# The Ant Colony Optimization Algorithm for the Travelling Salesman Problem

By

Dylan Galea

A Dissertation submitted in partial fulfillment of the requirements

for the Degree of Bachelor of Science(Honours)

in Mathematics as main area

DEPARTMENT OF MATHEMATICS

FACULTY OF SCIENCE

UNIVERSITY OF MALTA

May 2019

# Acknowledgements

I would like to thank everyone who either helped me academically or on a personal level throughout the writing of this dissertation. Firstly, I would like to express my deepest appreciation to my supervisor Professor Lauri, whose guidance and assistance was invaluable.

Secondly, I would like to thank my mother, father, brother, and my girlfriend for always being supportive. Without your constant love and support especially during hard times, I wouldn't have reached university let alone writing this dissertation.

# Abstract

The Travelling Salesman Problem is a classical NP-Complete problem. Being NP-Complete, no algorithm that can solve the Travelling Salesman Problem in feasible time is known. As a result, heuristic algorithms and metaheuristic algorithms must be used to approximate a solution in feasible time. A metaheuristic algorithm which can approximate the Travelling Salesman Problem in feasible time is the Ant Colony Optimization Algorithm. In this dissertation, a variant of the Ant Colony Optimization algorithm called the Ant Colony System will be theoretically analyzed. The main theoretical result states that if given enough time to execute, the ACS algorithm applied to the Travelling Salesman Problem always finds an optimal solution. That being said, this result does not specify how much iterations need to be executed for the algorithm to find an optimal solution. As a result, the Ant Colony System will also be empirically analyzed, where the main aim of this empirical analysis is to study how the ACS algorithm approaches the optimal value as the number of iterations are increased. Apart from this, another aim of the empirical analysis is to check whether the Ant Colony System gives better approximations than the Nearest Neighbour Algorithm and the Twice Around the Minimum Spanning Tree Algorithm, which are two classical heuristic algorithms.

After presenting the theory behind the algorithms mentioned in the previous paragraphs, these algorithms where implemented and applied to Travelling Salesman Problem instances defined in the 2-D Euclidean Space. The approximation results indicate that for Travelling Salesman Problem instances defined in the 2-D Euclidean Space, the Ant Colony System algorithm gives better approximations than the Nearest Neighbour Algorithm and the Twice Around the Minimum Spanning Tree algorithm. From the empirical analysis it was also concluded that, for TSP instances defined in the 2-D Euclidean Space, the Ant Colony System algorithm approaches the optimal value in a manner which is inversely proportional to the number of iterations executed. As a result it could be concluded that, although it is theoretically known that the Ant Colony System algorithm will find an optimal solution if given enough time, it may do so in infeasible time.

# Contents

# 1    Introduction

The Travelling Salesman Problem is a Mathematical problem that has been studied long since the 18th century by mathematicians Sir William Rowam Hamilton, and Thomas Penyngton Kirkman. In it's most general form, the Travelling Salesman Problem can be described as follows. Suppose that a salesman is given a number of cities where the cost of travelling between the cities is known. In the Travelling Salesman Problem, the salesman is asked to find a trip of least cost which goes through every city exactly once, and returns back to the starting city. Although the Travelling Salesman Problem is intuitively easy to understand, it has a number of important real life applications. For example, suppose that a company has distinct items stored at a different warehouse, and assume that an order consisting of different items is received from a customer. In addition to this, suppose that a truck is to pick up the items ordered from the different warehouses, and deliver the order to the customer. Due to expenditure in fuel and other deliveries, the company may need to find the shortest way to gather the distinct items for different warehouses, and deliver them to the customer. This problem can be easily formulated as the Travelling Salesman Problem by letting the salesman be the truck, the cities be the different warehouses, and the cost of travelling between cities be the cost of travelling from one warehouse to another. As a result, the company can determine the trip of least cost by solving the Travelling Salesman Problem. However, unfortunately no algorithm that can solve the Travelling Salesman Problem in feasible time is known. [9], [16]

Although no algorithm that can solve the Travelling Salesman Problem in feasible time is known, this does not mean that the problem cannot be approximated in feasible time [7]. Therefore, the aim of this dissertation is to theoretically and empirically analyze an algorithm called the Ant Colony Optimization Algorithm which can approximate the Travelling Salesman Problem in feasible time. The theoretical aim of this dissertation is to present an Ant Colony Optimization algorithm variant called the Ant Colony System algorithm, and then argue that if given enough time to execute, the Ant Colony System algorithm will always find an optimal solution when applied to the Travelling Salesman Problem [9]. It is important to note that, this result does not state how much iterations need to be executed for the Ant Colony System algorithm to find an optimal solution when applied to the Trav-

elling Salesman Problem. In addition to this, there are no results which state how many iterations need to be executed for the Ant Colony System to find an optimal solution for the Travelling Salesman Problem [9]. Hence, the main aim of the empirical analysis is to try and study how the Ant Colony System approaches towards an optimal solution of the Travelling Salesman Problem in practice. Another aim of the empirical analysis is to apply the Ant Colony System on specific instances of the Travelling Salesman Problem, and compare it's results to those obtained by the Twice Around the Minimum Spanning Tree, and the Nearest Neighbour algorithm. As a result, one can then deduce which algorithm should be used to approximate the Travelling Salesman Problem for the considered instances.

Therefore, this dissertation is divided into four main chapters. Firstly, the first chapter is split into 2 sections where, the first section presents the basic graph theoretic concepts which will be used throughout the dissertation. The second section will present the theory of computation which will help in formalizing the theory of NP-Completeness, and what it means when it is said that the Travelling Salesman Problem is NP-Complete. Secondly, the second chapter is divided into two sections. The first section presents the mathematical definition of the Travelling Salesman Problem, and properties belonging to the Travelling Salesman Problem. Afterwards, it will be shown that the Travelling Salesman Problem is NP-Complete, and hence approximation algorithms are needed to obtain sub-optimal solutions. The second section will then present two classical algorithms which are known to approximate the Travelling Salesman Problem. These algorithms are the Twice Around the Minimum Spanning Tree algorithm, and the Nearest Neighbour algorithm. In this section, the aim is to show that when applied to specific instances of the Travelling Salesman Problem, both algorithms return a solution with cost smaller than some upper bound. Thirdly, the third chapter presents the Ant Colony Optimization algorithm. In this chapter, the Ant Colony Optimization algorithm will be described, along with a variant of the algorithm called the Ant Colony System algorithm. The main aim of this section is to deduce that when applied to the Travelling Salesman Problem, the Ant Colony System will always find an optimal solution if given enough time to execute. Finally, the last chapter will present the empirical analysis of the Nearest Neighbour algorithm, the Twice Around the Minimum Spanning Tree algorithm, and the Ant Colony System algorithm. In this chapter, one of the aims is

to present the approximation results of the Nearest Neighbour algorithm, the Twice Around the Minimum Spanning Tree algorithm, and the Ant Colony System when applied to specific instances of the Travelling Salesman Problem. The results from different algorithms are then compared and hence it could be deduced which algorithm produces the best approximations for the considered instances. In addition to this, another aim of this chapter is to try and study how the Ant Colony System approaches towards an optimal solution for the considered Travelling Salesman Problem instances. The next chapter is the chapter about the graph theoretic concepts and computational theory which will be used in this dissertation.

# 2    Background Theory

As already discussed in Chapter 1, the main aim of this chapter is to present the background theory which will be needed throughout this dissertation. The background theory which will be needed is related to graph theory and computational theory. The following section starts by discussing the graph theoretic concepts which will be used throughout.

## 2.1    Some Graph Theory

Graph theory is the study of a mathematical structure called a graph. A graph can be defined formally as shown in definition 2.1.1 below.

**Definition 2.1.1.** *A graph G is a pair (V,E), where 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 [3]. A graph G defined by the pair (V,E) will be denoted as G(V,E) or G.*

A graph defined using Definition 2.1.1 is called an undirected graph. There is also the concept of a directed graph were $E \subseteq \{(u,v) : \forall u,v \in V, u \neq v\}$ [3]. However, in this dissertation it can be assumed that any graph that will be considered is undirected, and that for every vertex $u$, there are no edges of the form $\{u,u\}$, 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 terminologies, 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 2.1.2 below.

**Definition 2.1.2.** *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 $\{u,v\}$. [13]*

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

**Definition 2.1.3.** *A vertex v in a graph G is incident to an edge e in G if v is an end vertex of e.* [13]

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

**Definition 2.1.4.** *The degree of a vertex v in a graph G denoted by deg(v) is, the number of edges incident with v in G.* [13]

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 dissertation a graph is sometimes given formally using sets or as a pictorial representation, assuming that one can be converted into another. Example 2.1.5 below depicts how a graph can be represented pictorially, and illustrates Definitions 2.1.2, 2.1.3, and 2.1.4.

**Example 2.1.5.** 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 :
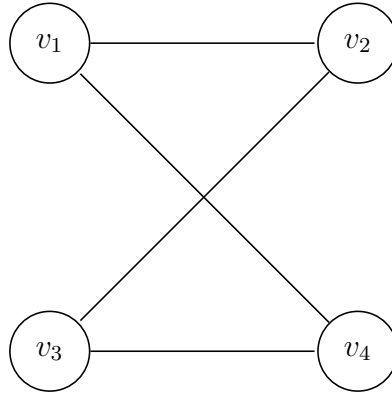


Using definition 2.1.2, two adjacent vertices in $G$ are $v_1$ and $v_2$. On the other hand, two non adjacent vertices in $G$ are $v_1$ and $v_3$.
By definition 2.1.3, $v_1$ and $v_2$ are incident to the edge $\{v_1, v_2\}$.
Using definition 2.1.4, the degree of every vertex in $G$ is 2.

There are many important graphs in graph theory, one of them being the complete graph on $n$ vertices.

**Definition 2.1.6.** *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 $K_n$.* [13]

Given any graph $G(V, E)$, one can also define graph theoretic structures that lie within $G$ such as walks and paths.

**Definition 2.1.7.** *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.* [13]

**Definition 2.1.8.** *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 2.1.8 can now be used to define cycles and connectivity in a graph.

**Definition 2.1.9.** *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.* [13]

**Definition 2.1.10.** *Given a graph G(V,E), a cycle in G is a path $u_1$, ..., $u_k$ together with the edge $\{u_1, u_k\}$, where k $\geq$ 3* [13].

By Definitions 2.1.7 and 2.1.8 above, it is clear that a path is a special instance of a walk. Similarly, from Definitions 2.1.8 and 2.1.10, 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, since according to definitions 2.1.7, 2.1.8 and 2.1.10, cycles and paths are formulated in terms of walks, then they could only be perceived as sequences of vertices and not actual graphs. However, this is not the case because cycles and paths 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 2.1.5, 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 dissertation. The usefulness of this remark will be seen when defining Hamiltonian cycles. Example 2.1.11 is constructed for better understanding of Definitions 2.1.6, 2.1.8, 2.1.9 and 2.1.10.

**Example 2.1.11.** Consider the graph $G(V,E)$ below:
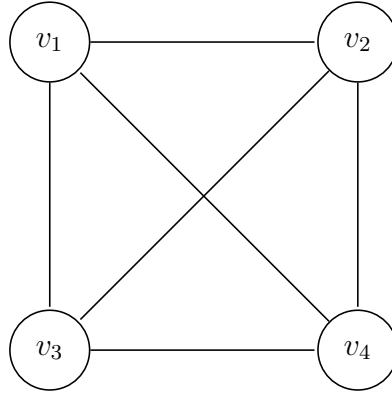


Since every vertex in $G$ is adjacent to every other vertex, then by Definition 2.1.6, $G$ must be complete. Hence, $G$ must be $K_4$. Since $G$ is complete, it must also be connected because, there is a path $P_2$ between any two distinct vertices of $G$.

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 2.1.12.** *Given a graph $G(V,E)$ and a graph $H(V',E')$, H is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$* [13].

Clearly, by definition 2.1.12 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 observation helps to define 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 2.1.1 to define another important class of graphs called weighted graphs. It must also be noted that all definitions presented so far apply also to weighted graphs.

**Definition 2.1.13.** *Given a graph G(V,E), a weight function is a function $f : E \mapsto \Re^+$* [13]. *The real numbers assigned to each edge are called weights.*

Note that in Definition 2.1.13, the weights are taken to be positive. According to [13], there could be cases where negative weights would be appropriate, however in this dissertation it is to be assumed that when considering a weight function, the weights are positive.

**Definition 2.1.14.** *A weighted graph is a graph G(V,E) with a weight function f* [13]. *This will be 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 where the vertices are the set of towns in the railway network, and two vertices are connected if there is a direct route using the railway 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 this, the shortest path between two 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 2.1.15.** *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 2.1.12, any weighted graph G is a subgraph of itself, therefore, it's weight can be calculated. This is highlighted in Example 2.1.16 below.
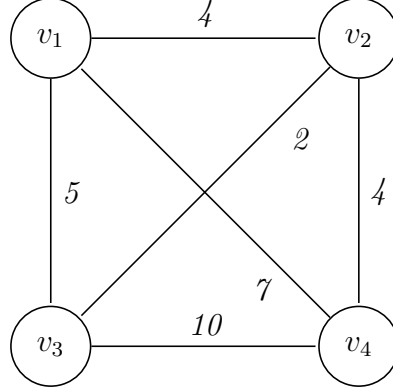
**Example 2.1.16.** Consider the weighted graph *G(V,E,f)* such that, *G(V,E)* is the graph in Example 2.1.11 with the weight function *f* defined below:

$f(\{v_1, v_2\}) = 4$
$f(\{v_1, v_3\}) = 5$
$f(\{v_2, v_3\}) = 2$
$f(\{v_3, v_4\}) = 10$

$f(\{v_2, v_4\}) = 4$

$f(\{v_4, v_1\}) = 7$

Then by Definition 2.1.14, the graph below is a weighted graph.



As highlighted previously, the above graph is a subgraph of itself. Therefore, it's weight can be calculated. As a result, by Definition 2.1.15, the weight of $G$ is 32.

According to Guichard [12], trees are another useful class of graphs.

**Definition 2.1.17.** *A tree is a connected graph with no cycles* [12].

Having defined the basic building blocks in graph theory, 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 2.1.18.** *$H(V',E')$ is a spanning subgraph of $G(V,E)$ if $H$ is a subgraph of $G$ and $V' = V$* [17].

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

**Definition 2.1.19.** *A graph $H$ is a spanning tree of $G$ if $H$ is a tree, and $H$ is a spanning subgraph of $G$* [17].

**Definition 2.1.20.** *Given a graph $G$, $C$ is a Hamiltonian cycle of $G$ 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.* [17]

It is worth mentioning that Definition 2.1.20 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 2.1.18, 2.1.19 and 2.1.20.

**Example 2.1.21.** Let $G$ be the graph in Example 2.1.16. Then, according to Definition 2.1.18, the two graphs below are two spanning subgraphs of $G$ because, they contain all the vertices of $G$ and are subgraphs of $G$ .
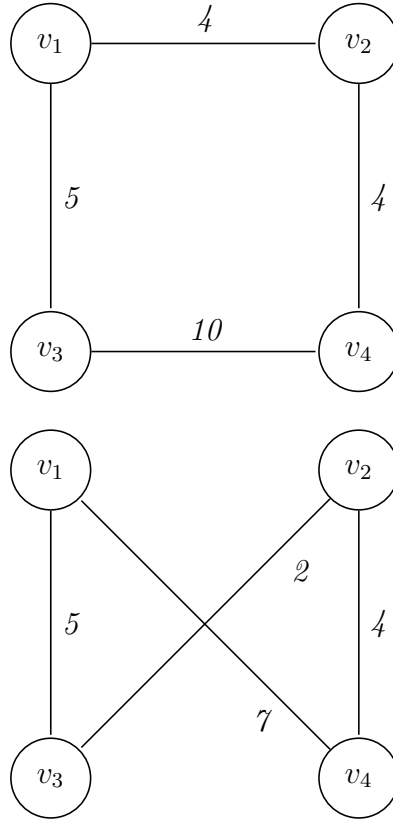
It must also be said that by Definition 2.1.20, the two graphs above are Hamiltonian cycles of $G$ because, they are spanning sub-graphs of $G$, and are Cycle sub-graphs of $G$. Since the above graphs are sub-graphs of $G$, by Definition 2.1.15, their weight can be calculated by summing up the weights of the edges. Thus, the Hamiltonian cycles above have weight 23 and 18 respectively.

Given the graph $G$ in Example 2.1.16, the two graphs below are spanning trees of $G$ of weight 18 and 19 respectively.
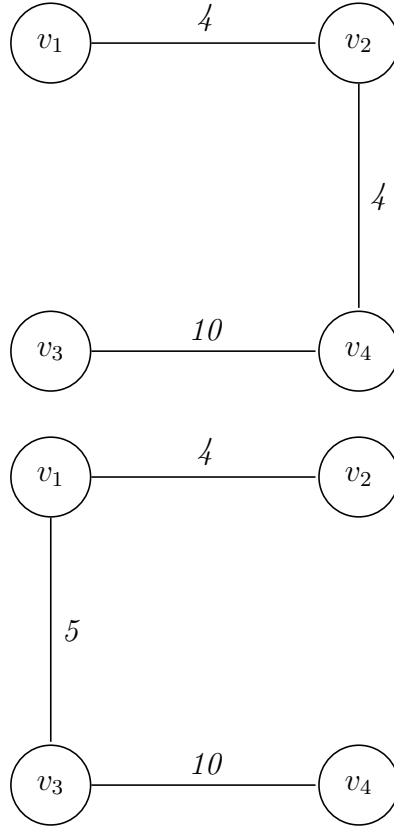
This example also shows that within the same weighted graph, there could be multiple Hamiltonian cycles and spanning trees of different weight.

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 2.1.22 gives a necessary and sufficient condition for a graph to have a spanning tree. Note that the proof of Theorem 2.1.22 was constructed using ideas from [17].

**Theorem 2.1.22.** *A graph $G$ has a spanning tree $\iff$ it is connected* [17].

*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

11

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 $G$. $\qquad\square$

Theorem 2.1.22 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 [12]. 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 [12], 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 [12] that these conditions are not necessary because, there are Hamiltonian graphs that do not have many edges. For example, $C_n$ has only $n$ edges but is Hamiltonian. A sufficient but not necessary condition for Hamiltonianicity is Ore's Theorem below. Note that the proof of Theorem 2.1.23 was constructed using ideas from [17].

