Algorithms Individual Report (COMP0005)

Passawis Chaiyapattanaporn(20021096)

Introduction

This report will go into the data structures and algorithms implemented to support the requested API. All data on the charts shown are taken by an average of 5 runs on 5 sample graphs for that specific number of vertices. For complexity analysis V are the vertices of the graph(stations) and E are the edges of the graph (lines connecting stations).

API

LoadStationsAndLines

To support reading the contents of both the files I implemented a method to loop through contents of the file, skipping the first line as it is the column title for both of the files. For 'londonstations.csv' I read each row and store the content in a dictionary with the station's name as its key. After reading the first csv file we loop through the 'londonrailwaylines.csv', which provides connections between stations, and use the names to index into the stored. Using the 'londonrailwaylines.csv' I created an undirected adjacency list graph with edge weights using the latitude and longitude information between the stations. The weight of the edges is calculated using haversine and the format is in miles. The theoretical complexity of this API function is $O(M+N)$, where M is the number of stations loaded and N is the railway station connections(edges). The space complexity of this API would be $O(P+Q)$, where P is the number of stations and Q is the size of the helper class StationInfo.

Minstops

This API calculates the minimum of stops from one station to another, regardless of the actual distance. It is implemented using Breadth-First-Search algorithm and eagerly stopping when it finds the target station for efficiency, instead of computing the path for the whole graph which normal BFS does. This would give this API a theoretical complexity of $O(V+E)$. Where E in this case is the number of edges till the target vertex is found, not the total edges of the graph. The worst case for this is when the we have a complete graph plus one vertex, which is the target vertex, that is connected to the last vertex of the complete graph(near_complete). Breath-First-Search must then go through all the edges in the graph which would degrade the complexity into $O\left(V+V*\frac{V-1}{2}\right)=O(V^2)$ for the worst case, since it cannot stop early since the target vertex is at the very end.

To test for experimental complexity, I have created a random graph of varying vertices and edges, to simulate the railway map. It shows a quadratic trend for the near complete graph (figure 1) which supports our worst-case theoretical complexity. For average (figure 2) and sparse (figure 3) graph they both show linear trend with slight bumps due to the multiple samples taken of the graphs

generated. The result figure 2 and figure 3 thus support our theoretical complexity of $O(V + E)$ as well.

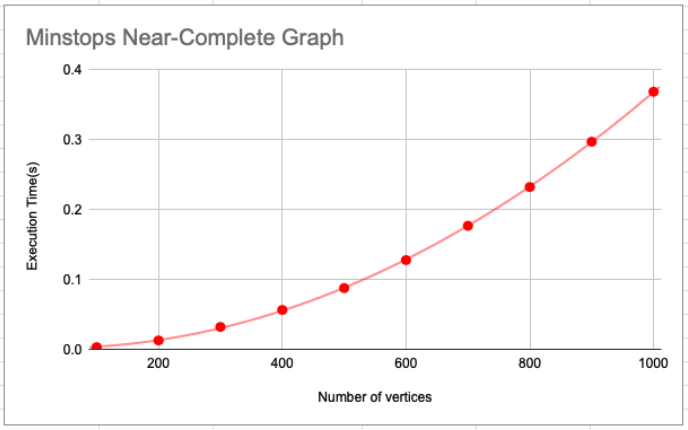

Figure 1:Minstops Worst Case

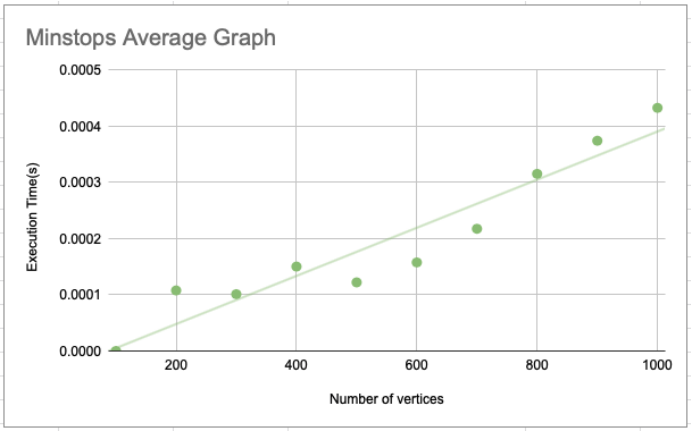

Figure 2: Minstops Average Graph Result

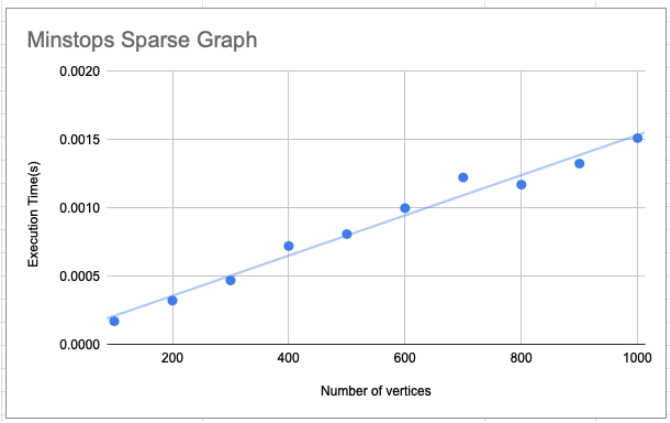

Figure 3: Minstops Complete Graph Result

Mindistance

This API calculates the minimum distance to travel between two stations. It is implemented by using Dijkstra's algorithm without decrease-key. The data that was given is a sparse graph because most of the stations are mostly connected to only two other stations. From the tests comparing the lazy and eager Dijkstra from figure 4, it shows that on sparse graphs tested lazy Dijkstra is faster. The test is done by having two different binary heap priority queue, one that supports decrease-key in logn time by having a dictionary storing the position of each values inside the heap and another that does not.

