Algorithm Theory & Design COMP333

Assignment 1: Part 2

Report

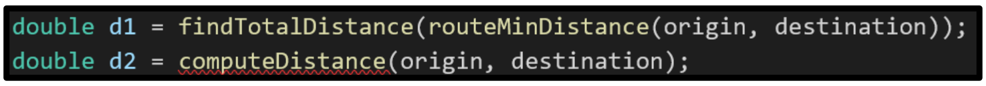
Bradley Kenny	45209723
John Kim	45163782
Mark Smith	45176655

Stage 3

Part 1: computeRatio()

Brute Force

This method started off as a simple division operation of *d1/d2*[Fig 1] but, we found that this was too expensive when we got up to the `computeAllRatio()` method. This was because *d1* would call `routeMinDistance()` every time the ratio must be filled in.



```
double d1 = findTotalDistance(routeMinDistance(origin, destination));
double d2 = computeDistance(origin, destination);
```

Figure 1. Rudimentary computeRatio()

Lookup Table

Realised that operating `routeMinDistance()` for every operation is too expensive and have implemented a lookup table. Further to this, we implemented a function that would generate a key based on the alphabetical ordering of the two strings. It then uses the lookup table such that it functions like if when looking for Chatswood -> Macquarie University, then look up if the combination name *Chatswood-Macquarie University* exists in the table, if not, compute `routeMinDistance()` for all the intermediate stations. That is: *Chatswood-Macquarie University, Chatswood-Macquarie Park, Chatswood-North Ryde*. Then when we look up Chatwood -> Beecroft, it computes *Chatswood-Beecroft, Chatswood-Cheltenham, Chatswood-Epping*, and since we've got distance ratio from *Chatswood-Macquarie University*, we stop the computing thereon. This was computed via using a helper method named `mapRatios()`.

It should be noted that upon the completion of `computeAllRatio`, we no longer needed this function for our implementation so we reverted it back to a simple division described previously for simplicity's sake.

Part 2: computeAllRatio()

Complete Brute Force

Originally, we attempted to do a simple brute force to get the answers before optimising. This was completed using a double FOR loop that iterates through all stations in *stationList*, adding other stations to a `HashMap<station i, <station j, computeRatio(I ,j)>>`. However, this resulted in ~35 seconds per test, and looked for any other ways to optimise the method.

Add(I, j) && Add(j, i)

When adding stations to a new `HashMap`, double down on adding stations, that is: when we compute `HashMap<station i, <station j, computeRatio(i, j)>>`, operate on its complement station (j), to compute `HashMap<station j, <station i, computeRatio(I, j)>>`. This then reduced the resulting time to ~15 seconds per test.

Lookup Table Integration

This lookup table method was using a dynamic programming approach where the code recognises that there are identical sub problems found within. Such as *Chatswood -> Beecroft* and *Chatswood -> Macquarie University* will shave a subproblem that is *Chatswood -> Macquarie University*[Fig 2]. Using the newly altered `computeRatio()`, resulting time was reduced down to ~4 seconds per test.

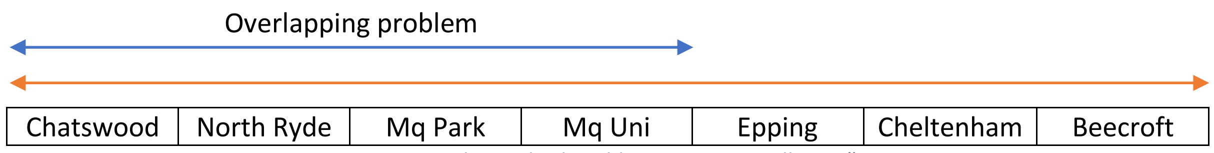


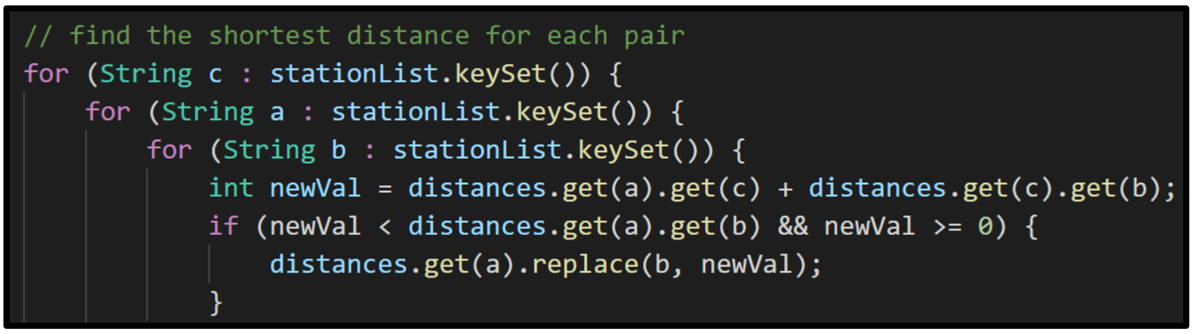
Figure 2. Identical subproblem in computeAllRatio()

Floyd-Warshall

Upon being stuck at ~4 seconds for the best result, we began researching for some hints that may lead to the solution. Results from the Internet suggested looking into “Floyd-Warshall”, “Bellman-Ford”, “Johnson’s” algorithms, and after inspecting their intended use, we’ve decided that Floyd-Warshall was the one we should be attempting to integrate.

Using GeeksForGeek’s algorithm template for Floyd-Marshall[1], we were able to produce a rudimentary solution[Fig 3]. It operates on the same basis of implementing O(n^2) for populating a `HashMap` with `Integer.MAX_VALUE`, much like `routeMinDist()` from Stage 1. This is implemented using a double FOR loop to check whether we have a pairing between two pairs. If there is, we add their pairing to the table. If not, we add a `MAX_VALUE` to be able to recognise there is no pairing.

After populating the table, we implemented the rest of the *Floyd-Warshall* method. This included a triple FOR loop that was able to find distances between each set of three items, if a new shortest distances was found, we would update the table with it. Originally, we would also use `computeRatio()` and add ratios to a lookup table. This resulted in ~2.3 seconds per test.



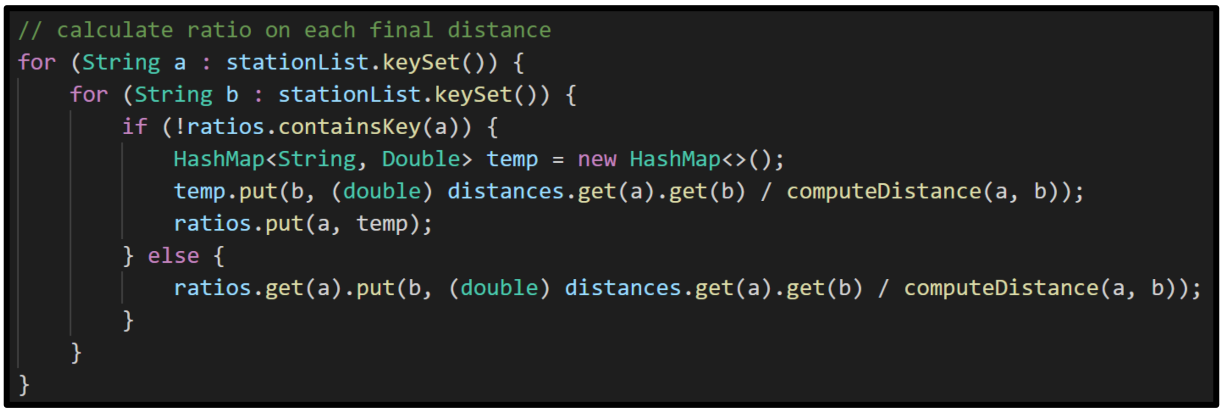
```
// find the shortest distance for each pair
for (String c : stationList.keySet()) {
    for (String a : stationList.keySet()) {
        for (String b : stationList.keySet()) {
            int newVal = distances.get(a).get(c) + distances.get(c).get(b);
            if (newVal < distances.get(a).get(b) && newVal >= 0) {
                distances.get(a).replace(b, newVal);
            }
```

Figure 3. Simple Floyd-Marshall

Final Solution

A minor fix from the previous stage has reduced from ~2.3 seconds to sub 1 second. Previously, we would calculate the ratio within each loop and then add it to a look up table. We modified this such that it will only calculate on each final distance [Fig 4].

