CSE 202 NOTES FOR NOVEMBER 14, 2002

Minimum Spanning Tree

As another example of a successful greedy algorithm, consider the following problem.

Instance: A weighted undirected graph $G = (V, E, w)$
Solution Format: Set of edges T.
Constraints: T connects all nodes in V.
Objective: Minimize $\sum_{e \in T} w(e)$.

What are the decision points?
Edges.

What are the options?
In T or out of T.

What happens when you make a choice for an edge?
Uh-oh. We have a problem here. When we choose an edge to be in the set T, then the new problem isn't exactly the same as the old problem: T has to span all nodes in G, but now if two nodes are connected, the new problem may or may not include one of those nodes. This is a violation of the self-similar substructure that we've been emphasizing over the last month, so how can we deal with this?

Obviously, we generalize. The first approach we can think of is to generalize the problem to be a minimum spanning forest, rather than a minimum spanning tree, with the added constraint that when all nodes in G are in the forest that T become a tree. I don't know what happens when you go down that route, but it will probably be difficult to think about; instead we will take another approach: if you decide that an edge is in the set T, then merge the two vertices that are endpoints to the edge into a single vertex—this new problem is exactly the same as the old problem, though the size of the graph is smaller by one node.

I will list the merge algorithm here and refer to it later. For the following assume that

$$w(e = \{u, v\}) = \begin{cases} \text{weight of edge } \{u, v\} \text{ if } \{u, v\} \in E \\ \infty \text{ otherwise} \end{cases}$$

Merge($G = (V, E)$, $\{u, v\}$):
Let w be a new node.
$G \leftarrow G - \{u, v\} \cup \{w\}$
for $v' \in N(u) \cup N(v)$ **do**
  $e' = \text{argmin}_{\{u,v'\},\{v,v'\}} \{w(e')\}$
  $E \leftarrow E - \{\{u, v'\}, \{v, v'\}\} \cup \{e'\}$
end for

Theorem 0.1. *If $e = \{u, v\}$ is an edge in G, $G' = Merge(G, \{u, v\})$, and T' is an optimal spanning tree for G', then $T = T' \cup \{e\}$ is an optimal spanning tree for G.*

Proof: Let T_1 be any spanning tree for G containing e. Then $T_1 - \{e\}$ is a spanning tree for G' (by assumption), and if T_2 is a spanning tree for G', $T_2 \cup \{e\}$ is a spanning tree for G (again, by assumption). Now, $\text{Cost}(T - \{e\}) = \text{Cost}(T) - w(e)$, so minimum cost edge e minimizes $\text{Cost}(T) = \text{Cost}(T' + \{e\})$ because we assume that T' is an optimal spanning tree for G'.□

So, now our easy decision point property is to select the minimum cost e in G, and the greedy option is to include it. We can prove soundness the usual way, by modifying the solution.

If e is the minimum-cost edge in G and T' is any spanning tree in G such that $e \notin T'$, we will show that there exists a spanning tree T with $\text{Cost}(T) \leq \text{Cost}(T')$ and $e \in T$. T' must contain a path $u, x_1, x_2, x_3, \ldots, v$ that connects u and v, and we can delete $\{u, x_1\}$. Let $T = T' - \{u, x_1\} \cup \{u, v\}$. Clearly T is still a spanning tree, and $\text{Cost}(T) = \text{Cost}(T') + w(e) - w(\{u, x_1\})$. Since $w(e) \leq w(\{u, x_1\})$ by our choice of e, we know that the $\text{Cost}(T) \leq \text{Cost}(T')$. What's kind of strange is that T constructed in this fashion isn't necessarily the optimal spanning tree—we would need to choose either $\{u, x_1\}$ or $\{x_k, v\}$, based on which edge costs more. Deleting either of these nodes gives us two choices for the spanning tree; what we have shown is that neither of these trees is worse than the original tree, but not that the one that we picked for the proof was optimal.

The algorithm, then, is to select the smallest e, merge the nodes to form G', and repeat on G' until there is only a single node left.

