# The Ant Colony Optimisation for the Travelling Salesman Problem

Dylan Galea

March 5, 2019

# 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* $E \subseteq \{\{u, v\} : \forall\ u,v \in V$ *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]

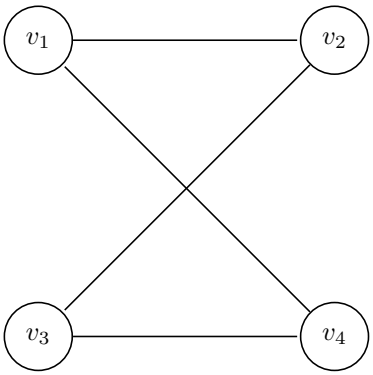
It is sometimes also required to know how many edges are incident to a specific vertex in a graph.

**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 $G$ such as walks and paths.

**Definition 1.1.7.** *Given a graph G(V,E), a walk is a sequence of vertices (not necessarily distinct)* $u= u_1, u_2, ...,u_n = v$ *such that,* $\forall$ $i \in$ *[n-1],* $\{u_i, u_{i+1}\} \in E$. *Such a walk is usually referred to as a u-v walk, were u and v are the end vertices of the walk.* [2]

**Definition 1.1.8.** *Given a graph G(V,E), a path in G joining any 2 vertices* $u, v \in V$, *is a u-v walk with no repeated vertices.*

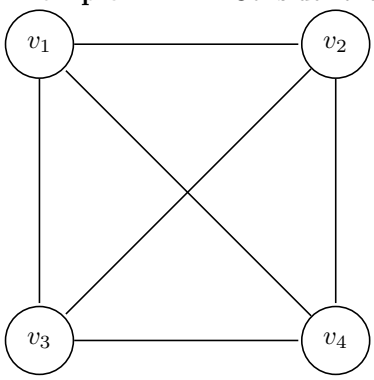
Definition 1.1.8 can now be used to define cycles and connectivity in a graph.

**Definition 1.1.9.** *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 1.1.10.** *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.* [3]

By Definitions 1.1.7, 1.1.8 above, it is clear that a path is a special instance of a walk. Similarly from Definitions 1.1.8 and 1.1.10, 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, 1.1.8 and 1.1.10, since cycles and paths are formulated in terms of walks, then they 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 throughout this project. This will later be useful when defining Hamiltonian cycles. For better understanding of definitions 1.1.6, 1.1.8, 1.1.9 and 1.1.10, example 1.1.11 is constructed.

**Example 1.1.11.** 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 walks in $G$ are:
1. $v_1\ v_2\ v_3\ v_4\ v_1\ v_2$
2. $v_1\ v_4\ v_2\ v_3\ v_1\ v_3$
3. $v_4\ v_3\ v_1\ v_4\ v_1$

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 1.1.12.** *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 1.1.12 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 1.1.13.** *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 1.1.13, 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 1.1.14.** *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 [4], 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 1.1.15.** *Given a weighted graph G(V,E,f), the total weight of any subgraph H(V′,E′,f) of G is:*

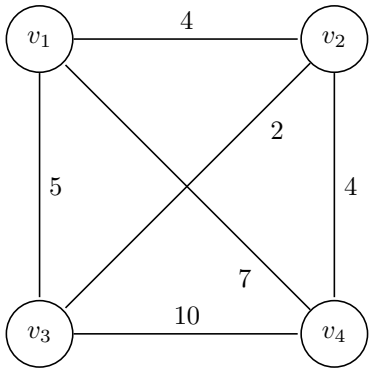
$$\sum_{e \in E'} f(e)$$

. [4]

It is important to note that by definition 1.1.12, any weighted graph G is a subgraph of itself, therefore, it's weight can be calculated. This is highlighted in Example 1.1.16 below.

**Example 1.1.16.** Consider the weighted graph *G(V,E,f)* such that, *G(V,E)* is the graph in example 1.1.11 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 1.1.14, the graph below is a weighted graph.




Also according to definition 1.1.12, the above graph is a subgraph of itself. Therefore it's weight can be calculated, where by definition 1.1.15, the weight of G is 32.

According to Guichard [5], trees are another useful class of graphs.

**Definition 1.1.17.** *A tree is a connected graph with no cycles.* [5]

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 1.1.18.** *H(V',E') is a spanning subgraph of G(V,E) if H is a subgraph of G and $V' = V$.* [6]

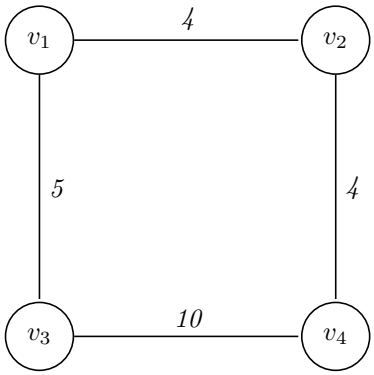
There are many spanning subgraphs, however the ones that are relevant to this thesis are spanning trees and spanning cycles, the latter mostly known as Hamiltonian cycles.

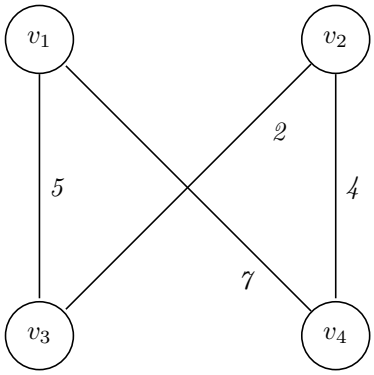
**Definition 1.1.19.** *A graph H is a spanning tree of G if H is a tree and H is a spanning subgraph of G.* [6]

**Definition 1.1.20.** *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.* [6]

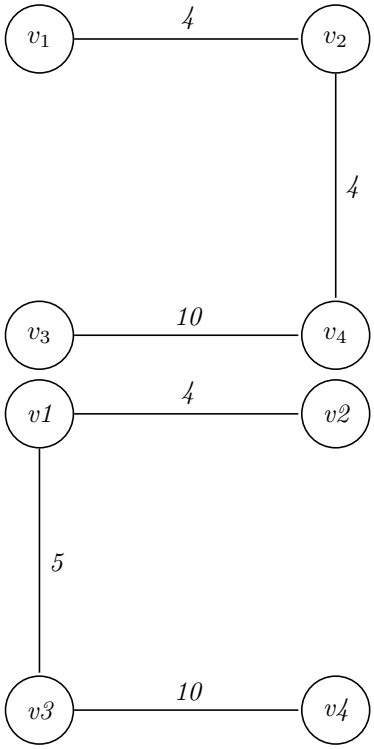
It is worth mentioning that definition 1.1.20 holds because, cycles can be represented by Cycle graphs due to the construction discussed earlier. What follows now is an example that illustrates better definitions 1.1.18, 1.1.19 and 1.1.20.

**Example 1.1.21.** Let $G$ be the graph in example 1.1.16. Then, according to definition 1.1.18, 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 1.1.20, the above two graphs are Hamiltonian cycles in $G$ because, they are spanning sub-graphs of $G$ and are Cycle sub-graphs of $G$. Since the above graphs are sub-graphs of $G$, by definition 1.1.15, 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 1.1.16, 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 sub-graph. In fact, theorem 1.1.22 gives a necessary and sufficient condition for a graph to have a spanning tree.

**Theorem 1.1.22.** *A graph G has a spanning tree* $\iff$ *it is connected* [6].

*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.
($\impliedby$) 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 $G$. [6] $\square$

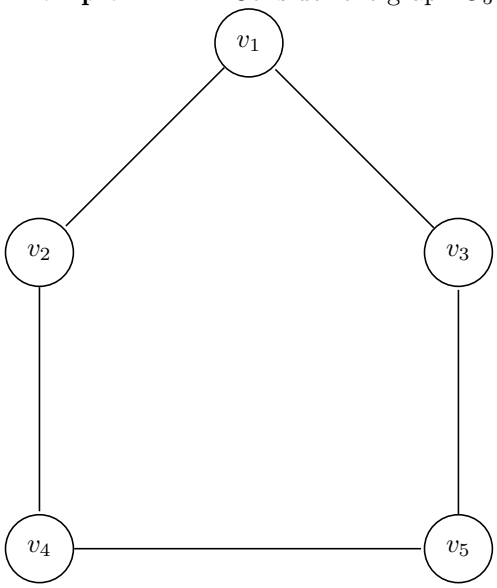
Theorem 1.1.22 confirms that for a graph to have a spanning tree, the graph must be connected and vice-versa. Thus, for spanning trees, the necessary and sufficient condition is connectivity. However, the same cannot be said about Hamiltonian cycles because, no necessary and sufficient conditions are 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 [5], 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 [5], 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 1.1.23** (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.* [6]

*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. [6] □

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 1.1.24 below is a counter example to show why Ore's theorem gives a sufficient but not necessary condition.

**Example 1.1.24.** 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.* [7]

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. [7]

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.* [7]

In [7] 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 [7]. 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 [7]. 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 $G$ [8]. 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 [9]. 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 [7]. 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.* [7], [10]

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, $T(n) = a \times n$. [7], [10]

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). [7], [10]

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. [7], [10]

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. [7], [10]

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 $f(n) \leq cg(n)\ \forall\ n \geq n_0$.* [10]

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 [10]. 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)).* [10]

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 [11] and [12].

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 [11]. 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 [7]. On the other hand, problems that

require superpolynomial (exponential/factorial) time algorithms to be solved are intractable problems [7]. 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* or *100*, 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

| **Time Complexity** \ **Input Size** | **10** | **100** |
|---|---|---|
| $O(n)$ | $3.171 \times 10^{-16}$ *years* | $3.171 \times 10^{-15}$ *years* |
| $O(n^2)$ | $3.171 \times 10^{-15}$ *years* | $3.171 \times 10^{-13}$ *years* |
| $O(2^n)$ | $3.247 \times 10^{-14}$ *years* | $4.0197 \times 10^{13}$ *years* |
| $O(n!)$ | $1.1507 \times 10^{-10}$ *years* | Very large to compute |

Table 2 above was adapted from [13]. In Table 2 above it is assumed that the machine the algorithms can be executed on perform $10^9$ computational steps per second [13]. 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. [7]

**Definition 1.2.11.** *The class P consists of all the problems that can can be solved in polynomial time.* [7]

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 a mathematical object that guarantees the solution.* [14]

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. For convenience, in this project, 'verifying a solution/certificate if it is correct' will be referred to as 'verifying a solution'. 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 about languages and verification algorithms which are beyond the scope of this project. A more formal discussion about certificates is given in [7].

**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 $C$ and an instance (input) of $A$ as inputs, the algorithm can verify if $C$ is correct.* [7]

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 verified by a polynomial time algorithm. Note that as discussed before, when verifying a certificate, we are implicitly verifying the solution. 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 [7]. 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 problem to be verifiable 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.

Note that, as shown in Example 1.2.14, to show that a problem is verifiable in polynomial time, it is only needed to be shown that a certificate of the solution 'yes' can be verified in polynomial time'. This is because, any certificate is correct when the answer is 'no'. 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.* [7], [15]

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. [7]

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 $\alpha$ of $A$ into an instance $\beta$ of $B$ with the following properties:*

- *The algorithm takes polynomial time to execute*
- *The solution to $\alpha$ is yes $\iff$ the solution to $\beta$ is yes.* [7]

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 $\alpha$ 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 $\alpha$ into an instance $\beta$ of $B$
- Execute $B^\star$ on the input $\beta$ to get answer $\gamma$ = yes/no
- Give $\gamma$ as an answer for $\alpha$.

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 $\alpha$ is yes $\iff$ the solution to $\beta$ is yes. [7]

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 $\alpha$ of $A$ into an instance $\beta$ of $B$ were again, $\alpha$ can be solved in polynomial time as discussed in the previous paragraph. Since $\alpha$ 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 $A$. [7]

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.* [7]

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 an alleged solution for an instance of A can be verified in polynomial time. As discussed before, this can be done by checking that a certificate for the solution 'yes' can be verified 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 $A$ [15]. However, this is not the case because reduction algorithms are transitive [15]. 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 [15].

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. [7]

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

## 2.1 Problem Definition, Properties and Complexity Class

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 2.1.1 below.

