CIS 320: Algorithms October 6, 2021

Lecture 10

Lecturer: Aaron Roth *Scribe: Aaron Roth*

Greedy Algorithms I: Unweighted Interval Scheduling

We've studied several frameworks for coming up with polynomial time algorithms: divide and conquer, and dynamic programming. These frameworks both look for some kind of high-level recursive structure to take advantage of: if we can solve the *right* sub-problems optimally, we can stitch them together. But it seems tempting sometimes just to *go for it* — to start constructing the solution right away, with choices that seem reasonable, without having a theory of how they will fit together with the rest of the solution. Algorithms like this are termed *greedy*, and often have the advantage that they are very fast. But when do they work? In the next several lectures, we'll examine different kinds of problem structures that allow such algorithms to find optimal solutions. We'll start with the "unweighted" version of the interval scheduling problem we considered previously. The goal is to schedule *as many* non-intersecting jobs as possible — i.e. it is the weighted interval scheduling problem in the special case where all weights are identical.

Definition 1 *An instance of the unweighted interval scheduling problem is given by a collection* $I = \{s(i), f(i)\}_{i=1}^{n}$ *of* n jobs $(s(i), f(i))$. *Each job has a real valued* start time $s(i)$, *and* finish time $f(i)$. *Think of a job as representing an interval of utilization of some resource spanning* $[s(i), f(i)]$

A feasible solution *to the unweighted interval scheduling problem is a subset* $S \subseteq [n]$ *of jobs such that for each* $i \in S$, *the intervals* $[s(i), f(i)]$ *are non-overlapping. The goal of the (unweighted) interval scheduling problem is to find a feasible solution* $S \subseteq [n]$ *of maximal cardinality* $|S|$.

What should a "greedy" algorithm look like? The premise is that we should always pick the next interval subject to our current feasibility constraints, in a way that seems myopically to be "best", but its not clear how we should evaluate what is best. Here are two proposals that do not work (you should be able to come up with simple examples demonstrating why):

1. Always pick the next compatible job i that has the earliest start time $s(i)$.
2. Always pick the next compatible job i that has the shortest duration $f(i) - s(i)$.

But here is a proposal that seems pretty good:

Always pick the next compatible job i *that has the earliest finish time* $f(i)$.

At least its hard to think of a counter-example. Can we prove that this always arrives at the optimal solution?

First we recall the definition of a compatible job:

Definition 2 *A job* i *is compatible with a job* j *if the intervals* $[s(i), f(i)]$ *and* $[s(j), f(j)]$ *are non-overlapping. A job* i *is compatible with a collection of jobs* S *if* i *is compatible with every* $j \in S$. *A collection of jobs* S *is feasible if for each* $i, j \in S$, i *and* j *are compatible.*

Here is our proposed algorithm:

Algorithm 1 GreedyInterval(I)

Let $S = \emptyset$, $R = [n]$.
while R is not empty **do**
  Choose the job i with smallest finishing time $i = \arg\min_{i \in R} f(i)$.
  Add i to S and **Remove** i from R.
  Delete all jobs in R that are incompatible with i.
end while
Return S.

We can first observe that by construction, our algorithm returns a feasible set S:

Claim 3 *For any instance I, GreedyInterval(I) returns a feasible solution S.*

Our goal will then be to show that the solution S it returns is optimal: namely, for any feasible solution O, $|S| \geq |O|$. Our plan of attack is to show that for any optimal solution O, the solution S that our algorithm is building up is always "staying ahead" of O. We will show that the intermediate solutions S produced by our algorithm are always "better" than an appropriately defined prefix of O, and as a result, we will show that the final output S is at least as large as O.

To carry out this argument, lets introduce some notation. Suppose our algorithm outputs a set S of size $|S| = k$. It consists of k jobs $S = \{i_1, \ldots, i_k\}$ that we without loss of generality number in the order in which they are added to S — i.e. i_1 is the first job that was added to S, i_2 is the 2nd, and so forth. Now consider an optimal solution O of size $|O| = m$. We write $O = \{j_1, \ldots, j_m\}$, where without loss of generality, we number the jobs in their natural left-to-right ordering. Note that because O is feasible, the ordering of the start times is the same as the ordering of the finish times. So j_1 starts and finishes before j_2, which starts and finishes before j_3, and so on. Our goal is to show that $k = m$.

The intuition for our greedy rule is that we want our resource to become free as soon as possible after we satisfy the next request. We will formalize this by showing that the set S selected by our algorithm "stays ahead" of O in the sense that for all r, $f(i_r) \leq f(j_r)$. Thus our algorithm always has at least as many options for selecting the next job as the optimal solution would have had during its construction.

Lemma 4 *For all indices $r \leq k$, $f(i_r) \leq f(j_r)$.*

Proof We prove this by induction. The base case is $r = 1$, and holds by definition, since we choose $i_1 = \arg\min_j f(j)$.

