**CSCI 3901 Assignment 3**

**External Documentation File**

**Abhay Lohani(B00989230)**

1. **Introduction**

This code emphasizes building the MapPlanner class. Making the MapPlanner class a part of robust code for route planning. As the shortest way connecting street is not always the fastest way possible which is also depicted in this code as its ability to determine the best routes that avoid left turns is one of its most notable capabilities. Apart from facilitating the proper management and querying of municipal streets, this class also helps users determine the maximum distance they can drive from a designated depot. This study applies modifications to Dijkstra's algorithm and uses ideas from graph theory to address the particular difficulties of urban routing.

1. **Key Methods and Usability**

* **MapPlanner(int degrees)**Route planning and identifying whether a path segment indicates a turn or a straight journey are the responsibilities of the MapPlanner object.   
    
  Degrees Parameter: This is the threshold angle specified through this parameter. If the angle is less than equal to the threshold then continue movement in a straight line, and if the angle exceeds the threshold then there is a need for a turn.  
    
  TurnType Method: Based on the degrees argument, this method, which is part of the Point class, computes the angle between two points and determines if the movement is straight or curved.
* **depotLocation(Location depot)**This method sets the starting location of your journey, called the depot.
* **addStreet(String streetID, Point start, Point end)**Expands the map by adding a new street. The streetId is the unique identification for the street. This method establishes the street's beginning and finishing locations. Views the beginning and finishing locations as junctions. Represents the points using integer coordinates (x, y) in meters. Specifies the "side" of the roadway in relation to the path that leads from one place to another, taking into consideration the street's two-way nature.  
    
  The three parameters that it takes in are:  
    
  streetId: A unique identifier for the street.   
  start: The starting point of the street.   
  end: The ending point of the street.
* **furthestStreet()**Determines which street is the most distant from the depot and also determines the distance by taking the shortest route through the streets that are currently in place. takes intersection turns into account. The farthest street assumes no ties.It has a return value as a string that returns the streetId of the far most street.
* **routeNoLeftTurn(Location destination)**

The MapPlanner class's routeNoLeftTurn() method is in charge of figuring out the quickest path from the depot to a destination without taking any left turns. In order to ensure that the route complies with a restriction—no left turns are permitted, with the exception of U-turns at dead ends—this method assesses the direction of travel between connected streets.

1. **Data Structures**

**There were different types of data structures used in this code as under:  
  
a)HashMap:   
  
Type:** Map<String, Point[]>

**Usage:**The variable streetMap is a HashMap that stores street information, where each key (String) represents a street ID, and the value is an array of Point objects (Point[]) representing the start and end points of the street.

**Purpose:**A HashMap is used to quickly look up streets by their IDs and retrieve associated points.

**b)Array:  
  
Type:** Point[]

**Usage:** The values in the streetMap are arrays of Point objects. These arrays store the start and end coordinates of a street.

**Purpose:** Arrays are simple structures used here to hold exactly two points (start and end) for each street.

|  |  |  |
| --- | --- | --- |
| **Class/Method/Field** | **Description** | **Assumptions/External Use** |
| Helper | A utility class that contains static methods for calculating distances, midpoints, turn types, and handling dead ends. | Must be defined separately with methods: distanceBetween, midpoint, calculatePathDistance, turnType, and isDeadEnd. |
| Helper.distanceBetween | Calculates the distance between two points (Point objects). | Requires two Point objects and returns a double representing the distance. |
| Helper.midpoint | Returns the midpoint between two points (Point objects). | Requires two Point objects and returns a Point. |
| Helper.calculatePathDistance | Calculates the path distance between two points, considering the street map and other route details. | Requires the map of streets, start point, and end point. |
| Helper.turnType | Determines the turn type (e.g., left, right, U-turn) between two points based on a threshold angle (turnThresholdDegrees). | Needs two Point objects and an angle threshold. Returns an enum TurnDirection. |
| Helper.isDeadEnd | Checks whether a street is a dead end. | Takes a street ID and map; returns true if the street is a dead end. |
| Location | Represents ageographical location with at least a String getStreetId() method to return the street associated with the location. | Must be implementedseparately with a getStreetId() method to return a String. |
| Point | Represents a 2D point (x, y coordinates) on a map. | Must be defined with x and y properties or getter methods. |
| Route | Represents a route with the ability to add legs (street segments) and keep track of directions and streets | Must be implemented separately with methods like addLeg and internal structure to track streets and directions. |
| Route.addLeg | Adds a new leg to the route, containing a street ID and the direction type (turn). | Needs two parameters: the street ID (String) and turn type (TurnDirection). |
| TurnDirection | Enum-like class or object defining turn types (e.g., left, right, U-turn, etc). | Should contain at least UTurn, LeftTurn, and RightTurn values. |

