COMP0005 Algorithms

London Railway Network API

Cheng Loo

Overview

This report discusses the data structures and algorithms that have been chosen to implement for this coursework, and the computational complexity of each choice. For the sake of clarity and due to limited space, data structures and complexity in each portion will be labelled as **DS** and **CC** respectively.

The Railway Network API

Main DS: Weighted Undirected Graph
The choice for using an undirected graph as the main DS is simple. The stations and bi-directional paths between the stations fit perfectly as vertices and edges respectively, and the functions to calculate minimum stops and minimum distance can be implemented quite efficiently with a graph.

1 Loading of Stations and Lines

DS: Dictionaries
The CSV files are loaded into 2 separate dictionaries. One stores the station names as keys and station objects (containing the coordinates) as values, while the other places all the stations as values into the corresponding lines as keys. This is due to the need to constantly obtain information about the stations, and dictionaries provide a cheap and efficient solution, giving only a $O(1)$ time complexity on average to get a value using a key.

CC (Theoretical): $O(N)$
The file 'londonstations.csv' contains the name and coordinates of every station, and every station has to be iterated through to be added into the dictionaries. Similarly, for the file 'londonrailwaylines.csv', adjacent stations on the same line are listed, which is proportional to the number of stations on the network, and once again every entry has to be iterated. Hence, the CC for this function is $O(N)$ where N represents the number of stations.

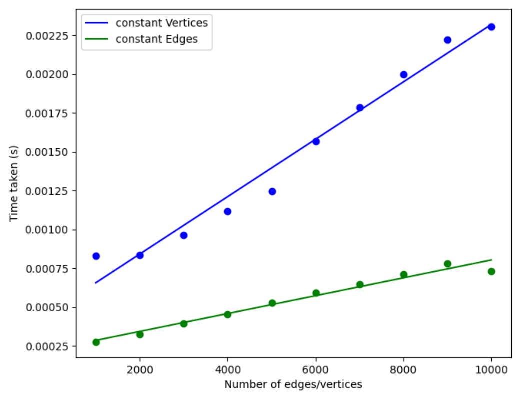
2 Minimum number of stops between 2 stations

Aim: Find the minimum number of stops to be travelled through regardless of distance

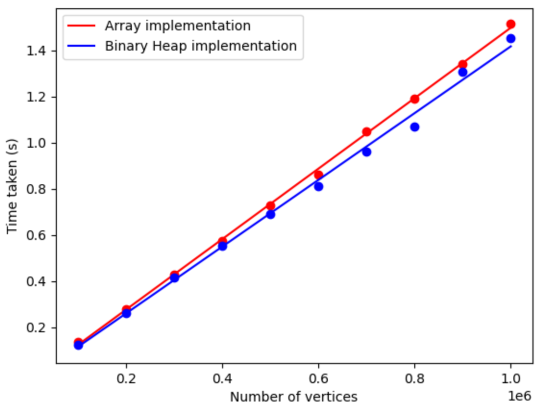
Algorithm: Breadth-First Search (BFS)
BFS is an algorithm that finds the shortest path (smallest number of vertices traversed) from a starting vertex to all other vertices on an unweighted graph. This is very fitting for this function as the distance between the stations (weights on graph) can be ignored completely.

CC (Theoretical): Linear $O(E + V)$ where E, V represents number of edges and vertices
The main bulk of the cost comes at the point when a vertex is added to the queue, and the adjacency list of that vertex has to be iterated through to choose the next vertex (i.e. time taken scales proportionally to the number of connected vertices).

CC (Experimental): Linear $\sim O(E + V)$
To test the theoretical CC, 2 separate lines were plotted as seen on Figure 1a. One of them is plotted by varying the number of vertices and keeping the number of edges constant, and vice versa for the other



(a) BFS time analysis



(b) Dijkstra's time analysis

Figure 1: Shortest Path algorithms tests

one. This shows that the time taken scales somewhat proportionally to the number of edges and vertices, and approximately has a time complexity of $O(E + V)$.

3 Minimum distance travelled between 2 stations

Aim: Find the minimum distance traversed between 2 stations going from station to station.

Algorithm: Dijkstra's Algorithm
The main deliberation for this function is between Edge-Weighted DAG, Dijkstra's algorithm, and the Bellman-Ford algorithm. By comparison, Edge-Weighted DAG has the lowest time complexity at $O(E+V)$, however it must be implemented in a graph that does not contain cycles, hence it cannot be used. As there are no negative weights in the graph, the better solution will be Dijkstra's as its CC is $O(ElogV)$ in both average and worst cases while Bellman-Ford has a CC of $O(EV)$ in both cases.