**Theorem 2.1.23** (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.* [17]

*Proof.* Suppose that $G(V, E)$ is a graph satisfying all the conditions in the theorem statement, but is not Hamiltonian. 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-adjacent vertices in H. Since by construction $G$ is a subgraph of $H$, $u$ and $v$ must be non-adjacent 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 $v_1$ and $v_i$ in $H$,

then there must not be an edge between $v_{i-1}$ and $v_n$ because, $v_1, v_i, v_{i+1}, ..., v_n, v_{i-1}, v_{i-2}, ..., v_1$ would be a Hamiltonian cycle in $H$, thus H would be Hamiltonian. Therefore, $deg(v_n) \leq n - 1 - \alpha$

$\implies deg(v_1) + deg(v_n) \leq \alpha + n - 1 - \alpha$ in $H$

$\implies deg(v_1) + deg(v_n) \leq n - 1$ in $H$

$\implies deg(v_1) + deg(v_n) < n$ in $G$ since $G$ is a subgraph of $H$.

This contradicts the assumption that $deg(v_1 = u) + deg(v_n = v) \geq n$ in $G$

Therefore, $G$ must be Hamiltonian. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\square$

It is important to note that the above proof uses the fact that $K_n$ is Hamiltonian. This is true because, any cycle $C$ consisting of vertices from $K_n$ only is always a subgraph of $K_n$. Therefore, any cycle $C$ that spans $K_n$ is a spanning subgraph of $K_n$. Hence, by Definition 2.1.20, $C$ must be a Hamiltonian cycle of $K_n$, and therefore, $K_n$ is Hamiltonian. Example 2.1.24 below is a counter example which shows why Ore's theorem gives a sufficient but not necessary condition.

**Example 2.1.24.** Consider the graph $C_5$ *below.*
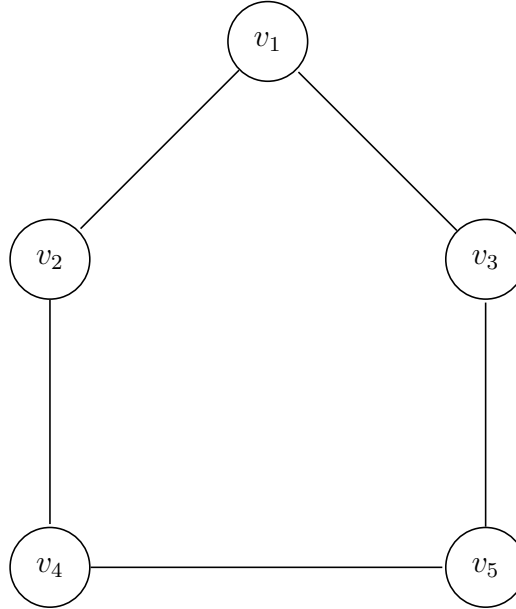


$C_5$ above is Hamiltonian because it contains the Hamiltonian cycle $v_1, v_2, v_4, v_5, v_3, v_1$. However, $deg(v_1) + deg(v_5) = 4 < 5 = n$. Therefore, the condition in Ore's Theorem is not a necessary condition.

The discussion in this section indicates that determining whether a graph is Hamiltonian is a very difficult problem. This means that there are certain problems that are harder than other problems. To formally reason about such problems, some computational theory must first be presented. This is done in the next section.

## 2.2  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 the upcoming chapters. The discussion will now proceed by defining two different types of problems, and describing the relationship between them.

**Definition 2.2.1.** *Suppose that a problem P has many possible solutions such that each solution has an associated value. Then, P is an optimization problem if P requires to find the solution with the best (minimum/maximum) value.* [7]

Therefore, the aim of the algorithm solving an optimization problem is to find the solution with the best (minimum/maximum) value. A maximization problem is an optimization problem that requires to find the solution with the maximum value. On the other hand, a minimization problem is an optimization problem that requires to find the solution with the minimum value. A solution with the best value is called an optimal solution to the optimization problem being solved, and it's value is known as the optimal value. Note that an optimal solution is not termed as 'the' optimal solution because, there could be many solutions having the same optimal value. Another type of problems area called decision problems. [7]

**Definition 2.2.2.** *A Decision problems is a problem whose solution is either a yes or a no* [7].

In [7] it is argued that given an optimization problem, one can transform the given optimization problem into a decision problem, where the decision problem is easier to solve than the optimization problem. Therefore, if some optimization 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 rigorously

defined later in this section. This concept was only mentioned here to show that there exists a relationship between optimization problems and decision problems. Before going into harder computational theory, the example below is given to show how an optimization problem can be transformed into a related decision problem.

**Example 2.2.3.** Two optimization problems are:

1. The Shortest Path Problem

2. The Minimum Weight Spanning Tree problem

Given a connected weighted graph $G$ and two vertices $u$ and $v$ of $G$, the Shortest Path Problem is the task of finding a path $P$ between $u$ and $v$ in $G$ such that, the weight of $P$ is the minimum amongst all paths joining $u$ and $v$ in $G$ [4]. 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 [13]. The Shortest Path Problem is an optimization problem because, from all the possible paths joining $u$ and $v$ in the graph $G$, a path having the optimal value (minimum weight) joining $u$ and $v$ is chosen. In this case, the optimal solution 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 the optimization problems above 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 are the following:

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 two problems above 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. A common property which is analyzed is the running time of an algorithm. This is defined in Definition 2.2.4 below.

**Definition 2.2.4.** *The running time of an algorithm A given an input I is the number of steps executed by A when given I. The running time of an algorithm A can be represented by*

*the function $T : \mathbb{N} \mapsto \mathbb{N}$ where the domain represents the size of the input, and the co-domain represents the amount of steps executed by the algorithm, given an input size.* [7], [1]

What follows is an example which demonstrates how the running time of an algorithm can be computed.

**Example 2.2.5.** 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. 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], [1]

Expressing the running time in this way seems pretty easy to understand. However, this representation has a number of 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 may be 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 will be used because it 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 of an algorithm is polynomial in the input size, then it is known that the problem is in the class P (These terms will be defined later). [7], [1]

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 2.2.5 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 true 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], [1]

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

**Example 2.2.6.** Suppose that the $n$-bit addition algorithm in Example 2.2.5 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 $n$ asymptotic behaviour. Note that this is common to both running time functions. [7], [1]

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 2.2.7.** *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 $f(n) \leq cg(n)$ $\forall$ $n \geq n_0$.* [1]

Definition 2.2.7 implicitly states that when writing $f(n) = O(g(n))$, $g(n)$ is an upper-bound to $f(n)$ as the input size $n$ grows without bound [1]. 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 2.2.7 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 dissertation Big O Notation will be used for the worst-case asymptotic analysis of algorithms. The following is a continuation of Example 2.2.5.

**Example 2.2.8.** By Example 2.2.5, the algorithm that performs two $n$-bit addition has a running time function *T(n)= an*. Therefore, by Definition 2.2.7, *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 $T(n) = O(g(n))$, and $h(n)$ has a higher order of growth than $g(n)$, then, $T(n) = O(h(n))$. For example, in Example 2.2.8 $T(n) = O(n^2)$ as well since $n^2$ is an upperbound to $n$, and hence, $n^2$ 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 $g(n) = O(f(n))$.

Given the above, the time complexity of an algorithm can now be defined formally.

**Definition 2.2.9.** *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)).* [1]

Definition 2.2.9 does not apply only to the worst case running time function. However, recall that when discussing running time functions it was highlighted that it can be assumed that, the worst-case running time function is always 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. For clarity, 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 that 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 [15] and [5].

Table 2.2.1: 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 2.2.1, the time-complexities are ordered in ascending order of growth using the curves in [20]. 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 of growth of an algorithm is, the more steps the algorithm needs to execute for any input, and therefore, the less efficient the algorithm is. In general, problems that can be solved in polynomial time are tractable problems [7]. On the other hand, problems that require super-polynomial (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 2.2.2 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 2.2.2 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 2.2.2: Execution Time Given Input Size and Time Complexity

| Input Size<br>Time Complexity | 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 2.2.2 above was adapted from [15]. In Table 2.2.2 above it is assumed that the machine the algorithms can be executed on perform $10^9$ computational steps per second, as in [15]. It can be concluded from Table 2.2.2 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 if the input size is large. This true because, the rate of growth of the time complexity function is very large. What follows is an example which wraps up all the time complexity concepts discussed so far.

**Example 2.2.10.** In Example 2.2.8, $T(n) = O(n)$. Therefore, according to Definition 2.2.9, the $n$-bit addition algorithm has a time complexity of $O(n)$. Therefore by Table 2.2.1, the $n$-bit 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, and hence in polynomial 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. In fact, one of these classes is the class P defined below. [7]

**Definition 2.2.11.** *The class P consists of all the problems that can can be solved in polynomial time* [7].

Therefore, according to Definition 2.2.11, 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 2.2.5. The $n$-bit addition problem is in the class P because it can be solved in linear time (as Discussed in Example 2.2.10). Apart from the class P, there is also the class NP. The following two definitions help in defining the class NP. Note that Definition 2.2.12 was adapted from information in [14].

**Definition 2.2.12.** *A certificate of a solution is a mathematical object that guarantees the solution.*

It can be deduced from Definition 2.2.12 that if a certificate is correct, then the solution is correct. Therefore, one can verify if a given solution is correct by actually verifying that the certificate is correct. For convenience, in this dissertation, 'verifying a solution/certificate if it is correct' will be referred to as 'verifying a solution'. Note that Definition 2.2.12 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 2.2.13.** *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 2.2.13 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, the solution is implicitly verified. What follows now is an example that explains Definition 2.2.12 and Definition 2.2.13.

**Example 2.2.14.** To explain Definition 2.2.12 and Definition 2.2.13, 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 distinct vertices $S = v_1, ..., v_n$ on $|V| = n$ vertices. Note that if the solution to a Hamiltonian cycle problem instance 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 when the answer is 'no' because, since there is no Hamiltonian cycle in $G$, there is no certificate that represent a Hamiltonian cycle in $G$. As a consequence of these observations, for the Hamiltonian cycle problem to be verifiable in polynomial time, there must be 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 2.2.14, 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 2.2.15.** *The class NP consists of all decision problems that can be verified in polynomial time* [7], [9].

It is important to note that according to Definition 2.2.15, 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 2.2.3). Reasoning about decision problems is easier because, the related decision problem is not harder than the original optimization problem. As a result, one can prove that an optimization 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, the concept of a reduction algorithm must first be defined.

**Definition 2.2.16.** *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.*

*For convenience, if there exists a reduction algorithm $R$ which transforms any instance $\alpha$ of $A$ into an instance $\beta$ of $B$, then it will be said that 'R transforms $A$ to $B$' or 'A is reducible to B'.* [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 $B$ where 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 $A$. After defining reduction algorithms, the class of NP-Complete problems can now be defined. [7]

**Definition 2.2.17.** *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 2.2.17 states that for a problem $A$ to be in the class 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 difficult because, this requires checking that every problem in NP can be reduced to $A$. However, this is not the case because by Exercise 34.3-2 in [7], reduction algorithms are transitive. Hence, considering a known NP-Complete problem $H$, since $H$ is NP-Complete, every problem in NP can be reduced to $H$. Therefore, if a reduction algorithm from $H$ to $A$ can be found, then by transitivity of reduction algorithms, every problem in NP can be reduced to $A$ using a reduction algorithm [7].

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 (because it is guaranteed that the algorithm gives a correct solution in polynomial time). Note that to be precise, the problems in P must first be converted to a related decision problem if they are to be in NP. But since this can be done, and the related decision problem is no harder than the original problem, the result holds. Therefore, every problem in P is

in NP and hence, P $\subseteq$ NP. Showing that NP $\subseteq$ P may seem to be easy as well. In fact, to show that NP $\subseteq$ P, it must be shown that $\exists$ an NP-Complete problem $C$ that can be solved in polynomial time. This would be enough because, by the NP-Completeness 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. Unfortunately, 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]

It is important to note that as indicated in [9], the classes P, NP and NPC are usually defined more formally using models of computations such as Turing Machines. This was not done in this way because Turing Machines are out of the scope of this dissertation. For a more formal definition of these classes, the reader can find more information in [10].

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

# 3    The Travelling Salesman Problem

As discussed in Chapter 1, this chapter is divided into two main sections. In the first section, the Travelling Salesman Problem will be mathematically defined, and some properties about it will be discussed. This will lead to showing that the Travelling Salesman Problem is in NPC. As a result, by the discussion in Section 2.2, no polynomial time algorithm which solves the Travelling Salesman Problem is known. According to [7], this does not mean that the Travelling Salesman Problem cannot be approximated in polynomial time. Hence in the second section, two algorithms that can approximate the Travelling Salesman Problem in polynomial time will be presented, along with some theoretical results about them.

## 3.1    Problem Definition, Properties and Complexity Class

The Travelling Salesman Problem can be defined mathematically as follows.

**Definition 3.1.1.** *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 3.1.1, 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 3.1.1 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 2.1.10, cycles are defined on graphs

having at least 3 vertices. The following example is used to clearly explain what the TSP defined in Definition 3.1.1 is all about.

**Example 3.1.2.** Consider the complete graph $K_4$ with the weight function $f$ defined in Example 2.1.16. The distinct Hamiltonian cycles 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 3.1.1, the solution of the TSP given the graph in Example 2.1.16 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 3.1.3 for any $K_n$.

Example 3.1.2 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 solution [11]. 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 [11].

**Lemma 3.1.3.** *For $n \geq 3$, $K_n$ has $\frac{(n-1)!}{2}$ distinct Hamiltonian cycles* [11].

*Proof.* Suppose that $V$ and $E$ are the set of vertices 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, although 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 $K_n$. $\qquad\qquad\square$

As discussed previously, Lemma 3.1.3 confirms that a brute-force algorithm is not efficient enough to solve the TSP for large inputs. This means that either a new algorithm that solves the TSP in polynomial time must be designed, or it can be shown that the TSP is in NPC. As discussed in Section 2.2, 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 it will be shown that TSP is as hard as any problem in NP. It is important to note that to prove that TSP is in NP, the TSP defined in Definition 3.1.1 must be converted into a related decision problem. This is needed because in Definition 3.1.1, the TSP was presented as an optimization problem, and because, by Definition 2.2.15, the class NP contains decision problems only. The TSP's related decision problem is the following :

**Definition 3.1.4** (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 3.1.1 into a related decision problem, it is now time to show that TSP is in NP. This is shown in Lemma 3.1.5 below. Note that the proof of Lemma 3.1.5 was constructed using ideas in [7].

**Theorem 3.1.5.** *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 2.2, 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. $\quad\square$

To show that the 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 2.2, 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 it has already been showed in Example 2.2.14 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 too long. 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 3.1.6.** *The TSP is an NP-Complete problem* [7].

*Proof.* Consider the TSP as in Definition 3.1.4. By Theorem 3.1.5, the 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. It will now be shown that the Hamiltonian cycle problem can be reduced to the TSP. Let $G(V, E)$ be an arbitrary instance of the Hamiltonian cycle problem such that $|V| = n$ . An instance of the TSP can be constructed from $G$ as follows. From $G$ construct 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 of the TSP can be the pair $(G', 0)$, where $G'$ is the constructed complete weighted graph, and $k = 0$ ($k$ as defined in Definition 3.1.4). From Definition 2.2.16, one can show that the above transformation is a reduction algorithm if the transformation can be carried in polynomial time, and if the solution to the Hamiltonian cycle problem given $G$ is yes $\iff$ the solution to the TSP given $(G', 0)$ is yes. According to [17], $K_n$ 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, it must be shown 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, it has been 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 the transitivity of reduction algorithms, the TSP is as hard as any problem in NP. Therefore, the TSP must be NP-Complete. [7] □

Theorem 3.1.6 confirms that no polynomial time algorithm solving the TSP is known. However, according to [7], there are three ways to get around NP-Completeness. Firstly, if the instance of 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 (solutions who's value is very close to the optimal value) in polynomial time [7]. Algorithms that return near-optimal solutions to a problem are called approximation algorithms [7]. In this dissertation, the third case will be considered. In fact, in the next section, two approximation algorithms that can return near-optimal solutions to the TSP in polynomial time will be presented.

## 3.2   Approximation Algorithms

As already discussed, the main aim of this section is to present two approximation algorithms that can return near-optimal solutions to the Travelling Salesman problem in polynomial time, and to show that for specific instances of the TSP, the value of the near-optimal solutions is bounded from the above. However, before doing so, some background theory on approximation algorithms must be presented.

**Definition 3.2.1.** *Suppose that A is an optimization problem with positive 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($\frac{C}{C^\star}$, $\frac{C^\star}{C}$) ≤ p(n). If an algorithm X has an approximation ratio p(n), then X is said to be a p(n)-approximation algorithm.* [7]

In other words, Definition 3.2.1 states that if an approximation algorithm has an approximation ratio $p(n)$, then for any instance of size $n$, the approximation algorithm will return a solution whose value is at worst $p(n)\times$optimal value. In Definition 3.2.1, if an optimization problem is a maximization 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. This is true because, $\frac{C^\star}{C} = 1$ 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. Note that the proof of Theorem 3.2.2 was constructed using ideas from [7].

**Theorem 3.2.2.** *Suppose that P ≠ NP, then, for any constant p ≥ 1, no polynomial time approximation algorithm with approximation ratio p exists for the TSP* [7].

*Proof.* Consider the TSP as defined in Definition 3.1.1. Suppose that P ≠ 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 2.2.14. Construct an instance $G'(V', E', f)$ of the TSP from $G$ such that, $V' = V$, $E' = \{\{u, v\} : u, v \in V$ 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 3.1.6, this construction can be done in polynomial time. Now, consider the TSP 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 2.1.15, $h$ 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 $G'$. It will now be shown that $A$ can be used to check if $G$ is Hamiltonian. Therefore, one gets 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 3.1.5, 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 2.2, 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.

∴ There is no polynomial time approximation algorithm with a constant approximation ratio for the TSP. □

As discussed in Section 2.2, the P=NP problem is an open problem. Therefore, a polynomial time approximation algorithm with a constant approximation ratio for the TSP is not known, otherwise, the contrappositve of Theorem 3.2.2 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 subsections is called the Metric-TSP defined below.

