1 Greedy algorithms

Today and in the next lecture we are going to discuss *greedy algorithms*. "Greedy" in this context means "always doing the locally optimal thing". E.g., a greedy algorithm for driving to some destination might be one that at each intersection always takes the street heading most closely in the direction of the destination. In general, this approach might or might not find an optimal solution (e.g., get you to the destination in the quickest way or even at all). However, what's interesting is that a large number of important problems can indeed be solved by greedy algorithms with the right notion of "locally optimal". We will discuss several examples of this in these lectures.

Greedy algorithms generally work by starting from some initial empty or "base" solution, and building it up one item at a time, by adding in the item that has the best "score" according to some measure that the algorithm defines. Typically we will prove correctness of a greedy algorithm by induction: we'll assume inductively that the solution built so far satisfies some invariant, and then show that this invariant is maintained after each step. If we can do this–and the base case satisfies the invariant too–then we immediately get that the final output also satisfies the invariant. Then, hopefully there is a simple argument to say that if the final output satisfies the invariant then it must be optimal. A nice feature of greedy algorithms is that they are generally fast and fairly simple, so (like divide-and-conquer) it is a good first approach to try.

2 Scheduling

Our first example to illustrate greedy algorithms is a scheduling problem called *interval scheduling*. The idea is we have a collection of *jobs* (tasks) to schedule on some machine, and each job j has a given start time s_j and a given finish time f_j. If two jobs overlap, we can't schedule them both. Our goal is to *schedule as many jobs as possible* on our machine.

Example: Suppose the jobs are the following 5 intervals: $[1,3], [2,4], [3,5], [4,6], [5,7]$. Then, best is to schedule the three jobs $[1,3], [3,5], [5,7]$. Any other solution will schedule at most two jobs.

Let's now consider several greedy algorithms that do *not* work. Let's see if we can find counterexamples for each. For terminology, let's call a job a "candidate job" if it has not yet been scheduled and does not conflict with any already-scheduled job.

Algorithms that do not work: Starting from the empty schedule, so long as at least one candidate job exists,

- Always add in the *shortest* candidate job.
- Always add in the candidate job with the *earliest start time*.
- Always add in the candidate job with the *fewest conflicts* with other candidate jobs.

Now, let's give an algorithm that works, and prove correctness for it.

An algorithm that works: Starting from the empty schedule, so long as at least one candidate job exists,

- Always add in the candidate job with the *earliest finish time*.

Proof of correctness: To prove correctness, we will prove the following invariant: at every step, the solution produced by the algorithm so far is a subset of the jobs scheduled in some optimal solution (i.e., it can be extended to an optimal solution without removing any already-scheduled jobs). We can prove this by induction.

This invariant is clearly true at the start, when no jobs have yet been scheduled by the algorithm. Now, assume it is true after i jobs have been scheduled, and let S be an optimal solution that includes the i jobs scheduled by the algorithm so far. Let j be the job the algorithm schedules next. If $j \in S$, then we are done—our induction is maintained. If $j \notin S$, let $j' \in S$ be the job in S with the earliest finish time that is *not* one of the i jobs scheduled by the algorithm. Notice that job j' must be a candidate job since it does not conflict with any other job in S and S includes all the i jobs scheduled by the algorithm so far. Therefore it must be the case that the finish time of j' is greater than or equal to the finish time of j. This in turn means we can modify S to maintain our invariant by removing j' from it and adding j. In particular, j does not conflict with any job in S that starts after j' finishes (since j finishes earlier) and j does not conflict with any job in S that finishes before j' starts (since by definition of j', all such jobs belong to the set of i jobs scheduled by the algorithm, and j does not conflict with any of them).

So, our invariant is true all the way through the algorithm's operation, which means that when it halts, it must have found an optimal solution (if it is a strict subset of an optimal solution, then there will still exist at least one candidate job). ■

3 Huffman codes

Huffman coding is a form of data compression used to send text more compactly. The high-level idea is the following. Suppose that you have an alphabet with some number of characters, and your text consists of sequences of letters from that alphabet. For instance, let's imagine we have 128 characters, like ASCII. We could use 7 bits to encode each one, since $2^7 = 128$. However, if some characters are used more frequently than others, we might be better off with an encoding that uses fewer bits for the frequent characters and more bits for the less frequent characters. This could give us a savings overall.

One important technical point when we go to encodings that use a different number of bits per character is we want our encoding to be *prefix-free*. This means that no character can have an encoding that is a prefix of some other character's encoding. E.g., if we use 010 for the letter "a", then we can't use 01 for the letter "b": otherwise, we wouldn't know when one letter ends and the next one begins.

One way to think of prefix-free encodings is that we are viewing our characters as leaves in a binary tree, where each edge is labeled by a 0 or a 1. Let's say that from any internal node, the left branch is 0 and the right branch is 1. For example, suppose we have four letters, "a", "b", "c", and "d". We could use a balanced binary tree with 00 for "a", 01 for "b", 10 for "c", and 11 for "d" or an unbalanced tree, say 0 for "a", 10 for "b", 110 for "c", and 111 for "d". This would be a savings if "a" is very common – more common than "c" and "d" combined.

Suppose we know the frequency of every letter in our alphabet. That is, in a sequence σ of characters, we know the fraction f_i of them that will be letter i (where $f_1 + f_2 + ... = 1$). What tree will minimize the total number of bits needed to encode σ? This is a Huffman code.

Huffman code problem:

- **Given:** frequencies f_i for each letter i in the alphabet.
- **Goal:** output the tree that minimizes the number of bits needed to encode a string in which letters appear in proportion to their given frequencies.

Building up intuition:

Suppose we have $f_i > f_j$, what does this tell us about where letters i and j are in the optimal tree? Can it be the case that i is strictly deeper in the tree than j? No. Why not? (Could just swap them).

Can the optimal tree have just a single deepest leaf? No. Why not? If there was, you could move it up to its parent and have a better tree. In fact, this reasoning also tells us that every leaf at the bottom level of the tree must have a sibling leaf.

Claim: consider two sibling leaves at the bottom level of an optimal tree. We may assume without loss of generality that they contain the two least frequent letters.

Proof: Let T be an optimal tree and consider two sibling leaves at the bottom of T. If these leaves do not have the two least frequent letters, then just swap the two least-frequent letters with the letters currently in those leaves. This will not increase the cost of T. ■

The above claim suggests the following algorithm:

Huffman code algorithm. Given frequencies $f_1, ..., f_n$ for the n letters in our alphabet:

- Find the two least-frequent letters i and j.
- Replace i and j with a single letter ij of frequency $f_{ij} = f_i + f_j$.
- Recursively solve the new problem, obtaining a tree T_{n-1}.
- Replace the leaf in T_{n-1} containing ij with an internal node whose children are leaves containing i and j.

Example: Suppose we have five letters a,b,c,d,e with frequencies $f_a = 0.45, f_b = 0.20, f_c = 0.15, f_d = 0.1, f_e = 0.1$. The algorithm will first merge d and e, replacing them with de of frequncy $f_{de} = 0.2$. We now enter our recursive call, and merge c with either b or de – let's say we merge c with b, replacing them with bc of frequency $f_{bc} = 0.35$. Next we merge bc with de, replacing them with bcde of frequency $f_{bcde} = 0.55$. Finally, we merge a with bcde and are done. The tree in the end looks like this:

At this point we need to answer two questions: (1) does this really find the best tree, and (2) how much time does the algorithm take? That is, is it correct and what is its running time? Let's look at running time first and then analyze correctness (usually you want to do this in the other order, since running time isn't so important if the algorithm doesn't give you the right answer but it's easier to consider running time first here).

Running time: Let n denote the number of letters in our alphabet. If we use a priority-queue (heap) data structure, we can find and remove the two elements i, j of lowest frequency in time $O(\log n)$ and then insert ij with frequency $f_i + f_j$ in time $O(\log n)$. It takes $O(n)$ time to build the heap initially, and then we have $n - 1$ operations of cost $O(\log n)$ for a total time of $O(n \log n)$.

Correctness: We will prove correctness by induction on the size n of the alphabet. First, to be precise, the cost of a tree T is $cost(T) = \sum_{k=1}^{n} depth(k) f_k$, where $depth(k)$ is the depth of letter k in the tree (i.e., $depth(k)$ is the number of bits used to encode letter k).

Clearly the algorithm works correctly in the base case $n = 2$, so to prove correctness for general $n > 2$ we may assume inductively it is correct on any alphabet of size $n - 1$. Let $f_1, ..., f_n$ denote our n frequencies, and let T_n be the tree produced by the algorithm. Let i and j be the two letters of least frequency merged in the first step of the algorithm, and let T_{n-1} denote the tree produced in the recursive call. By our inductive hypothesis, T_{n-1} is an optimal tree for the $n - 1$ letter alphabet in which i and j have been replaced by ij of frequency $f_{ij} = f_i + f_j$.

To prove that T_n is optimal, let us first relate $cost(T_n)$ to $cost(T_{n-1})$. Notice that the costs are almost the same, except that i and j are represented by a string (the same string) that is one bit shorter in T_{n-1}. Therefore, $cost(T_n) = cost(T_{n-1}) + f_i + f_j$.

Now, suppose for sake of contradiction that T_n was not optimal and the optimal tree T'_n had strictly smaller cost. As noted in the Claim earlier, since i and j are the letters of least frequency in the alphabet, we may assume that i and j are sibling leaves in T'_n. Let T'_{n-1} denote the tree produced by replacing the common parent of i and j in T'_n with a leaf labeled ij of frequency $f_{ij} = f_i + f_j$. As noted above, we have $cost(T'_n) = cost(T'_{n-1}) + f_i + f_j$. But since T_{n-1} was optimal, this means $cost(T'_n) \geq cost(T_{n-1}) + f_1 + f_j = cost(T_n)$, contradicting our assumption. ■