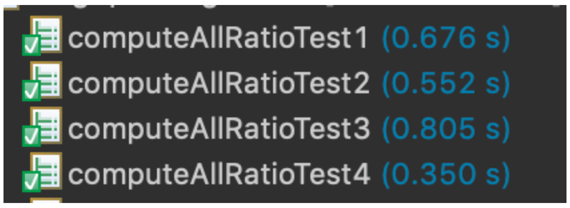
[1] https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/



```java
// calculate ratio on each final distance
for (String a : stationList.keySet()) {
    for (String b : stationList.keySet()) {
        if (!ratios.containsKey(a)) {
            HashMap<String, Double> temp = new HashMap<>();
            temp.put(b, (double) distances.get(a).get(b) / computeDistance(a, b));
            ratios.put(a, temp);
        } else {
            ratios.get(a).put(b, (double) distances.get(a).get(b) / computeDistance(a, b));
        }
    }
}
```

Figure 4. Final computeRatio() solution

This implementation meant that our tests would consistently perform under 1 second.

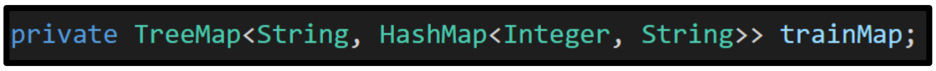


```
computeAllRatioTest1 (0.676 s)
computeAllRatioTest2 (0.552 s)
computeAllRatioTest3 (0.805 s)
computeAllRatioTest4 (0.350 s)
```

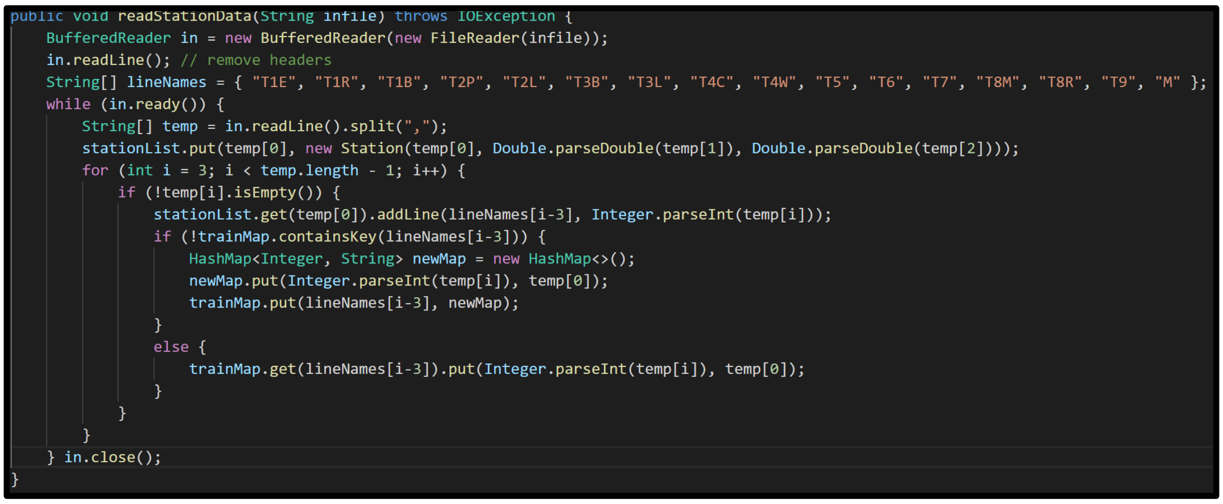
Stage 4

routeMinStopWithRoutes()

