The Auckland Road System

In this assignment, you will implement a Road System for Auckland, to provide the user with route finding using A* search, and critical intersection identification with the Articulation Points algorithm.

For this assignment, you may reuse the core part of your program (for graph representation and display) from assignment 1, or you can use the template code provided.

Resources

The assignment webpage also contains:

- An archive of the template code.
- An archive of the road data files.
- The marking guide.

The data

There are various sources of geographical information such as road data. With the help of Andrew Rae and Dr. Mairead de Roiste from SGEES, we have a collection of data on roads in the Auckland region. The data is from the NZ Open GPS Project (`http://nzopengps.org/`). The original data is in a format designed for certain GIS tools; we have processed the data somewhat to make it easier for you to work with. The data is now in the form of a collection of tab separated files. The details are given below.

Road data is not particularly simple. A road system consists of a set of roads. Each road has a name and goes along a particular path (a sequence of coordinate positions). The road also has properties such as its type, its speed limit, whether it is one-way, etc. A road may have different properties in different parts - the speed limit might be higher in one part than another, or part of it might be one way.

Roads intersect with each other, but not just at their end points. A road might have a large number of intersections along it, each intersecting with a different road. The intersections will break up a road into a sequence of road segments; each segment is a part of a road going between two intersections (or an intersection and the end of a road).

There are also constraints on intersections - it may be forbidden to turn from one road to another at an intersection, even if going the other way is allowed (e.g. 'No Right Turn' signs).

Therefore, the data consists of three kinds of objects we care about:

Nodes are locations where roads end, join, or intersect. The node data can be found in the `nodeID-lat-lon.tab` file: a tab separated text file with one line for each node, specifying the ID of the node, and the latitude and longitude of the node. Note that latitude and longitude are specified in degrees, not distances. One degree of latitude corresponds to 111.0 kilometers. One degree of longitude varies, depending on the latitude, but we assume that in Auckland, one degree of longitude is 88.649 kilometers. This means that when you are computing distances between two points, you must scale the latitude difference by 111.0 and scale the longitude difference by 88.649.

Road Segments are a part of a road between two intersections (nodes). The only intersections on a road segment are at its ends. The data includes the length of each segment. The road segment data is in the `roadSeg-roadID-length-nodeID-nodeID-coords.tab` file: a tab separated text file with one line for each road segment, specifying the ID of the road object this segment belongs to, the length of the segment and the ID's of the nodes at each end of the segment, followed by the coordinates of the road segment for drawing it. The first line of the file specifies the fields in each line. The coordinates are given as a sequence of latitude and longitude coordinates of points along the centerline of the road segment. The coordinates consist of an even number of floating point numbers, and there are always at least two pairs of numbers (for each end of the road segment); some segments have a lot more coordinates.

Roads are a sequences of segments, with a name and other properties. These need not be an entire road - a real road that has different properties for some parts will be represented in the data by several road objects, all with the same name. A very important property of roads is whether they are one-way or not. The road data is in the `roadID-roadInfo.tab` file, with one line for each road object. The first value on the line is the roadID. The columns are specified at the top of the file. The meaning of the numeric columns is specified in the `README.txt` file.

Later (Challenging) stages of this assignment 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`.

There are two datasets:

large/: a set of data for the complete Auckland region, (30035 roads, 12875 distinct road names, 354760 intersections, and 42480 road segments),

small/: a much smaller set of data for a region around the central city (746 roads, 481 distinct road names, 1080 intersections, 1412 road segments). The small data set will be helpful for testing your program since it should be much faster to load.

The template code

The template code has its main method in **Mapper.java**. It has the following functionalities:

- Parse the data into the program and store as a **Graph**. In the graph, each **Node** is an intersection, and each road **Segment** is an edge which connects two intersections.
- Display the graph on the screen. It can shift the display, and zoom in/out.
- When clicking on the screen, it will highlight the closest node in the graph, and show its information in the text field.

If you use the template code, feel free to modify the classes if necessary. For example, the template **Node** class contains a single adjacency list as below:

```java
public class Node {
  public final int nodeID;
  public final Location location;
  public final Collection<Segment> segments;
  ...
```

You should separate the **incoming** and **outgoing** segments for route finding.

Note: There are two parsing class files (`Parser.java` and `Parser-Stream.java`) in the template code. `Parser.java` uses traditional implementation, while `Parser-Stream.java` uses Stream functionality (introduced in Java 8). The `Parser-Stream.java` is provided as an example to help you get familiar with how Stream works in Java (not required in this course though). But the two parsing classes have the same functionality, so feel free to use either of them.

Route finding

The route finding 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.

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.

Hint: The graph may be disconnected and contain multiple isolated connected components. Your program should be able to find all the articulation points for all the connected components.

Hint: There are **240** articulation points in the small graph, and **10853** articulation points in the large graph.

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) in the *text output area at the bottom of the window*. 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* search you used, particularly the designed heuristic function. 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.

- Implement the articulation points algorithm to find *all* the nodes that are articulation points, then highlight them. 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.)

To submit

You should submit four things:

- All the source code (.java files) 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
  - 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.