**Definition 3.2.3.** *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.* [22]

Having presented some background theory on approximation algorithms, it is now time to present 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. The approximation ratios of both these algorithms are not known when applied to the TSP defined in Definition 3.1.1. That being said, it will be shown that their approximation ratios are known when applied to Metric-TSP instances. The following subsection presents the Nearest Neighbour Algorithm.

### 3.2.1 Nearest Neighbour Approximation Algorithm

The Nearest Neighbour Algorithm (NNA) is a simple algorithm that can find near-optimal solutions to the TSP [2]. 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 [2]. The NNA can be described as shown in Algorithm 3.2.1 below. Note that Algorithm 3.2.1 was constructed using ideas from [2].

---

**Algorithm 3.2.1** : $\text{NNA}(G(V, E, f))$

---

1: Randomly select any vertex $u$ as starting vertex, mark it as visited, and set $current\_vertex = u$.

2: Find an edge of least weight between vertex $current\_vertex$ and an unvisited vertex $v$.

3: Mark $v$ as visited and set $current\_vertex = v$.

4: If all vertices have been visited, move to vertex $u$ and go to line five, otherwise, go to line two.

5: Return the sequence of vertices in the order of how they were visited.

---

From Algorithm 3.2.1 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 (as it will be shown in Example 3.2.4).

What must be guaranteed however 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 this, excluding the starting vertex, each vertex cannot be visited more than once because by line two in Algorithm 3.2.1, the algorithm only chooses edges that lead to unvisited vertices. Therefore, by Definition 2.1.20, 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, approximations cannot be given in feasible time (note that feasible time is

at worst polynomial time). According to [19], 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 which shows how the NNA constructs an approximate solution for the TSP.

**Example 3.2.4.** Consider the graph $G$ in Example 2.1.16. The NNA constructs a solution to the TSP defined on $G$ in the following way. Assume that randomly the NNA chooses 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 Algorithm 3.2.1 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.
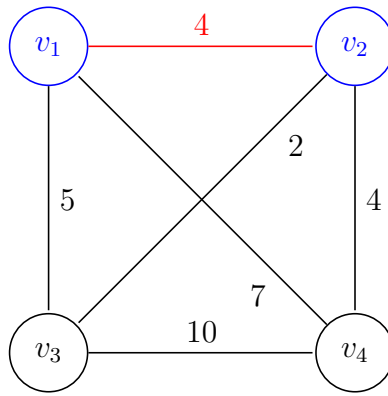
Step 1:
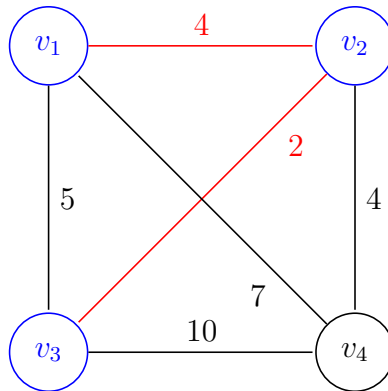
Mark the starting vertex $v_1$ as visited
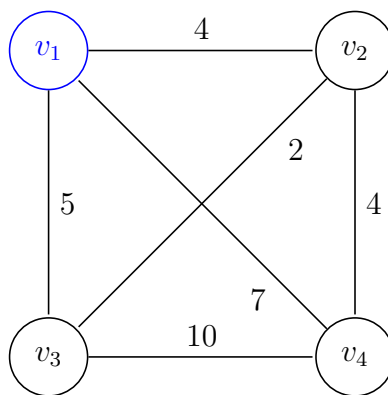


Step 2:

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.

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.
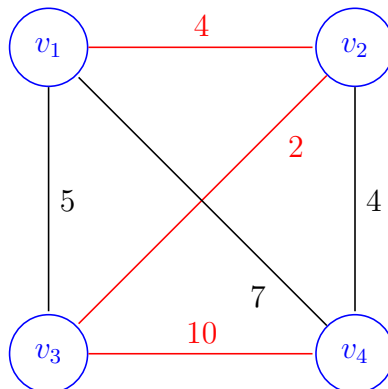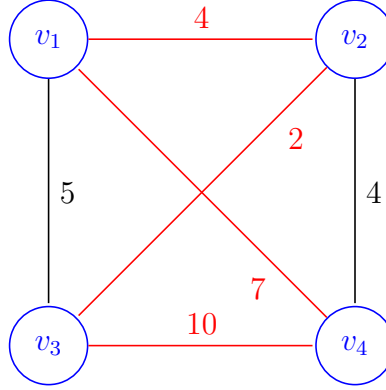


Step 4:

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.

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 3.2.4, the minimum weight Hamiltonian cycle was not found. In fact, according to Example 3.1.2, 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 3.2.5.** Let $M \in \mathbb{Z}^+$. Consider the graph $G$ below:
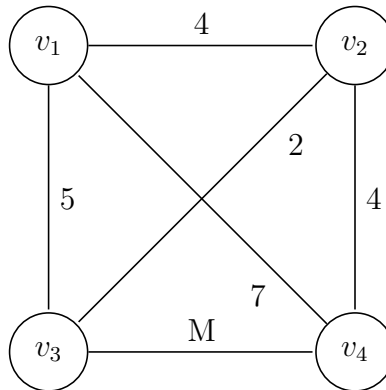


Assuming that $M$ is a very large number, the minimum weight Hamiltonian cycle of $G$ is $v_2, v_3, v_1, v_4, v_2$ of weight 18. This minimum weight Hamiltonian cycle is found if the NNA

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 3.2.5 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 3.2.5, 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 3.2.5, 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. It can be deduced from Theorem 3.2.2 that, it is known that the NNA does not have a constant approximation ratio unless P=NP. In addition to this, By Theorem 1 in [6], unless P=NP, the approximation ratio of the NNA is not known in general. However, Rosenkrantz, Stearns and Lewis [19], 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 $G$. Note that Lemma 3.2.6 was constructed using ideas from [19].

**Lemma 3.2.6.** *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 $\sum_{u\in V} l_u \leq \frac{1}{2}(\lceil log_2 n \rceil + 1)W^\star$. [19]*

*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 2.1.15, $T^\star = \sum_{\{i,j\} \in E''} f(\{i, j\})$. Therefore, by the first bullet point in Lemma 3.2.6, $T^\star = \sum_{\{i,j\} \in E''} f(\{i, j\}) \geq \sum_{\{i,j\} \in E''} min(l_i, l_j)$. 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, $min(2k, n) = 2$k. 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 Claim 2 can be proved for $k = \frac{1}{2} min(2k, n)$, then Claim 2 can be proved 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 $\alpha_1 + ..$

40

$+ \alpha_{min(2k,n)} = |E''|$, 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 $k = 2^j$,

$\sum_{j=0}^{\lceil log_2 n \rceil - 1} W^\star \geq \sum_{j=0}^{\lceil log_2 n \rceil - 1} 2 \sum_{i=2^j+1}^{min(2^{j+1},n)} l_i$.

$\implies \lceil log_2 n \rceil W^\star \geq 2 \sum_{i=2}^{n} l_i$. Now, bullet point number two in Lemma 3.2.6 implies that, $W^\star \geq 2l_1$.

$\implies \lceil log_2 n \rceil W^\star \geq 2 \sum_{i=1}^{n} l_i$.

$\implies \frac{1}{2} (\lceil log_2 n \rceil + 1) W^\star \geq \sum_{i=1}^{n} l_i = \sum_{u \in V} l_u$. $\qquad \square$

Having proved Lemma 3.2.6, 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 $G$. Note that the proof of Theorem 3.2.7 was constructed using ideas from [19].

**Theorem 3.2.7.** *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, $\frac{W}{W^\star} \leq \frac{1}{2}(\lceil log_2 n \rceil + 1)$.* [19]

*Proof.* Define $G(V, E, f)$, $W^\star$, and $W$ as stated in Theorem 3.2.7, and apply Algorithm 3.2.1 to $G$. Let $g : V \mapsto \Re^+$ be the 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 $l_v = g(v)$. It will be shown that $g$ satisfies the two conditions of Lemma 3.2.6.

**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

41

*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 $p_2$ respectively. 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, $2l_u = 2f(\{u, v\}) \leq$ 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 3.2.6, $W = \sum_{u \in V} l_u \leq \frac{1}{2} (\lceil log_2 n \rceil + 1) W^\star$.

$\implies W \leq \frac{1}{2} (\lceil log_2 n \rceil + 1) W^\star$

$\implies \frac{W}{W^\star} \leq \frac{1}{2} (\lceil log_2 n \rceil + 1)$. $\qquad \square$

The implication of Theorem 3.2.7 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, the quality of the approximation cannot worsen

more than some fixed bound. Unfortunately, Rosenkrantz, Stearns and Lewis [19], 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 3.2.8 below without proof because, the proof is too long.

**Theorem 3.2.8.** *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 $G$.* [19]

Theorem 3.2.7 and 3.2.8 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 constant 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.

### 3.2.2 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 to the TSP [7]. The TAMSA computes an approximate solution to 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 connected 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 $V'$. [13]

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 connected weighted graph $G$ together with an arbitrary starting vertex $r$ as input, Prim's algorithm always returns a minimum weight spanning tree of $G$. Theorem 3.2.9 was constructed using ideas from [21].

**Theorem 3.2.9.** *Prim's algorithm always returns a spanning tree* [21].

*Proof.* Let $G(V, E, f)$ be a connected 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 subgraph of $G$ with no cycles. Note that $T$ is a subgraph 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 subgraph 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

44

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, $s = u_1, u_2, ..., u_k, u_{k+1} = u$ 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, one can start from vertex $u/v$, travel to $s$ using $P/P'$, and travel to $v/u$ using $P'/P$. Therefore, $u$ and $v$ must be connected by a path. Hence, $T$ must be a connected spanning subgraph 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 2.1.19, $T$ is a spanning tree of $G$. $\square$

Theorem 3.2.9 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 3.2.10 was constructed using ideas from [21].

**Theorem 3.2.10.** *Prim's algorithm always returns a minimum weight spanning tree* [21].

*Proof.* Let $G(V, E, f)$ be a connected 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 inputs. 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 $e$, one gets 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. $\qquad \square$

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

**Example 3.2.11.** Consider the graph $G$ in Example 2.1.16, and the brief description of Prim's algorithm given in this subsection. 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.
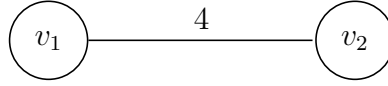
Since not all vertices in $V$ have been added to $V'$, the same procedure is repeated. Hence, the edge $\{v_2, v_3\}$ is chosen because, it is the edge of least weight connecting a vertex in $V'$ to a vertex not in $V'$. Therefore, $T$ is the following graph
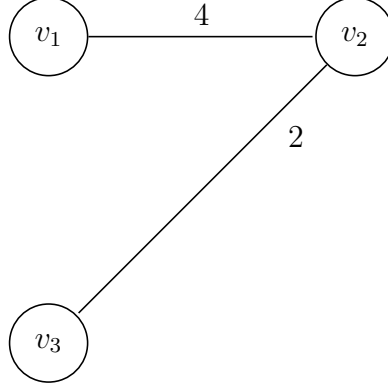


Again, since not all vertices in $V$ have been added to $V'$, the same procedure is repeated. In this case, $\{v_2, v_4\}$ is the edge of least weight connecting a vertex in $V'$ to a vertex not in $V'$. Therefore, $T$ is the following graph.
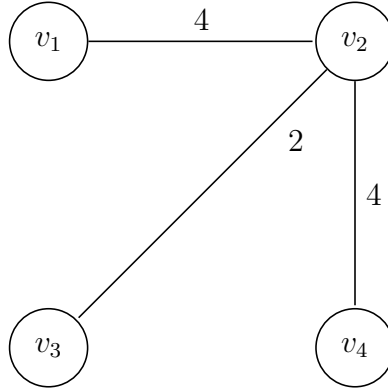


Since all vertices in $V$ have been added to $V'$, the algorithm now terminates by returning $T$.

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 dissertation, 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 3.2.1 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]
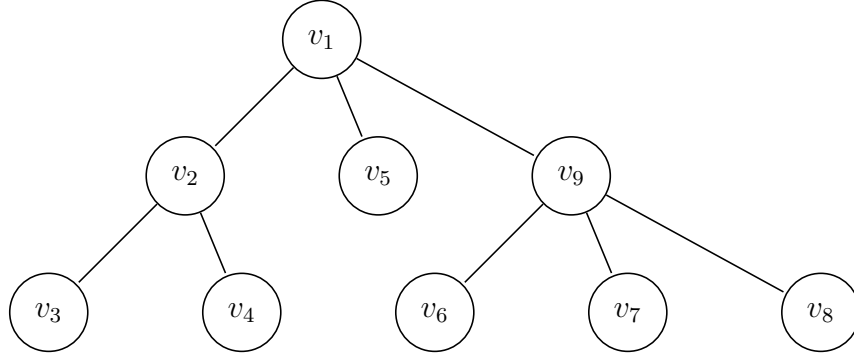


Figure 3.2.1: Tree example

Having briefly described important algorithms that will be used by the TAMSA algorithm, it is now 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 pseudocode below, which was adapted from [7].

---
**Algorithm 3.2.2** : TAMSA($G(V, E, f)$)

---
1: Choose an arbitrary vertex $r \in$ V as starting vertex.

2: Starting from $r$, compute a minimum weight spanning tree $T$ of $G$ using Prim's algorithm.

3: Perform a preorder walk on T starting from $r$, and let $S$ be the walk returned by the preorder tree walk algorithm.

4: Add $r$ to $S$ as a final vertex and return S.

---

For the TAMSA to be a useful approximation algorithm for 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 3.2.10 and Theorem 3.2.9, 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 in line four of Algorithm 3.2.2 above. Therefore, it can be concluded that $S$ must be a Hamiltonian cycle. The next step is to check that the Hamiltonian cycle is computed 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 Algorithm 3.2.2 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, line number three in Algorithm 3.2.2 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 would have a log-linear time complexity. Therefore, using Table 2.2.1 in Section 2.2, and and the fact that the NNA has a quadratic time complexity (discussed in Section 3.2.1), the TAMSA is more efficient asymptotically because, it has a smaller order of growth. Note that, the analysis of Prim's algorithm's time complexity is not given because, it requires theory about data structures such as Heaps to be presented, and this is out of the scope of this dissertation. What follows is an example which demonstrates how the TAMSA computes a Hamiltonian cycle.

**Example 3.2.12.** Consider the graph $G$ in Example 2.1.16, and Algorithm 3.2.2 given in this subsection. Assume that randomly the TAMSA chooses vertex $v_1$ as starting vertex.

From Example 3.2.11, the minimum spanning tree $T$ returned by Prim's algorithm starting from vertex $v_1$ is the graph below.
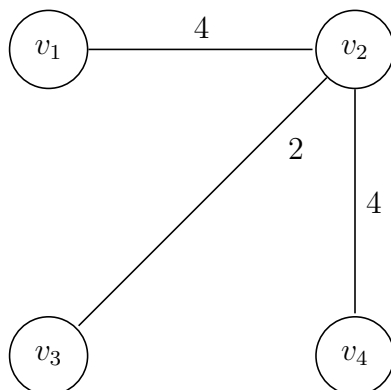


Now, according to bullet line number three in Algorithm 3.2.2, the TAMSA must perform a preorder tree walk on $T$ above. Consider the brief description of the preorder tree walk algorithm. A preorder walk of $T$ is computed in the following way. Note that, in the following demonstration, the blue vertices 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, $S = v_1$.
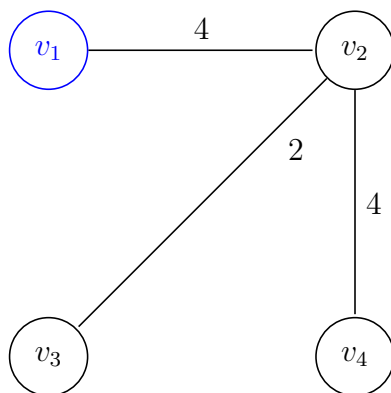


Since not all vertices have been visited, the algorithm recursively calls itself on the unvisited neighbors of $v_1$, which in this case is only $v_2$. Hence, $v_2$ is visited and added to $S$. Therefore $S = v_1, v_2$.
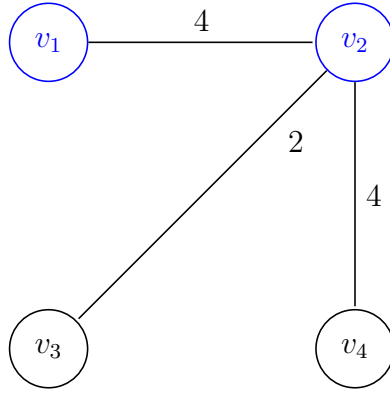
Since not all vertices have been visited, the algorithm recursively calls itself on the unvisited neighbors of $v_2$, which in this case are $v_3$ and $v_4$. Assuming that $v_3$ is visited before $v_4$, $v_3$ is marked as visited and added to $S$. Therefore $S = v_1, v_2, v_3$.
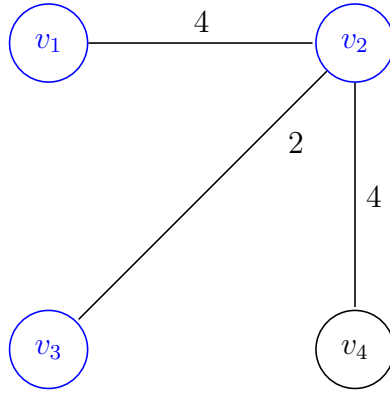


Since not all vertices have been visited, and $v_3$ has no neighbors, the algorithm now visits another neighbor of $v_2$ which in this case is $v_4$. Hence $v_4$ is marked as visited and added to $S$. Therefore $S = v_1, v_2, v_3, v_4$.
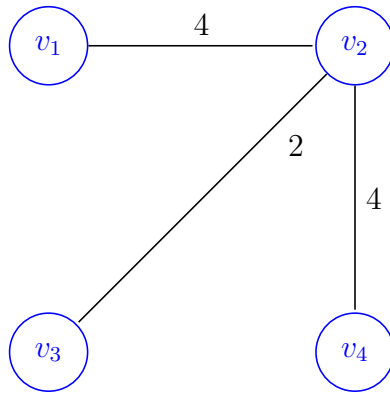


Since all vertices have been visited, the TAMSA adds $v_1$ to $S$ and returns it. Hence the TAMSA returns the Hamiltonian cycle $v_1, v_2, v_3, v_4, v_1$ of weight 23.

