Lecture Notes: Greedy Algorithms

We will see some more examples of greedy algorithms. The main example for today is an algorithm for the popular "spanning tree" problem.

Disclaimer: These lecture notes are informal in nature and are not thoroughly proofread. In case you find a serious error, please send email to the instructor pointing it out.

Recap: greedy algorithms

Greedy algorithms, as we saw last class, are ones in which the algorithm makes a sequence of choices, where each choice is made in a *myopic* manner, choosing what is best at the current state without looking ahead. The example of matching (assigning gifts to children to maximize the total happiness) showed the issues with such algorithms. Generally, greedy algorithms are good heuristics. Understanding why they do/don't work often gives intuition about the structure of the problem (which may lead to other algorithms).

We also saw a problem in which the greedy algorithm does indeed produce the optimal solution — the problem of scheduling a collection of jobs on one machine in order to reduce the average completion time (or equivalently, the sum of the completion times).

Comment on the analysis. The challenging aspect of greedy algorithms, even when they work, is the analysis of correctness. One typically needs to come up with a reasoning that is tailored to the structure of the problem being solved.

Spanning trees

Let us start with a definition of the popular MST problem.

Minimum spanning tree (MST). Let $G = (V, E)$ be an undirected graph with edge weights. The goal is to choose a subset of the edges so that the vertices are all connected (i.e., there is a path from every vertex to every other vertex only using the chosen edges), and the sum of the weights of the chosen edges is minimized.

The problem has many natural motivations. A popular one is to view the weighted graph as giving the costs of connecting different nodes on a network, and the goal is to find a "communication backbone" for the network.

Comment. Assuming that all the edge weights are positive, the minimum weight collection of edges will form a *tree*. I.e., it will be a **connected, acyclic subgraph** of G. It is connected by requirement, and it is acyclic because otherwise we can remove an edge from a cycle and reduce the total weight, while keeping connectivity.

An example of a simple graph and its MST are shown in Figure **??**.

Algorithm for MST

What is a natural greedy algorithm for MST? The first suggestion in class was the following:

> Start with the smallest weight edge, and keep adding the edges in the increasing order of weight, until all the vertices are connected.

The problem with this algorithm is of course, that we could have edges between vertices that are already connected, and thus these edges are "redundant". The natural modification is to add edges only if they are not redundant. In fact, this algorithm turns out to produce a minimum weight spanning tree! It's known as Kruskal's algorithm.

We will skip the analysis of Kruskal's algorithm, and instead present a slightly different (but still greedy) algorithm. The idea here is to "build" a connected component of the graph, one vertex (and edge) at a time. Interestingly, the algorithm looks rather different from Kruskal's algorithm, but it also finds the minimum weight spanning tree! The following is known in the literature as Prim's algorithm.

> The algorithm has $n-1$ iterations. In iteration i, we have a subset S_i of the nodes that have been connected so far (we initialize with a singleton set $S_0 = \{u\}$, for some $u \in V$). We attempt to add one node to this set in every iteration. We do so by adding the node that has the least cost of connecting to S_i. (Ties are broken arbitrarily.) For some vertex $v \notin S_i$, the cost of connecting to S_i is defined to be the weight of the minimum edge from v to some node in S_i. If there is no such edge, we define the cost to be ∞.

This is clearly a greedy algorithm, as we are *building up a solution*, by making the choice that incurs the least cost (i.e., edge weight) in each step.

To illustrate the algorithm, consider its execution on the following two simple examples (that only differ by the weight of one edge). The order in which the edges are added are shown in the caption.

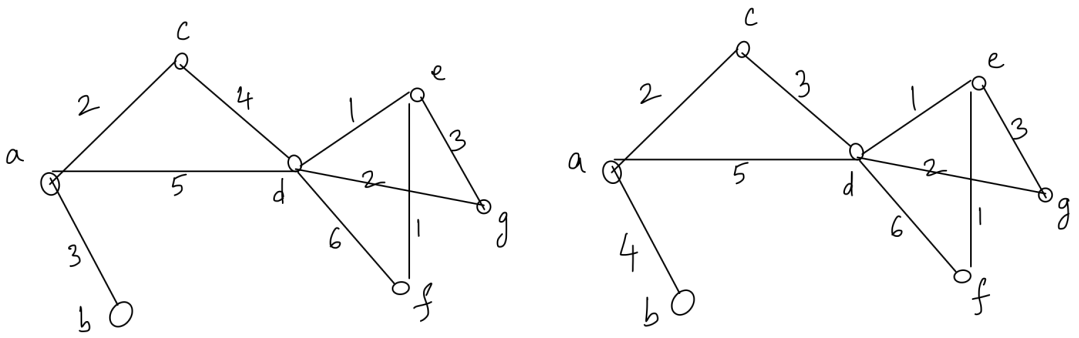

Figure 1: Prim's algorithm starting with $\{a\}$ chooses the edges in the following order for the graph on the left: ac, ab, cd, de, ef, dg, and the following for the one on the right: ac, cd, de, ef, dg, ab. In this case the *set* of edges chosen is the same in both the case. Even this need not always be true.

