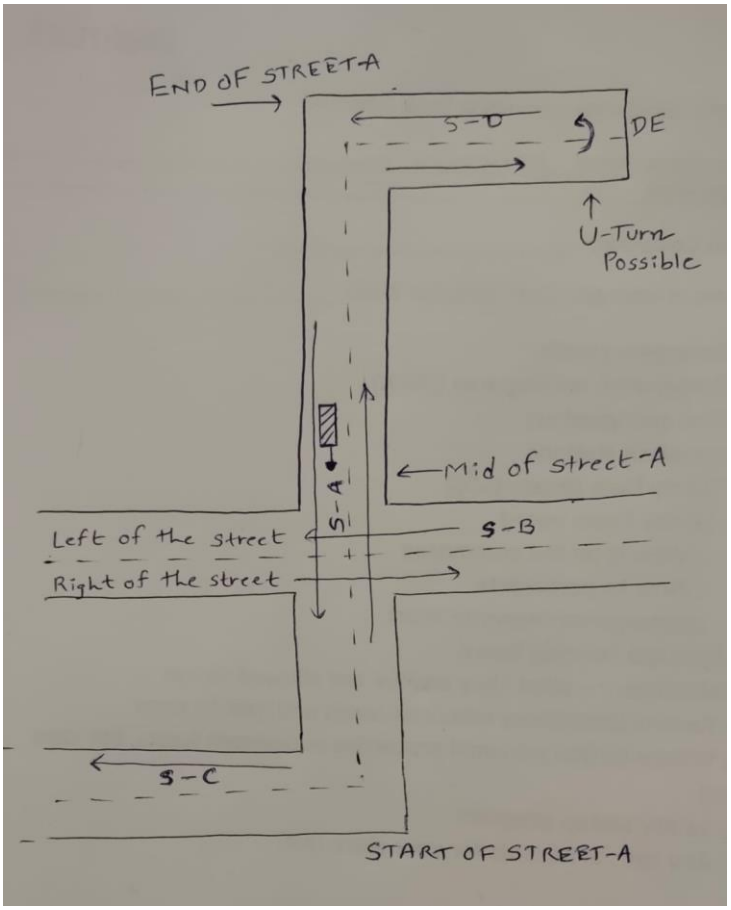
CSCI – 3901: Assignment 3

Overview

Implement a java program that can work with interrelated classes and gives the shortest route with a certain consideration. This prototype uses the nodes to connect each element with each other and is used to store the shortest route for traveling.

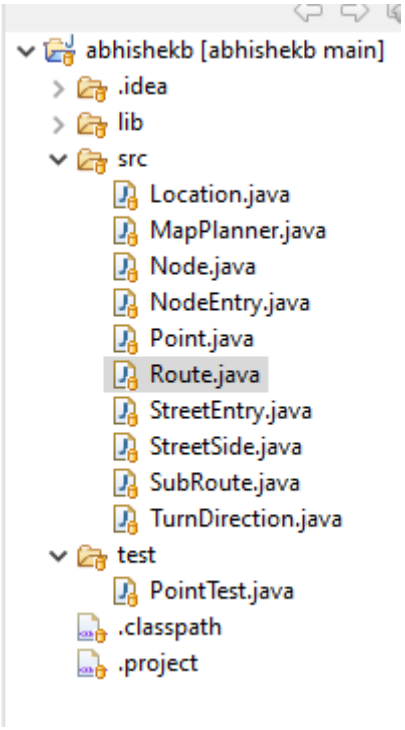
Requirements:

1. User would be able to select the location on the start point and end point.
    - Street side (right / left)
    - User considered to be in the middle of any street.
2. Users can add streets and paths.
    - This information needs a street name and street id which stores street data with its start and end point.
3. Map can't take any "U" turn between the street.
    - Only possible "U" turn is at the end of the street.
4. If, the path is not reachable without a left turn, avoid that path.
    - Just print can't be reached if it is not reachable without taking a left turn.



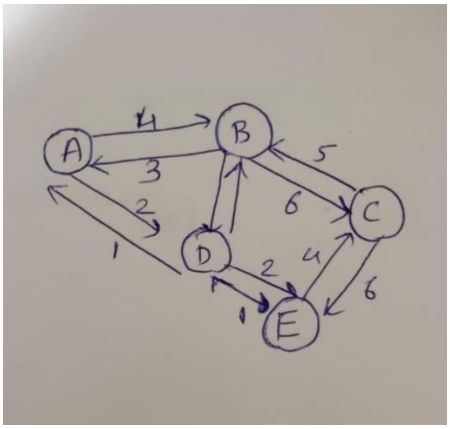
Files and Data

In this code, we have MapPlanner class which contains the main methods to implement an entire idea. With that, we have support classes that contain the methods to support the main class. In this support class, Point and Location classes are oriented to the map locations and Streets and points on the street where as Route and SubRoute classes are mainly focused on traveling on the map. Which helps to implement the main logic of the travel around the street on the map. TurnDirection and StreetSide java classes are Enum type which supports Route and Point classes parallelly. We have Node, NodeEntry, and StreetEntry classes to support the Point and Location classes.



