INTERNATIONAL BACCALAUREATE

MATHEMATICS (ANALYSIS AND APPROACHES) EXPLORATION
HIGHER LEVEL

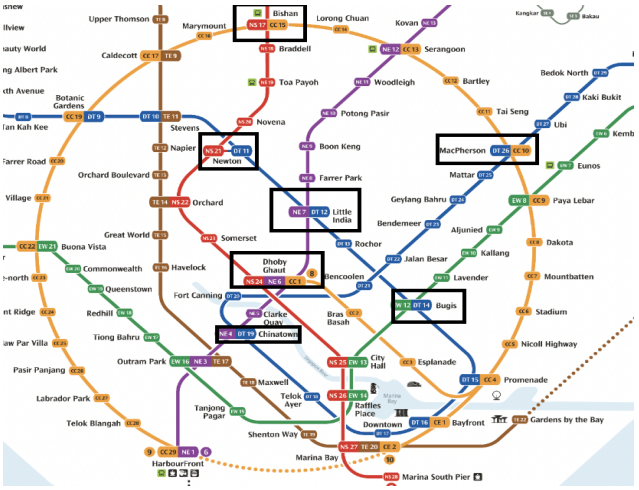
Exploring and evaluating 3 different methods to solve the Travelling Salesman Problem (TSP), using a real-life application, to find which one is optimum in different scenarios

Total number of pages: 22

Introduction

Singapore's public transportation system is one of the most-well connected and efficient systems in the world. A key contributing factor is Singapore's Mass-Rapid Transport (MRT) system – Singapore's train system. As of November 2022, Singapore's MRT system was made up of more than 140 MRT stations which were connected by 5 MRT lines. (LTA, 2022) During a field trip in secondary school, I took part in a scavenger hunt. The goal of the scavenger hunt was to visit 7 MRT stations and take a picture at each of them, and return back to the starting MRT station, in the shortest time possible.

Figure 1: Part of the MRT map of Singapore showing the 7 MRT stations I had to visit (Andres, 2023)



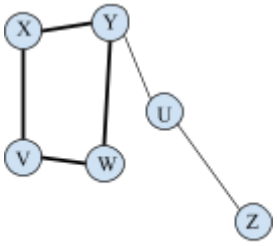
The 7 MRT stations I visited were Bishan, Newton, Dhoby Ghaut, Chinatown, Little India, Bugis and MacPherson.[in black rectangles in Fig. 1] **The starting MRT station was Newton.** During the scavenger hunt, my group went from station to station randomly, without much strategy.

However, a few years later, I came across the Travelling Salesman Problem(TSP). The TSP is an algorithmic problem which finds the shortest route between a set of locations that are visited.(TechTarget, 2020) For a TSP, a particular location cannot be visited more than once and the starting location has to be returned to. I then realised that the scavenger hunt I took part in modelled the TSP. This piqued my interest and I wanted to explore if I could apply the TSP to this scenario to find a more efficient route. For my application of the TSP, I have to visit 6 other stations and return back to the starting station. In my scavenger hunt, there was no rule that we could not re-visit the MRT stations. However, if the MRT stations (except

Newton) were visited more than once, the route taken would definitely not be optimal. Hence, it can be assumed that the stations can only be visited once in order to achieve an optimal (or at least efficient solution). Thus, the scavenger hunt models a TSP. In the TSP, the various points (stations) are known as vertices and are connected by edges. The weight of the edge denotes the time taken to travel between the 2 vertices the edge connects. To solve the TSP, the shortest route (taking the least time) has to be found where all vertices are visited and the starting vertex is returned to. My application of the TSP is relatively simple, but it can be extended to a large number of vertices (e.g. 50) with significant utility. E.g. deliveries to multiple locations within a country.

The TSP is an optimization problem, and is part of graph theory. Graph theory is a sector of mathematics that is usually concerned with a group of points connected with lines (Carlson, S. C. 2022, December 23). The points are known as vertices or nodes. The TSP cannot be applied to all graphs. For a TSP, the graph has to be a cycle since the starting vertex has to be returned to. A cycle is when the edges of the graph form a path and the starting vertex is returned to.(Weisstein, 2000) A path is a sequence of unique vertices connected through edges present in a graph. (Fulber, 2022) More specifically, a graph of the TSP builds on the concept of a Hamiltonian cycle – a graph cycle visiting each vertex only once. (Weisstein, 2003) A graph possessing a Hamiltonian cycle is said to be a Hamiltonian graph.

Figure 2: An example of a Hamiltonian graph



For example, referring to Figure 2, both Y-X-V-W and Y-U-Z are paths. However, only Y-X-V-W-Y (bolded) is a Hamiltonian cycle, since each vertex is visited once and returns to the starting vertex of Y. However, the unbolded path, Y–U–Z ,is not a Hamiltonian cycle as although each vertex is only visited once, the starting vertex is not

returned to. Although the graph is a Hamiltonian graph, the TSP can only be applied to the bolded edges (the Hamiltonian cycle). For this exploration, I will be solving an application of the Hamiltonian cycle – the TSP, to find an efficient route for that scavenger hunt.

Methods to solve the TSP

There are many mathematical methods to solve the TSP including branch and bound and the Brute-force approach which give the optimum solutions, and other algorithms such as the Nearest Neighbour Algorithm and the Christofides algorithm are approximation algorithms which do not always give the optimal solution, but provide a solution with a certain level of efficiency. To deepen my understanding of the different algorithms used to solve the TSP, I will be exploring and using 3 methods to solve my application of the TSP to evaluate which is the most optimum or efficient, and which should be used to solve the TSP in a particular circumstance. I will be evaluating the use of the Brute-force approach, the Nearest Neighbour algorithm and the Christofides algorithm.

Aim of the exploration

Exploring and evaluating 3 different methods to solve the Travelling Salesman Problem (TSP), using a real-life application, to find which one is optimum in different scenarios.

