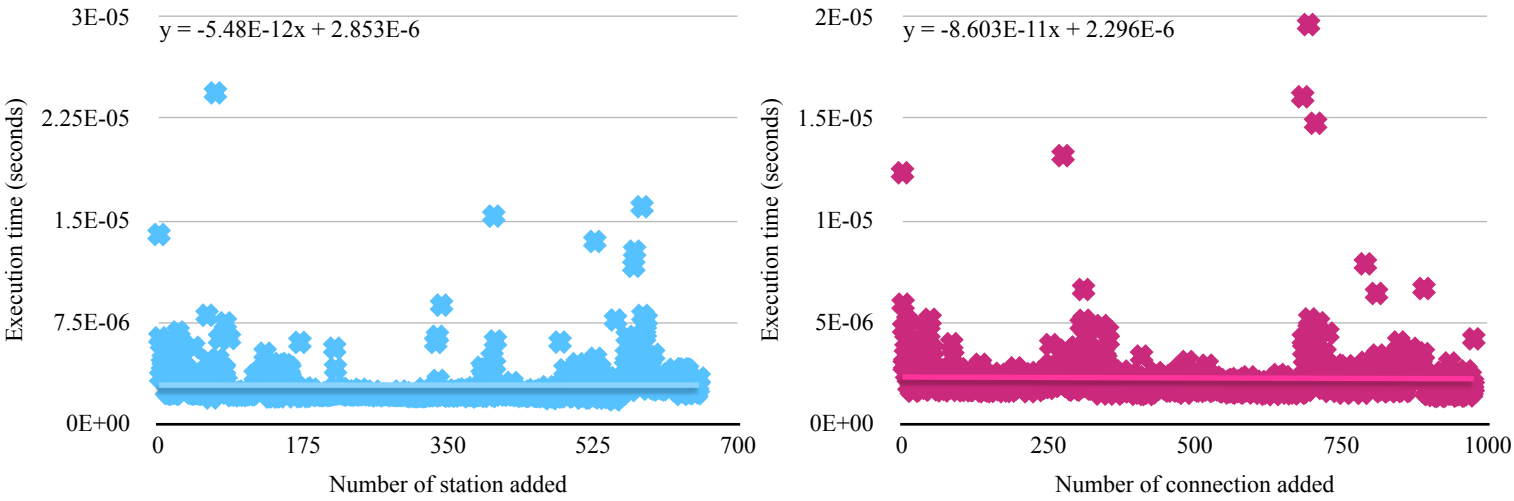
Analysing the effectiveness of employing various data structures and algorithms to reduce graph processing complexity in the context of the London Railway Network

Modelling the London Railway Network

The London Railway Network is made up of around 650 stations and approximately 30 lines. The requested API is focused on finding and optimising connections between these stations. An undirected weighted graph is is the most suitable data structure to help model this network. This is because adding and looking up each station and connection is a constant time operation with time complexity O(1). Removing a connection from the graph is also an operation of O(1). The following definitions hold throughout this document: *V - stations, E - lines connecting stations and ~ - approximately equal.* An adjacency list will be used to construct the graph data structure as opposed to a matrix as |E| ~ |V|. This means the graph will be sparse and so an adjacency list implementation will be faster and use less space than a matrix.



The plots above were generated by measuring execution time in seconds for each and every station and connection added. They both show a constant time complexity O(1), as the gradient of both lines is approximately 0.

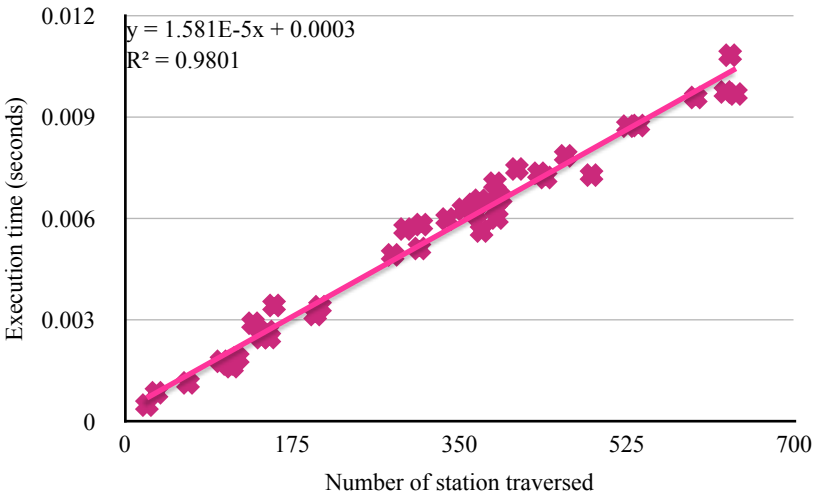
Minimum distance between two stations

The API also requests the program to be able to calculate the minimum distance in miles through the railway network between two input stations given their latitude and longitude coordinates. There are two algorithms for computing this: Dijkstra's and A* search. Both were implemented, tested and evaluated before finalising Dijkstra's for the API.