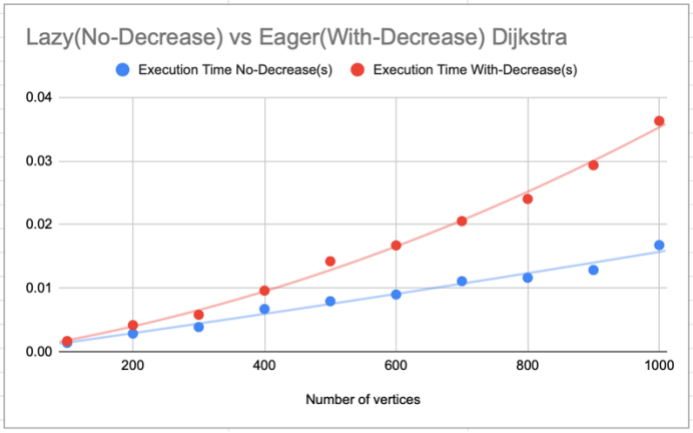

Figure 4: Dijkstra Comparison

For experimental complexity graphs with varying numbers of vertices are created, and for this also the branching rate. The graph generation method made for testing takes in parameters that allows me to control the edge creation rate to vary the graph to be denser or sparser. For Dijkstra the

complexity with binary heap is $O((V+E)logV)$.The worst case for the Dijkstra algorithm would be when the graph is complete which would result in the theoretical complexity to be $O\left(\left(V+V*\frac{V-1}{2}\right)logV\right)=O(V^2logV)$.

From figure 5 it shows a quadratic trend when the graph is a complete graph hence supporting our theoretical complexity of $O(V^2logV)$. In the case of the average (figure 6) and the sparse (figure 7) case both of these show a linearithmic trend for the result generated with sparse results being faster in this case. This makes sense as sparse graph would have fewer total edges(E) since our theoretical complexity is $O((V+E)logV)$, and since the number of vertices remain the same and the target to travel to remains the same for each value of vertices, the only differing variable here are the number of edges. Hence, the number of edges is the differing factor that causes the increase in execution time.