**Definition 2.1.1.** *Given a complete weighted graph $G$ on at least three vertices, the Travelling Salesman Problem (TSP) is the problem of finding a minimum weight Hamiltonian cycle in $G$.* [4]

According to Definition 2.1.1, a weighted graph $G$ must be complete for the TSP to be defined. 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 in this way. 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 2.1.1 also assumes, 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 1.1.10, cycles are defined on graphs having at least 3 vertices.

The following example is used to clearly explain what the TSP defined in Definition 2.1.1 is all about.

**Example 2.1.2.** Consider the complete graph $K_4$ with the weight function $f$ defined in example 1.1.16. 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 2.1.1, 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 three distinct Hamiltonian cycles. This fact is a special case of Lemma 2.1.3 for any $K_n$.*

Example 2.1.2 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 [16]. However, $K_n$ 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 [16].

**Lemma 2.1.3.** *For $n \geq 3$, $K_n$ has $\frac{(n-1)!}{2}$ distinct Hamiltonian cycles [17].*

*Proof.* Suppose that $V$ and $E$ are the set of verties and the set of edges of $K_n$ respectively. Any Hamiltonian cycle in $K_n$ can be represented by a permutation of $V$ and vice-versa. For example the Hamiltonian cycle $h = v_1, ..., v_n, v_1$ can be represented by the permutation $v_1\ v_2\ ...\ v_n$ and vice-versa. Therefore, the number of Hamiltonian cycles in $K_n$ is equal to the number of permutations of $V$. Since there are $n!$ permutations of $V$, there must be $n!$ Hamiltonian cycles in $K_n$. Now, in terms of Hamiltonian cycles, $h$, is the same as the Hamiltonian cycle $v_2, ..., v_n, v_1, v_2$. This is true because, altough the order of vertices is different, they use the same edges from $E$. Therefore, in general there are $n$ distinct permutations (one for every vertex in $K_n$ when used as a starting vertex of the vertex sequence) representing the same Hamiltonian cycle. In addition to this, each permutation would still represent the same Hamiltonian cycle if the vertices are in reversed order. This is true because, again the same edges from $E$ are used. Therefore, there must be $\frac{n!}{2n} = \frac{(n-1)!}{2}$ distinct Hamiltonian cycles in $K_n$. □

As discussed in previous paragraphs, Lemma 2.1.3 confirms that a brute-force algorithm is not efficient enough to solve the TSP for large inputs. 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 2.1.1, 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 :

**TSP defined as a decision problem:**

*Given an instance $(G, k)$ where $G$ is a complete weighted graph and $k$ is an integer, does $G$ contain a Hamiltonian cycle of weight at most $k$* [7]?

Having converted the TSP in Definition 2.1.1 into a decision problem, it is now time to show that TSP is in NP. This is shown in Lemma 2.1.4 below.

**Theorem 2.1.4.** *TSP is in NP* [7].

*Proof.* Consider the TSP's decision problem variant. To show that TSP is in NP, it must be shown that an alleged solution for the TSP can be verified in polynomial time. As discussed in section 1.2, this can be done by verifying in polynomial time a certificate for the solution 'yes'. Suppose that $G(V, E)$ is 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 when it encounters $v_i$ in the given sequence and then 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 first 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 [7]. Now it is time to show that TSP is an NP-Complete problem. To show that TSP is an NP-Complete problem, it remains to be shown that the TSP is as hard as any problem in NP. This can be done as discussed in section 1.2, by taking a known NP-Complete problem and reducing it to the TSP. The known NP-Complete problem that is going to be used is the Hamiltonian cycle problem. Note that we already showed in Example 1.2.14 that the Hamiltonian cycle problem is in NP. However, it will not be shown that the Hamiltonian cycle problem is as hard as any problem in NP, because, the proof is out of the scope of this project. In [7], it is shown that the Hamiltonian cycle problem is as hard as any problem in NP, by reducing the Vertex-Cover problem into the Hamiltonian cycle problem.

**Theorem 2.1.5.** *TSP is an NP-Complete problem* [7].

*Proof.* Consider the TSP as a decision problem. By Theorem 2.1.4 TSP is in NP. Therefore, it remains to be shown that TSP is hard as any problem in NP. By Theorem 34.13 in [7], the Hamiltonian cycle problem is NP-Complete. We will now show that the Hamiltonian cycle problem can be reduced to the TSP. 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 constructed as a decision problem, can be the pair $(G', 0)$ where $G'$ is the complete weighted graph, and $k = 0$. 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 [6], $K_n$ 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. [7] □

Theorem 2.1.5 confirms that no polynomial time algorithm which solves the TSP is known. However, according to [7], 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 [7]. Secondly, there could be special cases of the problem that could be solved in polynomial time [7]. Thirdly, there are algorithms that can be designed to find near-optimal solutions in polynomial time [7]. Algorithms that return near-optimal solutions to a problem are called approximation algorithms [7]. The discussion will now proceed by presenting some concepts about approximation algorithms.

**Definition 2.1.6.** *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. The algorithm 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 algorithm X has an approximation ratio p(n) then, X is said to be a p(n)-approximation algorithm.* [7]

Using Definition 2.1.6, if an optimisation problem is a maximisation problem, then, $0 \leq C \leq C^\star$. Hence, the approximation ratio is $\frac{C^\star}{C}$, giving 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 $\frac{C}{C^\star}$, 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 $\frac{C}{C^\star} = 1 \implies C = C^\star$. [7]

It is important to note that many problems have polynomial time approximation algorithms with a small constant approximation ratio [7]. However, this is not always the case because, there are problems whose best known polynomial time approximation algorithms have approximation ratios that grow as the size of the input increases [7]. It is now natural to ask whether the TSP has a polynomial time approximation algorithm with a constant approximation ratio. According to [7], it turns out that if a polynomial time approximation algorithm with a constant approximation ratio for the TSP exists, then P=NP.

**Theorem 2.1.7.** *Suppose that $P \neq NP$, then, for any constant $p \geq 1$, no polynomial time approximation algorithm with approximation ratio p exists for the TSP* [7].

*Proof.* Consider the TSP defined in Definition 2.1.1. Suppose that P $\neq$ NP and that there exists a constant $p \geq 1$ such that, $A$ is a polynomial time $p$-

approximation algorithm for the TSP. WLOG assume that $p$ is an integer, if not round it to the nearest integer. Let $G(V, E)$, $|V| = n$ be an instance of the Hamiltonian cycle problem defined in Example 1.2.14. Construct an instance $G'(V', E', f)$ of the TSP from $G$ such that, $V' = V$, $E' = \{\{u, v\} : u, v \in V \text{ and } u \neq v\}$ and

$$f\colon V' \times V' \to \{1, p|V| + 1\},\ f(\{u, v\}) = \begin{cases} 1 & \text{if } \{u, v\} \in E \\ p|V| + 1 & \text{otherwise} \end{cases}$$

As discussed in Theorem 2.1.5, this construction can be done in polynomial time. Now consider the TSP defined on the graph $G'$. If $G$ contains a Hamiltonian cycle $h$, then $f$ assigns a weight of 1 to each edge of $h$ in $G'$. Therefore, by Definition 1.1.15, $h$ has weight $|V|$ in $G'$. If $G$ is not Hamiltonian, any Hamiltonian cycle in $G'$ must contain edges which are not in $E$. As a result, the weight of any Hamiltonian cycle in $G'$ is at least $p|V| + 1 + |V| - 1 = p|V| + |V| = (p + 1)|V| > p|V| \geq |V|$ $(\star)$. Now, apply the approximation algorithm $A$ on $G'$, and let $w^\star$ be the optimal value of the TSP defined on $G'$. We will now show that $A$ can be used to check if $G$ is Hamiltonian. Therefore, we get the following two cases :

**Case 1.** *$G$ is Hamiltonian*

Let $h'$ be a Hamiltonian cycle in $G'$. Since $G$ is Hamiltonian, $h'$ either consists only of edges that are in $E$, or consists of edges some of which are in $E$ and some of which are not. Therefore, by the previous paragraph, $h'$ can either have weight $|V|$ or weight of at least $p|V| + |V|$. This means that in general, the Hamiltonian cycles in $G'$ can only have a weight of $|V|$ or a weight of at least $p|V| + |V|$. Now, since in (*), $p|V| + |V| > |V|$, the minimum weight Hamiltonian cycle in $G'$ must have weight $|V|$. Therefore, $w^\star = |V|$. Now, since $A$ is a $p$-approximation algorithm, $A$ guarantees that it will return a Hamiltonian cycle which has weight less than or equal to $pw^\star = p|V|$ when such cycle exists. Therefore, since $G'$ has Hamiltonian cycles of weight $|V| \leq p|V|$, and by $(\star)$ the remaining Hamiltonian cycles are of weight at least $p|V| + |V| > p|V|$, the approximation algorithm will always return a Hamiltonian cycle of weight $|V|$. Therefore, whenever $G$ is Hamiltonian, $A$ will return a Hamiltonian cycle of weight $|V|$.

**Case 2.** *$G$ is not Hamiltonian*

Since $G$ is not Hamiltonian, by the paragraph before Case 1 above, every Hamiltonian cycle in $G'$ has at least a weight of $p|V| + |V|$. Therefore, the minimum weight Hamiltonian cycle in $G'$ has at least a weight of $p|V| + |V|$. Now, since the TSP is a minimization problem, any solution returned by $A$ must have a value greater than or equal to the the optimal value. Therefore, A must return a Hamiltonian cycle having weight at least $p|V| + |V|$. Now, since by $(\star)$, $p|V| + |V| > p|V|$, $A$ must return a Hamiltonian cycle of weight greater than $p|V|$. As a result, whenever $G$ is not Hamiltonian, $A$ must return a Hamiltonian cycle having weight greater than $p|V|$

From Case 1 and Case 2 above it can be concluded that, by checking the weight of the Hamiltonian cycle returned by A, one can determine whether $G$ is Hamiltonian or not. As discussed in Theorem 2.1.4, this check can be done in polynomial time. Therefore, since $A$ is a polynomial time algorithm, the whole procedure solves the Hamiltonian Cycle Problem in polynomial time. As discussed in Section 1.2, if one NP-Complete problem can be solved in polynomial time, then P=NP. Now, by Theorem 34.4 in [7], the Hamiltonian Cycle problem is NP-Complete, therefore P=NP. Contradiction.
$\therefore$ There is no polynomial time approximation algorithm with a constant approximation ratio for the TSP. □

The proof of Theorem 2.1.7 was constructed using ideas in [7].

As discussed in Section 1.2, the P=NP problem is an open problem. Therefore, we do not know of a polynomial time approximation algorithm with a constant approximation ratio for the TSP, otherwise, the contrappositve of Theorem 2.1.7 would imply that P=NP, contradicting the fact that the P=NP problem is an open problem. It is important to note that, it is not easy to determine the approximation ratio of an approximation algorithm. In fact, there could be cases that an approximation ratio may only be determined for specific instances of a problem [7]. One of the specific instances of the TSP that will be considered in the following section is called the Metric-TSP defined below.

**Definition 2.1.8.** *The Metric-TSP is the TSP defined on a complete weighted graph $G(V, E, f)$ on at least 3 vertices, with the additional property that $\forall$ u,v,w $\in$ V, $f(\{u, v\}) \leq f(\{u, w\}) + f(\{w, v\})$. The additional property is called the triangle inequality.* [18]

What follows is a subsection on two approximation algorithms that are suitable for the TSP. These approximation algorithms are the Nearest Neighbour Approximation Algorithm and the Twice Around The Minimum Spanning Tree Approximation Algorithm. Both of these algorithms can be applied to the TSP defined in Definition 2.1.1, however, we do not know their approximation ratios when applied to the TSP defined in Definition 2.1.1.

## 2.2 Approximation Algorithms

### 2.2.1 Nearest Neighbour Approximation Algorithm

The Nearest Neighbour Algorithm (NNA) is a simple algorithm that can find near-optimal solutions for the TSP [19]. The NNA takes as input, an instance of the TSP, and outputs a sequence of all visited vertices in the order of how they were visited [19]. According to [19], the NNA can be described by the following steps:

**NNA Procedure:**

- Randomly select any vertex $u$ as starting vertex, mark it as visited, and set $current\_vertex = u$.
- Find an edge of least weight between vertex $current\_vertex$ and an unvisited vertex $v$.
- Mark $v$ as visited and set $current\_vertex = v$.
- If all vertices have been visited, move to vertex $u$ and go to bullet point five, otherwise, go to bullet point two.
- Return the sequence of vertices in the order of how they were visited.