Formulating a graph with vertices and edges to solve the TSP

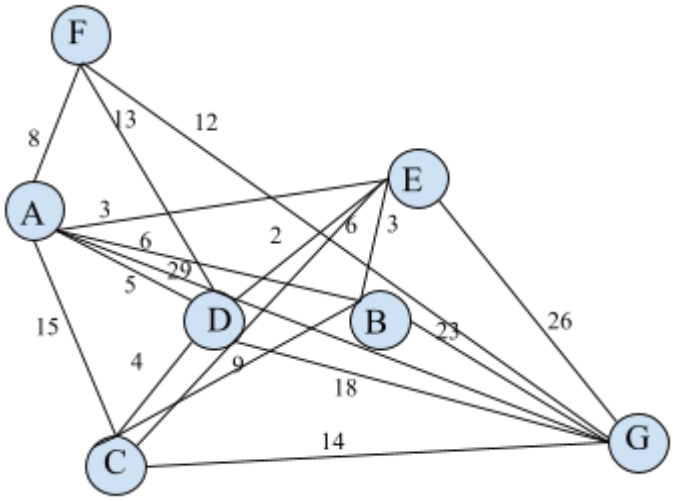
To make it less cumbersome, each MRT station is labelled with a letter. A- Newton, B- Bugis, C- Chinatown, D- Dhoby Ghaut, E- Little India, F- Bishan, G- MacPherson

In this graph of the TSP, the edges are two-directional, since the edge between the MRT stations can be travelled in either direction (the train service is available in both directions) i.e. One can travel from A to B or B to A, and both would take the same time. The edge between the vertices denotes a direct train available between them (the case where 2 or more trains had to be taken between stations [vertices] was not considered)

Table 1: Table of time, in minutes, taken to travel between vertices (using Google Maps)

Vertex	A	B	C	D	E	F	G
A	0	6	15	5	3	8	29
B	6	0	9	-	3	-	23
C	15	9	0	4	6	-	14
D	5	-	4	0	2	13	18
E	3	3	6	2	0	-	26
F	8	-	-	13	-	0	12
G	29	23	14	18	26	12	0

The dashes (-) represent the situation where the vertices have no direct edge between them
All this information can be collated into a simple diagram
Figure 3: A graph of the vertices A, B, C, D, E, F and G, connected by their respective edges



Key: A-B represents the edge between vertex A and vertex B, and has the same weight as B-A since the edges are two-directional. The weight of the edge is given beside the respective edge.

This graph is not a complete graph. A complete graph is a graph in which every pair of vertices is connected by an edge. (Weisstein, 2001) For example, vertex F and E are not connected by an edge.

Using the Brute-force approach to solve the TSP

The Brute-force approach can be used to solve the TSP. The Brute-force approach calculates and compares all possible routes that can be formed, to determine the shortest unique solution. To solve the TSP using the Brute-Force approach, one must calculate the total number of routes and then find the sum of the weights of the edges comprising the route, which gives the time taken for that route. The Brute-force approach is the most straight-forward way to solve the TSP; find the time taken for all the possible routes and

choose the shortest route. **Let *t* be defined as the time taken for the most optimal route.** The Brute-force approach finds *t*.

Figure 4 : Diagram showing all the various routes formed using the Brute-force approach for the first 3 vertices

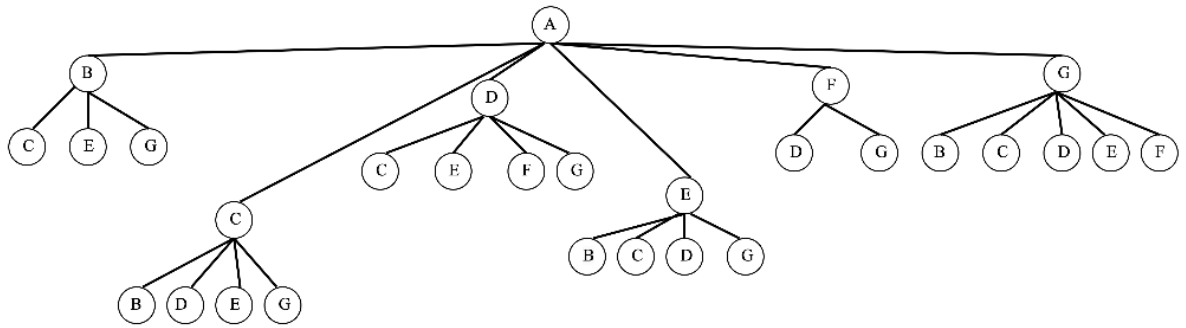


Figure 4 only shows the possible "branches"(permutations) for the first 3 vertices, and there are already 22 different possibilities. If all the permutations of the different vertices are found, this could result in a very large number of total possible routes. **In the case that every vertex is connected to every other vertex**, there are 6! = 720 possibilities. In general, there are (n-1)! possibilities for a complete graph **(*n* refers to the number of vertices in the TSP)**, when the starting vertex is fixed. This is because from vertex A, 6 possible vertices can be visited, and after this vertex is visited there are 5 possible vertices that can be visited next, and so on until the last vertex is reached. However, the graph is not a complete graph and the number of routes is below 720. Nonetheless, there are still a large number of possible routes, and calculating the time taken for all of the different routes can be cumbersome.

Due to the cumbersome nature of the Brute-force approach, for my application of the TSP, the possible routes and the time taken for each route are only found in the case that B is the second vertex in the route (after vertex A). In this case, the shortest route is A-B-E-D-C-G-F-A, taking 49 minutes. (refer to Table 4 in the Appendix) There may or may not be a more optimal route with a weight below 49 minutes when the time taken for all the other routes is found. Hence, $t \leq 49$.

Using the Nearest Neighbour algorithm to solve the TSP

The TSP can be solved using the Nearest Neighbour algorithm which provides an approximation of *t,* and gives a route with a certain level of efficiency. The Nearest Neighbour algorithm can be used to find the upper bound (maximum value) of *t* and gives a route that minimises the time taken, however it is not always the most optimum route. The Nearest Neighbour algorithm is a 'greedy' heuristic algorithm. A 'greedy' algorithm takes the best immediate, or local, solution while finding a solution at all times. Greedy algorithms find the overall, or globally, optimal solution for some optimization problems, but may find less-than-optimal solutions for some instances of other problems. (Black, 2005)

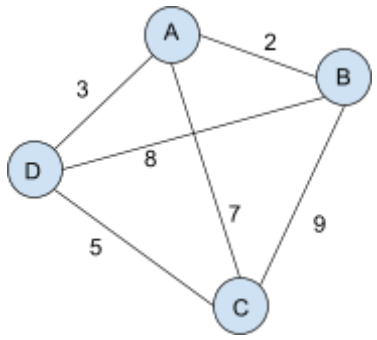
Table 2: The steps and their justification for the Nearest Neighbour algorithm

Step	Step	Justification for carrying out the step
1	Initially, all vertices are unvisited	The TSP involves a Hamiltonian cycle, where all the vertices are unvisited, and have to be visited, and the starting vertex has to be returned to.
2	From the starting vertex, visit the nearest unvisited vertex. From that vertex, visit the nearest unvisited vertex again	Since the Nearest Neighbour algorithm is a greedy algorithm, it constantly finds the nearest unvisited vertex, to obtain an efficiency of some sort. Hence, from the starting vertex the nearest vertex is found, and this process is repeated until all the vertices are visited.
3	Once all vertices have been visited, return back to the starting vertex	Since the solution for the TSP is a Hamiltonian cycle, the starting vertex has to be returned to from the last vertex, after all the vertices are visited. This is one of the key problems of the algorithm, and is explained below.

