Greedy Algorithms (Chapter 4)

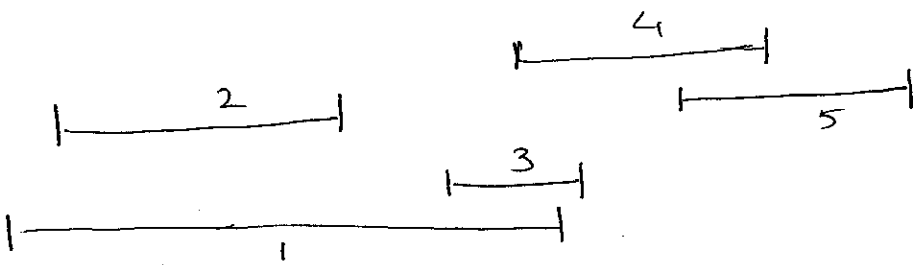
We now explore some ~~prob~~ optimization problems for which the space of solutions is huge. In some such situations, simple greedy heuristics efficiently find an optimal solution. We'll focus on developing the skill of understanding ~~when greedy algorithms~~ whether a greedy algorithm works or not.

Let us start with the interval scheduling problem of Chapter 4.1. We are given a set $\{1, 2, \ldots, n\}$ of requests, each of which is an interval with a start time s_i and

finish time f_i.

A subset of the requests are said to be <u>compatible</u> if no two of them overlap.

The problem then is to find the largest subset of compatible requests.



In the above example, the optimal solution is $\{2, 3, 5\}$.

By a greedy algorithm in this context, we mean a rule to pick one interval. We apply this rule to

pick one interval from our problem instance. We delete the picked ~~and~~ interval and the intervals that overlap with the picked interval. We recurse on the remaining set of intervals.

Here are some possible rules:

1. Pick the interval that starts first.
2. Pick the shortest interval.
3. Pick the interval that overlaps with the least number of intervals.
4. Pick the interval that finishes first.

While some of these rules are motivated by reasonably good intuition, we discover ~~that~~ after some work that there are instances where 1, 2,

and 3 don't work.

4 seems to work in all the examples we work out. Does it always work? Or is it just harder to find a counterexample?

This is why it is important to <u>prove</u> that rule 4 works. Before that, let us state the algorithm it leads to in a convenient form:

$R \leftarrow$ All requests; $A \leftarrow \phi$

While $(R \neq \phi)$ do

  Choose $i \in R$ with smallest finish time.

  Add i to A.

  Delete from R i and all requests that overlap with i.

endwhile

Return A.

<u>Observation 1.</u> A is a compatible set of requests.

This does not require much of a proof.

Suppose $i_1, i_2, \ldots, i_k$ are requests in A ordered by start time (or finish time, which yields the same order because of compatibility).

Let $j_1, j_2, \ldots, j_m$ be requests in optimal solution O, similarly ordered. We know that $m \geq k$ since O is optimal.

Our goal is to show that $m = k$, and thus A is also optimal.

The key observation is the following. It states that A always "stays ahead" of O. We prove it by induction.

Observation 2:

For each $r \leq k$, we have

$$f_{i_r} \leq f_{j_r}.$$

<u>Proof</u>: By Induction, in class. See Textbook otherwise.

We now show that $k \geq m$. Since we know $m \geq k$, this means $m = k$.

<u>Observation 3</u>: $k \geq m$.

<u>Proof</u>: Suppose $m > k$. By Observation 2, $f_{i_k} \leq f_{j_k}$. Now $s_{j_{k+1}} \geq f_{j_k}$, and so $s_{j_{k+1}} \geq f_{i_k}$. Thus j_{k+1} does not overlap with $i_1, \ldots, i_k$ and is therefore contained in R after the while loop in which i_k is picked by our algorithm. This is a contradiction,

since algorithm terminated after iteration of while loop in which i_k is picked, thus implying $R = \phi$. ☒

<u>Implementation of Algorithm</u>:

Sort requests by increasing finish time.

~~Pick first request, then ~~scroll scroll~~ over all ~~requests~~ ~~into~~ subsequent~~

Pick first request, go through sorted array till we find a request that starts after the first one ends, make that the second request picked, go through following elements of sorted array till we find a request that starts after the second picked one ends, pick that as third request, and so on.

This implementation is somewhat different from the algorithm as stated - when we pick a request, we do not immediately "delete" all requests that overlap with it, but we will "delete" them eventually.

The implementation runs in $O(n \log n)$ time.