Notes for CIS 3200: Introduction to Algorithms
Greedy Algorithms

Erik Waingarten

Contents

1 Greedy Algorithms

We will begin the discussion with the following problem. We receive a sequence of "jobs" that we need to do, and each of the n jobs has a time that it takes to complete job $i \in [n]$. We will denote $t_i \geq 0$ to be the time it takes to complete job i. Now, the total time available is T, and we get rewarded for every job that we complete. Similarly to the optimization problems that we studied in dynamic programming, we want to understand how many jobs we may complete subject to some constraints:

- **Input**: A sequence of completion times $t_1, \dots, t_n \geq 0$ and a total time T.
- **Output**: The goal is to determine the most number of jobs that we may complete while keeping our time constraint, i.e., we want to maximize the objective function $|S|$ over subsets $S \subset [n]$ subject to $\sum_{i \in S} t_i \leq T$.

We can of course consider a dynamic programming solution to select the subset S which maximizes the size subject to our constraint, but you can likely come up yourself with a *greedy* algorithm to solve this problem:

> **Potential Simple Algorithm**: Sort the jobs in non-decreasing order of their completion time t_i, and then pick the most number of jobs while we have enough time left. In other words, if we let $t_1 \leq t_2 \leq \cdots \leq t_n$ be the sorted order (after a re-assignment of indices), find the largest $k \in [n]$ for which
>
> $$\sum_{i=1}^{k} t_i \leq T.$$
>
> Then, we do jobs corresponding to 1 to k.

Note that this does not look like dynamic programming solutions, but we can probably all agree that the above, even simpler, algorithm gives a correct solution.

1.1 Greedy as a Type of Dynamic Programming

When faced with a new problem, we will still follow a similar pattern where we ask for a top-level guide, which will determine which property of our particular problem we will exploit in order to admit efficient algorithms. In particular, in this case, we can ask a top-level guide:

> **Top-Level Guide**: Which job should we do first?

Now, in a dynamic programming solution, once we had specified a top-level guide, we set up the notation that we need in order to explore all of the possibilities to our top-level guide. In particular, we usually considered the top-level guide as a means to recurse on various options which we will combine once we know the answers. Greedy algorithms build on top of the dynamic programming solution, by proving an even more special property of the problem. Namely, we try to answer the top-level guide directly *so that we only explore one option* (which essentially means we make a decision). In particular, we can say:

> **Answer to Top-Level Guide**: Do the job which will finish first, as this is the "cheapest" job to complete.

Notice that we've used something more special about our problem that allows the greedy algorithm to work. We were able to give a precise answer to the top-level guide, so we can make a single decision and then consider the remaining jobs. We expect that the running time is now much much faster, we are only exploring "one subproblem." The tradeoff will be: we will get faster and simpler algorithms, but, the class of problems we will be able to solve is smaller and the proof of correctness will be more challenging.

Proof of Correctness of Our Greedy Algorithm. It is very often the case that we can structure the proof of correctness of a greedy algorithm as a proof by contradiction. For our problem, we need to argue that the set of jobs that our algorithm picks is the set which maximizes our objective function subject to the constraints. So we define the notation to do our proof:

- Let G denote the set of jobs that our greedy algorithm takes.
- Let S_o denote the optimal solution (the set of jobs which maximizes $|S_o|$ while keeping the time constraint).
- Assume For Contradiction: That $|G| < |S_o|$.

We will now complete the proof by using a combination of *progress arguments* and *exchange arguments.*

Claim 1.1 (Progress Lemma for Greedy Algorithms)**.** *For all k, the first k jobs picked by the greedy algorithm (i.e., the first k jobs of G) take at most as much time as any set of k jobs. In other words, for any set T of size k, the time of T is at least the time of the first k is G.*

The proof is trivial, since the first k jobs in G are the cheapest sum of k jobs. Now, we will do the following argument which essentially completes the proof.

Exchange Lemma for Greedy Algorithms Suppose that the greedy algorithm completed and returned a set G which satisfies $|G| = k$. By our assumption (for the sake of contradiction) $|G| < |S_o|$, and there must be a job j where $j \in S_o \setminus G$. Consider a set $T \subset S_o \setminus \{j\}$ of size k and by our progress lemma, we have

$$\sum_{i \in G} t_i \leq \sum_{i \in T} t_i \leq \sum_{i \in S_o} -t_j \leq T - t_j.$$

In summary, the cost of the greedy solution $\sum_{i \in G} t_i \leq T - t_j$ and it does not contain j. Now we have a contradiction, since the greedy algorithm would not have terminated with G, as it could add another job t_j without violating the time constraint.

