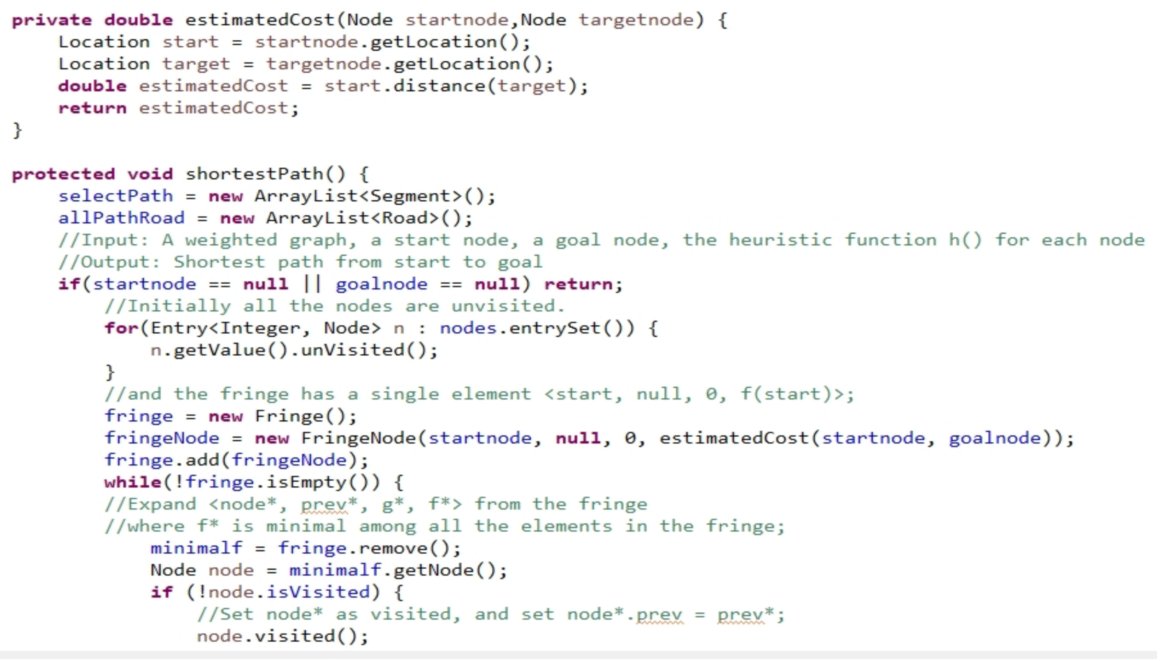
A* search algorithm:

```
Initially all the nodes are unvisited, and the fringe has a single element <start, null, 0, f(start)>;
While (the fringe is not empty) {
Expand <node*,prev*,g*,f*> from the fringe, where f* is minimal among all the elements in the fringe;
if (node* is unvisited) {
    Set node* as visited, and set node*.prev = prev*;
        if (node* is the target node){
            Obtain the shortest path based on the .prev fields;
            break;
        }
            for (edge = (node*, neigh) outgoing fromnode*) {
            if (neigh is unvisited) {
            g = g* + edge.weight;
            f = g + h(neigh);
            add a new element <neigh,node*,g,f> into the fringe;
            }
            }
    }
}
```

