The Ant Colony Optimisation for the Travelling Salesman Problem

Dylan Galea

November 26, 2018

Contents

1 Introduction

Before defining the Travelling Salesman Problem and proving properties about it, a number of graph theoretic concepts that will be used throughout, must first be defined. Therefore, what follows is a sub-section that introduces a number of graph theoretic concepts which are required for the Travelling Salesman Problem.

1.1 Some Graph Theory

Graph theory is the study of a structure called a graph. A graph can be defined formally as shown in definition 1.1.1 below.

Definition 1.1.1. *A graph G is a pair (V,E) were V is any non empty finite set called the set of vertices of G, and* $E \subseteq \{\{u,v\} : \forall\ u,v \in\ V \text{ and } u \neq v\}$ *is called the set of edges of G* [1]*. A graph G defined by the pair (V,E) is denoted by G(V,E) or G.*

A graph defined using definition 1.1.1 is called an undirected graph. There is also the concept of a directed graph were $E \subseteq \{(u,v) : \forall u, v \in V, u \neq v\}$ [1]. However, in this thesis it can be assumed that any graph that will be considered is undirected unless otherwise stated. It can also be assumed that there are no edges between same vertices unless otherwise stated. It must also be noted that by this definition, there cannot be multiple edges joining any 2 vertices. The reason is that sets do not allow repetition of elements. Thus, each element in the edge set is unique. The discussion will now proceed by introducing more graph theoretic terminology, with examples that illustrate these terminologies.

When two vertices are joined by an edge, they are said to be adjacent. This is defined formally in definition 1.1.2 below.

Definition 1.1.2. *Given a graph G(V,E),* $\forall\ u,v \in\ V$*, u and v are said to be adjacent if* $\{u,v\} \in E$*. Also, if u and v are adjacent, then u and v are said to be end vertices of the edge* $\{u,v\}$*.* [2]

There is also a special name associated with the case when a vertex is an end vertex of an edge.

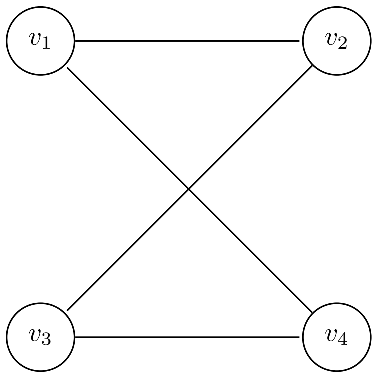
Definition 1.1.3. *A vertex v in a graph G is incident to an edge e in G if v is an end vertex of e.* [2]

It is sometimes also required to know how many edges are incident to a specific vertex in a graph.

Definition 1.1.4. *The degree of a vertex v in a graph G denoted by deg(v) is, the number of edges incident with v in G.* [2]

Any graph $G(V,E)$ can also be represented pictorially by drawing the vertices of G using circles, and by drawing the edges of G using lines between adjacent vertices. As a result, in this thesis a graph is sometimes given formally using sets or as a pictorial representation, assuming that one can be converted into another. Example 1.1.5 below depicts how a graph can be represented pictorially.

Example 1.1.5. Consider the graph *G(V,E)* such that V=$\{v_1, v_2, v_3, v_4\}$ and $E = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_1\}\}$.
Then G can be represented pictorially as :



Using definition 1.1.2, two adjacent vertices in G are v_1 and v_2. On the other hand, two non adjacent vertices in G are v_1 and v_3.
By definition 1.1.3, v_1 and v_2 are incident to the edge $\{v_1, v_2\}$.
Using definition 1.1.4, the degree of every vertex in G is 2.

There are many other examples of graphs, one of them being the complete graph on n vertices.

Definition 1.1.6. *A graph G(V,E) is said to be complete if* $\forall$ *v,w* $\in$ *V* $v \neq w$, *v is adjacent to w. The complete graph on n vertices is denoted by* K_n. [2]

Given any graph $G(V, E)$, one can also define graph theoretic structures that lie within G, one of them being a path.

Definition 1.1.7. *Given a graph G(V,E), a path in G joining any 2 vertices u, v* $\in$ *V, is a sequence of vertices* $u = u_1$, u_2, ...,u_n $= v$ *in which no vertex is repeated and,* $\forall$ $0 < i < n$, $\{u_i, u_{i+1}\} \in E$. [3]

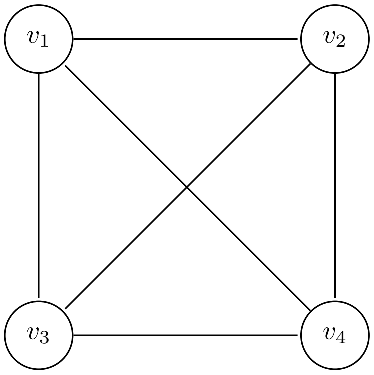
Definition 1.1.7 can now be used to define cycles and connectivity in a graph.

Definition 1.1.8. *A graph G(V,E) is said to be connected if* $\forall$ *u,v* $\in$ *V* $u \neq v$, *u and v are joined by a path.* [2]

Definition 1.1.9. *Given a graph G(V,E), a cycle in G is a path on* $n \geq 4$ *vertices, such that, the first vertex and the last vertex are equal.* [4]

By definitions 1.1.7, 1.1.9 above, it is clear that a cycle is a special instance of a path, with the only difference being that in a cycle, the first vertex and the last vertex are equal. Another thing worth mentioning is that, according to definitions 1.1.7 and 1.1.9, cycles and paths are sequences of vertices and not actual graphs. However, this is not the case because they can be represented easily as graphs. For example, given the path/cycle $u_1, u_2, ..., u_n$ a new graph $G(V, E)$ can be created such that, $V = \{u_1, u_2, ..., u_n\}$ and $E = \{\{u_i, u_{i+1}\} : \forall i, 0 < i < n\}$. For example, consider the cycle v_1, v_2, v_3, v_4, v_1, then the graph depicted in example 1.1.5, is the graph representing this cycle. Such graphs are known as Cycle/Path graphs and are denoted by C_n/P_n respecitvely, n being the number of vertices in the graph. Since this construction can be done, cycles/paths will be treated as both graphs and sequences. This will later be useful when defining Hamiltonian cycles. For better understanding of definitions 1.1.6, 1.1.7, 1.1.8 and 1.1.9, example 1.1.10 is constructed.

Example 1.1.10. Consider the graph *G(V,E)* below:



Since every vertex in G is adjacent to every other vertex, G must be complete. Therefore G must be K_4. Since G is complete, it must also be connected because, there is a path P_2 between any two distinct vertices of G.
Some examples of paths in G are:
1. v_1 v_2 v_3 v_4
2. v_1 v_4
3. v_4 v_3 v_1
Some examples of cycles in G are:
1. v_1 v_2 v_3 v_4 v_1
2. v_1 v_4 v_2 v_3 v_1
3. v_4 v_3 v_1 v_4

Another important graph theoretic concept is that of subgraphs.