From the NNA procedure above it can be deduced that, the idea behind the NNA is to try and find a minimum weight Hamiltonian cycle by always moving from one vertex to another using the edge of least weight. This means that the NNA tries to choose an edge which looks the best at that point in time, hoping that it would lead to a minimum weight Hamiltonian cycle. It is important to note that although the goal of the NNA is to find a minimum weight Hamiltonian cycle, the goal is not guaranteed [20].

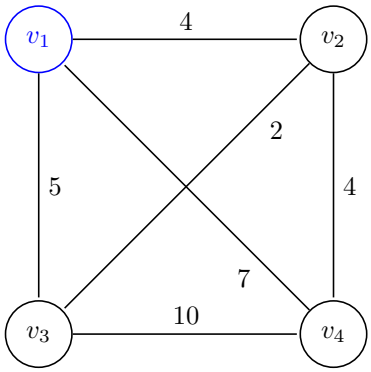
What we must guarantee is that the sequence of vertices returned by the NNA forms a Hamiltonian cycle. This guarantee is given because, all the vertices in the graph must be visited before the algorithm terminates. In addition to that, excluding the starting vertex, each vertex cannot be visited more than once because by bullet point number two in the NNA Procedure, the algorithm only chooses edges that lead to unvisited vertices. Therefore, by Definition 1.1.20, the sequence of vertices outputted by the algorithm represent a Hamiltonian Cycle.

It is also important to check that the NNA can execute in polynomial time. Otherwise, for large inputs we cannot give approximations in feasible time. According to [21], the NNA has a quadratic time complexity. This is true because of the following. Suppose that $current\_vertex = u$, where $u$ is an arbitrary vertex in the TSP instance. Then, the NNA determines the next vertex $v$ to move to by checking every vertex adjacent to $u$. Therefore, since each vertex is adjacent to $n-1$ other vertices, the NNA must perform $n-1$ steps for each of the $n$ vertices, which yields a total of $n(n-1)$ steps. Therefore the NNA must have a quadratic time complexity. What follows is an example showing how the NNA constructs an approximation for the TSP.

**Example 2.2.1.** Consider the graph $G$ in Example 1.1.16. The NNA constructs a solution to the TSP defined on $G$ in the following way. Assume that randomly the NNA choses to start from vertex $v_1$. The following graphs give a pictorial representation on how the NNA will progress step by step on $G$, using the NNA Procedure above. In these graphs, the blue vertices are the visited vertices, and black vertices are the unvisited vertices. Also, the chosen edges will be coloured red whilst the unchosen edges will be coloured black.

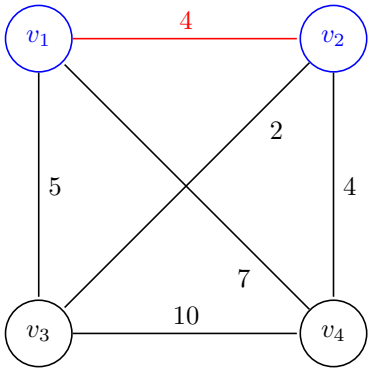
Step 1:

Mark the starting vertex $v_1$ as visited



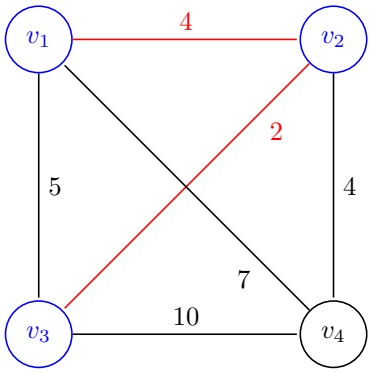

Step 2:

Edge $\{v_1, v_2\}$ is chosen because it has the least weight amongst all edges that $v_1$ is adjacent to. Therefore, move to $v_2$ and mark $v_2$ as visited.



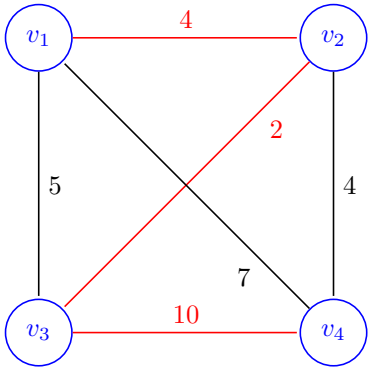

Step 3:

Since not all vertices are visited, steps 1 and 2 above are repeated and thus, edge $\{v_2, v_3\}$ is chosen. Therefore, move to vertex $v_3$ and mark $v_3$ as visited.



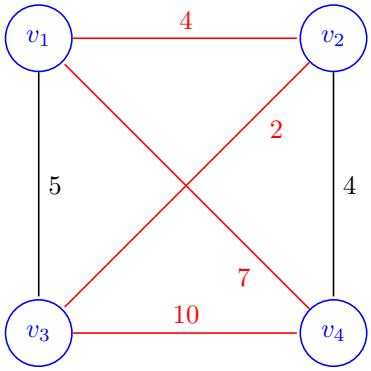

Step 4:

Since not all vertices are visited, steps 1 and 2 above are repeated and thus, edge $\{v_3, v_4\}$ is chosen. Therefore, move to vertex $v_4$ and mark $v_4$ as visited.




Step 5:

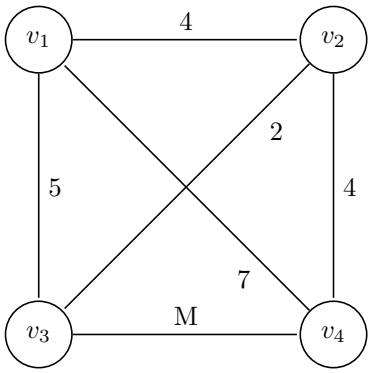
Since all vertices are visited, the NNA moves to the starting vertex.



Finally, the NNA returns the Hamiltonian cycle $v_1$, $v_2$, $v_3$, $v_4$, $v_1$ of weight 23.

It is important to note that in Example 2.2.1, the minimum weight Hamiltonian cycle was not found. In fact, according to Example 2.1.2, the solution for the TSP given $G$ is the Hamiltonian cycle $v_1$, $v_3$, $v_2$, $v_4$, $v_1$ of weight 18. Note that the NNA would have found the optimal solution if it had started from vertex $v_2$. Since the approximate solution value does not differ a lot from the optimal value, this example seems to indicate that the NNA returns good approximations. However, this is not the case because, there could be instances where the NNA returns very bad approximations. The following example shows when the NNA can give bad approximations.

**Example 2.2.2.** Let $M \in \mathbb{Z}^+$. Consider the graph $G$ below:




Assuming that $M$ is a very large number, the minimum weight Hamiltonian cycle of $G$ is $v_2, v_3, v_1, v_4, v_2$ of weight 18. This minimum weight Hamiltonian cycle is found if the NNA starts from vertex $v_2$. Now suppose that the NNA starts from vertex $v_1$. Then, NNA returns the Hamiltonian cycle $v_1$, $v_2$, $v_3$, $v_4$, $v_1$ of weight $M + 13$. This means that, since $M$ is a very large number, the NNA would return an approximate result that differs a lot in weight compared to the optimal solution. Therefore, in this case, the NNA returns a very bad approximation.

It can be concluded from Example 2.2.2 that the quality of the result returned by the NNA depends heavily on which vertex the algorithm starts with, especially when the weights differ a lot. As seen in Example 2.2.2, this happens when the edge with a very large weight has to be used in order to complete the Hamiltonian cycle, or when all other vertices have been visited. In fact in Example 2.2.2, edge $\{v_3, v_4\}$ has to be used because, *current_vertex* was equal to $v_3$ but only $v_4$ was unvisited.

Since the NNA is an approximation algorithm, it is natural to ask what is it's approximation ratio when applied to the TSP. As discussed in Section 2.0,

we can already confirm from Theorem 2.1.7 that it is known that the NNA does not have a constant approximation ratio unless P=NP. As a result, the approximation ratio of the NNA applied to the TSP defined in Definition 2.1.1 is not known, otherwise, P=NP. However, Rosenkrantz, Stearns and Lewis [22], proved that, the NNA applied to an instance $G$ of the Metric-TSP has an approximation ratio that is logarithmic in the number of vertices in $G$. In what follows, a lemma will be proved. This will help in proving that the approximation ratio of the NNA applied to an instance $G$ of the Metric-TSP, is logarithmic in the number of vertices in $G$ .

**Lemma 2.2.3.** *Let G(V,E,f) be a complete weighted graph on n vertices, and $W^\star$ be the optimal value of the Metric-TSP defined on G. Suppose that there exists a mapping $g : V \mapsto \Re^+$ such that, for all $v \in V$, $g(v) = l_v$ and that the following two conditions hold:*

- *$f(\{u, v\}) \geq min(\ l_u,\ l_v)$ for all u,v $\in V$*
- *$l_u \leq \frac{1}{2}W^\star$ for all $u \in V$*

*Then $\sum_{u\in V} l_u \leq \frac{1}{2}(\lceil log_2 n\rceil + 1)W^\star$. [22]*

*Proof.* Let $G(V, E, f)$ be a complete weighted graph on $n$ vertices, satisfying all the conditions in the Theorem statement. Suppose that $V = \{i|1 \leq i \leq n\}$ and that, for all $i$, $j \in$ V, if $i \leq j$ then $l_i \geq l_j$. Clearly, this just re-names and re-orders the vertices, hence, the generality of the proof is not effected.

**Claim 1.** *$W^\star \geq 2 \sum_{i=k+1}^{min(2k,n)} l_i$, for all $1 \leq k \leq n$, where $k \in \mathbb{N}$.*

Let $H(V', E', f)$ be the complete weighted subgraph of $G$, such that, $V'$ $= \{i|1 \leq i \leq min(2k, n)\}$. Let $T$ be a sequence of vertices representing a Hamiltonian cycle in $H$, such that, the vertices are ordered as they would appear in a minimum weight Hamiltonian cycle of $G$. Let $C(V', E'', f)$ be the Cycle graph obtained from $T$ with weight $T^\star$. By the triangle inequality, for all $\{i, j\}$ $\in E''$, $f(\{i, j\})$ is less than or equal to the weight of the path from $i$ to $j$ in a minimum weight Hamiltonian cycle of $G$. This holds because, the vertices in $T$ appear in the same order as in the minimum weight Hamiltonian cycle. Therefore, $T^\star \leq W^\star$. Now, by Definition 1.1.15, $T^\star = \sum_{\{i,j\}\in E''} f(\{i, j\})$. Therefore, by the first bullet point in Lemma 2.2.3, $T^\star = \sum_{\{i,j\}\in E''} f(\{i, j\}) \geq$ $\sum_{\{i,j\}\in E''} min(l_i, l_j)$. For each $i \in [min(2k, n)]$, let $\alpha_i$ be the number of edges $\{i, j\} \in E''$, such that, $i > j$ ( i.e $l_i = min(l_i, l_j)$). Then, $\sum_{\{i,j\}\in E''} min(l_i, l_j)$ $= \sum_{i\in E'} \alpha_i l_i$
$\implies T^\star \geq \sum_{i\in E'} \alpha_i l_i = \sum_{i=1}^{min(2k,n)} \alpha_i l_i$.
Now, for all $i \in [min(2k, n)]$, $\alpha_i \leq 2$. This is true because, every vertex $v_i$ is incident to exactly two edges of $C$. In addition to this, $\alpha_1$ + .. + $\alpha_{min(2k,n)}$ $= |E''|$, because, for all $\{i, j\} \in E''$, either $i \geq j$ or $j > i$. Therefore, either $\alpha_i$ or $\alpha_j$ is incremented by one. Another observation is that, $k$ is at least half of the number of vertices in $C$. This is true because of the following. Suppose that, $min(2k, n)$ = 2k. Then, by the construction of $C$ and $H$, $C$ contains $2k$

vertices. Hence, $k$ must be half of the number of vertices in $C$. Now, suppose that $min(2k, n) = n$. Then, by the construction of $C$ and $H$, $C$ contains $n$ vertices. Now, $n \leq 2k$, hence, $k$ must be at least half of the number of vertices in $C$. Therefore, from both cases, it can be deduced that $k$ is at least half of the number of vertices in $C$.

**Claim 2.** $\sum_{i=1}^{min(2k,n)} \alpha_i l_i \geq 2 \sum_{i=k+1}^{min(2k,n)} l_i$.

Since $k$ is at least half the number of vertices in $C$, and each $l_i$ is positive, $2 \sum_{i=k+1}^{min(2k,n)} l_i$ has the largest value when $k = \frac{1}{2}min(2k, n)$. Therefore, if we can prove Claim 2 for $k = \frac{1}{2}min(2k, n)$, we prove it for all possible $k$. Let $k = \frac{1}{2}min(2k, n)$ and $\alpha_i = 2$ for all $i \in [k + 1, min(2k, n)]$. Then, $\alpha_{k+1} + ... + \alpha_{min(2k,n)} = 2(min(2k, n)$ - $k) = 2(min(2k, n)$ - $\frac{1}{2}min(2k, n)$ $) = min(2k, n) =$ number of edges in $C$. Therefore, since $\alpha_1 + .. + \alpha_{min(2k,n)} = |E''|$, then, $\alpha_i = 0$ for all $i \in [k]$. Therefore, $\sum_{i=1}^{min(2k,n)} \alpha_i l_i = \sum_{i=k+1}^{min(2k,n)} 2l_i = 2 \sum_{i=k+1}^{min(2k,n)} l_i$. Therefore $\sum_{i=1}^{min(2k,n)} \alpha_i l_i \geq 2 \sum_{i=k+1}^{min(2k,n)} l_i$, and hence, Claim 2 is proved.

Now, since $W^\star \geq T^\star \geq \sum_{i=1}^{min(2k,n)} \alpha_i l_i$. Then, by Claim 2, $W^\star \geq 2 \sum_{i=k+1}^{min(2k,n)} l_i$, and hence Claim 1 is proved aswell.

By Claim 1, $W^\star \geq 2 \sum_{i=k+1}^{min(2k,n)} l_i$.
Hence, $\sum_{j=0}^{\lceil log_2 n \rceil - 1} W^\star \geq \sum_{j=0}^{\lceil log_2 n \rceil - 1} 2 \sum_{i=k+1}^{min(2k,n)} l_i$. Letting $k = 2^j$, we get that, $\sum_{j=0}^{\lceil log_2 n \rceil - 1} W^\star \geq \sum_{j=0}^{\lceil log_2 n \rceil - 1} 2 \sum_{i=2^j+1}^{min(2^{j+1},n)} l_i$.
$\implies \lceil log_2 n \rceil \; W^\star \geq 2 \sum_{i=2}^{n} l_i$. Now, bullet point number two in Lemma 2.2.3 implies that, $W^\star \geq 2l_1$.
$\implies \lceil log_2 n \rceil \; W^\star \geq 2 \sum_{i=1}^{n} l_i$.
$\implies \frac{1}{2} \; (\lceil log_2 n \rceil + 1) \; W^\star \geq \sum_{i=1}^{n} l_i = \sum_{u \in V} l_u$. □

The above proof was constructed using ideas from [22]. Having proved Lemma 2.2.3, it is now time to prove that, when applied to an instance $G$ of the Metric-TSP, the NNA has an approximation ratio that depends logarithmically on the number of vertices in $G$.

**Theorem 2.2.4.** *Suppose that $G(V, E, f)$ is an instance of the Metric-TSP. Let $W^\star$ be the optimal value of $G$, and apply the NNA on $G$. Suppose that $W$ is the value of the approximation returned by the NNA. Then, $\frac{W}{W^\star} \leq \frac{1}{2}(\lceil log_2 n \rceil + 1)$* [22].

*Proof.* Define $G(V, E, f)$, $W^\star$ and $W$ as stated in Theorem 2.2.4. Consider the NNA Procedure described in this section and apply it to $G$. Let $g : V \mapsto \Re^+$ be a mapping defined by $g(v) = f(\{v, u\})$, where, $u$ is the vertex chosen by the NNA when *current_vertex* $= v$. For all $v \in V$, let $l_v = g(v)$. We show that $g$ satisfies the two conditions of Lemma 2.2.3.

**Condition 1** : $f(\{u, v\}) \geq \min(\, l_u, \, l_v)$ for all $u$,$v \in V$

Let $u, v \in V$. Suppose that $u$ was set as visited by the NNA before $v$. Then, since $G$ is complete, $\{u, v\}$ was a candidate edge for the NNA when *current_vertex* was equal to $u$. Therefore, $f(\{u, v\})$ is greater than or equal to the weight of the edge selected when *current_vertex* was equal to $u$. Note that the weight of the selected edge is $l_u$. Therefore, $f(\{u, v\}) \geq l_u$ (*). Using the same argument, if $v$ was set as visited by the NNA before $u$, then, $f(\{v, u\}) = f(\{u, v\}) \geq l_v$ (**). Now, since the NNA must visit all veritices, either $u$ is visited before $v$ or vice-versa. As a result, one of (*) or (**) must hold. Hence, $f(\{u, v\}) \geq \min(l_u, l_v)$. Therefore, since $u$ and $v$ are arbitrary, Condition 1 holds.

**Condition 2** : $l_u \leq \frac{1}{2}W^\star$ for all u $\in$ V

Let $u, v \in$ V such that, $l_u = f(\{u, v\})$. Let $h$ be a minimum weight Hamiltonian cycle in $G$. Since $h$ is spanning a spanning subgraph, both $u$ and $v$ are in $h$. Therefore, $h$ can be expressed as the disjoint union of two paths joining $u$ and $v$. One of the paths can be obtain by starting from vertex $u$ and traversing through $h$ clockwise until finding $v$. The other path can be obtained by starting from vertex $u$ and traversing through $h$ anti-clockwise until finding $v$. Let these paths be $p_1$ and $p_2$. By the triangle inequality, the weight of $p_1$ and the weight of $p_2$ are both greater than or equal to $f(\{u, v\})$ in $G$. Therefore, summing both inequalities we get, $2l_u = 2f(\{u, v\}) \leq$ weight of $p_1$ + weight of $p_2$ = weight of $h$ $= W^\star$. As a result, $l_u \leq \frac{1}{2}W^\star$. Therefore, since $u$ is arbitrary, Condition 2 holds.

Now, for all $v$ in $V$, $l_v$ is the weight of the edge chosen by the NNA when *current_vertex* $= v$. Therefore, $\sum_{u \in V} l_u = W$. Using Lemma 2.2.3, $W = \sum_{u \in V} l_u \leq \frac{1}{2}\left(\lceil log_2 n \rceil + 1\right) W^\star$.
$\implies W \leq \frac{1}{2}\left(\lceil log_2 n \rceil + 1\right) W^\star$
$\implies \frac{W}{W^\star} \leq \frac{1}{2}\left(\lceil log_2 n \rceil + 1\right)$. □

The proof of Theorem 2.2.4 was constructed using ideas in [22].

The implication of Theorem 2.2.4 is that, when applying the NNA on an instance $G$ of the Metric-TSP, the quality of the approximation depends on the number of vertices in $G$. This is true because of the following. As the number of vertices increases, the approximation ratio increases logarithmically. Therefore, for very large $n$, $\frac{1}{2}(\lceil log_2 n \rceil + 1)$ may become very large that it may allow bad approximations to be returned by the NNA. These bad approximation would be returned because, their value would still be less than $\frac{1}{2}(\lceil log_2 n \rceil + 1) \times$ optimal value. One way of solving this problem is to try and deduce whether the NNA can have a constant approximation ratio when applied to the Metric-TSP. This would solve the problem because, as the number of vertices increases, we are guaranteed that the value of the approximation is within a constant factor of the optimal value. In other words, as the number of vertices increases, approximations whose value is greater than the approximation ratio would not become eligible for the NNA. However, Rosenkrantz, Stearns and Lewis [22], proved that there is an instance $G$ of the Metric-TSP such that, when the NNA is applied to

$G$, it has an approximation ratio that is strictly bounded below by a logarithmic function of the number of vertices in $G$. This means that in general, the NNA cannot have a constant approximation ratio when applied to the Metric-TSP. This fact is stated in Theorem 2.2.5 below without proof because, the proof is too long.

**Theorem 2.2.5.** *For all $m > 3$, $m \in \mathbb{N}$, there is an instance $G(V, E, f)$ of the Metric-TSP such that, $|V| = 2^m - 1$ and $\frac{W}{W^\star} > \frac{1}{3}(log_2(n+1)) + \frac{4}{9}$, where $W$ is the optimal value of $G$, and $W^\star$ is the value of the approximation returned by the NNA when it is applied to $G$.* [22]

Theorem 2.2.4 and 2.2.5 imply that in general the NNA cannot have a constant approximation ratio when applied to the Metric-TSP. However, this does not mean that the Metric-TSP cannot be approximated using an approximation algorithm that has a constant approximation ratio. In fact, an approximation algorithm that is known to have a contstant approximation ratio when applied to the Metric-TSP is the Twice Around the Minimum Spanning Tree Approximation Algorithm. The Twice Around the Minimum Spanning Tree Approximation Algorithm will be presented in the next subsection.

### 2.2.2 Twice Around the Minimum Spanning Tree Approximation Algorithm

Similarly to the NNA, the Twice Around the Minimum Spanning Tree Algorithm (TAMSA) is an algorithm that can find near-optimal solutions for the TSP [7]. The TAMSA computes an approximate solution for the TSP by using two other algorithms. In what follows, these two algorithms will be briefly described without delving into any specific details related to their implementation.

The first algorithm used by the TAMSA is Prim's algorithm. Prim's algorithm takes a weighted graph $G(V, E, f)$ and an arbitrary starting vertex $r \in V$ as inputs, and returns a minimum weight spanning tree $T(V', E', f)$ of $G$. Prim's algorithm computes $T$ in the following way. The algorithm initially starts with empty $V'$ and empty $E'$. Then, the algorithm proceeds by adding $r$ to $V'$. Afterwards, the algorithm adds to $E'$, the edge of least-weight connecting a vertex $v'$ in $V'$ to a vertex $v \notin V'$. Following this, $v$ is added to $V'$. The whole procedure is then repeated until all vertices in $V$ are added to $V'$. [2]

An important question that must be answered is whether the procedure described in the previous paragraph does return a minimum weight spanning tree. Typically, one needs to show that the mathematical structure returned by Prim's algorithm is that of a spanning tree, whose weight is the least possible among all spanning trees of the input graph. Note that, the answer to this question is vital for algorithms that use Prim's algorithm to produce a minimum weight spanning tree. In fact, if Prim's algorithm does not return a minimum weight spanning tree, the algorithms that use Prim's algorithm need to use another algorithm to compute a minimum weight spanning tree. The next two theorems

confirm that given an arbitrary complete weighted graph $G$ together with an arbitray starting vertex $r$ as input, Prim's algorithm always returns a minimum weight spanning tree of $G$. Theorem 2.2.6 was constructed using ideas from [23].

**Theorem 2.2.6.** *Prim's algorithm always returns a spanning tree.* [24]

*Proof.* Let $G(V, E, f)$ be a complete weighted graph, and $T(V', E', f)$ be the graph returned by Prim's algorithm when given $G$ and an arbitrary vertex $s \in V$ as input. Note that $s$ is the starting vertex for Prim's algorithm. To show that $T$ is a spanning tree, it must be shown that $T$ is a connected spanning sub graph of $G$ with no cycles. Note that $T$ is a sub graph of $G$ because, Prim's algorithm constructs $T$ using vertices and edges that belong to $G$.

Suppose that $T$ does not span $G$. Then, there exists a non-empty set of vertices $X \subseteq V$ such that, $X \cap V' = \emptyset$. Let $x \in X$. Then, $x \notin V'$ and hence, there are no edges in $E$ of the form $\{v, x\}$ where $v \in V'$, otherwise, Prim's algorithm would have chosen the edge and added $x$ to $V'$. Now, since $G$ is connected, there exists a path $P$ in $G$ joining $s$ to $x$. In addition to this, $s \in V'$ and $x \in X$. Hence, $P$ must contain an edge of the form $\{v, x\}$, where $v \in V'$. This means that $E$ contains an edge of the form $\{v, x\}$, where $v \in V'$. Contradiction. Therefore, $T$ must be a spanning sub graph of $G$.