For the inductive case, we let $r > 1$ and assume that our induction hypothesis is true for $r - 1$: $f(i_{r-1}) \leq f(j_{r-1})$. Since O is feasible, we know that $f(j_{r-1}) \leq s(j_r)$. By our inductive hypothesis, we therefore also have that $f(i_{r-1}) \leq s(j_r)$. Therefore, at round r of our algorithm, $j_r \in R$ is still in the set of feasible intervals. Since our algorithm chooses $i_r \in \arg\min_{i \in R} f(i)$, we must therefore have that $f(i_r) \leq f(j_r)$, which is what we want. ■

We're almost done. All that remains is to show that the above Lemma leads us to a contradiction if in fact $k < m$.

Theorem 5 *For any instance I, GreedyInterval(I) returns an optimal solution S.*

Proof Suppose otherwise: that is, GreedyInterval returns $S = \{i_1, \ldots, i_k\}$, but an optimal solution $O = \{j_1, \ldots, j_m\}$ is such that $m > k$. By Lemma 4, we have that $f(i_k) \leq f(j_k)$. Since O is feasible, we have that $s(j_{k+1}) \geq f(j_k) \geq f(i_k)$. Therefore, after i_k is added to S, j_{k+1} remains compatible with S, and hence $j_{k+1} \in R$. But this contradicts the fact that the algorithm has halted and output S, since the algorithm only halts when R is empty. ■

And what about the run-time? If the jobs are already sorted in ascending order of their finishing time, then the algorithm can be implemented in time $O(n)$ by taking a single pass through the sorted list of jobs. If they are not yet sorted, we can sort them in time $O(n \log n)$, obtaining total run time $O(n \log n)$

Given a sorted list of jobs, we can implement the algorithm in a single pass as follows. We always select the first job $i_1 = 1$. Then, given that we have selected jobs $i_1, \ldots, i_{k-1}$ we keep iterating through the sorted array until we find the first job such j that $s(j) \geq f(i_{k-1})$. When we do so, we select it: $i_k = j$, and continue.

There is another closely related problem that we can also solve with a greedy algorithm. Suppose we have more than one resource (e.g. computer), and we *must* schedule all of the jobs, using multiple resources if necessary. Given an instance I of the interval scheduling problem, we would like to schedule all jobs by assigning each to resources such that:

1. The jobs scheduled to each resource are feasible (non-overlapping), and

2. We use as few resources as possible.

This is called the *interval partitioning problem*, since we can view it as the problem of dividing the jobs amongst different resources such that none of the intervals assigned to a single resource are overlapping.

Definition 6 *An instance of the interval partitioning problem is given by a collection I of jobs $\{(s(i), f(i)\}_{i=1}^{n}$. A feasible solution to I is a partitioning $S_1, S_2, \ldots, S_k \subseteq [n]$ of the jobs such that each set S_i consists of non-overlapping intervals. The goal of the interval partitioning problem is to find a solution that minimizes the number of partition elements k.*

To get a handle on what an optimal solution might look like, it is helpful to have the concept of the *depth* of an instance.

Definition 7 *The depth of an instance I, written $d(I)$, is the maximum number of intervals in I that overlap a single point. In other words, it is:*

$$d(I) = \max_x |\{i : s(i) \leq x \leq f(i)\}|$$

It is straightforward to see that:

Lemma 8 *In any feasible solution to an interval partitioning instance I, $k \geq d(I)$.*

This is because there is a point x such that $|\{i : s(i) \leq x \leq f(i)\}| = d(I)$, and every job in this set must be assigned to a different resource.

However, there is a simple greedy algorithm is able to achieve this bound, and hence is optimal. It attempts to assign a label from $1, \ldots, d(I)$ to each job, which can be viewed as partitioning the jobs into $d(I)$ parts.

Algorithm 2 GreedyIntervalScheduler(I)

Sort the intervals in ascending order by start time, denoted as $I_1, \ldots, I_n$.
for $j = 1$ to n **do**
  For each interval I_i that comes earlier than I_j in sorted order and overlaps it, exclude I_i's label from consideration for I_j.
  If there are any remaining labels in $\{1, \ldots, d(I)\}$ that have not been excluded, assign such a label to I_j.
end for

Theorem 9 *On any instance I, GreedyIntervalScheduler(I) assigns every job a label in $\{1, \ldots, d(I)\}$, and no two jobs assigned the same label overlap.*

Proof First we argue that every job is assigned a label. Consider any job I_j. Since the jobs are sorted in order of start time, there can be at most $d(I) - 1$ preceding intervals I_i that overlap it, since otherwise there would be more than $d(I)$ intervals overlapping the point $x = s(I_j)$, contradicting the definition of depth. Thus there is a remaining label in $\{1, \ldots, d(I)\}$ that has not been excluded, which is assigned to I_j. Moreover, any two intervals I_i and I_j assigned to the same label do not overlap. Without loss of generality, assume that $i < j$. Then if I_i overlaps with I_j, then when job j is under consideration, by construction, I_i's label will be excluded. ■

Because we know that *any* feasible solution must use at least $k \geq d(I)$ many resources, and we have an algorithm that finds a feasible solution that never uses more than $k \leq d(I)$ many resources, we know that our algorithm must be optimal.