Assignment Two Report

Auckland Road System

Route Finding:

- Specifies Start and End points of the path by the use of left and right clicks. Selected points are displayed as its NodeID's on the top left text boxes as 'origin' and 'destination'. The 'Find Path' buttons are pressed to initiate the A* algorithm search depending on Distance or Time.
- Contains a Priority Queue of FringeNode elements, each FringeNode has <Node, Parent, costToHere, totEstCost>. Its priority is defined by the totalCostToGoal.
- Uses an appropriate cost measure (sum of segment lengths) and an appropriate heuristic – Euclidean distance between start and goal.
- Highlights and prints out the roads on the route, no duplicates and gives the right lengths for each road along with the total length of the route.
- Uses one-way roads correctly, identifies if a segment is one-way only if it is then ensure that we go from start to end otherwise consider other segments/routes.
- Takes into account restriction information, recognizes not to take a path that you cannot turn into, finds another route if this is the case.
- Takes into account intersection constraints – traffic lights, checks that if the 'end' field of the segment in question has lights and if it does then add an additional cost to the totEstCost – make the path more expensive.

Pseudocode A:*

```
Initialise Fringe (PriorityQueue<FringeNode>), List<Segment> path
Initialise: All Nodes: set pathFrom – null, setVisited – false

Enqueue – Start Node <origin, null, 0, calcDistHeuristic(origin, destination)>

while( fringe is not empty){

      FringeNode fn = fringe.poll();   //Dequeue –based on priority of FringeNode
      Node node = fn.getNode();

      if( parent of fringeNode not null){    //Exclude initial case where no seg

            seg = get segment corresponding to Node and Parent of the FringeNode
            path.add(seg);      Add segment to path
```

```
	}

	if( node not visited){
		mark Node as visited
		set Node pathFrom to parentNode of FringeNode
		set Node cost to costToHere of FringeNode
	}

	if( node == goal)
		break;			//Reached end goal

	Node nhb = null;
	for(Segment seg : node.outNeighbours()){		//Process neighbours

		if (seg is restricted)		//Uses restriction data
			..consider other segments

		nhb = getNhbOneWay(s);		//Uses one-way Data
		..check if segment is one way
		..if it is then ensure that nhb = end Node of segment
		..otherwise consider other segments


		double costToNeigh = costToHere + seg.length;
		double estTotCost = costToNeigh + calcDistHeuristic(node, nhb);

		fringe.offer(new FringeNode(nhb, node, costToNeigh, estTotCost);
	}
}

calcDistHeuristic(Node start, Node end){}	..calculates Euclidean distance from start to end

getNhbOneWay(Segment s){} ..returns the right NHB node depending if segment is oneWay or not

FringeNode(Node node, Node parent, Node costToHere, Node totEstCost){}  ..FringeNode Element
```

Path Cost & Heuristic Estimate:

- The Distance cost function is defined by the sum of segment lengths and its heuristic estimate is based on the Euclidean distance from point A to point B. The heuristic is consistent as the difference from the heuristic of A to the heuristic of B is less than or equal to the edge weight from A to B. i.e. (h(A) – h(B) <= Wxy). Essentially, the heuristic improves as we get closer to the goal, distance to goal gets shorter and shorter. It is also admissible as it underestimates the remaining cost at each point of the path.
- The Time cost function is defined by the sum of time taken to get to a certain point, time is calculated by using the formula time = distance/speed for each segment along a path. The Time heuristic is both admissible and consistent as it ensures that at each point it always underestimates the remaining cost and the heuristic improves as we go along the path – i.e. time gets shorter and shorter as we get closer to the goal.

Articulation Points:

- Finds the articulation points on all components of the graph (including sub-graphs) by using the iterative version of the algorithm.
- Highlights the articulation points and uses an undirected graph – ignores one way.

Pseudocode:

```
Initialise: List<Node> articulationPoints, Stack<StackElement> activationStack
Initialise: All Node: setVisited – false, setDepth(INF)

for(Node start : graph.getDisconnectedNodes()){     //List of disconnected Nodes

      setDepth – 0
      numSubTrees – 0

      for ( Each neighbour of start ) {
            if( nhb.depth == INF){
                  findArtPts(nhb, start);          //Find ArtPts
                  numSubTrees++;                   //Increment SubTrees
            }
            If(numSubTrees > 1)
                  Then start is an artPts, add to list
      }
}
```

```
findArtPts(Node nhb, Node start){

	activationStack.push(parent)	//Push FirstNode onto stack

	while(stack not empty){

		elem = stack.peek
		node = elem.getNode();

		if(elem has no children){				//FIRST CASE – NO CHILDREN
			set the Node depth to the elements depth
			set the Elements reach to the depth of the element

			for( Neighbour of Node)
				if(neighb != node)
					elem.getChildren().add(nhb)
			get the neighbours of the node and add it to its children
		}

		else if( elem has children){			//SECOND CASE – CHILDREN TO PROCESS

			child = dequeue from children
			if(child.depth < INF)
				set elem reachback to Min(elem.reach, child.depth)
				..set its reachback to the lower value; between the elements reach back or the childs depth
			Else
				Push onto stack a new Element containing the child, add 1 level deeper to its depth, and another
			stack element containing the Node of the current element along with initial values
		}
		Else							//FINAL CASE – FOUND ART PTS
			If(node != firstNode){
				If(elements reach back is greater or equal to the depth of its parent Node){
				Then we have found an articulation point
				-  the Node inside the parent of the current element
```

```
                        Get the parent stack element and set its reach back as the minimum between the reachBack of the
      Parents element and the elements reach back

            Finally.. pop the element
                        }
                  }
      }
}
```

```
Graph.getDisconnectedNodes(){}   ..Returns a list of disconnected Nodes from each of the sub components of the graph. Achieved
by performing a Breadth First Search on each component of the graph.

StackElement(Node node, int reach, StackElement parent){}  ..To construct a new stack element containing a parent stackElement

StackElement(Node node, int reach, PriorityQueue<Node> children, boolean alt ){} ..To construct with children to process
```

Testing:

- Testing was done by using a combination of print statements and debugging tools. The most common way to test was each time a FringeNode was processed print out its elements – Node, Parent, CostToHere, TotCost. Trace its output to ensure that at each point of the path the Nodes being processed are correct and that the costs and heuristic estimates are conforming to the rules of consistency and admissibility i.e. is it underestimating the remaining cost and is the distance to goal improving/reducing as we get closer to the goal.