In summary, we have done an *exchange argument*, which we can summarize with the following phrase. In the above case, we have assumed that an execution of the greedy algorithm produced a non-optimal solution. Even though we do not (algorithmically) know the optimal solution, we can, in the analysis, consider one optimal solution S_o. We then use the non-optimality to perform an "exchange" which we consider as a simple (and local) change to show that the greedy output (which we assumed was the output of greedy) was not—a contradiction.

> **Exchange Argument**: We assume that the greedy algorithm produced a non-optimal solution. Then, we use the non-optimality to show that some part of the algorithm executed in an incorrect fashion, and obtain a contradiction.

Now, the running time of the algorithm is $O(n \log n)$, since we need to sort the jobs according to the running time and then pick one at a time.

2 (Weighted) Activity Selection Problem

We saw this problem in the context of dynamic programming, and here we will re-visit the problem *when there are no weights.* Recall the definition of the problem:

- **Input**: We receive a collection of activities $1, \ldots, n$, where the i-th activity has start time s_i and end time t_i.
- **Goal**: The goal is to maximize the number of activities selected (i.e., all "rewards" are set to 1) while ensuring that no two activities conflict, which means that there are never two selected activities whose start/end time interval intersects.

Let's proceed with the same approach as in dynamic programming, where we asked what is the "top-level guide,"

> **Top-Level Guide**: Which activity should I pick first?

There are various ways here to be greedy. We could (i) sort the activities according to the start times and pick the first one, or (ii) sort the activities according to the end times and pick the first end-times. In general, we are posing the high-level question, but then we are positing an answer which will allow us to explore only one potential answer to the high-level question. Our task is now slightly more challenging, since we must determine whether the answer to the top-level guide is actually true or not (whereas in dynamic programming, we never posited a solution). So, once we've asked the top-level guide, we must determine the "top-level solution." Let me emphasize that the process of finding an potential answer to the top-level question is an iterative process where we will pose a potential answer and try to determine whether it works or doesn't work—one would need to prove it.

> **Potential Top-Level Answer**: Find the activity with the earliest finishing time. (?)

This strategy will actually work (although there are many natural strategies which do not) and we must now prove that if we always follow this strategy, we will select the maximum number of activities. This gives us the complete description of the algorithm:

1. We sort the activities according to their end times.
2. We pick the activity which finishes first and we remove all of the conflicting activities, and recurse.

We will now prove this using an exchange argument.

Suppose G is the set of activities selected by the greedy algorithm, and assume that there were k chosen sequences given by $i_1, \dots, i_k \in [n]$, and we sort them by finish time (so that i_1 was the first activity chosen, followed by i_2, and so on until i_k). Suppose that $S_o \subset [n]$ is the optimal solution with activities $j_1, \dots, j_\ell$ which are sorted by end time. We now have the following plan:

Proof Plan: We will slowly modify the activities in S_o to look more and more like G.

In particular, suppose we let

$$h^* = \text{ smallest index such that } i_{h^*} \neq j_{h^*}, \text{ or } \infty.$$

Which means that activities $i_1, \dots, i_{h^*}$ and $j_1, \dots, j_{h^*}$ are exactly the same. If $h^* = \infty$ then they must be equal. So, here is the special observation:

Observation 1. The end time for j_{h^*} must be strictly after the end time for i_{h^*}. The reason for this is that before h^*, the optimal and the greedy solution are exactly the same, and then the greedy solution picked the first end time, and hence must not conflict with any activity which did not conflict with j_{h^*}.

- Suppose that we considered another solution S_o' which was exactly the same as S_o but instead of j_{h^*}, we picked i_{h^*}.
- Then, S_o' is another optimal solution where the first h^* elements are equal to that of G.

This means that, if there is an optimal solution S_o which agrees with greedy on the first $h^* - 1$ activities, there is an optimal solution S_o' which agrees with greedy on the first h^* activities. This means that G must also be an optimal solution (one can prove this formally by an induction). Hence, our algorithm is correct and the running time becomes $O(n \log n)$ (dominated by the time it takes to sort the activities according to their end time).

3 Huffman Coding Algorithm

We will now look at another problem which admits a greedy solution, although it will turn out that the proof of correctness will be a trickier, and we will break the problem down by making observations before we can fully attack it. The problem setup is that, we have an alphabet Σ and we want to be able to transmit documents along a channel which only admits binary bits, so we need a method to translate documents (which are sequence of strings) onto "code words" consisting of just bits $\{0, 1\}$. The goal is to design our codebook.

- **Input**: We receive as input the collection of characters in our alphabet $\Sigma = \{1, 2, \dots, n\}$ which we will assume is the elements in $[n]$, and we will also receive a "probability" p_i that the symbol i will come next (remember the setup for the optimal binary search trees).
- **Output**: The goal is to come up with a mapping which assigns for each symbol i a binary string in $\{0, 1\}^*$ such that we will be able to encode our documents by placing the encodings of each symbol.