Definition 1.1.11. *Given a graph G(V,E) and a graph H(V′,E′), H is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$.* [2]

Clearly by definition 1.1.11 and the construction of Cycle/Path graphs, if A is a cycle/path in G then the Cycle/Path graph representing A is a subgraph of G. This leads to defining the concept of two distinct path/cycles. Two paths/cycles in G are said to be distinct if, when they they are constructed as Path/Cycle subgraphs of G they differ in at least one edge. After defining some important concepts, the next step is to extend definition 1.1.1 to define another class of graphs called weighted graphs. It must be noted that all definitions presented so far apply also to weighted graphs.

Definition 1.1.12. *Given a graph G(V,E), a weight function is a function* $f : E \mapsto \Re^+$ [2]. *The real numbers assigned to each edge are called weights.*

Note that in definition 1.1.12, the weights are taken to be positive. According to [2], there could be cases were negative weights would be appropiate. However, unless otherwise stated, it is to be assumed that when considering a weight function, the weights are positive.

Definition 1.1.13. *A weighted graph is a graph G(V,E) with a weight function* f [2]. *This is denoted by the triple G(V,E,f) or G.*

According to Bondy and Murty [5], weighted graphs occur regularly in applied graph theory. For example, a railway network can be represented by a weighted graph were, the vertices are the set of towns in the railway network, and there are edges between 2 vertices in the graph if, there is a direct route from one town to another, without visiting other towns in the process. The weight function would then represent the cost of travelling directly from one town to another. In addition to that, the shortest path between 2 towns in the network may be required. It is clear that in order to try and solve such problems, the total weight of a subgraph must first be defined.

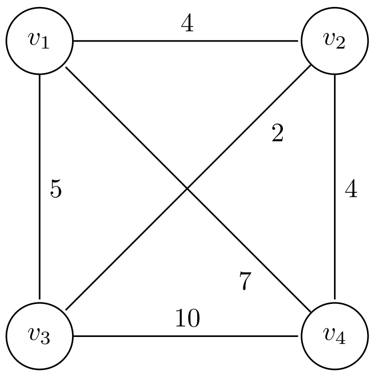
Definition 1.1.14. *Given a weighted graph G(V,E,f), the total weight of any subgraph H(V′,E′,f) of G is:*

$$\sum_{e \in E'} f(e)$$

. [5]

It is important to note that by definition 1.1.11, any weighted graph G is a subgraph of itself, therefore, it's weight can be calculated. This is highlighted in Example 1.1.15 below.

Example 1.1.15. Consider the weighted graph *G(V,E,f)* such that, *G(V,E)* is the graph in example 1.1.10 with weight function f such that,
$f(\{v_1, v_2\}) = 4$
$f(\{v_1, v_3\}) = 5$
$f(\{v_2, v_3\}) = 2$
$f(\{v_3, v_4\}) = 10$
$f(\{v_2, v_4\}) = 4$
$f(\{v_4, v_1\}) = 7$
Then by definition 1.1.13, the graph below is a weighted graph.



Also according to definition 1.1.11, the above graph is a subgraph of itself. Therefore it's weight can be calculated, where by definition 1.1.14, the weight of G is 32.

According to Guichard [6], trees are another useful class of graphs.

Definition 1.1.16. *A tree is a connected graph with no cycles.* [6]

Having defined the basic graph theoretic concepts, it is now time to define harder concepts that use previous definitions. It is important to note that the following concepts can be applied to both weighted and unweighted graphs. Therefore, in the remaining definitions the graph being considered can either be weighted or unweighted.

Definition 1.1.17. *$H(V',E')$ is a spanning subgraph of $G(V,E)$ if H is a subgraph of G and $V' = V$.* [7]

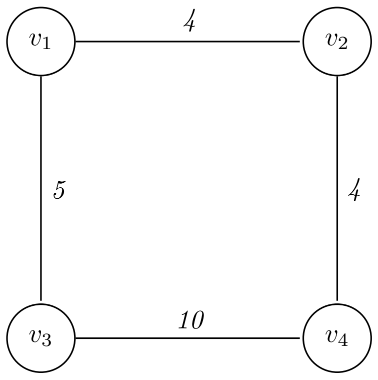
There are many spanning subgraphs, however the ones that are relevant to this thesis are spanning trees and spanning cycles, the latter mostly known as Hamiltonian cycles.

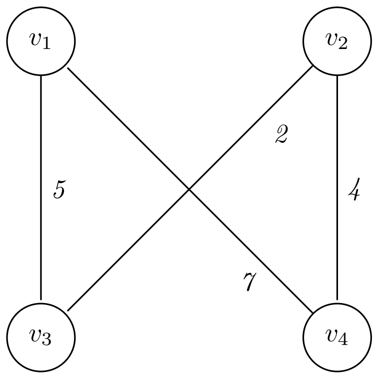
Definition 1.1.18. *A graph H is a spanning tree of G if H is a tree and H is a spanning subgraph of G.* [7]

Definition 1.1.19. *Given a graph G, C is a Hamiltonian cycle in G if C is a cycle and C is a spanning subgraph of G. Also, a graph that contains a Hamiltonian cycle is called a Hamiltonian graph.* [7]

It is worth mentioning that definition 1.1.19 holds because, cycles can be represented by Cycle graphs due to the construction discussed earlier. What follows now is an example that illustrates better definitions 1.1.17, 1.1.18 and 1.1.19.

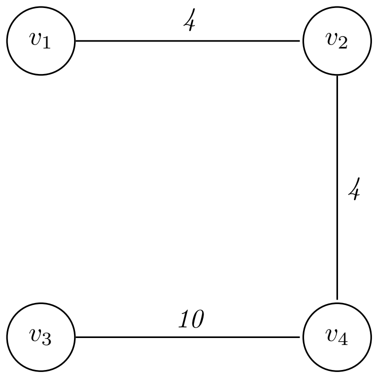
Example 1.1.20. Let G be the graph in example 1.1.15. Then, according to definition 1.1.17, the two graphs below are two spanning subgraphs of G because, they contain all the vertices of G and are subgraphs of G .

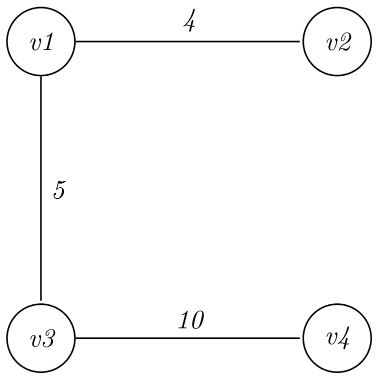




It must also be said that by definition 1.1.19, the above two graphs are Hamiltonian cycles in G because, they are spanning subgraphs of G and are Cycle sub-graphs of G. Since the above graphs are subgraphs of G, by definition 1.1.14, their weight can be calculated by summing up the weights of the edges. Thus, the Hamiltonian cycles above have weight 23 and 18 respectively.

Given the same graph G in example 1.1.15, the two graphs below are spanning trees of G of weight 18 and 19 respectively.





This example also shows that within the same weighted graph, there could be multiple Hamiltonian cycles and spanning trees of different weight.

Having defined Hamiltonian cycles and spanning trees, it is natural to ask whether there are necessary and sufficient conditions in a graph that guarantee it is Hamiltonian or that it contains a spanning tree as subgraph. In fact, theorem 1.1.21 gives a necessary and sufficient condition for a graph to have a spanning tree.

