ASCII WEEK 3

Jacob Ma

September 28, 2022

1 Greedy Algorithm

Definition 1.1: Greedy Algorithm for Interval Selection

- Sort Interval by non-decreasing finish times $f(i)$
- Start with remaining interval R = all jobs, accepted intervals $A = \{\}$
- while R! = $\{\}$
  - Let i be a remaining interval with earliest finish time (smallest index)
  - Add i to A
  - Remove from R the interval i and all intervals intersecting it.
- output A

Lemma 1.1.1: Does this always output a *legal* solution?

Yes. Overlaps are impossible, because whenever an interval is picked, everything intersecting it is deleted.

Property 1.2: Running Time:

Sorting: $O(n \log n)$
Start with intervals: $O(1)$ or maybe $O(n)$
TO implement the loop in $O(n)$, use that the array is now sorted.
Run once through the array from $1, 2, ..., n$
In each iteration, if the current intervals starts after the most rec only picked one finishes, then pick this one.
Otherwise, skip it.
$\implies$ Use heavily that we sorted by finish time.
Total: $O(n \log n)$

Problem 1: Why do we think this greedy algorithm is optimal?

High-level intuition: The algorithm picks things that end as early as possible, so it can pick more things.
Slightly more precisely: For any number k of intervals picked, greedy ensures to finish as early as possible.
Precise formulation: Let $i(1) < i(2) < ... < i(k)$ be the first k intervals picked by greedy, and $j(1) < j(2) < ... < j(k)$ k intervals without overlap we compare against.
Then, $f(i(k)) \leqslant f(j(k))$.
$\Longrightarrow$ The Greedy Algorithm stays ahead.

Proof. Proof Idea: Induction on k (number of intervals picked)

Definition 1.3: Invariant

When you prove that a program is correct, you often call your induction statement / hypothesis a *program invariant* or just *invariant*.
Specially when you prove something about a loop, you often call the induct statement *loop invariant*.

Base Case: $k = 1$
Greedy picks the one with smallest $f(i)$ overall, so $f(i(1)) \leqslant f(j(1))$.
Alternative Base Case: $k = 0$
No intervals picked $\Longrightarrow$ vacuously true.
Induction Step $k \to k+1$: Assume statement is true for k (or up to k in strong induction), and prove that it is also true for $k+1$. We get to assume $f(i(k)) \leqslant f(j(k))$
We want to prove that $f(i(k+1)) \leqslant f(j(k+1))$
Because j was a legal solution (no overlaps), $s(j(k+1)) > f(j(k))$.
By Induction Hypothesis, $f(j(k)) \geqslant f(i(k))$, so $s(j(k+1)) > f(i(k))$.
$\Longrightarrow$ The interval $j(k+1)$ is legal to add the i solution.
$\therefore$ because greedy picks the best available, and $j(k_1)$ is available, greedy picked $j(k+1)$ or something that finished earlier.
$\Longrightarrow$ $f(i(k+1)) \leqslant f(j(k+1))$
This finished the induction step □

Now return to our "real goal": showing that Greedy selects as many intervals as possible.

Proof. Proof by contradiction. Assume that there is another solution that has more intervals - let's say greedy has t, and the other solution has $t+1$. Greedy has $i(1), ..., i(t)$, and the other solution $j(1), ..., j(t+1)$
By what we just proved using induction, $f(i(t)) \leqslant f(j(t))$, so $j(t+1)$ was available to greedy.
So greedy would not have terminated $\Longrightarrow$ Contradiction.
So greedy has a largest number of intervals. □

Variations/Extensions:

- dependencies between intervals (e.g. lecture pre-requirements)

- values/priorities/weights of events
- distances between events
- times are flexible
- multi-person coordination
- multiple people/machines
- you don't known all of the input head of time $\Longrightarrow$ stochastic optimization or online algorithms

Definition 1.4: Scheduling with Deadlines

Givn n jobs, each with a duration/time $t(i)$ and deadline $d(i)$.
You are not told when you need to do specific jobs. All jobs must be performed. We are worried abouit jobs being late.
Lateness of job i $L(i)$ is its finish time minus its deadline (or 0 if it finishes before the deadlie $\Longrightarrow$ no early finish bonus).

