External Documentation

- Assignment 3

(Version 1.0)

Table of Contents

1. Overview

This program performs the following operation:

- **Add an Intersection (int x, int y):**
After taking an x and y coordinate of Intersection from the user, it checks if Intersection is new (non-existing) and add it to the graph of "Halifax Map".

- **Define Road (int x1, int y1, int x2, int y2):**
After taking the source and destination coordinate from the user defines a new road between the Intersection.

- **Navigate (int x1, int y1, int x2, int y2):**
After taking the Source (x1, y1) and destination (x2, y2) coordinates from the user, the program uses the Dijkstra's algorithm to find out the shortest path existing between source and destination and print the path, if any path exists.

The solution delivered is based on several important factors as the implementation of data the structure used for storing the Intersection, Road, and Graph on overall.

2. Files and external data

Below is the java files for the Implementation of the required operation:

- **Halifaxmap.java**: Program that has methods to add Intersection "addIntersection", define road between Intersection "defineRoad" and navigate(navigate) through the map to find out the shortest distance between source and destination.

Three java files are implementing the Intersection, Road and Graph object:

- **Intersection.java**: Implements a blueprint of the intersection, having x and y as coordinates attribute and different methods to perform an operation on those attributes.

- **Road.java**: Implement a blueprint of road, having the source and destination of type intersection and distance value between them.

- **Graph.java**: Implementation providing graph structure to store all the Set of Intersection and Roads defined. It is the class which have all the methods and sub-methods contributing to form a Dijkstra's algorithm to find out the shortest possible path between any source and destination, if any such path exists.

3. Data structures and their relationship to each other

The program treats Intersection and Road as two different Objects which are eventually a part of the graph which represents the Halifax map.

The graph object uses below data structure to store and manage the data:

- **HashSet of Intersection**: We are storing all the Intersection into a HashSet, which takes care of any duplicate (already existing) intersection is being added.

- **HashSet of Roads**: We are storing all the Roads into a HashSet, which takes care of any duplicate (already existing) roads are being added.

The above two forms the graph object, remaining data structured used as shown below are used while finding the shortest path.

- **Set of visited Intersection:** We use this while navigating from source to destination to find the path and stores all the visited Intersection into this data structure i.e. Set<Intersection>.

- **Set of Unvisited Intersection:** We use this while navigating from source to destination to find the path and when we find the next possible Intersection that can be visited, we store that Intersection into this data structure i.e. Set<Intersection>.
  The Intersection is at first added to set of unvisited Intersection and later on removed and added to set of visited Intersection.

- **Map of distance traveled <key, value> as < Intersection, distance>**: Used to store the intersection as key with the overall distance of that Intersection from the source as the value to that specific entry.

- **Map of Predecessors <key,value> as < Intersection, Intersection>:** While traversing through the graph using the Dijkstra's algorithm we store the path traveled from one intersection to another in this map, **starting with the Source as value to first entry** and the **Destination as key to last entry.**
  We **iterate through this map to find out reference to each key, value pair starting from destination until we reach the source**, **to find the path in the backward direction**. In the end, **we reverse the path to find the actual path.**

4. Assumptions

- We do not need to worry about roundoff error in the lengths of the paths when determining the shortest path. You can safely round the length of any road to an integer.
- The roads are straight lines between the intersections. A road from (x1, y1) to (x2, y2) has length square_root ((x2 – x1)2 + (y2 – y1)^2).

5. Key algorithms and design elements

1. **Adding the New Intersection**: This Method checks if the coordinates provided for Intersection are new (non-existing) and add it to the graph of "Halifax Map" and return true. Returns false, otherwise.

2. **Defining the new road:** After taking the source and destination coordinate from the user, it checks if the source and destination intersection are existing in the graph. If existing, it checks if the source and destination Intersection provided is the same or not, and if they are not the same, the distance between the intersection is calculated and road with source, destination, and distance between them is created and True is returned. Return false, otherwise.

3. **Navigate:** We are using Dijkstra's algorithm as described below:

   - After taking the Source (x1, y1) and destination (x2, y2) coordinates from the user, we verify if the source and destination are known intersection in the graph.
   - We take the source, place the source in "distance" Map as key and zero as the value, then we insert it into the Unvisited set of the intersection.
   - Going further we iterate the below steps until the unvisited set of Intersection is having no other elements.

     I. We find the "Intersection" available in the Unvisited set of the intersection, having the minimal distance from the Source looking into "distance" map.

     II. We pick the "Intersection" found from the above step and find its corresponding neighbors and place them in the list of neighbors.

     III. We compare the distance of all the neighbors (available in the list) from the source and find the "closest neighbor" with minimal distance.

     IV. Place the neighbor intersection and the distance into the "Distance" map.

     V. Place the neighbor intersection into the set of the unvisited intersection.

     VI. Make an entry into the map "predecessors" as < neighbor intersection, Intersection>.

     VII. Remove the "intersection" from the unvisited set of intersections.

4. Once the above iteration ends Iterate through the "predecessors" map **find out reference to each key, value pair; starting from destination until we reach the source**, **to find the path in the backward direction**.

5. In the end, **we reverse the path to find the actual path.**

6. We print the path obtained, otherwise if path obtained is null, print "No path."

6. Limitations

The current design is to operate on the Bidirectional graph, assuming the road is not a "one-way" road.
Considering the graph as two dimensional i.e. Operates on the intersection considering it in two dimensional plain i.e. x and y coordinate
plain i.e. x and y coordinates.