Algorithm Theory & Design  
COMP333

# Assignment 1: Part 2 Report

# Bradley Kenny | 45209723 John Kim | 45163782 Mark Smith | 45176655

# Stage 3

> A discussion on the general approach of the algorithm that you are using (exhaustive search,

greedy, dynamic programming).

> An explanation on how the algorithm works, including its name if it is an existing algorithm

(use an example if this helps explain your code).

> An explanation for parts of your code that may help in understanding your implementation

(do not write the entirety of your code without any explanation!).

## Part 1: computeRatio()

**Brute Force**  
This method started off as a simple division operation of *d1/d2* 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);

**Lookup Table**Realised that operating *routeMinDistance()* for every operation is too expensive and have implemented a lookup table, which 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()*.

## Part 2: computeAllRatio()

**Complete Brute Force**  
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 has 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*. Using the newly altered *computeRatio()*, resulting time was reduced down to ~4 seconds per test.

Overlapping problem

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Chatswood | North Ryde | Mq Park | Mq Uni | Epping | Cheltenham | Beecroft |

**Floyd Warshall**

Upon being stuck at ~4 seconds for the best result, we’ve tried to Google for some hints that may lead to the solution. Results from Google 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]](#footnote-1), we were able to get a rudimentary solution. It operates on the same basis of implementing O(n^2) for populating a HashMap with *Integer.MAX\_VALUE*,

**~2.3s**

**Final Solution**

# Stage 4

## routeMinStopWithRoutes()

> An explanation of any classes or data structures that you added to the project

and how you use them in the algorithm.

Read lines data and integrate it to Station

If

<https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>

# Individual Group Work Responsibilities

## Brad

## John

## Mark

1. <https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/> [↑](#footnote-ref-1)