Data structure and relation

I have used nodes to map the route for traveling. Where vertices contain the time taken by the user to travel.



Assumption

- All streets given to add streets are two-way streets.
- A location defined on a street is at the center of the street.
- The distance to the furthest street is unique. You won't be given a map with the two furthest streets at exactly the same distance from the depot.
- A street id names the part of the street between two intersections uniquely.
- 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 driver to turn around at the end of the street.
- Streets will only cross one another at an intersection.

Key algorithm and design elements.

For the above implementation, the key algorithm is the Dijkstra algorithm with the special feature of restricting travel to the left.

```
function routeNoLeftTurn(destination, graph):
    distance = {}
    previous = {}
    queue = []
    visited = set()
    start_node = graph.get("depot")
    distance[start_node] = 0
    add start_node to the queue with priority 0

    while the queue is not empty:
        current_entry = remove the node with the highest priority from the queue
        current_node = current_entry.node
        if current_node is in visited:
            continue
        add current_node to visited
        if current_node.label == destination.getStreetId():
            break ;
        for each street in current_node.getStreets():
            neighbor_node = graph.get(street.to.label)
            if neighbor_node is None:
                continue
            neighbor_distance = distance.get(neighbor_node, infinity)
            new_distance = distance[current_node] + street.distance
            if new_distance < neighbor_distance and not isLeftTurn(street, current_node,
previous.get(current_node)):
                distance[neighbor_node] = new_distance
                previous[neighbor_node] = current_node
```

```
                add neighbor_node to the queue with priority new_distance

    // build the route
    route = an empty route
    current_node = graph.get(destination.getStreetId())
    loop_node = current_node
    while loop_node is not None and loop_node.label != "depot":
        prev_node = previous.get(loop_node)
        street = None
        for s in prev_node.getStreets():
            if s.to.label == current_node.label:
                street = s
                break
        if street is not None and not isLeftTurn(street, prev_node, loop_node):
            turn_direction = getTurnDirection(street, prev_node, loop_node)
            append turn_direction and street.streetId to the route
        loop_node = prev_node
    append TurnDirection.Left and the street ID of the first leg to the route
    reverse the order of loops and streets in the route
    return the route

function isLeftTurn(street, prev_node, current_node):
    if current_node is None:
        return false
    prev_street = None
    for s in current_node.getStreets():
        if s.to.label == prev_node.label:
            prev_street = s
            break
    if prev_street is None:
```

```
        return false
    return prev_street.angle - street.angle > 90
```

Explanation:

The routeNoLeftTurn algorithm is a function that takes in a destination and a graph of nodes and streets. Its purpose is to find the shortest route from the starting point (the "depot" node) to the destination while avoiding left turns.

To start, the algorithm creates several variables, including distance, previous, queue, and visited. distance and previous will keep track of the distance and previous node for each node in the graph, while queue will hold nodes that the algorithm still needs to process, sorted by priority (the lowest priority nodes have the highest distance). visited is a set that will keep track of nodes that the algorithm has already visited.

The algorithm then sets the distance of the starting node (the "depot" node) to 0 and adds it to the queue with a priority of 0.

The main loop of the algorithm runs while the queue is not empty. In each iteration, the algorithm removes the node with the highest priority from the queue, marks it as visited, and checks if it is the destination node. If it is, the algorithm breaks out of the loop.

Next, the algorithm looks at each street that connects to the current node and checks if it is a valid next step. A valid next step is one where the distance to the neighbor node is less than the current distance and there is no left turn required to take that street. If the neighbor node is not already in the distance dictionary (meaning it hasn't been visited yet), the algorithm adds it to the queue with a priority equal to the new distance.

Once the algorithm has found the shortest path to the destination node, it uses the previous dictionary to construct the final route. Starting with the destination node, the algorithm follows the previous nodes back to the starting node, adding each street to the route as it goes. The algorithm checks each turn to make sure it is not a left turn, and if it is not, it adds the turn direction and the street ID to the route. The final step is to add a left turn and the street ID of the first leg to the beginning of the route and then reverse the order of the route.

The isLeftTurn function is used to determine if a turn is a left turn. It takes in the current street, the previous node, and the current node, and returns true if the turn is a left turn and false otherwise. To determine if a turn is a left turn, the function checks the angles of the previous street and the current street. If the difference between the angles is greater than 90 degrees, the turn is a left turn.

Overall, the routeNoLeftTurn algorithm is an efficient way to find the shortest route from one point to another while avoiding left turns.

Limitations

- The addStreet() method in the MapPlanner class doesn't return any meaningful error message if the street cannot be added. It just returns false.
- The furthestStreet() method in the MapPlanner class uses a PriorityQueue to implement Dijkstra's algorithm, which doesn't work properly if there are negative edges in the graph. This method doesn't check for negative edges.
- The furthestStreet() method assumes that the starting location is on a street that exists on the map, but it doesn't check if that is the case.