**Pseudocode for A*:**

Input: A weighted graph, of  all the nodes and roads in Auckland. A start node (intersection), an end node (intersection) and the heuristic function h() for each node (**straight line distance between 2 nodes**)

Output: the shortest path from start to goal

All nodes are unvisited (unvisited being defined as **a list of unvisited nodes**, rather than a boolean, as a boolean has to be reset) and the fringe (**priority queue of fringe objects, holding node, prev node, g** (cost from start) **and h**(estimated cost to end)) has a single element <start, null, 0, f(start)>;

**Method:**
**Initialisation:**
- A list of nodes (unvisited nodes)
- A PriorityQueue of fringe objects

**Iteration:**
```
While (the fringe is not empty){

    expand the <node*, prev*,g*,f*> from the fringe, which is the corresponding element where
    f* is the minimal cost among all the elements in the fringe (Called this node 'current')

    if(current is unvisited){
        set current as visited,
        set current.prev=prev*;

        if (current.node == end node) break;
        if (no restriction on the path from prev->node->neigh){
            for (edge = (node*, neigh) outgoing from node*){
                if (neigh is unvisited){
                    g=g*+edge.weight;
                    f=g+h(neigh);
                    add a new element <neigh, node*,g,f> into the fringe
                }
            }
        }
    }
}
```
Obtain the shortest path using the prev field

**Pseudocode for Articulation Points**
**Initialisation**:
*Initialise* depth of all nodes as depth(node)=∞, meaning all nodes are unvisited;
Initially, Articulation Points is an *empty set*. (If it is not empty, the code has been **run before**, and thus does not need to be rerun, just **return the set**)
Create a copy of all the nodes, to ensure even disconnected parts of the graph are checked.
While the **copy of the nodes is not empty:**

Randomly select a node as the **root node**, set depth(root) to 0, numSubTrees(0).

```
for (each neighbour of the root){
    if (depth(neighbour)==∞){
        **iterateArtPoints**(neighbour, 1, root)
        remove root from the large set of all nodes
        numSubTrees++
    }
    if (numSubTrees>1) Then add root toArticulation points set
}

**iterateArtPoints**(firstNode, depth, root){
    initialise stack <first node, depth, root>
    **repeat until** (stack is empty){
        peek next item on stack (n*,depth*,parent*)
        if(depth n* =∞ ){
            depth(n*)=depth* (depth stored in tuple)
            reachback(n*)=depth*
            children (n*)= all neighbours apart from parent
        }
        else if (children (n*)is not empty){
            get a child from children (n*) and remove it from children (n*)
            if (depth(child)<∞) then reachback (n*)=min (childdepth, reachback)
            else push <child, depth(n*)+1, n*>
        }
        else {
            if (n* is not the first node){
                reachback (parent*)=min (reachback(n*), reachback(parent))
                if (reachback(n*)>=depth(parent*) then add parent* into APs
            }
            remove <n*, depth*, parent*> from stack
            remove node from the large set of all nodes
        }
    }
}
```

**Report:**

**Features of my program:**

- A* search, extended from the minimum requirement
  - Left and then right clicking will select two nodes, start and end. Pressing the distance button will then provide the shortest route (highlighted) based on distance
  - Pressing the time button will highlight the shortest distance based on time, aka speed limits
  - Toggling the class button (green for on, clear for off) followed by the time button, will run the same time based search, but taking road classes into account. This means that it will prefer higher classed routes, even if they are seen as slightly slower.
  - One way roads are incorporated, meaning you cannot travel the wrong way down a road.
  - The output is printed to the screen, meaning that road names, the length traveled on them, and time taken, are all printed, along with the total time and kilometres taken.
  - Restriction data is also included, this means that if there is a restriction on a particular path, it will not be taken.
- Articulation Points
  - Clicking the articulation points button will toggle the appearance of articulation points (will be highlighted)

**Path Cost and Heuristic:**

- **Distance**:
  - Actual cost: The length of the Segment connecting this and the previous node. This is added to the existing cost from the start, calculated the same way.
  - Heuristic: Straight line distance between given node and the end node
- **Time**:
  - Actual cost: The length of the Segment connecting this and the previous node, divided by the speed of that road. This is added to the existing cost from the start, calculated the same way.
  - Heuristic: Straight line distance between given node and the end node divided by the max speed.
- The key differences between the actual and estimated cost is that the heuristic is calculated as a straight line, the shortest distance from a->b. The actual cost is the length of each segment along the way. With time, the heuristic is divided by the mac speed, while the actual cost is divided by the speed of the road.

**Testing**:

- A* Testing:
  - The basic testing was done by selecting two nodes and calculating the expected path, and seeing if they match.
  - For one ways, I found roads which are one ways, and made sure the path would not go through them
  - Restrictions were the same as one ways, I manually found an intersection not allowed, and tested that it would not take the path.
- Articulation points
  - Testing these was relatively easy, as we were given the expected number of points. I simply printed the number it found (size of the set) and compared it for the small and large maps.