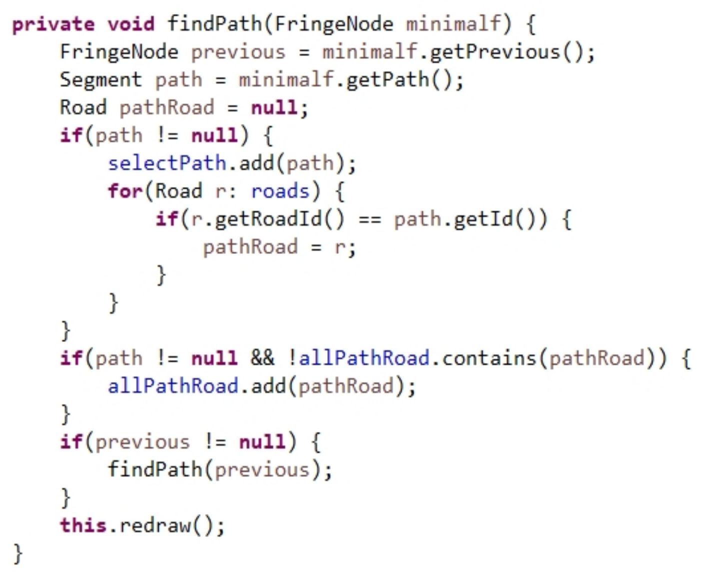
This is A* search algorithm which come from lecture notes that I used in my minimum part. Firstly, I choose two nodes in graph that was implemented by add button start point and end point in initialise method in GUI. Adding mouse listener for the button and set start and goal node as false. When click button, start node and goal node in Auckland system road will become true. For illustrate the button on graph, I add JPanel pathfinder which contain the start point, end point and find shortest path. In addition, put pathfinder to control. I put node to start and end in click on method, record the location of node and put it to Node start and end, and finally draw these two point in redraw method.
Finding the shortest path of startnode and endnode, I use this A* search algrithm which come from lectrue notes in my minimum part. At first, I creat the fringe node which contain the all single element <node, prev, g, f> and match the correct segment of fringe node by using the segmentIn and segmentOut which was store In Auckland road system segment buffer reader. And the fringe node method implements comparable that contain compare to method range by total cost. That means it will order directly from large to small by total cost when fringe node added in to queue. Then creating fringe Queue which store all node of fringe, g represent the cost from start , f represent the total cost. According to the A* search algorithm to complete code. Firstly, I decide whether the start node and goal node is null if one of them is null, that means it lose the condition to find shortest path. Thus, only return and jump this method. if both of them are not null, initialise the first element in fringe is <start, null, 0, f(start)>, f(start) is the straight line distance from start to goal node then I create method estimated cost method for this, and add isVisted to check if the node was visited. If the node is unvisited, set it is visited and remove the fringe node which is biggest total cost in fringe, because I add fringe node in order

and queue is first in first out. Thus, final one in fringe queue is biggest. If node is goal then break, else add renew the cost from node and its' all outgoing segment (neighbor) to fringe queue. Edge weight   is the length of segment.
Finally, obtain the all path that through goal node and it is shortest which represent find path method. This method will run when the node is target and break while loop. Finding final fringe node, if the fringe node have previous node then this fringe node equal the previous one and add the segment of fringe node to List<Segment>. Drawing line by using this list.



```java
private double estimatedCost(Node startnode,Node targetnode) {
    Location start = startnode.getLocation();
    Location target = targetnode.getLocation();
    double estimatedCost = start.distance(target);
    return estimatedCost;
}

protected void shortestPath() {
    selectPath = new ArrayList<Segment>();
    allPathRoad = new ArrayList<Road>();
    //Input: A weighted graph, a start node, a goal node, the heuristic function h() for each node
    //Output: Shortest path from start to goal
    if(startnode == null || goalnode == null) return;
        //Initially all the nodes are unvisited.
        for(Entry<Integer, Node> n : nodes.entrySet()) {
            n.getValue().unVisited();
        }
        //and the fringe has a single element <start, null, 0, f(start)>;
        fringe = new Fringe();
        fringeNode = new FringeNode(startnode, null, 0, estimatedCost(startnode, goalnode));
        fringe.add(fringeNode);
        while(!fringe.isEmpty()) {
        //Expand <node*, prev*, g*, f*> from the fringe
        //where f* is minimal among all the elements in the fringe;
            minimalf = fringe.remove();
            Node node = minimalf.getNode();
            if (!node.isVisited) {
                //Set node* as visited, and set node*.prev = prev*;
                node.visited();
```



```java
private void findPath(FringeNode minimalf) {
    FringeNode previous = minimalf.getPrevious();
    Segment path = minimalf.getPath();
    Road pathRoad = null;
    if(path != null) {
        selectPath.add(path);
        for(Road r: roads) {
            if(r.getRoadId() == path.getId()) {
                pathRoad = r;
            }
        }
    }
    if(path != null && !allPathRoad.contains(pathRoad)) {
        allPathRoad.add(pathRoad);
    }
    if(previous != null) {
        findPath(previous);
    }
    this.redraw();
}
```