We started to implement this by utilising `routeMinStop()` from Stage 1, which was based off a *GeeksForGeek*'s BFS template algorithm[2]. Our line of thinking was to create a new data structure that takes in information from *lines_data.csv* and integrate the new line code, line name and station number in the corresponding station `Line` object. A `TreeMap<String, HashMap<Integer, String>>`[Fig 5] that corresponds to <*train line code*, <*station number*, *station name*>>.



```java
private TreeMap<String, HashMap<Integer, String>> trainMap;
```

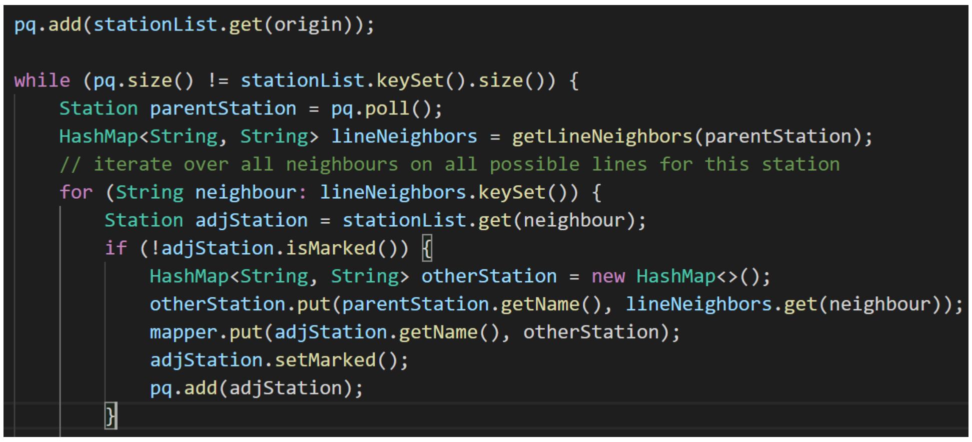


```java
public void readStationData(String infile) throws IOException {
    BufferedReader in = new BufferedReader(new FileReader(infile));
    in.readLine(); // remove headers
    String[] lineNames = { "T1E", "T1R", "T1B", "T2P", "T2L", "T3B", "T3L", "T4C", "T4W", "T5", "T6", "T7", "T8M", "T8R", "T9", "M" };
    while (in.ready()) {
        String[] temp = in.readLine().split(",");
        stationList.put(temp[0], new Station(temp[0], Double.parseDouble(temp[1]), Double.parseDouble(temp[2])));
        for (int i = 3; i < temp.length - 1; i++) {
            if (!temp[i].isEmpty()) {
                stationList.get(temp[0]).addLine(lineNames[i-3], Integer.parseInt(temp[i]));
                if (!trainMap.containsKey(lineNames[i-3])) {
                    HashMap<Integer, String> newMap = new HashMap<>();
                    newMap.put(Integer.parseInt(temp[i]), temp[0]);
                    trainMap.put(lineNames[i-3], newMap);
                }
                else {
                    trainMap.get(lineNames[i-3]).put(Integer.parseInt(temp[i]), temp[0]);
                }
            }
        }
    } in.close();
}
```

Figure 5. Data structure for lines_data.csv

Replacing *adjacentStations* from Stage 1 with all adjacent stations provided the context of the train line data. Simply put, get the *origin* Station and add to a *PriorityQueue*, and using `pq.poll()` and trawl through the list of train lines and if found, add adjacent stations to pq[Fig 6]. For example, If Chatswood = Origin, find all the station lines (T1E, T1R, T1B, T9 & M) and add adjacent stations to the pq, that is: Roseville, Artarmon, and North Ryde. Then using *pq.poll()* BFS through all the stations until the line reaches destination. From this point, we have information on all stops between those points, as well as the line that each station is using and the stop number of the station on that respective line.