Theorem 1.1.21. *A graph G has a spanning tree $\iff$ it is connected* [7].

Proof. ($\implies$) Let $G(V, E)$ be a graph having a spanning tree $T(V', E')$ as one of it's subgraphs. Let $v1, v2 \in V$. Since, T is a spanning tree of G, then, T is a spanning subgraph of G. Thus, $v1, v2 \in V'$. Also, since T is a tree, T must be connected. Therefore, $\exists$ a path P joining vertices v_1 and v_2 in T. But since T is a subgraph of G, then P is also a path in G. Therefore G must be connected.
($\Leftarrow$) Conversely, let $G(V, E)$ be a connected graph. Then, if G has no cycles, G itself must be a spanning tree. If G has cycles, delete an edge from a cycle in G. Clearly, the resultant graph is still connected and contains one less cycle. Repeat this procedure untill no more cycles are left in the graph. Then, the resultant graph G' would be a connected subgraph of G having no cycles(i.e a tree). Also, since by the deletion procedure, no vertex was deleted from G, G' is a spanning subgraph of G. Therefore G' is a spanning tree of G. [7] □

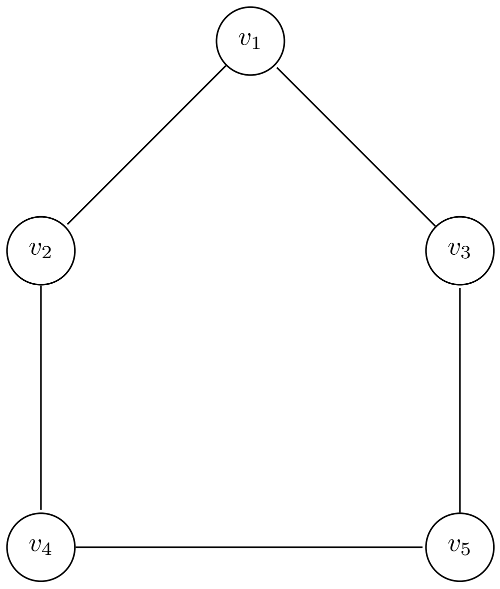
Theorem 1.1.21 confirms that for a graph to have a spanning tree, the graph must be connected and vice-versa. Thus, for spanning trees, the necessary and sufficient condition is connectivity. However, the same cannot be said about Hamiltonian cycles because, no necessary and sufficient conditions are known for a graph to be Hamiltonian. In fact, there are sufficient conditions for a graph to be Hamiltonian, however, these conditions are not necessary. There are also necessary conditions, some of which are trivial, such as, if G is Hamiltonian then G must be connected, but this is not necessary and sufficient. According to Guichard [6], these sufficient conditions typically say that for a graph to be Hamiltonian it must have a lot of edges. But it is also argued in [6], that these conditions are not necessary, because, there are Hamiltonian graphs that have few edges. For example, C_n has only n edges but is Hamiltonian. One such sufficient but not necessary condition for Hamiltonianicity is Ore's Theorem below.

Theorem 1.1.22 (Ore's Theorem)**.** *Let G be a graph on $n \geq 3$ vertices such that if v and w are not adjacent in G then $deg(v) + deg(w) \geq n$. Then G is Hamiltonian.* [7]

Proof. Suppose that $G(V, E)$ is a graph satisfying all the conditions in the theorem statement but is not Hamiltonian. Then since G is not Hamiltonian and K_n is Hamiltonian, G must be a subgraph of K_n having fewer edges than K_n. Therefore, add edges to G between non adjacent vertices to obtain a subgraph $H(V', E')$ of K_n such that adding an edge to H would create a subgraph of K_n which is Hamiltonian. Let $u, v \in V'$ be 2 non-adjecent vertices in H. Since by construction G is a subgraph of H, u, v must be non-adjcent in G. Therefore $deg(u) + deg(v) \geq n$ in both G and H. Since adding an edge to H creates a resultant graph that is Hamiltonian, then, adding an edge between u and v creates a Hamiltonian graph. Therefore, in H there must be a path joining u and v containing all the vertices of H. Let the path be $u = v_1, v_2, ..., v_n = v$. Now suppose $deg(v_1) = \alpha$ in H. Now $\forall i, 1 < i < n$, if there is an edge between u_1 and u_i in H, then there must not be an edge between u_{i-1} and u_n because, $u_1, u_i, u_{i+1}, ..., u_n, u_{i-1}, u_{i-2}, ..., u_1$ would be a Hamiltonian cycle in H, thus H would be Hamiltonian. Therefore, $deg(u_n) \leq n - 1 - \alpha$
$\implies deg(u_1) + deg(u_n) \leq \alpha + n - 1 - \alpha$ in H
$\implies deg(u_1) + deg(u_n) \leq n - 1$ in H
$\implies deg(u_1) + deg(u_n) < n$ in G since G is a subgraph of H.
This contradicts the assumption that $deg(u_1 = u) + deg(u_n = v) \geq n$ in G
Therefore, G must be Hamiltonian. [7] □

It is important to note that the above proof uses the fact that K_n is Hamiltonian. This is true because any C_n is always a subgraph of K_n. Therefore any C_n that spans K_n is a subgraph of K_n. Example 1.1.23 below is a counter example to show why Ore's theorem gives a sufficient but not necessary condition.

Example 1.1.23. Consider the graph C_5 *below.*



C_5 above is Hamiltonian because it contains the Hamiltonian cycle $v_1, v_2, v_4, v_5, v_3, v_1$. However, *deg*($v_1$) + *deg*($v_5$) = $4 < 5 = n$. Therefore, the condition in Ore's Theorem is not a necessary condition.

To conclude, these facts seem to indicate that determining whether a graph is Hamiltonian is a very difficult problem and that there are certain problems that are harder than other problems. In order to reason about such problems, some computational theory must first be established. This is done in the next subsection.

1.2 Some Computational Theory

After defining some important graph theoretic concepts, it is now time to present some important computational theory that will be used in later sections. The discussion will now proceed by defining two different types of problems and describing the relationship between them.

Definition 1.2.1. *Suppose that a problem P has many possible solutions such that each solution has an associated value. Then P is an optimisation problem if P requires to find the solution with the best value.* [8]

Therefore, according to definition 1.2.1, the aim of the algorithm solving an optimisation problem would then be to find the solution with the best (min/max) value. An maximisation problem is an optimization problem that asks to find the solution with the maximum value. On the other hand, a minimisation problem is an optimisation problem that asks to find the solution with the minimum value. The solution with the best value is called an optimal solution to the problem, and it's value is known as the optimal value. An optimal solution is not termed as 'the' optimal solution because, there could be many optimal solutions having the same optimal value. [8]

Another type of problems is called decision problems.

Definition 1.2.2. *Decision problems are problems whose solutions are either a yes or a no.* [8]

In [8] it is argued that, given an optimisation problem, one can transform this optimisation problem into a decision problem such that, the decision problem is easier than the optimisation problem. Therefore, if some optimisation problem is easy to solve, the decision problem version is easy to solve as well [8]. On the other hand, if the decision problem is hard to solve then this would mean that the original optimization problem is also hard to solve [8]. Note that the notion of 'hard' and 'easy' problems will be defined rigorously later in this section. This concept was only mentioned here to show that there exists a relationship between optimisation problems and decision problems. Before going into harder computational theory, an example is given to undestand how an optimisation problem can be transformed into a related decision problem.