4	The sum of the weights of the route, denoted by *s,* gives the upper bound and the route taken is an optimum route	The Nearest Neighbour algorithm gives an upper bound for the TSP because it does not account for the last edge used, and does not consider the repercussions of taking an edge with a low weight. (explained further below) However, in the case that there are no repercussions, *s* can be the weight of the optimal route.($s=t$) Hence, the Nearest Neighbour algorithm provides the optimal solution, or one with a total weight greater than *t,* thus giving an upper bound for *t.* Hence, $t \leq s$.

The Nearest Neighbour algorithm finds an optimum path because at each vertex, it moves to the next nearest vertex. Hence, in a way, it eliminates a lot of nonoptimal paths, thus giving a more efficient solution than a lot of other routes.

Figure 5: A Hamiltonian graph with only 4 vertices



Referring to Figure 5, if the Nearest Neighbour algorithm is applied from vertex A, the optimal route will be A-B-D-C-A, taking 22 minutes. This is still less than various other routes; for example, if one takes the route A-D-B-C-A, it would take 27 minutes. The paths A-D and D-B for example, were not considered by the algorithm (while obtaining the route A-B-D-C-A) because these edges have a greater weight than the other edges connected to vertex A and D respectively. Thus, the Nearest Neighbour algorithm does provide a certain level of efficiency.

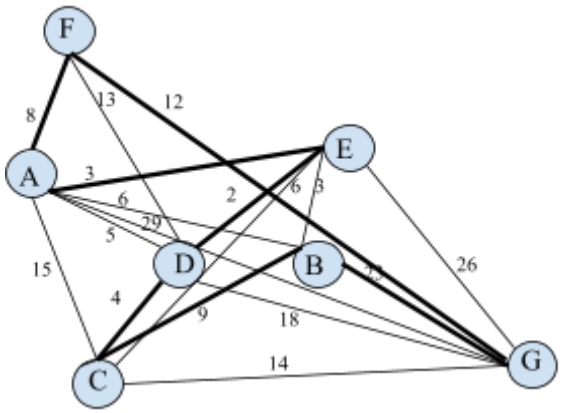
However, the Nearest Neighbour algorithm does not always give the optimal solution because it does not take into account the last edge taken (from the last vertex to the starting vertex), and the repercussions of taking a short edge which may result in the need to take a longer

edge later. If the following edges or the last edge are significantly longer because an edge with a lower weight was chosen by the algorithm initially, the time taken for the route found using the Nearest Neighbour algorithm be greater than *t*. Referring back to Figure 5, the path A-D-C-B-A takes 19 minutes, however this was not considered in the Nearest Neighbour algorithm as in the first step, it chooses to go to vertex B instead of D as B is nearer, however does not factor in the repercussion that vertex B is much further away from C than D. Hence the Nearest Neighbour algorithm sometimes does not give the **most** efficient route.

The Nearest Neighbour algorithm gives the optimum route sometimes, but in certain cases such as above, it only provides an efficient, but not the optimum route. Hence, the total weight of the route given is equal to or above *t,* giving the upper bound of *t*. The Nearest Neighbour algorithm gives a maximum of $\frac{1}{2}[\log_2(n)] + \frac{1}{2}$ times the duration of the optimal route. (Rosenkrantz et al., 1977)

Applying the Nearest Neighbour algorithm to my application of the TSP

Figure 3 (for reference)



Referring to Figure 3, from vertex A, the nearest vertex is E. From vertex E, the nearest unvisited vertex is D. From vertex D, the nearest unvisited vertex is C. From vertex C, the nearest unvisited vertex is B. From B only vertex G can be visited since all the other vertices except F are visited (and there is no direct edge connecting vertex B to F). From G, the only unvisited vertex is F. From F, the starting vertex A is returned to (since all the vertices have been visited). The edges used are A-E, E-D, D-C, C-B, B-G, G-F and F-A. The sum of their weights, *s,* is 3 + 2 + 4 + 9 + 23 + 12 + 8 = 61 minutes. Hence, $t \leq 61$ minutes. In my application of the TSP,

the Nearest Neighbour algorithm gives a maximum of $\frac{1}{2}[\log_2(7)] + \frac{1}{2} = 1.90$ (3s.f.) times of *t*. In other words, $t \leq s \leq 1.90t$.

Using the Christofides algorithm to solve the TSP

The Christofides algorithm is a heuristic algorithm to find a near-optimal solution to the TSP and gives a solution of a certain level of efficiency.(Black, 2020) The route provided by the Christofides algorithm has a total weight of a maximum of $\frac{3}{2}$ times of *t*. (Gilbert & Halim, 2016) The Christofides algorithm can only be used if the graph of the TSP is symmetrical – when the distances between any two vertices (the weight of the edges) are the same in both directions. (OpenAI, personal communication, March 13, 2023) Moreover, the graph has to satisfy the triangle inequality – for every three vertices x, y, and z, it should be the case that for the weight of the edges, *w, w (x y) + w(y z) ≥ w(x z).* (Gendreau et al., 1997) My application of the TSP satisfies both conditions hence the Christofides algorithm can be applied to find an approximation for the optimal solution. The Christofides algorithm finds the Eulerian cycle for the graph, and then converts it into a Hamiltonian cycle by skipping repeated vertices (refer to Table 3).