From Example 3.2.12 and Example 3.2.4, 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 3.2.5. This is true because of the following. Suppose that $G(V, E, f)$ is the graph in Example 3.2.5, and $v_1$ is the vertex chosen randomly by the TAMSA in line number one of Algorithm 3.2.2 above, 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 2.1.16. 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 edge's 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 3.2.11, 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 value. 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 3.1.1. Similarly to the NNA, due to Theorem 1 in [6], the approximation ratio of the TAMSA when applied to the TSP is unknown unless P=NP. In addition to this, Theorem 3.2.2 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 3.2.13 below. Before proving Theorem 3.2.13, 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 subsection. 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 3.2.1 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 Metric-TSP instances. Note that the proof for Theorem 3.2.13 below was constructed using ideas from [7].

**Theorem 3.2.13.** *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 2.1.19, $T^\star$ 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 to the weight of the minimum weight Hamiltonian cycle. Therefore, letting $T$ be the minimum

weight spanning tree of $G$, the following inequality holds:

$$f(T) \leq f(T^\star) \leq f(H^\star)$$

$$\implies f(T) \leq f(H^\star) \tag{3.2.1}$$

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

$$f(W) = 2f(T) \leq 2f(H^\star) \text{ (By Equation 3.2.1)}$$

$$\implies f(W) \leq 2f(H^\star) \tag{3.2.2}$$

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 $G$, for all $u, c, v \in V$. 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.

$$f(H) \leq f(W) \tag{3.2.3}$$

As described in this subsection, 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 3.2.3 and Equation 3.2.2 it can be concluded that,

$$f(H) \leq f(W) \text{ (By Equation 3.2.3)}$$

$$\implies f(H) \leq 2f(H^\star) \text{ (By Equation 3.2.2)} \tag{3.2.4}$$

Hence, from Equation 3.2.4 it can be concluded that the value of the approximate solution returned by the TAMSA is within a constant factor of 2 from the optimal value. Hence, the TAMSA is a 2-approximation algorithm for the Metric-TSP. $\qquad\square$

Therefore, Theorem 3.2.13 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 3.2.1 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 dissertation because, Christofides' algorithm requires presenting mathematical concepts which are out of the scope of this dissertation. These mathematical concepts are those of a minimum weight perfect matching.

In Section 3.2.2 and Section 3.2.1, the aim was to present algorithms that approximate the TSP in polynomial time, and rigorously prove their approximation ratios when applied to a special case of the TSP (Metric-TSP). Although having approximation guarantees is important, it is sometimes better in practice in terms of accuracy of approximations, to design an algorithm that is guaranteed to find an optimal solution if given enough time to execute. Note that this statement will be justified when comparing the algorithm's performance in Chapter 5.

In the following section, two classes of algorithms called the heuristics and metaheuristics will be defined. Afterwards, an example of a metaheuristic called the Ant Colony Optimization Algorithm is going to be defined. After doing so, an example of an Ant Colony Optimization Algorithm called the Ant Colony System Algorithm will be given. After defining the Ant Colony System algorithm, the main aim is to prove that if given enough time to execute, the Ant Colony System applied to an arbitrary TSP instance $G$ will always find an optimal solution of $G$ (note that the time may be arbitrarily large).

# 4 The Ant Colony Optimization Algorithm

As already discussed, the main aim of this Chapter is to present the Ant Colony Optimization (ACO) Algorithm, and describe how it can be applied to the TSP using the Ant Colony System approach. After doing this, the aim is to argue on why if given enough time to execute, the Ant Colony System algorithm always find an optimal solution of the TSP instance it is applied to. To understand the nature of the ACO algorithm better, 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 near-optimal solutions to a problem in polynomial time, without guaranteeing that an optimal solution will be found [9]. From what has been discussed so far, one can deduce that the TAMSA and the NNA are heuristic algorithms because, they both try to obtain near-optimal solutions to the TSP, without guaranteeing that the returned solution is an optimal one. 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 [9]. 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 from 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 [9]. Note that for convenience, the terms 'metaheuristic algorithm' and 'algorithm' are used interchangeably in this chapter.

The ACO algorithm is a metaheuristic algorithm which was inspired from the behavior of ants in real ant colonies. Ants are capable of finding the shortest path from a food source to their nest and vice-versa without using any visual signals. 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 prefer to move into directions which are rich in pheromone. The trail of pheromone left behind by an ant while walking is called a pheromone trail. It is important to note that, an ant may still move in a direction which is not rich in pheromone, however, the probability of doing so is very small. The movement of ants in directions which are not rich in pheromone is important in real ant colonies because,

it helps ants explore other ways of constructing a path from a food source to the nest which may result into a shorter path. [8], [9]

The reason why ants are able to find short 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 to 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 (the pheromone trail is not yet constructed). In addition to this, pheromone evaporates gradually over time. Hence, the longer a path $P$ is, the more pheromone evaporation $P$ experiences until the ants using $P$ reach their destination. As a result, the ants would probabilistically prefer to move back to the food source using the shorter paths becaus, they simply would contain more pheromone. Over time, this would result into higher accumulation of pheromone on a shortest path, which would make the majority of the ants choose a shortest path in order to travel from their nest to a food source and vice-versa. What follows is an example which demonstrates at an abstract level how the ACO metaheuristic can be applied to the Shortest Path Problem defined in Example 2.2.3. [8], [9]

**Example 4.0.1.** Suppose that the shortest path from vertex $v_1$ to vertex $v_4$ of the graph $G$ below must be found.
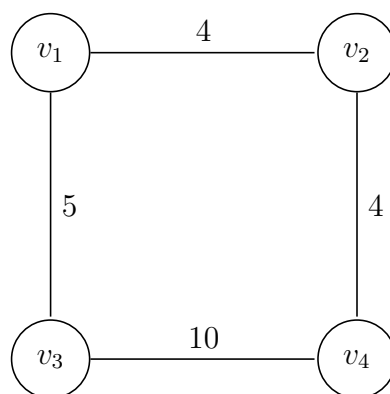


Clearly, the ACO framework can be applied to the Shortest Path Problem on $G$ by letting

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 choose to move to $v_3$. Note that in the graphs below, the red and blue paths represent that different paths taken by the ants.
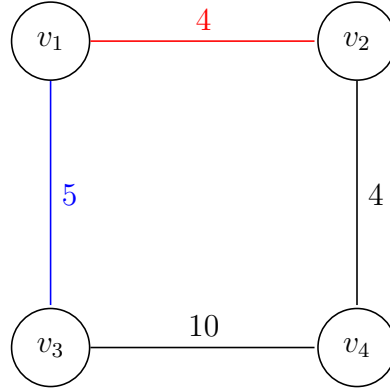


Suppose that a constraint is added such that each ant does not visit vertices that it has already visited. This can be easily done by assigning a list of vertices to each ant $k$ which contains the vertices that ant $k$ has already visited. As a result, the ants at $v_3$ and $v_2$ have only one vertex to move to because $v_1$ has already been visited. Therefore, all ants from $v_2$ and $v_3$ move to vertex $v_4$.
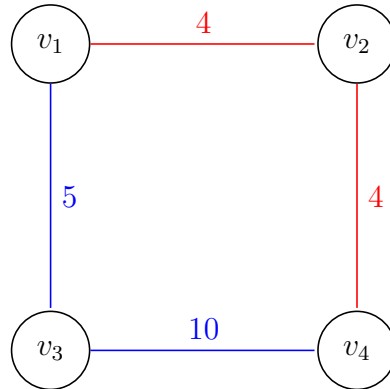


Now, since the red path is shorter than the blue path, the ants following the red path arrive at vertex $v_4$ before the ants following the blue path. Hence, on the way back to vertex $v_1$, the red path has more pheromone than the blue path because, more pheromone is evaporated

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 the instance $(G, v_1, v_4)$, can then return the best found path $P$ (in this case the red path) when either the majority of the ants are traveling through $P$, or, it can return $P$ after a maximum number of iterations are reached.

Having described the basic idea behind the ACO metaheuristic, it is now time to describe the requirements the algorithm which implements the ACO framework must follow, in order to be successfully applied to the TSP. Let $G(V, E, f)$ be an instance of the TSP. As described in Example 4.0.1, 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 4.0.1 above, by giving each ant a list of vertices called the 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 (ants in the algorithm) to compute a Hamiltonian cycle, otherwise, ants may produce walks of $G$ with repetition of vertices. Another important thing is that the artificial ants must keep the order of how the 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. In what follows, a specific type of ACO metaheuristic will be discussed in detail with specific reference to how it can be applied to the TSP. This metaheuristic algorithm is called the Ant Colony System (ACS) algorithm in literature. [8], [9]

The ACS algorithm for the TSP can be described as follows. Suppose that $G(V, E, f)$ is a TSP instance. At the start of the algorithm, each ant is placed on a random vertex of $G$. At each time step, every ant $k$ is moved to a vertex of $G$ which is not in the working memory of ant $k$, and this is repeated until all vertices of $G$ are in the working memory of ant $k$.

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 the TSP is based on finding shorter connections, then the length of the edges must also effect the direction the ants must move to. Therefore, the probabilistic function must be a function of both the pheromone accumulated on the 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 chance $e$ has of 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. [8], [9]

Another important component of the ACS algorithm is local trail updating. 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. Local trail updating is important because, it increases the probability that ants explore other edges that were not visited by any other ant, and hence, lowers the probability that the ants generate the same sub-optimal solution repeatedly. 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. The ACS algorithm encourages this behavior by making the ants that produce the shorter Hamiltonian cycles deposit more pheromone on the edges constituting their Hamiltonian cycle at every iteration. As a result, there would be more pheromone present on edges that constitute the shorter Hamiltonian cycles. This procedure is called global trail updating. One way to go about global trail updating is to make the ant that produces the best Hamiltonian cycle since the first iteration of the algorithm, add more pheromone on the edges constituting it's Hamiltonian cycle. Note that the ant that produces the best Hamiltonian cycle since the first iteration will be referred to as the 'global best ant', and it's Hamiltonian cycle will be referred to as the 'global best Hamiltonian cycle'. Another way to go about global trail updating is to

make the ant that produces the iteration's best Hamiltonian cycle deposit more pheromone on the edges constituting it's Hamiltonian cycle. Note that the ant which produces the iteration's best Hamiltonian cycle will be referred to as the 'iteration's best ant'. According to [9], the former is better than the latter empirically. Hence, in what remains, global trail updating is performed by the global best ant. [8], [9]

It is important to note that there are different ACO algorithms that could have been applied to the TSP. The ACS algorithm was chosen because, it is one of the most empirically successful ACO algorithms in terms of accuracy of approximations [9]. It is important to note that later on in this chapter, other ACO algorithms will be briefly described, by discussing how they vary from the ACS algorithm. The discussion will now proceed by presenting the pseudocode of an algorithm (Algorithm 4.0.1) for the TSP that implements the ACS framework. In Algorithm 4.0.1 below, $G(V, E, f)$ is an instance of the TSP, and $g$ is a probabilistic function with the properties described in the previous paragraphs. Also, $a.current\_vertex$ is the variable containing the vertex that ant $a$ is currently on. Note that Algorithm 4.0.1 was constructed using ideas from [8].

**Algorithm 4.0.1** : ACS_TSP($G(V, E, f)$)

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 = $ Iteration's best ant.

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 4.0.1 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 [8], 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 [8] will be presented. Note that the probabilistic function, and the local/global trail updating formula will be presented in their general form (i.e. for the ACS algorithm applied to any problem). While presenting these formulas, it will be discussed how they can be applied to the TSP. Note that from this point onward, any definitions and terms that will be defined, will be used throughout the entire section.

Let $G(V, E, f)$ be an instance of the TSP. An ant $k$ on vertex $r$ chooses the vertex $s$ to move to among those vertices which do not belong to its working memory $M_k$, by applying

the probabilistic formula below.

$$s = \begin{cases} \underset{u \notin M_k}{\operatorname{argmax}} \left\{ [\tau(\{r,u\})].\,[\eta(\{r,u\})]^\beta \right\} & \text{if } q \leq q_0 \\ \\ S & \text{otherwise} \end{cases} \tag{4.0.1}$$

where $\tau(\{r,u\})$ is the amount of pheromone on the edge $\{r,u\}$, $\eta(\{r,u\})$ is a problem specific function where for the TSP it was chosen to be the inverse of the weight of the edge $\{r,u\}$, $\beta$ is a parameter which weighs the 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:

$$p_k(\{r,s\}) = \begin{cases} \dfrac{[\tau(\{r,s\})].\,[\eta(\{r,s\})]^\beta}{\sum_{u \notin M_k} [\tau(\{r,u\})].\,[\eta(\{r,u\})]^\beta} & \text{if } s \notin M_k \\ \\ 0 & \text{otherwise} \end{cases} \tag{4.0.2}$$

where $p_k(r,s)$ is the probability of ant $k$ on vertex $r$ to choose vertex $s$. [8]

The idea behind Formula 4.0.1 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 $q \leq q_0$, since $\eta$ was chosen to be the inverse of the weight of an edge, 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 4.0.1. In addition to this, the smaller the weight of an edge is, the more chance an edge has to be the maximum element in the argument list of Formula 4.0.1. Therefore, it can be concluded that the case $q \leq q_0$ makes an ant choose the edge with the largest amount of pheromone per edge weight. 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 as required by the ACO metaheuristic. [8], [9]

When $q$ is less than $q_0$, the ant randomly picks a vertex with probability defined by the probability distribution in Formula 4.0.2 above. The idea behind Formula 4.0.2 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 vertex $s$ depends on the amount of pheromone on the edge $\{r, s\}$ per edge weight of $\{r, s\}$, 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 large pheromone and small weight. However, unlike the case $q \leq q_0$ in Equation 4.0.1, 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 to vertices with low probability defined by Formula 4.0.2, may help the algorithm find Hamiltonian cycles of lesser weight. [8], [9]

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 algorithm. According to [8], 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. [8], [9]

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 which belong to shorter Hamiltonian cycles with more pheromone. It was also discussed that this will be achieved by forcing the global best ant to add more pheromone on the edges that belong to it's Hamiltonian cycle. The global trail updating formula can be seen in Formula 4.0.3 below, and it is the one presented in [9]. Therefore, the global best ant performs global trail updating on every edge $\{r, s\}$ forming part of it's solution, using the formula below:

$$\tau(\{r, s\}) = (1 - p).\tau(\{r, s\}) + p.\Delta\tau(\{r, s\}) \tag{4.0.3}$$

where $\tau(\{r, s\})$ is the amount of pheromone on edge $\{r, s\}$, $p$ is the evaporation rate which mimics evaporation of pheromone in real ant colonies, and $\Delta\tau(\{r, s\})$ is a problem specific

value which for the TSP is chosen to be the inverse of the weight of the global best Hamiltonian cycle.

From Formula 4.0.3 it can be deduced that, the shorter the global best Hamiltonian cycle is, the larger $\Delta\tau(\{r,s\})$ is, and hence, the more pheromone is added to the edges of the global best Hamiltonian cycle. As a result, this will accumulate more pheromone on the edges belonging to shorter Hamiltonian cycles, and thus make shorter Hamiltonian cycles more desirable for other ants. As already remarked, although it is desirable to have a lot of pheromone on the edges that form part of short Hamiltonian cycles, the ACS algorithm must not create a situation where an edge is always chosen by all ants. Otherwise, as previously discussed, the ants may not find shorter Hamiltonian cycles. As it can be seen in Formula 4.0.3, this problem is solved by reducing some of the added pheromone by the aid of the evaporation rate $p$. As a result, $p$ is typically a value in the interval (0,1]. In fact, Dorigo and Gambardella [8], found out empirically that the best value of $p$ is 0.1. [8]

For Algorithm 4.0.1 to be implemented, there remains to present the local trail updating formula. The local trail updating formula that will be presented is the one suggested in [9]. Therefore, when an ant on vertex $r$ chooses vertex $s$, it modifies the pheromone on the edge $\{r,s\}$ using the local trail updating formula below.

$$\tau(\{r,s\}) = (1 - \zeta).\tau(\{r,s\}) + \zeta.\tau_0 \qquad (4.0.4)$$

where $\tau_0$ is the 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 $0 < \zeta < 1$ is the evaporation rate. It is important to note that $\tau_0$ and $\zeta$ are usually chosen in such a way that after the local trail updating formula is applied, the pheromone on the chosen edge is reduced. This is important because as discussed before, the aim of local trail updating is to avoid having all ants choosing the same edges. As a result, an experimentally good value for $\zeta$ was found to be 0.1, and an experimentally good value for $\tau_0$ was experimentally found to be $\dfrac{1}{nC^{nn}}$ where $n$ is the number of vertices in the TSP instance, and $C^{nn}$ is the length of the Hamiltonian cycle returned by the NNA. It is important to note that normally, the pheromone values of the edges at the start of the algorithm are initialized to the value of $\tau_0$. Another observation is that although $\zeta$ and $p$ both represent the evaporation rate, their

values can be different. [8], [9]

As already remarked, the ACO algorithm that was applied to the TSP in this section was the ACS algorithm. However, there are other ACO algorithms that could have been applied to the TSP. Two of these ACO algorithms are the Ant System (AS), and the Elitist Ant System (EAS). The main difference between the ACS, AS and EAS is how the pheromones on the edges are updated from one iteration to the next. For example in the AS, no pheromone update is carried out until all ants compute a solution of the problem they are applied to. In addition to this, after all ants compute their solution, every edge in the graph gets it's pheromone reduced. After all edges get their pheromone reduced, all ants add more pheromone on the edges they used. One can observe that this differs a lot from the ACS because, the AS performs pheromone evaporation on all edges in the graph. In addition to this, unlike the ACS, the AS does not perform global trail updating. The other ACO algorithm variant that was mentioned in this paragraph is the EAS. The EAS is an extension of the AS, where the only difference is that at each iteration, the global best ant adds more pheromone on the edges that constitute it's solution. Note that the AS and the EAS where only briefly described to show that there are other ACO algorithms that can be applied to any problem including the TSP. A detailed explanation on these ACO algorithms is given in [9]. [9]

Since ants are capable of finding a shortest path from a food source to their nest, it is now natural to ask whether the ACS algorithm is able to find a minimum weight Hamiltonian cycle if given enough time to execute, when applied to the TSP. More generally, if a problem $P$ can be solved by an ACO algorithm $A$, will $A$ eventually find an optimal solution of $P$? In literature, when an optimization algorithm (algorithm for an optimization problem) $A$ finds an optimal solution for a problem, $A$ is said to converge. It is important to note that, if an ACO algorithm $A$ converges, then $A$ can find the optimal solution of every problem that it can be applied to. Unfortunately, it is not easy to determine whether the ACO metaheuristic converges because, the brief history of the ACO metaheuristic is a history of experimental research [9]. In addition to this, the fact that the ACO metaheuristic can be applied to a wide variety of problems, makes theoretical analysis very difficult [9]. Due to these reasons, all of the convergence results about the ACO metaheuristic that are known of only apply to

specific variants (ex. ACS) [9].

As already hinted in the previous paragraph, the theoretical property of algorithms that will be analyzed in this project is that of convergence. In general, there are at least two types of convergence in the ACO framework. These are convergence in value, and convergence in solution. An optimization algorithm converges in value if it generates an optimal solution at least once. On the other hand, an optimization algorithm converges in solution if it reaches a state which repeatedly generates the same optimal solution. It is important to note that, although convergence in solution is stronger than convergence in value, in this chapter only convergence in value will be considered. The reason is that, in optimization, the algorithm solving the optimization problem will never find a better approximation value once an optimal solution is found. For example, in Algorithm 4.0.1, once the value of best_ham_cycle.weight becomes equal to the optimal value, it will never change again because, the optimal value is the smallest possible value. In what follows, an ACO algorithm variant called $ACO_{bs,\tau_{min}}$ will be defined, and it will be shown that the $ACO_{bs,\tau_{min}}$ algorithm converges in value if given enough time to execute. Afterwards, it will be shown that the proof of convergence of $ACO_{bs,\tau_{min}}$ applies also to the ACS. As a result, it can then be confirmed that, when applied to the TSP, the ACS will eventually converge to the weight of a minimum weight Hamiltonian cycle if given enough time to execute (i.e. when applied to the TSP, if given enough time to execute, the ACS will always find an optimal solution at least once). [9]

In what has been discussed so far, it has always been assumed that the ants perform walks on the instance of the optimization problem that is inputted to the ACO algorithm implemented. For example in the ACS for the TSP, the ants perform walks directly on the TSP graph instance. However, this can only be done if the instance of the problem being consider is a graph instance. Optimization problems who's instances are not a graph require their instances to be transformed into graph instances. Otherwise if not, the ACO algorithm cannot be applied on that problem. Therefore, before defining the $ACO_{bs,\tau_{min}}$ algorithm, a number of concepts must first be formalized. Note that the terms/concepts that will be defined/formalized/generalized from this point onward will be used throughout in what follows. A minimization problem $P$ can be defined as a triple $(S, f, \Omega)$ where, $S$ is the set

of all candidate solutions of $P$, $f : S \mapsto \Re^+$ is called the cost function, and $\Omega$ is a set of constraints which defines the set of feasible candidate solutions $\widetilde{S} \subseteq S$. As a consequence of this definition, $s^\star \in \widetilde{S}$ is an optimal solution of $P$ if $f(s^\star) \leq f(s)$ for all $s \in S$. In general, $P$ can be mapped to a new problem $A$ which can be characterized in terms of a graph as follows. Let $C = \{c_1, c_2, ..., c_N\}$ be a set of components/vertices, where $N$ is the size of $C$, and let $X$ be the finite set of states the algorithm solving the problem $A$ can be in, defined in terms of all possible sequences $(c_i, c_j, ..., c_h, ...)$ over the elements of $C$, such that the length of every state is finite. Denote by $n$ the maximum possible length of an element $x$ in $X$. In addition to this, let $C$ and $S$ be in such a way that $S \subseteq X$. Also, let $\widetilde{X} \subseteq X$ be a set of states of $A$ such that, if an element $x$ is in $\widetilde{X}$, then, $x$ can be transformed into an element $s$ which satisfies $\Omega$ by adding more components to $x$. As a result, $s \in \widetilde{S}$. In addition to this, let $S^\star$ be the set of all optimal solutions of $A$. As a result, $S^\star \subseteq \widetilde{X}$ and $S^\star \subseteq S$. Given the definitions in this paragraph, an ACO algorithm can then be applied to $A$ such that, the ants perform walks on the graph $G_c = (C, L, T)$, where $L$ is the set of edges that fully connects the components in $C$, and T is a matrix of pheromone trails, where $\tau_{ij}=T_{ij}$ denotes the pheromone value of the edge $\{c_i, c_j\}$ in $G_c$. It is important to note that in this case, the graph $G_c$ will not posses the property of changing with respect to time. Hence, the problems that will be considered in this chapter are static with respect to time (static problems for short). Therefore, the convergence proofs that will be presented later on in this chapter only apply to static problems. Note that the TSP is a static problem because, the graph instance does not change during an algorithm's execution. Note that, what is important to take from these formalizations is that, if a minimization problem $P$ is to be solved using an ACO algorithm, then a graph $G_c$ must be constructed for the ants to solve $P$. [9]

To understand better the concepts formalized in the previous paragraph, these concepts are now going to be presented in the context of the TSP. The TSP can be represented formally as an optimization problem as follows. Given a TSP instance $G(V, E, g)$, $S$ can contain tuples of vertices in $G$, where repetition of vertices in the tuples is allowed. As a result, $S$ contains all possible walks in $G$. Since in the TSP, weights of walks must be optimized, the cost function $f$ is chosen to be $g$. The set of feasible candidate solutions $\widetilde{S}$ must contain the solutions which satisfy the problem's constraints. Hence, in the context of the TSP, the

solutions in $\widetilde{S}$ must form a Hamiltonian cycle. As a result, the constraints in $\Omega$ must force $\widetilde{S}$ to contain only Hamiltonian cycles. Therefore, two constraints are only needed in $\Omega$. The first constraint in $\Omega$ is that every $s$ in $\widetilde{S}$ must be of the form $(v_1, v_2, ..., v_n, v_1)$, where apart from $v_1$, no other vertex is allowed to be repeated in $s$. The second constraint in $\Omega$ is that for every $s$ in $\widetilde{S}$, $s$ must contain all the vertices in $V$. The next term that needs to be defined is $S^\star$. In the context of the TSP, $S^\star$ is the set of all minimum weight Hamiltonian cycles in $G$. Now, since the TSP's instance are graphs, the ants can directly use $G$ to perform it's walks. As a result, $G_c = G$, $C = V$ and $E = L$. What remains to be defined is $X$ and $\widetilde{X}$. In the context of the TSP, $X = S$ and $\widetilde{X}$ is the set of all elements in $X$ of the form $x = (v_1, v_2, ..., v_n)$, such that if more vertices are added to $x$, $x$ is transformed into a Hamiltonian cycle.

Given the definitions and terms defined in the previous paragraphs, it is now time to define the $ACO_{bs,\tau_{min}}$ algorithm. The algorithm starts by initializing the pheromones in T to $\tau_0 > 0$, where, $\tau_0$ is the same parameter as in Equation 4.0.4. Secondly, each ant is placed on a vertex of the graph $G_c$, where the vertex is chosen according to problem the $ACO_{bs,\tau_{min}}$ algorithm is applied to (normally the starting vertices are chosen randomly with uniform probability). Thirdly, the ants are moved around $G_c$, and $\Omega$ is used to prevent infeasible solutions from being constructed. Afterwards, step 2 and step 3 are repeated for a number of iterations, where the terminating condition depends on the problem the $ACO_{bs,\tau_{min}}$ algorithm is applied to. Similarly to the ACS, there remains to describe the probabilistic function to be used by ants when choosing new components/vertices (vertices and components are used interchangeably) of $G_c$, and how the pheromones are updated at each iteration. Suppose that ant $k$ is currently on component $c_i$. Having constructed the solution $x_h = (v_1, v_2, ..., v_h = c_i)$, ant $k$ chooses the component $c_j$, according to the following probability distribution.

$$P(v_{h+1} = c_j | x_h) \begin{cases} \dfrac{\tau_{ij}^{\alpha}}{\sum_{\{c_i, c_l\} \in N_i^k} \tau_{il}^{\alpha}} & \text{if } \{c_i, c_j\} \in N_i^k \\ 0 & \text{otherwise} \end{cases} \qquad (4.0.5)$$

where, $\{c_i, c_j\} \in N_i^k$ iff the solution $(v_1, v_2, ..., v_h = c_i, c_j)$ built by ant $k$ is an element of $\widetilde{X}$ (i.e. the solution built by ant $k$ forms part of a feasible solution), and $\alpha$ is a parameter in the interval $(0, \infty)$. As it can be seen from the probability distribution in Equation 4.0.5, the

ants in the $ACO_{bs,\tau_{min}}$ algorithm probabilistically prefer to move to feasible components of $G_c$ (the solution is still in $\widetilde{X}$ when the new component is chosen) connected by edges which are rich in pheromone, as required by every ACO algorithm. Note that, if an edge $\{c_i, c_j\}$ is not in $N_i^k$, then $c_j$ has no chance of being selected because, it's probability is zero. This had to be done because of the following. Suppose that an ant has constucted the solution $(v_1, v_2, ..., v_h = c_i)$. If $\{c_i, c_j\}$ is not in $N_i^k$, then the solution $(v_1, v_2, ..., v_h = c_i, c_j)$ is not $\widetilde{X}$, hence it invalidates $\Omega$. Therefore, $(v_1, v_2, ..., v_h = c_i, c_j)$ is not a feasible solution, and hence it should not be constructed. [9]

After defining the probabilistic function, there remains to describe how the pheromones in T should be updated from one iteration to the next. Once all the ants have completed a solution in $\widetilde{X}$, the values in T are decreased by a small factor $0 < p \leq 1$, where $p$ is the same parameter as in Equation 4.0.3 (the evaporation rate). In other words, the pheromone levels of all edges $\{c_i, c_j\}$ in $G_c$ are decreased by a small factor $p$ using the formula below:

$$\tau_{ij} = (1 - p)\tau_{ij} \tag{4.0.6}$$

Let $s^{bs}$ be the global best solution of the $ACO_{bs,t_{min}}$ algorithm. After pheromone evaporation is performed, the pheromone value of the edges constructing $s^{bs}$ are increased by a value $q_f(s^{bs})$, where $q_f$ is defined by $q_f : \widetilde{S} \mapsto \Re^+$ such that, if $f(s_1) > f(s_2)$ then $q_f(s_1) \leq q_f(s_2)$ ($f$ is the minimization's problem cost function). In literature, $q_f$ is called a quality function, and the procedure of updating the pheromones of the edges constructing the global best solution is called global-best pheromone update. For example in the ACS, $q_f(s) = \frac{1}{f(s)}$, where clearly, if $f(s_1) > f(s_2)$ then $q_f(s_1) \leq q_f(s_2)$. Therefore, after Equation 4.0.6 is performed for every edge in $G_c$, the following equation is performed for every edge $\{c_i, c_j\}$ in $s^{bs}$:

$$\tau_{ij} = \tau_{ij} + q_f(s^{bs}) \tag{4.0.7}$$

An additional constraint is also put on the pheromones in T. The constraint says that, the smallest entry of T cannot be less than some parameter $\tau_{min} > 0$. As a result, after Equations 4.0.6 and 4.0.7 are performed, the algorithm must check all the entries of T, to verify that every entry in T is greater than $\tau_{min}$. If an entry $e$ in T is less than $\tau_{min}$, the algorithm must overwrite the value at entry $e$ to that of $\tau_{min}$. Therefore, for every edge $\{c_i, c_j\}$ in $G_c$, the

$ACO_{bs,\tau_{min}}$ algorithm updates $\tau_{ij}$ using the following formula:

$$\tau_{ij} = max\{\tau_{ij}, \tau_{min}\} \tag{4.0.8}$$

To summarize, the pheromone update procedure that will be used in the $ACO_{bs,\tau_{min}}$ algorithm can be characterized by the following. Pheromone update is performed only after all ants compute a solution. After all ants compute their solutions, every edge in $G_c$ gets it's pheromone reduced by a factor $p$, which mimics pheromone evaporation in real ant colonies. After pheromone evaporation is carried out, global-best pheromone update is performed to reward solutions which are more feasible than others, with more pheromone. Finally, if any entry $e$ in T has value less than $\tau_{min}$, then $e$ is overwritten to $\tau_{min}$. In what follows it will be assumed that $\tau_{min} < q_f(s^\star)$, where $s^\star$ is the optimal solution of the problem the $ACO_{bs,\tau_{min}}$ algorithm is applied to. Note that this can be achieved by setting $\tau_0 = \frac{q_f(s')}{2}$, where $s'$ is any solution in $\widetilde{S}$ ($\tau_{min} \leq \tau_0 = \frac{q_f(s')}{2} \leq \frac{q_f(s^\star)}{2} < q_f(s^\star)$). [9]

After defining the $ACO_{bs,\tau_{min}}$ algorithm, it is now time to present two lemmas. These two lemmas will be used to show that the $ACO_{bs,\tau_{min}}$ algorithm converges in value to an optimal solution $s^\star$ of the static problem the algorithm is applied to. The first lemma states that the pheromone value of any edge in $G_c$ cannot exceed that of $\tau_{max} = \frac{q_f(s^\star)}{p}$, even if the algorithm is executed for an infinite number of iterations. In what follows, all the concepts/terms defined in this chapter are assumed. In addition to these definitions, let $\theta$ denote an arbitrary iteration of the $ACO_{bs,\tau_{min}}$ algorithm, and $\tau_{ij}(\theta)$ denote the value of $\tau_{ij}$ at iteration $\theta$. Note that the proof of Lemma 4.0.2 was constructed using ideas from [9].

**Lemma 4.0.2.** *For any $i$, $j \in [N]$, $\lim_{\theta \to \infty} \tau_{ij}(\theta) \leq \tau_{max} = \frac{q_f(s^\star)}{p}$* [9].

*Proof.* Since $f(s^\star) < f(s) \ \forall \ s \in \widetilde{S}$ such that $s \neq s^\star$, then, by definition of $q_f$, $q_f(s^\star) \geq q_f(s)$ $\forall \ s \in \widetilde{S}$, such that $s \neq s^\star$ (*).

Now, from Equation 4.0.7 it can be deducated that, $q_f$ determines the amount of pheromone to be added to the edges constituting the global-best solution $s^{bs}$. In addition to this, edges which do not form part of $s^{bs}$ do not get more pheromone. Hence, (*) implies that the maximum amount of pheromone that can be added to any edge $\{c_i, c_j\}$ is $q_f(s^\star)$, and this is true for all iterations of the $ACO_{bs,\tau_{min}}$ algorithm. As a result, by Equations 4.0.6 and 4.0.7,

and the fact that $\tau_{min} < q_f(s^\star)$, the maximum amount of pheromone that can be present on any edge after one iteration is $(1-p)\tau_0 + q_f(s^\star)$, where $\tau_{ij}(0) = \tau_0$ as already described. Similarly, the maximum amount of pheromone that an edge can have at iteration two is $(1-p)^2\tau_0 + (1-p)q_f(s^\star) + q_f(s^\star)$. Hence, in general, any edge in iteration $\theta$ can have a maximum pheromone value of $\tau_{ij}^{max}(\theta) = (1-p)^\theta\tau_0 + \sum_{i=1}^{\theta}(1-p)^{\theta-i}q_f(s^\star)$. Now, since $\tau_{ij}(\theta)$ $\leq \tau_{ij}^{max}(\theta)$ for any $\theta$, then $\lim_{\theta\to\infty}\tau_{ij}(\theta) \leq \lim_{\theta\to\infty}\tau_{ij}^{max}(\theta)$. In addition to this, $\tau_{ij}^{max}(\theta) = (1-p)^\theta\tau_0$ $+ \sum_{i=1}^{\theta}(1-p)^{\theta-i}q_f(s^\star) = (1-p)^\theta\tau_0 + \sum_{i=0}^{\theta-1}(1-p)^iq_f(s^\star)$. Therefore, $\lim_{\theta\to\infty}\tau_{ij}^{max}(\theta) = \lim_{\theta\to\infty}$ $(1-p)^\theta\tau_0 + \lim_{\theta\to\infty}\sum_{i=0}^{\theta-1}(1-p)^iq_f(s^\star)$. Hence, $\lim_{\theta\to\infty}\tau_{ij}^{max}(\theta) = \lim_{\theta\to\infty}\sum_{i=0}^{\infty}(1-p)^iq_f(s^\star) = \frac{q_f(s^\star)}{p}$. Therefore, $\lim_{\theta\to\infty}\tau_{ij}(\theta) \leq \lim_{\theta\to\infty}\tau_{ij}^{max}(\theta) = \frac{q_f(s^\star)}{p}$, and hence, $\lim_{\theta\to\infty}\tau_{ij}(\theta) \leq \frac{q_f(s^\star)}{p}$ as required. $\quad\square$

The next lemma that will be proved states that, once an optimal solution $s^\star$ is found by the $ACO_{bs,\tau_{min}}$ algorithm, and the number of iterations $\theta$ become arbitrarily large, the pheromone values of all edges constituting $s^\star$ converge to $\tau_{max}$ defined in Lemma 4.0.2. In what follows, for all $i$, $j \in [N]$ such that $\{c_i, c_j\}$ constitutes $s^\star$, let $\tau_{ij}^\star(\theta)$ denote the value of $\tau_{ij}$ after $\theta$ iterations. Note that Lemma 4.0.3 was constructed using ideas from [9].

