COMP261 Assignment 2 - 300492683

The Auckland Road System program loads two different city roadmaps which contain nodes as intersections and segments as roads. After the map has been loaded, the user is able to click on a node, this will be the start node, another click on a different node will be the end node, then the search will begin and it will find the shortest distance between the two nodes.

The program has attempted the minimum and the core stages, and touches on the completion stages. The search for the shortest path is done by using an A* search to only add a node to the search, only if it's on the correct direction and does not contain a restriction. The function uses fringe objects, a cost and heuristic, and a priority queue to move through the search. The fringe is sorted based on the lowest heuristic and cost. Once the search is complete, another method is called to create the path. This is created with the previous node's properties and is added to a stack. Then another method is called to loop over the stack and prints all the road's names and distances, highlighted the segments where the order of the segments are correctly in order. The segments are highlighted rather than the roads due to the reason that travelling the entire road can be unnecessary, and segments gives the correct path of the shortest distance as we can come off the road to another road much quicker.

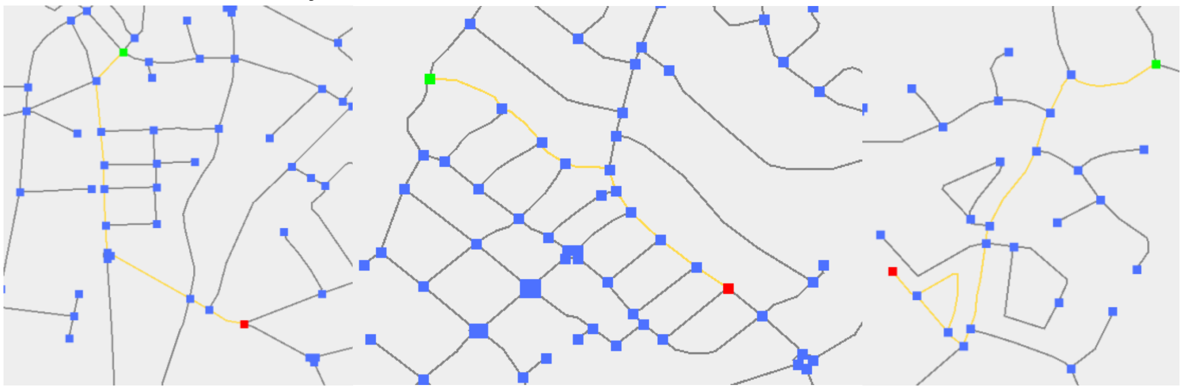
```
START PSEUDOCODE MAIN SEARCH
CLASS FringeObject
        FIELDS will contain basic getters and setters:
        currNode, prevNode, costFromStart, estCost
END FringeObject


FUNCTION findShortestPath
        PARAMETERS startNode, endNode
        currNode, prevNode, costFromStart, estCost represents the FringeObject
        INIT PriorityQueue fringe sorted by lowest estCost property
FOR each node in graph
        SET unvisit node
        SET prevNode to null
OFFER new FringeObject to fringe with startNode, null, 0, distance endNode
        WHILE fringe is not empty
        TAKE FringeObject FROM top of fringe (sets to current node)
                IF currNode is unvisited
                SET currNode to visited
                SET currNode.prev to prevNode
                IF  currNode is endNode
                BREAK
        FOR EACH currNode : segment neighbour
                GET the neighbourNode
                IF neighbourNode is unvisited
                costFromStart = costFromStart + (distance from currNode to
neighbourNode )
                estCost = costFromStart + neighbourNode's length
       OFFER new FringeObject to fringe: neighbour, currNode, costFromStart, estCost
        CALL pathFromPrev passing it the currNode and endNode
        CALL printShortestPath passing the return of pathFromPrev
END findShortestPath
```

```
FUNCTION printShortestPath
        PARAMETERS Stack of segments
        IF Stack is empty
                PRINT "Path Not Found."
        ELSE
                FOR EACH segment in Stack
                TAKE segment off Stack, print the road name and segment length
                ADD segment length to pathDist
                DRAW highlighted segment
        PRINT pathDist
END printShortestPath
```

```
FUNCTION pathFromPrev
        PARAMETERS endNode, finalNode
        INIT pathSeg - empty Stack of segments
        SET currNode to finalNode
        IF currNode not equal to endNode
                RETURN Stack
        WHILE prevNode is not null
        FOR EACH segment in currNode
                IF start or end segment is prevNode
                PUSH segment to Stack
        SET currNode to prevNode
        RETURN Stack
END pathFromPrev
END PSEUDOCODE
```

Describe your path cost and heuristic estimate:
The path cost is the total cost of the segment length between two nodes. From the total cost, it's used to calculate the estCost (the overall estimated cost which is the cost and estimated distance to the end node), by adding the heuristic estimate to the total cost, which is used to sort the fringe. The heuristic estimate uses the distance function in the Location class to calculate distance from the current node to the end node.

