Csci 335 Assignment 4

Due April 30, 2019

Graph Representation (20 points)

You will read a directed graph from a text file. Below is an example:

Graph1.txt

```
5
1 2 0.2 4 10.1 5 0.5
2 1 1.5
3 2 100.0 4 50.2
4
5 2 10.5 3 13.9
```

First line is the number of vertices. Each vertex is represented by an integer from 1 to N. Each line is of the form

<vertex> <connected vertex 1> <weight 1> <connected vertex 2> <weight 2> …

For each vertex you have a list of the adjacent vertices with positive edge weights. For instance, in the above example, vertex 1 is connected to vertex 2 (edge weight 0.2), to vertex 4 (edge weight 10.1) and to vertex 5 (edge weight 0.5). Vertex 2 is connected to vertex 1 (edge weight 1.5), vertex 4 has no outgoing edges, etc.

Represent a graph using an adjacency list. In order to test your implementation you will also read a second text file (let us call it AdjacencyQueries.txt) that will contain a set of pair of vertices. Your program (name it CreateGraphAndTest) will have to first create the graph by reading it from text file Graph1.txt. It will then open the file AdjacenyQueries.txt and for each pair of vertices in it you will cout whether the vertices are adjacent or not, and if they are you will cout the weight of the edge that connects them.

For example if the file AdjanceQueries.txt contains
4 1
3 4
1 5
5 1
1 3
Then the output should be
4 1: Not connected
3 4: Connected, weight of edge is 50.2
1 5: Connected, weight of edge is 0.5
5 1: Not connected

1 3: Not connected

So, your program can be called for example as:

./**CreateGraphAndTest** Graph1.txt AdjacencyQueries.txt

Dijkstra's Algorithm Implementation 1 (40 points)

Implement Dijkstra's Algorithm, **using a priority queue (i.e. heap).**

Write a program that runs as follows:

`./FindPaths <GRAPH_FILE> <STARTING_VERTEX>`

This program should use Dijkstra's Algorithm to find the shortest paths from a given starting vertex to all vertices in the graph file. The program should output all paths in the form:

Destination: Start, V1, V2, … , Destination, Total cost: X

You should print out the paths to every destination.

For example if you run the program having as input Graph2.txt (provided) starting from vertex 1, i.e.

./FindPaths Graph2.txt 1

Then the output should be

1: 1, Cost: 0.0.

2: 1, 2, Cost: 2.0.

3: 1, 4, 3, Cost: 3.0.

4: 1, Cost: 1.0.

5: 1, 4, 5, Cost: 3.0.

6: 1, 4, 7, 6, Cost: 6.0.

7: 1, 4, 7, Cost: 5.0.

Simulating a random graph (40 points)

Create a program that will generate a random undirected graph of N vertices. In order to achieve that you will generate pairs of random numbers (i1, i2) with 1 <= i1 <= N, and 1 <= i2 <= N, and you will add the edge (i1, i2) to the graph. Keep also the sets of connected vertices. In order to achieve this use the Union-Find data structure we described in class. Before you start adding vertices you will have N sets in

your Union-Find data structure, one set for each vertex. When an edge (i1, i2) is added to your graph, then the sets of vertices i1 and i2 should be united. Keep adding random edges until all the vertices are connected (i.e. you end up having one set in your Union-Find data structure).

At the end of program display the following information:

a) The number of edges that your final graph contains.
b) The smallest out-degree for a vertex.
c) The largest out-degree for a vertex.
d) The average out-degree for a vertex.

This exercise should demonstrate that graph does not have to be dense in order to have full connectivity. That means that you shouldn't expect for the total number of edges to be O(N^2).

For the implementation you can use the graph data structure you have created for the previous parts. Note that since this is an undirected graph, whenever you add an edge i1->i2, you should also add the edge i2->i1. You can set the weight of all edges to 1.0. Also note that you should not add an edge more than once (i.e. before adding an edge you should check whether the edge is already part of the graph, and you shouldn't add if it is).

In order to generate random numbers between 1 and maximum_value, you can use something like the following (it will generate a sequence of random integers with values from 1 to 1000)

```
#include <cstdlib>
#include <ctime>
using namespace std;

srand(time(0)); //use current time as seed for random generator
const int maximum_value = 1000;
while (true) {
  const int random_variable = rand() % maximum_value + 1;
  cout << random_variable << endl;
  // break from loop based on a condition.
}
```

Your program should run as follows:

./**TestRandomGraph** <maximum_number_of_nodes>

For example to check for a graph of 100 nodes you should run

./**TestRandoGraph** 100