To show that $T$ is connected, it must be shown that any two distinct vertices in $V'$ are connected by a path. Let $u$, $v \in V'$ such that, $u \neq v$. Now, before Prim's algorithm adds $u$ to $V'$, it connects it with a vertex $u_k$ which is already in $V'$. But before $u_k$ was added to $V'$, Prim's algorithm must have connected $u_k$ to a vertex $u_{k-1}$ that is already in $V'$. Repeat this argument until reaching a vertex $z$, which is not connected to another vertex from a previous iteration of Prim's algorithm. Note that a vertex $z$ exists because, since there are a finite number of vertices in $V$, there are a finite number of vertices in $V'$ that could have been chosen before $u$. From the description of Prim's algorithm above, this argument terminates when $z = s$. This is true because, $s$ is the only vertex in $V$ which was not connected to another vertex in $V'$ prior to being added to $V'$. Now, suppose that $k$ vertices were added to $V'$ before $u$. Then, $s = u_0, u_1, u_2, ..., u_k, u_{k+1} = u$ must be a path in $T$ joining $s$ to $u$ because, $\{u_i, u_{i+1}\} \in E'$ for all $i \in [k]$. Now, the same argument can be repeated for $v$ and hence, $s$ and $v$ must also be connected by a path $P'$. Therefore, since both $u$ and $v$ connect to the same vertex $s$ via a path, we can start from a vertex $u$ ($v$), travel to $s$ using $P$ ($P'$), and travel to $v$ using $P'$ ($P$). Therefore, $u$ and $v$ must be connected by a path. Hence, $T$ must be a connected spanning sub graph of $G$.

Suppose that $T$ contains a cycle $C$. Let $\{c_1, c_2\}$ be the last edge of $C$ that was added by Prim's algorithm to $E'$ (hence the edge that caused the cycle to be created). WLOG, suppose that $c_1$ was added to $V'$ by Prim's algorithm before $c_2$. Then, since Prim's algorithm adds an edge between a vertex in $V'$ and a vertex not in $V'$, it must be that $c_2$ is not in $V'$ before $\{c_1, c_2\}$ is added to $E'$. Hence, before $\{c_1, c_2\}$ is added to $E'$, there are no paths in $T$ joining $c_1$ to

$c_2$. This means that, no cycle is created when adding $\{c_1, c_2\}$ to $E'$ because, $c_1$ would be connected to $c_2$ by exactly one path (the path $c_1, c_2$). Contradiction, since $\{c_1, c_2\}$ is the edge that caused the cycle to be created. Therefore, $T$ must be a connected spanning sub graph of $G$ without cycles, hence, by Definition 1.1.19, $T$ is a spanning tree of $G$. □

Theorem 2.2.6 confirms that Prim's algorithm returns a spanning tree. Now, it must be shown that the spanning tree returned by Prim's algorithm has the least possible weight among all spanning trees of $G$. Theorem 2.2.7 was adapted using ideas from [24].

**Theorem 2.2.7.** *Prim's algorithm always returns a minimum weight spanning tree.* [24]

*Proof.* Let $G(V, E, f)$ be a complete weighted graph, and $T(V', E', f)$ be the spanning tree returned by Prim's algorithm when given $G$ and an arbitrary vertex $s \in V$ as input. Note that, $s$ is the starting vertex for Prim's algorithm. Let $T^\star(V^\star, E^\star, f)$ be a minimum weight spanning tree of $G$. If $T = T^\star$, the theorem holds.

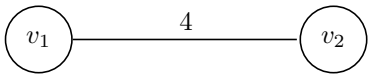
Suppose that $T \neq T^\star$. Then, $E' \setminus E^\star \neq \emptyset$. Let $e = \{v_1, v_2\} \in E' \setminus E^\star$. Let $S \subset V'$ be the set of vertices in $V'$ just before Prim's algorithm adds the edge $\{v_1, v_2\}$ to $E'$. Although $e \notin E^\star$, there is a path $P$ joining $v_1$ to $v_2$ in $T^\star$, because, $T^\star$ is a spanning tree. Now, by construction of $S$, $v_1 \in S$ and $v_2 \notin S$. Hence, P is a sequence of vertices beginning with vertices in $S$, and ending with vertices which are not in $S$. Therefore, there exists some edge $e' = \{x, y\} \in E^\star$ such that $x \in S$ and $y \notin S$. Now, in the iteration of Prim's algorithm when $V'$ $= S$, both $e$ and $e'$ connect a vertex which is in $V'$ to a vertex which is not in $v'$. Hence, both $e$ and $e'$ are eligible to be chosen by Prim's algorithm at this stage. However, it is known that $e$ is chosen instead of $e'$. This means that, $e$ is an edge of lesser weight than $e'$. Now, if the edge $e'$ is removed from $E^\star$ and replaced with the edge $e$, we get a path $P''$ in $T^\star$ joining $x$ to $y$. Hence, $(T^\star \setminus P) \cup P''$ would still remain a spanning tree. Let $T^\sim$ be the spanning tree $(T^\star \setminus P) \cup P''$. Then, $T^\sim$ differs from $T^\star$ exactly in the edges $e$ and $e'$, where, $f(e) \geq f(e')$. This means that, $T^\sim$ has weight at most that of $T^\star$. Now, since $T^\star$ is a minimum spanning tree, it must be a spanning tree of least weight. Hence, $f(T^\sim)$ $= f(T^\star)$. Therefore, if this procedure is repeated for every edge in $E' \setminus E^\star$, $T^\star$ would be converted into a spanning tree $W' = T$, where $f(T) = f(W') = f(T^\star)$. Therefore, $T$ must be a minimum weight spanning tree. Note that this holds because, the procedure can be repeated only for a finite number of times. This is true because, since $G$ is a finite graph, $|E' \setminus E^\star|$ is a finite number. □

What follows is an example that demonstrates how Prim's algorithm constructs a minimum weight spanning tree of a graph.

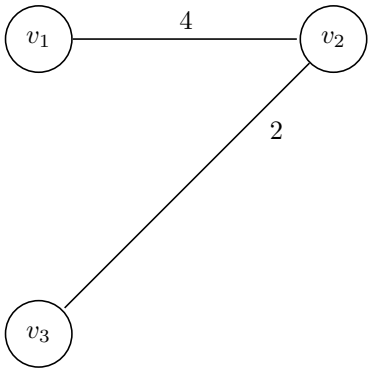
**Example 2.2.8.** Consider the graph $G$ in Example 1.1.16, and the brief description of Prim's algorithm. Let $T(V', E')$ be the graph with an empty set of vertices, and an empty set of edges. Assuming that the starting vertex is $v_1$,

Prim's algorithm starts by adding $v_1$ to $V'$. Hence $T$ is now the following graph:

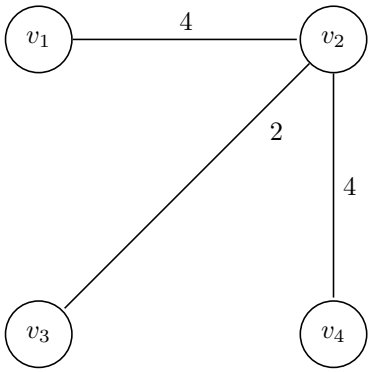

The algorithm now proceeds by adding the edge of least weight from $E$ to $E'$, which connects a vertex $v' \in V'$ to a vertex $v \notin V'$. Also, $v$ is added to $V'$. In this example, these properties are satisfied by the edge $\{v_1, v_2\}$, hence, $T$ is the following graph.




Since not all vertices in $V$ have been added to $V'$, the same procedure is repeated. Hence, the edge $\{v_2, v_3\}$ is chosen because, it is the edge of least weight connecting a vertex in $V'$ to a vertex not in $V'$. Therefore, $T$ is the following graph




Again, since not all vertices in $V$ have been added to $V'$, the same procedure is repeated. In this case, $\{v_2, v_4\}$ is the edge of least weight connecting a vertex in $V'$ to a vertex not in $V'$. Therefore, $T$ is the following graph.




Since all vertices in $V$ have been added to $V'$, the algorithm now terminates by returning $T$.

The second algorithm used by the TAMSA is the preorder tree walk algorithm, more commonly known as the depth first traversal tree algorithm. Given a tree $T(V, E, f)$ and an arbitrary starting vertex $r \in V$ as inputs, the preorder tree walk algorithm returns a walk $S$ of $T$ containing all vertices in $V$, where the vertices are ordered according to how they were visited while performing the walk. The preorder tree walk algorithm computes a walk $S$ in the following way. At the start of the algorithm, $S$ is empty. Then, starting from $r$, the algorithm marks $r$ as visited and adds $r$ to $S$. Following this, the algorithm recursively calls itself on all unvisited neighbors of $r$. The entire procedure is repeated until all vertices in $V$ are added to $S$. Note that in this section, the execution of the preorder tree walk algorithm on an arbitrary tree will be termed as a preorder walk. For example, a preorder walk of the tree in Figure 1 below, results in the vertex sequence $v_1, v_2, v_3, v_4, v_5, v_9, v_6, v_7, v_8$. [7]

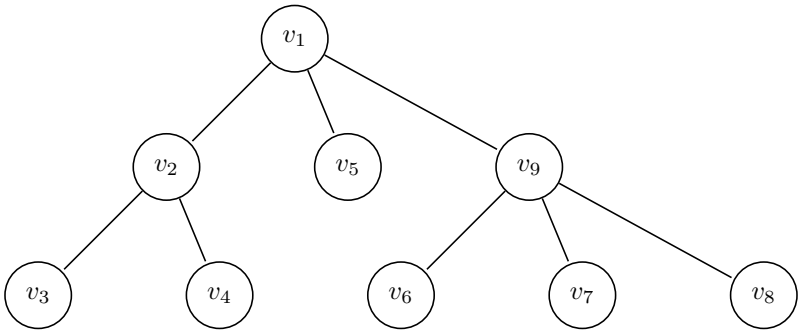



Figure 1: Tree example

Having briefly described important algorithms that will be used by the TAMSA

algorithm, now it is time to describe the TAMSA itself. The TAMSA takes an instance of the TSP as input, and returns a sequence of vertices representing a Hamiltonian cycle [7]. The TAMSA algorithm can be described using the pseudo code below which was adapted from [7].

**TAMSA(**$G(V, E, f)$**):**

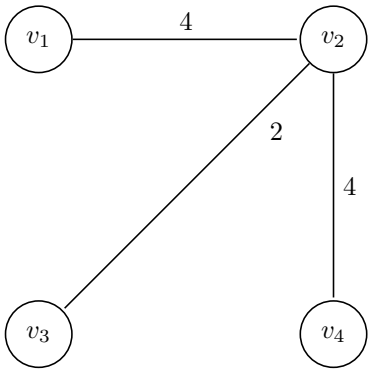
- Choose an arbitrary vertex $r \in$ V as starting vertex.
- Starting from $r$, compute a minimum weight spanning tree $T$ of $G$ using Prim's algorithm.
- Perform a preorder walk on T starting from $r$, and let $S$ be the walk returned by the preorder tree walk algorithm.
- Add $r$ to $S$ as a final vertex and return S.

For the TAMSA to be a useful approximation algorithm that approximates the TSP, it must guarantee that it will return a Hamiltonian cycle, and that the Hamiltonian cycle is computed in polynomial time. Let $G$ be an instance of the TSP, and let $T$ be the minimum weight spanning tree of $G$ returned by Prim's algorithm. By Theorem 2.2.7 and Theorem 2.2.6 it is guaranteed that $T$ is a minimum weight spanning tree. Now, let $S$ be the sequence of vertices returned by a preorder walk on $T$. Since $T$ has no cycles and, visited vertices can never be visited again, a preorder tree walk of $T$ can never result into a walk with repetition of vertices. Hence, $S$ has no repetition of vertices. In addition to this, since $T$ is connected, the preorder tree walk algorithm terminates when all vertices of $G$ have been added to $S$. Therefore, $S$ must contain all vertices of $G$. Hence, $S$ must be a path containing all vertices of $G$. As a result, for $S$ to be a Hamiltonian cycle, the first vertex in $S$ must be repeated as a final vertex. This is done by bullet point number four in the TAMSA pseudo code. Therefore, it can be concluded that $S$ must be a Hamiltonian cycle. The next step is to check that the TAMSA computes a Hamiltonian cycle in polynomial time.

