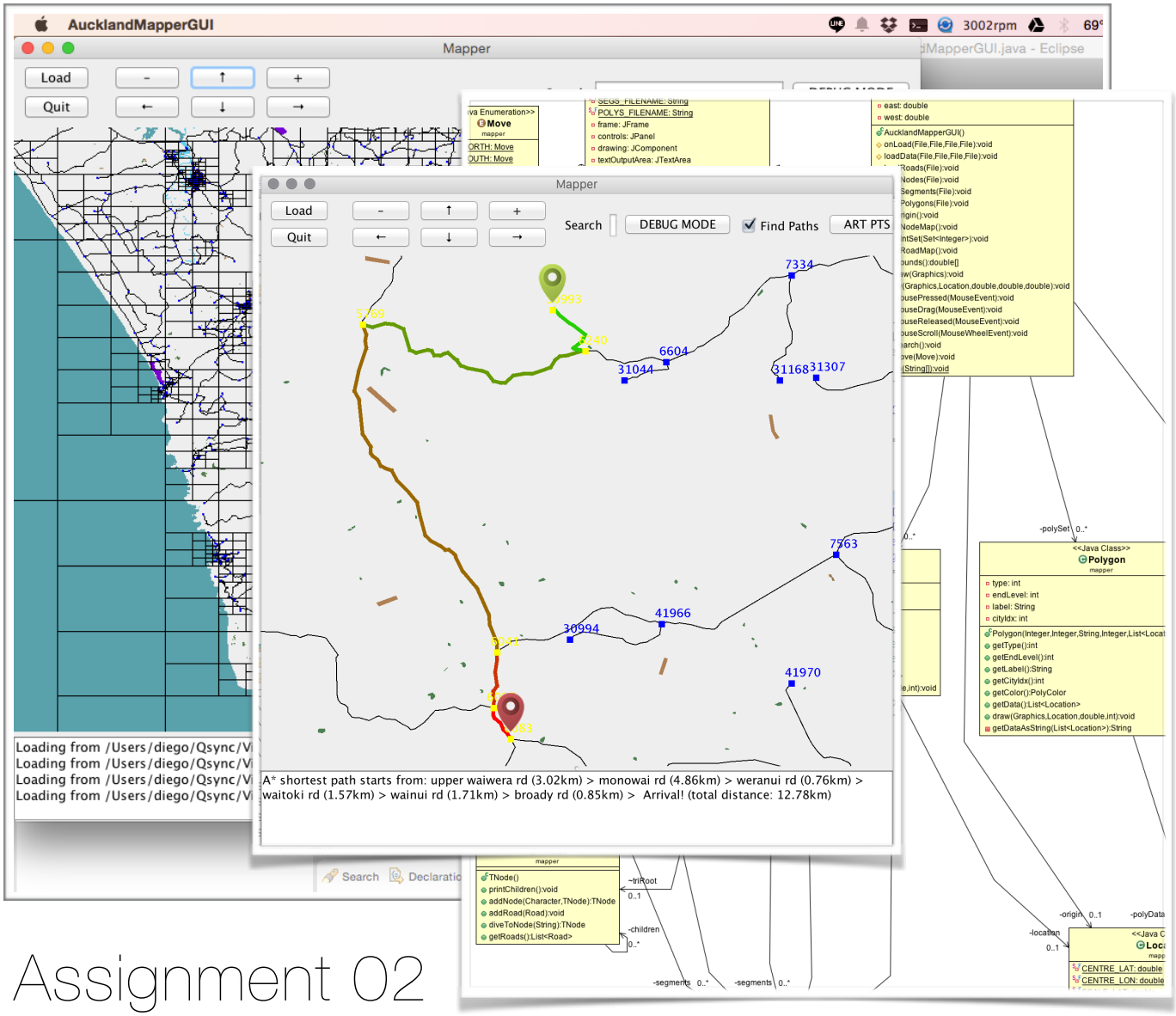
COMP261 2015 T1



Assignment 02
Auckland Map

Prepared by: Diego Trazzi
31 March 2015
ID number: 300338937

NEW FEATURES FOR V02

Minimum and Core features

- [x] Has a way of specifying the start and end of a route.
- [x] Has a priority queue of appropriate elements (node, parent, costToHere, totEstCost).
- [x] Uses an appropriate cost measure (sum of segment lengths).
- [x] Uses an appropriate heuristic.
- [x] Uses the appropriate graph structure of segments from a node.
- [x] Prints out the roads on the route.
- [x] Correctly selects shortest paths.
- [x] Finds articulation points in one part of the graph.
- [x] Uses the correct graph structure, i.e. ignores one way.
- [x] Displays the selected nodes.
- [x] Finds articulation points in all components of the graph.
- [x] Uses the iterative version of the algorithm
- [x] Do they have a report with pseudocode of their algorithms in it?

Completion features

- [x] Uses one-way roads correctly in the route-finding.
- [x] Highlights the route on the map as well as printing it.
- [x] Removes duplicate roads from the printout, and give the right lengths and total length.

Challenge features

- [ ] Allows user to select distance or time, and find fastest route, using road class and speed limit data, and using an admissible heuristic.
- [x] Can they explain and justify their cost function and heuristic?
- [x] Takes into account restriction information.
- [x] Takes into account intersection constraints such as traffic lights to prefer routes with fewer lights.

Additionally, instead of pick up the template model for assignment two, I have opted fro pursuing with my version of the code and addressing the notes received during my previous revision session.

BUG FIXES FROM V01

Although, probably this work is useless in the eyes of the tutors and marker stuff :-(, I have addressed the feedback received during the previous revision session:

A. In the previous version the trie was very slow, therefore to reduce the overhead I have updated the search query to start only after pressing enter.

B. It was also noted that the panning wasn't working as Linux might have different mapping for BUTTON2. This new version of the Auckland Mapper has now a better panning/zooming system, which works on OS X and Linux, and also draws node points with variable size (according to zoom factor).

C. The zoom factor has constraints to avoid zooming too close in the map, or too far out.