DS: Binary Heap Min-Oriented Priority Queue
After choosing the algorithm, the priority queue accompanied becomes the bottleneck. Between an unordered array and a binary heap, binary heap runs more efficiently on sparse graphs. As shown on Figure 1b, tests are carried out on both the array and binary heap implementation, with the binary heap coming out slightly ahead in terms of efficiency, and the gap seems to only widen as the number of vertices increases, hence binary heap is chosen.

CC (Theoretical): Linearithmic $O(ElogV)$
The cost of this function is dominated by the functions within the priority queue, namely the insert, decreaseKey, and deleteMin functions. These take $O(logV)$ to execute and it increases to $O(ElogV)$ as every edge in the queue has to be relaxed.

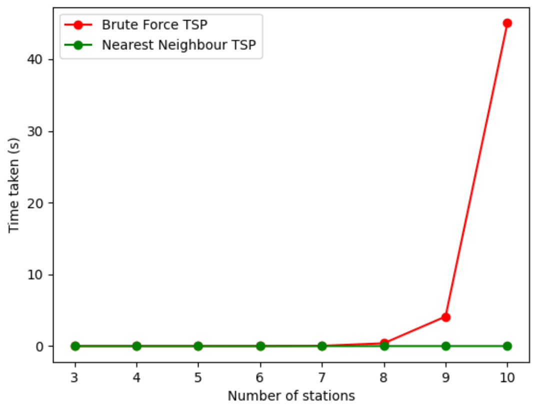
CC (Experimental): Linear $O(V)$
In Figure 1b, the graph of time taken against the number of vertices is plotted and while the line of best fit suggests that it grows in linear time. When large numbers of vertices are used, it may eventually grow into a linearithmic curve. However, for the sake of this test, it shall be considered to have a linear CC ($O(V)$).

4 Forming a new railway line by joining stations with the least amount of tracks

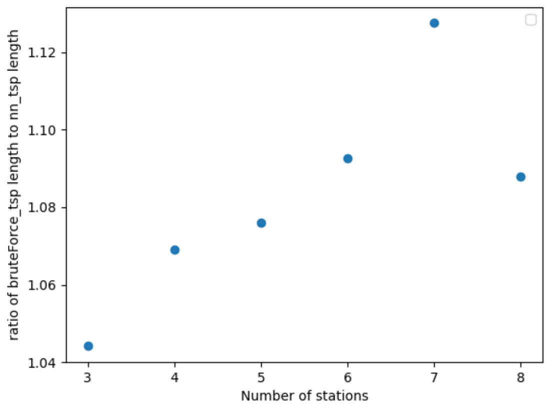
This particular function is similar to the Travelling Salesman Problem (TSP). TSP is a notable computational problem that poses the question: "Given a set of cities and that the distance from one city to another is not negligible, what is the shortest possible path that visits each city exactly once and returns to the starting city?”

Brute Force Algorithm
This API treats this function the way TSP is tackled, with the exception that it does not have to loop back to the station it started from. However, being an NP-hard problem, an optimal algorithm that works on large numbers of stations (cities) does not exist yet. There is, though, a brute force (BF) algorithm that calculates the minimum distance of all possible permutations of the routes, and selects the best route. This method however grows at a time of $O(N!)$, hence is only sustainable for up to about 20 stations, after which the computational cost is too heavy to handle.

Nearest Neighbour Algorithm
A possible solution that greatly lowers the cost of the operation, but may not return the optimal route, is the nearest neighbour (NN) algorithm. This algorithm simply finds the station that has the minimum distance from the starting station, adds it to the route, sets it as the starting stations, and repeats until all of the stations are added. This grows at a time complexity of $O(N^2)$, which is cheaper than the BF algorithm, but with the drawback that the route produced may be too far off from the optimal.



(a) Brute Force vs NN



(b) NN best route vs optimal route

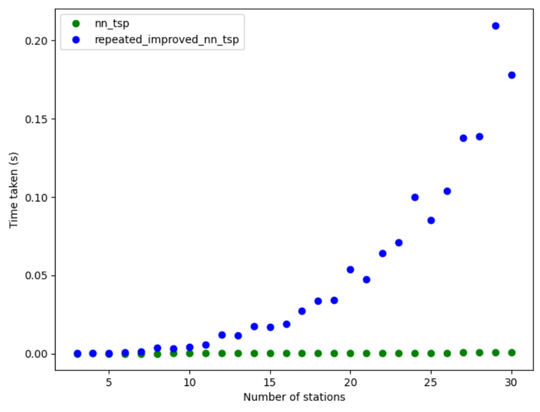
Figure 2: Nearest Neighbour algorithm tests

Experimentally, as shown on Figure 2a, the time taken for the BF algorithm grows at a much faster rate than the NN algorithm, which is seemingly instantaneous compared to BF (at 10 stations, time taken for NN is about 10^5 times faster). However, the problem of using the NN algorithm clearly shows in Figure 2b. When the length of the best route produced from using NN is compared to the length of the optimal route, the NN route at 7 stations is about 12% longer than the optimal route, and the trend suggests that the difference will increase with the number of stations.