**Lemma 4.0.3.** *Once an optimal solution $s^\star$ is found by the $ACO_{bs,\tau_{min}}$ algorithm, the following holds:*

*For all $\{c_i, c_j\}$ in $G_c$ constituting $s^\star$, $\lim_{\theta\to\infty}\tau_{ij}^\star(\theta) = \tau_{max} = \frac{q_f(s^\star)}{p}$. [9]*

*Proof.* Once an optimal solution $s^\star$ is found by the $ACO_{bs,\tau_{min}}$ algorithm, global-best pheromone update is performed only on $s^\star$, for the entire duration of the $ACO_{bs,\tau_{min}}$ algorithm. This is true because, the global best optimal solution is only changed in the $ACO_{bs,\tau_{min}}$ algorithm if another solution with lesser cost is found, where at the same time, no solution in $\widetilde{S}$ has cost less than $s^\star$. Therefore, let $s^\star$ be the first optimal solution found by the $ACO_{bs,\tau_{min}}$ algorithm, and $\theta^\star$ be the iteration when $s^\star$ is found. Then, for any connection $\{c_i, c_j\}$ constituting $s^\star$, the amount of pheromone on $\{c_i, c_j\}$ after the first iteration that $s^\star$ is found is $\tau_{ij}^\star(\theta^\star + 1) = (1-p)\tau_{ij}^\star(\theta^\star) + q_f(s^\star)$. Similarly, for the second iteration after $s^\star$ is found, the amount of pheromone on every $\{c_i, c_j\}$ constituting $s^\star$ is $\tau_{ij}^\star(\theta^\star + 2) = (1-p)^2\tau_{ij}^\star(\theta^\star) + (1-p)q_f(s^\star) + q_f(s^\star)$. Therefore, for $\sigma$ iterations after $s^\star$ is found, the amount of pheromone on every $\{c_i, c_j\}$ constituting $s^\star$ is $z = \tau_{ij}^\star(\theta^\star + \sigma) = (1-p)^\sigma\tau_{ij}^\star(\theta^\star)$ $+ \sum_{i=1}^{\sigma}(1-p)^{\sigma-i}q_f(s^\star)$. Therefore, since by Lemma 4.0.2 $\tau_{ij}^\star(\theta^\star)$ is bounded from the above, then by the proof of Lemma 4.0.2, as $\sigma \to \infty$, $z \to \frac{q_f(s^\star)}{p}$. Therefore, regardless of what

$\tau_{ij}^\star(\theta^\star)$ is, the pheromones on the edges of $s^\star$ will converge to $\frac{q_f(s^\star)}{p}$ once $s^\star$ is found. Hence, $\lim\limits_{\theta\to\infty} \tau_{ij}^\star(\theta) = \frac{q_f(s^\star)}{p}$ once an optimal solution is found. $\qquad\square$

It is now time to show that, the $ACO_{bs,\tau_{min}}$ algorithm finds the optimal solution $s^\star$ of the static problem it is applied to, at least once if given enough time to execute. In other words, it will be shown that the $ACO_{bs,\tau_{min}}$ algorithm converges in value to an optimal solution of the static problem it is applied to. Again, the proof is constructed using ideas from [9].

**Theorem 4.0.4.** *Let $P^\star(\theta)$ be the probability that the $ACO_{bs,\tau_{min}}$ algorithm finds an optimal solution at least once in the first $\theta$ iterations. Then, for an arbitrary small $\epsilon > 0$ and for an arbitrarily large $\theta$, $P^\star(\theta) \geq 1\text{-}\epsilon$, and by definition of $\epsilon$ $\lim\limits_{\theta\to\infty} P^\star(\theta) = 1$. [9]*

*Proof.* Due to Lemma 4.0.2 and Lemma 4.0.3, the pheromone value of any edge in $G_c$ is in the interval $[\tau_{min}, \tau_{max}]$. Hence, given a partial feasible solution $x_h \in \widetilde{X}$, any feasible choice performed by Formula 4.0.5 when selecting any component has probability $p_{min} > 0$. From Formula 4.0.5 it can be observed that, the smallest value that $p_{min}$ can take occurs when the pheromone on the chosen edge is $\tau_{min}$, and all the other edges have $\tau_{max}$ pheromone on them. In other words, the smallest value of $p_{min}$ can be represented by $\widetilde{p}_{min} = \dfrac{\tau_{min}^\alpha}{(N-1)\tau_{max}^\alpha + \tau_{min}^\alpha}$, where $N$ is the number of vertices in $G_c$. Note that, by definition of $\widetilde{p}_{min}$, $\widetilde{p}_{min} \leq p_{min}$. As a result, any feasible solution $s' \in \widetilde{S}$ can be generated by the $ACO_{bs,\tau_{min}}$ algorithm with probability $\widetilde{p} \geq \widetilde{p}_{min}^n > 0$, where $n$ is the maximum length of $s'$. Note that a solution $s'$ is generated with probability $\widetilde{p}_{min}^n$ if, all components in $s'$ are chosen with probability $\widetilde{p}_{min}$. Since this holds for all $s' \in S$, it holds for any optimal solution $s^\star$ (therefore, in this case $\widetilde{p}$ is the probability of $s^\star$ being generated). Let $s^\star$ be any optimal solution of the problem the $ACO_{bs,\tau_{min}}$ algorithm is applied to. The probability of not generating $s^\star$ in the first $\theta$ iterations is $(1 - \widetilde{p})^\theta$. Therefore, the probability of generating $s^\star$ at least once in the first $\theta$ iterations is $\widetilde{P}^\star(\theta) = 1 - (1 - \widetilde{p})^\theta$. Now, since $P^\star(\theta)$ is the probability of generating any optimal solution at least once in the first $\theta$ iterations, then it must be that $\widetilde{P}^\star(\theta)$ is a lower bound of $P^\star(\theta)$ for all $\theta$. Hence, letting $\epsilon = (1 - \widetilde{p})^\theta$, then $P^\star(\theta) \geq 1\text{-}\epsilon$. As a result, if $\theta \to \infty$ then, $1 = \lim_{\theta\to\infty} \widetilde{P}^\star(\theta) \leq \lim_{\theta\to\infty} P^\star(\theta) \leq 1$. Hence, $\lim_{\theta\to\infty} P^\star(\theta) = 1$. $\qquad\square$

As discussed previously, Theorem 4.0.4 guarantees that the $ACO_{bs,\tau_{min}}$ algorithm will find an optimal solution if it is executed long enough. What is interesting is that the theorem also applies to other variants of ACO algorithms which are different from the $ACO_{bs,\tau_{min}}$

algorithm in the way they update the pheromone on the edges of $G_c$. The main idea that was used to show that the $ACO_{bs,\tau_{min}}$ algorithm converges was that, the probability $P(s)$ of constructing a solution $s \in \widetilde{S}$ is always greater than a small constant $\epsilon' > 0$. As seen in the proof of Theorem 4.0.4, this fact was obtained directly from the fact that $0 < \tau_{min} < \tau_{max} < \infty$. Therefore, if an ACO algorithm $A$ satisfies these two facts, then Theorem 4.0.4 applies to $A$. In what remains, it will be discussed on why the ACS satisfies both of these facts, and hence, why Theorem 4.0.4 applies to the ACS. [9]

The ACS algorithm differs from the $ACO_{bs,\tau_{min}}$ algorithm in three ways. The first difference is in the probabilistic function that the ACS uses to construct solutions (Equation 4.0.1 and Equation 4.0.2). The second difference is that in the ACS, pheromone evaporation is not performed on all of the edges, but to those of the global best solution, and to the chosen edges while the ants construct solutions. The third difference is that in the ACS, while an ant constructs a solution, the ant performs local pheromone update on the edges that it chooses. One can notice that in the ACS, the local pheromone update and the global pheromone update are of the form $\tau_{ij}(\theta + 1) = (1 - \mu)\tau_{ij}(\theta) + \mu b$ where $b = q_f(s^{bs})$ or $\tau_0$ and $\mu = p$ or $\zeta$. Therefore, $\tau_{ij}(\theta) = (1 - \mu)^\theta \tau_{ij}(0) + b(1 - (1 - \mu)^\theta)$. Hence, as $\theta \to \infty$, $\tau_{ij}(\theta) \to b$ (*). Hence, the sequence of pheromones $\tau_{ij}(\theta)$ can either decrease to $b$ if $\tau_{ij}(0) > b$ as $\theta$ increases, or it can increase to $b$ if $\tau_{ij}(0) < b$ as $\theta$ increases (**). Now, since local pheromone update decreases the pheromone value of an edge, one can consider the global pheromone update only and obtain an upper bound on $\tau_{ij}(\theta)$. By the result obtained in (*), and the fact that local pheromone update is ignored, one can conclude that the maximum value that $\tau_{ij}(\theta)$ can take is $q_f(s^\star)$. In addition to this, since $\tau_{ij}(0){=}\tau_0 < q_f(s^{bs}) = b$, then by (**), $\tau_{ij}(\theta)$ increases, hence, $\tau_{ij}(\theta)$ cannot fall below the value of $\tau_0$. As a result, $\tau_{min} = \tau_0$ and $\tau_{max} = q_f(s^\star)$ exist for the ACS. [9]

Therefore, given the results obtained in the previous paragraph, what remains to be shown is that any feasible solution in $\widetilde{S}$ can be constructed with non-zero probability. Suppose that an ant $k$ is on component/vertex $c_i$. Suppose also that the edge $e = \{c_i, c_j\}$ with the property, $c_j \notin M_k$ is not the most feasible edge to choose (i.e if $q \leq q_0$ $e$ will not be chosen). Hence, according to Equation 4.0.1, the edge $\{c_i, c_j\}$ is only chosen when a biased randomized choice

is made. Therefore, the probability of choosing edge $\{c_i, c_j\}$ is that of $(1 - q_0).p'$, where $p'$ is the probability of choosing the edge $\{c_i, c_j\}$ defined by the probabilistic function in Equation 4.0.2. Now, if $0 < \eta_{ij} < \infty$ for all $i, j \in N$ (which in the case of the TSP it is), and $\beta < \infty$, then for all $i, j \in [N]$, $\eta_{ij}$ belongs to the interval $[\eta_{min}, \eta_{max}]$, where $\eta_{min}$ and $\eta_{max}$ are the minimum and maximum elements that all $\eta_{ij}$'s may take, respectively. As a result, a lower bound on $p'$ would be $\widetilde{p'} = \dfrac{\tau_{min}\eta_{min}^{\beta}}{(N-1)\tau_{max}\eta_{max}^{\beta} + \tau_{min}\eta_{min}^{\beta}}$. Hence, the probability of choosing the edge $\{c_i, c_j\}$ is bounded from below by $(1 - q_0)\widetilde{p'}$. As a result, since $i$ and $j$ are arbitrary, the value $0 < (1 - q_0)\widetilde{p}_{min}$ is a lower bound to the probability of any choice while constructing the solution. Therefore, the probability of any feasible solution to be generated is greater than zero. As a result, due to what has been discussed before, Theorem 4.0.4 applies to the ACS. [9]

From the above two paragraphs it can now be deduced that the ACS algorithm converges in value to the optimal value of any static problem instance it is applied to, if given enough time to execute. Hence, when applied to the TSP, the ACS converges in value to the weight of a minimum weight Hamiltonian cycle, if given enough time to execute. This means that when applied to the TSP, the ACS is guaranteed to find a minimum weight Hamiltonian cycle at least once (i.e finds), if it is executed long enough. It is important to note that, although it is guaranteed that the ACS will find a minimum weight Hamiltonian cycle when applied to the TSP if it is executed long enough, it is not guaranteed to do so in polynomial time. In fact according to [9], there are no theoretical results about the speed of convergence of any ACO algorithm. Therefore, the algorithmic performance in terms of speed of convergence of any ACO algorithm can only be measured empirically [9]. Another important deduction that could be made from Theorem 4.0.4 is that, the ACS is not susceptible to the problem presented in Example 3.2.5. This is true because, Theorem 4.0.4 guarantees that a minimum weight Hamiltonian cycle will be found if given enough time, regardless of the starting positions of every ant. Hence, the question that remains to be asked is whether in practice, the ACS applied to the TSP gives better results than other classical heuristic algorithms for the TSP, such as the NNA and the TAMSA.

Having presented the theory behind the NNA, the TAMSA, and the ACS algorithm, it

is now time to analyze the theory empirically. Therefore, an entire chapter will be dedicated to the empirical analysis of the NNA, the TAMSA, and the ACS algorithm, where in this chapter, the following will be done. Firstly, a brief description of the NNA implementation, the TAMSA implementation, and the ACS algorithm implementation which where carried out will be given. Secondly, the TAMSA and the NNA will be tested on Metric-TSP instances of different sizes, where the aim of the test is to check how much close the approximation values returned by the algorithms are to the bounds stated in Theorem 3.2.13 and Theorem 3.2.7 respectively. Thirdly, an empirical test will be carried out on the ACS algorithm, where the aim of this test is to analyze how the ACS algorithm approaches towards an optimal solution, as the number of iterations increase. Finally, to determine which algorithm gives the best approximation results, the approximation values of the ACS algorithm applied to the TSP will be compared to those of the TAMSA and the NNA. Note that due to this comparison, the ACS will be applied to the same Metric-TSP instances which will be applied to the TAMSA and the NNA.

# 5 Computational Work

Having presented the TAMSA, the NNA, and the ACS algorithm theoretically, it is now time to evaluate this theory empirically. Therefore, this chapter is divided into four sections. The first three sections contain the following. Firstly, the subsection dedicated to a brief description of the implementation of the algorithm will be given. Note that all algorithms where written by the author in Java. Secondly, the second subsection will deal with the approximation results returned by the algorithm. In the third subsection, in the case of the NNA and the TAMSA, these results are going to be used in order to check how close are the approximations to the approximation ratios stated in Theorems 3.2.7 and 3.2.13. On the other hand, in the case of the ACS algorithm, an analysis will be carried out to study how the implemented ACS algorithm approaches towards an optimal solution, as the number of iterations increase. Finally, in the fourth an final section, the approximation results obtained for every algorithm will also be used to compare the quality of the approximations returned by ACS algorithm, to those of the NNA and the TAMSA. The following section is about the empirical analysis of the NNA.

## 5.1 The NNA Empirically

As already mentioned, the main aim of this section is to present the approximation values returned by the NNA given TSP instances of different sizes, and to analyze how close are these approximation values to the approximation ratio stated in Theorem 3.2.7. In addition to this, the factor by which the approximation values vary from the optimal value with respect to instance size will be analyzed. However, before doing all this, the implementation of the NNA which was carried out will be briefly described. Note that, for a detailed description on the NNA implementation which was written, the reader is encouraged to either read the inline comments in the file NearestNeighbourHeuristicEngine.java located in the folder MathematicsThesis/src/TSP/ApproximationAlgorithms/NearestNeighbour/Approximation, or the java docs located in the same folder as the NearestNeighbourHeuristicEngine.java file.

### 5.1.1 Implementation Description

The implementation of the NNA was based on Algorithm 3.2.1 in Section 3.2.1. However, there are two main differences which were carried out. The first difference is that, the NNA

was implemented in such a way that it does not choose a random vertex as a starting vertex, but it gives each vertex the opportunity to act as a starting vertex. In other words, if $G(V, E, f)$ is the inputted TSP instance, $|V|$ Hamiltonian cycles are computed, where each Hamiltonian cycle has a different starting vertex from V. That being said, the NNA then returns the weight of the Hamiltonian cycle of least weight among the $|V|$ Hamiltonian cycles that it computes. A consequence of this change is that, given $G$, the NNA will always return the best approximation of $G$ that it can return. Another difference is that unlike the NNA pseudocode presented in Algorithm 3.2.1, this NNA implementation returns the weight of the approximated Hamiltonian cycle, not the actual Hamiltonian cycle. This was done in this way because, it will be much easier in Section 5.4 to compare the results of the NNA to other algorithms.

The NNA's logic was implemented using two methods which where called, $approximateTsp$ and $closestCity$. Given a vertex $u$ and information related to the input graph instance $G$, the $closestCity$ method iterates over all the vertices of $G$, and returns the unvisited vertex which is connected to $u$ via the edge of least weight, among all edges incident to $u$. This was carried out by iterating over all the vertices in $G$, and implementing a minimum function over all the feasible edges (unvisited and connected to $u$), in terms of their weights. On the other hand, the $approximateTsp$ method iterates over all the vertices in $V$, and at every iteration $i$, a Hamiltonian cycle $H$ is computed by starting from vertex $i$, and always visiting the vertex returned by the $closestCity$ method, until all vertices in $G$ are visited. After $H$ is computed, if $H$ is the Hamiltonian cycle of least weight that has been encountered since the first iteration, the weight of $H$ is stored in the variable $bestTourLength$. After the $approximateTsp$ method completes all the iterations, the value stored in $bestTourLength$ is returned, and the NNA stops executing.

After briefly describing the NNA implementation which was written, it is now time to present the approximation results obtained when the NNA algorithm was applied to TSP instances of different sizes. In addition to this, the results are then compared with the approximation ratio stated in Theorem 3.2.7, to check how close they are to the approximation ratio. This will eventually suggest whether the approximation ratio in Theorem 3.2.7 is a trivial bound

or not. Note that, since the NNA was optimized to return the best approximation possible that it can return, this analysis only applies to the best result that the NNA can return. In other words this means that, although the best approximation value to be returned by the NNA may be far away from the bound stated in Theorem 3.2.7, the NNA as presented in Section 3.2.1 may still return approximations which are close to the theoretical bound. Note that the analysis will be restricted to the best result possible because in practice, it is more desirable to obtain the best result possible from an algorithm.

### 5.1.2 Results and Analysis of Results

The NNA implementation described in the previous paragraph was applied to TSP instances of different sizes, obtained from the TSPLIB library in [18]. The TSPLIB library is a database consisting of different TSP instances of different sizes, whose optimal value is known. Each TSPLIB instance is stored in a text file in the database, containing information relevant to the input graph instance. The TSPLIB instances which where used in this project contain a number of coordinates, that represent the position of the vertices in the considered space. In addition to this, the TSP instances also contain information on how the distance between these vertices should be defined. For this project, the space and metric that will be considered are $\Re^2$, and the Euclidean distance $d$ respectively, where the Euclidean distance between two points $x = (x_1, x_2)$ and $y = (y_1, y_2)$ in $\Re^2$ is defined by the following formula:

$$d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} \qquad (5.1.1)$$

The space $(\Re^2, d)$ was chosen because, the approximation ratio presented in Theorem 3.2.7 applies for the NNA when applied to Metric-TSP instances. Note that the fact that $(\Re^2, d)$ is a metric space will not be proven because it is beyond the scope of this project. Another reason why the space $(\Re^2, d)$ was chosen was that the majority of the TSPLIB instances are defined in the space $(\Re^2, d)$. Note that since all the TSPLIB instances which where used are defined in $(\Re^2, d)$, the empirical analysis which will be carried out for the NNA, TAMSA, and the ACS algorithm only applies to cases where the input graph is defined in $(\Re^2, d)$. Having briefly described the TSP instances which were used, it is now time to present the approximation values returned by the NNA, when applied to chosen TSP instances defined

in $(\Re^2, d)$. These results are summarized in Table 5.1.1 below.