Solution

Many mnatural ways to aggreagte $L(i)$ across jobs (sums, weighted sums, penalities,...)
Here, we will study *maximum* lateness: Goal is to minimze $\max_i L(i)$
Observation: Because all jobs are available from the start, it never helps to split up a job - if you puhnhsied it all to the end, it would finish at the same time, and other jobs woiuld finish ealier.
(This would not be true any more if jobs had release times, so that they onluy become available midway through.)
$\Longrightarrow$ From now on, only look at solutions that do not split up jobs.
Greedy Rules:

(1) Longest job first.

If we have short jobs with early deadlines, this is a problem.

Example: $(t:10, d:100), (t:1, d:1)$

(2) Shortest job first.

Example: $(t:10, d:10), (t:1, d:100)$

(3) Least "slack" $d(i) - t(i)$

Example: $(t:10, d:10), (t:1, d:2)$

(4) Earliest deadline first: sort the jobs from most urgent to least urgent (ignore durations)

$\Longrightarrow$ This one acrtually works, and we will analyze it.

Intuition for why this algorithm *should* work:

(1) An optimal solution should never contain an idle time

$\Longrightarrow$ From now on, focus on solutions wihtout idle time.

$\Longrightarrow$ Solution is completely characterized by the order in which jobs are

(2) If a solution were not greedy, then we could perform a swap and make it no worse. This swao should make it more similar to Greedy.

Greedy Solution: $1, 2, 3, ..., n$

OPT solution: something different

We want to perform swaps on the optimum solution to make it *more similar* to the greedy solution, while not increasing the maximum lateness with the swap.

That way, if we keep swapping, eventually, the OPT will be equal to greedy, and we have shown that greedy is infact optimal.

If OPT is not equal to $(1, 2, 3, ..., n)$, is there an *adjacent* pair that is out of order?

Yes - this can be proved with a simple induction proof.

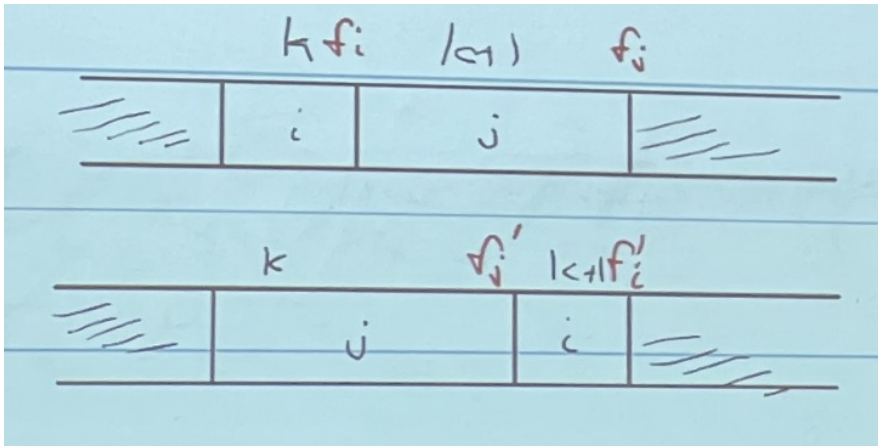
So we focuas on such pair:

In the OPT solution, the two jobs in positions $k, k+1$ are out of ordcer compared to the greedy solution, where they are in positions i, j, with $j < i$.

What happens if we swap these two jobs?

$f_i' = f_j$

or all other jobs, the finish time (and thus lateness) stays the same.



Swapping i and j

Lateness: $L_j' = f_j' \leqslant f_j - d_j = L_j$

$L_i' = f_i' - d_i = f_j - d_i \leqslant f_j - d_j = L_j \leqslant$ max lateness

Because the lateness of all other jobs stay the same, and the new lateness of both i and j is at most the old lateness of j, the maximum lteness did not increase with this swap.

$\Longrightarrow$ The modified optimial soluiton is at least as good as the old one.

By repreating this, we eventually get a solution that must be equal to the greedy one, shwoing that greedy's maximum lateness is no more than OPT's.

Definition 1.5: Kendall Tau Distance

Notion of distance from OPT to greedy is the total number of pairs (adjacent or not) that are out of order. This is called Kendall tau distance between orders.

Each swap fiexs exactly one pair (and cannot creat new ones), so this distance decreases by one with each adjacent swap. We could do a formal proof by induction on this Kendall tau distrance - the inducaito setp would be our swaps.

Why did we start swapping at OPT, rathewr than at the greedy soltuion.

If we proved that every swap from the greedy solution makes it no beteer, we would have shown that greedy is *locally optimal*, but not cesessarily *globally optimal*. It could be theat any one swqp makes things worse, but along sequence of multiple swaps results in an improvment.