[2] https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

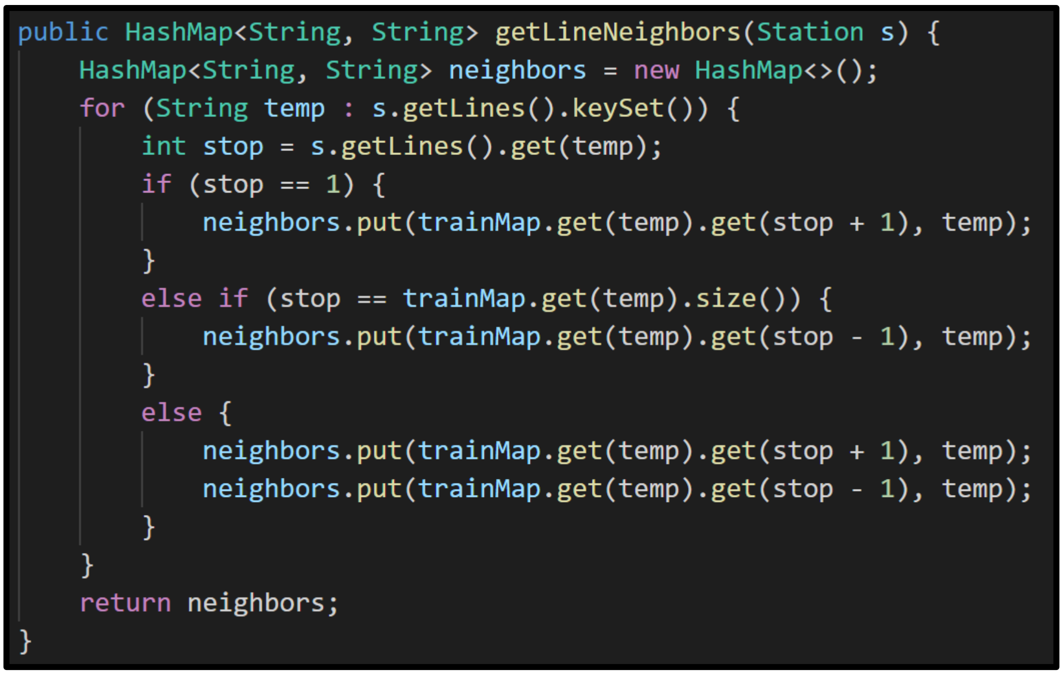


```
pq.add(stationList.get(origin));

while (pq.size() != stationList.keySet().size()) {
    Station parentStation = pq.poll();
    HashMap<String, String> lineNeighbors = getLineNeighbors(parentStation);
    // iterate over all neighbours on all possible lines for this station
    for (String neighbour: lineNeighbors.keySet()) {
        Station adjStation = stationList.get(neighbour);
        if (!adjStation.isMarked()) {
            HashMap<String, String> otherStation = new HashMap<>();
            otherStation.put(parentStation.getName(), lineNeighbors.get(neighbour));
            mapper.put(adjStation.getName(), otherStation);
            adjStation.setMarked();
            pq.add(adjStation);
        }
```

