Assignment 21AIE212- DAA

Submitted By: Vikhyat Bansal [CB.EN.U4AIE21076]

Table of contents

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.

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.

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 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.

The second method is more general and is called an exchange argument. It involves considering all potential solutions to the issue and progressively converting them into the answer discovered by the greedy algorithm without compromising their quality.
Once more, it will be evident that the greedy algorithm has discovered a solution that is at least as effective as any other.

Interval Scheduling : The greedy algorithm stays ahead

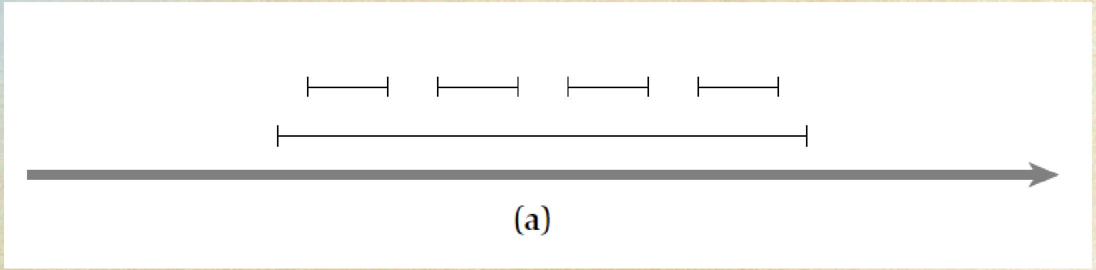
Question:

We have a set of requests {1, 2, . . . , *n*}; the *i*th request corresponds to an interval of time starting at *s(i)* and finishing at *f (i)*.
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.*

Basic Approach

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.

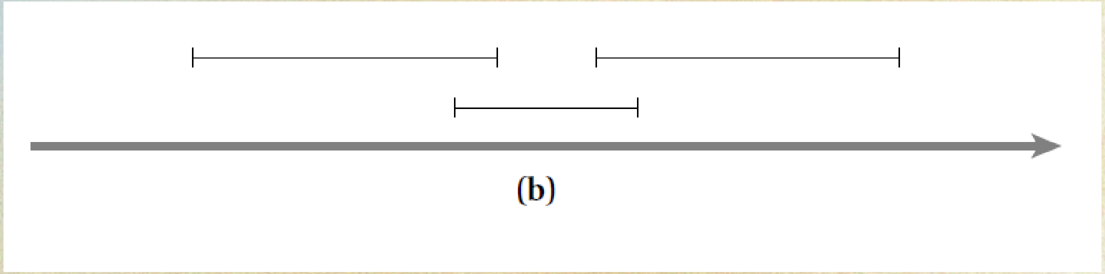
Some Approaches
Approach - 1


(a)

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 requests.

Approach is depicted in the image above.

Some Approaches
Approach - 2



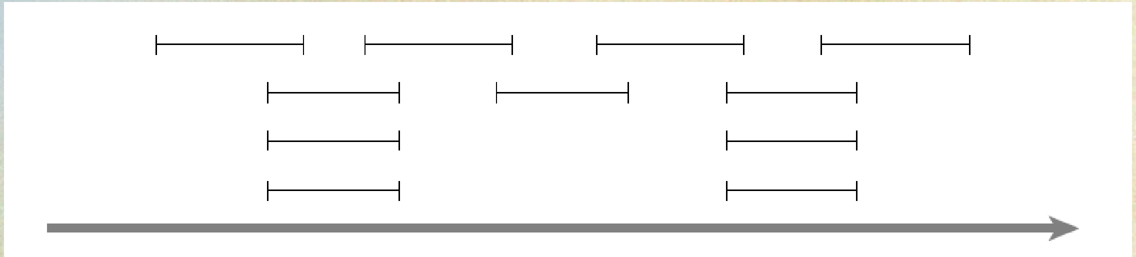
Above approach might suggest that we should start out by accepting the request that requires the smallest interval of time—namely, the request for which *f (i)* − *s(i)* is as small as possible. As it turns out, this is a somewhat better rule than the previous one, but it still can produce a suboptimal schedule.

From above image it can be seen that accepting the short interval in the middle would prevent us from accepting the other two, which form an optimal solution.

Approach is depicted in the image above.