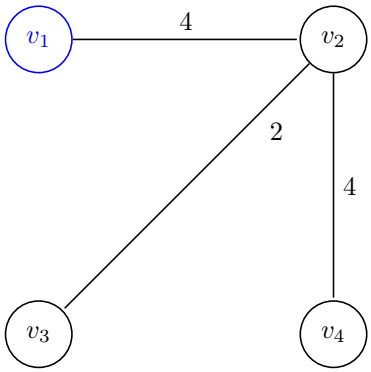
Given a TSP instance $G(V, E, f)$, the time complexity of the TAMSA is $O(|V|^2)$ [7]. This is true because of the following. Consider the TAMSA pseudo code above. Clearly, the largest amount of work to be performed by the TAMSA is to compute a minimum weight spanning tree $T$ of $G$, and to compute a preorder tree walk of $T$. The preorder tree walk of $T$ takes $O(|V|)$ steps because, the preorder tree walk algorithm terminates when all the vertices in $G$ have been visited (this is true due to reasons mentioned in the previous paragraph). Hence, bullet point number three takes $O(|V|)$ time. Now, according to Exercise 23.2-2 in [7], there is an implementation of Prim's algorithm that takes $O(|V^2|)$ time in order to compute a minimum weight spanning tree. Thus, the total time complexity of the TAMSA is $O(|V^2|) + O(|V|) = O(|V^2|)$. However, in [7] it is shown that the asymptotic time complexity of Prim's algorithm can be improved to $O(|E|log(|V|))$ by using a binary min-heap. This would mean that the time complexity of the TAMSA is no longer $O(|V^2|)$, but is $O(|E|log(|V|))$. As

a result, the TAMSA has a log-linear time complexity. Therefore, using Table 1 in Section 1.2, and and the fact that the NNA has a quadratic time complexity (discussed in Section 2.1.1), the TAMSA is more efficient asymptotically because, it has a smaller order of growth. Note that, the analysis of the time complexity of Prim's algorithm is not given because, it requires theory about data structures such as Heaps to be established, and this is out of the scope of this project. What follows is an example which demonstrates how the TAMSA computes a Hamiltonian cycle.

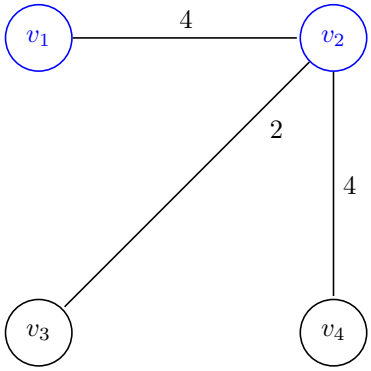
**Example 2.2.9.** Consider the graph $G$ in Example 1.1.16, and the TAMSA pseudo code presented in this section. Assume that randomly, the TAMSA chooses vertex $v_1$ as starting vertex. From Example 2.2.8, the minimum spanning tree $T$ returned by Prim's algorithm starting from vertex $v_1$ is the graph below.



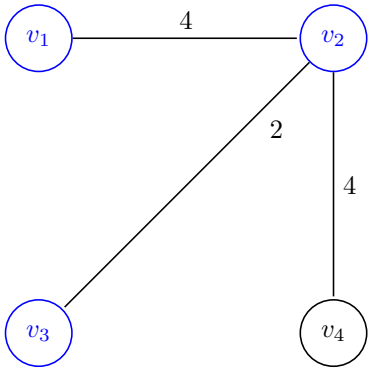

Now, according to bullet point number three in the TAMSA pseudo code, the TAMSA must perform a preorder tree walk on $T$. Consider the brief description of the preorder tree walk algorithm. A preorder walk of $T$ is computed in the following way. Note that, in the following demonstration, the blue vertices are the vertices that have already been visited by the preorder tree walk algorithm, and the black vertices are those vertices that have not yet been visited by the preorder tree walk algorithm.
The algorithm starts by visiting the starting vertex $v_1$ and adding it to $S$. Therefore, $S = v_1$.



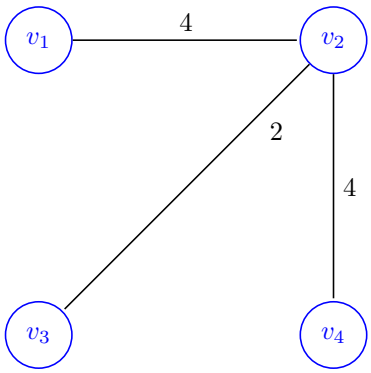

Since not all vertices have been visited, the algorithm recursively calls itself on the unvisited neighbors of $v_1$, which in this case is only $v_2$. Hence, $v_2$ is visited and added to $S$. Therefore $S = v_1, v_2$.




Since not all vertices have been visited, the algorithm recursively calls itself on the unvisited neighbors of $v_2$, which in this case are $v_3$ and $v_4$. Assuming that $v_3$ is visited before $v_4$, $v_3$ is marked as visited and added to $S$. Therefore $S = v_1, v_2, v_3$.



Since not all vertices have been visited, and $v_3$ has no neighbors, the algorithm now visits another neighbor of $v_2$ which in this case is $v_4$. Hence $v_4$ is marked as visited and added to $S$. Therefore $S = v_1, v_2, v_3, v_4$.




Since all vertices have been visited, the TAMSA adds $v_1$ to $S$ and returns it. Hence the TAMSA returns the Hamiltonian cycle $v_1, v_2, v_3, v_4, v_1$ of weight 23.

From Example 2.2.9 and Example 2.2.1 one can observe that the NNA and the TAMSA returned the same Hamiltonian cycle. Also similarly to the NNA, the TAMSA returns bad approximations for the graph presented in Example 2.2.2. This is true because of the following. Suppose that $G(V, E, f)$ is the graph in Example 2.2.2, and $v_1$ is the vertex chosen randomly by the TAMSA in bullet point number one of the TAMSA pseudo code, when given $G$ as input. The minimum weight spanning tree of $G$ is the same as the minimum weight spanning tree of the graph $G'$ in Example 1.1.16. This is true because, $G$ and $G'$ differ only in the weight of the edge $\{v_3, v_4\}$, were this edge is not included in the minimum weight spanning tree of $G'$. Hence, if $f(\{v_3, v_4\})$ is very large (greater than 10), and the other edges weights remain the same, the edge $\{v_3, v_4\}$ will not be included in the minimum weight spanning tree of $G$ and hence, other edges will be used (which are identical to those of $G'$). Since the minimum weight spanning tree of $G$ is the same as that of $G'$, then by Example 2.2.8, the TAMSA will return the Hamiltonian cycle $H = v_1, v_2, v_3, v_4, v_1$ when given $G$ as input. Now, since $f(\{v_3, v_4\})$ is very large and $\{v_3, v_4\}$ is an edge in H, the weight of $H$ would be very large. As a result, the weight of $H$ would differ a lot from the optimal solution. Note that, this problem would not happen if the preorder traversal is started from vertex $v_4$. In fact, the Hamiltonian cycle returned by the TAMSA would be $H = v_4, v_2, v_3, v_1, v_4$ (assuming $v_3$ is visited before $v_1$ when considering $v_2$), where the edge $\{v_3, v_4\}$ of weight $M$ is not in $H$. Therefore, from this paragraph it can be concluded that the quality of the approximation returned by the TAMSA depends on the starting vertex of the preorder walk.

Consider the TSP as defined in Definition 2.1.1. Similarly to the NNA, the

approximation ratio of the TAMSA when applied to the TSP is unknown. In addition to this, Theorem 2.1.7, guarantees that the TAMSA is not a constant approximation algorithm for the TSP, unless P=NP. Therefore, since it is not known whether P=NP, it is not known if the TAMSA is a constant approximation algorithm for the TSP. However, what is known is that the TAMSA is a 2-approximation algorithm for the Metric-TSP [7]. This fact is proved in Theorem 2.2.10 below. Before proving Theorem 2.2.10, the concept of a full walk will be explained because, it is going to be used in the proof.

Consider the preorder tree walk algorithm description provided in this section. Let $S$ be an arbitrary Hamiltonian cycle returned by the preorder tree walk algorithm. Suppose that, the preorder tree walk algorithm is currently at a vertex $u$. According to the preorder tree walk algorithm description, the algorithm first marks $u$ as visited, adds $u$ to $S$, and then it recursively calls itself on all the unvisited neighbors $v_1, ..., v_k$ of $u$. From the preorder tree walk algorithm description it can be deduced that, the algorithm returns to $u$ when all unvisited vertices in the subgraph connected to the vertex $v_i$ for all $i \in [k]$, are visited. In a preorder tree walk, when returning back to a vertex $v$, $v$ is not added again to $S$. To the contrary, in a full walk, when returning to a vertex $v$, $v$ is added again to $S$. Therefore, a full walk of a tree adds every vertex $v$ to $S$ whenever $v$ is first visited, and when $v$ is returned to after visiting one of the subgraphs connected to $v$ [7]. In other words, a full walk is a preorder tree walk which lists a vertex every time it is visited. For example, a full walk of the tree in Figure 1 of this section would result into the walk, $v_1, v_2, v_3, v_2, v_4, v_2, v_1, v_5, v_1, v_9, v_6, v_9, v_7, v_9, v_8, v_9, v_1$. Note that in a full walk, each edge is traversed exactly twice. This is true because, at each vertex $v$, the algorithm uses an edge $e$ to traverse to a neighbor $w$ of $v$, and uses the same edge $e$ when returning back to the vertex $v$, after all the unvisited vertices in the subgraph connected to $w$ are visited. Having described what a full walk is, it is now time to prove that the TAMSA is a 2-approximation algorithm when applied to the Metric-TSP. Note that the proof for Theorem 2.2.10 below was adapted using ideas from [7].

**Theorem 2.2.10.** *The TAMSA is a 2-approximation algorithm for the Metric-TSP* [7].

*Proof.* Let $G(V, E, f)$ be an instance of the Metric-TSP, and $H^\star$ be a minimum weight Hamiltonian cycle of $G$. A spanning tree $T^\star$ can be obtained from $H^\star$ by deleting an edge from $H^\star$. This is true because, when deleting an edge from $H^\star$, no vertices of $G$ are deleted, hence, all vertices in $V$ would still be in $T^\star$. In addition to this, $T^\star$ has no cycles because, it contains less edges than vertices. As a result, by Definition 1.1.19, $T^\star$ is a spanning tree. Now, since every edge weight is positive, deleting an edge from a Hamiltonian cycle would result into a spanning tree of weight less than that of the Hamiltonian cycle. Furthermore, the weight of any spanning tree is greater than the weight of the minimum weight spanning tree. Hence, the weight of the minimum weight spanning tree gives a lower bound on the weight of the minimum weight Hamiltonian cycle.

Therefore, letting $T$ be the minimum weight spanning tree of $G$, the following inequality holds:

$$\begin{aligned} f(T) \leq f(T^\star) \leq f(H^\star) \\ \implies f(T) \leq f(H^\star) \end{aligned} \tag{1}$$

Now, let $W$ be a full walk of $T$. It was argued in this section that, a full walk of a tree $T'$ uses an edge of $T'$ exactly twice. Hence,

$$\begin{aligned} f(W) = 2f(T) \leq 2f(H^\star) \text{ (By Equation 1)} \\ \implies f(W) \leq 2f(H^\star) \end{aligned} \tag{2}$$

However, $W$ is not a Hamiltonian cycle because it contains repetition of vertices. Now, since $G$ satisfies the triangle inequality, a path $P = u, v$ in $G$, has less weight than a path $P' = u, c, v$ in $G$, $u, c, v \in V$. This means that, since a full walk returns to a vertex $u$ from a vertex $c$ and traverses to a vertex $v$, the full walk has a larger weight than that of traversing directly from $c$ to $v$ in $G$. Hence, by removing repetition of vertices in $W$, a sequence of vertices $H$ with the following properties is obtained. The first property is that $H$ is a Hamiltonian cycle. This is true because, $H$ contains every vertex exactly once (excluding the starting vertex of the cycle). The second property is that $H$ has a smaller weight than $W$. This is true because of the triangle inequality implications described in this paragraph. Due to the second property, the inequality below is satisfied.

$$f(H) \leq f(W) \tag{3}$$

As described in this section, the difference between a full walk and a preorder walk is that in a preorder walk, vertices are not listed again when they are returned to. Hence, a preorder walk on a tree $T'$ is equivalent to performing a full walk on a tree $T'$ and deleting the repetition of vertices. This means that $H$ is the output of the preorder tree walk algorithm when given $T$ as input. Hence, $H$ is the Hamiltonian cycle outputted by the TAMSA. Therefore, from Equation 3 and Equation 2 it can be concluded that,

$$\begin{aligned} f(H) \leq f(W) \text{ (By Equation 3)} \\ \implies f(H) \leq 2f(H^\star) \text{ (By Equation 2)} \end{aligned} \tag{4}$$

Hence, from Equation 4 it can be concluded that, the value of the approximate solution returned by the TAMSA is within a constant factor of 2 of the optimal value. Hence, the TAMSA is a 2-approximation algorithm for the Metric-TSP. □

Therefore, Theorem 2.2.10 guarantees that, the value of the approximation returned by the TAMSA when applied to the Metric-TSP, is never more than twice the optimal value, even for very large instance sizes. Recall that, this was not the case for the NNA. In fact, in Section 2.1 it was shown that, the approximation ratio for the NNA when applied to the Metric-TSP increases

logarithmically with respect to the input size. That being said, it is stated in [7] that although the TAMSA has a constant approximation ratio, it is not the best practical choice for the Metric-TSP. In fact, Christofides gave a 3/2-approximation algorithm for the Metric-TSP. However, this will not be discussed in this project because, Christofides' algorithm requires presenting mathematical concepts which are out of the scope of this project. Such mathematical concepts are those of a minimum weight perfect matching.