These are certain definitions used to prove or are in the Christofides algorithm:

Eulerian path: A Eulerian path visits every edge in the graph once (and an edge cannot be visited more than once) (GeeksforGeeks, 2023)

Eulerian cycle: A Eulerian cycle is a Eulerian path that starts and ends on the same vertex (GeeksforGeeks, 2023)

Multigraph: The term multigraph refers to a graph in which more than one edge exists between 2 vertices (Harary,1994) (Gross and Yellen, 1999)

Degree of a vertex: The number of edges incident to a vertex (Lehman et al., 2021)

Matching: A graph where no two edges share an endpoint (i.e. all vertices have a degree of 1)

(GeeksforGeeks, 2022)

Construction of the Christofides Algorithm

Table 3:The steps for the Christofides algorithm and their justification or proof

Step	Step	Proof / Justification
1	Let the graph of the TSP be represented by *G,* where *G = (V, E)*	Since *G* is a graph of the TSP, it contains a Hamiltonian cycle connecting *V* vertices via *E* edges.
2	The solution to the TSP in *G* is given by an Eulerian cycle.	The Hamiltonian cycle in *G* also consists of an Eulerian cycle. In a Hamiltonian cycle, if a particular edge was to be used again, a vertex would be visited again. In the Hamiltonian cycle, only the starting vertex is visited twice, but an edge cannot be used twice to revisit the starting vertex because it would result in a route with only 2 vertices e.g. A-B-A. Thus, each edge can also only be used once in a Hamiltonian cycle. Hence, the Hamiltonian cycle of this TSP also consists of an Eulerian cycle when there are more than 2 vertices. (which is always the case for the TSP as the TSP will not be used for a graph with 2 vertices as it is pointless)

3	However, for an Eulerian cycle, all vertices need to have an even degree.	In an Eulerian cycle, the vertices can be visited multiple times. However, since edges cannot be repeated, the edge used to visit a vertex cannot be used to visit another vertex from that vertex. For each time a vertex is visited, and is used to visit another vertex, a "pair" of edges is used. Thus, in order for a Eulerian cycle to exist, a vertex has to be connected to p "pairs" of edges, where $p \in Z^+$, hence all vertices have an even degree and are visited p times.
4	Find the minimum spanning tree, T, of G on V .	The minimum spanning tree of a weighted graph connects the vertices together with a minimum weight (using V - 1 edges)(refer to page 15). However, it does not form a Hamiltonian cycle (which has V edges), thus $t > wt(T)$. The use of the minimum spanning tree and minimum cost perfect matching is to ensure all the vertices have an even degree (explained in Step 7) A minimum spanning tree is obtained using Kruskal's algorithm. (see page 15)
5	Let O be the vertices in T with an odd degree.	Since all the edges in a graph are connected to 2 vertices, each edge = 2 degrees (an edge contributes 1 degree each for 2 vertices). There are E number of edges, where $E \in Z^+$, hence there are a total of 2 E degrees amongst V. Hence, the total degree of all vertices, V, is even. Now consider the vertices in V - O, where V - O is the set of vertices with an even degree

		Since they have an even degree, the total degree of this set of vertices can be represented as $2k$, where $k \in Z^+$ The degree of O = degree of V - degree of $(V-O) = 2E-2k = 2(E-k)$, hence the total degree of the vertices with an odd individual degree is even, since E and k are positive integers. **Hence, there must be an even number of vertices in *O* (since the product of an odd number and an even number is always even)**
6	Find M, the minimum cost perfect matching for O	A minimum cost matching is needed so all the vertices have an even degree(explained in Step 7). A perfect matching is when all vertices have a degree of 1. For a perfect matching, the number of vertices in the graph has to be even. This is since in a perfect matching, each vertex is only connected to one edge, thus vertices can be thought of as "pairs". If there are an odd number of vertices, the additional vertex has to be connected to a vertex in a "pair" to ensure a connected graph, which would result in that vertex in the "pair" being connected to more than one edge, hence would no longer be a perfect matching. The idea of a perfect matching can be hence used to pair up vertices in a multigraph with an odd degree, which is possible as the number of vertices in a graph with an odd degree is always even (proved above). A minimum cost perfect matching is found to ensure the lowest possible weight of the edges to obtain an efficient route.

7	Construct the multigraph $G_2 = (V, T \cup M)$. All vertices in G_2 have an even degree.	T includes all the vertices in G, and in T, there are vertices with an odd degree (O) and vertices with an even degree (V - O). For O, the vertices are connected by an additional edge by the minimum cost perfect matching, hence in G_2, they have an even degree. The vertices in V - O remain the same with an even degree. All vertices have an even degree.
8	Now, every vertex has an even degree, thus the Eulerian cycle for G can be found on edges $T \cup M$. Repeated vertices are skipped to obtain a Hamiltonian cycle.	Although a Hamiltonian cycle in G also consists of an Eulerian cycle, an Eulerian cycle does not always consist of a Hamiltonian cycle, as in an Eulerian cycle, a vertex can be visited more than once to ensure an edge is not repeated. Hence, the Eulerian cycle does not give the route for the TSP directly. To obtain the Hamiltonian cycle, the repeated vertices in the route obtained have to be skipped. However, if edges are skipped, edges $\notin T \cup M$ may have to be used so that route can be achieved.

The Christofides algorithm gives a route with a weight of maximum $\frac{3}{2}$ times of t.

Proof:

Let X be the route for t. Let X' be the cycle on odd vertices which are not repeated. X' has an even number of vertices (proven above). Assume that $X' = (V_1, V_2, V_3 ... V_{2k})$. Moreover, cost $(X') \leq \text{cost}(X)$ since X' has fewer or equal number of vertices than X. (It is equal when no vertex has an even degree) Two possible matchings can now be constructed,

$M_1 = (V_1, V_2), (V_3, V_4), (V_5, V_6) ... (V_{2k-1}, V_{2k})$

$M_2 = (V_2, V_3), (V_4, V_5), (V_6, V_7) ... (V_{2k}, V_1)$

Both of these matchings have $\frac{1}{2}O$ edges. Let M be the minimum cost perfect matching,

hence, cost (M) ≤ cost (M_1) and cost (M) ≤ cost (M_2)

cost (M_1) + cost (M_2) = cost (X')

cost (X) = t, hence cost(X') ≤ t (because cost(X') ≤ cost(X))

Together, the inequality, 2 cost(M) ≤ cost(M_1) + cost(M_2) ≤ t is obtained

Hence, **cost(M)** ≤ $\frac{1}{2}t$

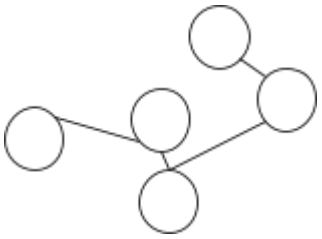
cost(T) ≤ t , since T is the minimum spanning tree of the graph

Let B be the Hamiltonian cycle given by the Christofides algorithm

cost(B) ≤ cost(T) + cost(M) (if vertices are repeated, they are skipped, hence at least one edge in T or M is skipped)

Since **cost(M)**≤ $\frac{1}{2}t$ and **cost(T)** ≤ t, **cost(B)**≤ $\frac{3}{2}t$ (Gilbert & Halim, 2016)

Figure 6: An example of a spanning tree between 5 vertices



Kruskal's algorithm is used in the Christofides algorithm to find the minimum spanning tree of a weighted, connected graph. (GitHub, 2022) A spanning tree is simply a tree connecting all the vertices in a graph. The minimum spanning tree is a spanning tree with the lowest weight. The steps of the Kruskal's algorithm with a brief explanation is given –

1. Sort all edges in ascending order of their weight edges, and pick the smallest edge

Kruskal's algorithm essentially tries to find the edges with the smallest weight which connect the vertices. Picking the edges in ascending order helps to build a minimum spanning tree, by considering edges which have smaller weights first, then edges with larger weights.

2. Check if the new edge creates a cycle or loop in the spanning tree. If it does not form a cycle or a loop in the minimum spanning tree, the edge is not used.

If the edge creates a cycle or a loop, it is no longer a spanning tree and there is a vertex being connected to the remaining vertices with 2 edges. If an edge forms a loop, again there is extra edge being included as the edge is only connected to that vertex itself, (not connected to any other vertex) hence is not part of the minimum spanning tree.

3. Repeat until *V* - 1 edges are in the minimum spanning tree (*V* is the number of vertices in the graph) (S, 2023)

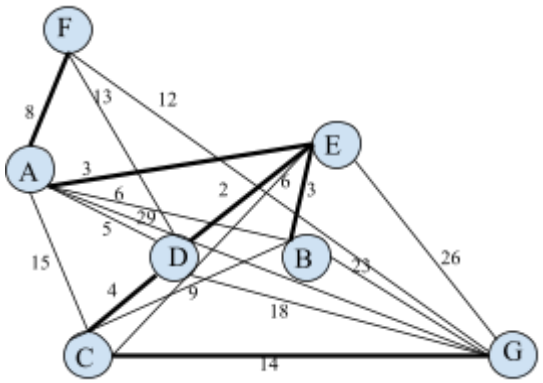
A connected graph with *V* vertices can be connected with *V*-1 edges, hence if any edge after *V*-1 edges are found by the algorithm is added to the minimum spanning tree, it will result in the formation of a cycle (a vertex is connected by 2 edges to the remaining vertices), hence does not give a spanning tree.

A simplified method for the Christofides algorithm is as follows

1. Find the minimum spanning tree, *T*, of *G* using Kruskal's algorithm
2. Let *O* be the vertices in *T* with an odd degree.
3. Find the minimum cost perfect matching of *O*, *M*
4. Construct the multigraph $G_2 = (V, T \cup M)$, where all vertices in G_2 have an even degree.
5. Find the Eulerian cycle of the graph, and skip repeated vertices so that a Hamiltonian cycle is obtained (Gilbert & Halim, 2016)

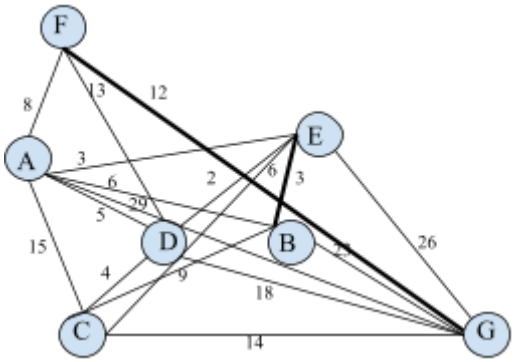
Applying the Christofides algorithm to solve my application of the TSP

Figure 7 : The minimum spanning tree of the TSP graph



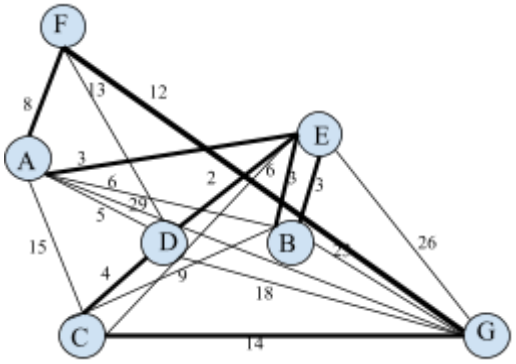
Kruskal's algorithm can be used to find the minimum spanning tree of the TSP. The bold edges represent the minimum spanning tree of the TSP. These edges constitute *T*.

Figure 8: *M*, the vertices in *T* with an odd degree, and their minimum perfect matching



Vertices B, E, F and G have an odd degree in *T*. Their minimum cost matching is represented by the bolded edges.

Figure 9: The multigraph of $T \cup M$



In the multigraph $T \cup M$, there are 2 edges between vertex B and E, both with a weight of 3. There is a Eulerian cycle – A-F-G-C-D-E-B-E-A. When the repeated vertices are skipped, the Hamiltonian cycle, A-F-G-C-D-E-B-A is obtained. Its weight is given by 8 + 12 + 14 + 4 + 2 + 3 + 6 = 49 minutes.

Evaluating the methods to solve the TSP

Time complexity is the amount of time taken by an algorithm to run, as a function of the length of the input. (Great Learning, 2022) Time complexity does not give the operational time for the algorithm, but rather shows how the execution or running time changes with the input (in the case of the TSP, the number of vertices). There is a relation between the input data size (number of vertices) (n) and the number of operations performed (N) with respect to time, and is denoted as Order of growth in time complexity and given notation $O(n)$.(Great Learning, 2022) There are different time complexities such as $O(n)$, $O(n^2)$, $O(\log n)$ and $O(n!)$. The fastest growing function of the input size is the leading (dominant) term; its coefficient is not included in the time complexity equation. (OpenAI, personal communication, March 25, 2023)

The Brute-force approach usually has a time complexity of *O(n!)*. In the Brute-force approach, all the possible routes are formed, to find the route with the minimum total weight. As mentioned earlier, in the Brute-force approach, (n-1)! routes can be formulated, when the starting vertex is fixed. However, if the starting vertex is not fixed, one out of n vertices have to be chosen first. Hence, $n(n\text{-}1)! = n!$ routes can be formulated. Thus, $n!$ number of routes have to be generated, its permutation has to be checked (an edge between 2 vertices may not exist), and if the route is viable, the weights of its edges are summed. If each route being generated, checked and the weights of its edges being summed is considered an operation, there are a total of $n!$ operations that have to be performed. Thus, the time complexity of the algorithm can be given by $O(n!)$. **However, for my application of the TSP, since the starting vertex is fixed, the time complexity is given by *O([n-1]!)***.

The Christofides algorithm has a time complexity of $O(n^3)$.

Step 1: Find the minimum spanning tree, *T*, of *G*, using the Kruskal's algorithm

For the first step of the Christofides algorithm, a minimum spanning tree of the graph has to be found. Kruskal's algorithm is used to find the minimum spanning tree, which has a time complexity of $O(E \log(E))$, where E is the number of edges on the graph of the TSP. (Christine, 2020) The entire proof for this algorithm is not shown because it involves a fair amount of computer science, but it is justified. For this algorithm, the edges have to be sorted and considered to be added in the minimum spanning tree in ascending order of weights, and if the edge considered for the minimum spanning tree creates a loop or a cycle, it is not used in the minimum spanning tree. When the edges are sorted in ascending order, computer science algorithms such as mergesort are used, which has a time complexity of $O(n \log n)$ [n refers to the elements that have to be sorted] Thus, the time complexity to sort the edges in ascending order can be expressed as $O(E \log E)$ (Piterman, 2020) Then, each of the edge has

to be checked to see if it forms a cycle or a loop, hence E operations have to be performed (1 operation per edge), hence its time complexity can be represented as $O(E)$. Hence, the total time complexity of Kruskal's algorithm is $O(E \log E) + O(E)$. (OpenAI, personal communication, March 25, 2023)

Step 2: Let *O* be the vertices in *T* with an odd degree.

To find the vertices with an odd degree, every vertex has to be checked if it has an odd or even degree. This step takes an equal amount of time for each vertex. Hence, the execution time increases by the same amount for each vertex added to the input. Thus, the time complexity of this step of the algorithm is $O(n)$.

Step 3: Find the minimum cost perfect matching, *M,* of *O*

For Step 3, a minimum cost perfect matching has to be found on the odd vertices, which has a time complexity of $O(n^3)$ found using the Kuhn-Munkres algorithm. (Takhirov, 2017) This time complexity equation is not proved, but is justified. The time complexity increases at an increasing rate as the number of vertices increases. For simplicity sake, a complete graph is considered. When there are 2 vertices with an odd degree whose perfect matching needs to be found, there is only 1 edge between them.(this refers to the total number of edges, E, in the complete graph) When this is raised to 4 vertices, there are a total of 6 edges between them. When there are 6 vertices, there are a total of 14 edges between them and so forth. Hence, as n is increased by 2, the number of edges that can be formed between them increases at an increasing rate. To find the minimum cost perfect matching, all the possible combinations of these edges to obtain perfect matchings have to be found and evaluated to find the one with the lowest weight. As the number of edges increases at an increasing rate, the number of combinations of edges that can be formed also increases at an increasing rate. If finding the weight of each possible combination of edges is considered an operation, the execution time of the algorithm increases at an increasing rate, and the time complexity of finding the

minimum cost perfect matching can be expressed as $O(n^3)$. Even in an incomplete graph, as the number of vertices increases, the number of edges increases, hence the time complexity can still be expressed as $O(n^3)$.

Step 4: Construct the multigraph $G_2 = (V, T \cup M)$. All vertices in G_2 have an even degree.

When there are n vertices in a graph, a matrix or table is filled with $n \times n$ weights of the edges (e.g. Table 1.2). Hence for n vertices, n^2-n different edges have to be checked and evaluated if they can be part of the graph or not. (E.g., an edge between 2 vertices might not exist in an incomplete graph) n vertices are subtracted because each vertex does not have an edge to itself. In a multigraph of $\boldsymbol{T} \cup \boldsymbol{M}$, there is just a repetition of a certain edge in E, hence even in a multigraph, still only n^2-n edges have to be checked. If each time the edge is checked is considered an operation, the time complexity of this step can be represented as $O(n^2)$, since n^2 is the leading term. In the case that the graph is symmetrical, the edge from one vertex to another is the same in the opposite direction. Hence, only $\frac{n^2-n}{2}$ edges have to be checked. The time complexity equation is still given by $O(n^2)$.

Step 5: Find the Eulerian cycle of the graph.

In an Eulerian cycle, all edges in the graph have to be visited. If each edge visited is considered an operation, as the number of edges in E increases, the execution time increases by an equal amount for every edge added, hence the time complexity can be represented as $O(E)$.(Aksoy, 2022)

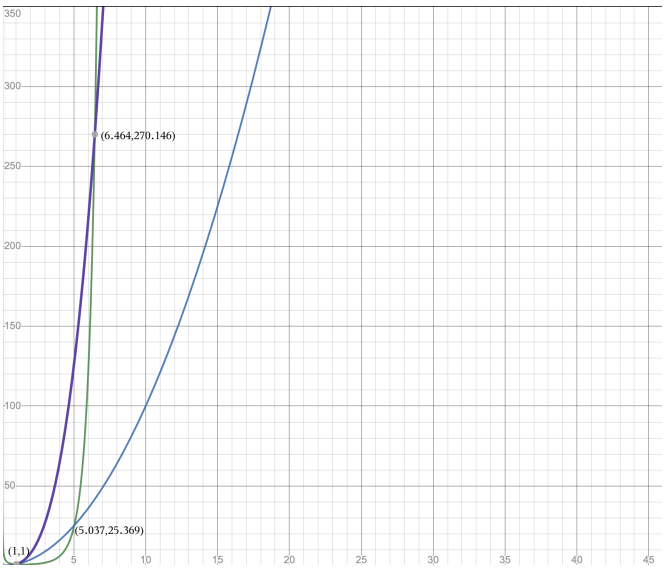
Step 6: Skip repeated vertices so that a Hamiltonian cycle is obtained.

To obtain a Hamiltonian cycle by skipping repeated vertices, the Eulerian cycle has to be checked for repeated vertices. The length of the Eulerian cycle needed to be checked increases by the same amount per vertex added, hence the execution time of the algorithm increases by the same amount for every vertex added. The time complexity can be represented as $O(n)$.

Hence, the time complexity of the Christofides algorithm can be written as such – ***O(n³) +O(n²) + 2O(n) + 2 O(E) + E* log*(E).*** In the graph of the TSP, there is a **maximum** of $(n \times n) - n = n^2 - n$ edges, since all TSP graphs are not multigraphs (OpenAI, personal communication, March 27, 2023). Hence the worst case time complexity can be represented as $O(n^3) + O(n^2) + 2\,O(n) + 2\,O(n^2 - n) + (n^2 - n)\log(n^2 - n)$. The worst case time complexity of an algorithm is the greatest time that it may take to solve a problem, assuming the worst possible input.(OpenAI, personal communication, March 27, 2023) The dominant term is n^3, **the overall time complexity of the Christofides algorithm is *O(n³).***

The time complexity of the Nearest neighbour algorithm is *O(n²).* For this algorithm, the nearest vertex from the starting vertex has to be found. Hence, n - 1 vertices have to be checked and evaluated, to see which is the nearest vertex (and if it is connected or not from the starting vertex in the first place). This is repeated for the second vertex, where n -2 unvisited vertices have to be checked. If each vertex being checked is considered an operation, a total of $(n - 1) + (n - 2) + (n - 3) + \ldots + 1$ operations have to be done. This models an arithmetic sequence, with a common difference of -1, and there are $n - 1$ terms. $S_{n-1} = \frac{(n-1)}{2}(n-1+1) = \frac{(n-1)}{2}(n) = \frac{1}{2}n^2 - \frac{1}{2}n$. Since the leading term is n^2, **the time complexity of the Nearest Neighbour Algorithm can be given as *O(n²).***

Figure 10: Graph of *O* against *n*, where *O* represents the execution time of the algorithm



Key: Green – $(n-1)!$ (Brute-force approach)

Blue – n^2 (Nearest Neighbour Algorithm)

Purple – n^3 (Christofides algorithm)

As seen in the graph, when there is only 1 vertex ($n = 1$), all the algorithms have the same time complexity. However, as n is increased to 5, the Brute-force approach and the Nearest Neighbour algorithm have almost the same time complexity (but the Brute-force approach has a lower time complexity), however the Christofides algorithm has a significantly higher time complexity. **Hence, if $n \leq 5$, the Brute-force approach might be a better method, especially because the exact solution is obtained in a relatively short execution time.** When there are 6 or less vertices, the Brute-force approach has a lower time complexity than the Christofides algorithm. When $n>6$, the Brute-force approach has a higher time complexity than the Christofides algorithm, however the Nearest Neighbour algorithm has a significantly lower time complexity than both these algorithms.**The Nearest Neighbour algorithm might be the best method to solve the TSP when $n>6$, as it provides a solution with a reasonable efficiency in a very short execution time. However, if the time minimised is very significant (e.g., if t is very large), or the efficiency of the route obtained is very important, the Christofides algorithm might be better suited as it provides a more efficient solution than the Nearest Neighbour algorithm when $n>6$.**

For my application of the TSP, both the Brute-force approach (when partly used) and the Christofides algorithm gave a route taking the shortest time (49 min) of the 3 algorithms. However, the Christofides algorithm took significantly shorter time, hence for my application of the TSP (with 7 vertices), **the Christofides algorithm is the best algorithm to use for my application of the TSP.** However, for my application, if it was expanded to include more MRT stations, e.g.above 10, the Nearest Neighbour algorithm might be a better choice, as it has a lower time complexity, but still gives a solution of reasonable efficiency.

The algorithms have other advantages and disadvantages too. The Nearest Neighbour algorithm is relatively simple and not very cumbersome, but its efficiency in finding the

optimal solution is somewhat limited. If there are a large number of vertices such as 50, the solution obtained is more than 3 times the value of *t.* However, the upper bound it gives can be useful to reduce the time taken for the Brute-force approach. When summing the weights of the edges of a route, if the weights of all the edges are not added yet and the weight of the route is already equal to or greater than the upper bound, the route and all the subsequent routes that can formed from that sequence of vertices can be eliminated as they will not give *t*. In certain cases, the Christofides algorithm might be better, as even though it is more cumbersome, it always results in a route that takes a maximum of $\frac{3}{2}$ times of *t*, independent of the number of vertices. Hence, it is good to use when the number of vertices is large, but the distance (weight) between the vertices is less (the value of *t* is less), so that a maximum of $\frac{3}{2}$ times of *t* would not result in a significantly larger time taken. For example, it is useful for deliveries to a large number of houses in a small district.

Limitations of the exploration

The exploration has a few limitations and weaknesses as it was dependent on a few assumptions. Firstly, Google Maps was used to estimate the time taken for the train to travel from one vertex (MRT station) to another. This was assumed to be correct, but if it is not, it can cause inaccuracies in the edges of the graph and hence the route found by the different algorithms. Moreover, the case where more than one train or alternative forms of public transportation between the MRT stations was not considered, which may have provided a more optimal solution. For example, going from vertex E (Little India) to vertex G (Mac Pherson) takes 26 minutes via a direct train, however if a bus is used, it only takes 23 minutes. The time needed to switch trains and to wait for the trains is also not considered, leading to a slight inaccuracy in the time taken for each route. For example, if a certain route takes slightly longer than the other, but has lower waiting times for all its trains, it could be a more optimal route.

Appendix

Table 4: The time taken for the various routes using the Brute-force approach

Route taken	Time taken
A-B-C-D-E-G-F-A	67
A-B-C-D-F-G-E-A	73
A-B-C-E-G-F-D-A	77
A-B-C-E-G-D-F-A	86
A-B-C-E-D-F-G-A	77
A-B-C-E-D-G-F-A	61
A-B-C-G-E-D-F-A	78
A-B-C-G-F-D-E-A	59
A-B-G-C-E-D-F-A	72
A-B-G-E-C-D-F-A	86
A-B-G-F-D-C-E-A	67
A-B-G-F-D-E-C-A	77
A-B-E-C-D-F-G-A	73
A-B-E-C-D-G-F-A	57
A-B-E-C-G-D-F-A	68
A-B-E-C-G-F-D-A	59
A-B-E-D-C-G-F-A	49
A-B-E-D-F-G-C-A	65
A-B-E-G-C-D-F-A	74
A-B-E-G-F-D-C-A	79

Bibliography

Aksoy, Y. (2022, April 26). *Finding the Eulerian Cycle with Hierholzer's Algorithm | by Yusuf Aksoy*. Medium. Retrieved March 16, 2023, from https://medium.com/@yusufaksoyeng/finding-the-eulerian-cycle-with-hierholzers-algorithm-f60bb773db3c

Andres, G. (2023, January 19). Singapore's MRT network: How has it evolved and what will it look like by 2030? *CNA*. https://www.channelnewsasia.com/singapore/singapore-mrt-network-map-how-it-has-evolved-cross-island-line-jurong-region-line-3215061

Black, P. E. (2005, February 2). *greedy algorithm*. greedy algorithm. Retrieved February 6, 2023, from https://xlinux.nist.gov/dads/HTML/greedyalgo.html

Black, P. E. (2020, November 24). Christofides algorithm. Retrieved February 21, 2023, from https://xlinux.nist.gov/dads/HTML/christofides.html

Carlson, S. C., & Hosch, W. L. (2022, December 27). *Graph theory | Problems & Applications | Britannica*. Encyclopedia Britannica. Retrieved February 5, 2023, from https://www.britannica.com/topic/graph-theory

Chase, C., Chen, H., Noah, A., & Wilder-Smith, M. (n.d.). *An Evaluation of the Traveling Salesman Problem*.