Example 1.2.3. Two optimisation problems are:
1. The shortest path problem
2. The minimum weight spanning tree problem
Given a connected weighted graph G, the shortest path problem is the task of finding a path P between two vertices in G such that, the weight of P is the minimum amongst all paths joining these two vertices in G [9]. On the other hand, given a connected weighted graph G the minimum weight spanning tree problem is the task of finding a spanning tree in G of minimum weight [10]. The shortest path problem is an optimsation problem because, from all the possible paths in the graph, a path having the optimal value (minimum weight) is chosen. In this case the best path is the path with minimum weight. Similarly, using the same reasoning, the minimum weight spanning tree problem is an optimisation problem.
The decision problems related to these optimisation problems can be obtained by specifying a bound on the value to be optimized [8]. Therefore, the decision problems related to the shortest path problem, and the minimum weight spanning tree problem defined in this example are:
1.Given a connected weighted graph *G,* two vertices u and *v,* and an integer *k,* does a path exist from u to v in G of weight at most *k?*
2.Given a connected weighted graph *G,* and an integer k, does G contain a spanning tree of weight at most *k?*
The above two problems are decision problems because the answer to these problems is either a yes or a no.

Reasoning about computational problems is usually done by analyzing the properties belonging to the algorithms that solve these problems. One property that is analyzed is called the running time of an algorithm. This is defined in definition 1.2.4 below.

Definition 1.2.4. *Given an input I, the running time of an algorithm A is the number of steps executed by A given I. The running time of an algorithm A can be represented by the function $T : \mathbb{N} \mapsto \mathbb{N}$ were the domain represents the size of the input, and the co-domain represents the amount of steps taken by the algorithm to execute given an input size.* [8], [11]

The following is an example on how the running time of an algorithm can be constructed.

Example 1.2.5. Consider an algorithm that performs the addition of two n-bit strings. Then if the addition of 2 bits takes a computational steps, the addition of 2 $n-$bit strings will take $a \times n$ steps, since, each computation must be performed n times (i.e on each bit in the string). Therefore, the running time of this algorithm for an input of size n can be represented by the function, *T(n)* $= a \times n$. [8], [11]

Expressing the running time in this way seems pretty easy to understand. However, it has problems. Typically, the running time of an algorithm is fixed for any input, however, the running time function is not defined on the actual inputs, but, on the input sizes. As a result, since the same input size might represent different inputs, there could be different running times for the same input size n, as this depends on which input is given. This means that for any input size n, there are different running time functions giving different running times. These different running time functions represent the different cases that could arise in an algorithm given any input size n. The most important runnning time functions are the worst-case running time, the best-case running time and the average-case running time. The worst/best/average case running time are the longest/shortest/average running time of an algorithm given any input of size n respectively. Therefore the problem is what running time function should be used to reason about algorithms. Although the average-case running time generally has a greater practical significance, it is often very difficult to compute. Therefore, the worst-case running time is usually used, that is, finding the running time function that gives the worst-case running time for any input of size n. The worst-case running time of an algorithm is an upper bound to all other running times, therefore, this assures that the algorithm will never get a longer running time for any input of size n. So, if the worst case running time is polynomial in the input size, then we know that the problem is in class P (These terms will be defined below). [8], [11]

Another problem is that, the running time of an algorithm can be difficult to compute precisely because it depends on the computer the algorithm is executed on. This informally means that, an algorithm may need different amount of computational steps on different computers for the same input. For example, the number of steps taken for adding 2-bits as seen in example 1.2.5, may be different on different computers, resulting into running time functions that represent the same worst/best/average running time case, differing only in multiplicative constants and low order terms. To get around this problem, a mechanism that identifies a common property between all running time functions is needed. This mechanism is called asymptotic analysis. This means making the input size increase without bound so that only the order of growth of the running time function is relevant. This works because, as the input size increases, the low-order terms and multiplicative constants of the running time function get dominated by the size of the input itself. This means that, given any running time function, the low-order terms and multiplicative constants of the function can be ignored leaving only the order of growth (which is common across all running time functions representing the same best/worst/average running time case). In a running time function, the order of growth is given by the highest order term. This is because, the highest order term has the greatest effect on the running time function as the input size increases. For example, given the running time function $T(n) = an^2 + bn + c$, n^2 is the term that increases the running time the most as the input size increases without bound. [8], [11]

The following example is used to explain better what has been discussed in the previous paragraph.

Example 1.2.6. Suppose that the n-bit addition algorithm in example 1.2.5, has a worst-case running time function $T(n) = a \times n$ on computer A and a worst-case running time function $T(n) = b \times n$ on computer B where, a/b represent the number of computational steps required to perform 2-bit addition on computers A/B respectively. The asymptotic analysis of the algorithm concludes that, the n-bit addition algorithm has an n^2 asymptotic behaviour. Note that this is common to both running time functions. [8], [11]

To express the asymptotic behaviour of functions mathematically, asymptotic notations are normally used. One of these notations is called the Big O Notation, whose exact definition is given below.

Definition 1.2.7. *Let $f : \mathbb{N} \mapsto \mathbb{N}$ and $g : \mathbb{N} \mapsto \mathbb{N}$ be two monotonic functions. Then $f(n) = O(g(n))$ if $\exists\ c > 0$ and $n_0 > 0$ such that $f(n) \leq cg(n)\ \forall\ n \geq n_0$.* [11]

Definition 1.2.7 implicitly states that when writing $f(n) = O(g(n))$, $g(n)$ is an upperbound to $f(n)$ as the input size n grows without bound [11]. Therefore, if $f(n)$ is the worst case running time function of an algorithm and $f(n) = O(g(n))$, then it is guaranteed that as $n \to \infty$ the algorithm will never get a running time longer than $g(n)$ times some constant. Note that definition 1.2.7 can be used even for the asymptotic analysis of the average and best case running times. However, unless otherwise stated, it can be assumed that in this thesis Big O Notation will be used for the worst-case asymptotic analysis of algorithms. The following is a continuation to example 1.2.4.

Example 1.2.8. By example 1.2.4, the algorithm that performs two n-bit addition has a running time function $T(n) = an$. Therefore, by definition 1.2.7, $T(n) = O(n)$ because, since a is a natural number $\exists\ z \geq a$ such that, $an \leq zn\ \forall\ n \in \mathbb{N}$. Therefore at each step, the order of growth of $T(n)$ is linear with respect to the input size n.

It is important to note that if $T(n) = O(g(n))$, then, $T(n) = O(h(n))$ if $h(n)$ has a higher order of growth than $g(n)$. For example, in example 1.2.7 $T(n) = O(n^2)$ as well since, n^2 is an upperbound to n therefore, it is an upperbound to $T(n)$. However, unless otherwise stated, it can be assumed that whenever $T(n) = O(g(n))$, $g(n)$ is a tight asymptotic bound to $T(n)$. A tight asymptotic bound to $T(n)$ is a function $g(n)$ such that, if $T(n) = O(f(n))$ then $g(n) = O(f(n))$.

So we have the following definition.

Definition 1.2.9. *Given a function g(n) and an algorithm A having a running time function T(n), A is said to have a time complexity of O(g(n)) if T(n) = O(g(n)).* [11]

Definition 1.2.9 does not apply only to the worst case running time function. However, recall that when discussing running time functions it was said that it can be assumed that, the worst-case running time function is taken. Therefore, it can be assumed that the worst-case running time function is taken when saying that an algorithm has an $O(g(n))$ time complexity, or when saying that an algorithm is an $O(g(n))$ algorithm. This will sometimes also be termed as, the worst-case time complexity of the algorithm. Note that when it is said that a problem can be solved in $O(g(n))$ time, this means the problem can be solved by an algorithm that has an $O(g(n))$ worst-case time complexity.

The following table summarizes some of the most common time complexities encountered in computational theory, and what they are commonly known as. The table was adapted using information from [12] and [13].

Table 1: Time Complexities and Terminology Used.

Time Complexity	**Terminology Used**
O(1)	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n \log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(n^c)$, $c > 0$ constant	Polynomial time
$O(a^n)$, $a \geq 2$ constant	Exponential time
$O(n!)$	Factorial time

In Table 1, the time-complexities are ordered in ascending order of growth [12]. This means that, for example, constant time algorithms have a smaller order of growth than logarithmic time algorithms. It must be noted that the higher the order or growth, the more steps the algorithm needs to execute for any input and therefore, the less efficient it is. In general, problems that can be solved in polynomial time are tractable problems [8]. On the other hand, problems that require superpolynomial (exponential/factorial) time algorithms to be solved are intractable problems [8]. In general, tractable problems are considered to be easy to solve, whereas intractable problems are hard to solve. Table 2 below is constructed to show why problems that can be solved in polynomial time are tractable, whilst, problems that cannot be solved in polynomial time are intractable. Table 2 gives the execution time of any algorithm given an input of size *10*, *100* or *1000*, provided that the algorithm's time complexity is $O(n)$, $O(n^2)$, $O(2^n)$ or $O(n!)$.

Table 2: Execution Time Given Input Size and Time Complexity

Input Size / **Time Complexity**	**10**	**100**	**1000**
$O(n)$	1×10^{-8} *seconds*	1×10^{-7} *seconds*	1×10^{-6} *seconds*
$O(n^2)$	1×10^{-7} *seconds*	1×10^{-5} *seconds*	1×10^{-3} *seconds*
$O(2^n)$	1.024×10^{-6} *seconds*	1.2676×10^{21} *seconds*	Very large to compute
$O(n!)$	3.6288×10^{-3} *seconds*	Very large to compute	Very large to compute

Table 2 above was adapted from [14]. In Table 2 above it is assumed that the machine the algorithms can be executed on perform 10^9 computational steps per second [14]. It can be concluded from Table 2 above that, as the size of the input increases, only the polynomial functions have small values. This means that a polynomial time algorithm does not have a very large execution time even if the size of the input is increased substantially. On the other hand, an algorithm with an exponential or factorial time complexity has a very large execution time, because, the rate of growth of the time complexity function is very large.

The following is an example to wrap-up all the time complexity concepts discussed so far.

Example 1.2.10. In example 1.2.8, *T(n) = O(n).* Therefore according to definition 1.2.9, the n-bit addition algorithm has a time complexity of *O(n).* Therefore by Table 1, the n-addition algorithm has a linear time complexity, or simply is a linear-time algorithm. As a result, the n-bit addition problem can be solved in linear time.

In what has been presented so far in this section it has always been assumed that, any problem has an algorithm that solves it. However, this is not the case because there are computational problems that cannot be solved by any computer, no matter how much processing power and time is dedicated to them. In addition to that, there are problems that can be solved in polynomial time and others which cannot. This seems to indicate that there are several classes of problems. One of these classes is the class P defined below. [8]

Definition 1.2.11. *The class P consists of all the problems that can can be solved in polynomial time.* [8]

Therefore according to Definition 1.2.11, the class P consists of tractable problems. An example of a problem in the class P is the n-bit addition problem defined in Example 1.2.5. The n-bit addition problem is in the class P because, it can be solved in linear time (as discussed in example 1.2.10). Apart from the class P, there is also the class NP. The following two definitions help in defining the class NP.

Definition 1.2.12. *A certificate of a solution is an object that guarantees the solution.* [15]

It can be deduced from Definition 1.2.12 that if a certificate is correct, then the solution is correct. Therefore, one can verify if a solution is correct by actually verifying that the certificate is correct. Note that Definition 1.2.12 is not the exact mathematical definition of a certificate. The exact mathematical definition of a certificate is not given because, it requires discussing concepts which are beyond the scope of this project.

Definition 1.2.13. *Suppose that S is an arbitrary solution to a problem A. Then, A can be verified in polynomial time if given a certificate C of S, $\exists$ a polynomial time algorithm such that, when supplied an instance (input) of A and C as inputs, the algorithm can verify if C is correct.* [8]

In other words, Definition 1.2.13 states that, a problem can be verified in polynomial time if the certificate that lead to the solution can be checked if it is correct by a polynomial time algorithm. Note that as discussed before, when checking that the certificate is correct, we are implicitly checking that the solution is correct. What follows now is an example that explains Definition 1.2.12 and Definition 1.2.13.

Example 1.2.14. To explain Definition 1.2.12 and Definition 1.2.13, the Hamiltonian Cycle problem is going to be used. Given a graph $G(V,E)$, the Hamiltonian cycle problem is the task of determing whether G has a Hamiltonian cycle [8]. A certificate of a solution to the Hamiltonian cycle problem can be a sequence of vertices $S = v_1,\ ...,\ v_n$ on $|V| = n$ vertices. Note that if the solution to the Hamiltonian cycle problem is 'yes then, the certificate would guarantee the solution if it is a Hamiltonian cycle in G. On the other hand, if the solution to the Hamiltonian cycle problem is a 'no', then, any certificate given guarantees the solution. The solution is guaranteed because, since there is no Hamiltonian cycle in G, there is no certificate that represent a Hamiltonian cycle in G. Therefore, for the Hamiltonian cycle to be verified in polynomial time, we must have a polynomial time algorithm that checks whether the certificate is a Hamiltonian cycle when the answer is 'yes'. This can be done by creating an algorithm that checks if $\forall\ i \in$ *[n-1]*, $\{v_i, v_{i+1}\} \in E$ *and* $\{v_n,\ v_1\} \in E$. Therefore, the checker algorithm has a worst case *O(n)* time complexity because, it must check all edges in the cycle. Hence, the checker algorithm takes polynomial time to execute. As a result, the Hamiltonian cycle problem can be verified in polynomial time.

Now the class NP can be defined.

Definition 1.2.15. *The class NP consists of all decision problems that can be verified in polynomial time.* [8], [16]

It is important to note that according to definition 1.2.15, a problem A is in the class NP if A is a decision problem. This will not be a limiting factor because, as discussed before, every optimization problem can be transformed into a related decision problem by specifying a bound on the solution to be optimized (see example 1.2.3). Reasoning about decision problems is easier because, the related decision problem is not harder than the original optimisation problem. As a result, one can prove that an optimisation problem cannot be solved in polynomial time by showing that, the related decision problem cannot be solved in polynomial time. [8]

Another important class of problems is the class of NP-Complete problems. To define this class, a reduction algorithm must first be defined.

Definition 1.2.16. *Suppose that A and B are two decision problems. A reduction algorithm is an algorithm that transforms any instance α of A into an instance β of B with the following properties:*

- *The algorithm takes polynomial time to execute*
- *The solution to α is yes $\iff$ the solution to β is yes.* [8]

A reduction algorithm has important applications in computational theory. In fact, suppose that there are two decision problems A and B such that an instance α of the problem A needs to be solved in polynomial time. Suppose also that, B can be solved in polynomial time using algorithm $B^\star$ and there exists a reduction algorithm R from A to B. Then A can be solved in polynomial time in the following way :

- Use R to transform α into an instance β of B
- Execute $B^\star$ on the input β to get answer γ = yes/no
- Give γ as an answer for α.

It is important to note that A can be solved in polynomial time because, the total time of the above procedure is polynomial. The above procedure is polynomial because, the total time of executing two polynomial time algorithms is the summation of two polynomials, which by properties of polynomials is still polynomial. It is important to note that, the above could only be done because the solution to α is yes $\iff$ the solution to β is yes. [8]

Another important application of reduction algorithms is to show that a polynomial time algorithm does not exist for a particular decision problem. Let A and B be decision problems such that A cannot be solved in polynomial time. Suppose that $\exists$ a reduction algorithm R from A to B. Then it can be shown that B cannot be solved in polynomial time because, if B can be solved in polynomial time, then R can be used to transform any instance α of A into an instance β of B were again, α can be solved in polynomial time as discussed in the previous paragraph. Since α is arbitrary, A can be solved for all inputs in polynomial time and hence, A can be solved in polynomial time. This contradicts the assumption that A cannot be solved in polynomial time. Therefore, when using a reduction algorithm from a decision problem A to a decision problem B, one implicitly makes the statement that decision problem B is as hard (in terms of tractability as discussed before) as decision problem A. If not, this would contradict properties that belong to the problem A. [8]

After defining reduction algorithms, the class of NP-Complete problems can now be defined.

Definition 1.2.17. *A decision problem A is in the class NP-Complete (NPC) if A is in NP and A is as hard as any problem in NP.* [8]

In other words, definition 1.2.17 states that for a problem A to be in NPC, A must be in NP and that any problem in NP can be transformed using a reduction algorithm into A. Therefore, to show that a problem A is in NPC, one must show that it is in NP, and that every problem in NP can be reduced to A. Showing that A is in NP is not difficult because, it only requires to show that a given certificate of a solution of any arbitrary instance of A can be checked if it is correct in polynomial time. On the other hand, showing that every problem in NP is reducible to A may seem diffcult because, this requires checking that every problem in NP can be reduced to A [16]. However, this is not the case because reduction algorithms are transitive [16]. Hence, considering a known NP-Complete problem H, since H is NP-Complete, every problem in NP can be reduced to H. Therefore, if one can find a reduction algorithm from H to A, then by transitivity of reduction algorithms, every problem in NP can be reduced to A using a reduciton algorithm [16].

The classes P, NP and NPC have very important implications in computational theory. One of the most important problems in computational theory is whether P = NP. It is known that P $\subseteq$ NP. This is true because, since every problem in P can be solved in polynomial time, then implicitly the polynomial time verification has been done by the algorithm solving the problem. Therefore, every problem in P is in NP and hence, P $\subseteq$ NP. Note that this could be done because, every problem can be converted into a decision problem. Showing that NP $\subseteq$ P may seem to be easy as well. In fact, to show that NP $\subseteq$ P, one only needs to show that $\exists$ an NP-Complete problem C that can be solved in polynomial time, because, by the NP-Completness of C, every problem in NP can be reduced using a reduction algorithm to C and hence, every problem in NP could be solved in polynomial time since the reduction algorithm and the algorithm solving C, would take a total time that is still polynomial. However, no polynomial time algorithm has yet been discovered for any problem in NPC. In addition to this, no one has yet been able to prove that such polynomial time algorithms do not exist for NP-Complete problems. This essentially means that, the NP $\subseteq$ P problem is still an open question. [8]

After defining the important computational theory, it is now time to define the Travelling Salesman Problem, and present important properties about it. This will all lead to showing that the Travelling Salesman Problem is NP-Complete, hence, showing that no polynomial time algorithm has yet been discovered to solve the Travelling Salesman Problem. All of this will be done in the next section.

2 The Travelling Salesman Problem

After defining all the graph theoretic concepts and computational theory required, it is now time to define the Travelling Salesman Problem. The Travelling Salesman Problem is defined mathematically in Definition 2.0.1 below.

Definition 2.0.1. *Given a complete weighted graph G, the Travelling Salesman Problem (TSP) is the problem of finding a minimum weight Hamiltonian cycle in G.* [5]

According to Definition 2.0.1, a weighted graph G must be complete for the TSP to be solved given G. However, this is not a limiting factor because, every weighted graph G' can be converted into a complete weighted graph, by adding the missing edges in G' with a very large weight. Note that this can be done because, if G' is an incomplete weighted Hamiltonian graph, the minimum weight Hamiltonian cycle in G' would never be effected when transforming G' into a complete graph. The minimum weight Hamiltonian cycle can never be effected because, since the missing edges have a very large weight, they can never be part of the minimum weight Hamiltonian cycle when added. Therefore, since incomplete weighted graphs can be represented as complete weighted graphs, it can be assumed that complete weighted graphs are always being considered when discussing the TSP.

Definition 2.0.1 also specifies indirectly that, if G is a complete weighted graph then G is Hamiltonian. This is true because, any 2 distinct vertices in K_n are adjacent, therefore, every Hamiltonian cycle consisting of n vertices must be in K_n. Note that for the TSP, K_2 and K_1 are implicitly excluded because, by definition 1.1.9, cycles are defined on graphs having at least 3 vertices.

The following example is used to clearly explain what the TSP defined in Definition 2.0.1 is all about.

Example 2.0.2. Consider the complete graph K_4 with the weight function f defined in example 1.1.15. The distinct Hamiltonian cycles (defined in section 1.1) of G are:

- v_1, v_2, v_3, v_4, v_1 with weight *23*
- v_1, v_3, v_4, v_2, v_1 with weight *23*
- v_1, v_3, v_2, v_4, v_1 with weight *18*

Therefore, by definition 2.0.1, the solution to the TSP given the graph K_4 with weight function f would be, the minimum weight Hamiltonian cycle v_1, v_3, v_2, v_4, v_1. Note that K_4 has *3* distinct Hamiltonian cycles. This fact is a special case of lemma 2.0.3 for any K_n.

Example 2.0.2 suggests implicitly that, to solve the TSP on any complete weighted graph, one can do it by a brute-force algorithm. A brute-force algorithm for the TSP is one that computes all the Hamiltonian cycles in a graph and chooses the one of least weight as a soltution [17]. However, according to Krivelevich [18], K_n contains $\frac{(n-1)!}{2}$ distinct Hamiltonian cycles. Therefore, a brute-force algorithm is inefficient because, it must generate all the distinct Hamiltonian cycles in the graph taking factorial time to do so [17].

