CSCI 3901 Assignment 3

Due date: Wednesday, March 8, 2023 at 11:59pm Halifax time. Submissions through your CS gitlab repository in https://git.cs.dal.ca/courses/2023-winter/csci-3901/assignment3/????.git where ???? is your CS id.

Problem 1

Goal

Work with inter-related information.

Background

Planning travel routes in a city is a well-known problem for which algorithms like Dijkstra's algorithm[1] provide efficient implementations for shortest routes. However, the shortest route by distance is not always the fastest route by time.

UPS, for example, has software to plan routes in a city that avoids making left turns across oncoming traffic. They found that waiting for a left turn traffic light or for a break in oncoming traffic to allow a left turn could take more time than just making 3 right turns around a block.

You will develop a Java class that will solve simplifications of routing and facility location problems in a city map.

Problem

Implement a class called MapPlanner that will accept information about the streets and locations in a city and will then answer some questions on route planning and facility locations.

Your MapPlanner class will include the following methods, at a minimum:

- Constructor (int degrees)
- Boolean depotLocation(Location depot)
- Boolean addStreet(String streetID, Point start, Point end)
- String furthestStreet()
- Route routeNoLeftTurn(Location destination)

These methods use the following support classes and for which you will be given base classes on which you can extend or build upon:

Point	captures a 2-d (x, y) point with integer coordinates

[1] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm, retrieved February 17, 2023

Location	captures a place on the map, specifically a street id and whether you are on the left or right side of the street
Route	captures a travel route, namely which street id to go to next and the type of transition onto that street (left turn, right turn, straight ahead, U-turn). Methods include: - int legs() – number of street legs in the route. A leg is a trek between two adjacent intersections - Double length() – the length of the route - List<SubRoute> loops() – the list of loops in the route, in order that they appear along the path. A loop is a part of the route that crosses the same intersection twice. - Route simplify() – return a copy of the current route that only contains the turns for the route.
SubRoute	captures a contiguous subset of streets and turns within a larger travel route. Methods include: - Route extractRoute() – convert this part of a route into a route of its own
TurnDirection	An enumerated type of turns at an intersection or the end of a street, namely Right, Left, Straight, and UTurn.
StreetSide	An enumerated type that identifies the side of the street, namely Left or Right. The “left” and “right” sides come from going along the street in the direction from start to end as parameters to addStreet.

For simplicity, any time we mention a Location (street and side of street), we take the location to be at the centre of that street.

Your implementation of the methods can add exception throwing to the method signatures. You are responsible for making the code robust, including any methods or stubs provided to you as a start to the assignment.

MapPlanner(int degrees) constructor

Create the MapPlanner object. The “degrees” parameter identifies whether you are deemed to be going straight or turning at an intersection. If your deviation from the straight forward is <= “degrees” then we would say that you are going straight. Otherwise, you are turning (Figure 1). The template code contains the method “turnType” in the Point class that uses these degrees in its calculation.

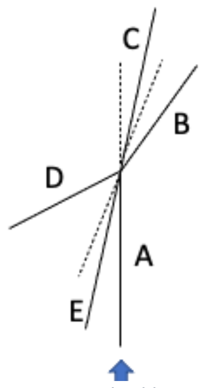

Figure 1 Dashed lines represent the minimum degrees for a turn from A at the intersection. From the bottom of A and going to the intersectino, B is a right turn, C is straight (well, straight enough), D is a left turn, and E is a U-turn.

Boolean depotLocation(Location depot)

Identify that your starting depot is the given location on the city map.

Boolean addStreet(String streetId, Point start, Point end)

Identify a street for your map, starting at the start point and ending at the end point. Both the start and end points should be considered as intersections. Point coordinates will be integer (x,y) points where the coordinate system measures distances in metres.

When the code refers to the side of a street, that side (left or right) is relative to this start -> end direction on the street. However, the street is a two-way street.

