COMP261 Assignment 2 Report

Xiaobin Zhuang ID: 300519184

What my code does:

Minimum part:

I have implement the basic A* search function by following the pseudocode given by the lecturer. This has been implemented in three method in the Mapper class called **onAStar, findShortestPath** and **trackBack.**

The **findShortestPath** method will be called when select the startNode and targetNode, and then find the shortest path between them.

The **trackBack** will be called when get the shortest path from the start to the target node by using the “prev” nodes to backtrack from the target node to the start.

In the **onClick** method, i have added code that specifies the start and end of a route. The first node clicked is highlighted green which indicates the **startNode** and the next node clicked is highlighted red which indicates the **targetNode**.

After clicking two nodes, **findShortestPath** method in the **onAStar** will be called when click A* button. Users can find the shortest path between **startNode** and **targetNode**.

I have created a **Fringe** class that helps represent the "path so far" in the A* Search. The Fringe class implements **Comparable <Fringe>,** which refer to [1]. It has a Fringe object with four fields (currentNode, previouseNode, g_value, f_value).

In the **Node** class, I have added a **getSegment** method to get the segment between this node and the other node.

In the **Graph** class, I have added a **highlightedTargetNode** and **highlightedSegments** to highlight the clicked nodes and shortest path.

Pseudocode of A* search:

```
Protected void onAStar() {
        if (startNode or targetNode = null) {
                Does’t run A* search;
        } else {
                findShortestpath;
```

highlight the path, print out the road name in the path and the distance;

}

Initialise startNode and targetNode;

}

For **findShortestPath** method:

Public void findShortestPath(Node startNode, Node targetNode) {

creat a **Set** of **visited** value;

The first Fringe object is created with (startNode, null, 0, f(startNode)) and is added to the fringes queue;

while (fringes is not empty){

Expand (currentNode, prevNode, g_value, f_value) from the Fringe object where f_value is the smallest among all elements in the fringes;

if (visited doesn't contain currentNode){

add currentNode into visited and currentNode.previous = prevNode;

if (currentNode is equal to the targetNode){

trackBack from this Fringe object and break;

}

For (each unvisited neighbour of the current node) {

if (visited doesn't contain neigh){

g_neigh = g_current + Segment(neigh, currentNode).length;

f_neigh = g_neigh + neigh.distance(targetNode);

add new Fringe (neigh, currentNode, g_neigh, f_neigh) into fringes;

}

}

}

}

My g_value and h_value:

My h_value is the distance from a node to whatever the specified goal node as recommended in the assignment.

My g_value is the sum of segment lengths up to the current point in the path.

Core part:

I have implement the Articulation points search function by following the pseudocode given by the lecturer.

My articulation point method uses the **iterative version** of the algorithm. It uses 2 method **onAPs()** and **iterAPs()** and a class **IterAPsObject** which takes a node, integer depth and node root to help with determining if a node is an articulation point.

Nodes are highlighted yellow if a node is an articulation point.

Pseudocode of Articulation points:

```
Protected void onAPs() {
        Initialise depth(node) = ∞, APs = {};
        Randomly select a node as the root node (I select Node 12420), set depth(root) =
        0, numSubTrees = 0;

        for (each neighbour of root) {
                if (depth(neighbour) = ∞) {
                iterAPs(neighbour, 1, root);
                numSubTrees ++;
                }
                if (numSubTrees > 1) then add root into APs;
        }
}

Public void iterAPs(Node firstNode, int depth, Node root) {
        Initialise stack as a single element;
        while (stack is not empty) {
                peek the last element of the stack and expend its field;
                //Case 1:
                if (depth(n*) = ∞) {
                        depth(n*) = depth*;
                        reachBack(n*) = depth*;
                        children(n*) = all the neighbours of n* except parent*;
                }
                // Case 2:
                else if (children(n*) is not empty) {
```

```
                get a child from children(n*) and remove it from children(n*);
                 if (depth(child) < ∞) {
                        reachBack(n*) = min(depth(child);
                        reachBack(n*));
                        else {
                                push into stack;
                        }
                }
        }
        // Case 3:
        else {
                if (n* is not firstNode) {
                        reachBack(parent*) = min(reachBack(n*);
                        reachBack(parent*));
                        if (reachBack(n*) >= depth(parent*) {
                                add parent* into APs;
                        }
                }
                remove <currentNode, currentDepth, parentNode>from stack;
        }
}
```

I have found 240 articulation points in the small graph, which is correct. But I found only 10032 articulation point in the large graph, which is different with expected value.

Completion part:

I have completed the completion part.
Avoid the wrong way down a one-way street.
Highlighted the route on the map by yellow colour.
Print out the road name with out duplicates, the distance of each road and the total distance.

Challenge part:

Explain and justify the designed heuristic (why they are admissible and consistent):
Admissible: because it can stop the algorithm after visiting the goal node. All estimates cost smaller than the true cost**.**
Consistent: no need to revisit the same node. f=g+h is monotonically non-descreasing.

What my code doesn't

Challenge part:

I haven't done the rest of the challenge part.

Reference:

[1] https://www.callicoder.com/java-priority-queue/#:~:text=A%20priority%20queue%20in%20Java,at%20the%20time%20of%20creation.&text=So%20when%20you%20remove%20an,specified%20ordering%20is%20removed%20first.