Minimum Spanning Trees

In this lecture we will explore the problem of finding a minimum spanning tree in an undirected weighted graph (if one exists). First let's define a tree, a spanning tree, and a minimum spanning tree:

tree: A connected graph without cycles. (A cycle is a path that starts and ends at the same vertex.)

spanning tree: a subtree of a graph that includes each vertex of the graph. A subtree of a given graph as a subset of the components of that given graph. (Naturally, these components must form a graph as well. Thus, if your subgraph can't just have vertices A and B, but contain an edge connecting vertices B and C.)

Minimum spanning tree: This is only defined for weighted graphs. This is the spanning tree of a given graph whose sum of edge weights is minimum, compared to all other spanning trees.

Crucial Fact about Minimum Spanning Trees

Let G be a graph with vertices in the set V partitioned into two sets V_1 and V_2. Then the minimum weight edge, e, that connects a vertex from V_1 to V_2 is part of a minimum spanning tree of G.

Proof: Consider a MST T of G that does NOT contain the minimum weight edge e. This MUST have at least one edge in between a vertex from V_1 to V_2. (Otherwise, no vertices between those two sets would be connected.) Let G contain edge f that connects V_1 to V_2. Now, add in edge e to T. This creates a cycle. In particular, there was already one path from every vertex in V_1 to V_2 and with the addition of e, there are two. Thus, we can form a cycle involving both e and f. Now, imagine removing f from this cycle. This new graph, T' is also a spanning tree, but it's total weight is less than or equal to T because we replaced e with f, and e was the minimum weight edge.

Each of the algorithms we will present works because of this theorem above.

Each of these algorithms is greedy as well, because we make the "greedy" choice in selecting an edge for our MST before considering all edges.

Kruskal's Algorithm

The algorithm is executed as follows:

```
Let $V = \varnothing$
For i=1 to n-1, (where there are n vertices in a graph)
    $V = V \cup e$, where e is the edge with the minimum edge
                  weight not already in V, and that does NOT
                  form a cycle when added to V.
Return V
```

Basically, you build the MST of the graph by continually adding in the smallest weighted edge into the MST that doesn't form a cycle. When you are done, you'll have an MST. You HAVE to make sure you never add an edge the forms a cycle and that you always add the minimum of ALL the edges left that don't.

The reason this works is that each added edge is connecting between two sets of vertices, and since we select the edges in order by weight, we are always selecting the minimum edge weight that connects the two sets of vertices.

In order to do cycle detection here, one idea is to keep track of all the separate clusters of vertices. Initially, each vertex is in its own cluster. For each edge added, you are merging two clusters together. Indicate this by changing a variable that stores the cluster ID values of a vertex to be the same as every other vertex in the cluster. An edge can NOT be added in between two vertices within the same cluster.

Carefully analyzing this description shows that a Disjoint Set precisely provides the necessary functionality. Initially, each set in the disjoint set keeps track of each of the connected components as we're building the MST. If two vertices are already connected, they will be in the same exact set in the Disjoint Set. Alternatively, if they aren't connected, this represents an edge we can add to connect two disjoint trees that is safe to add to our minimum spanning tree.

Prim's Algorithm

This is quite similar to Kruskal's with one big difference:

The tree that you are "growing" ALWAYS stays connected. Whereas in Kruskal's you could add an edge to your growing tree that wasn't connected to the rest of it, here you can NOT do it.

Here is the algorithm:

1) Set S = ∅.
1) Pick any vertex in the graph.
2) Add the minimum edge incident to that vertex to S.
3) Continue to add edges into S (n-2 more times) using the following rule:

    Add the minimum edge weight to S that is incident to S but that doesn't form a cycle when added to S.

Once again, this works directly because of the theorem discussed before. In particular, the set you are growing is the partition of vertices and each edge you add is the smallest edge connecting that set to its complement.

For cycle detection, note that at each iteration, you must add exactly one vertex into the subgraph represented by the edges in S. (You can think of "growing" the tree as successively adding vertices that are connected instead of adding edges.) Thus, we can just keep a boolean array of storing which vertices have been used and required that the edge that we add into the tree has exactly one vertex that hasn't been visited yet.