**Overview:**

This program will let users enter many intersections in (x, y) co-ordinate. User will be able to create links between any two intersection as well. After the user has created a graph, they can search for the shortest path between any two intersection by entering the co-ordinates. User can enter as many co-ordinates they want and similarly, can create as many roads as they want. After this they can search for the shortest path between any two intersection many times. All this can be done until user enters "Quit" command.

**Files Uploaded:**

1. HalifaxMap.java => This file has a class called HalifaxMap which contains all the functions for the commands entered by users.
2. Co_Ordinates.java => This file has a class called Co_Ordinates which stores all the co-ordinates entered by the user.
3. Edges.java => This file has a class called Edges which stores all the links between the co-ordinates entered by the user.
4. mainClass.java => This file has the main class where user will be asked to enter the commands. This file will not be used as Prof. McAllister will be using his main class file.
5. External_Document.pdf => This file contains the details for the program written.

**Data Structure:**

To store the co-ordinates and edges, ArrayList has been used in the program.
Suppose user enters 4 co-ordinates: (1, 1), (2, 2), (3, 3) & (4, 4). It will be stores in an ArrayList **vertices** of type **Co_Ordinates**.

| | X | Y |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 2 | 2 |
| 2 | 3 | 3 |
| 3 | 4 | 4 |

Now suppose user wants to create link between (1, 1) : (2, 2) , (1, 1) : (3, 3) & (3, 3) : (4, 4), it will be stores in an ArrayList **edge** of type **Edges** from both ways. It will also store the distance between them.

| | Intersection_1 | Intersection_2 | Distance |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 2 | 0 | 2 | 3 |
| 3 | 2 | 0 | 3 |
| 4 | 2 | 3 | 1 |
| 4 | 3 | 2 | 1 |

The graph is stores in adjacency matrix, i.e., 2-Dimensional array of type integers.

$$\begin{array}{c|cccc|} & 0 & 1 & 2 & 3 \\ \hline 0 & -1 & 1 & 3 & -1 \\ 1 & 1 & -1 & -1 & -1 \\ 2 & 3 & -1 & -1 & 1 \\ 3 & -1 & -1 & 1 & -1 \\ \hline \end{array}$$

<u>**Required Functions for Commands:**</u>

1. <u>boolean newIntersection (int x, int y):</u>

This function is used to add new intersection of the map. It will take x and y co-ordinate as a parameter. It will create an object of type **Co_Ordinates** and pass these values to the constructor of the class **Co_Ordinates**. After which this object will be added to the ArrayList **vertices**.

The function will return **FALSE** if:

- there is any issue while adding
- the co-ordinates already exist

If the above conditions are not met, then the function will return **TRUE**.

2. <u>boolean defineRoad (int x1, int y1, int x2, int y2):</u>

This function is used to create roads between two intersections. It will take x1 & y1 of 1st intersection and x2 & y2 of 2nd intersection as parameter. It will get the index of the co-ordinate by invoking the method getIndex() .Then it will create an object of type Edges ad pass these values to the constructor of the class **Edges**. It will store the values from (x1, y1) to (x2, y2) and (x2, y2) to (x1, y1). It will also store the distance between them.

The function will return **FALSE** if:

- any one of the two intersection is not found in ArrayList **vertices**.
- road exists between the given co-ordinates
- there is any issue while creating road

If the above conditions are not met, then the function will return **TRUE**.

3. <u>void navigate (int x1, int y1, int x2, int y2):</u>

This function will provide the shortest path between the source and the destination. The function uses Dijkstra's Algorithm to find the same.
First, it will find the index of the source and destination by invoking the method getIndex(). Then it will check few conditions where the function should print "No Link". Below are the conditions.

- if source does not exist in the graph
- if destination does not exist in the graph
- if there are no roads in the graph
- if either the source or destination is connected to the rest of the graph

The function will print only single co-ordinate when the source and destination are same and it exists in the graph.

If the above conditions are not met then function will execute further. It will create a adjacency matrix by invoking the method createMatrix().

After creating the adjacency matrix, the logic for finding the shortest path from source to all the nodes will be executed. The logic will be executed till all the vertices are visited, i.e., we have the all the shortest distance from source to other nodes. (By invoking minimumDistance())
We will even store the previous element so that we can print the path from source till destination. The printing part will be done by invoking the method printPath().

**Additional Functions:**

HalifaxMap

1. int minimumDistance (int distanceList[], boolean visited[]):

This function will return the intersection with the minimum distance from the current intersection.

2. void printPath (int currentIntersection, int previous[]):

This function will recursively print the co-ordinate from source to destination. This method will be first invoked by navigate() method.

3. void createMatrix ():

This function is used to create the adjacency matrix. It will store the distance of the two intersection. The rest of the matrix contains the value -1.

4. boolean isConnected (int intersection):

This function is used to check if the given intersection is connected to the rest of the graph.

5. int distance (int x1, int y1, int x2, int y2):

This function is used to calculate the distance between the two given co-ordinates. It will use the following formula square_root((x2 – x1)$^2$ + (y2 – y1)$^2$) and return the **result**.

6. int getIndex (int x, int y):

This function is used to return the index of the given co-ordinate. It will check this value from the vertices ArrayList. If the value is not found it will return **-1**.

7. void print () (For testing purpose only):

This function is used to print all the vertices and edges in the graph. This program is used for testing purpose to check if the correct roads have been created.

**Co_Ordinates**

1. Co_Ordinates (int x, int y):

This is the constructor for class Co_Ordinates which stores the value of x and y.

**Edges**

1. Edges (int intersection_1, int intersection_2, int weight):

This is the constructor for class Edges which stores the value of index of intersection 1, index of intersection 2 and distance between them.

## Assumptions:

1. The roads are bi-directional
2. The program will execute successfully for negative value of x or y.