

---

# COMP333 Assignment 1 - Part 1 Report

## Group 20:

Shathakarun (Vipul) Manthena – 44890524

Jake Garth – 44887841

Sven Lees – 44645910

---

## 1. Stage 3

### 1.1 Algorithm Design

Given the description of Stage 3, the requirements for this stage are clearly understood. For the design, after finding the route, the intermediary stops included in the found route are stored in a 2D array. All the possible combinations of intermediary stops in the route are added to the returning array. Some of the longest routes are first computed by finding the stations with greatest difference in latitude and longitude.

### 1.2 Implementation of Code

To achieve the algorithm design, two methods i.e. 'computeRatio' and 'computeAllRatio' are used along with helper methods. The following is a description of the code:

#### 1.2.1 computeRatio

This method is fairly simple to understand. The method calculates the shortest route between two stations which is computed by routeMinStop() from stage 2. Also, it calculates the distance between two stations computed by the method computeDistance(). Storing the distance using routeMinStop in 'currentd' and the value of computeDistance() in 'newd' the ratio is calculated. The method returns a float by calculating 'currentd/newd'.

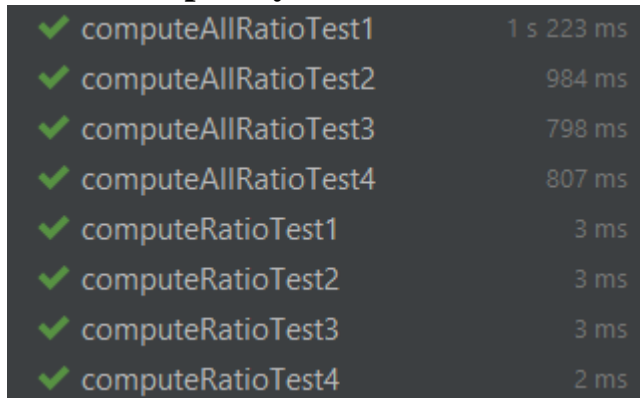
#### 1.2.2 computeAllRatio

This method uses the ratio returned by 'computeRatio' for all the station pairs in the rail network and returns a hash map containing the ratios.

The object we are returning is 'HashMap<String, HashMap<String, Double>> a = new HashMap<>();'. Initially we find the stations with the greatest differing distance by using their co-ordinates. We then ensure the arraylists begin with the opposing corner stations. This results in achieving a reduced computational redundancy by computing the longer routes first. Next, we implement a series of for loops and IF statements to achieve the methods purpose. We check if the current pair has already been computed, otherwise it skips to the next pair of stations.

It then calculates the ratio of the pair and stores it in the hash map. Finally, a HashMap containing all the ratios is returned.

### 1.3 Time Complexity



✓ computeAllRatioTest1	1 s 223 ms
✓ computeAllRatioTest2	984 ms
✓ computeAllRatioTest3	798 ms
✓ computeAllRatioTest4	807 ms
✓ computeRatioTest1	3 ms
✓ computeRatioTest2	3 ms
✓ computeRatioTest3	3 ms
✓ computeRatioTest4	2 ms

## 2. Stage 4

Please note that there is overlap between Jake Garth's report and this report due to using the same code.

### 2.1 Approach

Stage 4 "routeMinStopWithRoutes()" is an extension of "routeMinStop" from Stage 1 (using Jake Garth's submission). In order to find the list of stops between two Stations, a greedy approach was chosen for this stage. The reasoning is that the number of stops needs to be kept to a minimum, so a greedy approach is the obvious choice. Thus, Dijkstra's algorithm is used, and by default a BFS search is used. This route is then saved in an ArrayList which can then be used to find the lines that were travelled on. Once this list of stops has been found, the line on which a route is occurring can now be found. The way a line has been found is that it compares two adjacent Stations in the route and checks what lines they both share. The

line in which they share is then recorded and added in to the ArrayList. In the instance there are multiple lines shared by two adjacent stations, the line that is last recorded as being shared is used.

## **2.2 Algorithm Design**

1. Dijkstra's algorithm was used in `routeMinStops()` to find the route and stops between two Stations. This route was stored in an ArrayList.
2. This ArrayList is then inserted in to the "`LineList()`" method.
3. `LineList()` will check if there has been a Line already recorded in the ArrayList using the "`findRecentLine()`" method.
4. If a line has NOT already been recorded in the ArrayList, insert the line that the first two Stations are on at the start of the ArrayList. The "`findLine()`" method finds the line that two adjacent Stations share
5. If a line has already been recorded, iterate through the ArrayList till two adjacent stations are not on the most recently recorded line in the ArrayList. The "`findRecentLine()`" method will find the most recently recorded line in the ArrayList. This new line will be inserted in to the ArrayList.
6. Repeat steps 4 and 5 until the last station in the ArrayList is reached and then return this ArrayList.

## **2.3 Algorithm Design**

In this stage we extend the functionality of the '`routeMinStop()`' from stage 2, by specifying the which train line(s) are being used to travel between stations and noting any transfers taking place as well. The '`routeMinStopWithRoutes()`' method will incorporate the required functionality for stage 4. The following helper functions were used:

- a. `routeMinStop()`: The purpose of this method is to return the route with the minimum number of stops to go from one Station to another station. This `routeMinStop()` is the same used in Jake Garth's Stage 1 Report.
- b. `findLine()`: This returns the line that two stations share. If multiple lines are shared, it returns the last line confirmed.
- c. `findRecentLine()`: Is used to check the most recent line that has been recorded in the ArrayList.
- d. `LineList()`: Iterates through the ArrayList and inserts a new string outlining the line and direction of a route whenever `findLine()` is

different to findRecentLine(). I.e. Whenever a line has been swapped, it is inserted in to the ArrayList.

The next section will explain how all these method have been implemented to achieve the required functionality.

## 2.4 Implementation of Code

In this section we describe how 'routeMinStopWithRoutes()' has been implemented.

### 2.4.1 routeMinStopWithRoutes()

First, it checks if the origin and destination station are valid. If not, then an empty ArrayList is returned. Then, the reset() method is called. This just empties out all the global data used in the "routeMinStop()" method. "routeMinStop()" is then called and returns the ArrayList with the route that has the least stops. This ArrayList is then put in to the LineList() method which then adds all the lines used to the ArrayList. This new ArrayList is then returned.

```
public ArrayList<String> routeMinStopWithRoutes(String origin, String destination) {
    if(!stationList.containsKey(origin)||!stationList.containsKey(destination)){
        return new ArrayList<>();
    }
    reset(); //clears all the global data
    ArrayList<String> stops = routeMinStop(origin, destination); //Uses Jake Garth's routeMinStop from Stage 1,
    //This gives an ArrayList of the min stop route that will be taken
    ArrayList<String> first = new ArrayList<>();
    first.addAll(stops);
    first = LineList(first); //Apply the "LineList()" method to the original ArrayList. This will include the lines taken.
    return first;
}
```

### 2.4.2 LineList()

LineList() works by iterating through the entire ArrayList. At each station or line within the ArrayList, it checks three conditions to determine what to do:

1. If the current string is a line, go to next string in the ArrayList.
2. If the current string is on a new line compared to the most recently recorded line, insert a new string with the new line. Then, go to the next String in the ArrayList. The "findLine()" and "findRecentLine()" methods are used to check this.
3. If the current string is a station that is on the most recently recorded line, go to next String in the Arraylist.

### 2.4.3 findLine()

This method works by checking the station\_data.csv to determine what line two adjacent stations share. When this has been found, the direction needs to be found to determine the order in which the new string needs to be written. It checks the two stations to determine which one is closer to what side of a line. For example, if the stations “Dundas” and “Telopea” were in the ArrayList in that order, the string being inserted will need to be “T6 towards Carlingford from Clyde”, as Telopea is further away from Clyde than Dundas .

```
public String findLine(String stationName, String nextStation) {
    int size = firstRow.length - 1;
    String line = null;
    for (int i = 3; i < size; i++) {
        //lineStationData stores the lines on which a Station exists on
        //if two adjacent Stations run on the same line, this line can be used
        if (!lineStationData.get(stationName)[i].equals("") && !lineStationData.get(nextStation)[i].equals("")) {
            //line stores the line being used, by using the lineData global data structure
            //this "if" statement determines the direction in which the route is taking, based on how many stops away
            //the Stations are from the origin of the train line
            if (Integer.parseInt(lineStationData.get(stationName)[i]) < (Integer.parseInt(lineStationData.get(nextStation)[i]))) {
                line = lineData.get(firstRow[i])[1] + " towards " + lineData.get(firstRow[i])[3] + " from " + lineData.get(firstRow[i])[2];
            } else {
                line = lineData.get(firstRow[i])[1] + " towards " + lineData.get(firstRow[i])[2] + " from " + lineData.get(firstRow[i])[3];
            }
        }
    }
    return line;
}
```

### 2.4.4 findRecentLine()

Find recent line takes in the input ArrayList<String> and then returns the most recent String that is a Line (i.e. not a Station). As each string of a line contains the words “towards” and no stations contain the word “towards”, this method just found the most recent string with the word “towards” in it and returned it.

## 2.5 New Classes and Data Structures

Two new hash map’s and an array were used to store the data from the CSV files.

```
public HashMap<String, String[]> lineData = new HashMap<>();
public HashMap<String, String[]> lineStationData = new HashMap<>();
public String[] firstRow = new String[20];
```

“lineData” was used to store all the information from the lines\_data.csv. This information would then later be used to insert a string in to the ArrayList describing the lines the route travels on.

“firstRow” contains the first row from the station\_data.csv. This allowed a line to be paired with a non-empty cell in the CSV file.

“lineStationData” was used to store all the information from the station\_data.csv. This information was then used to determine what two adjacent stations shared a Line.

No new classes were used for Stage 4.