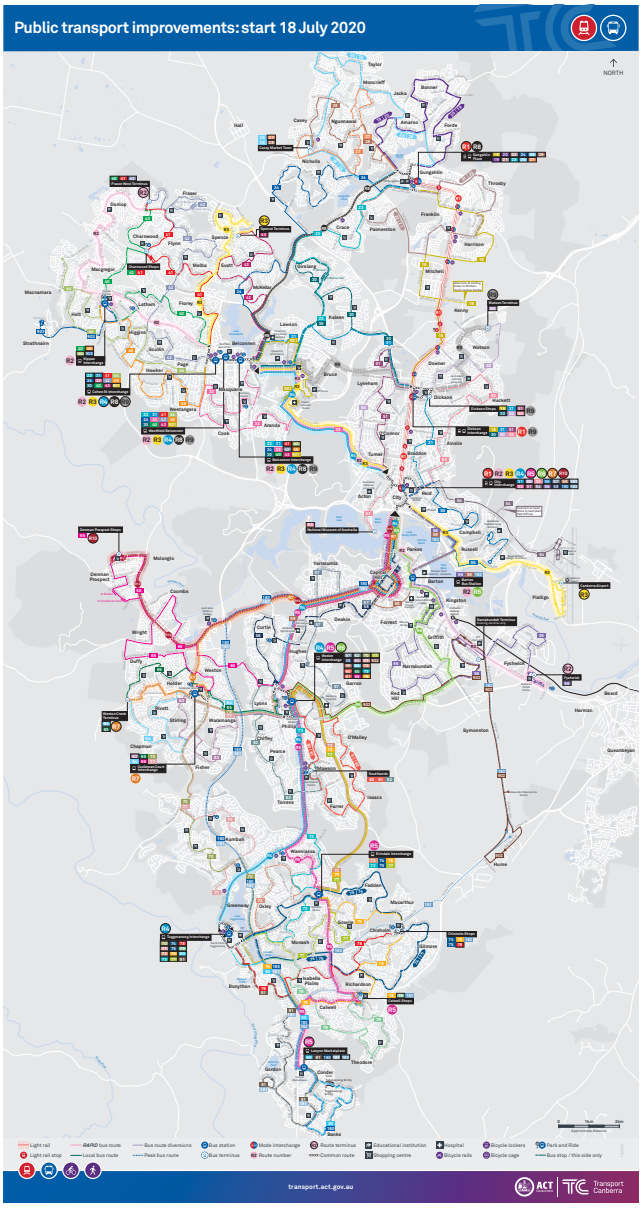
II.2315 – Project : Advanced algorithm and programming

Australia - Canberra



isep
École d'ingénieurs du numérique

Sarah HEOUAINE - David LAMY-VERDIN - Elia TSO

2020 - 2021

Table of contents

1 Introduction

1.1 City of choice

We selected the city of Canberra, capital of Australia, for our project. This city has a total of 2433 stations and 2759 links.

1.2 Goals of the project

This project had for first goal to create a graph representing the transport map of Canberra out of mere data. After this graph has been created, we could use it to reach other goals described in the below list:

- Search algorithms
  Implementation of the Bread-First Search Algorithm
  Implementation of the Dijkstra Algorithm

- Applications of those algorithms
  Searching for shortest paths
  Splitting the map into clusters

Through this project, optimization also had to be done as searching for shortest paths and splitting into clusters a graph as large as Canberra Transport Map was particularly demanding on resources and took a considerable amount of time.

1.3 Collection of data : GTFS files

To build our graph, we used data retrieved from Australia Government official website. This data come as multiple *.txt* files. After studying them, it has been figured out that only *stops.txt* and *stop_times.txt* were actually useful.