Table 5.1.1: NNA Appoximation Results.

| TSPLIB instance | Instance size | optimal value | approximation value |
|---|---|---|---|
| eil51 | 51 | 426 | 513.61 |
| rat99 | 99 | 1211 | 1369.53 |
| ch130 | 130 | 6110 | 7198.74 |
| gil262 | 262 | 2378 | 2768.34 |
| lin318 | 318 | 41029 | 49215.61 |
| pcb442 | 442 | 50778 | 59064.27 |
| u574 | 574 | 36905 | 44605.14 |
| p654 | 654 | 34643 | 40910.31 |
| u724 | 724 | 41910 | 51795.59 |
| u1060 | 1060 | 224094 | 276638.82 |

Using the results in Table 5.1.1 above, the approximation factor of every TSP instance can be computed and compared with the approximation ratio stated in Theorem 3.2.7. Note that the approximation factor is the value obtained from calculating the fraction $\frac{\text{approximation value}}{\text{optimal value}}$. As a result, the smaller the approximation factor is, the better it is because, the closer the approximation value is to the optimal value. The approximation factors of the chosen instances are all summarized in Table 5.1.2 below.

Table 5.1.2: NNA Further Results.

| TSPLIB instance | Approximation Factor | Bound from Theorem 3.2.7 |
| --- | --- | --- |
| eil51 | 1.21 | 3.5 |
| rat99 | 1.13 | 4 |
| ch130 | 1.18 | 4.5 |
| gil262 | 1.16 | 5 |
| lin318 | 1.2 | 5 |
| pcb442 | 1.16 | 5 |
| u574 | 1.21 | 5.5 |
| p654 | 1.18 | 5.5 |
| u724 | 1.24 | 5.5 |
| u1060 | 1.23 | 6 |

From Table 5.1.2 one can observe that for each chosen instance, the approximation factor is not close to the approximation ratio stated in Theorem 3.2.7. This means that the approximation ratio stated in Theorem 3.2.7 does not reflect well what in practice the NNA can at best return. Note that as described before, this statement applies only to the optimized NNA implementation which was carried out in this project. This means that the NNA implementations who's approximation factors depend on the starting vertex, could result into the bound being sharp. Another observation from Table 5.1.2 is that although the instance size is increased, the approximation factor remains within an average of approximately 1.19. This means that although in general for any Metric-TSP, it is theoretically known that the approximation ratio increases logarithmically with respect to the number of vertices, the empirical results suggest that for TSP instances in $(\Re^2, d)$, the best solution that the NNA may return given any instance is within a constant approximation ratio which is close to 1.19. From these observations it can be concluded that in practice, the NNA implementation which was carried out in this project returns approximations which are closer to the optimal value than what Theorem 3.2.7 suggests. Having empirically analyzed the NNA, it is now time to empirically analyze the TAMSA.

## 5.2 The TAMSA Empirically

As already described, the main aim of this section is to analyze the TAMSA empirically. Similarly to what has been done for the NNA, the main aim of the TAMSA's empirical analysis is to test whether the theoretical bound stated in Theorem 3.2.13 is trivial or not. However before doing so, the TAMSA implementation which was carried out will be briefly described. Note that the TAMSA was written in the file TwiceAroundMSTHeuristic.java, which can be located in the folder MathematicsThesis/src/TSP/ApproximationAlgorithms/TwiceAroundMST. Also, for a more detailed description about the TAMSA's implementation, the reader is encouraged to either read the java docs located in the same folder as the TAMSA implementation, or read the inline documentation in the TwiceAroundMSTHeuristic.java file.

### 5.2.1 Implementation Description

The TAMSA implementation was based on Algorithm 3.2.2 presented in Section 3.2.2, however, with two differences. The first difference is that, the TAMSA implementation which was carried out does not choose the starting vertex randomly. Instead, it gives every vertex in the input graph instance the opportunity to act as a starting vertex. As a result, if $G(V, E, f)$ is the input graph instance, $|V|$ Hamiltonian cycles are generated, and then the TAMSA returns the Hamiltonian cycle of least weight that it generates. This modification was done so that given $G$, the TAMSA always returns the best approximation of $G$ that it can return. The second difference is that the TAMSA returns the weight of the best Hamiltonian cycle it finds, not the Hamiltonian cycle itself. Again, this was done so that in Section 5.4, the approximations from different algorithms can be easily compared.

The TAMSA logic was implemented using two functions named $approximateTSP$, and $DFS$. The $approximateTSP$ function iterates over the set of vertices $V$, and gives each element of $V$ the opportunity to act as a starting vertex. At each iteration $j$, a minimum weight spanning tree $T$ of $G$ is calculated using Prim's Algorithm, by starting from vertex $j$. Afterwards, the $DFS$ function is used to compute a preorder tree walk $W$ of $T$ by starting from vertex $j$. Then, the weight of the Hamiltonian cycle $H$ constructed from $W$ is calculated, and if $H$ is the Hamiltonian cycle of least weight that the TAMSA has encountered since the first itera-

tion, the weight of $H$ is stored in the variable $result$. As a result, after the $approximateTSP$ function finishes iterating over $V$, $result$ would contain the best approximation value for $G$ that the TAMSA has encountered. Hence, the value of the variable $result$ is returned.

As indicated in the previous paragraph and Subsection 3.2.2, the TAMSA requires the use of Prim's Algorithm to compute a minimum weight spanning tree. Therefore, Prim's Algorithm had to be implemented, where it's implementation can be found in classes PrimMST.java, and PrimsPriorityQueueEntry.java, both located in the folder MathematicsThesis/src/TSP/Approximation Algorithms/TwiceAroundMST. Recall that in Section 3.2.2, it was argued that Prim's Algorithm repeatedly chooses the edge of least weight connecting a vertex which is already in the MST, to a vertex which is not in the MST, until all vertices are in the MST. Therefore, edges possessing this property are needed to be ranked according to their weight. This was carried out by implementing a priority queue consisting of vertices which are still not in the MST, where the topmost entry in the priority queue is the vertex connected by an edge of least weight to a vertex which is already in the MST.

Prim's Algorithm logic described in Section 3.2.2 was implemented using the function 'calculateMinimumWeightSpanningTree'. Given a starting vertex, the 'calculateMinimumWeightSpanningTree' function first creates a priority queue entry for each of the vertices, and adds each vertex's priority queue entry to the priority queue. This was done because, at the start of the algorithm no vertex is in the MST. Then, the 'calculateMinimumWeightSpanningTree' function starts popping out vertices from the top of the priority queue, where for each popped vertex $u$, the priority queue entry of an adjacent vertex $x$ to $u$ which is not in the MST is modified to $f(\{x, u\})$, if $f(\{x, u\})$ is less than the weight of any edge which has been encountered having $x$ as one of it's end vertices. In addition to this, $u$ is added to the MST, by connecting it to a vertex which is in the MST such that the connecting edge is of least possible weight. This procedure is then repeated until the priority queue is empty (i.e. all vertices are in the MST).

Having briefly described the implementation of the TAMSA, it is now time to empirically analyze the TAMSA. As already mentioned, the approximation results of the TAMSA applied

to various TSP instances will first be given, and then used to check if the approximation ratio stated in Theorem 3.2.13 is sharp.

## 5.2.2 Results and Analysis of Results

Similarly to what has been done in Section 5.1.2, the TAMSA was applied to TSPLIB instances of different sizes. The TSPLIB instances which where inputted to the TAMSA were the same instances which where applied to the NNA. This was done so that in Section 5.4, the approximations of different algorithms can be compared easily. It is important to note that similarly to the NNA, given a TSP instance $G$, the TAMSA was optimized to return the best approximation of $G$ that it can return. Another important note is that the TAMSA was applied to instances defined in $(\Re^2, d)$, where $d$ is the metric defined in Equation 5.1.1. As a result, the empirical analysis of the TAMSA which will be carried out in this subsection applies only to TSP instances which are defined in $(\Re^2, d)$. This was done in this way because of the same reasons which where discussed in Section 5.1.2. The approximation results of the TAMSA are summarized in Table 5.2.1 below.

Table 5.2.1: TAMSA Appoximation Results.

| TSPLIB instance | Instance size | optimal value | approximation value |
|---|---|---|---|
| eil51 | 51 | 426 | 576.30 |
| rat99 | 99 | 1211 | 1599.15 |
| ch130 | 130 | 6110 | 8104.31 |
| gil262 | 262 | 2378 | 3268.79 |
| lin318 | 318 | 41029 | 56327.52 |
| pcb442 | 442 | 50778 | 68975.65 |
| u574 | 574 | 36905 | 49302.34 |
| p654 | 654 | 34643 | 47296.09 |
| u724 | 724 | 41910 | 59409.74 |
| u1060 | 1060 | 224094 | 301216.19 |

Using the results in Table 5.2.1, one can also compute the approximation factor of each TSPLIB instance, and compare it to the approximation ratio stated in Theorem 3.2.13. The approximation factor of each instance in Table 5.2.1 is presented in Table 5.2.2 below.

Table 5.2.2: TAMSA Further Results.

| TSPLIB instance | Approximation Factor | Bound from Theorem 3.2.13 |
|---|:---:|:---:|
| eil51 | 1.35 | 2 |
| rat99 | 1.32 | 2 |
| ch130 | 1.33 | 2 |
| gil262 | 1.37 | 2 |
| lin318 | 1.37 | 2 |
| pcb442 | 1.36 | 2 |
| u574 | 1.34 | 2 |
| p654 | 1.37 | 2 |
| u724 | 1.42 | 2 |
| u1060 | 1.34 | 2 |

Comparing Table 5.1.2 with Table 5.2.2 one can observe that the approximation ratio of Theorem 3.2.13 is much closer to the best approximation factor that the TAMSA can return, compared to how close is the approximation ratio in Theorem 3.2.7 to the best approximation factor that the NNA can return. In addition to this, one can observe from Table 5.2.2 that, the best approximation factor that the TAMSA can return is approximately 1.36 on average. Therefore, compared to the NNA, although the approximation ratio in Theorem 3.2.13 is much closer to the best approximation factor that the TAMSA can return, the approximation ratio still does not reflect well the approximations that are to be expected from the TAMSA in practice. Also, the approximation factor for each TSPLIB instance did not vary too much from 1.36. This could also mean that in practice, given a Euclidean TSP instance $G$, the best approximation factor that the TAMSA can return is within a constant $a$ from the optimal value, where $a$ is a value which is close to 1.36. Having empirically analyzed the TAMSA, it is now time to empirically analyze the ACS algorithm when applied to the same TSPLIB instances which where applied to the TAMSA and the NNA.

## 5.3 The ACS Empirically

As already described, the main aim of this section is to empirically analyze the ACS algorithm when applied to the TSP. Unlike the TAMSA and the NNA, no approximation ratio is known

for the ACS algorithm when applied to the TSP. On the other hand, as discussed in Chapter 4, what is known is that if the ACS is given enough time to execute, it will converge in value to the optimal value of the input graph instance. In addition to this, it was also argued in Chapter 4 that, there are no theoretical results about the speed of convergence of any ACO algorithm. Hence, apart from presenting the approximation results of the ACS when applied to different TSP instances, the speed of convergence of the ACS algorithm when applied to some chosen TSP instances will be studied. The speed of convergence will be studied by constructing a plot depicting how the ACS algorithm approaches the optimal value of the input graph instance as the number of iterations increases. As a result, this plot could then show the nature of the speed of convergence of the ACS algorithm when applied to the chosen TSP, and hence, if it is expected that the ACS algorithm finds an optimal value in polynomial time.

But before conducting the above, the ACS algorithm implementation which was carried out will be briefly described. The ACS algorithm was written in the file ACOEngine.java located in the folder MathematicsThesis/src/TSP/ApproximationAlgorithms/ACO. Note that in the following subsection, the ACS algorithm implementation will not be described in great detail. The reason is that java docs and inline comments where written to describe the specific details of the implementation. Note that the jav adocs can be located in the same path as the ACOEngine.java file, and the inline comments are written in the ACOEngine.java file.

### 5.3.1  Implementation Description

The ACS algorithm implementation which was carried out was based on Algorithm 4.0.1, with only one difference. The only difference is that instead of returning the best Hamiltonian cycle it finds, the ACS algorithm was implemented to return the weight of the best Hamiltonian cycle that it finds. Again, this was done so that in Section 5.4, the results of the ACS algorithm can be easily compared to those of the TAMSA and the NNA. The ACS algorithm logic was implemented using a number of functions located in the ACO-Engine.java file. However, the main function is called the $approximateTSP$ function. The $approximateTSP$ function starts by placing $m$ ants on randomly selected vertices of the input graph instance. Then, all ants are moved to a new vertex according to the probabilistic function $determineNextCity$, which implements Formula 4.0.1 and 4.0.2. When an ant is moved

to a new city, the chosen edge's pheromone is locally updated using the *localTrailUpdating* function, which implements Formulas 4.0.4. When each ant computes a Hamiltonian cycle, the weight of the iteration's best Hamiltonian cycle is compared to the weight of the global best Hamiltonian cycle, and if the weight of the iteration's best Hamiltonian cycle is smaller than that of the global best Hamiltonian cycle, the global best Hamiltonian cycle is set to be equal to the iteration's best Hamiltonian cycle. Afterwards, the global best ant deposits pheromone on the global best Hamiltonian cycle using the *globalTrailUpdating* function, which implements Formula 4.0.3. The whole procedure is then repeated until a termination condition is met. When the termination condition is met, the weight of the global best Hamiltonian cycle is returned as an approximation.

As it could be observed in Algorithm 4.0.1, and the discussion in Chapter 4, there are a number of parameters who's value is still not given. Such parameters are, the number of ants in the algorithm, $\beta$ as defined in Formula 4.0.1, and more. In general, different TSP instances require different parameter values for optimal execution, however, according to [9], the values summarized in Table 5.3.1 were empirically found to be robust across many problems and instances. Therefore, in the implementation of the ACS algorithm, the parameter values represented in Table 5.3.1 below where taken. Note that in Table 5.3.1 below, $C^{nn}$ denotes the approximation value returned by the NNA, and $n$ denotes the number of vertices in the TSP graph instance.

Table 5.3.1: ACS algorithm parameter values.

| Parameter | Value |
|---|---|
| $\beta$ | 5 |
| $p$ | 0.1 |
| $m$ | 10 |
| $\tau_0$ | $\frac{1}{nC^{nn}}$ |
| $\zeta$ | 0.1 |
| $q_0$ | 0.9 |

Another important parameter in the ACS algorithm which has not yet been described is the termination condition of the ACS algorithm. In this implementation, the ACS algorithm

was implemented to stop executing after 10000 iterations. Note that the termination condition was chosen in this way because, the ACS algorithm could not be made to terminate when it finds an optimal value because, Theorem 4.0.4 does not specify how much time needs to pass for the ACS algorithm to find an optimal value (i.e. this could be arbitrarily large). Having briefly described the ACS algorithm implementation, it is now time to empirically analyze the ACS algorithm when applied to the specific TSP instances.

### 5.3.2  Results and Analysis of Results

Similarly to the NNA and the TAMSA, the ACS algorithm was applied to TSPLIB instances of different sizes. The TSPLIB instances which where inputted to the ACS algorithm where the same instances which where inputted to the TAMSA and the NNA. Again, this was done so that the TAMSA, the NNA, and the ACS algorithm could be compared in terms of approximation accuracy. As a consequence of this decision, the empirical analysis which will be carried out in this section only applies to TSP instances which are defined in $(\Re^2, d)$, where $d$ is the metric defined by Equation 5.1.1. The approximation results of the ACS algorithm applied to different TSPLIB instances are summarized in Table 5.3.2 below. Note that on every TSPLIB instance $I$ the algorithm was executed three times, and the smallest approximation value obtained from the three runs was written in Table 5.3.2. In addition to this, each run was executed for a number of iterations as indicated under the '#iters' column in Table 5.3.2. Note that for every TSP instance, the algorithm was executed for a number of iterations which is much greater than the termination condition described in Section 5.3.1. This was done to try and obtain the best result possible. Note that, for large TSP instances, less iterations where executed because, the ACS algorithm takes too long to execute.

Table 5.3.2: ACS Algorithm Appoximation Results.

| TSPLIB instance | Instance size | optimal value | approximation value | #iters |
|---|---|---|---|---|
| eil51 | 51 | 426 | 432.77 | 200,000 |
| rat99 | 99 | 1211 | 1221.01 | 200,000 |
| ch130 | 130 | 6110 | 6209 | 200,000 |
| gil262 | 262 | 2378 | 2401 | 200,000 |
| lin318 | 318 | 41029 | 43483 | 100,000 |
| pcb442 | 442 | 50778 | 51829 | 70,000 |
| u574 | 574 | 36905 | 40798.46 | 50,000 |
| p654 | 654 | 34643 | 35200.82 | 50,000 |
| u724 | 724 | 41910 | 43384.83 | 50,000 |
| u1060 | 1060 | 224094 | 246771 | 50,000 |

Similarly to what has been done in Section 5.2.2, and Section 5.1.2, the approximation factor will be calculated for each instance in Table 5.3.2. The approximation factor results are summarized in Table 5.3.3 below.