D. Zooming is now more user friendly: entered on mouse position

E. Node IDs are displayed as the zooming factor increases.

F. When displaying intersections, the textbook doesn't show segment repetitions.

A* SEARCH

To perform an A* search on a set of nodes we need to allow each node to: remember if was visited already, which node came from, the distance travelled so far and the heuristics distance to the goal.

Throughout the research conducted (workshops, help-desks and talking to colleagues) I noted that this algorithm could be implemented in two different ways: either by adding the required information onto the original Node class, or by making a "wrapper class" for the Node class and adding the field necessary for the A* algorithm onto this wrapper class.

Node Class

Adding the fields for the A* to your existing Node class might be easier to implement but adds an O(n) cost to the algorithm as will have to pre-process (reset) all the nodes in the graph.

Wrapper Class

Making an A* node "wrapper class", which contains a visited field, from field, and an instance of the intersection nodes has the benefit that we'll only need to initialise those nodes (neighbours) necessary for the A* at runtime when making the comparisons.

Finally, because I need to be able to compare nodes, we need to implement the wrapper class with the Comparable interface, so that I can perform comparisons.

By implementing the Comparable interface the priority queue of SearchNodes will put the nodes with smallest distance to the head of the queue.

Assertions

While implementing larger parts of code, such A* star or articulation points, I found useful to add some sanity checks thought the code by using assertion points, which can be enabled at runtime from the Java VM. An example of assertion is checking that the number of edges returned from a node equals the number of neighbours.

```
public Map<Node, Segment> getNeighbours(){
    Map<Node, Segment> neighbours = new HashMap<Node, Segment>();
    for (Segment seg : segments) {
        if (seg.getNodeStart().equals(this)) { // if node start is this node, neighbor is nodeEnd --> add nodeEnd, distance
            neighbours.put(seg.getNodeEnd(), seg);
        } else { // else, the node end is this node, neighbor is the nodeStart --> add nodeEnd, distance
            neighbours.put(seg.getNodeStart(), seg);
        }
    }
    assert segments.size() == neighbours.size();
    return neighbours;
}
```

A* Pseudo code

To implement the A* search I utilised the following pseudo-code:

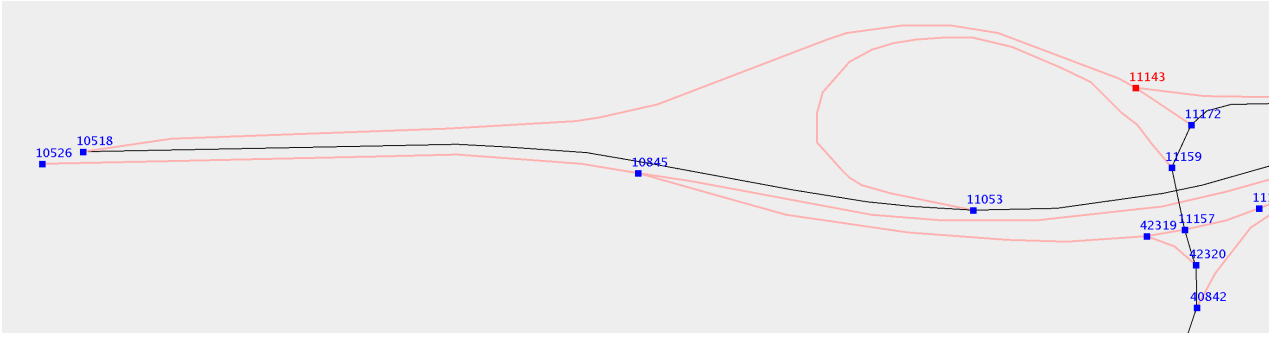
```
Create new PriorityQueue of SearchNode     -> FRINGE
Create new set of visited search nodes     -> visitedSearchNodeSet
Create new set of visited nodes            -> visitedNodeSet

Add the first node (startNode) to the fringe

While the fringe size > 0:
        dequeue the highest node
        if this node is not being visited yet
        add it to the visitedSet and process it:
                BASE CASE: if the node equals goal, add exit here and return this node
                otherwise:
                        calculate the cost to here
                        add the node to the visited set
                        get the neighbours of this searchNode and if are not visited, add them to the fringe
```

One way direction and code checking

To implement A* working with one direction street (such as high-ways) I have updated the node class to differentiate between those segments who are going into the node (segmentIn) and those going outward (segmentOut). To implement the code correctly and make sure it giving appropriate direction, I have also extended the GUI to display nodes ID dynamically with scale factor to avoid overcrowding when zoomed out. This plus an extensive use of verbose information allowed me to check, and discover quite a few errors in the GPS data, which I believe coming out from manipulation and merging of some original data. For example, node **10518**, according to Google Maps, sees cars travelling from EAST to WEST, but our data seems to have

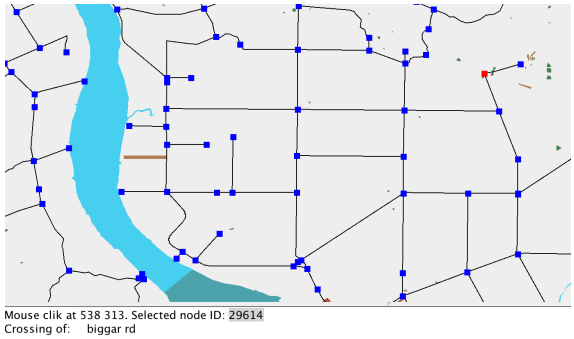


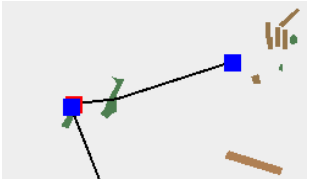
merged the information and with another segment (black line) and thinks there is an out segment from 11053 to 10518, which is incorrect as shown in the debugging window:

```
Mouse clik at 1076 470.
Selected node ID: 11053
Crossing of:    state highway 16  northwestern motorway
Node 11053 [segments=3, segmentsIn= 37368 11159, segmentsOut= 10518]
```

Data validation

I have noticed during my tests that there is some missing data between the nodeID which are extremely close to one-another. For example, node **29614**, although exists and present on my maps, and has one segment, it's not connected to the rest of the road-system, because there are no edges between 29614 and 19612. These two coordinates are very close to each other so cannot be seen by naked eye. I have used google maps and other formulas to calculate the actual distance between measurements and are ~13m apart. In conclusion, the two point look like one and if you click on the one on its left and the one on his right there is no connecting segment.





