Project 5 – Design Description

Tony Shen

1. Overview of Classes

For this project, I implemented the header file `undirectedGraph.h`, using `class undirectedGraph`. This class allowed us to set up our undirected graph, then perform all the required operations outlined in the project description, including our implementation of Dijkstra's algorithm.

undirectedGraph		
nodeCount	int	
edgeCount	int	
adjacencyMatrix	double**	
degrees	int*	
cities	string *	
i	(string name)	void
setd	(string name1, string name2, double d)	void
s	(string name)	int
degree	(string name)	void
graph_nodes	()	int
graph_edges	()	int
d	(string name1, string name2)	double
shortest_d	(string name1, string name2)	double
print_path	(string name1, string name2)	void
clear	()	void

Member Variables

- **nodeCount**: tracks the number of nodes that have been inserted in the graph
- **edgeCount**: tracks the number of edges that have been inserted in the graph
- **adjacencyMatrix**: pointer to a 2-D array storing the distances (weight of edges) between nodes
- **degrees**: pointer to an array containing the degrees of each node in the graph
- **cities**: pointer to an array which connects cities to specified node #s (order in which city inserted)

Member Functions

- **i**: inserts a node (city) into the graph
- **setd**: assigns a distance to an edge (road) between two cities
- **s**: searches for a city
- **degree**: returns the degree of the specified city
- **graph_nodes**: returns the number of cities
- **graph_edges**: returns the number of roads
- **d**: prints the distance of the road directly connecting two cities
- **shortest_d**: finds the shortest distance between two cities using Dijkstra's algorithm
- **print_path**: prints the above shortest path between two cities using Dijkstra's algorithm
- **clear**: deletes all nodes and edges from the graph

2. Constructors/Destructor/Operator overloading

For the above `class undirectedGraph`, the constructor first initialized the values of **`nodeCount`** and **`edgeCount`** to 0, since we start with an empty graph. Next, memory is allocated for the three arrays, **`adjacencyMatrix`**, **`degrees`**, and **`cities`**. The initial value of all the entries in **`adjacencyMatrix`** is set to -1. Since distance can never be negative (and we prevent this from being input in **`setd`**), we can use this to verify there is no edge present between the two cities, represented as the array indices. The initial degree of each node is set to 0. The array **`cities`** associates a unique node number (in order of insertion) with a city name, and all initial entries are set to an empty string.

The class destructor was very straightforward, deleting all the array elements then deallocating all their memory.

3. Test Cases

Test 1: Calling operations on non-existent nodes, ensuring program detects and auto-fails

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, setting the shortest distance to involve the greatest number of edges (setting large distances on the paths with fewer edges), then checking **`shortest_d`** and print path output correctly

Test 4: Repeating Test 3 but making the two paths (one with many edges, other with less) equal in distance, then checking **`shortest_d`** and **`print_path`** output correctly

Test 5: Using **`print_path`** on adjacent cities that also have a longer connection involving a greater distance and more edges, ensuring the shorter path is always printed

4. Performance

For this project, the data structure used was an array, not a minimum priority queue. I understand that by doing this I will be unable to get the maximum mark for this project due to performance (15%), per the email sent.

Dijkstra's algorithm time complexity:

1. Fill out **`cost`** matrix:		$O(n^2)$
2. Set **`distances`** to the correct entry in costs:		$O(n)$
3. Update **`mindistance`** by checking with **`distances`**:		$O(n)$
4. Updating **`distances`** by checking unvisited nodes	+	$O(n)$
		$O(n^2)$ (Dijkstra's Algorithm)

Average Case: **$O(n^2)$**

We still need to go through all the steps above as there is no direct connection between nodes.

Best Case: **O(1)**

We have an else if statement which checks if there is a direct distance between the two nodes.

Worst Case: **$O(n^2)$**

We must visit the maximum number of nodes at each city, and the path takes all cities. When checking entries in the distances array, each entry is smaller than the previous, causing the maximum amount of code to be executed.