Some Approaches

Approach - 3


(c)

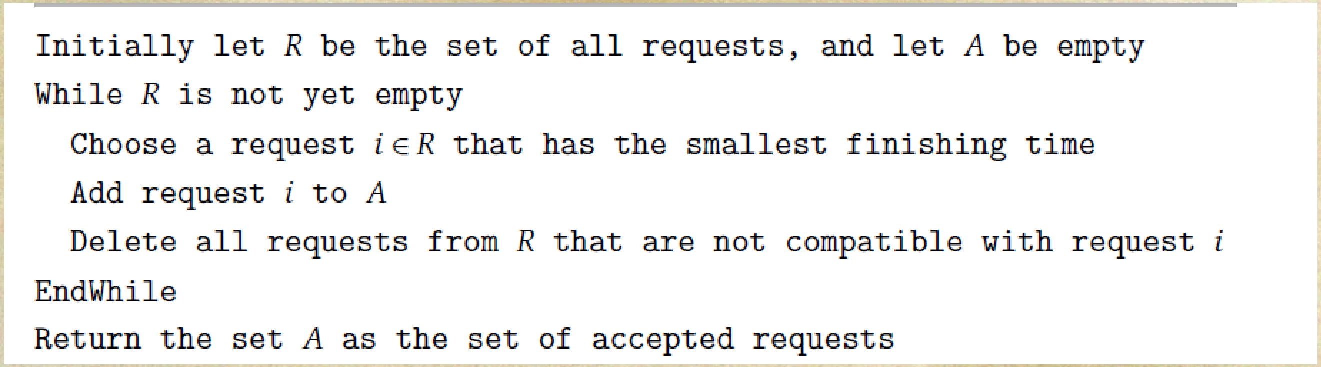
In the previous greedy approaches, our problem was that the second request competes with both the first and the third—that is, accepting this request made us reject two other requests. We could design a greedy algorithm that is based on this idea: for each request, we count the number of other requests 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.").

Contradictory example: The unique optimal solution of the above image 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.

Some Approaches
Approach - 4

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.

Simple Algo:



```
Initially let $R$ be the set of all requests, and let $A$ be empty
While $R$ is not yet empty
  Choose a request $i \in R$ that has the smallest finishing time
  Add request $i$ to $A$
  Delete all requests from $R$ that are not compatible with request $i$
EndWhile
Return the set $A$ as the set of accepted requests
```

Analyzing the algorithm

As a start, we can immediately declare that the intervals in the set *A* returned by the algorithm are all compatible.

What we need to show is that this solution is optimal. So, for purposes of comparison, let O be an optimal set of intervals. Ideally one might want to show that *A* = 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*| = |O|, that is, that *A* contains the same number of intervals as O and hence is also an optimal solution.

We introduce some notation to help with this proof. Let *i*1, . . . , *ik* 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 O be denoted by *j*1, . . . , *jm*. Our goal is to prove that *k* = *m*. Assume that the requests in 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.

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*)* ≤ *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* ≥ 1, the *r*th accepted request in the algorithm's schedule finishes no later than the *r*th request in the optimal schedule.

Proof of Correctness

a) *For all indices $r \le k$ we have $f(i_r) \le f(j_r)$.*

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.

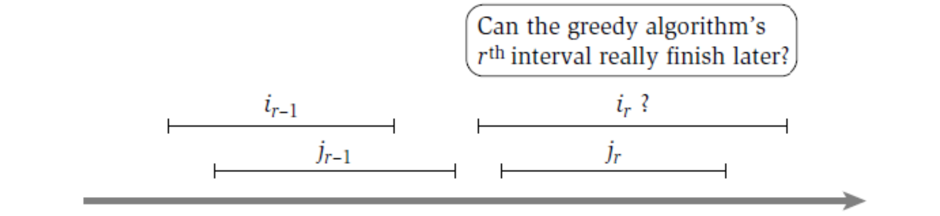

Can the greedy algorithm's r^{th} interval really finish later?
i_{r-1}
i_r ?
j_{r-1}
j_r

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

The induction hypothesis lets us assume that $f(i_{r-1}) \le f(j_{r-1})$. In order for the algorithm's r_{th} interval not to finish earlier as well, it would need to "fall behind" as shown. But there's a simple reason why this could not happen: rather than choose a later-finishing interval, the greedy algorithm always has the option (at worst) of choosing j_r and thus fulfilling the induction step.

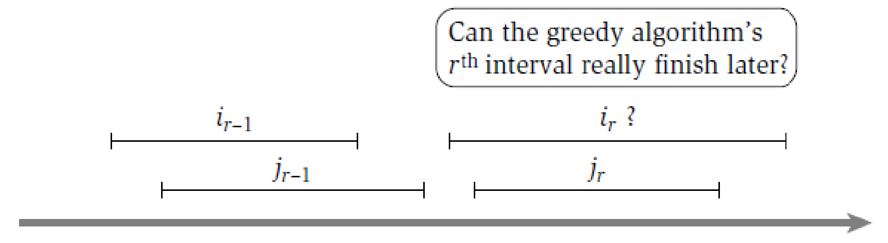
We know (since O consists of compatible intervals) that $f(j_{r-1}) \le s(j_r)$. Combining this with the induction hypothesis $f(i_{r-1}) \le f(j_{r-1})$, we get $f(i_{r-1}) \le s(j_r)$. Thus the interval j_r is in the set R of available intervals at the time when the greedy algorithm selects i_r .