I have added extra checks, to avoid nullPointer exceptions and to fall in infinite loops.

As the picture shown on the left there are certain nodes which almost overlap and are not connected by a segment in between. This results in portions of the graph being unconnected although look (and should be) connected.

One example is :

29614 -36.078330 174.054180
29612 -36.078220 174.054240

Coordinates	Distance	Found By	Date
S 36 4.693 E 174 3.254	13 meters NE (23°)	-	-

Download Waypoints | Map Waypoints

Order cost and heuristic

A* guarantees to find the lowest cost path in a graph with nonnegative edge path costs, provided that we use an appropriate heuristic. To make the heuristic estimate appropriate it must be admissible, i. e. it should, for any node, produce either an underestimate or a correct estimate for the cost of the cheapest path from that node to any of goal nodes. This means the heuristic should never overestimate the cost to get from the node to the goal. The heuristic should also be monotonic, meaning that should always decrease or increase, being consistent among all the samples.

Finally, the heuristic should also be cheap to compute. It should be certainly O(1), and should preferably look at the current node alone. Recursively evaluating the cost as you propose will make your search significantly slower, not faster, therefore other algorithms, such as Dijkstra could be of better use if the O cost was larger than O(1). In our case, the cartesian distance between two points is a good heuristic because is both consistent (monotonic) and admissible.

The Big O notation for the A* algorithm will vary between the best case - O(log n) - and worst case scenario O(n*e). In the case the heuristic are all equal to 0 for example, the A* will perform just like a Dijkstra and process every node. So h(n)=0 is still an admissible heuristic, only the worst possible one. So, of all admissible heuristics, the tighter one estimates the cost to the goal, the better it is.

ARTICULATION POINTS

Although in A* search I implemented a new wrapper class to avoid traversing the entire graph on each iteration, in the case of articulation points it's necessary to visit all the nodes in the graph, therefore fields such as depth can be stored on the Node class directly.

Pseudo Code Recursive Articulation Points

To implement the articulation points algorithm I utilised the following pseudo-code:

```
Copy all the nodes into the articulation nodeVisited set
Initialise : for each node: node.depth = infinite , articulationPoints = { }
start .depth = 0, numSubtrees = 0

for each neighbour of start
         if neighbour.depth == infinite then
         recArtPts(neighbour, 1, start)
           numSubtrees ++
     if numSubtrees > 1 then add start to articulationPoints

recArtPts(node, depth, fromNode):
         node.depth = depth, reachBack = depth,
         for each neighbour of node other than fromNode
                  if neighbour.depth < infinite then
                           reachBack = min(neighbour.depth, reachBack)
           else
                  childReach = recArtPts(neighbour, depth +1, node)
                  reachBack = min(childReach, reachBack )
                  if childReach >= depth then add node to articulationPoints
         remove node from nodeVisited set
         return reachBack
```

Because the graph has multiple components I had to run the algorithm multiple times (k-times), and to avoid traversing the entire graph several times, I have added a set (hashset) to store those nodes which haven't been processed yet. In this way I was able to perform faster runs of the articulation points algorithm. Here is an output of the actual time taken to find all the articulation points in the large dataset:

```
"Articulation point with recursion method took: 27 milliseconds"
```

27 milliseconds to analyse all 35,000+ nodes, not bad!

Order cost

The order cost for the articulation point in the Tarjan (this is what this algorithm is named) procedure is called once for each node; the for-all statement considers each edge at most twice. The algorithm's running time is therefore linear in the number of edges and nodes in G, i.e. O(|V|+|E|). Therefore the order cost for running the articulation points is O(n).

UML

Finally, although, not required by the assignment handout I have prepare a UML representation of of my Assignment. I have done this though out the working stages for sanity check and for better organisation. Here is the final representation: