CSE347 Lecture 2

Greedy Algorithms I: scheduling

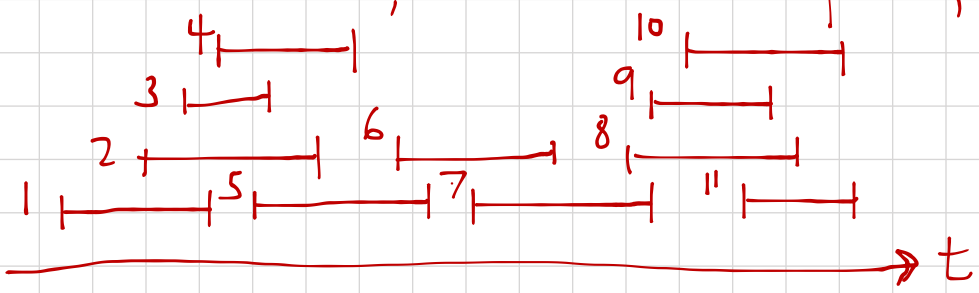
1. Problem: Interval Scheduling
2. Greedy Algorithms
3. Analysis — "greedy stays ahead"
4. Variant: scheduling all intervals — a structural analysis

Many of the problems we desire to solve with algorithms are optimization problems. These problems are described by two parts: constraints that specify legal solutions/structures, and an objective that assigns a cost or reward to each solution. For example, today we will consider "scheduling" problems: there is a resource — a room, a computer, a catering crew, etc. — and there is a set of requests to use the resource. These requests (in our current variant) have the form, "can I reserve [the resource] starting at time s until time f?" Naturally, we can only allocate these resources to one requester at a time, and two requests starting at times s_1 & s_2, and finishing at times f_1 & f_2 are compatible if the intervals $[s_1, f_1)$, $[s_2, f_2)$ do not overlap, i.e., $[s_1, f_1) \cap [s_2, f_2) = \phi$; equivalently, $s_2 \geq f_1$ or $s_1 \geq f_2$. More generally, a subset A of the requests

$[s_1, f_1), [s_2, f_2), \ldots, [s_n, f_n)$ is compatible if every pair $[s_i, f_i)$ and $[s_j, f_j)$ $(i \neq j)$ in A is compatible.

A legal solution to this scheduling problem is a compatible subset of the requests (which we grant). Our objective is the size of this subset — we want to find a subset that is as large as possible

Consider, for example, the following set of requests



What is the largest set of requests we can grant?

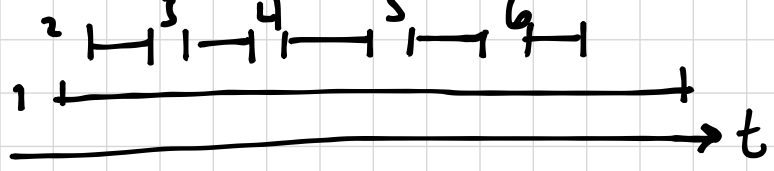
{1, 5, 7, 11}, which has size 4 (or {1, 5, 7, 10})

As is common, we could solve this problem by iterating over all subsets, checking to see if they are compatible, and taking the largest set. Such an algorithm may consider all or most of the 2^n subsets, and is thus impractical unless n is quite small — notice, even this small problem has >2000 candidate solutions. We'll see that a simple kind of algorithm is much better.

Greedy Algorithms

Roughly, a greedy algorithm is one that builds up a candidate solution by making a series of small steps that optimize some objective, without regard for how these choices will impact our future options. There may be many objectives we could optimize

for a given problem, and hence many possible greedy algorithms. For example, one greedy objective for scheduling is to pick the job that starts first; the corresponding greedy algorithm then builds its set A by choosing the request with the smallest s_i that is compatible with all of the intervals chosen so far. Naturally, this can fail badly, e.g.,

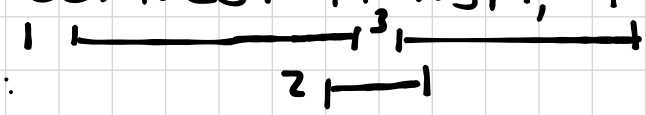


we can schedule {2, 3, 4, 5, 6}, but this greedy algorithm chooses {1}.

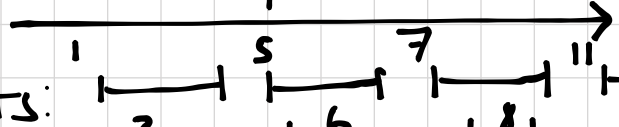
What are some other greedy objectives?

E.g., shortest job, fewest conflicts, earliest finish

Except for earliest finish, these have fatal flaws—e.g.,



shortest job:



takes {2}, not {1, 3}

fewest conflicts:

takes one each from {1,2,3,4} + {8,9,10,11} along with 6, whereas {1,5,7,11} is larger.