**Table to depict the logical flow**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Step** | **Conditions/Decision** | **Outcome** |
| **depotLocation()** | Set depot location | Does streetMap contain the depot.getStreetId()? | If yes, depot is set; if no, depot is not set. |
|  |  | Is depotLocation now non-null? | If yes, depotLocation is ready for routing logic. |
| **addStreet()** | Add a street with its start and end points | Does streetMap already contain the streetId? | If no, add the street; if yes, do nothing. |
|  |  | Are the start and end points valid? | If yes, store the street points in streetMap. |
| **furthestStreet()** | Find the street farthest from the depot | Is depotLocation null? | If yes, return null; if no, calculate distance. |
|  | Calculate distance between depot and each street midpoint | Is the calculated distance greater than the current max distance? | If yes, update max distance and record the street. |
|  |  | Did a valid street with the farthest distance emerge? | Return streetId of farthest street. |
| **routeNoLeftTurn()** | Plan a route to the destination without left turns | Is depotLocation or destination invalid? | If yes, return null (route fails). |
|  |  | Is there a valid next street leading towards the destination? | If yes, proceed with the route; if no, terminate route. |
|  | Analyze turn direction between the streets | Is the turn within the allowed degree threshold? | If no, discard route leg and recalculate. |
|  |  | |  | | --- | |  |  |  | | --- | | Is the turn a U-turn but not at a dead-end? | | If yes, return null (route fails). |
| **findNextStreet()** | Find the next street to reach the destination. | Is there a direct connection between currentStreet and destinationStreet or not. | If yes, return destinationStreet. |

**Testing**

**a)Basic Functionality Test**

a.1)Depot location on a valid street

a.2)Depot location on an invalid street

a.3)Add a valid street with non-overlapping coordinates

a.4)Add an existing street (should return false)

a.5)Add a street that overlaps with another street

**b)Tests of Distance Calculation and Route Planning**

b.1)Furthest street in a simple straight-line layout

b.2)Furthest street in a branched layout

b.3)Furthest street when the map has only one street

b.4)Furthest street in a grid-like layout

b.5)Route with no left turns possible

b.6)Route requiring left turn exception (dead end)

b.7)Route with no valid path due to left-turn restriction

b.8)Route with U-turn at a dead end

**c)Error Handling Tests**

c.1)Depot location not set but method furthestStreet() is called

c.2)Add street with invalid points

c.3)Route planning with no depot set

c.4)Route planning to an invalid destination street

**d)Boundary Test Cases**

d.1)Add streets at the extreme ends of the coordinate plane

d.2)Route planning with extremely large coordinates

d.3)Turn angle exactly equal to threshold degrees

**e)Control Flow Test**

e.1)Add multiple streets with branching intersections

e.2)Route forming a loop

e.3)Attempt to use a depot on the street that hasn’t been  
added to map

**f)Data Flow Test**

f.1)Ensure all streets and depot are initialized correctly before calling route planning methods.

f.2)Test passing different degree values to ensure proper angle-based turn calculations

**g)Stress and Performance testing**

g.1)Add 500+ streets with varying coordinates

g.2)Complex street planning with large street