Lemma 2.0.3 below states that, the complete graph K_n on $n \geq 3$ vertices contains, $\frac{(n-1)!}{2}$ distinct Hamiltonian cycles. Therefore, it confirms that a brute-force algorithm cannot efficiently solve the TSP.

Lemma 2.0.3. *For $n \geq 3$, the complete graph on n vertices has $\frac{(n-1)!}{2}$ distinct Hamiltonian cycles [19].*

Proof. Let $G(V, E) = K_n$ be the complete graph on $n \geq 3$ vertices. Since $\forall\ u,v \in V,\ u \neq v \implies \{u, v\} \in E$, then, every permutation of the set V must represent a path in K_n containing every vertex of K_n and vice-versa (1-1 correspondence). Let A be the set of all permutations of the set V, and B be the set of all paths in K_n containing every vertex of K_n. Now, let H be the set of all Hamiltonian cycles in K_n. Each Hamiltonian cycle in H can be transformed into a path in B by removing the last vertex from the sequence representing the cycle. On the other hand, each Hamiltonian path can be transformed into a Hamiltonian cycle by repeating the first vertex as the last vertex without removing any vertices. Therefore, there is a 1-1 correspondence between H and B. As a result, by the transitivity of 1-1 correspondence, there is a 1-1 correspondence between A and H. Hence, the number of Hamiltonian cycles in K_n is equal to the number of permutations of V. However , different permutations of the set V may represent the same Hamiltonian Cycle in K_n since, the same edges would be used from E but in a different order of the vertices. In fact, consider the Hamiltonian Cycle C represented by the permutation v_1, v_2, ..., v_n. In terms of Hamiltonian cycles, the permutation v_1, v_2, ..., v_n is the same as the permutation v_2, ..., v_n, v_1 because, the same edges in E are used. Therefore, for each Hamiltonian cycle in K_n, we can have n permutations representing the same Hamiltonian cycle each starting from a different vertex, but using the same edges. Also, for each of these n permutation representations, the reverse of each of these permutations represent the same Hamiltonian Cycle, with the difference being that the cycle is traversed in reverse order. Therefore, there are $2n$ permutations representing the same Hamiltonian Cycle. Therefore, there are $\frac{n!}{2n} = \frac{(n-1)!}{2}$ distinct Hamiltonian cycles in K_n. □

Note that the proof for Lemma 2.0.3 was constructed using ideas in [19].

As discussed in previous paragraphs, Lemma 2.0.3 confirms that a brute-force algorithm will not be efficient enough to solve the TSP for large instance sizes. This means that, we can either try and design a new algorithm that solves the TSP in polynomial time, or, try and prove that TSP is an NP-Complete problem. As discussed in section 1.2, the latter would confirm that no polynomial time algorithm is known for the TSP. It is important to note that, this does not mean that TSP cannot be solved in polynomial time.

In what follows, it will be proven that the TSP is an NP-Complete problem. Therefore, first it will be shown that TSP is in NP, and secondly, that TSP is as hard as any problem in NP (as discussed in section 1.2). It is important to note that, to prove that TSP is in NP, TSP must be converted into a decision problem. This is needed because in Definition 2.0.1, TSP was presented as an optimisation problem, and because, by Definition 1.2.15, the class NP contains decision problems. The TSP's related decision problem is the following :

Given an instance (G, k) were G is a complete weighted graph and k is an integer, does G contain a Hamiltonian cycle of weight at most k [8]?

Having converted the TSP in Definition 2.0.1 into a decision problem, it is now time to show that TSP is in NP. This is shown in Lemma 2.0.4 below.

Theorem 2.0.4. *TSP is in NP* [8].

Proof. Consider the TSP's decision problem variant. To show that TSP is in NP, it must be shown that TSP can be verified in polynomial time. Suppose that $G(V, E)$ is a a complete weighted graph on n vertices. A certificate of a solution to the TSP given G can be a sequence of n vertices. Suppose that v_1, v_2, ..., v_n is an arbitrary certificate. To check that the certificate is correct, two checks on this certificate must be carried out by an algorithm. Firstly, the algorithm must check that no vertex in the certificate is repeated (i.e. that the certificate is a Hamiltonian cycle). Secondly, the algorithm must check that, the total weight of the Hamiltonian cycle obtained from the certificate is at most k. Therefore, the algorithm is made up of the following two parts:

- In the first part, the algorithm must check that for all $i \in [n]$, v_i appears only once in the certificate. This part of the algorithm can be implemented by, keeping an array A of boolean values having length n such that, for each v_i, the algorithm checks that $A[i]$ is 0. If $A[i]$ is set to 0, the algorithm sets $A[i]$ to 1 and moves to the next vertex in the solution sequence. If while traversing the certificate, the checker algorithm notices that for $j \in [n]$, $A[j]$ has already been set to 1, the algorithm stops because this means that, v_j is repeated and hence the certificate is not a Hamiltonian cycle. As a result, the certificate is incorrect. If this does not happen, the algorithm starts executing the next part (next bullet point). Clearly, this part of the algorithm has a worst-case $O(n)$ time complexity were n is the length of the solution sequence. The worst-case occurs when the checker algorithm needs to traverse all the vertices in the solution sequence. As a result, this part of the checker algorithm can be executed in polynomial time.

- The checker algorithm must now check that, the total weight of the Hamiltonian cycle represented by the sequence of vertices is at most k . This part of the checker algorithm can be implemented by keeping a floating point variable named 'total_weight', and summing the weight of each edge used in the Hamiltonian cycle to the variable 'total_weight'. Since a Hamiltonian cycle contains n edges, this part of the checker algorithm has a worst-case $O(n)$ time complexity because, all edges must be summed to calculate the total weight of the Hamiltonian cycle. After the total weight of the solution sequence is calculated, it is checked in $O(1)$ time whether 'total_weight' $\leq k$. If this inequality is satisfied, the checker algorithm declares the solution as correct, otherwise declares the solution as incorrect. Therefore, it can be concluded that, this part of the checker algorithm has an $O(n)$ worst-case time complexity. Hence, this part of the checker algorithm can be executed in polynomial time.

Since the checker algorithm is made up of two parts that execute in polynomial time, then, the whole algorithm runs in polynomial time. Therefore, it can be concluded that, any certificate can be verified if it is correct or not in polynomial time by the above checker algorithm. Therefore, the TSP can be verified in polynomial time and hence, it is in NP. □

Note that the above proof was constructed using the ideas found in [8]. Now it is time to show that TSP is an NP-Complete problem.

Theorem 2.0.5. *TSP is an NP-Complete problem* [8].

