# The Journey Planner Network

The first assignment aims to read a Journey Planner network, display it on the screen, 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 data files.
- The marking guide.

## 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.

  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 the data from the files** described below and **construct an appropriate data structure to store all the information** in the files.
- **Display the data visually** by drawing all the nodes and edges. Each edge should be drawn as a straight line. The program must allow the user to either **view** the whole network, or **zoom in** on a smaller region. 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 stop in a text box, and then **highlight the stop, and all the trips going through the stop**. To do this, the program should store all the stops in a **trie** structure which will act as a searchable index into the collection of stop objects.
- Allow the user to click on a place on the visualisation, and then **highlight and display information about the closest stop to the clicked position**.

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

# The data

The data is for Journey Planning in the Northern Territory of Australia, obtained from `https://data.gov.au/dataset/journey-planner-data-nt`. We have cleaned and processed the data to make it easier for you to work with. The details are given below.

**stops.txt** stores the stops in the journey planning network. In the file, the first line is the title. From the second line, each line represents a stop, including its ID, name, latitude and longitude. The entries are separated by tab.

**trips.txt** stores the trips of the transport network. Each trip is a sequence of stops. In the file, the first line is the title. From the second line, each line represents a trip. The first entry is the *trip ID*. Then, the sequence of stops in this trip is represented by the second to the last entry of the line. The entries are separated by tab as well. **In each trip, there is a directed edge from each stop in the sequence to its successive stop.**

# Your program

Your program should read the data from the data files into an appropriate set of data structures, and then draw the graph. The program should also have the proper functionalities when the user presses the buttons, clicks on the map to select a stop, and type something into the search box. The details of the functionalities are given in the next section.

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 graph, 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 **Stop** objects, a collection of **Trip** objects, and a collection of **Connection** objects. You will need to access stops both by their IDs and by their names. The first can be accomplished with a Hash 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 **Stops** and **Connections** form a graph, with the *Stops* as the nodes, and the *Connections* as the edges. The graph is directed (each trip is directed), is a multi-graph (there can be more than one connection between two stops belonging to different trips), but has no 'looped' connections that start and end at the same stop.

You need to choose an appropriate data structure for this graph. I recommend using **Stop** objects and **Connection** objects, where the **Stop** objects include a list of all the incoming and outgoing **Connections**. The **Stop** objects also need to store their location, both for drawing them and for finding the stop the user clicks on. The **Connection** objects need to store the **Stops** at each end, and the **Trip** they belong to. You should decide whether to use the IDs in the data structure or to simply connect **Stop** and **Connection** objects directly to each other.

## Solving the problem in stages

**Minimum** – Parsing, data structures, and drawing.

- Construct classes to represent **Stops**, **Trips**, and **Connections** between two stops. The class should have methods to read the data from the files and construct the objects.

  *Hint*: Use the **Location** class to represent positions of the **Stops**. 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 stop and trip files.

*Hint*: When reading the **Trips**, you need to add each of the **Connections** in the trip to the appropriate **Stop** objects.

*Hint*: Loading the data is best done with a **BufferedReader**. A tutorial can be found from `https://www.mkyong.com/java/how-to-read-file-from-java-bufferedreader-example/`.

- Draw the graph by filling in the **redraw** method left abstract in the **GUI** class. This method should call a draw method on the **Connection** and **Stop**, 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 a stop with the mouse, and the program will then highlight it, and print out the name of the stop and the id of all the trips going through the stop. 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 stops to find the closest one to the mouse position. (Use the methods in **Location** to convert between pixel based positions on the screen and the locations in the **Stops**.)
- Implement the behaviour of the search box in the top right, which should allow a user to select a **Stop** 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 stop with name (*exactly*) matching their input, and highlight all the trips going through that stop.

  *Hint*: Remember that there may be multiple **Trip** objects going through the stop, and each **Trip** object may have multiple **Connections** in it.

  *Hint*: The search box, and its contents, can be accessed with the **getSearchBox** method in **GUI**.
- The text output area at the bottom of the window should be used to show information about stops and trips, 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 the stops

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 **Stop** objects, indexed by their name. This data structure will need methods to add a new stop, and to retrieve all stops matching a given prefix.

- Improve your search using your trie. Highlight *all* stops that start with the prefix typed into the search box. For example, there are 4 stop names starting with "**VRD**". If the user's search query *exactly* matches a stop name, it should only highlight and show exact matches. Print out the name of the matched stops in the text output area. Also, highlight all the trips going through the match stops.

  *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 stops to find the closest one 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 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.