Dijkstra's first sets the distance of the start station to 0 and all other stations to positive infinity then also marks all stations as unvisited. This has time complexity O(|V|) as a dictionary is being used which has insertion time complexity O(1). The start station and its distance from the start station in miles (calculated using the haversine formula) are pushed to a priority queue. The following steps are repeated until the current station is equal to the target station. The algorithm updates the distances of the connected stations and pushes it to the priority queue if the current station distance + distance to connected station < connected station distance. Assume in the worst case that all stations are to be added to the priority queue, then in this case the time complexity is O(log |V|) for pushing and popping stations. This is repeated for all unvisited connections of the current station. In the worst case that each station is connected to all other stations the time complexity is O(|E|). Finally the current station is popped off the priority queue and the new current station is set by popping the minimum distance of the priority queue. The overall worst case time complexity is therefore O(|E| log |V|) + O(|V|) ~ O(|V| + E log |V|), where *O(|E| log |V|) - time complexity for pushing and popping all stations to the priority queue at most once in the worst case and O(|V|) - time complexity for initialising distances and marking stations as unvisited*.

The A* search algorithm was implemented similarly to Dijkstra's except the heuristic cost was added to the priority queue instead. This algorithm will give almost an 3-fold reduction in stations traversed as the heuristic function *f(C, T) = e(C, T) + o(S, C, T)* where S - start station, *C - current station and T - target station* gives priority to stations that are more likely to lead to the target station. *e(C, T)* calculates the distance in miles between the current

and target station and *o(S, C, T)* penalises stations that are far off the line formed from by joining the start station to the target station. However A* was not used in the final API as in some cases the heuristic would over estimate and provide sub-optimal solutions and so Dijkstra's was settled on as optimality was preferred.



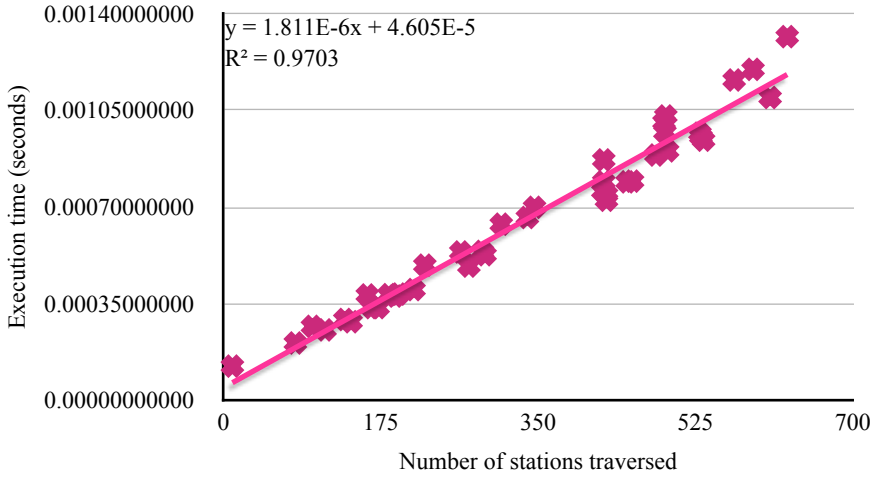
The plot to the left was constructed by measuring execution time against number of stations traversed during dijkstras algorithm. A 10 fold average was taken for execution time. The plot supports the time complexity of Dijkstra's is O(E log V). Looking at a few plots helps support the claim, for example when the number of stations traversed, i.e. V doubled from 175 to 350 then the execution time also doubled from just under 0.003 seconds to just under 0.006 seconds. The R^2 value is also close to 1 suggesting a strong linear correlation.

Minimum stops between two stations

The API also requests the program to be able to calculate the minimum number of stops through the railway network between two input stations. A breadth first search was used for this.

The algorithm first sets the previous to -1, number of stops to 0 and visited to False for all stations. This has time complexity O(|V|) as a dictionary is being used which has insertion time complexity O(1). The start station is marked as visited and then enqueued to a queue, this has time complexity of O(1) as popping off the dictionary and adding to a queue are O(1) operations. The following code is repeated until the queue is empty. The algorithm visits all unvisited stations connected to the current station and marks each of them as visited, increments the number of stops traversed and sets their previous stations to the current station. This takes O(E) as the connected stations at the end of all edges has to be checked. The overall time complexity is therefore O(E) + O(V) ~ O(E + V) *where O(E) ~ time complexity for searching all connected stations and O(V) ~ time complexity for initialising previous stops, number of stops and visited for all stations.*

The plot to the right was constructed by measuring execution time against number of stations traversed during BFS. A 10 fold average was taken for execution time. The plot to the right supports the time complexity of O(E + V). The trend line is linear and this is supported by the R^2 value of 0.9703 which is quite close to 1. Furthermore, when the number of stations traversed doubles from 175 to 350, the execution time also doubles from 0.00035 to almost 0.0007 seconds. This proves the time complexity is linear.



