CSCI 3901 Assignment 3

Name : Yogish Honnadevipura Gopalakrishna(B00928029)

Overview

This program implements a Map. In this map, points containing (X,Y) coordinates are added at first using "**Point"** class objects. Then, streets are formed by connecting two points and given a unique name using the "**addStreet**" method of "**MapPlanner"** class. After map formation, a depot is added onto a particular street and street side using the "**depotLocation**" method of MapPlanner class which is used as a starting location. The variable "**degrees**" which is passed on as a parameter to the MapPlanner constructor defines the turn from one street to another.

This Program has two main features:

- furthestStreet() - Given a depot location, this method returns the furthest street from the street where the depot is located.
- NoLeftTurn() - This method takes a Location object as a parameter which is used as the destination. This method returns the route from start to destination where the route involves no left turn made except when an intersection has only two streets. If there is no route, it returns null if no route exists. This method returns a Route Object through which the route information can be extracted.

Files and external data

There are Eleven main files:

- DjikstraAlgorithm.java -> This program contains Djikstra's algorithm through which the the shortest distance between two nodes and streets are calculated
- Helper.java -> This program contains a method to identify if the streets(containing 2 points) are the same, method to identify at which vertex the route should start from when given a street side of a location.
- MapPlanner.java -> This method is used to build the map by adding streets and also used for navigating through the map through which the furthest street and route with no left turn are calculated.
- Point.java -> This method contains (X,Y) coordinates. Two points make up a street.
- Location.java -> captures a location on the map with street name and street side.

- TurnDirection.java -> It's an enumerated type which tells the turn at an intersection.
- StreetSide.java -> It’s also an enumerated type which is used to identify the side of the street.
- streetObj -> This class captures the data related to the street, namely its ID, its start and end point and the distance between them.
- Route -> This class has the complete route from the start to destination location where no left turns are made. It contains the total distance covered, streets covered in the route.
- SubRoute -> This class is used to identify the route which contains loops
- DataRoute -> This class is used as an additional class to return multiple data regarding the route back to the MapPlanner class.

Data structures and their relations to each other

- Stored the Point objects as a set in an attribute called “points”. A set is apt, as a point can be common to multiple streets.
- Stored the StreetID as keys and their respective points as values in a HashMap in an attribute called “mapOfStreetsToPoints”. This map is used to identify street coordinates while building an adjacency matrix.
- Stored the street objects as an Arraylist in an attribute called “streets”. These objects contain information of the street's starting and ending point, distance between them, and the name of the street.

Choices

- Created a constructor for Route class for taking the data related to the Route.
- When there are multiple streets connected to the furthest vertex, I have considered the longest street as the furthest street.

Key algorithms and design elements

The algorithm and design used for implementing the **FurthestStreet** is as follows:

1. Convert the set “Points” into an arraylist so that it can be traversed to create the adjacency matrix.
2. Traversing the list and populated the matrix if two points make up a street and used the distance between them as the weight of the graph.
3. When given a depot location, use the street side to decide the starting point of the graph
4. Use the Dijkstra's Algorithm to get the distance from source node to every other node
5. Get the farthest node from the array returned from the Dijkstra algorithm.

6. Return the street ID if the farthest vertex is connected to only one street, if the street is connected to multiple streets, return the street ID of the longest street.

The algorithm and design used for implementing the **routeNoLeftTurn** is as follows:

1. Considering the Street as the vertex of the graph
2. Traversing through the point array(streets) to create adjacency matrix
3. If a point is common among two streets, then that point will be considered as an intersection to those streets, and the turn required to travel between streets is calculated.
4. If the turn is left, and only two streets are connected to it, then the turn is populated in the graph, else not considered.
5. If any turn other than left, then the graph will be populated and the weight will be the distance from the start of the first street to the end of the second street.
6. After the graph is populated, then Djikstra's algorithm is used to calculated the distance between source to the destination
7. Djikstra's algorithm is required to return multiple data, therefore a class namely "DataRoute " is created and an object of the class is returned.
8. MapPlanner object uses the returned object's data and creates the Route object and the Route object is returned.

Limitations

- Not able to implement loops and simplify in Route class

References

- https://www.programiz.com/dsa/dijkstra-algorithm (Accessed : 5 March, 2023)
- https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-in-java-using-priorityqueue/ (Accessed : 8 March, 2023)
- https://stackoverflow.com/questions/74169640/dijkstras-algorithm-shortest-path-in-directed-graph-find-the-last-node-taken-to (Accessed : 8 March, 2023)

Test Cases

MapPlanner Class

Constructor(int degrees)

Input validations:
- degrees is positive
- degrees is negative
- degrees is other than “int” type

Boundary cases:
- degrees is 0
- Degrees is 360

Boolean depotLocation(Location depot)

Input validations:
- depot is null
- Verifying the attributes of depot location are not null

Control Flow:
- Street not present in the map

Data Flow
- Calling depot location before adding streets
- Calling depot location after adding streets

Boolean Addstreet(String streetID, Point start, point end)

Input Validations:
- streetID is null
- streetID is empty
- Already existing streetID
- Start or end is null

Boundary Cases:
- streetID is empty
- streetID is not empty

Control Flow:

- Point start and end present in the map

Data Flow:

- Calling after depot location is set
- Calling before depot location is set
- Calling before further street and no left turn
- Calling after further street and no left turn

String FurthestStreet()

Data Flow:

- Calling after depot location is set
- Calling before depot location is set
- Calling before adding streets
- Calling after adding streets

Route routeNoLeftTurn(Location Destination)

Input validations(same as depot Location):

- depot is null
- Verifying the attributes of depot location are not null

Control Flow(same as depot Location):

- Street not present in the map

Data Flow(same as furthest street):

- Calling after depot location is set
- Calling before depot location is set
- Calling before adding streets
- Calling after adding streets
-

POINT CLASS

point(x,y)

Input Validations:

- x and y are positive
- x and y are negative
- x and y are non-integers

Boundary Case:

- x and y are 0
- X and y are max values possible