The greedy algorithm selects the available interval with *smallest* finish time. Since interval j_r is one of these available intervals, we have $f(i_r) \le f(j_r)$. This completes the induction step.

Proof of Correctness

b) *The greedy algorithm returns an optimal set A.*

- 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 above proof (a) with $r = k$, we get that $f(i_k) \le f(j_k)$. 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.



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

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

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 $S[1 \dots n]$ 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

- 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.

- 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 *vi* (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*

Scheduling All Intervals

Question:

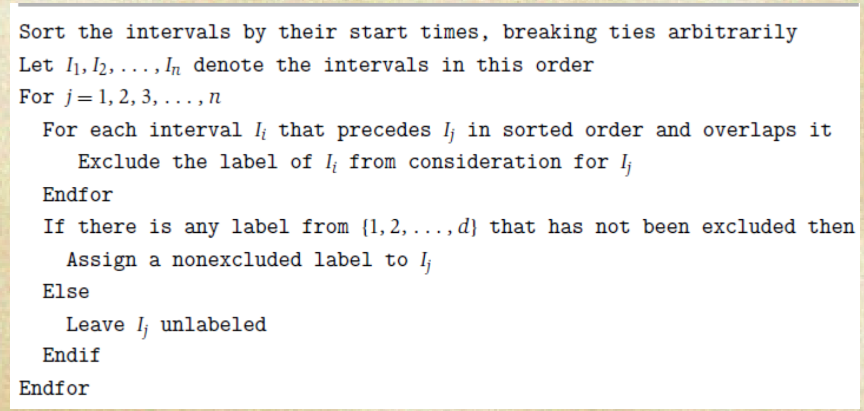
In the Interval Scheduling Problem, there is a single resource and many requests in the form of time intervals, so we must choose which requests to accept and which to reject. A related problem arises if we have many identical resources available and we wish to schedule *all* the requests using as few 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.

Basic Approach

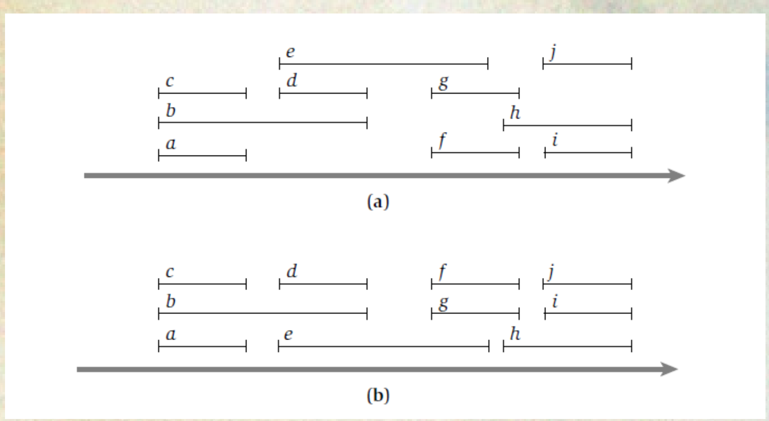
In general, one can imagine a solution using *k* resources as a rearrangement of the requests into *k* rows of nonoverlapping intervals: the first row contains all the intervals assigned to the first resource, the second row contains all those assigned to
the second resource, and so forth.

Analyzing 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, . . . , *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.



```
Sort the intervals by their start times, breaking ties arbitrarily
Let $I_1, I_2, \ldots, I_n$ denote the intervals in this order
For $j = 1, 2, 3, \ldots, n$
  For each interval $I_i$ that precedes $I_j$ in sorted order and overlaps it
     Exclude the label of $I_i$ from consideration for $I_j$
  Endfor
  If there is any label from $\{1, 2, \ldots, d\}$ that has not been excluded then
    Assign a nonexcluded label to $I_j$
  Else
    Leave $I_j$ unlabeled
  Endif
Endfor
```



Proof of Correctness

a) *If we use the greedy algorithm above, every interval will be assigned a label, and no two overlapping intervals will receive the same label*

First let's argue that no interval ends up unlabeled. Consider one of the intervals *Ij*, and suppose there are *t* intervals earlier in the sorted order that overlap it. These *t* intervals, together with *Ij*, form a set of $t + 1$ intervals that all pass over a common point on the time-line (namely, the start time of *Ij*), 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 *Ij*.

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*.

THANK YOU !!