Figure 6. Implementation of GFG's BFS

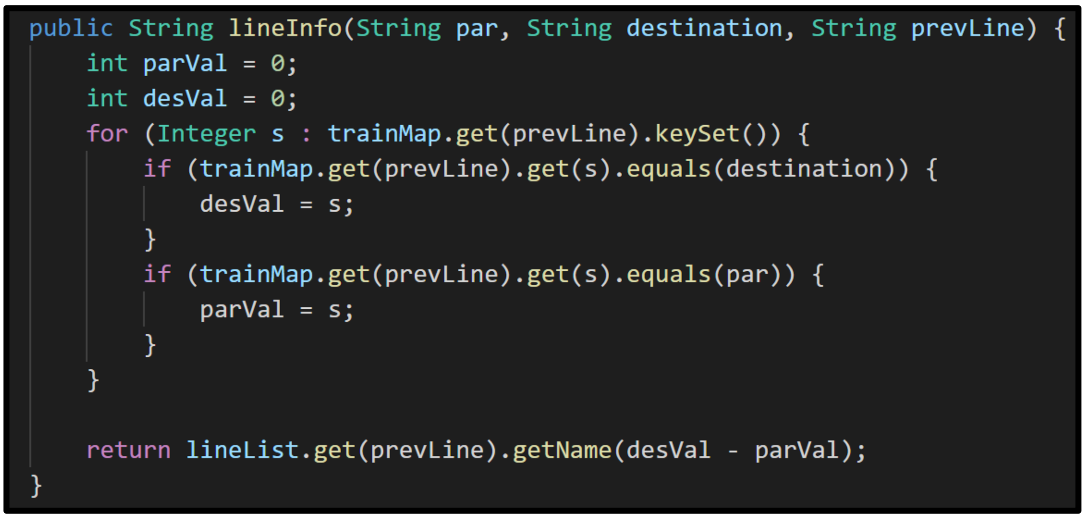
A helper method[Fig 7] that determines which are the neighbouring stations from each train line list. While iterating through all the possible stations, we need to keep track of which way it's going towards, and this can be judged using station number, which this method tries to achieve. It written to be used for creating an output for A -> Z or Z -> A; Hornsby towards Gordon or Gordon towards Hornsby.



```
public HashMap<String, String> getLineNeighbors(Station s) {
    HashMap<String, String> neighbors = new HashMap<>();
    for (String temp : s.getLines().keySet()) {
        int stop = s.getLines().get(temp);
        if (stop == 1) {
            neighbors.put(trainMap.get(temp).get(stop + 1), temp);
        }
        else if (stop == trainMap.get(temp).size()) {
            neighbors.put(trainMap.get(temp).get(stop - 1), temp);
        }
        else {
            neighbors.put(trainMap.get(temp).get(stop + 1), temp);
            neighbors.put(trainMap.get(temp).get(stop - 1), temp);
        }
    }
    return neighbors;
}
```

Figure 7. Neighbouring stations and direction determiner

Once the function reaches the destination, then it builds an ArrayList in reverse from Destination to Origin. One of the more difficult part of producing the output is the custom writing that signifies when trainline has been changed, for example, Beecroft -> Chatswood: T9 to Epping, from Epping, change to Metro to Chatswood. This method[Fig 8] looks at the station's previous line, if the number successfully increases/decreases by 1, then they're still on the same train line. Otherwise, it has changed train line and update the station number.



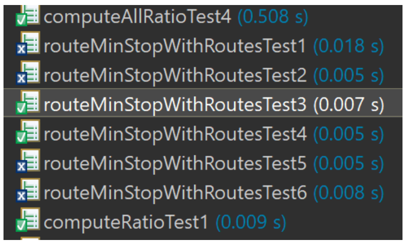
```java
public String lineInfo(String par, String destination, String prevLine) {
    int parVal = 0;
    int desVal = 0;
    for (Integer s : trainMap.get(prevLine).keySet()) {
        if (trainMap.get(prevLine).get(s).equals(destination)) {
            desVal = s;
        }
        if (trainMap.get(prevLine).get(s).equals(par)) {
            parVal = s;
        }
    }

    return lineList.get(prevLine).getName(desVal - parVal);
}
```

Figure 8. Get trainline name associated to the stop

Successful Tests

Having written tests[Fig 10] of our own, we were able to produce some successful results[Fig 9].