String furthestStreet()

Give the street id of the street that is reachable from the depot and whose centre is furthest from the depot. For the distance calculation, the path to the furthest street must follow the streets on the map and can make any turn at intersections. You can assume that there will not be a tie for the furthest street.

Route routeNoLeftTurn(Location destination)

Starting at the depot, report the shortest route to the given destination using the streets in the map. The route cannot make left turns at intersections. The only exception to the "no left turn" rule is if there are only two streets meeting at the intersection. It is possible that some destinations cannot be reached with this "no left turn" rule.

Each street segment that you follow in the route is called a leg of the route. The first leg of the route will be leaving the depot and going onto a street, so will be the street name of the depot and a left turn (since the depot is in the middle of the street and is not at an intersection).

"Shortest" here comes from the length of the path. The typical process to compute such a path is Dijkstra's algorithm. In this case, the "no left turn" constraint means that you will need some adjustment. It's possible to either change Dijkstra's algorithm or the data that you are giving to the algorithm. In my view, the easier of the solutions is to change how you are thinking about the map rather than changing the algorithm.

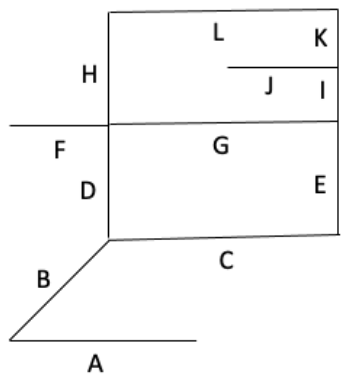

Figure 2 Example map of streets

Consider the map in Figure 2 as an example. Suppose that the depot is on street G on the side where the G appears. If the destination street is D and the side is the street opposite to the letter D (inside the box) then the route would be

- Turn right on G
- Turn right on E
- Turn right on C
- Turn right on D

If the destination is street J precisely where the J lies then the route would be

- Turn right on G
- Turn right on E
- Turn right on C
- Turn right on D
- Straight onto H
- Turn right on L
- Turn right on J
- U-turn on J

If the destination is anywhere on edges A, B, or F then you can't reach the destination with only right turns from the current depot.

Assumptions

You may assume that

- All streets given to addStreet are two-way streets.
- A street id names the part of the street between two intersections uniquely.
- A location defined on a street is at the centre of the street.
- All streets with a given (x, y) endpoint will meet at that point and form an intersection.
- A street that ends on its own would allow a drive to turn around at the end of the street (not the middle) as a UTurn.
- Streets will only cross one another at an intersection.
- If two streets meet at a T intersection then it will be presented as 3 streets meeting at one intersection and not as one long street with another one intersecting in the middle of the long street.
- The distance to the furthest street is unique. You won't be given a map with two furthest streets at exactly the same distance from the depot.

Constraints

- You cannot do a UTurn at an intersection that isn't a street dead end (end of a street with no connecting street).
- You may use any data structures from the Java Collection Framework.
- If you use a graph then you may not use a library package to for your graph or for the algorithm on your graph.
- If in doubt for testing, I will be running your program on timberlea.cs.dal.ca. Correct operation of your program shouldn't rely on any packages that aren't available on that system.

Notes

- Pick the order in which you will implement your methods. Implement the simpler ones first to get familiar with your data representation. For example, you might choose to implement functionality to find the shortest path and that allows left turns before then adjusting something to remove the left turns.

Marking scheme

- Documentation (internal and external), program organization, clarity, modularity, style – 4 marks
- Thorough (and non-overlapping) list of test cases for the problem – 3 marks
- Robustness of the code – 2 marks
- Marks for each of the requested methods:
    - addStreet + depotLocation – 2 marks
    - furthestStreet – 4 marks
    - routeNoLeftTurn – 6 marks
    - loops+ simplify – 3 marks
    - supporting methods (getters, length, legs, appendTurn, extractRoute, …) – 3 marks