Matthew Ulmer

11/14/17

HW3

For this homework assignment I decided to find all minimum spanning trees in a graph by first using a recursive Kruskal's algorithm and by using a recursive Prim's algorithm for extra credit.

**Kruskal's Implementation**

For my Kruskal approach, I start by creating a forest the size of the graph using the DisjoinSet class, a priority queue for all the edges in the graph, a blank spanning tree, and a hash set, unique, to determine whether or not the current tree has already been found. These parameters along with the graph and an empty list of spanning trees is passed into the method findTreesAux which recursively finds all minimum spanning trees of a graph.

**findTreesAux**

This method starts off with a while loop that ends once there are no edges left in the queue. After taking an edge out of the queue, I check to see if there are any other edges in the graph that have identical weight. If that is the case, I branch off here and make multiple paths. Otherwise there is no point for branching off as that edge is a key edge for that path and must be included. But before officially branching off, I check to see if the target and source of the edge are in different forests. If they are in the same forest, there is no point in doing anything and I just look at the next edge of lowest weight. If they aren't, I create a copy of the current tree, a copy of the current edge queue, and a copy of the current state of the forest. In order to make the copy of the forest I ended up creating a new constructor in the DisjointSet class. That code is displayed below.

```
public DisjointSet(DisjointSet disjoint){
    s_root = new int[disjoint.s_root.length];
    for(int x=0; x<disjoint.s_root.length; x++){
        s_root[x] = disjoint.s_root[x];
    }
}
```

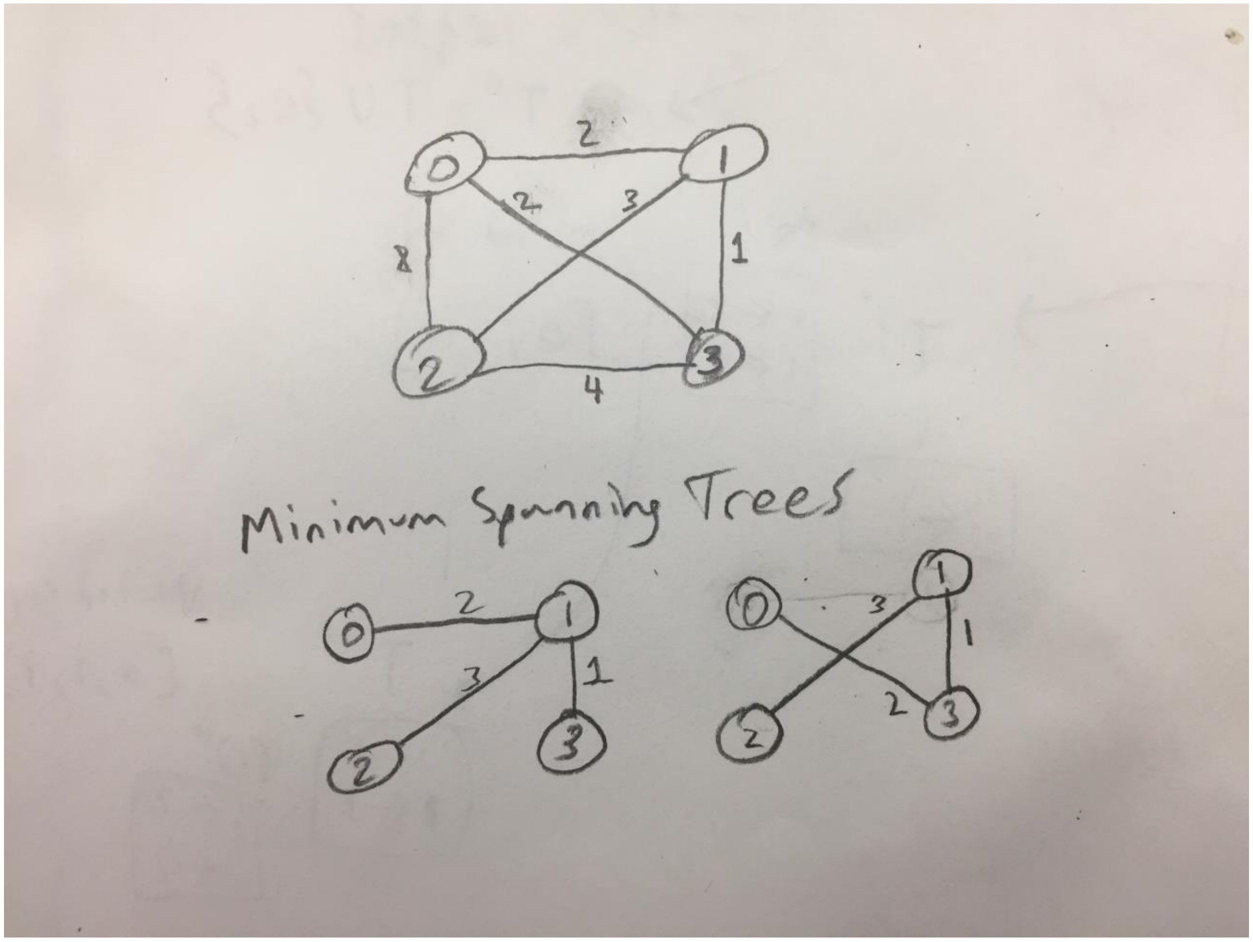
These copies are made in case there are multiple edges with the same weight that connect two separate forests. This way I can traverse down both paths of spanning tree generation. Next I add the edge to the treeCopy and then check the size and if it has all of the vertices in the graph the verifyTree method is called to determine if it should be added to the list of minimum spanning trees to be returned. This method first checks to determine if any trees have been found yet. If not, then the tree is added to trees and it's undirected sequence is added to the hash set of strings. In this situation there is no need to compare any weights as it is the first tree found. As it is the first tree found, it is also a minimum spanning tree. In future calls to verifyTree the weight of the tree is compared to the first minimum spanning tree found. If the weights are equal there is one more step that must be accomplished before adding that tree to the list of trees and that is to check if the current tree's unique undirected sequence has been added to the hash set yet. If not it is a new minimum spanning tree and is added to trees and the hash set. Otherwise it is just an already found minimum spanning tree, but with source and target of at least one edge being swapped.

**Prim's Implementation**

My Prim's approach is very similar to my Kruskal one in that I use recursion on a findTreesAux method to find a subset of the total amount of spanning trees that contains all minimum spanning trees and filters those minimum spanning trees out by using a verifyTree method. The findTreesAux method is essentially the same, except I am not making copies of a disjoint set. Instead I am making copies of a set of visited nodes. Also, instead of updating the forest with newly added nodes, I am now updating the visited list as well as adding incoming edges to newly added nodes to the queue as long as the edge connects to a node currently not in the tree.

**Extra Credit Graph**

```
Graph graph = new Graph(4);
graph.setUndirectedEdge(0, 1, 2);
graph.setUndirectedEdge(0, 2, 8);
graph.setUndirectedEdge(0, 3, 2);
graph.setUndirectedEdge(1, 3, 1);
graph.setUndirectedEdge(1, 2, 3);
graph.setUndirectedEdge(2, 3, 4);
```


0
1
2
3
2
2
3
8
1
4
Minimum Spanning Trees
0
1
2
3
2
3
1
0
1
2
3
3
1
2