COMPSCI 330: Design and Analysis of Algorithms January 31

Lecture 7: Greedy Algorithms I

Lecturer: Rong Ge *Scribe: Chenwei Wu*

1 Overview

In this lecture, we will talk about greedy algorithms. Greedy algorithm is a way to break a large, complicated problem into smaller sub-problems. However, the way of breaking the problems in a greedy algorithm is different from those of divide and conquer and dynamic programming.

Specifically, if a problem requires to make a sequence of decisions, then in greedy algorithm, we always make the "best" choice for the first decision given the current situation. This automatically reduces the problem to a smaller sub-problem which requires making one fewer decisions.

Thus, to design a greedy algorithm, we need to decide what is a "good" choice for a problem. A good choice for a problem always leads us to the optimal solution. For a specific problem, there may be several intuitive choices that make some sense. However, not all of them are good choices. The way to distinguish good and bad choices is to find counterexamples or proofs. If we can find a counterexample for a choice, then it is a bad choice. If we can prove the correctness of a choice, then it is a good choice.

Although easy to devise, greedy algorithms can be hard to analyze. The correctness of a greedy algorithm is often established via proof by contradiction, and that is always the most difficult part for designing a greedy algorithm. In this lecture, we will demonstrate greedy algorithms for solving *interval scheduling* problem and prove its correctness.

2 Interval Scheduling

2.1 Problem Statement

Suppose there are n meeting requests, and meeting i takes time (s_i, t_i)—it starts at s_i and ends at t_i. The constraint is that no two meeting can be scheduled together if their intervals overlap. Our goal is to schedule as many meetings as possible.

Example 1. Suppose we have meetings $(1,3), (2,4), (4,5), (4,6), (6,8)$ that look like this:

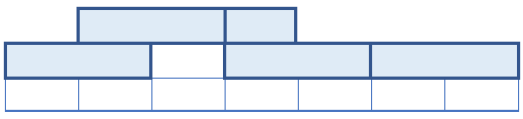

Figure 1: Meetings' intervals.

We should think of intervals as open; that is, $(2,4)$ does not overlap with $(4,5)$.

The solution to this instance is to schedule 3 meetings, *e.g.*, $\{(1,3), (4,5), (6,8)\}$. That is because $(1,3)$ and $(2,4)$ intersect, and $(4,5)$ and $(4,6)$ intersect. Thus, we can schedule at most 3

Figure 2: Counterexample for choice A

Figure 3: Counterexample for choice C

meetings, and we can indeed schedule 3 meetings, which means the maximum number of meeting we can schedule is 3.

Of course, there can be more than one solutions, *e.g.*, $\{(1,3),(4,6),(6,8)\}$ is also an optimal solution.

2.2 Designing the Algorithm: Listing the Possible Choices

To design a proper greedy algorithm for Interval Scheduling problem, we need to first think of the possible choices that may work. Specifically, the decision we need to make at each step is to select a meeting to schedule. Here we list three possible ideas:

A. Schedule the meeting that started earliest
B. Schedule the meeting that ends earliest
C. Schedule the meeting that takes least amount of time

2.3 Counterexamples for Some Ideas of Interval Scheduling

To check whether a choice is a good choice or not, we can try to find counterexample for those choices.

A. Schedule the meeting that started earliest

Intuitively, if we first schedule the meeting that started earliest, then the first meeting may be so long that it intersects all the other intervals. As shown in Figure 2, one counterexample for this choice is: $(1,10)$, $(2,4)$, $(5,7)$. In this example, rule A will choose $(1,10)$ because it starts the earliest, but this is bad because the optimal solution has two meetings $(2,4)$ and $(5,7)$.

C. Schedule the meeting that takes least amount of time

Intuitively, if we first schedule the meeting that takes least amount of time, then that interval may intersect two other non-intersecting intervals. As shown in Figure 3, a counterexample for this choice is: $(5,7)$, $(2,6)$, $(6,10)$. In this example, rule C will choose $(5,7)$, but the optimal solution is $(2,6)$ and $(6,10)$.

2.4 Description of Interval Scheduling Algorithm

Let us again follow our guideline for designing greedy algorithms.

(i) We ask first what decisions one needs to make. The answer here is kind of trivial. Clearly, we want to schedule intervals.

(ii) What is the "best" local option? In particular, what is the first meeting to schedule?