Proof. Consider the TSP decision version. To show that TSP is an NP-Complete problem, we must show that it is in NP, and that TSP is as hard as any problem in NP. By Theorem 2.0.4 TSP is in NP. Therefore, it remains to be shown that TSP is hard as any problem in NP. This can be done as discussed in section 1.2, by taking a known NP-Complete problem and reduce it to TSP. By theorem 34.13 in [8], the Hamiltonian cycle problem defined in Example 1.2.14 is NP-Complete. Let $G(V, E)$ be an arbitrary instance to the Hamiltonian cycle problem such that $|V| = n$. An instance of the TSP can be constructed from G as follows. From G construct the the complete weighted graph $G'(V', E', f)$ such that, $V' = V$, $E' = \{(u, v) : u, v \in V \text{ and } u \neq v\}$ and
$f\colon V' \times V' \to \{0, 1\}, f(u, v) = \begin{cases} 0 & \text{if } (u, v) \in E \\ 1 & \text{otherwise} \end{cases}$
Therefore, an instance to the TSP's can be the pair (G',0). From Definition 1.2.16, one can show that the above transformation is a reduction algorithm if the transformation can be carried in polynomial time and that, the solution to the Hamiltonian cycle problem given G is yes $\iff$ the solution to the TSP given $(G', 0)$ is yes. According to [7], K_n contains $\frac{n(n-1)}{2}$ edges. Therefore, the algorithm implementing the above transformation has an $O(n^2)$ worst-case time complexity because, it needs to create $\frac{n(n-1)}{2}$ edges between n vertices at worst. Now, for the above transformation to be a reduction algorithm, there remains to show that the solution to the Hamiltonian cycle problem given G is yes $\iff$ the solution to the TSP given $(G', 0)$ is yes. Therefore, we need to show that G has a Hamiltonian cycle $\iff$ G' has a Hamiltonian cycle of weight at most 0. Suppose that G contains some Hamiltonian cycle h. Since h is a Hamiltonian cycle in G, each edge of h is in E, therefore h has weight 0 in G'. Therefore, h

must be a Hamiltonian cycle in G' of weight at most 0. Suppose that G' contains a Hamiltonian cycle C of weight at most 0. Then, each edge of the C must belong to G. Therefore, C must be a Hamiltonian cycle in G. Therefore, we have shown that there is a reduction algorithm from the Hamiltonian cycle problem to the TSP. Therefore, TSP is as hard as the Hamiltonian cycle problem. Now since the Hamiltonian cycle problem is NP-Complete, it is as hard as any problem in NP. This means that by transitivity, the TSP is as hard as any problem in NP. Therefore, TSP must be NP-Complete. [8] □

Theorem 2.0.5 confirms that no polynomial time algorithm is known for the TSP. However, according to [8], there are three ways to get around NP-Completeness. Firstly, if the input to an NP-Complete problem is small, an exponential algorithm may be good enough [8]. Secondly, there could be special cases of the problem that could be solved in polynomial time [8]. Thirdly, there are algorithms that can be designed to find near-optimal solutions in polynomial time [8]. Algorithms that return near-optimal solutions to a problem are called approximation algorithms [8]. The discussion will now proceed by presenting some important terminologies about approximation algorithms, and then present some approximation algorithms that can be used for the TSP.

Definition 2.0.6. *Suppose that A is an optimisation problem with postive values assigned to each solution, and X is an approximation algorithm for A. Let $C^\star$ be the value of an optimal solution of A, and C be the value of the solution returned by X. X is said to have an approximation ratio p(n) if, for any input of size n, $max(\frac{C}{C^\star}, \frac{C^\star}{C}) \leq p(n)$. If an X has an approximation ratio p(n) then, X is said to be a p(n)-approximation algorithm.* [8]

Using definition 2.0.6, if an optimisation problem is a maximisation problem then, $0 \leq C \leq C^\star$. Hence, the approximation ratio is $\frac{C^\star}{C}$, which gives the factor of how large is the optimal value to the approximate solution value. On the other hand, if an optimisation problem is a minimization problem then $0 \leq C^\star \leq$ C. Hence, the approximation ratio is $\frac{C^\star}{C}$, which gives the factor of how large is the approximate solution value to the optimal value. Note that, a 1-approximation algorithm is an algorithm that produces the optimal solution since, $\frac{C^\star}{C} = 1$ and $\frac{C}{C^\star} = 1 \implies C = C^\star$. [8]

In the following subsection, two approximation algorithms suitable for the TSP will be presented along with theoratic results about them. These are the nearest neighbour approximation algorithm and the twice around the minimum spanning tree approximation algorithm.

2.1 Approximation Algorithms

2.1.1 Nearest Neighbour Approximation Algorithm

2.1.2 Twice Around the Minimum Spanning Tree Approximation Algorithm

2.2 The Ant Colony Algorithm

3 Experimental Data

4 Conclusion

References

[1] P. E. Black and P. J. Tanenbaum, "Dictionary of algorithms and data structures," Aug 2017. [Online]. Available: https://xlinux.nist.gov/dads/HTML/graph.html

[2] J. M. Harris, J. L. Hirst, and M. J. Mossinghoff, *COMBINATORICS AND GRAPH THEORY*, 2nd ed. SPRINGER, 2008.

[3] C. Thompson. [Online]. Available: https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume18/thompson03a-html/node6.html

[4] "Mathematics — walks, trails, paths, cycles and circuits in graph," Jul 2018. [Online]. Available: https://www.geeksforgeeks.org/mathematics-walks-trails-paths-cycles-and-circuits-in-graph/

[5] J. A. Bondy and U. S. R. Murty, *Graph theory with applications*, 5th ed. Elsevier Science Publishing, 1982.

[6] D. Guichard, "An introduction to combinatorics and graph theory," Jul 2018.

[7] S. S. Ray, *Graph theory with algorithms and its applications: in applied science and technology.* Springer, 2013.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed. MIT PRESS.

[9] "Shortest path algorithms." [Online]. Available: https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/

[10] "Applications of minimum spanning tree problem." [Online]. Available: https://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/

[11] V. S. Adamchik, "Algorithmic complexity," 2009. [Online]. Available: https://www.cs.cmu.edu/ adamchik/15-121/lectures/Algorithmic Complexity/complexity.html

[12] [Online]. Available: https://programming.guide/big-o-notation-explained.html

[13] T. Carter, "Tractability of computation," May 1999. [Online]. Available: https://csustan.csustan.edu/ tom/MISC/qc-article/node10.html

[14] R. E. Pettis, "Analysis of algorithms advanced programming/practicum 15-200." [Online]. Available: https://www.cs.cmu.edu/ pattis/15-1XX/15-200/lectures/aa/index.html

[15] "What is the definition of p, np, np-complete and *np*-hard?" [Online]. Available: https://cs.stackexchange.com/questions/9556/what-is-the-definition-of-p-np-np-complete-and-np-hard

[16] "Np-completeness — set 1 (introduction)," Sep 2018. [Online]. Available: https://www.geeksforgeeks.org/np-completeness-set-1/

[17] S. Goyal, "A survey on travelling salesman problem," 01 2010.

[18] M. Krivelevich, "On the number of hamilton cycles in pseudo-random graphs," *Electronic Journal of Combinatorics*, vol. 19, 11 2011.

[19] "How many hamiltonian cycles are there in a complete graph k_n $(n \geq 3)$ why?" Dec 2012. [Online]. Available: https://math.stackexchange.com/questions/249817/how-many-hamiltonian-cycles-are-there-in-a-complete-graph-k-n-n-geq-3-why