Data-Structure-Improved Version of Kruskal's Algorithm

We are assuming here that the minimum cost edges are at the front of a list, and that they are sorted by cost. So we need to augment the algorithm a little, to be:

```
Sort edges by cost
for i = 1 upto n − 1 do
  Go through edges in order:
  for ∀e_i ∈ E do
    if e_i = {u_i, v_i} ∈ T then
      next;
    end if
    add {u_i, v_i} to T.
    merge u_i and v_i.
  end for
end for
```

So the structure that we need to deal with is a partition of the set of vertices, because we will be conglomerating nodes as the algorithm progresses. That is, we need to represent disjoint sets (or rather, a set of sets whose intersection is the empty set). What we will do is figure exactly which operations we will need to perform in order to carry out this algorithm, and then try to puzzle which data structure would make the most sense for it. In doing this we will consider a couple of possible implementations which will have different effects on the running times.

What are the queries that we perform? Are the $\{u, v\}$ in the same set? Or, in stronger form, each set can be given a name, which is an element of the set called the

leader. So, given u, return the leader of u. The main query operation is FindLeader, which will take as input a node, and return as output the string representation of the leader of the sets; two nodes are in the same set if their leaders are the same.

How are the structures changing? We will change the values of two sets by merging. Merge(u, v) takes the set associated with u and the set associated with v and combines them into a single set. This gives us two potential leaders for a set, which must be dealt with. The main update operation is Merge, which takes as input two vertices and sets their leaders properly (and takes care of self loops and all that).

Our first implementation will use a lazy merge: add a pointer from one leader to the other leader at every merge. Since each set is a tree, the leader is the root of the tree. In this case, merging takes $O(1)$ time, because you simply adjust a pointer. Finding the leader is $O(d)$ where d is the depth; we haven't shown yet that the tree is balanced or binary. The rule for the merge should be to point from the shorter tree to the taller tree. At each node x, let's define three functions, $l(x)$ which returns the leader of x; $p(x)$ which returns the parent of x in the current tree; and $s(x)$, which is the size of the subtree rooted at x. Merge(u, v) is $O(1)$: it compares $s(l(u))$ with $s(l(v))$, puts a pointer at the smaller of $\{l(u), l(v)\}$ to the other, and takes care of minor bookkeeping. There are two properties which, in class, were stated as lemmas but not proved directly, so I will simply state them as fact: at every x that isn't the leader, $s(p(x)) \geq 2s(x)$; and the depth of x is less than or equal to $\log s(x)$. In this implementation, FindLeader takes $O(\log n)$ time, as does Merge. The overall running time for this algorithm, then, is $O(m \log n + n \log n) \in O(m \log n)$, since usually $m > n$.

We can improve this by having each node point directly to the leader, and maintain this relationship when we call Merge, and this will be our second implementation. At first this seems like it will be more work than the tree-of-points method, since we actually have to visit all the nodes in one of the disjoint subsets and make adjustments. However, through the magical use of amortized analysis, we will show that across all of the iterations in the algorithm this approach will not require practically more than a constant number of operations!

To see this, consider the cost of all m FindLeader operations that we will perform. We use the entire series of costs, and bound this series by some function—this is called amortized analysis.

The cost of m FindLeader operations is determined by the number of pointer changes plus the overhead of the calls. Note that, similar to the second property I asserted without proof in the first implementation, every time we change the pointer of x, $s(p(x))$ at least doubles (though it may more than double). This is because we're choosing the larger of the two disjoint sets to have the leader of the new combined set; x is in the set that is smaller than the other one, so minimally the parent of x will have more than twice as many children. Of course, it is not the leader, but this doesn't matter particularly. We'll use the term "big step" to denote the operation of adjusting a particular node's parent pointer, and "small step" to denote the operation of adjusting the set that a node is in (but not changing the leader of x). Then the central quantity of interest is $O(\#\text{ big steps}) + O(\#\text{ small steps}) + O(m)$, which is less than or equal to $m \log^* n + O(\#\text{ small steps}) + O(m)$. This is because if $\log^* s(x) = k$ then we can changes x's parent pointer as a small step less at most $A_4(2, k)$ times. I will only wave my hands here to justify this, but I believe that the

idea is that because the size of the parent of x is (at least) doubling every time we adjust the pointers, and because we only adjust the smaller set, we will only visit a node when the other sets get larger than the set that the node is in. This can't happen very often because the total number in all the sets is fixed. That this is the Ackerman function is a little harder to see, and I'm not sure I can prove this nicely.

So, you can see, at least, you can see that it might be true if I haven't convinced you completely, that we end up with a time sequence of $O(m \log^* n + n \log^* n) = O((m + n) \log^* n)$, and this implementation actually works out to be faster than the tree-of-pointers implementation.