Project 4 – Design Description

Tony Shen

1. Overview of Classes

For this project, I implemented two header files, each with their own primary class. We will start with *disjoint.h*, which utilized class disjoint. This class provided all the required variables and functions to allow manipulation of disjoint sets for our application of Kruskal's algorithm. This class included pointers to pointers to two structs, node and node_info.

disjoint		
nodes	node**	
sets	node_info**	
set_counter	int	
initial_num_sets	int	
find_set	(int item)	int
num_sets	()	int
union_sets	(int index1, int index2)	void

Member Variables

- **nodes**: linked list element, struct containing content, pointers to_info and next
- **sets**: keeps track of set information, pointers head and tail, size of the set
- **set_counter**: number of sets
- **initial_num_sets**: initial number of sets from first input command (n #)

Member Functions

- **find_set**: returns index of the first node inside set that contains the node searching for
- **num_sets**: returns the number of sets remaining
- **union_sets**: combines two sets, required for Kruskal's algorithm

In our file *tree.h*, we utilized class tree. This class allowed us to perform the required operations described in the project requirements on our spanning tree, setting up the tree then finding its MST.

tree		
INF	static const double	
graph	double**	
num_nodes	int	
num_edges	int	
degrees	int*	
i	(int u, int v, double w)	bool
d	(int u, int v)	bool
degree	(int u)	int
edge_count	()	int
clear	()	void
mst	()	pair<double, int>

Member Variables

- ***INF***: maximum of double type, used for initial values
- **graph**: pointer to a pointer to a 2d array storing weights between edges

- **num_nodes**: maximum number of nodes in the tree from first input command (n #)
- **num_edges**: number of edges currently in the graph
- **degrees**: pointer to an array containing the degrees of each node in the graph

Member Functions

- **i**: inserts a new edge with weight w between nodes u and v
- **d**: deletes the edge between nodes u and v
- **degree**: returns the degree of the specified node
- **edge_count**: returns the number of edges connected in the graph
- **clear**: clears everything, resetting weights back to *INF*, num_edges to 0
- **mst**: calculates the weight of the mst in a connected graph

In addition to tree, the file *tree.h* also includes the class illegal_argument. This is an empty class, used in the main function in a try catch block, indicating an instance in which a failure should be outputted.

2. Constructors/Destructor/Operator overloading

For class disjoint, the constructor initialized the arrays of nodes and sets (for initial_num_sets entries). This was done by first initializing the nodes array, setting the values pointed to by the sets array to match this value (setting each initial set size to 1 since all nodes are separated at first). Then the nodes array was connected through the pointer to_info. The class destructor was very straightforward, deleting the array elements then deallocating all their memory.

For class tree, the constructor made sure to first check for m <= 0, which would result in an output of "failure". We set num_nodes to m, then initialized the graph and degrees arrays. The initial value of num_edges, as well as each array entry in degrees was set to 0, since no edges were inserted yet. Each entry in the 2-D array graphs was set to *INF*, our representation of an initial value (for a graph not connected. The class destructor was again implemented in a straightforward fashion, deallocating and deleting all the memory used by the graph and degrees arrays.

3. Test Cases

Test 1: Creating multiple smaller trees but not connecting them, ensure mst outputs "Not Connected"

Test 2: Updating an edge value and ensuring that the function uses this updated value

Test 3: Filing up a large graph to its capacity, have the mst involve the max number of edges (by making the other weights significantly larger)

Test 4: Connect all nodes to one single node, mst should be the sum of all these weights

4. Performance

Looking at our mst running time:

- put all edge information in graph: O(n) +
- sort edges ascending: O(nlog(n)) +
- connect necessary sets together: O(n) **= O(nlog(n) + n) – Kruskal's Algorithm**