Chatting, M. (2018). A Comparison of Exact and Heuristic Algorithms to Solve the Travelling Salesman Problem. *The Plymouth Student Scientist*, *11*(2), 53-91.

Christine, S. (2020, June 15). *Kruskal's Minimum Spanning Tree Algorithm*. Tutorialspoint. Retrieved March 16, 2023, from https://www.tutorialspoint.com/Kruskal-s-Minimum-Spanning-Tree-Algorithm

Fulber, V. (2022, November 6). *Graph Theory: Path vs. Cycle vs. Circuit*. Baeldung. Retrieved March 8, 2023, from https://www.baeldung.com/cs/path-vs-cycle-vs-circuit

Gao, Y. (2020, February 14). *Heuristic Algorithms for the Traveling Salesman Problem | by Opex Analytics | The Opex Analytics Blog*. Medium. Retrieved November 24, 2022, from https://medium.com/opex-analytics/heuristic-algorithms-for-the-traveling-salesman-problem-6a53d8143584

GeeksforGeeks. (2022, January 7). *Mathematics | Matching (graph theory)*. GeeksforGeeks. Retrieved March 16, 2023, from https://www.geeksforgeeks.org/mathematics-matching-graph-theory/

GeeksforGeeks. (2023, February 6). *Eulerian path and circuit for undirected graph*. GeeksforGeeks. Retrieved February 21, 2023, from https://www.geeksforgeeks.org/eulerian-path-and-circuit/

GeeksforGeeks. (2023, February 17). *Merge Sort Algorithm*. GeeksforGeeks. Retrieved March 22, 2023, from https://www.geeksforgeeks.org/merge-sort/

Gendreau, M., Laporte, G., & Hertz, A. (1997, August 1). An Approximation Algorithm for the Traveling Salesman Problem with Backhauls. *Operations Research*, *45*(4).

Gilbert, S., & Halim, S. (2016, August 30). *CS4234: Optimisation Algorithms Lecture 4 TRAVELLING-SALESMAN-PROBLEM*.

GitHub. (2022, June 8). *Minimum Spanning Tree - Kruskal - Algorithms for Competitive Programming*. CP-Algorithms. Retrieved March 26, 2023, from https://cp-algorithms.com/graph/mst_kruskal.html

Great Learning. (2022). *Time Complexity: What is Time Complexity & its Algorithms?* Great Learning. Retrieved March 13, 2023, from https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/

Kao, M.-Y. (Ed.). (2008). *Encyclopedia of Algorithms*. Springer.

Khandelwal, V. (2023, February 20). *What is Merge Sort Algorithm: How does it work and its Implementation*. Simplilearn. Retrieved March 22, 2023, from https://www.simplilearn.com/tutorials/data-structure-tutorial/merge-sort-algorithm

Kuo, M. (2020, January 1). *Solving The Traveling Salesman Problem For Deliveries*. Routific Blog. Retrieved February 5, 2023, from https://blog.routific.com/blog/travelling-salesman-problem

Lehman, E., Thomson, F., & Meyer, A. R. (2021, June 29). *11.1: Vertex Adjacency and Degrees*. Engineering LibreTexts. Retrieved March 16, 2023, from https://eng.libretexts.org/Bookshelves/Computer_Science/Programming_and_Computation_Fundamentals/Mathematics_for_Computer_Science_(Lehman_Leighton_and_Meyer)/02%3A_Structures/11%3A_Simple_Graphs/11.01%3A_Vertex_Adjacency_and_Degrees

LTA. (2022, November 9). *Rail Network - Singapore*. Land Transport Authority. Retrieved March 6, 2023, from https://www.lta.gov.sg/content/ltagov/en/getting_around/public_transport/rail_network.html

OpenAI. (2021). ChatGPT [Computer software]. Retrieved March 13, 2023, from https://openai.com/

OpenAI. (2021). ChatGPT [Computer software]. Retrieved March 25, 2023, from https://openai.com/

OpenAI. (2021). ChatGPT [Computer software]. Retrieved March 27, 2023, from https://openai.com/

Piterman, S. (2020, November 29). *Breaking Down MergeSort. And Understanding O(N log N) Time… | by Sergey Piterman | Outco*. Medium. Retrieved March 25, 2023, from https://medium.com/outco/breaking-down-mergesort-924c3a55c969

Quinn, C., Sangwin, C. J., Haese, R. C., Haese, M., & Blythe, P. (2014). *Mathematics for the International Student: Mathematics HL (option) : Discrete Mathematics, HL Topic 10, FM Topic 6, for Use with IB Diploma Programme*. Haese Mathematics.

Rosenkrantz, D., Stearns, R., & Lewis, P. M. (1977, September 1). An Analysis of Several Heuristics for the Traveling Salesman Problem. *Siam Journal on Computing*, *6*(3). 10.1137/0206041

S, R. A. (2023, February 13). *Kruskal Algorithm: Overview & Create Minimum Spanning Tree*. Simplilearn. Retrieved February 20, 2023, from https://www.simplilearn.com/tutorials/data-structure-tutorial/kruskal-algorithm

Takhirov, Z. (2017, July 19). *Hungarian Algorithm – z-Scale*. z-Scale. Retrieved March 16, 2023, from http://zafar.cc/2017/7/19/hungarian-algorithm/

TechTarget. (2020, June). *What is traveling salesman problem (TSP)? - Definition from WhatIs.com*. TechTarget. Retrieved November 21, 2022, from https://www.techtarget.com/whatis/definition/traveling-salesman-problem

W, W. E. (n.d.). *Graph Loop -- from Wolfram MathWorld*. Wolfram MathWorld. Retrieved February 21, 2023, from https://mathworld.wolfram.com/GraphLoop.html

Weisstein, E. (2000). *Graph Cycle -- from Wolfram MathWorld*. Wolfram MathWorld. Retrieved February 5, 2023, from https://mathworld.wolfram.com/GraphCycle.html

Weisstein, E. (2001). *Complete Graph -- from Wolfram MathWorld*. Wolfram MathWorld. Retrieved March 16, 2023, from https://mathworld.wolfram.com/CompleteGraph.html

Weisstein, E. W. (2003). *Hamiltonian Cycle -- from Wolfram MathWorld*. Wolfram MathWorld. Retrieved March 26, 2023, from https://mathworld.wolfram.com/HamiltonianCycle.html