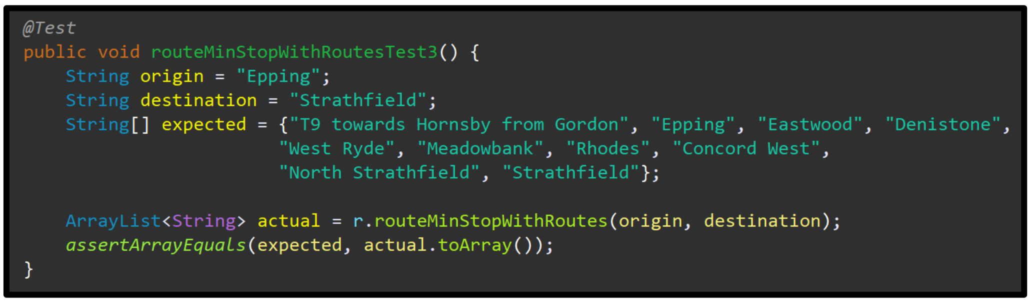


```
✓ computeAllRatioTest4 (0.508 s)
✗ routeMinStopWithRoutesTest1 (0.018 s)
✗ routeMinStopWithRoutesTest2 (0.005 s)
✓ routeMinStopWithRoutesTest3 (0.007 s)
✓ routeMinStopWithRoutesTest4 (0.005 s)
✗ routeMinStopWithRoutesTest5 (0.005 s)
✗ routeMinStopWithRoutesTest6 (0.008 s)
✓ computeRatioTest1 (0.009 s)
```

Figure 9. Successful tests results

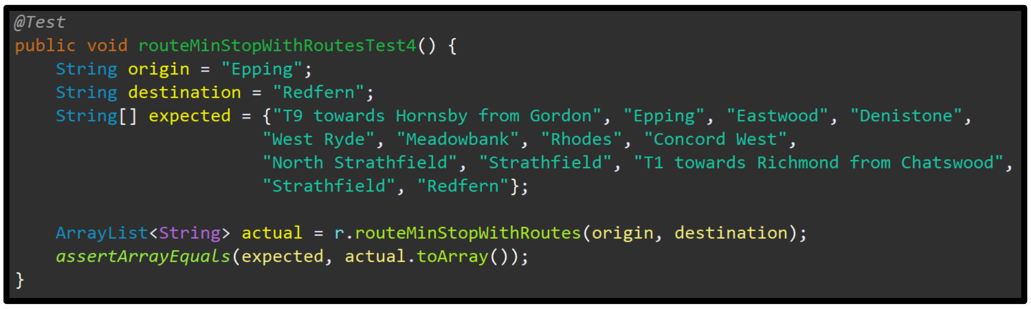
Test 3 Output: [T9 towards Hornsby from Gordon, Epping, Eastwood, Denistone, West Ryde, Meadowbank, Rhodes, Concord West, North Strathfield, Strathfield]

Test 3 Actual: arrays first differed at element [0]; expected:<T9 towards [Gordon from Hornsby]> but was:<T9 towards [Hornsby from Gordon]>



```java
@Test
public void routeMinStopWithRoutesTest3() {
    String origin = "Epping";
    String destination = "Strathfield";
    String[] expected = {"T9 towards Hornsby from Gordon", "Epping", "Eastwood", "Denistone",
                         "West Ryde", "Meadowbank", "Rhodes", "Concord West",
                         "North Strathfield", "Strathfield"};

    ArrayList<String> actual = r.routeMinStopWithRoutes(origin, destination);
    assertArrayEquals(expected, actual.toArray());
}
```



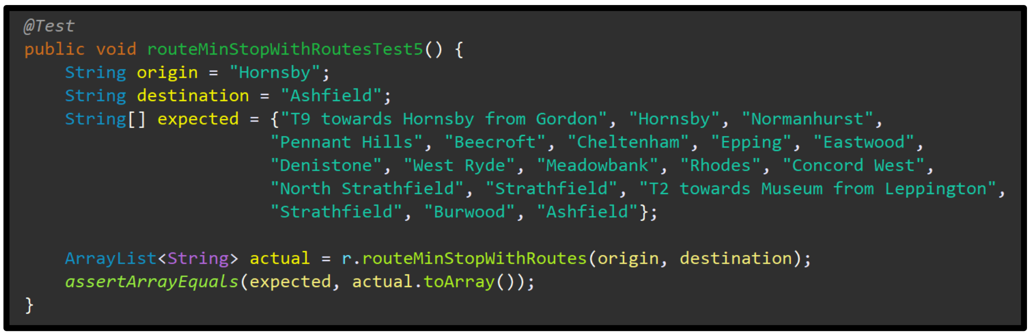
```java
@Test
public void routeMinStopWithRoutesTest4() {
    String origin = "Epping";
    String destination = "Redfern";
    String[] expected = {"T9 towards Hornsby from Gordon", "Epping", "Eastwood", "Denistone",
                         "West Ryde", "Meadowbank", "Rhodes", "Concord West",
                         "North Strathfield", "Strathfield", "T1 towards Richmond from Chatswood",
                         "Strathfield", "Redfern"};

    ArrayList<String> actual = r.routeMinStopWithRoutes(origin, destination);
    assertArrayEquals(expected, actual.toArray());
}
```