But, as we'll see, earliest finish indeed produces an optimal choice of requests. In rough pseudocode:

Input: a set of requests $R = \{[s_1, f_1), [s_2, f_2), \ldots, [s_n, f_n)\}$

```
Initialize A ← ϕ.
While R ≠ ϕ
    [s_i, f_i) ← [s_i, f_i) ∈ R with minimum f_i
    Add [s_i, f_i) to A.
    Remove all [s_j, f_j) conflicting with [s_i, f_i) from R.
```

Return A.

Analysis — "the greedy algorithm stays ahead"

We will see multiple ways of analyzing greedy algorithms (to show correctness). It is clear by design that the algorithm chooses a set of compatible requests — properly, if a pair of requests has a conflict, then if either of them is ever added to A, then the other is immediately removed from R, so it will not also be added to A.

What is less clear is why such a blind choice does not eliminate some other (much) better schedule.
Note: we can't even hope to show that A is the only optimal schedule, since there could be more than one equally good schedule (cf. our first example).
What we can show is that given any arbitrary set of compatible requests, the first r requests chosen by the earliest-finish algorithm finish at least as soon as the first r requests in the other set — therefore, if the other set includes an r+1th request, there is some r+1th request that earliest-finish can add, too.
It follows that earliest-finish finds a set of requests that is the same size or larger.

More precisely, suppose A contains requests $i_1, \ldots, i_k$

and the other set O contains $j_1, \ldots, j_m$ ordered by start/finish.

Lemma for all $r \le k$, $f_{i_r} \le f_{j_r}$

Proof by induction on r. For $r=1$, earliest-finish chooses i_1 to minimize f_{i_1} among all requests in R, so $f_{i_1} \le f_{j_1}$. ✓

Induction step: We know already that $f_{i_{r-1}} \le f_{j_{r-1}}$.

Since O is a set of compatible requests, $f_{j_{r-1}} \le s_{j_r}$!

Notice: since $f_{i_1} \le f_{i_2} \le \cdots \le f_{i_{r-1}} \le f_{j_{r-1}} \le s_{j_r}$, none of $i_1, \ldots, i_{r-1}$ conflict with j_r. Therefore, j_r is among the requests that are considered in the choice of i_r; since f_{i_r} is a minimum, $f_{i_r} \le f_{j_r}$ also. ▨

Theorem Earliest-finish returns an optimal set A.

Proof: Suppose for contradiction that $k < m$. By the lemma, we know that $f_{i_k} \le f_{j_k}$, where $f_{i_1} < f_{i_2} < \cdots < f_{i_k}$ and $f_{j_k} \le s_{j_{k+1}}$. Therefore j_{k+1} is compatible with A and would not be removed from R after (only) adding $i_1, \ldots, i_k$. Therefore, earliest-finish would have added a $k+1^{th}$ set, contradicting the definition of k. ✳

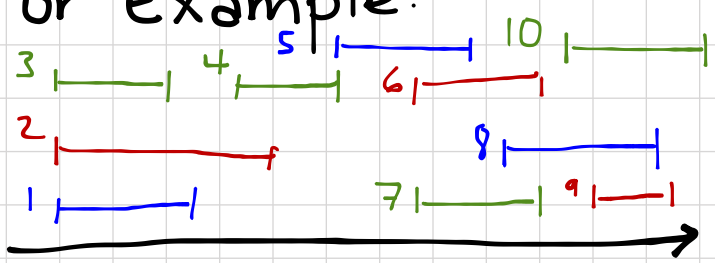
Running time: we must flesh out the pseudocode a little bit more to discuss the running time. Suppose we start by sorting R by finish time — this can be done in time $O(n \log n)$ (using, e.g., MergeSort). Now, we initialize $t=0$, and walk the list until we find a request i with $s_i \ge t$ — we only implicitly filter out the requests that conflict with our current set by skipping

over them. We then add this i to A, and put $t \leftarrow f_i$. Note that this maintains the invariant that, since all of the later requests have $f_j \geq f_i$, if $s_j < t = f_i$, then $f_i \in (s_j, f_j]$ so i & j have a conflict, whereas if $s_j \geq t$, then since $f_1 \leq \cdots \leq f_i = t \leq s_j$, i & j do not have a conflict. So, this implementation is correct and runs in time $O(n \log n)$.

Variant — Scheduling All Intervals

Here is a different kind of scheduling problem. Suppose we need to grant all of the requests, and we are seeking to minimize the number of rooms/computers/crews we use in doing so. A solution of the problem is now a partition of the set of requests R into sets $I_1, \ldots, I_d$ such that (1) no I_j contains a conflict and (2) d is as small as possible. I_j is then the schedule for the jth resource.

For example:



$I_1 = \{1, 5, 8\}$
$I_2 = \{2, 6, 9\}$
$I_3 = \{3, 4, 7, 10\}$

These requests can be served with d=3

Can we achieve $d < 3$? No — 1, 2, and 3 mutually conflict (at some common time) so they need different partitions. More generally, we may define the depth of a set of requests to be the maximum number that include any single common point of time. By the same argument, we have

Lemma: To schedule all requests, a number of resources at least as large as the depth is needed.

We will give a greedy algorithm that finds a partition of size equal to the depth. By this observation, it's impossible to do better, so the greedy algorithm will be optimal.