The Auckland Road System

In assignment 1, you built a program to read data about a collection of roads and display it as a map that the user could interact with. For this assignment, you will extend your program to provide the user with two tools: route finding using A* search, and critical intersection identification with the Articulation Points algorithm.

For this assignment, you may extend your program from assignment 1, or you can use the partial solution code provided (which works up to the completion, but no further).

Resources

The assignment webpage also contains:

- An archive of the template code and a small example.
- An archive of the road data files.
- The marking guide.

Note: In the template code, there are two class files (`Parser.java` and `Parser-Stream.java`). `Parser.java` uses traditional implementation, while `Parser-Stream.java` uses Stream functionality. They are essentially the same, so feel free to use either of them.

To submit

You should submit four things:

- All the source code for your program, including the template code if you use it. **Please make sure you do this**, without it we cannot give you any marks. **Again: submit all your `.java` files**.
- Any other files your program needs to run that *aren't* the data files provided.
- A report on your program to help the marker understand the code. The report should:
  - describe what your code does and doesn't do (e.g., which stages and extensions above did you do).
  - give a detailed pseudocode algorithm for the main search.
  - describe your path cost and heuristic estimate
  - give a detailed pseudocode algorithm for the articulation points.
  - outline how you tested that your program worked.

The report should be clear, but it does not have to be fancy – very plain formatting is all that is needed. It must be either a `txt` or a `pdf` file. It does not need to be long, but you need to help the marker see what you did.

Note that for marking, you will need to sign up for a 15 minute slot with the markers.

Route finding

The first feature is a route finding tool like that in Google maps or car navigation systems. The feature should allow the user to specify two intersections on the map and will then find (using **A* search**) and display the shortest route between those two locations. It should highlight the route on the map (by colouring all the road segments along the route) and should also output a list of all the roads along the route, along with the lengths of each part of the route and the total length of the route. For example, it might print a route in the form:

```
Beauchamp Street: .45km
Karori Road: 1.3km
Chaytor St:  1.11km
Glenmore St:  0.039km
Upland Road: 0.909km
Glen Road: 0.4km

Total distance = 4.208km
```

The basic version of the route finding will find the route with the shortest distance and display it on the map and by listing all the road segments (and lengths) along the route. There are improvements that you can make on this basic version:

- Combine adjacent segments of the route that are all from the same road.
- Allow the user to select between time and distance, so that it can either find the shortest distance route or the fastest route, assuming the speed limits specified in the road info file. (This data does not appear to be correct, but use it anyway).
- Take into account the turn restrictions in the file `restrictions.tab`, which specifies the turns that are not allowed at intersections (for example an intersection might have a 'no right turn' restriction from one of the roads entering the intersection). You will need to extend your data structure to cope with this.
- Take into account the delay at intersections, e.g. traffic lights will slow you down.

You should note that there are at least 100 pairs of intersections in the Auckland data that have two road segments connecting them. In other words, the graph of roads is a multigraph. There are no nodes that have a road segment that connects from the node back to the same node. Your program should take these properties into account.

Critical intersections

The second feature is an analysis tool that might be used by emergency services planners who want to identify every intersection that would have bad consequences for emergency services if it were blocked or disabled in some way. An intersection that is the only entrance way into some part of the map is an critical intersection ('articulation point'). Your program should identify all such intersections and colour highlight them. Note that emergency services don't care about one way roads - in an emergency they can go either way, if necessary. They also don't care about 'no right turn' restrictions.

The articulation points algorithm assumes that the graph is undirected, doesn't care about the lengths of edges, and doesn't care whether there are multiple edges between two nodes. In fact, all it needs is a collection of nodes and the set of neighbouring nodes of each node. This means that you cannot use exactly the same data structures as you used for the route finding algorithm. For the route finding, each node (intersection) needed a set of the edges (road segments) coming out of the node, where the segments included the length and the node at the other end; for the articulation points, each node needs a set of neighbouring nodes (i.e. the nodes at the other end of segments both in and out of the node). You should extend your program to build this structure as it reads the data.

The restrictions data

Added to the roads, nodes, and segments files from last assignment, later stages of this assignment may also use the restrictions data:

Restrictions prohibit traveling through one path of an intersection. The restriction data is in the `restrictions.tab` file, with one line for each restriction. Each line has five values: `nodeID-1`, `roadID-1`, `nodeID`, `roadID-2`, `nodeID-2`. The middle `nodeID` specifies the intersection involved. The restriction specifies that it is not permitted to turn from the road segment of `roadID-1` going between `nodeID-1` and the intersection into the road segment of `roadID-2` going between the intersection and `nodeID-2`.

Your program

You are suggested to solve this problem in stages as below, the provided marking guide gives a detailed breakdown of core/completion/challenge.

Minimum – Basic A*.

- Ignore the road class, speed, and restriction data; Extend your program to allow the user to select two intersections - start and end. Find the shortest path from the start location to the goal location, printing out the sequence of road segments (including road names and segment lengths). Your program should use an A* search algorithm. You may use Euclidean distance between a node and the goal node as the heuristic (see `Location.java`) For testing purposes, it may be useful to include the nodeID's and roadID's in the output.

  Hint: The report you write must include a detailed pseudocode specification of the A* algorithm you used. Write this detailed pseudocode algorithm before you try to code it up! And then update the pseudocode if you have to change the code later. It is seldom a good idea to launch into coding of this kind of program without working through the detailed design first.

Core – Articulation points.

- Ensure that, from a given `Node`, you can quickly find all the adjacent `Nodes`.
- Implement the articulation points algorithm to find *all* the nodes that are articulation points, then display all those points in a different colour. This can be done with either the recursive or iterative version of the algorithm, but the iterative version is worth more marks.

  Hint: again, write up a detailed pseudocode algorithm.

Completion – Improved route finding.

- Incorporate one-way roads into your route finding system, so that a route will never take you the wrong way down a one-way street.
- Make the output of the route nicer: it should merge a sequence of road segments all from the same road into a single step, and include the total length of the step.

Challenge – More improved route finding.

- Take into account the restriction information.
- Add buttons to select distance or time. Include road class and speed limit information to make your search prefer routes on high class roads and faster roads. You may have to do some experimenting to work out a good way of using these factors. You also need to make sure that your heuristic estimate is still a lower bound on the actual cost.
- Incorporate traffic light information and prefer routes with fewer traffic lights. (You may have to go and find the data yourself – some exists, but apparently it isn't very reliable.)