COMPSCI 330: Design and Analysis of Algorithms September 14

Lecture 6: Greedy Algorithms I

Lecturer: Rong Ge *Scribe: Fred Zhang*

1 Overview

In this lecture, we introduce a new algorithm design technique—*greedy algorithms*. On a high level, it bears the same philosophy as dynamic programming and divide-and-conquer, of breaking a large problem into smaller ones that are simple to solve.

Although easy to devise, greedy algorithms can be hard to analyze. The correctness is often established via proof by contradiction. We demonstrate greedy algorithms for solving *fractional knapsack* and *interval scheduling* problem and analyze their correctness.

2 Introduction to Greedy Algorithms

Today we discuss greedy algorithms. This is the third algorithm design technique we have covered in the class and is the last one. Previously, we saw dynamic programming and divide-and-conquer. All these paradigms attempt to break a large, complicated problems into many smaller ones. In divide-and-conquer, the idea is simply to partition the problem into independent parts and solve them. In dynamic programming, we collect a lot of small problems that look similar to the original problem and make a table to solve the small ones.

Now greedy is probably the most intuitive approach in algorithm design. If a problem requires to make a sequence of decisions, a greedy algorithm always makes the "best" choice given the current situation and never considers the future. You will see what we mean by "best" choice and current situation later. This automatically reduces the problem to a smaller subproblem which requires making fewer decisions. Suppose originally I have to make n decisions, but now I only worry about one local decision.

For a warm-up, consider you walk in Manhattan, and the destination in northeast. Then walking towards north or east always reduces your distance to the destination. Initially, the choice of going towards which direction does not matter, and you can be greedy. But consider you are driving, and sometimes the path is one-way. Then greedy does not always work, and you want to plan ahead to make sure you are not blocked by one-way street in future.

2.1 Design and Analysis

In designing greedy algorithm, we have the following general guideline:

(i) Break the problem into a sequence of decisions, just like in dynamic programming. But bear in mind that greedy algorithm does not always yield the optimal solution. For example, it is not optimal to run greedy algorithm for Longest Subsequence.

(ii) Identify a rule for the "best" option. Once the last step is completed, you immediately want to make the first decision in a greedy manner, without considering other future decisions.

The hard part usually lies in analysis. In divide-and-conquer and dynamic programming, sometimes the proof simply follows from an easy induction, but that is not generally the case in greedy algorithms. The key thing to remember is that greedy algorithm often fails if you cannot find a proof.

A common proof technique used in proving correctness of greedy algorithms is proof by contradiction. One starts by assuming that there is a better solution, and the goal is to refute it by showing that it is actually not better than what the algorithm did.

3 Fractional Knapsack

Now let us consider our first problem, the *fractional knapsack* problem. It is similar to the knapsack problem we solved via dynamic programming. Recall that we have knapsack of bounded capacity W, so it can hold items of total weight at most W. There are n items, each of weight $w_1, w_2, \ldots, w_n$. Each item also has a value $v_1, v_2, \ldots, v_n$. The difference of this fractional knapsack is that the items are infinitely divisible. You are allowed to put, say, $1/2$ (or any fraction) of an item into the knapsack. The goal is to select a set of fractions $p_1, p_2, \ldots, p_n$ for all items to maximize the total value

$$p_1 v_1 + p_2 v_2 + \ldots p_n v_n \tag{1}$$

subject to the capacity constraint

$$p_1 w_1 + p_2 w_2 + \ldots p_n w_n \leq W. \tag{2}$$

In other words, we allow p_i only to be 0 or 1 in the previous version of the knapsack problem, but now they can be anything between 0 and 1.

Example 1. Consider an example where the capacity $W = 10$ and there are three items

Item	Weight	Value
#1	6	20
#2	5	15
#3	4	10

If this is the integral knapsack problem, we would choose item #1 and #3, and this gives me total value 30. Of course, in fractional knapsack I can choose fractions of items. It turns out that the optimal solution is choose item #1 entirely and 0.8 fraction of item #2. The total weight is still 10, yet the value is now 32. One can see that allowing fractional solution may give us more valuable solutions.

3.1 Algorithm

Let us follow the guideline in Section 2.1 to design an algorithm.

(i) We first ask how one should break down the problem, *i.e.*, what decisions one needs to make. Of course, we would like to answer what item I should put into the knapsack and of what fraction. Answering this sequentially eventually yields a solution.

(ii) We specify a rule for finding the "best" item. We are faced with our first choice—what is the first item I should put into the knapsack? In this case, the rule is natural: we put in the item with maximum value per unit weight; that is,

$$\text{Find an item } i \text{ such that } v_i/w_i \text{ is maximized.}$$

Now there are two cases. If the knapsack can take in this item entirely, then I am happy to do that. Otherwise, I would put as large fraction as possible.

Example 2. We can check that the greedy rule indeed gives optimal solution for Example 1. We first put in item #1 entirely and second consider item #2, but since it is too large for the remaining capacity, we put as much as possible, which is a 0.8 fraction.

How can we implement the algorithm? We first sort all items by their "value per unit weight" and put them in a list. Then scan the list in descending order, first consider the highest "value per unit weight", try to put it in, and so on.

3.2 Analysis

Running time. The running time of the algorithm is easy to analyze. We sort the items first, which takes $O(n \log n)$ time and the rest takes $O(n)$ time. Therefore, this is an $O(n \log n)$ time algorithm.

Correctness. As said earlier, it can be hard to prove correctness for greedy algorithms. Here, we present a proof by contradiction.

Theorem 1. The algorithm described in Section 3.1 provides an optimal solution for the fractional knapsack problem.

Let me first give a sketch for the proof idea.

(i) Assume that there is a better solution, in the hope that we see a contradiction.

(ii) Argue that the claimed "better" solution is in fact not better than the one provided by our algorithm. (If someone comes up to you claiming that she has a better solution but you can always refute her, then that means your solution is the best possible.)

Proof. The index of the items is irrelevant. Hence, without loss of generality, assume items are sorted in decreasing order of v_i/w_i; that is,

$$v_1/w_1 \geq v_2/v_2 \geq \ldots \geq v_n/w_n \tag{3}$$

Suppose the algorithm gives a solution $\mathsf{ALG} = (p_1, p_2, \ldots, p_n)$. Only the last non-zero item in the solution is fractional, since the algorithm always tries to put in the item entirely and the last item considered fills the knapsack.

Now assume for a contradiction that there is a better solution. We call this hypothetical solution OPT and $\mathsf{OPT} = (q_1, q_2, \ldots, q_n)$. Since OPT is better than ALG, there must be a location where it differs from ALG. Suppose that the first location of the difference (from left to right) occurs at item i, where $p_i \neq q_1$ but $p_{i'} = q_{i'}$ for all $i' < i$.

Before item i, the two solutions are the same, and so are the remaining capacity till this point. By the definition of our algorithm, it tries to put as much item i as possible into the knapsack. Therefore, OPT does not select more fraction of item i than ALG, *i.e.*, $q_i < p_i$. (Otherwise, OPT would exceed the capacity.)

However, since OPT is better, there must be an item j ($j > i$) such that $p_j < q_j$. Consider one removes ϵ fraction of item j from OPT and uses the capacity that is freed up to accommodate more fraction of item i. Formally,

$$\begin{aligned} q_j &\leftarrow q_j - \epsilon \\ q_i &\leftarrow q_i + \frac{\epsilon \cdot w_j}{w_i}. \end{aligned} \tag{4}$$

The operation (4) does not violate capacity constraint because removing ϵ fraction of item j frees up $\epsilon \cdot w_j$ capacity, which can be used to take $\frac{\epsilon \cdot w_j}{w_i}$ additional fraction of item i. We call the modified solution OPT'

We claim that the new solution OPT' is as good as OPT. The value of this solution is given by

$$\begin{aligned} \text{value}(\mathsf{OPT}') &= \text{value}(\mathsf{OPT}) - \underbrace{\epsilon \cdot v_j}_{\text{loss on item } j} + \underbrace{\frac{\epsilon \cdot w_j}{w_i} \cdot v_i}_{\text{gain on item } i} \\ &\geq \text{value}(\mathsf{OPT}). \end{aligned}$$

The last inequality follows from the fact that $v_i/w_i \geq v_j/w_j$. The gain on item i is at least the loss on item j.

After the shift, OPT' is closer to ALG. We repeat this argument until the operation cannot be done. Eventually, OPT becomes ALG, and each step can only increase the value. This implies that value(OPT) $\leq$ value(ALG), a contradiction. □

Admittedly, this proof is more complicated than it needs to be. After class, I will post a slightly simpler proof on course website. The reason we show this complicated proof is that it is easier to generalize to the analysis of other greedy algorithms.

4 Interval Scheduling

In the remaining time, let us look at another problem, called *interval scheduling*. There are n meeting request, 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 3. Suppose we have meetings $(1, 3), (2, 4), (4, 5), (4, 6), (6, 8)$ that look like this:

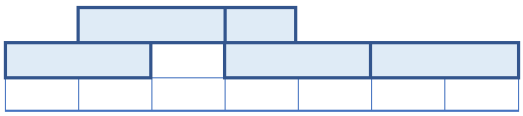

Figure 1: Meetings' intervals.

We should think of intervals as open; that is, $(2, 4)$ does not overlap with $(4, 5)$. The solution to this instance to schedule three meetings, *e.g.*, $\{(1, 3), (4, 5), (6, 8)\}$. Of course, there can be more than one solutions.

4.1 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.

4.2 Analysis

Running time. It is again easy to see that the algorithm runs in $O(n \log n)$ time, since the most expensive step is sorting.

Correctness. We shall focus on analyzing the correctness of this algorithm by first understanding the algorithm more concretely.

Example 4. Let us consider Example 3 again and see what our algorithm will do.

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

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

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

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

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

We see indeed the algorithm schedules three meetings, producing an optimal solution.

Let us now try to prove the algorithm. The idea is fairly simple. Suppose OPT schedules more meetings than ALG. Then at the first point where OPT disagrees with ALG, it must have scheduled a meeting that ends later than the one ALG schedules. Whatever OPT did afterwards, the algorithm will do no worse than that.

Theorem 2. The algorithm described in Section 4.1 always produces an optimal solution for the interval scheduling problem.

Proof. Assume algorithm schedules k meetings $\mathsf{ALG} = (i_1, i_2, \ldots, i_k)$. Further, suppose for a contradiction that OPT is a better solution producing $t > k$ meetings, and they are $\mathsf{OPT} = (j_1, j_2, \ldots, j_t)$. Let both solutions solutions be sorted in starting time; that is, $s_{i_1} < s_{i_2} < \ldots < s_{i_k}$ and $s_{j_1} < s_{j_2} < \ldots < s_{j_k}$.

Let p be the first meeting where $i_p \neq j_p$. By the design of the algorithm, we have that i_p ends before j_p, and therefore i_p ends before j_{p+1} starts. Hence, the new solution

$$\mathsf{OPT}' = (i_1, i_2, \ldots, i_p, j_{p+1}, j_{p+2}, \ldots, j_t) \tag{5}$$

is also a valid schedule that still schedules t meetings, as many as OPT does. This is similar to the proof of Theorem 1—OPT' is now closer to ALG. Again, one may repeat this argument. Eventually, we get to an OPT' where $i_p = j_p$ for all $p \leq k$, and the schedules would look like following

$$\begin{aligned} \mathsf{ALG} &= (i_1, i_2, \ldots, i_k) \\ \mathsf{OPT}' &= (i_1, i_2, \ldots, i_k, j_{k+1}, \ldots, j_t). \end{aligned}$$

The first k meetings are exactly the same, but OPT' continues to schedule $j_{k+1}, \ldots, j_t$ after i_k. This contradicts the design of the algorithm because if j_{k+1} is still available to be scheduled at the end of ALG, the algorithm would schedule it. □

5 Summary

We have discussed the greedy paradigm in algorithm design and showed that it gives algorithms that solve fractional knapsack and interval scheduling problems. The proofs for correctness of these algorithms are fairly tricky, and we will see more examples next lecture.