Note here that we would like to minimize the "expected" length of the encoding, while ensuring that we can unambiguously decode, where being unambiguous means that there are no codewords which is a prefix to another codeword. This will be a constraint in our optimization problem:

- **Constraint**: For every two characters i, j with $i \neq j$, it should be the case that our assigned codewords c_i and c_j are not prefixes, so that c_i is not a prefix of c_j (and similarly, c_j is not a prefix of c_i).

Our objective is then to minimize $\sum_{i=1}^{n} p_i \cdot |c_i|$.

Codebook as Binary Tree. The reason this problem is related to the problem we studied on optimal binary search trees is that there is a natural connection between "prefix-free" codebooks in $\{0, 1\}^*$ and binary trees. We will associate a codebook $c_1, \dots, c_n \in \{0, 1\}^*$ as a full binary tree (one where every internal node has two children) whose leaves are set to the indices $[n]$.

- There is a root and there is a left-child (corresponding to 0), and a right-child (corresponding to 1).
- Each of the indices is a leaf of the tree, and the root-to-leaf path to an index i may be found by taking the path according to c_i. In other words, we read the indices in c_i from left-to-right.

Therefore, we may re-cast the problem as an optimization over binary trees T where there are n leaves labeled $1, \dots, n$, and we want to minimize

$$f(T) = \sum_{i=1}^{n} p_i \cdot d_i(T),$$

where $d_i(T)$ is the depth of node i in T.

3.1 Towards an Algorithm

We will now begin to design the algorithm. The key is that we have not yet determined what to be greedy on. It will become useful in order to determine what the algorithm is, to study the structural properties of an optimal solution. The reason is that, if we are able to study the structural properties of the optimal solution, then it suffices for the algorithm to search through only the solution space consisting of solutions which have the property that we will show. The more we know about the optimal solution, the more constrained the search becomes, and thus, the faster that we may search through the space.

Step 1: Simple Observation on Full Binary Trees We will design the algorithm slowly by building up some structural facts aimed at capturing "what should we be greedy on?."

Observation 2. In any full binary tree with at least two leaves, there is a pair of sibling leaves at the greatest depth.

The proof is straight-forward: we pick the deepest node, then choose its parent, and then its sibling. If this is not a leaf, then there is a deeper node.

Step 2: Algorithm Given Topology. Suppose we are told the topology of the tree, but we are not told the labels of the leaves. Then, how do we figure out where to place each of the leaves? Its not hard to see that we should sort the nodes according to the least frequent to most frequent elements, and then place them from deepest to shallowest leaves. This is optimal, because we have essentially fixed the depths $d_j(T)$ for $j \in [n]$, and we now want to determine how to map $\sigma \colon [n] \to [n]$ so that we label the symbol i with leaf $\sigma(i)$.

Suppose that the optimal did not choose the greedy order as above. Then, there exists an index i, j where $p_i < p_j$ but $d_i(T) < d_j(T)$. If we *exchange* the two, we end up with another solution which has a strictly better objective. This is a contradiction, and hence not the optimal. In other words, once we know the topology, the greedy way to fill-in the leaves gives the desired guarantee. This means that, if we can find the right topology, the specific mapping of indices to the leaves is determined.

Combining the two steps, we have the following statement:

Lemma 3.1. *In the optimal full binary tree representing the encoding, the two least frequently occurring symbols (those which have smallest probabilities) will be siblings at the deepest node.*

Description of Algorithm. We now describe the algorithm, which will be a recursive algorithm which works on the least-frequent elements.

- If the number of symbols is one or two, we make these siblings at the leaf.
- Otherwise, we join the least two frequent symbols into one "super-symbol."
- We recurse on the $n - 1$ many symbols, and then we expand the leaf that we find into the two leaves.

Lemma 3.2. *Suppose that our greedy algorithm finds the optimal tree on alphabets of size $n - 1$. Then, it finds the optimal tree on an alphabet of size n.*

Proof. Suppose we arrange the symbols into non-decreasing order of probabilities. Then, we perform the substitution of the two elements into one, and we solve the recursive algorithm on the $n-1$ symbols, which we assume for inductive hypothesis that it is optimal. Then, we expand on the tree, and the increase in the objective is $p_1 + p_2$.

