The Auckland Road System

The first two assignments involve building a program that lets a user view and search the Auckland Road system. For this assignment, you are to build a program that will read a collection of files containing information about the roads in the Auckland region, display the information visually, and let the user view and search the data in several ways. The program will need several large data structures, and the key challenge of the assignment is to implement those data structures.

Resources

The assignment webpage also contains:

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

To Submit

You should submit these things:

- All the source code for your program, including the template code. **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.
  - describe the important data structures you used.
  - 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.

Requirements for Assignment 1

Your program should:

- **Read road data from the files** described below and **construct an appropriate data structure to store all the information** about the roads, road segments, and intersections. A road segments is a part of a road that connects two intersections. A road consists of a sequence of road segments. Intersections are either places where two roads meet, or the end of a road. The data structure must include a **graph** of the intersections (nodes) and road segments (edges), but it will also need to store information about the roads.
- **Display the road data visually** by drawing lines for all the roads segments. Each road segment should be drawn as a sequence of straight lines (the points along the road are specified in the data files). The program must allow the user to either **view** the whole of the Auckland region, or **zoom in** on a smaller region. At the minimum, the program should have two views - the whole region and Auckland central. Ideally, the user should be able to zoom and pan to arbitrary views of the data.
- Allow the user to enter the name of a road in a text box, and then **highlight the road** (all its road segments). To do this, the program should store all the road names in a **trie** structure which will act as a searchable index into the collection of road objects.
- Allow the user to click on the visual display of the roads, and then **highlight and display information about the closest intersection**. The information should include the location of the intersection and all the roads that go through the intersection.

The provided marking guide describes what you need to do for the minimum, core, completion, and challenge, as does this document.

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.

There are two datasets:

Full: 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.

Your program

Your program should read the data from the data files into an appropriate set of data structures, and then draw a map of the Auckland road system. The program should allow the user to click on the map to select an intersection, and should display the names of roads at the intersection. It should also allow the user to enter the name of a road into a search box, and should then highlight the road on the map by displaying (in colour) each of the road segments that make up the road.

You are not expected to create your own GUI for this assignment, rather, one is provided in the model code called `GUI.java`. This contains an area for drawing the map, a loading button, some navigation buttons, a search box, and a text output area. It is an *abstract class*; the behaviour of how to draw the drawing area, load the data, and what happens when you press the navigation buttons, click the screen, or enter something in the search box are left unimplemented. You will need to extend the `GUI` class and implement these using your data structures. For an example of how to do this, `SquaresExample.java` repurposes the `GUI` class to make an unrelated little game.

You will need to read and understand the first part of the `GUI` class to write your program. While UIs and Swing are not a focus of this course, it is recommended you try and understand how the rest of the `GUI` class works. You are also free to modify it to suit your program, or ignore it entirely and build your own.

Your program will need a collection of Road objects to represent information about the roads. You will need to access roads both by their ID's and by their names. The first can be accomplished with a Map of the road objects, with their IDs as the keys of the map. The second should use a Trie structure, since you need to be able to access all the roads whose names start with a specified prefix.

The intersection and road segment data forms a graph, with the *intersections* as nodes, and the *road segments* as the edges. The graph is directed (some roads are one-way), is a multi-graph (there can be more than one segment connecting the same two intersections), but has no 'looped' segments that start and end on the same intersection.

You need to choose an appropriate data structure for this graph. I recommend using `Node` objects and `Segment` objects, where the `Node` objects include a list of all the `Segments` joined to the intersection. The `Node` objects also need to store their location, both for drawing them and for finding the node the user clicks on. The `Segment` objects need to store their length, the `Nodes` at each end, the `Road` they are part of, and the list of coordinates for drawing them. You should decide whether to use the ID's in the data structure or to simply connect `Node` and `Segment` objects directly to each other; either way, finding connections between `Nodes`

and `Segments` should be efficient.

Solving the problem in stages

Minimum – Parsing, data structures, and drawing.

- Construct classes to represent `Roads`, `Nodes`, and `Segments`.
- Construct a class to store the collection of `Roads` and the collection of `Nodes`. The class should have methods to read the data from each of the files and construct the collections (it should read `Roads` and `Nodes` before reading the `Segments`).

  Hint: Use the `Location` class to represent positions of the `Nodes` and `Segments`. The class includes methods for converting from latitude/longitude to x/y coordinates in kilometers.
- Make methods to read the data files, parse them, and create your data structures. If you're using the `GUI` class, these methods should run when the `onLoad` method is called. This method is passed a `File` object for each of the four files you might be interested in.

  Hint: When reading the `Segments`, you need to add them to the appropriate `Node` and `Road` objects.

  Hint: The `Scanner` class is very inefficient, loading the data is best done with a `BufferedReader`.
- Draw the map by filling in the `redraw` method left abstract in the `GUI` class. This method should call a draw method on the `Segments` and `Nodes`, which should use the passed `Graphics` object. It will make the zooming easier if these methods also take an origin and scale factor as parameters, so they can calculate where on the graphics object they should be drawn.

Core – Using the graph structure and other functionality.

- Allow the user to navigate the map, i.e. implement panning and zooming with the buttons. Whenever these buttons are pressed, the `onMove` method (left abstract in `GUI`) is called, and is passed an enum representing which button was pushed; this is where you should write the movement logic.
- Make the program respond to the mouse so that the user can select an intersection with the mouse, and the program will then highlight it, and print out the ID of the intersection and the names of all the roads at the intersection. This can be done by implementing the `onClick` method left abstract by the `GUI` class, which is called whenever the mouse is clicked. This method is passed a `MouseEvent` object, which contains, among other things, the coordinates of click within the graphics area.