Analysis

Does the algorithm always compute the minimum weight spanning tree? Interestingly, it turns out that the answer is yes. Let us see how to prove it formally.

The obvious inductive statement would be something like the following:

Claim 1. *Let $\{u_1, u_2, \ldots, u_k\}$ (denoted S_{k-1}) be the vertices that are connected after the first $k-1$ iterations of the algorithm, and let the edges chosen be $e_1, e_2, \ldots, e_{k-1}$. Then for all $k \geq 1$, the edges $e_1, \ldots, e_{k-1}$ form a minimum spanning tree for S_{k-1}.*

This would clearly prove the optimality of the algorithm, as we can see by setting $k = n$. The base case is also easy to show. Now for the inductive case, suppose the statement is true with k and we wish to show

it for $k+1$. Consider the set S_k. How could the MST of S_k look like? If it so happens to look like the MST of S_{k-1} plus an edge, we could use the inductive hypothesis and be done. But this need not be the case – the tree for S_k need not contain any of the edges $e_1, \ldots, e_{k-1}$ (at least, we have not proved that it must!).

While a more involved argument can be made to make this induction work out, we prefer not to do it, and we instead present a different inductive statement.

Claim 2. *Let S_{k-1} and $e_1, e_2, \ldots, e_{k-1}$ be defined as in Claim 1. Then for all $k \geq 1$, there exists a minimum spanning tree for the* entire *graph G that includes the edges $e_1, e_2, \ldots, e_{k-1}$.*

Note that this is a strong statement: it is saying that the algorithm only picks edges that "can be completed" into a minimum weight spanning tree. Using this tree in the inductive procedure helps us avoid the need to consider a tree that can potentially look totally different, which was the problem in showing Claim 1. Let us now show Claim 2 inductively.

Proof of Claim 2. The base case of $k = 1$ is vacuous, as there are no edges. Thus, suppose that the claim is true for some k, and consider the case $k+1$. By the inductive assumption, we know that there exists a tree (call it T) that contains the edges $e_1, e_2, \ldots, e_{k-1}$, and is a minimum weight spanning tree for the entire graph G. Now, let e_k be the edge added by the algorithm next.

If T contains e_k, there is nothing to prove, as T itself can be used to show the inductive step.

Thus, suppose T does not contain e_k. Now, because T is a spanning tree of G, there must be a path from every vertex in G to every other vertex, using only the edges of T. Let the end points of e_k be ij. By the way the algorithm works, one of i, j is is S_{k-1} and the other is not. So without loss of generality, assume that $i \in S_{k-1}$. Now, consider the path in T from i to j. As the path starts in S_{k-1} and ends outside of it, there must be some edge e' in T that goes from one vertex in S_{k-1} to a vertex outside. (See Figure 2 below.)

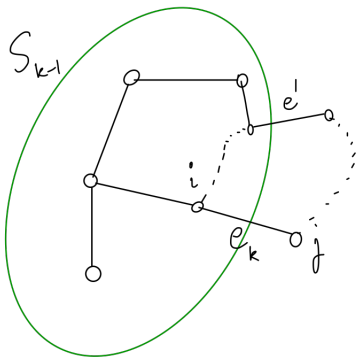

Figure 2: Figure showing S_{k-1}, e_k, e'.

Now by the way the algorithm works, $w(e_k)$ (i.e., the weight of edge e_i) is $\leq w(e')$, as e_k is the edge of the least weight that goes from S_{k-1} to $V \setminus S_{k-1}$. Thus, suppose we consider the tree T' in which we take T and replace the edge e' by e_k. It is easy to see that we still have a path between any two vertices of G, and so it is still a spanning tree (this is probably easiest to see using Figure 2 — any path between vertices that used e' can now use the "detour" that goes via e_k). Further, the total weight of the edges in T' is $\leq$ total weight of edges in T because $w(e_k) \leq w(e')$. Thus T' is also a minimum spanning tree of the graph G. Thus we found an MST containing $e_1, e_2, \ldots, e_k$, establishing the inductive step.

This completes the inductive proof of the claim. □

Summary. The algorithm for the minimum spanning tree is a natural one, but it is rather tricky to argue about. This is pretty common for greedy algorithms. The meta technique above —showing that there exists an optimum solution that is consistent with all the choices made so far— happens to be useful in other analyses as well.

Running time

We did not spend time on the running time in class. The main step in each iteration is to find the edge e_k to add. To do this quickly, it turns out we can maintain, for every vertex $u \notin S_{k-1}$, the least cost of connecting to S_{k-1}. These can be maintained in a priority queue, and it turns out that the entire procedure can be implemented in time $O((|E| + |V|) \log n)$ – working out the details of this is a simple (optional) exercise. There are slight improvements possible over this, but they are beyond our scope.