Project 3 Kruskal's algorithm

Bosco Han *y95han* *20651352*

Overview: In this project, there are two .CPP files. I use an adjacency list to hold the weighted edges inserted.
DisjoinSet.cpp: this class models the disjoin set and its related methods to compute the MST. Inside this class, there is also two structs I used to model each vertex and edge. Struct Node and Edge, respectively.

```
struct node {
    int parent;
    int rank;};
struct Edge {
    int src;
    int dst;
    double weight;};
```

Kruskaltest.cpp: this class parses the input parameters, doe the necessary splitting, error handling and calls the related methods in DisjoinSet class to compute the MST. This file has been consistent with previous projects.

The **UML diagram** of the class representing disjoint set is shown below:

The design decision I took is to put all the necessary work such as initiating the graph, inserting a weighted edge, deleting edges, getting degrees of vertices, clear, edge count and computing MST into the DisjointSet class. The constructor is called when the command **n** is given.

```
DisjointSet (int vertices) {
    this->V = vertices;
    this->E = 0;
    dsuf.clear();
```

```
        adjEdgeList.clear();
        adjList = new list<int>[vertices];
        dsuf.resize(vertices);
        for (int i = 0; i < vertices; i++) {
            dsuf[i].parent = -1;
            dsuf[i].rank = 0;
        }
    }
```

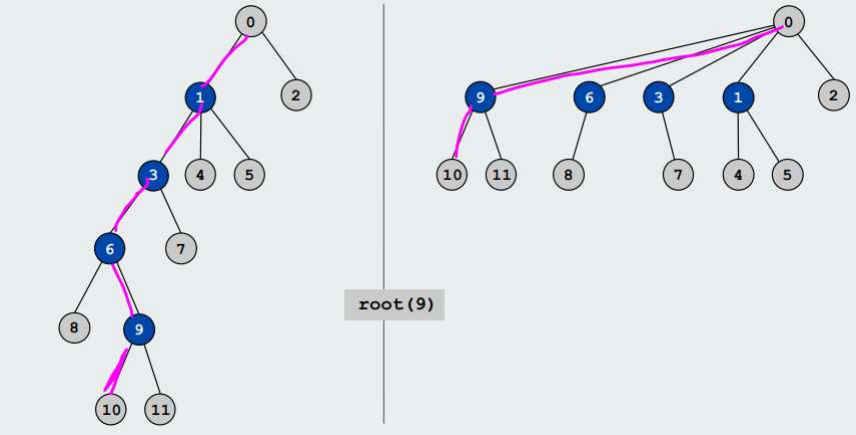
Essentially, the **constructor** sets the class variable V, and E, which keeps track of the vertices and edges for the current graph. I have also initiated vector<node> dsuf, which holds the parents and ranks of all vertices. This is used in the union-find operation. Vector<Edge> adjEdgeList is used to hold all edges from src -> dest with a weight attached. Because I have encapsulated all the edge attributes in struct Edge, doing any sort of DFS/BFS traversal on it will be cumbersome. Thus list<int> *adjList is used to hold list<vertex, List<vertex>> that has been added to the graph, it behaves like a traditional adjacency list for graph representation. Node that adjList is completely in sync with adjEdgeList as they both contain information on edges and vertex relations, however adjList's sole purpose is DFS/BFS.
On every **Insert,** addWeightedEdge() is called. Because the adjList can be used for a directed graph, I do the same insert operation twice into adjList. To represent an undirected graph, I add directions on my edges going both ways. From->to and to->from. This is so that DFS/BFS operations could traverse to and from neighboring nodes regardless of edge direction. The time complexity of this operation is O(1) as its just appending to lists.

```
        adjList[from].push_back(to);
        adjList[to].push_back(from);
```

On **Delete**, edges are removed from adjList and adjEdgeList, and that double directed edge I added into adjList is deleted twice to ensure that no connection exists between nodes 'from' and 'to': adjList[from].remove(to); and adjList[to].remove(from); adjEdgeList's removal is handled using the remove_if() method. When the size of adjEdgeList decreases by at least 1, the deleteEdge() method returns successful. The operation delete has an upper bound of O(E) as the remove_if() operation checks a condition on every edge held in adjEdgeList.
Degree loops thru elements of adjEdgeList and counts the occurrence of the vertex's appearance as either a src or dst vertex. The run time is proportional to the size of adjEdgeList, O(E), where E = # of edges inserted.
EdgeCount simply retrieves the class variable E, assuming it is updated correctly from other operations. O(1) time.
Clear operation clears the class variables and empties adjList and adjEdgeList. This is an O(1) process.

The **MST** operation triggers a series of events. First, I check if the graph is connected. This is done thru a BFS traversal of all the vertices. In the BFS traversal, a reference visited set of vertices is passed in, if the BFS returns with all vertices checked in the visited set, it means the graph is connected and I continue to compute the MST.

First, adjEdgeList is sorted on ascending edge weight. Next, loop thru the sorted edges, for each entry in adjEdgeList, I find the absolute parent of the source and destination vertex using the **find()** method. Initially, each vertex's parent value was initialized at -1. The first time find() method gets called, the parent value changes to the node itself. As union method gets utilized, the absolute parent of nodes will change and as a result, miniature spanning trees will be formed. Essentially, the find method recursively finds which set a vertex belongs to by following the parent node (dsuf[v].parent) of each vertex until a self loop is reached. Note that in my code, I am using path compression, so the find method also will set the parent of nodes to the absolute parent (parent of the parent of the parent…etc) as it traverses the call stack. Thus, keeping the tree flat.



Above is an example of path compression, here node 0 is the absolute parent. When future find() operation is called on node 10 for instance, the call stack would have to make 5 hops when not path compressed. With compression, it cuts down the find time to approximately O(log n), where n is the number of nodes in the set.

The **union_set()** method takes in the absolute src and absolute dest as two vertices, compares the ranks of two subsets represented by the vertices and mark them as one set. The set with a higher rank becomes the parent of the set with a lower rank. If the absolute source is found to equal to the absolute parent, they are skipped as they already belong to the same set, calling union on them will create a cycle. After union operation is done, the edge could be added to the minimum spanning tree list. Finally to get the result of **mst**, loop thru each element in the min spanning tree list and sum up their weights as now they contain all the crucial edges needed to form a min spanning tree with the lowest cost. The run time for weighted union find with path compression is *(M + N) lg* N*, for M operations and N nodes. In my code, M is the minimum of vertices and edges and lg *N signifies the number of times needed to take the log of a number until reaching 1. Since lg * N is a constant value, the performance of this algorithm is linear in practice.

I believe I have the best possible run time for implementing Kruskal's algorithm. The proof of the weighted union find with path compression run time is not very straightforward, thus here is the source in which I obtained the time complexity (page 31): https://www.cs.princeton.edu/~rs/AlgsDS07/01UnionFind.pdf

The **test cases** I considered while designing this included:

- all the cases that would cause an invalid argument to return, such as referring to nodes outside the valid range, and adding weights <= 0.
- Adding self edges and making sure that my code still behaved properly.
- Because of the nature of my design is using a directed approach to implement an undirected graph, I have tested inserting edges with the source and destination swapped.
- Tested that deleting works with my doubly directed edges. For ex. 3 <-> 6, If I were to call delete on 6;3, it behaves the same way as calling delete on 3;6