Intuitively, I would like to earlier meetings to start with, and the earlier the better. But it is not clear what we mean by "early". Consider two meetings $(1, 4)$ and $(2, 3)$. The first starts early and ends late. It is natural to conclude that one should schedule $(2, 3)$ because it ends early and allows me to schedule other meetings starting at 3 onwards. If I scheduled $(1, 4)$, I could only start another meeting at 4. Ending time is what affects future. (Here, we emphasize that the length of the interval is irrelevant. We simply want to maximize total number of scheduled meetings.)

Once you make the first decision, you can use the greedy rule repeatedly to find a solution. Hence, we have the algorithm:

Algorithm: always try to schedule the meeting with the earliest *ending* time.

It is simple to implement the algorithm. One starts by sorting all intervals by their ending times in ascending order. Then scan the intervals from the one with the earliest ending time, try to schedule the current interval, and if there is a conflict, then skip this interval.

2.5 Example Run of this Algorithm

Let us consider an example: $(1, 3)$ $(2, 4)$ $(4, 5)$ $(1, 6)$ $(6, 8)$ and see what our algorithm will do.

(1) First, the algorithm sort the meetings by their ending times, and get $(1, 3)$ $(2, 4)$ $(4, 5)$ $(1, 6)$ $(6, 8)$

(2) Clearly, $(1, 3)$ is the earliest ending meeting, and the algorithm schedules it.

(3) Then the algorithm looks at $(2, 4)$ and skips it, as it conflicts with $(1, 3)$, which is already scheduled.

(4) It then checks $(4, 5)$, and since there is no conflict, it is scheduled.

(5) Next, it looks at $(1, 6)$, and that conflicts with $(4, 5)$ and thus gets skipped.

(6) Finally, the algorithm checks out $(6, 8)$ and schedules it.

2.6 Analysis of Running Time

Sorting all the intervals by their ending times will take $O(n \log n)$ time. When we are scanning intervals, we need to check whether an interval conflicts with some interval that has already been scheduled. An important observation is that: If an interval intersects some of previously scheduled intervals, it must intersect the last scheduled interval. That is because the last scheduled interval has the largest ending time among all the scheduled intervals. Thus, at each step when we reach a new interval, we only need to check whether it conflicts the last scheduled interval. This process only takes $O(n)$ time. In a word, this algorithm runs in $O(n \log n)$ time.

2.7 Proof of Correctness for Interval Scheduling Algorithm

The intuition for this proof is that once we scheduled an interval, we will never regret. This proof is done via proof by contradiction.

Proof. Assume toward contradiction that the algorithm is not optimal, then there must be an instance of interval scheduling where algorithm does not schedule maximum number of meetings.

Let algorithm's solution for this instance be

$$ALG = (i_1, i_2, i_3, \cdots, i_\ell),$$

where i_ts $(1 \leq t \leq \ell)$ are indices of meetings scheduled, and the meetings are sorted by end time.

Let the optimal solution for this instance be

$$OPT = (j_1, j_2, j_3, \cdots, j_{\ell'}), \ell' > \ell.$$

Compare two solutions ALG and OPT, let k be the first index where $i_k \neq j_k$, i.e., ALG and OPT disagrees with the choice of the k-th meeting.

Claim 1. The solution $(i_1, i_2, \cdots, i_k, j_{k+1}, j_{k+2}, \cdots, j_{\ell'})$ is also an optimal solution that scheduled ℓ' meetings.

Proof of Claim. By the design of the algorithm, meeting i_k ends no later than meeting j_k. Thus,

$$end_{i_k} \leq end_{j_k} \leq start_{j_{k+1}},$$

where the second inequality comes from the fact that OPT is a valid solution. Therefore, i_k does not conflict with j_{k+1}. □

Repeat the above argument until for all $1 \leq k \leq \ell$, $i_k = j_k$. That means $(i_1, i_2, \cdots, i_\ell, j_{\ell+1}, \cdots, j_{\ell'})$ is a valid solution. However, this cannot happen because $j_{\ell+1}$ ends after i_ℓ, so it must be considered by the algorithm after i_ℓ is scheduled. Thus, by design of algorithm, ALG would also include $j_{\ell+1}$, contradiction!

By proof of contradiction, the assumption is false, ALG always finds the optimal solution. □

(Note: In this proof, we used "repeat the above argument until", which is not completely rigorous. To make it more rigorous, we can prove by induction on k that $(i_1, i_2, \cdots, i_k, j_{k+1}, j_{k+2}, \cdots, j_{\ell'})$ is a valid solution. However, we are not requiring you to do a induction in this case.)