In Section 2.1.1 and Section 2.1.2, the aim was to present algorithms that approximate the TSP, and rigorously prove their approximation ratios when applied to a special case of the TSP (Metric-TSP). Although knowing the approximation ratio of an algorithm is important, it is more convenient in practice to design an algorithm that is guaranteed to find an optimal solution if given enough time, even if it's approximation ratio is not known. This statement will become clearer when comparing the algorithm's performance in Chapter 3. An example of an algorithm that is guaranteed to find an optimal solution if given enough time is the Ant Colony Optimization Algorithm. This algorithm will be presented in the next subsection, and the main result in this section guarantees that the Ant Colony Optimization Algorithm will always find an optimal solution if given enough time (although the time may be infinite).

### 2.2.3 The Ant Colony Optimization Algorithm

Before describing the Ant Colony Optimization (ACO) algorithm, it is important to define two classes of algorithms. One important class of algorithms is that of heuristics. Heuristics are algorithms that seek to obtain good approximations of a problem in a short time (polynomial) [25]. From what have been discussed so far, one can deduce that the TAMSA and the NNA are heuristic algorithms because, they try to obtain good approximations of the TSP in polynomial time. The other important class of algorithms is that of metaheuristics. A metaheuristic is a set of algorithmic notions that can be used to define heuristic methods for a wide variety of problems [25]. In other words this means that, a metaheuristic algorithm is a general framework that can be used to define different heuristics which may not solve the same problem. Note that a metaheuristic is different than a heuristic because, heuristic algorithms are specifically written to solve only one problem. For example, the NNA and the TAMSA can only approximate the TSP. Some examples of metaheuristic algorithms are Tabu Search, Simulated Annealing, Evolutionary Computing, and the ACO algorithm [25].

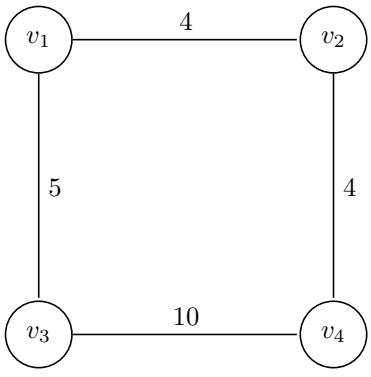
The ACO algorithm is metaheuristic algorithm [25]. The ACO algorithm's framework was inspired from the behavior of ants. Ants are capable of finding the shortest path from a food source to their nest and vice-versa without using any visual indications. Although ants do not use visual cues, it does not mean that they cannot communicate. In fact, ants deposit a chemical known as pheromone when they walk, and they probabilistically move into directions

which are rich in pheromone. It is important to note that an ant may still move to a direction which is not rich in pheromone, however, the probability of doing so is very small. In fact, the movement of ants in directions which are not rich in pheromone is important in real ant colonies because, it helps finding shorter paths from a food source to the nest if for example a sudden change in the environment causes a new shorter path to be created. [26], [25]

The reason why the natural behaviour of ants makes them choose shorter paths from a food source to their nest is because, pheromone accumulates quicker on shorter paths. Pheromone accumulates quicker on shorter paths because of the following. Suppose that ants start to wander randomly through different directions in their environment starting from a food source. Obviously, ants that choose shorter paths to their nest will arrive much quicker than the ants that choose longer paths. This means that, on the way back to the food source, the ants that used the shorter paths will smell more pheromone on the shorter paths than on the longer paths because, the ants that choose the longer paths would have still not arrived at their nest. In addition to this, pheromone evaporates. Hence, the longer a path is, the more evaporation it experiences until ants reach their destination. As a result, the ants would prefer probabilistically to move back to the food source using the shorter paths because they simply would contain more pheromone. This means that more pheromone continues to accumulate on the shorter paths, and evenetually this will result in the majority of the ants to choose the shortest path in order to travel from their nest to a food source and vice-versa. [26], [25]

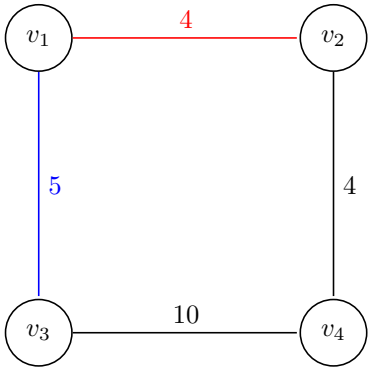
What follows is an example that demonstrates how the ACO metaheuristic can be applied to the Shortest Path Problem defined in Example 1.2.3.

**Example 2.2.11.** Suppose that the shortest path from vertex $v_1$ to vertex $v_4$ of the graph $G$ below must be found.

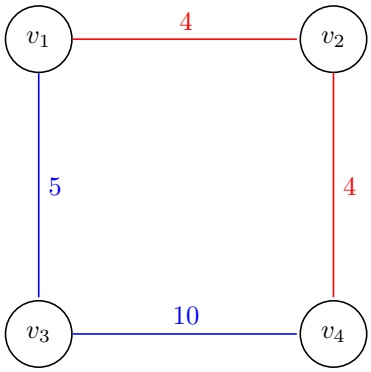



Clearly, the ACO framework can be applied to the Shortest Path Problem on $G$ by letting the ant's environment be $G$, and the food source and the nest be $v_1$ and $v_4$ respectively. The ants would then find the shortest path due to

the following. Since no ant has yet started to walk in it's environment, no pheromone has yet been deposited. Hence, for every ant starting at vertex $v_1$, $v_2$ and $v_3$ have the same probability of being chosen because no pheromone is yet deposited on the edges $\{v_1, v_2\}$, $\{v_1, v_3\}$. Hence, assuming that all ants at $v_1$ move at the same time, half of the ants probabilistically choose to move to $v_2$ and half of the ants probabilistically chose to move to $v_3$.




Suppose that each ant does not visit vertices that it has already visited. This can be easily done in the ACO algorithm implemented for the Shortest Path Problem by giving each ant a list of vertices that it has already visited. Then, ants at $v_3$ and $v_2$ have only one vertex to visit because $v_1$ has already been visited. Therefore, all ants from $v_2$ and $v_3$ move to vertex $v_4$.




Now, since the red path is shorter than the blue path, the ants following the red path arrive at vertex $v_4$ before the ants following the blue path. Hence, on the way back to vertex $v_1$, the red path has more pheromone than the blue path because, more pheromone is evaporated on the longer path until all ants arrive at $v_4$. Therefore, the majority of the ants at $v_4$ would probabilistically choose to travel to $v_1$ using the red path. As a result, the red path keeps on accumulating more pheromone than the blue path and eventually this would lead to a

situation were almost all ants would choose the red path when traveling from $v_1$ to $v_4$ and vice-versa. To conclude, the ACO algorithm for the Shortest Path Problem given $(G, v_1, v_4)$ as instance can then return the best found path $P$ (in this case the red path) when either the majority of the ants are travelling through $P$ (i.e. the algorithm converges) or, it can return $P$ after a maximum number of iterations.

After describing the ACO framework, it is now time to describe how the ACO metaheuristic can be applied to the TSP. Let $G(V, E, f)$ be an instance of the TSP. As described in Example 2.2.11, the ant's environment can be represented by $G$. In addition to this, since the goal of the TSP is to find a minimum weight Hamiltonian cycle, it does not matter which vertex represents the food source and which vertex represents the nest, because, all vertices have to be visited. What is important is that when an ant starts traversing vertices in a graph, it doesn't move to vertices that it has already visited. This can be done as discussed in Example 2.2.11 above, by giving each ant a working memory which is used to remember which vertices it has already visited. Note that although real ants do not posses a working memory, the working memory is vital for the artificial ants to compute a Hamiltonian cycle, otherwise, ants may produce walks with repetition of vertices. Another important thing is that ants must keep the order of how vertices are visited. This can be easily done by always adding a new vertex that an ant visits at the end of the working memory list. As a result, the working memory would contain the order of how the vertices were visited by an ant. In what follows, the fundamental components and logic of the ACO algorithm for the TSP will be described. [26], [25]

Suppose that $G$ is a TSP instance. At the start of the ACO algorithm, each ant is placed on a random vertex of $G$. At each time step, every ant is moved to a vertex of $G$ which is not in the ant's working memory until all vertices are visited by each ant. The new vertex that each ant must move to is determined using a probabilistic function. As discussed so far in this subsection, the ants must probabilistically prefer to move in directions with high pheromone. Hence, at each time step, the probabilistic function must be in such a way that every ant should probabilistically prefer to choose vertices, which are connected via edges to the vertex the ant is currently on with high pheromone. However, since in the TSP a minimum weight Hamiltonian cycle must be found, ants must also posses the capability to choose shorter edges. Therefore, the probabilistic function must be a function of both the pheromone accumulated on edges, and of the length of the edges connecting the current vertex the ant is on with a candidate vertex, where, the higher the weight of an edge $e$ is, the less is the probability of $e$ being chosen by an ant. Due to this probabilistic function, ants would probabilistically prefer to move to vertices that are connected by edges with high pheromone and which are close-by. The aim of the probabilistic function will become more clear when it is presented later on in this section. [26], [25]

Another important part of the ACO algorithm for the TSP is local trail up-

dating. Local trail updating is the reduction of pheromone on the edges chosen by an ant at each time step. The aim of local trail updating is to reduce the pheromone of an edge $e$, after it is chosen by an ant so that $e$ is less desirable for other ants. This is important in the ACO algorithm because, it increases the possibility of ants exploring other edges that were not visited by any ant, and hence, lowers the probability of the ACO algorithm to converge to a sub-optimal solution of the TSP instance (all ants generating the same suboptimal Hamiltonian cycle). It is important to note that, local trail updating was inspired from pheromone evaporation in real ant colonies. In what has been discussed so far, it was argued that pheromone accumulates quicker on shorter paths in real ant colonies. Therefore, for the ACO algorithm to mimic this behavior, at every iteration, the ants that produce the shorter Hamiltonian cycles must reward more pheromone to the edges constituting their Hamiltonian cycles. As a result, there would be more pheromone present on edges that constitute the shorter Hamiltonian cycles. One way to go about this is to make the ant that produces the global best Hamiltonian cycle so far in the ACO algorithm add more pheromone on the edges constituting it's Hamiltonian cycle. Another way is to make the ant that produces the iteration's best Hamiltonian cycle deposit more pheromone on the edges constituting it's Hamiltonian cycle. According to [25], the former is better than the latter empirically. Hence, in what remains, global trail updating is performed by the ant that produces the global best Hamiltonian cycle. [26], [25]

It is important to note that there are different ways how the ACO algorithm can be applied to the TSP. The ACO algorithm described in the previous paragraphs is known as the Ant Colony System (ACS) in literature. The ACS was chosen because, it is one of the best ACO algorithms in terms of accuracy of approximations [25]. In addition to this, as it will be seen later in this section, the main theorem of this project also applies to the ACS. It is important to note that later on in this section, other variants of the ACO algorithm for the TSP will be briefly described by discussing how they vary from the ACS. The discussion will now proceed by presenting the pseudocode for the ACS in Algorithm 1. Note that Algorithm 1 was constructed using ideas from [26]. In Algorithm 1 below, $G(V, E, f)$ is an instance of the TSP, and $g$ is a probabilistic function with the properties described in the previous paragraph. Also, $a.current\_vertex$ is the variable containing the vertex the ant $a$ is currently on.

