Lecture 8: Greedy Algorithms

Plan/outline

We introduce and study a "new" paradigm -- greedy choice. Imagine a problem in which we need to make a sequence of decisions. A greedy algorithm is one that makes decisions in a myopic manner: at every step, it takes the best choice, using some valuation function that only depends on the *current* state. Further, we assume that once decisions are made, they are irrevocable.

Greedy algorithms turn out to be easy (and natural) to come up with (perhaps because we're inherently myopic in decision making). Unfortunately, they are seldom optimal. In spite of this, we will see that there are many cases in which they still give reasonable insights into the structure of the corresponding problems.

Basic examples

Traveling salesman problem. One example that we already saw (Lecture 7) is that of the traveling salesman problem (where a salesman starts at city 1, needs to visit every other city precisely once and return to the start. Thus the salesman must make a sequence of decisions of the form "which vertex to visit next?"

The greedy algorithm here is to visit the closest unvisited city (to the current location). We saw that this procedure is not optimal.

Coin change. Suppose we are given coins of certain denominations, say 1c, 5c, 10c, 20c, 25c, 50c. Let us say that we need to make change for 75c using the fewest number of coins. The natural greedy algorithm is to start with the largest denomination less than 75c (in this case it is 50c), remove it and recurse.

It's easy to come up with cases in which this is not optimal. For example, suppose we wish to make change for 40c. The greedy algorithm first picks 25c, then it is forced to pick 10c and 5c. Meanwhile, the optimal solution (one with the fewest number of coins) is to choose two 20c coins.

We will see more examples in the next couple of lectures.

Scheduling

Consider the following scheduling problem: we have n jobs that need to be scheduled on a machine. The processing time of job i i s p_i, which is given as input. The aim is to find the order in which the jobs should be done, so as to minimize the *average completion time*.

Suppose we have three jobs with processing times p_1, p_2, p_3. Then, doing them in the order $(2, 1, 3)$ yields the following completion times: job 1 is completed at time $p_2 + p_1$. Job 2 is completed at time p_2. Job 3 is completed at time $p_2 + p_1 + p_3$. Thus the total completion time is $3p_2 + 2p_1 + p_3$. The average completion time is this quantity divided by 3.

The example already gives a hint of what"s going on. First, instead of minimizing the average completion time, we can focus on the total completion time. This quantity seems to depend "more strongly" on jobs that are processed early on.

This gives a hint that we should process the short jobs first. I.e., suppose we think of iteratively deciding which job to do next, then the greedy choice (one that minimizes the completion time of the *next* job) is to process the shortest job. It turns out that the greedy procedure is optimal for this problem! We will see a couple of different proofs of this.

Proof by direct computation.

The first proof is via a direct computation inspired by the example above. Suppose we decide to perform the n jobs in the order $(s(1), s(2), \ldots, s(n))$ where s is a permutation of $[n]$ (short-form for $\{1, 2, \ldots, n\}$). Then the completion time of the $s(i)$th job is $p_{s(1)} + p_{s(2)} + \cdots + p_{s(i)}$. Thus the total completion time is $np_{s(1)} + (n-1)p_{s(2)} + \cdots + 2p_{s(n-1)} + p_{s(n)}$.

We would like to find a permutation s so as to minimize this quantity. In other words, we have the numbers $p_1, p_2, \ldots, p_n$, we need to take one of them, multiply by n, take another, multiply by $(n-1)$, ..., so that in the end, the sum of these quantities is minimized. The optimal thus is to pick the smallest of the p_i to be multiplied by n, the next smallest to be multiplied by $(n-1)$, and so on. [Formally, one can show this by arguing that if $p_{s(1)} > p_{s(2)}$, then swapping them would result in a lower objective value.]

Proof by "structural" argument

In the above argument, we can see that we don't need to find a closed form for the total completion time. Instead, suppose we just ask: what happens if we swap neighboring jobs in some candidate ordering?

Specifically, consider an ordering in which the job lengths are $q_1, q_2, \ldots, q_n$ (a permutation of $p_1, \ldots, p_n$, as before). If we swap q_i, q_{i+1}, then the completion times of all the jobs before i will be unchanged. So also, the completion times of all the jobs after $i+1$ would be unchanged. The completion time of the job corresponding to q_i would change from $q_1 + \cdots + q_i$ to $q_1 + \cdots + q_{i+1}$, and the completion time of the q_{i+1} job would now be $q_1 + \cdots + q_{i-1} + q_{i+1}$. Thus the difference in total completion times is precisely $q_{i+1} - q_i$.

Now, if the optimum ordering was optimal, all such quantities must be ≥ 0, or else swapping would result in a smaller cost. Thus, for the optimal ordering, we have $q_{i+1} \geq q_i$, which implies that the jobs were in increasing order.

Remark. This argument is "conceptually" the same as the one before, but it doesn't involve first obtaining a closed form for the solution.

Another proof technique, which is common in greedy algorithms, is to argue inductively that there exists an optimal solution whose first t jobs are the ones chosen by the greedy algorithm. In essense, this argument is saying that the first t choices are ``correct''. We will see this argument in the case of the Minimum Spanning Tree problem.

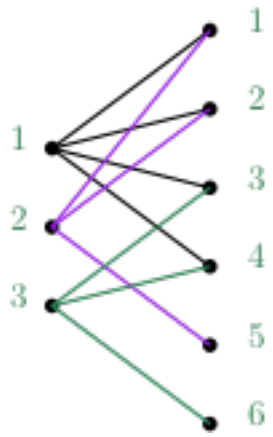
Set cover -- hiring to cover all skills

Consider the following *hiring* problem. Suppose we have a company in which we require expertise in m different areas, i.e., we have a set of m skills, and we require at least one employee who has skill i, for all $i \in [m]$. Now, presented a set of candidates, each of whom has a subset of the m skills, the goal is to pick the fewest number, so as to "cover" all the skills.

