12.3 Telescope Scheduling

Large, powerful telescopes are precious resources that are typically oversubscribed by the astronomers who request times to use them. This high demand for observation times is especially true, for instance, for the Hubble Space Telescope, which receives thousands of observation requests per month. In this section, we consider a simplified version of the problem of scheduling observations on a telescope, which factors out some details, such as the orientation of the telescope for an observation and who is requesting that observation, but which nevertheless keeps some of the more important aspects of this problem.

The input to this ***telescope scheduling*** problem is a list, L, of observation requests, where each request, i, consists of the following elements:

- a requested start time, s_i, which is the moment when a requested observation should begin
- a finish time, f_i, which is the moment when the observation should finish (assuming it begins at its start time)
- a positive numerical benefit, b_i, which is an indicator of the scientific gain to be had by performing this observation.

The start and finish times for an observation request are specified by the astronomer requesting the observation; the benefit of a request is determined by an administrator or a review committee for the telescope. To get the benefit, b_i, for an observation request, i, that observation must be performed by the telescope for the entire time period from the start time, s_i, to the finish time, f_i. Thus, two requests, i and j, ***conflict*** if the time interval $[s_i, f_i]$, intersects the time interval, $[s_j, f_j]$. Given the list, L, of observation requests, the optimization problem is to schedule observation requests in a nonconflicting way so as to maximize the total benefit of the observations that are included in the schedule.

There is an obvious exponential-time algorithm for solving this problem, of course, which is to consider all possible subsets of L and choose the one that has the highest total benefit without causing any scheduling conflicts. We can do much better than this, however, by using the dynamic programming technique.

As a first step towards a solution, we need to define subproblems. A natural way to do this is to consider the observation requests according to some ordering, such as ordered by start times, finish times, or benefits. Start times and finish times are essentially symmetric, so we can immediately reduce the choice to that of picking between ordering by finish times and ordering by benefits.

The greedy strategy would be to consider the observation requests ordered by nonincreasing benefits, and include each request that doesn't conflict with any chosen before it. This strategy doesn't lead to an optimal solution, however, which we can see after considering a simple example. For instance, suppose we had a list containing just 3 requests—one with benefit 100 that conflicts with two nonconflicting

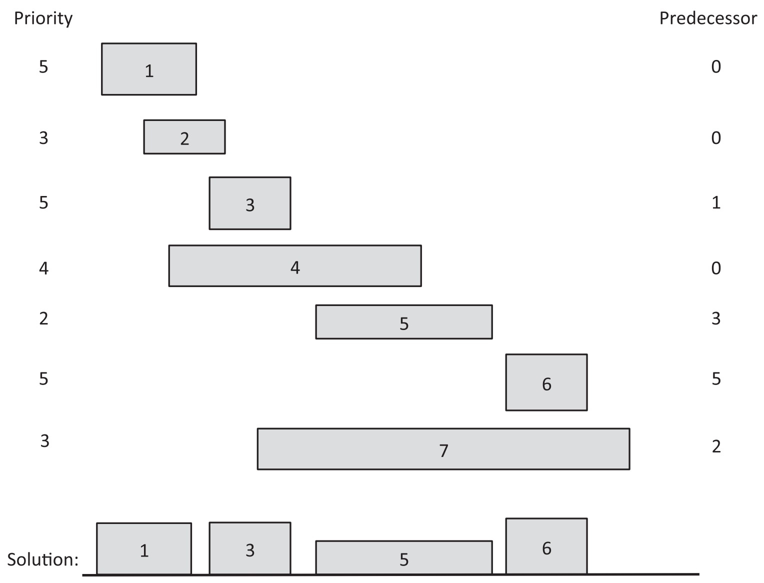

Figure 12.6: The telescope scheduling problem. The left and right boundary of each rectangle represent the start and finish times for an observation request. The height of each rectangle represents its benefit. We list each request's benefit on the left and its predecessor on the right. The requests are listed by increasing finish times. The optimal solution has total benefit 17.

requests with benefit 75 each. The greedy algorithm would choose the observation with benefit 100, in this case, whereas we could achieve a total benefit of 150 by taking the two requests with benefit 75 each. So a greedy strategy based on repeatedly choosing a nonconflicting request with maximum benefit won't work.

Let us assume, therefore, that the observation requests in L are sorted by nondecreasing finish times, as shown in Figure 12.6. The idea in this case would be to consider each request according to this ordering. So let us define our set of subproblems in terms of a parameter, B_i, which is defined as follows:

$B_i =$ the maximum benefit that can be achieved with the first i requests in L.

So, as a boundary condition, we get that $B_0 = 0$.

One nice observation that we can make for this ordering of L by nondecreasing finish times is that, for any request i, the set of other requests that conflict with i form a contiguous interval of requests in L. Define the ***predecessor***, $\mathrm{pred}(i)$, for each request, i, then, to be the largest index, $j < i$, such that requests i and j don't conflict. If there is no such index, then define the predecessor of i to be 0. (See Figure 12.6.)

The definition of the predecessor of each request lets us easily reason about the effect that including or not including an observation request, i, in a schedule that includes the first i requests in L. That is, in a schedule that achieves the optimal value, B_i, for $i \geq 1$, either it includes the observation i or it doesn't; hence, we can reason as follows:

- If the optimal schedule achieving the benefit B_i includes observation i, then $B_i = B_{\text{pred}(i)} + b_i$. If this were not the case, then we could get a better benefit by substituting the schedule achieving $B_{\text{pred}(i)}$ for the one we used from among those with indices at most $\text{pred}(i)$.
- On the other hand, if the optimal schedule achieving the benefit B_i does not include observation i, then $B_i = B_{i-1}$. If this were not the case, then we could get a better benefit by using the schedule that achieves B_{i-1}.

Therefore, we can make the following recursive definition:

$$B_i = \max\{B_{i-1},\, B_{\text{pred}(i)} + b_i\}.$$

Notice that this definition exhibits subproblem overlap. Thus, it is most efficient for us to use memoization when computing B_i values, by storing them in an array, B, which is indexed from 0 to n. Given the ordering of requests by finish times and an array, P, so that $P[i] = \text{pred}(i)$, then we can fill in the array, B, using the following simple algorithm:

```
$B[0] \leftarrow 0$
**for** $i = 1$ to $n$ **do**
    $B[i] \leftarrow \max\{B[i-1],\, B[P[i]] + b_i\}$
```

After this algorithm completes, the benefit of the optimal solution will be $B[n]$, and, to recover an optimal schedule, we simply need to trace backward in B from this point. During this trace, if $B[i] = B[i-1]$, then we can assume observation i is not included and move next to consider observation $i-1$. Otherwise, if $B[i] = B[P[i]] + b_i$, then we can assume observation i is included and move next to consider observation $P[i]$.

It is easy to see that the running time of this algorithm is $O(n)$, but it assumes that we are given the list L ordered by finish times and that we are also given the predecessor index for each request i. Of course, we can easily sort L by finish times if it is not given to us already sorted according to this ordering. To compute the predecessor of each request, note that it is sufficient that we also have the requests in L sorted by start times. In particular, given a listing of L ordered by finish times and another listing, L', ordered by start times, then a merging of these two lists, as in the merge-sort algorithm (Section 8.1), gives us what we want. The predecessor of request i is literally the index of the predecessor in L of the value, s_i, in L'. Therefore, we have the following:

Theorem 12.3: *Given a list, L, of n observation requests, provided in two sorted orders, one by nondecreasing finish times and one by nondecreasing start times, we can solve the telescope scheduling problem for L in $O(n)$ time.*