**Algorithm 1** : ACS($G(V, E, f)$)

1: $best\_ham\_cycle = \{\}$
2: $best\_ham\_cycle.weight = \infty$
3: **while** (!finished) **do**
4: Place $m$ new ants on $m$ randomly selected vertices in $V$.
5: **while** not all ants have completed a Hamiltonian cycle of $G$ **do**
6: **for** every ant $a$ **do**
7: move $a$ to vertex $v \in V$ chosen using the probabilistic function $g$.
8: Perform local trail updating on the edge $\{a.current\_vertex, v\}$
9: **end for**
10: **end while**
11: $A$ = ant which produces a minimum weight Hamiltonian cycle of this iteration.
12: **if** $best\_ham\_cycle.weight \geq A.Hamiltonian\_Cycle.weight$ **then**
13: $best\_ham\_cycle = A.Hamiltonian\_Cycle$
14: **end if**
15: Perform global pheromone update on the edges of $G$ constituting the Hamiltonian cycle $best\_ham\_cycle$
16: **end while**
17: Return $best\_ham\_cycle$

For Algorithm 1 to be implemented, there remains to define the probabilistic function $g$, the global trail updating formula, and the local trail updating formula. As stated in [26], there are different ways how the probabilistic function and the global/local trail updating formulas can be defined. However, the ones suggested by Dorigo and Gambardella in [26] will be presented. Let $G(V, E, f)$ be an instance of the TSP. An ant $k$ in vertex $r$ chooses the city $s$ to move to among those vertices which do not belong to its working memory $M_k$, by applying the probabilistic formula below.

$$s = \begin{cases} \underset{u \notin M_k}{\operatorname{argmax}} \left\{ [\tau(\{r,u\})]. \left[ \frac{1}{f(\{r,u\})} \right]^{\beta} \right\} & \text{if } q \leq q_0 \\ S & \text{otherwise} \end{cases} \tag{5}$$

where $\tau(\{r, u\})$ is the amount of pheromone on the edge $\{r, u\}$, $\beta$ is a parameter which weighs the relative importance of the pheromone level of an edge $\{r, u\}$ and the closeness of $u$ to $r$, $0 \leq q \leq 1$ is a random value chosen with uniform probability, $0 \leq q_0 \leq 1$ is another parameter, and $S$ is a random variable which is selected according to the following probability distribution:

$$p_k(r,s) = \begin{cases} \frac{[\tau(\{r,s\})].[\frac{1}{f(\{r,s\})}]^{\beta}}{\sum_{u \notin M_k} [\tau(\{r,u\})].[\frac{1}{f(\{r,u\})}]^{\beta}} & \text{if } s \notin M_k \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

where $p_k(r, s)$ is the probability of ant $k$ in city $r$ to choose city $s$.

The idea behind Formula 5 is the following. When an ant $k$ wants to move to a new vertex, it picks a random number $q$ in the interval [0,1], where each number in this interval has equal probability of being chosen. Then, if $q \leq q_0$, the edge with the largest amount of pheromone per edge weight is chosen. It is important to note that, the larger the amount of pheromone on an edge is, the more chance an edge has to be the maximum element in the argument list of formula 5. In addition to this, the smaller the weight of an edge is, the larger the value of $[\tau(\{r,u\})].\left[\frac{1}{f(\{r,u\})}\right]^{\beta}$ is. Therefore, it can be concluded that the case $q \leq q_0$ makes an ant choose the edge with the best combination of pheromone and closeness as required. In other words, the case $q \leq q_0$ forces an ant to choose the edge which has been chosen by the majority of ants. [26], [25]

When $q$ is greater than $q_0$, the ant randomly picks a vertex with probability defined by the probability distribution in Formula 6 above. The idea behind Formula 6 above is the following. If a vertex is already in the working memory of an ant, then it has no chance of being selected. Hence, it's probability is zero. This will ensure that ants will not produce a walk with repetition of vertices. If a vertex $s$ is not in the working memory of an ant $k$, the probability of ant $k$ on vertex $r$ to choose $s$ depends on the amount of pheromone on the edge $\{r,s\}$ per weight of the edge $\{r,s\}$, where, the larger this ratio is, the larger the probability of the vertex $s$ is to being chosen. Therefore this equation makes an ant probabilistically prefer edges with a combination of large pheromone edges, and small weight. However, unlike case $q \leq q_0$ in Equation 5, this does not mean that a vertex with a small probability cannot be chosen. This mimics the case when real ants move to directions which are not rich in pheromone. As discussed earlier, this is necessary in real ant colonies because it may help ants to find shorter paths from a food source to their nest. Similarly, the movement of artificial ants (ants in the ACO algorithm) to vertices with low probability (defined by Formula 6) may help the ACO algorithm to find Hamiltonian cycles of lesser weight. [26], [25]

Therefore, from the above two paragraphs it can be concluded that, an ant can either move to new vertices using experience gathered from previous ants with probability $q_0$ (using the largest amount of pheromone per edge weight), or it can perform a biased exploration towards edges of small weight but with high pheromone levels with probability $(1 - q_0)$. As a result, the value of $q_0$ determines the amount of exploration that will be performed by the ACS. According to [26], the best value of $q_0$ was empirically found to be 0.9. From this observation one can deduce that, it is more important for the ACS to gather information from previous ants than exploring other edges in the TSP instance. [26], [25]

The next formula that needs to be presented is the global trail updating formula. As discussed previously, the idea behind global trail updating is to reward edges belonging to shorter Hamiltonian cycles with more pheromone. As a result, since Equations 5 and 6 favor edges that have a lot of pheromone and of

least weight, shorter Hamiltonian cycles would have more chance of being chosen probabilistically by ants . Therefore, the amount of pheromone to be deposited on each edge of an iteration's best Hamiltonian cycle $H$ can be inversely proportional to the length of $H$. As a result, the smaller the weight of the edge $H$ is the more chance it has to be the minimum weight Hamiltonian cycle, hence the more pheromone is given, and the more chance it has to be taken by all ants. More formally, the global trail updating formula is the following. [26], [25]

$$\tau(\{r,s\}) = (1-\alpha).\tau(\{r,s\}) + \frac{\alpha}{\tau(\{r,s\})} \tag{7}$$

where $\tau(\{r,s\})$ is the amount of pheromone on edge $\{r,s\}$ and $\alpha$ is the evaporation rate which mimics evaporation of pheromone in real ant colonies. [26]

As discussed previously, for Algorithm 1 to be implemented, there remains to show what local trail updating formula will be taken. The local trail updating formula suggested by Dorigo and Gambardella [26] is the following:

$$\tau(\{r,s\}) = (1-\alpha).\tau(\{r,s\}) + \alpha.\tau_0 \tag{8}$$

where $\tau_0$ is a small value representing the amount of pheromone deposited on an edge when it is chosen by an ant, $\tau(\{r,s\})$ is the amount of pheromone on edge $\{r,s\}$, and $\alpha$ is the evaporation rate. It is important to note that $\alpha$ and $\tau_0$ are usually chosen in a way such that after the local trail updating formula is applied, the pheromone on a chosen edge is reduced. This is important because as discussed before, local trail updating is important because it avoids having all ants choosing the same edges, hence following the same paths which ultimately leads to all ants following the same Hamiltonian cycles. Therefore, local trail updating encourages exploration of the path space in the ant's environment. Again, the reduction of pheromone on edges is inspired from the evaporation of pheromone in a real ant colony setting. [26]

As already mentioned, the ACO algorithm for the TSP which was described in this section is referred to as the Ant Colony System in literature. However, there are many other types of ACO algorithms that can be applied to the TSP. Such ACO algorithms are the MAX-MIN Ant System, Rank-Based Ant System and Elitist Ant System. The main difference between all these ACO algorithms is how the pheromones are updated from one iteration to the next. However, these ACO algorithm variants will not be described in this project because, the main theorem of this section applies to a subset of the ACO algorithms TSP variants which include the ACS. A detailed explanation on these ACO algorithm variants is given in [25].

# 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] "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/

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

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

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

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

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

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

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

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

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

[13] 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

[14] "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

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

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

[17] "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

[18] V. V. Vazirani, *Approximation Algorithms*, 2nd ed. Springer-Verlag Berlin Heidelberg, 2003. [Online]. Available: https://doc.lagout.org/science/0_Computer Science/2_Algorithms/Approximation Algorithms [Vazirani 2010-12-01].pdf

[19] K. Arora, S. Agarwal, and R. Tanwar, "Solving tsp using genetic algorithm and nearest neighbour algorithm and their comparison," *International Journal of Scientific & Engineering Research*, vol. 7, no. 1, p. 10141018, Jan 2016. [Online]. Available: https://pdfs.semanticscholar.org/9fc2/8fac2603cfda11da21033a58bbb5c7b75e09.pdf

[20] A. A. Khan and H. Agrawal, "A comparitive study of nearest neighbour algorithm and genetic algorithm in solving travelling salesman problem," *International Research Journal of Engineering and Technology (IRJET)*, vol. 03, no. 05, p. 234238, May 2016. [Online]. Available: https://www.irjet.net/archives/V3/i5/IRJET-V3I550.pdf

[21] D. Rosenkrantz, R. Edwin Stearns, and P. M. Lewis II, "An analysis of several heuristics for the traveling salesman problem," *SIAM J. Comput.*, vol. 6, pp. 563–581, 09 1977.

[22] ——, "An analysis of several heuristics for the traveling salesman problem," *SIAM J. Comput.*, vol. 6, pp. 563–581, 09 1977.

[23] [Online]. Available: https://web.stanford.edu/class/archive/cs/cs161/cs161.1138/lectures/14/Small14.pdf

[24] [Online]. Available: http://www.oxfordmathcenter.com/drupal7/node/685

[25] M. Dorigo and S. Thomas, *Ant Colony Optimization*. MIT Press, 2004.

[26] M. Dorigo and L. M. Gambardella, "Ant colonies for the traveling salesman problem," *BioSystems*, 1997.

[27] E. Weisstein, "Hamiltonian cycle," Oct 2018. [Online]. Available: http://mathworld.wolfram.com/HamiltonianCycle.html

[28] "Travelling salesman problem — set 1 (naive and dynamic programming)," Sep 2018. [Online]. Available: https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/

[29] [Online]. Available: http://www.cse.msu.edu/ torng/360Book/Problems/

[30] E. Rowell, "Know thy complexities!" [Online]. Available: http://bigocheatsheet.com/

[31] E. W. Weisstein, "Mathworld–a wolfram web resource," Nov 2018. [Online]. Available: http://mathworld.wolfram.com/AdjacentVertices.html

[32] "Mathonline." [Online]. Available: http://mathonline.wikidot.com/complete-graphs

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

[34] E. W. Weisstein, "Connected graph." [Online]. Available: http://mathworld.wolfram.com/ConnectedGraph.html

[35] ——, "Vertex degree." [Online]. Available: http://mathworld.wolfram.com/VertexDegree.html

[36] P. E. Black, "optimization problem," Apr 2009. [Online]. Available: https://xlinux.nist.gov/dads/HTML/optimization.html

[37] "Np-complete problem." [Online]. Available: https://www.britannica.com/science/NP-complete-problem#ref97461

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

[39] Jun 1997. [Online]. Available: https://www8.cs.umu.se/kurser/TDBA77/VT06/algorithms/BOOK/BOOK2/NODE74.HTM

[40] T. Sttzle and M. Dorigo, "A short convergence proof for a class of aco algorithms," *Evolutionary Computation, IEEE Transactions on*, vol. 6, pp. 358 – 365, 09 2002.

[41] M. Dorigo, M. Birattari, and T. Sttzle, "Ant colony optimization," *Computational Intelligence Magazine, IEEE*, vol. 1, pp. 28–39, 12 2006.