Formally, we are given $S_1, S_2, \ldots, S_n$ that are subsets of $[m]$, and the goal is to pick some of them, say the indices $i_1, i_2, \ldots, i_k$, such that k is as small as possible, and $S_{i_1} \cup S_{i_2} \cup \cdots \cup S_{i_k} = [m]$.

Greedy algorithm. Let us imagine selecting one set at a time (i.e., hire employees one at a time). In each step, the natural strategy is to hire the one who *brings most to the table*, i.e., covers the maximum number of skills that are uncovered so far (breaking ties arbitrarily).

Is this always optimal? It is fairly easy to see that the procedure isn't always optimal. For example, consider the figure below. The vertices on the left represent people, and the ones on the right represent skills. Vertex i on the left is connected to j on the right if person i possesses skill j.



The greedy algorithm picks vertices 1, 2, 3 (because at the start, person 1 posseses the most skills), and once we chose this person, we are forced to pick 2, 3. The optimal solution, of course, is to pick only 2 and 3.

But in the set cover problem, the greedy algorithm posseses a rather nice feature: even though the solution obtained is not the optimal one, it is not *too far* from the optimum. Specifically, the following is a theorem one can prove:

Theorem (approximation ratio). Consider some instance of the set cover problem, and suppose that the optimum solution consists of k sets (people). Then the greedy algorithm picks no more that $k \ln m$ people. (As is standard, $\ln m$ refers to the natural logarithm of m.)

Remark. A statement like this is often referred to as an approximation guarantee. We are relating the value of the solution obtained by the algorithm to the *optimal* solution. The non-trivial thing is that the algorithm itself has no idea what the optimum solution is/looks like.

Proof. We must argue that if there is a small optimum solution, the greedy procedure also picks a reasonably small solution. Intuitively, we chose the greedy algorithm because it seems to make the most *amount of progress* in the current step. Can we quantify this?

Consider running t iterations of the algorithm. Let V_t denote the set of uncovered skills at that point, and for convenience, denote $u_t = |V_t|$. Now, the greedy algorithm in teration $(t+1)$ selects the person who has the most skills in V_t (breaking ties arbitrarily). Can we argue that there is some person who covers a "good fraction" of V_t?

We claim that there is a person who covers $\frac{u_t}{k}$ of these skills, or in other words:

Claim. $u_{t+1} \leq u_t - \frac{u_t}{k}$.

Consider any V_t. The greedy algorithm at iteration $(t+1)$ chooses the person j such that $|S_j \cap V_t|$ is maximized (recall that S_j is the skill set of person j). We thus need to argue that there *exists a* j such that $|S_j \cap V_t| \geq |V_t|/k$.

To prove this, consider the optimal solution, say it is $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$.

We know that $S_{i_1} \cup S_{i_2} \cup \cdots \cup S_{i_k} = [m]$. I.e., together the sets S_{i_r} cover all of $[m]$. So in particular, they must also cover all of V_t. In other words, we have that $(S_{i_1} \cap V_t) \cup (S_{i_2} \cap V_t) \cup \cdots \cup (S_{i_k} \cap V_t) = V_t$.

This implies that there must exist some r such that $|S_{i_r} \cap V_t| \geq |V_t|/k$, since if the union of k sets is V_t, then one of them must have size at least $|V_t|/k$. This proves that there exists some set S_j such that $S_j \cap V_t| \geq |V_t|/k$, which proves the claim.

Next, we see how the claim proves the theorem. We can write it as $u_{t+1} \leq u_t \left(1 - \frac{1}{k}\right)$, which when applied repeatedly yields $u_T \leq u_0 \left(1 - \frac{1}{k}\right)^T$, for all T.

By definition, $u_0 = m$ (initially all the elements are uncovered). Thus we have $u_T \leq m\left(1 - \frac{1}{k}\right)^T$. Suppose we can make the quantity on the RHS <1, then since u_T is always an integer, it must be zero! So the question is: for what T does $m\left(1 - \frac{1}{k}\right)^T$ become < 1?

There are many ways to see this. One is to solve directly, and get $T > -\frac{\ln m}{\ln(1-k^{-1})}$; then one can use the approximation that $\ln(1-x) \approx x$, to get $T > k \ln m$.

Another is to use the fact that $\left(1 - \frac{1}{k}\right)^k \approx \frac{1}{e}$, and get that if $T = \alpha k$, then we need $me^{-\alpha} < 1$, which is equivalent to $\alpha > \ln m$. This completes the proof of the theorem.

The main comment here is that the optimum solution *figures in the proof* even though the algorithm has no idea about it. The argument also does not say that the algorithm picks one of the optimal sets. It only says that at every step, the algorithm picks a set that *at that point*, looks as good as any of the optimal sets.

It's a nice exercise to come up with instances where the optimum solution has no overlap with the chosen sets.

Another remark is that this analysis is tight. There exist instances (easy enough to construct) where the greedy algorithm is $\approx \ln m$ factor worse than the optimum solution. A highly non-trivial result shown around 20 yrs ago, is that in fact, **no polynomial time algorithm** can achieve an approximation ratio better than $\ln m$ (even say $0.5 \ln m$) unless P=NP (we will discuss what this means towards the end of the course; for now, you may treat this as something extremely unlikely).