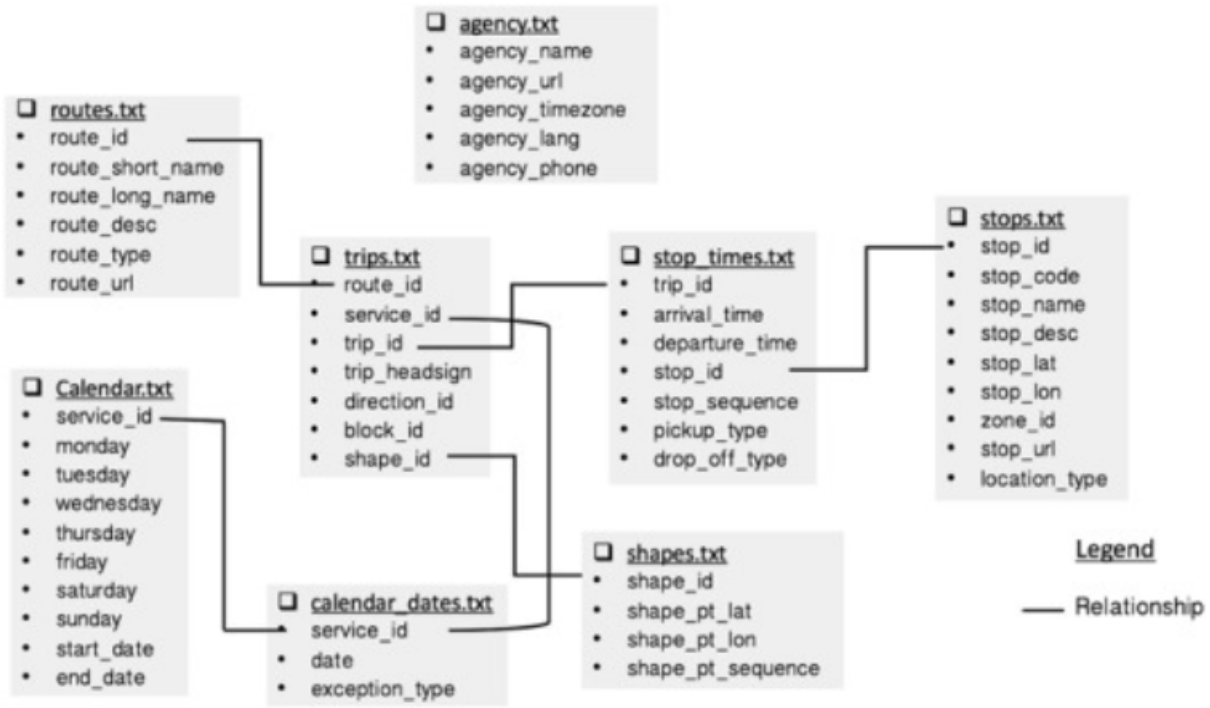


Figure 1: Data included in each *.txt* file and relations between them

Using only those, we could create the stations which are represented in our graph as nodes and we could also link each of them and thus creating our edges.

The *stops.txt* file provide us with *stop_id*, *stop_lat*, *stop_lon* which are all the information we needed to create our nodes while the *stop_times.txt* give us *trip_id* which allow us to create our edges. A trip informs about which stations are linked together and in which order as it defines a bus or subway line such as the R3 line of Canberra for example:

Figure 2: R3 bus line of Canberra

2 Creation of the graph

2.1 Graph

`Graph.java` is the class used for building graphs. The class contains the three following attributes:

- `private Map<Integer, List<DirectedEdge>> map = new TreeMap<Integer,List<DirectedEdge>>()`

  This `Map` is used to store the adjacency lists of every node of our graphs. Each `Key` of the `Map` designates a node of our graph and the corresponding `Value` of the `Map` is its adjacency list. The choice to use a `Map` over a `List` is because unlike graphs where nodes labels are consecutive numbers, the stations of Canberra are not labeled consecutively. Therefore, using a `Map` makes it easier to look for an adjacency list of a specific station. This would not have been possible with a `List`.

- `private boolean weighted` indicates if the graph is weighted or not.

- `private boolean directed` indicates if the graph is directed or not.

Besides these attributes, `Graph.java` also has several methods whose most important ones are the following:

- `private void convertTxt(File stopsFile, File stopTimesFile, boolean weighted, boolean directed)`

- `private void addNodesFromTxt(File stopsFile)`

- `private void addEdgesFromTxt(File stopTimesFile)`

- `private void addWeightsFromTxt(File stopsFile)`

The first method is used to initialize a graph by calling the other three methods and in the meantime setting the attributes `private boolean weighted` and `private boolean directed`.

The other three methods are used to parse *stops.txt* and *stop_times.txt*, retrieve the data and create the nodes and edges of our graph.

2.2 Nodes

For *stops.txt*, each line was containing these informations in this order: stop_id, stop_name, stop_lat, stop_lon. In order to create our nodes, we had to use the stop_id as labels for our nodes. To do so, the method `private void addNodesFromTxt(File stopsFile)` has been created.

```
private void addNodesFromTxt(File stopsFile) throws FileNotFoundException {

        Scanner myReader = new Scanner(stopsFile);
        myReader.nextLine(); // skip headers line
        int stop_id;

        while (myReader.hasNextLine()) {

                // split the line
                String line = myReader.nextLine();
                String[] arr = line.split(",");

                // add the id of the node
                stop_id = Integer.parseInt(arr[0]);
                if (this.map.containsKey(stop_id)) {
                        System.out.println("Node in duplicate: " + stop_id);
                } else {
                        List<DirectedEdge> list = new ArrayList<DirectedEdge>();
        this.map.put(stop_id, list);
                }

        }
}
```

Figure 3: `private void addNodesFromTxt(File stopsFile)`

The method takes the *stops.txt* as input and reads every line as a `String` using `java.util.Scanner`. After splitting the `String` using a comma as separator, the method then retrieves the first element of the output array which corresponds to stop_id and creates a new `Entry` to the `Graph Map` using the stop_id as `Key` and an empty `List<DirectedEdge>` as `Value` only if the `Graph Map` does not already contain this `Key`

2.3 Edges

For *stop_times.txt*, the informations are disposed as below: trip_id, arrival_time, departure_time, stop_id, stop_sequence, timepoint. As above, the method `private void addEdgesFromTxt(stopTimesFile)` takes *stop_times.txt* as input and split each line into an array using commas as separators.

The interesting data in stop_times.txt is both trip_id and stop_id. If from a line to the other, the trip_id is the same, we know that both stations of each line are part of the same bus or subway line and are therefore linked together by and edge. As the method has to link each line to the previous one to check if the trip_ id is the same and to create the edge between the two stop_id, we store the data of the current iteration as well as of the previous iteration.

```
[...]
if (directed){
        // if we are still in the same trip, then the 2 stations are connected
        if (trip_id.equals(trip_id0)) {
                if (!this.map.containsKey(stop_id0)) {
                        System.out.println("Node does not exist: " + stop_id0);
                } else {
                        List<DirectedEdge> edgesList = this.map.get(stop_id0);
                        boolean exists = false;
                        for (int i = 0; i < edgesList.size(); i++) { // to avoid
                        ↪  duplicates
                                if (edgesList.get(i).to() == stop_id) {
                                        exists = true;
                                        break;
                                }
                        }
                        if (!exists) {
                                DirectedEdge newEdge = new DirectedEdge(stop_id0,
                                ↪  stop_id, 1);
                                edgesList.add(newEdge);
                        }
                }
        }
} else if (!directed){
        if (trip_id.equals(trip_id0)) {
                if (!this.map.containsKey(stop_id0)) {
                        System.out.println("Node does not exist: " + stop_id0);
                } else if (!this.map.containsKey(stop_id)) {
                        System.out.println("Node does not exist: " + stop_id);
                } else {
                        List<DirectedEdge> edgesList0 = this.map.get(stop_id0);
                        List<DirectedEdge> edgesList = this.map.get(stop_id);
                        boolean exists = false;
                        for (int i = 0; i < edgesList0.size(); i++) { // to avoid
                        ↪  duplicates
                                if (edgesList0.get(i).to() == stop_id) {
                                        exists = true;
                                        break;
                                }
                        }
                        if (!exists) {
                                DirectedEdge newEdge0 = new DirectedEdge(stop_id0,
                                ↪  stop_id, 1);
                                DirectedEdge newEdge = new DirectedEdge(stop_id,
                                ↪  stop_id0, 1);
                                edgesList.add(newEdge);
                                edgesList0.add(newEdge0);
                        }
                }
        }
}
[...]
```

Figure 4: `private void addEdgesFromTxt(File stopTimesFile)`

The method then checks if the nodes are actually existing and if they do, add the edge to the right adjacency list:

- only to the source node adjacency list if the graph is directed
- to both the source and destinations nodes adjacency lists if the graph is not directed

2.4 Weights

```
[...]
for (Map.Entry<Integer,List<DirectedEdge>> entry : this.map.entrySet()) {
        for (DirectedEdge edge : entry.getValue()) {
                source = edge.from();
                destination = edge.to();
                sourceLat = mapCoord.get(source).getLatitude();
                sourceLong = mapCoord.get(source).getLongitude();
                destinationLat = mapCoord.get(destination).getLatitude();
                destinationLong = mapCoord.get(destination).getLongitude();
                distance = Math.sqrt(Math.pow((destinationLat-sourceLat),2) +
                ↪  Math.pow((destinationLong-sourceLong),2));
                edge.setWeight(distance);
        }
}
[...]
```

Figure 5: `private void addWeightsFromTxt(File stopsFile)`

The method `private void addWeightsFromTxt(File stopsFile)` is used only if the graph is weighted. It takes *stops.txt* as input and extracts both stop_lat and stop_lon for each station to store it into a `Map` of `Coordinates` which is a `Class` that simply has stop_lat and stop_lon as attributes. Once the `Map` of `Coordinates` created, the method calculates the euclidean distance between starting node and destination node for every edge of the graph and set it as weights.

$$d(a,b) = \sqrt{(a_{lon} - b_{lon})^2 + (a_{lat} - b_{lat})^2},$$

where a is the starting node, b is the destination node

Figure 6: Euclidean distance formula in two dimensions

Here is an example of what this method would give for the R3 bus line of Canberra:

stop_id: source	stop_lat: source	stop_lon: source	stop_id: destination	stop_lat: destination	stop_lon: destination	distance
3353	-35.307607	149.189463	3467	-35.316348	149.190329	0.008783793998
3467	-35.316348	149.190329	3011	-35.297298	149.150267	0.04436063958
3011	-35.297298	14.150267	3419	-35.278322	149.12815	0.02914189879
3419	-35.278322	149.12815	5514	-35.240007	149.067995	0.07132084723
5514	-35.240007	149.067995	5502	-35.23861	149.063546	0.004663175956
5502	-35.23861	149.063546	940	-35.24	149.060282	0.003547646544
940	-35.24	149.060282	4072	-35.21243	149.0607	0.02757316855
4072	-35.21243	149.0607	4091	-35.194784	149.060626	0.01764615516
4091	-35.194784	149.060626	4807	-35.200702	149.068587	0.009919689763
4807	-35.200702	149.068587	X	X	X	X

Figure 7: Table of coordinates and distances for edges of the R3 bus line of Canberra

3 Breadth-First Search

The Breadth-First Search algorithm has been implemented in the `BFSSP.java`. Methods of this class have all been made `static` as it is not necessary to store any data from one breadth-first search to the other. An user would likely use a breadth-first search on the graph of his choice and it makes more sense to just create a graph instead of having to create both a graph and a `BFS` instance. The method `public static List<Integer> bfs(Graph G, int startingNode)` takes a graph and a starting node as parameters and starts a breadth-first search.

While doing a breadth-first search, we need to store for each node three values:

- `private boolean marked` to know if the node has been reached at some point during the breadth-first search
- `private int previous` to know which node preceded the node during the breadth-first search
- `private int distance` to know the distance separating the node to the starting node of the breadth-first search

Just as for `Map` of `Graph.java`, it is more intuitive to use a `Map` to store all of these values. One approach could be to use three different `Map`, however, to use less memory and make the code more efficient we decided to use a single `Map` whose entries would store a tuple of three elements. This tuple contains a `boolean`, and two `int` and results in the creation of the `class` `MPD.java`.

```
public static List<Integer> bfs(Graph G, int startingNode) {
        List<Integer> path = new ArrayList<Integer>();
        List<Integer> toVisitNodes = new ArrayList<Integer>();
        for (Integer key : G.getMap().keySet()) {
                mapMPD.put(key, new MPD(false, -1, 0.0));
        }
        toVisitNodes.add(startingNode);
        mapMPD.put(startingNode, new MPD(false, -1, 0.0));

        if (G.isWeighted()) {

                System.out.println("Please consider using Dijkstra Algorithm on
                ↪  weighted graph if you would like to find shortest paths.");

        }

                while (!toVisitNodes.isEmpty()) {

                        int currentNode = toVisitNodes.remove(0);
                        mapMPD.get(currentNode).marked = true;
                        path.add(currentNode);

                        for (DirectedEdge edge : G.getMap().get(currentNode)) {
                                if (!toVisitNodes.contains(edge.to()) &&
                                ↪  !mapMPD.get(edge.to()).marked) {
                                        toVisitNodes.add(edge.to());
                                        mapMPD.get(edge.to()).previous = currentNode;
                                        mapMPD.get(edge.to()).distance =
                                        ↪  mapMPD.get(currentNode).distance+1;
                                }
                        }
                }
        //}
        return path;
}
```

When doing the breadth-first search, the first thing to do is to check whether or not the graph given in parameter is weighted or not. If it is weighted, the method warns the user that the breadth-first search cannot be used on weighted graph to look for shortest paths. However, it stills runs the breadth-first search, as even though it cannot be used to find shortest paths, it could still be used to discover connected nodes on a weighted graph.

The breadth-first search works as follows:

- Adds starting node given in parameter to `toVisitNodes` which is the list of nodes the method is planning to visit
- Sets the first element of `toVisitNodes` as the current node and removes it from list ; sets its `MPD.marked` attribute to `true`
- For every neighbor of the current node, if the neighbor is not marked yet and if the neighbor is not contained in `toVisitNodes`
- Loops until `toVisitNodes` is empty, meaning there is not any remaining connected node that has yet to be visited

3.1 Shortest paths

Assuming a breadth-first search has already been launched for the starting node of the shortest path the user is looking for. The shortest path can then first be printed.

Before printing a path, it is necessary to make sure this path actually exists. In order to do so, the method `public static boolean hasPathTo(int destination)` has been implemented. It simply returns a boolean indicating if a path exists or not. It can be done easily as the `marked` attribute of `MPD` solely serves this purpose and just returning its value does the trick.

`public static printShortestPath(int startingNode, int destinationNode)` first checks if the path exists using `hasPathTo()`. If it does not, the code warns the user by printing a message. Another check done by the method is the egality of `startingNode` and `destinationNode`. If these nodes are the same, there is no need to continue as there is no loop in the graph.

Once these checks done, if a path exists and the starting node is distinct from the destination node, the starts loooking for the shortest path. It does so by browsing `MPD.previous` attribute starting from the destination node and going all the way back to the starting node. As the path is discovered from destination to start, the code builds the path backwards.