Design Document: Project 3

Overview of Classes:

WeightedGraph

Defines a WeightedGraph object data structure. It keeps track of the number of nodes (vertices) and edges in the graph, as well as the degree of each edge. It also holds the edges in two other different structures: a matrix and an array of linked lists (each list is made of edges belonging to a particular vertex). It is fully responsible for managing both data structures. Its methods can add, remove, or verify that edges exist, or get the degree of a vertex. The mst function sorts all the entries into a minimum spanning tree using Prim’s algorithm, and prints the sum of all edge values in that tree. It is dependent on the illegal\_exception class.

illegal\_exception

A custom error object, it is responsible for creating an “illegal\_exception” when thrown. It is a dependency of the WeightedGraph class.

Implementation according to design:

I used both a default and overloaded constructor for WeightedGraph. The default constructor assigns all pointers to nullptr. The overloaded constructor sets the pointers to point to correctly sized arrays, based on the number of nodes to be used in the graph. Both constructors also set the number of nodes and edges to 0. The destructor for the WeightedGraph deletes the matrix using the standard procedure for deleting a double pointer array.

The weight matrix approach was used to organize the edges since its structure makes most of the methods simple. The addEdge function takes in 2 nodes and a weight and overwrites the corresponding weight value in the matrix. It also increments the degree of each node (held within the nodeDegrees array), as well as the number of edges if the edge is new. Likewise, the eraseEdge overwrites the corresponding matrix values to 0, and decrements the degree of each node. Similarly, the clearEdges function loops through all items in each linked list, deleting every element, and sets all entries in the matrix to 0 using 2 nested for loops. GetNodeDegree loops through the entries of the specified row of the matrix, and increases the degree for every number above 0 (representing an edge between the specified vertex, and the vertex in that column). It should be noted that these functions also verify that the input is valid, and some, where applicable, ensure the given edge exists in the tree.

The MST function is an implementation of Prim’s algorithm that executes in O(V2) time. I am aware that this is not the optimal implementation, but after many failed attempts, this was the best function I could come up with that consistently satisfied the test cases. The ideal implementation would involve creating a binary min heap where each node is a vertex object. Each vertex would have a key (corresponding to weight) and a parent. You would continually extract and delete the vertex with the smallest key. For each node with an edge to that vertex, if that node is in the queue, and the weight of the vertex is less than that node’s key, set the node’s key to the weight of that vertex, and set the parent of that node to the vector at the other end of that vertex. For my best attempt at implementing an algorithm similar to this, see the included “Best Attempt at O(ElogV)” folder.

I kept everything dynamic (not const), as all values needed to change from the original values assigned to them by the default constructor. The only public variable is numEdges and since it needs to be read from main. The num nodes, nodeDegrees array, and weight matrix do not, so they have all been set to private for safety.

Test Cases

I ran all methods from the weighted graph class when the list was empty, to ensure there was no unexpected behavior. I also ensured that methods threw an illegal argument exception and printed “illegal argument” when any methods were given invalid inputs. I tested checkAdjacent, getNodeDegree, and eraseEdge, and mst with vertices that were not in the tree, and tested checkAdjacent and eraseEdge with two identical nodes to ensure this behavior was triggered, and the next command was executed. I also tested mst with an unconnected graph. After compilation, I used Valgrind to verify that there were no memory leaks in any of my classes.

Design Evaluation

For the following evaluations, V is the number of vertices (stored in the program as numNodes).

Any call to MST would run 3 of its for loops V times. The first loop initializes all values of the key and inMST arrays (an θ(1+1) time task), taking θ(V) time overall. The second for loop has 2 other for loops nested within it, each performing an θ(1) operation V times. So, each nested for loop takes θ(V) time. Adding these, this means that the for loop that contains them takes θ((2V)2) = θ(V2) time. The third for loop, similar to the first one, runs an θ(1) operation V times to add all key values, and thus takes θ(V) time. Thus, the time complexity of MST is the sum of all the for loops on this level, the time complexity of is θ(V + V2 + V) = θ(V2). It would be possible to implement this algorithm in O(ElgV) time using the algorithm previously outlined. In that case, the second for loop would be replaced by what is equivalent to a heapify command (order lgV) inside a loop that runs E times, for a total runtime of O(ElgV).

Retrieving an edge count from the weighted graph class is an θ(1) operation, since it is simply printing out the value of the numEdges variable, a pre-calculated attribute of the weighted graph class. Likewise, retrieving the degree of a node is a θ(1) operation, since it only involves checking the input against a set of conditions (θ(1)), then returning the value of the nodeDegrees array at the specified index (θ(1)).

These runtimes are of the same order under a “best case”, “worst case”, and “average case” scenario. In other words, there is no best or worst case, since each method performs all operations every time.

UML Class Diagram