New Railway Line

The API also requests the program to be able to calculate the minimum distance in miles through an input set of stations without using any existing lines. This is a variation of the existing Travelling Salesman Problem (TSP) which is an NP - Complete problem meaning there is no polynomial time solution. There are many approximation based solutions to this including ant colony optimisation, 2 - OPT and Christofides algorithm. Ant colony optimisation is an algorithm for finding optimal paths by imitating the behaviour of ants searching for food and how ants are more likely to take paths that have been marked. This solution was not implemented due to time constraints despite research showing it to be an effective solution. 2 - OPT is an approximation algorithm that wasn't chosen to be

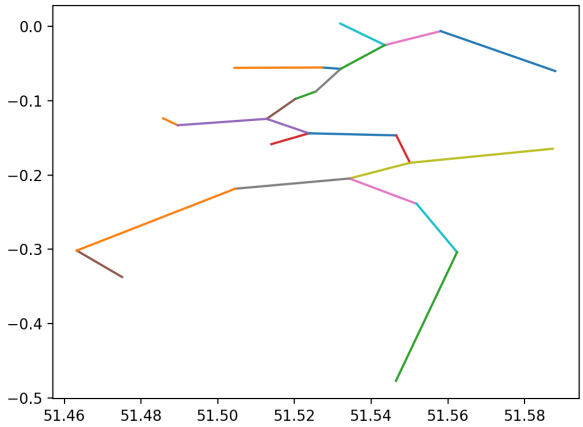
implemented in the final API as in the worst case it is 2 times the optimal solution. Christofides algorithm was finalised for the API as it will provide a 3/2 - OPT approximation to the new railway line algorithm.

Stage 1 - Graph

The first stage of the Christofides algorithm is to create a graph of all the station names in the input set. This is done by cloning stations from an initial graph which has the names, latitude and longitudes of all stations from the files. The time complexity for this is O(|V|) this is because a dictionary is being used to add stations to a graph and this has an insertion time complexity of O(1).

Stage 2 - Minimum spanning tree

The next stage of the Christofides algorithm is to create a minimum spanning tree of all the stations in the graph. Kruskal's algorithm was used for this as it has a time complexity of O(E log V). Kruskal's algorithm first sorts all the edges in the initial graph using pythons sorted function which has a time complexity of O(N log N) in the worst case where *N - number of items to sort*. Next, a Union-Find data structure is used to hold all sub trees that will form the minimum spanning tree. The following steps are repeated for all edges in the graph. The find operation is first used to ensure that two stations of an edge are not in the same tree, i.e. they don't have the same root. The find operation has time complexity O(log V) since the path compression optimisation is being applied in the Union-Find data structure. This is to make sure the edge does not form any cycles and no intermediate nodes are traversed again. Next, if the two subtrees i.e. the two stations connected to an edge do not share the same root then they are added to the minimum spanning tree and the union operation is applied to both trees. The union operation has time complexity O(log V) since the union by rank optimisation has been applied here so that the smaller depth tree is always attached under the root of the deeper tree. This stage produces an accurate minimum spanning tree as can be seen to the right.



Stage 3 - Find stations with an odd number of connections

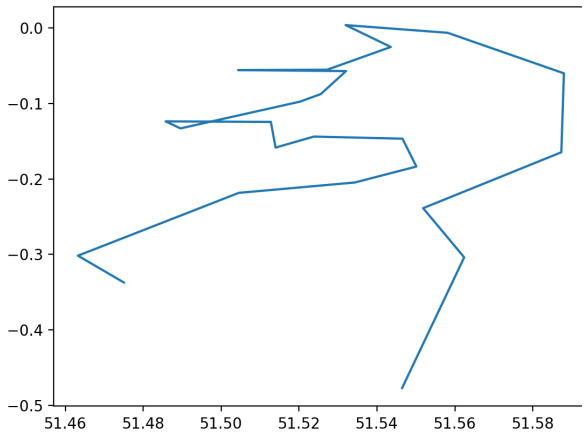
In this stage of the algorithm, a dictionary for storing the number of connections to each station is firstly created. For each edge of the tree, the connections count for each station is incremented by one. The time complexity for this is O(E) as looking up and insertion of dictionaries is O(1).