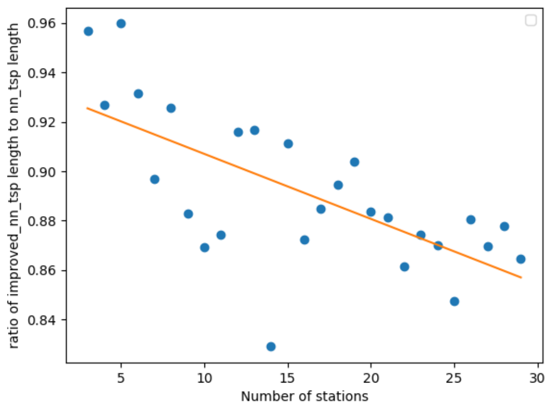
Heuristics
To improve NN to find the best route, there are 2 heuristics that are implemented: repetition and reversing segments. Repetition is simply executing NN by changing the starting station each time to find the best route. Reversing segments involves reversing the order of the stations in a particular route (e.g. route = [1,2,3,4,5]. By reversing [3,4,5], route = [1,2,5,4,3]) and check if it produces a shorter route. It is done on all segments of a route ([1,2], [2,3,4], and [2,3,4,5], etc) and returns the shortest possible route.

After implementing the above heuristics, tests were carried out as shown in Figure 3a and 3b. In Figure 3a, while the time complexity for improved NN seems to be exponential, it still grows at a much slower rate than BF (at 10 stations, BF took almost 50s while improved NN took ∼0.01s). Improved NN does fare much better than base NN in finding the shortest route, as shown on Figure 3b, and it seems to work even better as the number of stations grows (working ∼15% better than base NN on 30 stations). This means that it is likely that improved NN has an optimal range of stations at which it produces a relatively good route, while still being computationally feasible.

Algorithm Chosen: Brute Force + Improved Nearest Neighbour



(a) Improved NN vs Base NN



(b) Improved NN best route to Base NN best route

Figure 3: Nearest Neighbour heuristics tests

No. of stations <= 7: Use Brute Force

When there are 7 or fewer stations, BF algorithm is able to perform the function in $< 1s$. While the improved NN is also able to produce optimal routes, there are certain outlying results that may cause the algorithm to fail. Hence, to ensure that the optimal is produced and as it is also within reasonable execution time, the BF algorithm is chosen.

No. of stations > 7: Use improved Nearest Neighbour

The execution time of BF algorithm is now $> 1s$, and only increases at $O(N!)$. Improved NN, while generating sub-optimal routes, can be executed at a much lower cost and can support up to hundreds of stations. Given the context that this API is for the London Underground Network, in which the function to create a new railway line, if performed, should be executed on a set of 50 - 60 stations at most (current max number of stations for a line stands at 60). If such reasoning is to be followed, then it will make sense for improved NN to be used.

CC (Theoretical): $O(N^3)$

The CC of the base NN algorithm is $O(N^2)$. The repetition heuristic iterates through all the points again to change the starting point, hence $O(N^3)$.

CC (Experimental): $\sim O(N^3)$

According to Figure 4, the algorithm grows upwards in a quadratic/cubic like manner, however it stills grows much faster than the base NN algorithm, hence suggesting that it tends towards $O(N^3)$.

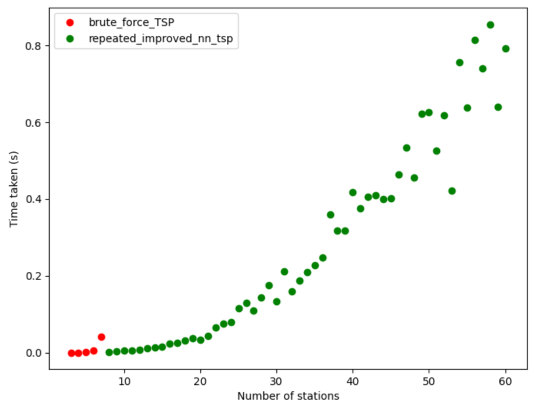


Figure 4: Time analysis of final algorithm