On the other hand, suppose you had an optimal tree for the n symbols. Then, the two sibling nodes contain p_1 and p_2, so that we could compress to get a solution on $n-1$ symbols of cost which decreases by $p_1 + p_2$. Thus, if we let opt_n denote the optimal cost for n symbols, we have that that the cost of our output

$$\text{cost of output} \leq \text{opt}_{n-1} + p_1 + p_2 \leq \text{opt}_n - (p_1 + p_2) + (p_1 + p_2) = \text{opt}_n.$$

The first inequality is our transformation, and the inductive hypothesis which assumes that we find the optimal solution when we join p_1 and p_2. Now, we also have that $\text{opt}_{n-1} \leq \text{opt}_n - (p_1 + p_2)$, because if we had the optimal tree on all nodes, we can also find one where p_1 and p_2 are siblings, and thus obtain a solution to the joining of 1 and 2 whose cost is $\text{opt}_n - (p_1 + p_2)$. $\square$

4 Scheduling Jobs with Deadlines and Rewards

We'll now do an example of a scheduling problem. The goal is that you are an organizer who wants to schedule a sequence of n jobs.

- Suppose we have jobs $j_1, \ldots, j_n$.
- Each job takes one unit of time.
- Each job has a deadline d_i, which specifies the (integer) time it must be completed.
- Furthermore, there is a reward r_i for completing the job before the deadline.

Note that in order to receive the reward r_i, you must start the job i before time $d_i - 1$ so that it is completed before time d_i. The goal will be to maximize the total reward that we can find.

Idea 1. A natural greedy strategy here is the following: start with the jobs whose deadline comes first, and then within those, maximize the reward. However, there is a simple counter-example, using the fact that within time t, one can complete at most t jobs; thus, we may be fooled into thinking that we must complete a job because of a deadline, and this would decrease the total number of remaining jobs to $t-1$. In particular, consider

- Let j_1 be a job whose deadline is 1 and has reward 1.
- Let j_2 be a job whose deadline is 2 and has reward 10.
- Let j_3 be a job whose deadline is 2 and has reward 10.

Here, the algorithm schedules j_1 first since it has the earliest deadline, but then can only pick a single job among j_2 and j_3. This way, the total reward obtained is 11, whereas if we had forgotten about j_1 and done j_2 and j_3 (even though the deadlines where 2), the total reward would be 20.

Idea 2. Another natural greedy strategy is to greedily choose the job which has the highest reward. However, there is another simple counter-example here. We may have that the a high-reward job has a later deadline, and by completing it first, we are missing out on a sooner job with less reward. For example, consider

- Let j_1 be a job whose deadline is 1 and has reward 1.
- Let j_2 be a job whose deadline is 2 and has reward 10.

In this case, the greedy strategy picks j_2 first and this means that it gets reward 10, but misses out on the reward coming from completing j_1 first, followed by j_2, which would get reward 11.

Idea 3. The reason why the prior ideas fail is that we are trying to find a scheduling of jobs which goes in increasing order of time, and it is more useful to go in *decreasing* order of time. Consider grouping the jobs according to their deadlines, and order them in decreasing order of their deadlines. Then, if we start filling the jobs backwards (starting from the greatest-time to the least-time), we can greedily pick the highest reward within the group. Then, we bunch the groups of jobs coming before it and we recurse. Note that this is the natural way to schedule jobs if we were scheduling in the reverse order.

From the implementation details, we note that there are quite a few ways to do that, but we essentially need an ordering of the objects according to their rewards within the particular deadlines, and we must be able to merge sorted lists. This can either be done with a heap (which maintains the max element), or via merging sorted lists with the MERGE algorithm.

4.1 Proof of Correctness

We will now also prove that this third idea is correct, which we will do via an exchange argument:

- Let G denote the sequence of jobs that we schedule, $g_1, \ldots, g_T$, where T is the total time steps. We also keep many dummy jobs j^* which have zero rewards but infinite deadlines. So, if we ever run out of jobs, we schedule j^*.
- Let O denote the optimal solution which schedules jobs $i_1, \ldots, i_T$. Again, we keep some dummy jobs.

Claim 4.1. *In the optimal solution, we might as well assume that the scheduled jobs occur in increasing order of deadlines. This is because we do not change the cost of the solution, and if one was feasible, then we only make the scheduled jobs more feasible.*

So now, we might as well prove our algorithm correct by an induction which goes from greatest deadline to least deadline. Well, what can we say about the last job scheduled by the optimal solution and that which is scheduled by greedy; it better be that the reward of optimum is at least as good as that of the greedy solution. We can then modify the optimal solution to one whose last job is the same as greedy:

- If O had picked the last job as being different from the last one G picks, and O does not pick it, then we can exchange.
- If O picked the same as the last job G picks, then we are good.
- If O had a different last job from G, but the last job that G picks comes sometime before in O, then since O was ordered in increasing deadlines, we can exchange it so that the last job of G appears in the last slot.

This means that O looks exactly like G in the last slot. We then use induction to show that the greedy solution and the optimal solution are the same.