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. The data structure that allows us to efficiently take care of this bookkeeping is a disjoint set.

Disjoint Sets

A disjoint set contains a set of sets such that in each set, an element is designated as a marker for the set. Here is a simple disjoint set:

{1}, {2}, {3}, {4}, {5}

clearly there can only be one marker for each of these sets. Given a disjoint sets, we can edit them using the union operation. For example:

union(1,3) would make our structure look like:

{1,3}, {2}, {4}, {5}

Here we would have to designate either 1 or 3 as the marker. Let's choose 1. Now consider doing these two operations:

union(1,4)
union(2,5) (Assume 2 is marked.)

Now we have:

{1,3,4}, {2,5}

Now, we can also do the findset operation.

findset(3) should return 1, since 1 is the marked element in the set that contains 3.

Disjoint Set Implementation

A set within disjoint sets can be represented in several ways. Consider {2, 4, 5, 8} with 5 as the marked element. Here are a few ways that could be stored:

```
   5              5            5
 / | \           /   \          |
2  4   8        2     8         8
                |              / \
                4             4   2
```

We can actually store a disjoint set in an array. For example, the sets {2,4,5,8}, {1}, {3,6,7} could be stored as follows:

1	5	7	5	5	7	7	2
1	2	3	4	5	6	7	8

The 5 stored in array location 2 signifies that 5 is 2's parent. The 2 in array location 8 signifies that 2 is 8's parent, etc.

Here is the visual display:

```
1          5                7
         /     \          /    \
        2       4        3      6
        |
       8
```

Based on this storage scheme, how can do implement the initial makeset algorithm and how can we implement a findset algorithm?

Union Operation

Given two values, we must first find the markers for those two values, then merge those two trees into one.

Consider union(5,1). We could do either of the following:

```
   1     5
   |    / | \
   5    2 4 1
  /  \  |
 2  4   8
 |
 8
```

We prefer the latter since it minimizes the height of the tree. Thus, in order to implement our disjoint sets efficiently, we must also keep track of the height of each tree, so we know how to do our merges. Basically we choose which tree to merge with which based on which tree has a smaller height. If they are equal we are forced to add 1 to the height of the new tree.

Here is how our array will change for each of the options above:

First option

1	5	7	5	1	7	7	2
1	2	3	4	5	6	7	8

Second option

5	5	7	5	5	7	7	2
1	2	3	4	5	6	7	8

Notice how quickly we can implement that change in the array!

Path Compression

One last enhancement we can add to disjoint sets is path compression. Every time we are forced to do a findset operation, we can directly connect each node on the path from the original node to the root. Here's the basic idea:

```
    1         final tree is          1
    |                              / | \
    5                             2  8  5
   / \                                  |
  2   4                                 4
  |
  8
```

1	5	7	5	1	7	7	2
1	2	3	4	5	6	7	8

First, you find the root of this tree which is 1. Then you go through the path again, starting at 8, changing the parent of each of the nodes on that path to 1.

1	5	7	5	1	7	7	1
1	2	3	4	5	6	7	8

then, you take the 2 that was previously stored in index 8, and then change the value in that index to 1:

1	1	7	5	1	7	7	1
1	2	3	4	5	6	7	8

It has been shown through complicated analysis that the worst case running time of t operations is $O(t\alpha(t,n))$. Note that $\alpha(t,n) \leq 4$ for all $n \leq 10^{19728}$, so for all practical purposes on average, each operation takes constant time.

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 = ∅. Create a priority queue of edges Q = ∅.
2) Pick any vertex in the graph.
3) Add the minimum edge incident to that vertex to S, and add all edges adjacent to S into Q.
4) 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.**

We do this (cycle detection) by deleting the minimum from Q until we get an item that connects a vertex from within S to a vertex out of 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.