Homework 8

Out: 11.30.23
Due: 12.6.23

1. [MST, 16 points]
Each of the following algorithms takes a connected weighted graph G(V,E) as input, and returns a set of edges T. For each algorithm, either explain why T is a minimum spanning tree or give a counter-example.

a. MAYBE-MST-A(G)

```
T = empty
for each edge e, taken in arbitrary order
    if T ∪ {e} has no cycles
        T = T ∪ {e}
return T
```

b. MAYBE-MST-B(G)

```
sort the edges into non-increasing order of edge weights w
T = E
for each edge e, in non-increasing order by weight
    if T – {e} is a connected graph
        T = T – {e}
return T
```

2. [MST, 12 points]
Give an algorithm to find a maximum spanning tree in a connected undirected graph.

3. [Single-Source Shortest Path, 12 points]
Design an efficient algorithm that outputs the overall number of paths that exist in a given directed acyclic graph. Analyze the runtime of your algorithm.

4. [Single source shortest path, 10 points]
Bellman Ford's shortest path algorithm has a runtime of Θ(VE). Explain why this is the case, and suggest a method of detecting whether the algorithm may stop early.

5. [Graphs, 50 points]
The *Graph* class, specified in the header file *Graph.h*, represents an undirected, weighted graph. Also provided is a sample *main.cpp* file, and a sample input file *graph.txt*. These demonstrate how we will test your code.

The input file is organized as follows:

- The first line indicates the (# of vertices) (# of edges) (type of graph)
- The (# of vertices) indicate the names of the vertices, i.e. if the (# of vertices) is 4, then the names of the vertices are {0, 1, 2, 3}.
- The rest of the lines indicate the (source vertex) (end vertex) (weight).

The *Graph* class in *Graph.h* contains the following members and methods:

- numVertices – The number of vertices in the graph.
- numEdges – The number of edges in the graph.
- vector <pair<int, int>> *adj – An adjacency list representation of the graph: every vertex has a vector of pairs containing the adjacent vertex number and weight of the edge connecting to it.

- ~~Graph() – Constructs a graph with no edges or vertices.~~
- ~~initVertices(int V) – Initializes the graph to contain V vertices.~~
- ~~getNumVertices() – Returns the number of vertices in the graph.~~
- ~~setNumEdges(int E) – Sets the number of edges in the graph.~~
- ~~getNumEdges() – Returns the number of edges in the graph.~~

- ~~addEdge(int v, int u, int weight) – Adds an undirected edge with the provided vertices and weight to the graph. The vertices of the new edge must currently exist in the graph.~~

- ~~print() – Prints a human-readable version of the adjacency list of the graph. Format is: vertex: adjacent_vertex_1 (weight_1) adjacent_vertex_2 (weight_2) ...~~

- ~~generateGraph(string fileName) – Constructs a graph from the file with the provided name. The file format is as follows: The first line contains the number of vertices and the number of edges, separated by a space, followed by optional additional text. The graph is assumed to contain vertices numbered 0 to 'number of vertices' - 1. Each of the remaining lines contain one edge specified by the source and destination vertices followed by the weight, and separated by spaces.~~

a. ~~Implement the Graph class in *Graph.cpp*, according to the above specifications. Do not modify any existing code in *Graph.h* (you may add members as you see fit.) Submit your modified *Graph.h* and *Graph.cpp*. Your code should compile with the provided files on the lab computers.~~

Your program should be executed from the command-line with an input file name as argument. It should print out the output as in the sample below (listing source vertex to end vertices and their respective weights in parentheses).

```
0: 1 (4)  2 (6)
1: 0 (4)  2 (2)  3 (15)
2: 0 (6)  1 (2)  3 (13)
3: 1 (15)  2 (13)  4 (3)  5 (1)
4: 3 (3)  5 (9)
5: 3 (1)  4 (9)  6 (11)
6: 5 (11)
```

b. Define and implement a public *modifiedDijkstra(int source)* method for the *Graph* class. This method should compute and print the weight and number of shortest paths from provided source vertex to all other vertices of the graph. You may assume that the graph is guaranteed to have non-negative weight edges, and that source is a vertex of the graph. The method should work for any source vertex. Use a maximum integer value to represent +∞ when initializing your distance array.

In a comment at the top of your method, provide a brief description of how you modified Dijkstra's shortest path algorithm on an undirected graph with non-negative edge weights to count the number of shortest paths from a given source s to a destination node d.

Do not modify any existing code in *Graph.h* (you may add members as you see fit.) Submit your modified *Graph.h* and *Graph.cpp*.

Your program should be executed from the command-line with an input file name as argument. It should print out the output as in the sample below.

```
0: 1 (4)  2 (6)
1: 0 (4)  2 (2)  3 (15)
2: 0 (6)  1 (2)  3 (13)
3: 1 (15)  2 (13)  4 (3)  5 (1)
4: 3 (3)  5 (9)
5: 3 (1)  4 (9)  6 (11)
6: 5 (11)

Shortest paths from node 0:
Distance to vertex 1 is 4 and there are 1 shortest paths
Distance to vertex 2 is 6 and there are 2 shortest paths
Distance to vertex 3 is 19 and there are 3 shortest paths
Distance to vertex 4 is 22 and there are 3 shortest paths
Distance to vertex 5 is 20 and there are 3 shortest paths
Distance to vertex 6 is 31 and there are 3 shortest paths
```