The simplest method will do a linear search through the collection of intersections to find one that is close enough to the mouse position. (Use the methods in `Location` to convert between pixel based positions on the screen and the locations in the `Nodes`.)

- Implement the behaviour of the search box in the top right, which should allow a user to select a road by entering its name. This can be done with the `onSearch` method, which is called whenever the user presses 'enter' in the search box. When they complete their entry, the program should highlight the road with name (*exactly*) matching their input by drawing all the segments in all the Road objects that have the given name in a highlighted colour.

  Hint: Remember that there may be multiple Road objects with the given name, and each Road object may have multiple Segments in it.

  Hint: The search box, and its contents, can be accessed wit the `getSearchBox` method in `GUI`.

- The text output area at the bottom of the window should be used to show information about roads and intersections, and can be accessed via the `getTextOutputArea` method in `GUI`.

Completion – Making a trie and improving search.

- To make the program more usable, we want the search function to highlight all roads whose name is prefixed by the search input, and not just an exact match. A linear search is not an efficient way of doing this, and using some hashing scheme wouldn't work either, as we need to do prefix-matching (unless the hashing scheme was very cleverly designed). A better way is to construct a trie data structure to store all the `Road` objects, indexed by their name. This data structure will need methods to add a new road, and to retrieve all roads matching a given prefix.

- Improve your search using your trie. Highlight and output the name of *all* roads that start with the prefix typed into the search box. For example, there are 8 road names starting with `pol`. If the user's search query *exactly* matches a road name, it should only highlight and show exact matches. For example, there are lots of roads whose name starts with `state highway 1` (for example `state highway 1` and `state highway 17`) but if the input is `state highway 1`, it should select *only* roads named exactly `state highway 1`.

  Hint: The `UPDATE_ON_EVERY_CHARACTER` variable in the `GUI` class can be set to `true` to make the `onSearch` method be called each time the user types a character in the search box, and not just when they press enter.

Challenge – **Quad-trees** and UI improvements.

- A linear search through the intersections to find the closest is also inefficient, but searching for inexact matches for 2D positions doesn't work with hashing or binary

search. (This problem is a standard problem in computer graphics.) The right structure to use for searching for the closest 2D point is called a *quad-tree* (see more details from Wikipedia: `https://en.wikipedia.org/wiki/Quadtree`). Learn about and implement a quad-tree index of all the nodes and use it to search for the closest node to the mouse location.

- The polygon data has gone unused up to now, but it makes the map look much nicer. Incorporate the polygon data into your program and improve your map drawing.

  The polygon information is in the `large/polygon-shapes.mp` file. It follows the "*Polish Format*". In the file, a polygon contains a `Type`, an `EndLevel`, a `CityIdx` and a list of `Data0` (the vertices of the polygon). You will need to use the `Data0` to draw the polygon. The `Type` information can be found from the user manual `http://magex.sourceforge.net/doc/cGPSmapper-UsrMan-v02.4.pdf` (page 95, section 11.3.3 *[POLYGON] types*). Each polygon has a zoom level, with higher levels being more zoomed out. In the file, the `EndLevel` specifies the maximum level that the polygon is visible at. If the current zoom level is higher than `EndLevel`, the polygon isn't visible.

- The user interface in the `GUI` class is functional, but could be improved further; find a way to significantly improve it. Two ideas are:
  - Implement navigation with the mouse and scroll wheel.
  - Add a drop-down suggestions box to the search box, which the user can select completions from.

Writing it yourself

Make sure that you write the code for the data structures yourself – you will not learn what you need to learn if you use code from somewhere else. You can build on code examples from somewhere else, but do not simply copy large segments of code and make sure that you acknowledge the source appropriately. If we identify any plagiarism, we **will** penalise it.

Appendix

The small and large graphs should look like below.

Figure 1: Small graph.

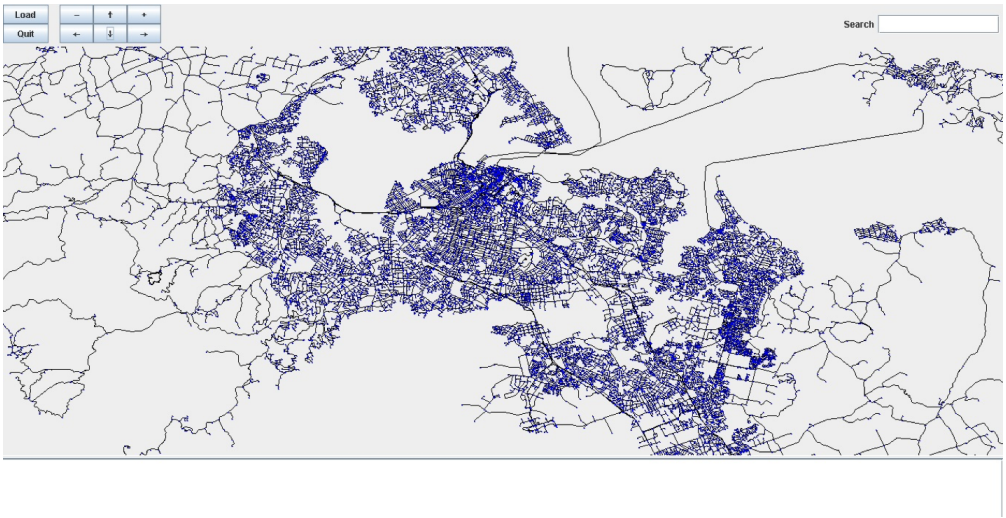

Figure 2: Large graph.