Figure 10. 2 of 6 tests created to check for its success

Correctly Identifying “Express” Lines

Test 5[Fig 11] **Output:** [T9 towards Hornsby from Gordon, Hornsby, Normanhurst, Thornleigh, Pennant Hills, Beecroft, Cheltenham, Epping, Eastwood, Denistone, West Ryde, Meadowbank, Rhodes, Concord West, North Strathfield, Strathfield, Burwood, T2 towards Leppington from Museum, Burwood, Ashfield]



```java
@Test
public void routeMinStopWithRoutesTest5() {
    String origin = "Hornsby";
    String destination = "Ashfield";
    String[] expected = {"T9 towards Hornsby from Gordon", "Hornsby", "Normanhurst",
                         "Pennant Hills", "Beecroft", "Cheltenham", "Epping", "Eastwood",
                         "Denistone", "West Ryde", "Meadowbank", "Rhodes", "Concord West",
                         "North Strathfield", "Strathfield", "T2 towards Museum from Leppington",
                         "Strathfield", "Burwood", "Ashfield"};

    ArrayList<String> actual = r.routeMinStopWithRoutes(origin, destination);
    assertArrayEquals(expected, actual.toArray());
}
```

Figure 11. Test 5 which tests express line T2L

This test correctly identifies the differences between the train line T2[Fig 12] which by default traverses Strathfield -> Burwood -> Croydon -> Ashfield. However, taking T2L will skip Croydon and make an express trip from Burwood, straight to Ashfield.

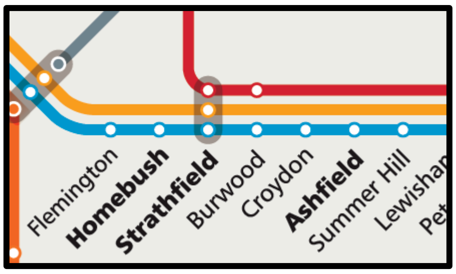


Figure 12. T2 train line

This test is significant as it successfully differentiates itself from routeMinStop() from Stage 1 and also successfully BFS through different train line codes (T2P vs T2L).

Unsuccessful Tests

As it can be seen below, our code incorrectly identifies which direction the train is moving towards. We believe that this is due to creating a path ArrayList backwards (destination to origin) and it is adversely affecting it's navigation. However, if we were to smooth out this operation, we would have been able to successfully produce correct output.

Test 1 Output: [T9 towards Hornsby from Gordon, Beecroft, Cheltenham, Epping, Metro towards Epping from Chatswood, Epping, Macquarie University, Macquarie Park, North Ryde, Chatswood]

Test 1 Actual: arrays first differed at element [0]; expected:<T9 towards [Gordon from Hornsby]> but was:<T9 towards [Hornsby from Gordon]>

Test 6 Output: [T8 towards Macarthur from Town Hall, Macarthur, Campbelltown, Leumeah, Minto, Ingleburn, Macquarie Fields, Glenfield, T5 towards Leppington from Schofields, Glenfield, Casula, Liverpool]

Test 6 Actual: arrays first differed at element [0]; expected:<T8 towards [Town Hall from Macarthur]> but was:<T8 towards [Macarthur from Town Hall]>