Outline how you tested that your program worked:
To test the program by just viewing it, I clicked on different intersections on the map and for each path highlighted it seemed that the program correctly found the shortest path, a few screenshots below. I also checked the A* search requirements and made sure that I have correctly implemented it. The heuristic function is also correct and consistent as the distance function already gets the shortest distance, and the overall cost never decreases. This is because even if the heuristic gets lower as we get closer to the end node, the overall cost will never decrease due to the total cost being added, therefore we will always have the lowest cost.



1. Show how to use A* search algorithm to search for the shortest path from node D to node H. You should show (1) at each step, the elements in the fringe and the element to be visited next, and (2) the final shortest path as a sequence of nodes.

(1):
Step 0: Fringe elements: {<D, null, 0, 25>}
Element to visit next: <D, null, 0, 25>
Step 1: Fringe elements: {<E, D, 10, 26>, <F, D, 8, 27>, <C, D, 14, 51>, <A, D, 15, 53>}
Element to visit next: <E, D, 10, 26>
Step 2: Fringe elements: {<F, D, 8, 27>, <H, E, 31, 31>, <C, D, 14, 51><A, D, 15, 53>}
Element to visit next: <F, D, 8, 27>
Step 3: Fringe elements: {<G, F, 18, 19>, <H, E, 31, 31>, <I, F, 23, 39>, <C, D, 14, 51>, <A, D, 15, 53>}
Element to visit next: <G, F, 18, 19>
Step 4: Fringe elements: {<H, G, 28, 28>, <I, F, 23, 39>, <C, D, 14, 51>, <A, D, 15, 53>}
Element to visit next: <H, G, 28, 28>

(2): Path is D, F, G, H.

2. Show how to use 1-to-1 Dijkstra's algorithm to search for the shortest path from node D to node H. You should show (1) at each step, the elements in the fringe and the element to be visited next, and (2) the final shortest path as a sequence of nodes.

(1):
Step 0: Fringe elements: {<D, null, 0>}
Element to visit next: <D, null, 0>
Step 1: Fringe elements: {<F, D, 8>, <E, D, 10>, <C, D, 14>, <A, D, 15>}
Element to visit next: <F, D, 8>
Step 2: Fringe elements: {<E, D, 10>, <C, D, 14>, <A, D, 15>, <G, F, 18>, <I, F, 23>}
Element to visit next: <E, D, 10>
Step 3: Fringe elements: {<C, D, 14>, <A, D, 15>, <G, F, 18>, <I, F, 23>, <H, E, 31>}
Element to visit next: <C, D, 14>
Step 4: Fringe elements: {<A, D, 15>, <G, F, 18>, <B, C, 22>, <I, F, 23>, <H, E, 31>}
Element to visit next: <A, D, 15>
Step 5: Fringe elements: {<G, F, 18>, <B, C, 22>, <B, A, 22> <I, F, 23>, <H, E, 31>}
Element to visit next: <G, F, 18>
Step 6: Fringe elements: {<B, C, 22>, <B, A, 22> <I, F, 23>, <H, G, 28>, <H, E, 31>, <I, G, 32>}
Element to visit next: <B, C, 22>
Step 7: Fringe elements: {<I, F, 23>, <H, G, 28>, <H, E, 31>, <I, G, 32>}
Element to visit next: <I, F, 23>
Step 8: Fringe elements: { <H, G, 28>, <H, E, 31>, <I, G, 32>}
Element to visit next: <H, G, 28> breaks out of loop
(2): Path is D, F, G, H.

3. A* search takes fewer steps than 1-to-1 Dijkstra's algorithm. Briefly describe the reason.

A* search takes fewer steps than Dijkstra's algorithm because it uses a heuristic function to calculate the estimated distance. The distance is calculated from a node to the goal node, where the value is added to the cost to calculate the F value used to sort the fringe. This means we always visit the node that is estimated to be the closest. This makes sure that we do not move in the wrong direction, even if the distance from the start is shorter, it prevents unnecessary steps for example expanding A and C in the questions above.