At beginning, I use the for each loop to find minimalF, but there are some mistake to loop articulationPoint. Thus I use comparable to get minimal f, this idea come from website. And if put find all path at last of algorithm, It will draw even never find goal node. Thus, I put the find all path to condition if the goal node was find.

```
ArticulationPoint Algorithm
while all visited node {
    Initialise count(node) = infinity, APs = {};
    Randomly select a node as the root node, set count(root) = 0,
    numSubTrees = 0;
        for (each neighbour of root) {
            if (count(neighbour) = infinity) {
                iterArtPts(neighbour, 1, root, unvisitednode);
                numSubTrees ++;
            }
        if (numSubTrees > 1) then add root into APs;
        }
    remove node
    }
---------------------------------------------------------------------------------------------------------------------------

iterArtPts(firstNode, count, root, unvisitednode) {
    Initialise stack as a single element <firstNode, count, root>;
        repeat until (stack is empty) {
            peek <n*,count*, parent*> fromstack;
            if (count(n*) = infinity) {
                count(n*)= count, reachBack(n*) = count;
                children(n*) = all the neighbours of n* except parent*;
            }
            else if (children(n*)is not empty) {
                get a childfromchildren(n*) and remove it fromchildren(n*);
                if (count(child)< infinity) then reachBack(n*) = min(count(child),
    reachBack(n*));
                else push <child, count+1, n*> into stack;
            }
            else {
                if (n* is not firstNode) {
                    reachBack(parent*)= min(reachBack(n*), reachBack(parent*));
                    if (reachBack(n*) >= count(parent*) then add parent* into
    APs;
                }
                remove <n*, count*, parent*> fromstack;
            }
        }
}
```

For articulation point, I create the Articulation point button to call findArticulationPoint which need all unvisited node, thus I through node that I store in loadNode, and create a list<Segment > APs to find all Articulation point and draw them by using the APS list. In Segment, I store segment to node. thus it can get all path of one node. Firstly, I create Articulation point

constructor which contain all element which it will use such as parent, count and reachBack and so on. Then I Initialise the count of node as infinity, and set APs as null. select firstNode in list as root and set root count and numSubTrees equal to zero. find all neighbours of root and iterative articulation point, if numSubTrees more then one that means root at end of edge and add it to APs. The purpose of that method is that find the articulation point of root. thus, all unvisited node should be renew when add articulation point. if complete through one node, then remove this node from list. That means this node had been visited and that is why have 4 parameter in this iterArtPts method. Fristly, I Initialise stack as single element < firstNode, count, root> then add this Articulation point to stack, repeat it until the stack is empty. inside of loop peek the element from stack and set node's count. Next, find all neighbours of node add these node to node's children, process correct children to stack and set child reach back. Finally, if node children is empty. Comparing reachBack and count of two node, pop articulation point from stack.
At beginning, I have some mistake in size of articulation point. I was using System.out to print the number and debug it. I find that it will add same node more than once. Thus, I use change list to hash set.

For completion, I change the position of goalnode and startnode, and get neighbour fromNode which is different from ToNode in minimal part. Thus, it can work fine to find reverse path with out duplicate. For print the length and total length on text, I only use two for each loop for road and segment, match segment and road, if they have same id, print length of this segment and add them together.

For challenge, I only change the estimated cost as length divide by speed and road class. Speed and road class, which had been store in assignment 1.