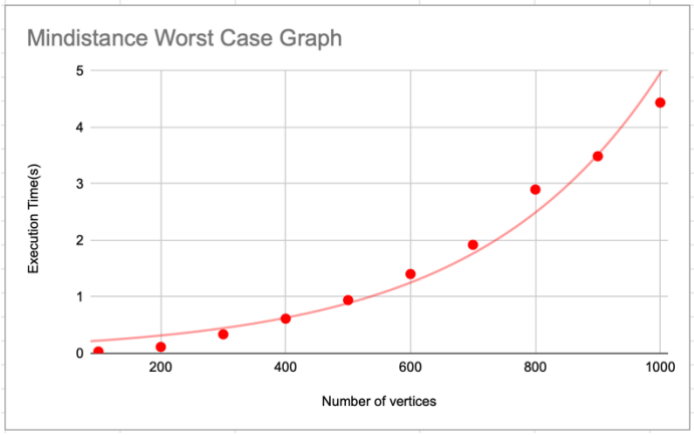

Figure 5: Mindistance Complete Graph Result

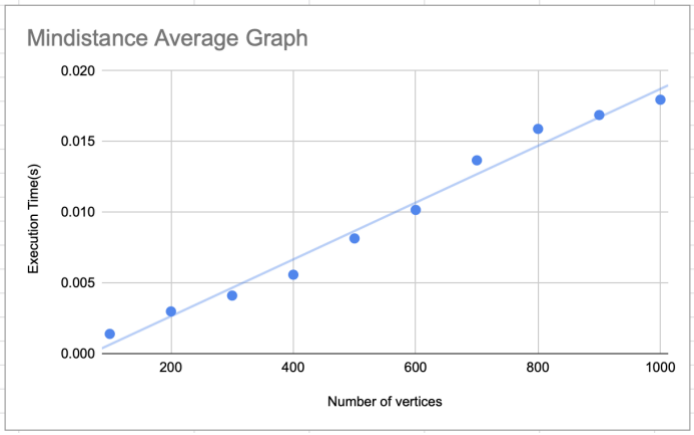

Figure 6: Mindistance Average Graph Result

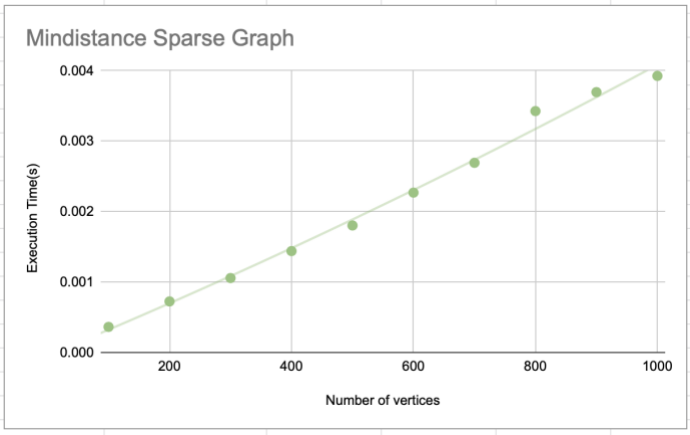

Figure 7: Mindistance Sparse Graph Result

New railway line

This API creates the hypothetical train line provided the list of station names. This problem is similar to travelling salesman, but without returning to the starting point. For this implementation I relaxed the problem to a Euclidean travelling salesman and used an approximation algorithm called Christofides. This is because the travelling salesman problem is NP hard, so it is better to use an approximation/heuristic algorithm to compute approximate results without taking as much time. For this API I had also assumed that no stations inside the stationSet could be repeated and no two stations are at the same place. The criteria for this API that I set is that it does not crash when computing large graphs and the result is acceptable in a good amount of time.

For this API a complete graph from the list of stations provided needs to be created. As this is a complete graph, this process would take $O(V^2)$ as every vertex has to loop through all the vertices to add an edge to it. In this case the V is the number of stations provided.

After the complete graph has been generated, we use Prim's Algorithm with binary heap to compute the minimum spanning tree (MST). I chose to use Prim's algorithm in this case as it is better than the alternative Kruskal's algorithm in a dense graph. The complexity of Prim's on average case is $O((V+E)logV)$ compared to Kruskal's $O(ElogE+V)$. In this case the complete graph will have $V * \left(\frac{V-1}{2}\right)$ edges which would degrade Kruskal's complexity to $O(V^2logV^2) = O(2V^2logV) = O(V^2logV)$ whereas Prim's algorithm in our implementation with binary heap will degrade to $O(V^2logV)$. (Same Big O but Prim's has less constant which makes it faster)

Then we need to compute a min-weight maximal matching of the odd degree vertices (vertices with odd number of edges), to do this loop through the MST and check each of the vertex adjacency's list length. This process would result in $O(V)$. To compute the min-weight maximal matchings I am using the greedy approximation algorithm. From the odd vertices create edges to all the other odd vertices and add it to a priority queue, this would have a complexity of $O(D^2logD^2)$, where D is the number of odd vertices. Dequeue each edge that the vertices have not been matched selected and add it into the MST until you get half of the number of odd vertices.

Perform Depth-First-Search on the combined graph which will return the eulerian tour by appending the name of the stations that have not been visited, this will have a complexity of $O(V+E)$.

The theoretical complexity of this API would be $O(V^2 + V^2logV + V + 2(D^2logD^2) + V + E)$ and could be simplified to $O(V^2logV)$.

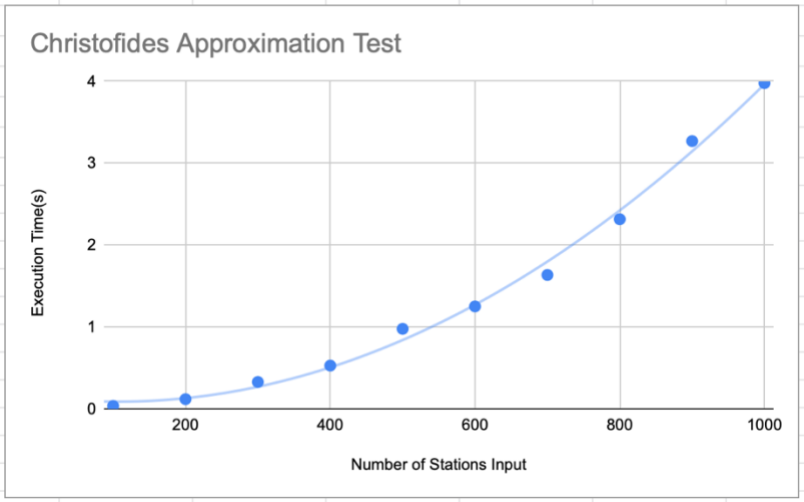

Figure 8: Christofides Results

From figure 8 we can see that the experimental analysis supports the theoretical complexity of $O(V^2logV)$, as the amount of input vertices increases the time increases quadratically. The method is more efficient than a brute force solution, whilst still producing an acceptable solution.

My implementation of Christofides could be improved in computation time by storing the initial complete graph in matrix form and simply indexing into them instead of recomputing the weights during the min-weight maximal matching computation. However, the space-complexity would increase quadratically and whilst testing on my current laptop I found the upper-bound to compute the number of vertices to be 10000 when representing the graph as an adjacency matrix. Hence to guarantee a result I chose to still represent my graph as an adjacency list for this API.