Stage 4 - Minimum distance matching

This stage of the algorithm involves pairing all the stations with an odd number of connections in such a way that the distance between both stations is as minimal as possible. There are many algorithms capable of doing this including the Blossom algorithm by Edmonds which has time complexity O(|E||V|^2). This algorithm will not be discussed however this was an impractical solution due to the time constraint for implementation. The minimum weight matching algorithm implemented in the API first loops through all odd vertexes until they have all been matched. Next, all odd vertexes are compared to each other and the pairing with the lowest minimum distance is added as an edge to the minimum spanning tree. This has time complexity O(|V|^2). Next the stations that form a pair are removed from the list of odd stations before repeating. This is done |V| times or essentially until all odd stations are paired up giving an overall time complexity of O(|V|^3). The current implementation in the API is not as efficient as the Blossom algorithm and this is a potential improvement for stage 4. This stage is important as it ensures an Euler path exists for the next stage.

Stage 5 - Euler path

The first step of this stage is to construct a dictionary whose values are the connections to each station. This has time complexity O(E) as all edges are looped through and their stations are inserted to the dictionary with time complexity O(1). The following steps are repeated whilst the minimum spanning tree has 1 or more edges. An Euler Path is initiated with the start station. Next the first station in the Euler Path with neighbours is found which is a O(|V|) operation as in the worst case all stations are looped through. The following steps are repeated until the station from the Euler Path has no neighbours left. The edge connecting the station to its neighbour is removed from the minimum spanning tree and from the dictionary of neighbours. Removing the edge has time complexity O(|E|) as in the index of the stations to remove have to be searched by looping all the edges in the minimum spanning tree. Then the neighbour is inserted to the Euler Path which has time complexity O(|V|) in the worst case that the station is inserted at the start of the list. An example Euler path can be seen to the right. The overall time complexity is O(|V| + |E|).



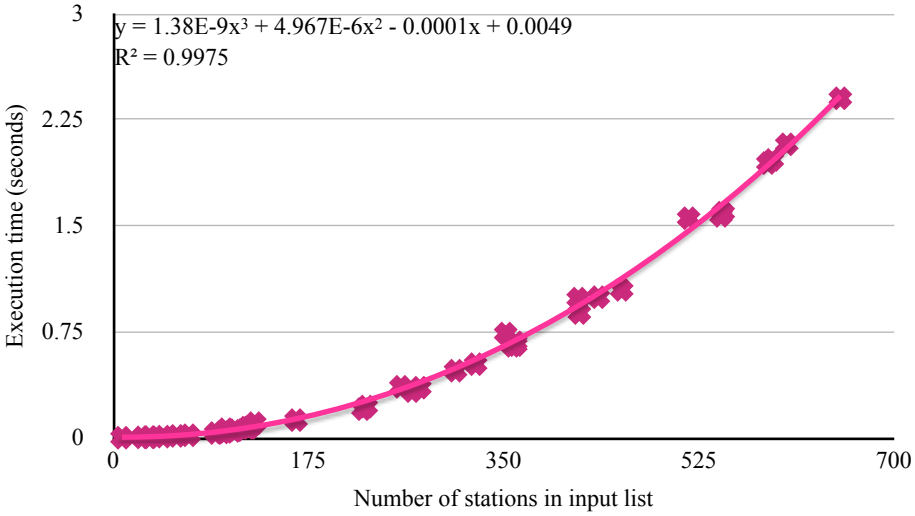
Stage 6 - Take shortcuts and decide start station

Finally, the algorithm loops through all the stations in the Euler Path removing stations that have already been visited. A dictionary is used to keep track of visited stations since it has insertion time complexity of O(1) and since looping all the stations has time complexity O(|V|), the overall time complexity of computing the hamiltonian path has time complexity O(|V|). Next, the start station that will minimise the travelled distance is chosen. This is done by first calculating the average latitude and longitude of all the stations in the hamiltonian path thus time complexity O(|V|). Computing the distance from each station in the hamiltonian path to the average latitude and longitude has time complexity of O(|V|). Therefore the overall time complexity of stage 6 is O(|V|) and this produces a final tour which is a 3/2 - OPT approximation to the minimum distance through all the stations in the input set.

Final time complexity

The final time complexity is O(|V|) + O(|E| log |V|) + O(|E|) + O(|V^3|) + O(|V| + |E|) + O(|V|) ~ O(3|V| + 3|E| + log|V| + |V|^3) ~ O(|V|^3 + |V| + log |V| + |E|) ~ O(|V|^3 + |E|).

Experimental analysis



The graph above was constructed by measuring execution time against number of stations in the input list and an average of 10 runs per set of input stations was taken to plot the above. The plot above supports the claim that the overall time complexity is O(|V|^3 + |E|) as the trend line has equation where the order of growth is 3 and there is a polynomial relationship between number of stations and execution time. The R^2 value is approximately equal to 1 which shows that the points fit these trend line well and there is a strong positive correlation.