\end{aligned}$$

■

Since the algorithm has a one-sided error, we can reduce the error probability by repeating the algorithm. Assume that we run the randomized min-cut algorithm $n(n-1)\ln n$ times and output the minimum size cut-set found in all the iterations. The probability that the output is not a min-cut set is bounded by

$$\left(1 - \frac{2}{n(n-1)}\right)^{n(n-1)\ln n} \leq e^{-2\ln n} = \frac{1}{n^2}.$$

In the first inequality we have used the fact that $1 - x \leq e^{-x}$.

1.5. Exercises

Exercise 1.1: We flip a fair coin ten times. Find the probability of the following events.

(a) The number of heads and the number of tails are equal.
(b) There are more heads than tails.
(c) The ith flip and the $(11-i)$th flip are the same for $i = 1, \ldots, 5$.
(d) We flip at least four consecutive heads.