Table 5.3.3: ACS Further Results.

| TSPLIB instance | Approximation Factor |
|---|---|
| eil51 | 1.02 |
| rat99 | 1.01 |
| ch130 | 1.02 |
| gil262 | 1.01 |
| lin318 | 1.06 |
| pcb442 | 1.02 |
| u574 | 1.11 |
| p654 | 1.02 |
| u724 | 1.04 |
| u1060 | 1.1 |

From Table 5.3.3 one can conclude that, the approximation factor of the ACS algorithm when applied to TSP instances defined in $(\Re^2, d)$ is approximately 1.041 on average. In ad-

dition to this, one can observe that, as the instance size increased, the approximation factor remained close to 1.041. This means that in practice, it might be that for TSP instances defined in ($\Re^2$, d), the ACS algorithm might have a constant approximation ratio which is close to the value of 1.043, if executed for 200,000-50,000 iterations. Note that, although it is known that the ACS algorithm applied to the TSP converges in value to the optimal value if given enough time to execute, neither of the experimental runs resulted into the ACS algorithm returning the optimal value of the TSPLIB instance it was applied to. This could mean that the ACS algorithm does not converge in value to the optimal value in polynomial time. To try and verify if this statement is correct or not, another empirical experiment was carried out on the ACS algorithm applied to the chosen TSPLIB instances.

This empirical experiment involved applying the ACS algorithm described in Section 5.3.1 on four instances from Table 5.3.2, where at every iteration of the ACS algorithm, the iteration number and the weight of the global best Hamiltonian cycle was outputted. Afterwards, the output was used to create a plot which shows how the ACS algorithm approaches the optimal value as the number of iterations increase. In addition to this, the empirical experiment was executed three times for 2,000 iterations on every instance, and results where recorded for each run. The results obtained from the three runs were then averaged, and the averaged results where used to construct the respective plots. The following plots represent how the ACS algorithm approximately on average approaches the optimal value of the TSPLIB instances 'eil51', 'gil262', 'pcb442', and 'p654' as the number of iterations increases respectively. Note that the plots where constructed using the Python Script 'plot_script.py' located in the folder named 'src'.

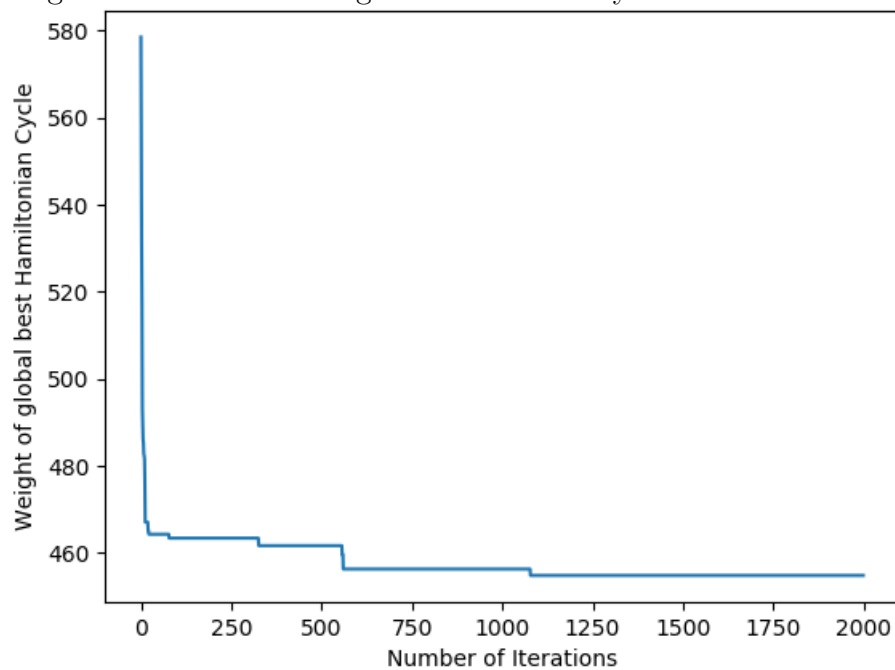Figure 5.3.1: Global Weight Hamiltonian Cycle vs Number of Iterations for eil51



Figure 5.3.2: Global Weight Hamiltonian Cycle vs Number of Iterations for gil262
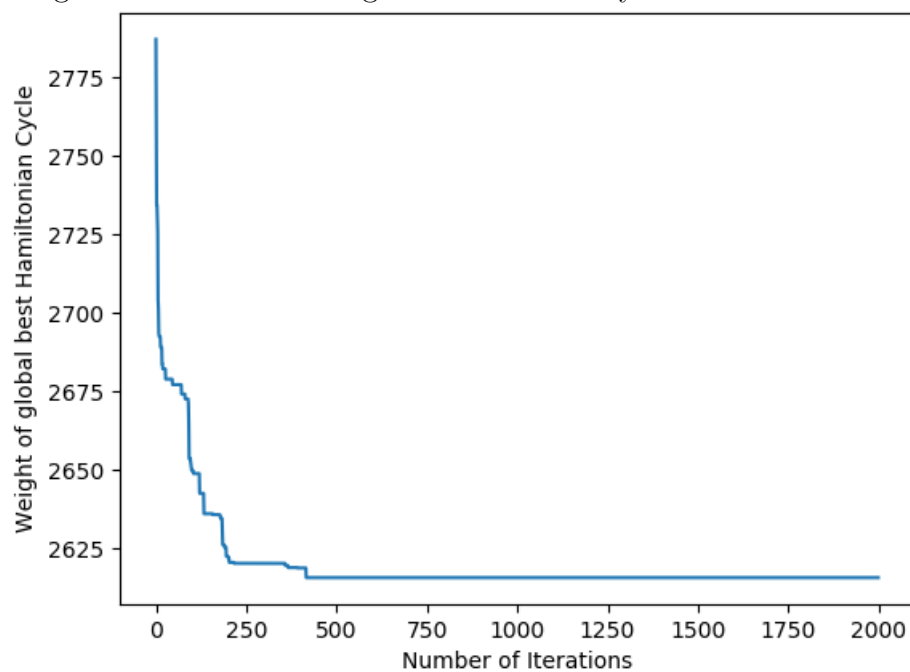
Figure 5.3.3: Global Weight Hamiltonian Cycle vs Number of Iterations for pcb442
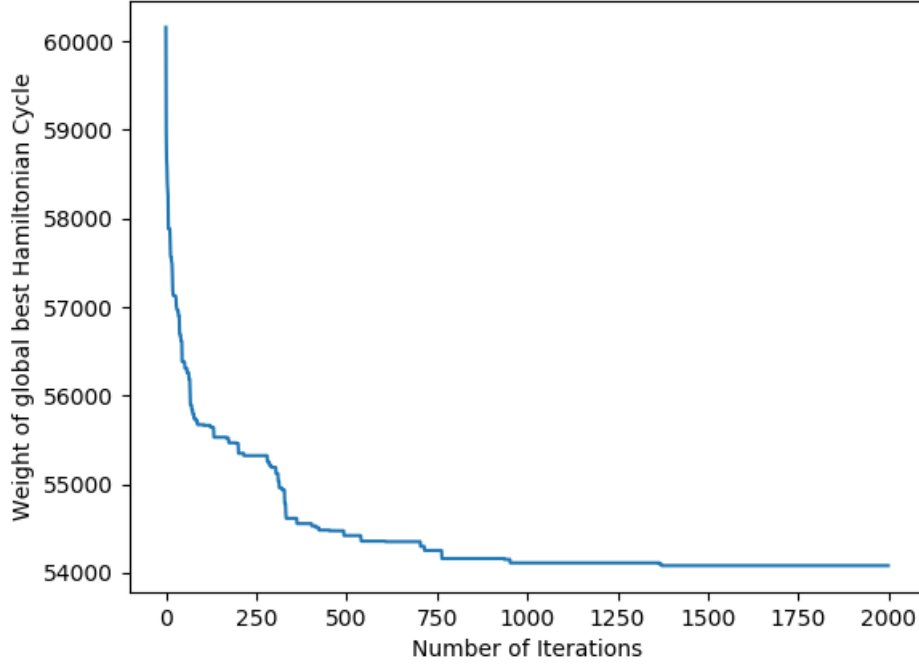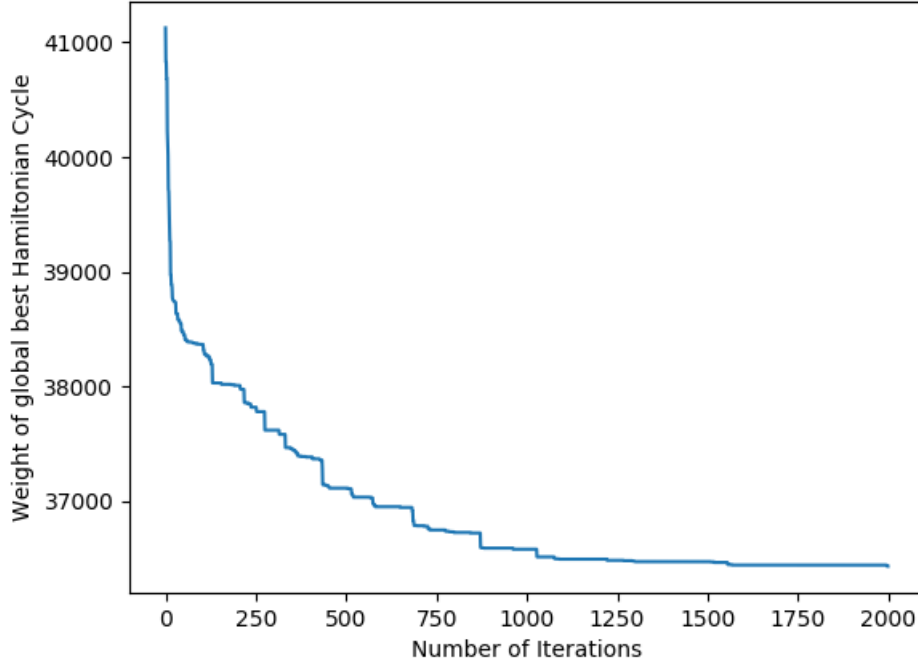


Figure 5.3.4: Global Weight Hamiltonian Cycle vs Number of Iterations for p654



From Figures 5.3.1, 5.3.2, 5.3.3, and 5.3.4 it can be deduced that the weight of the global best Hamiltonian cycle for instances in $(\Re^2, d)$ is inversely proportional to the number of

iterations executed. In fact, it can be observed from the figures above that when the number of iterations is small, the weight of the global best Hamiltonian cycle decreases at a faster rate compared to when the number of iterations is large. This is reflected in the figures above because, when the number of iterations is small, the curve decreases, but as the number of iterations increase, the rate of decrease of the weight of the global best Hamiltonian cycle tends to zero. This means that as the number of iterations increase, the ACS algorithm will find it harder to find shorter Hamiltonian cycles. Note that from the figures above, the observations mentioned in this paragraph become more evident as the instance size increases.

The observations described in the previous paragraph have some important consequences. The main consequence is that since the weight of the global best Hamiltonian cycle for instances in $(\Re^2, d)$ is inversely proportional to the number of iterations executed, then for instances in $(\Re^2, d)$, the ACS algorithm converges in value to the optimal value in a manner which is inversely proportional to the number of iterations. This means that since the rate of decrease of the global weight Hamiltonian cycles tends to zero as the number of iterations are increased, the ACS algorithm may reach a point where a lot of iterations need to pass for the ACS algorithm to find shorter Hamiltonian cycles. As a result, an infeasible amount of time may need to pass for the ACS algorithm to find shorter Hamiltonian cycles. For example, in the blue curve of Figure 5.3.3 above, a lot of iterations need to be executed after the 750th iteration for the ACS algorithm to find a shorter Hamiltonian cycle, compared to after the 150th iteration.

It can be concluded from the above paragraph that, although it is theoretically shown that the ACS will find an optimal solution if given enough time to execute, this empirical analysis suggests that an optimal solution is not to be expected because, an infeasible amount of time may need to pass for the ACS algorithm to find shorter Hamiltonian cycles. Having empirically analyzed the ACS algorithm, it is now time to compare the approximation results obtained from the empirical analysis of the TAMSA, the NNA, and the ACS algorithm, and deduce whether for instances defined in $(\Re^2, d)$, the ACS algorithm outperforms the NNA and the TAMSA in terms of accuracy of approximations.

## 5.4 Comparison of the Empirical Results of the ACS to those of the TAMSA, and the NNA

Comparing Tables 5.1.1, 5.2.1, and 5.3.2, it is evident that in terms of accuracy of approximations, the ACS algorithm outperforms both the TAMSA and the NNA for all the TSP instances in $(\Re^2, d)$ which where chosen. This suggests that for TSP instances defined in $(\Re^2, d)$, the ACS algorithm should be used instead of the TAMSA and the NNA, regardless of the fact that the TAMSA and the NNA have approximation ratios which are known. Therefore for instances in $(\Re^2, d)$, it can be confirmed that it is better in terms of accuracy of approximations to design an algorithm which is guaranteed to find an optimal solution if given enough time, rather than designing an algorithm who's approximation ratio is known (recall the statement made in Section 3.2.2).

Apart from the fact that the ACS algorithm outperformed both the TAMSA and the NNA in this empirical analysis, it is also good to deduce which algorithm from the TAMSA and the NNA gives the best approximations. From Tables 5.1.1 and 5.2.1 it can be deduced that the NNA outperforms the TAMSA in all instances defined in $(\Re^2, d)$ which it was applied to. This result may come as a surprise because, the NNA's approximation ratio increases logarithmically with respect to the instance size unlike that of the TAMSA which is constant. It is important to note that this result may only be true because, the TAMSA and the NNA where optimized as discussed in Sections 5.2.1 and 5.1.1 to return the best result possible. Hence, it may be the case that if not optimized, the TAMSA may give better results on average than the NNA. However, regardless of this, the empirical results suggest that for instances in $(\Re^2, d)$, the NNA at best will give better approximations than the TAMSA.

# 6 Conclusion

The theoretical and empirical analysis aims set out in Chapter 1 were all achieved. In fact, it was argued in Chapter 4 that due to Theorem 4.0.4, the ACS algorithm will always find an optimal solution when applied to the TSP if given enough time to execute. In addition to this, it was argued that Theorem 4.0.4 applies to every ACO algorithm that satisfies the following properties. The first property is that the matrix of pheromones T must have it's entries defined in some fixed interval $[\tau_{min}, \tau_{max}]$, where $\tau_{min}, \tau_{max} > 0$. The other property is that every feasible solution that can be constructed by an ant has a non-zero probability of being constructed. Although it was not part of the theoretical aims set out in Chapter 1, it was also shown that for specific instances of the TSP (Metric-TSP), the TAMSA and the NNA have approximation ratios as stated in Theorem 3.2.13 and Theorem 3.2.7 respectively. These approximation ratios were analyzed empirically, and it was concluded that for TSP instances that are defined in the 2-D Euclidean Space, if the TAMSA and the NNA are optimized to return the best result possible, the approximation ratios in Theorem 3.2.7 and Theorem 3.2.13 are not close to the approximation factors (approximation factor defined in Section 5.1.2) that are achieved in practice. This could prompt further theoretical analysis to study whether an approximation ratio that is closer to the approximation factors that can be achieved in practice can be found, for the case when the NNA or the TAMSA are optimized to return the best result possible, where the NNA and the TAMSA are applied to TSP instances defined in the 2-D Euclidean Space.

The aims of the empirical analysis were also achieved. In fact, it was concluded that the ACS algorithm returns better approximations than the TAMSA and the NNA for TSP instances defined in the 2-D Euclidean Space. This means that the ACS algorithm should be used over the NNA and the TAMSA, for approximating TSP instances defined in the 2-D Euclidean Space. In the future, one can also test whether the ACS algorithm gives better approximations than the NNA and the TAMSA, for TSP instances defined in other metric spaces or for TSP instances which are not defined in a metric space. This analysis would give a better indication on whether the ACS algorithm is more powerful on a wider scale than the NNA and the TAMSA. Apart from this, it was also concluded that the weight of the global best Hamiltonian cycle found by the ACS algorithm is inversely proportional to the number

of iterations executed by the ACS algorithm. As discussed in Section 5.3.2, this means that a lot of time may need to pass for the algorithm to find an optimal solution. However, in modern day applications, there is no use in letting the ACS algorithm run for a very long time to only get a slight improvement from the previous global best Hamiltonian cycle. As a result, in the future one can empirically study whether there is a best approximation factor that the ACS reaches in feasible time, across a wide range of TSP instances. As a result, the ACS algorithm can be written to stop it's execution once this best found approximation factor is reached.

# References

[1] V. S. Adamchik. Algorithmic Complexity, 2009. URL: `https://www.cs.cmu.edu/~adamchik/15-121/lectures/AlgorithmicComplexity/complexity.html`.

[2] 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(1):1014–1018, Jan 2016. URL: `https://pdfs.semanticscholar.org/9fc2/8fac2603cfda11da21033a58bbb5c7b75e09.pdf`.

[3] P. E. Black and P. J. Tanenbaum. Dictionary of Algorithms and Data Structures, Feb 2019. URL: `https://xlinux.nist.gov/dads/HTML/graph.html`.

[4] J. A. Bondy and U. S. R. Murty. *Graph Theory With Applications*. Elsevier Science Publishing, 5 edition, 1982.

[5] T. Carter. Tractability of computation, May 1999. URL: `https://csustan.csustan.edu/~tom/MISC/qc-article/node10.html`.

[6] C. Chekuri and S. Im. Approximation Algorithms, Jan 2011. URL: `https://courses.engr.illinois.edu/cs598csc/sp2011/Lectures/lecture_2.pdf`.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3 edition.

[8] M. Dorigo and L. Maria Gambardella. Ant Colonies for the Traveling Salesman Problem. *Bio Systems*, Vol. 43:73–81, Feb 1997.

[9] M. Dorigo and T. Stutzle. *Ant Colony Optimization*. The MIT Press, 2004.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[11] S. Goyal. A Survey on Travelling Salesman Problem. Jan 2010.

[12] D. Guichard. An Introduction to Combinatorics and Graph Theory, Jul 2018. URL: `https://www.whitman.edu/mathematics/cgt_online/cgt.pdf`.

[13] J. M. Harris, J. L. Hirst, and M. J. Mossinghoff. *Combinatorics and Graph Theory*. Springer, 2 edition, 2008.

[14] Kaveh and Tehada. What is the definition of $P$, $NP$, $NP$-complete and $NP$-hard?, Jul 2017. URL: `https://cs.stackexchange.com/questions/9556/what-is-the-definition-of-p-np-np-complete-and-np-hard`.

[15] Richard E. Pettis. Analysis of algorithms advanced programming/practicum 15-200. URL: `https://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/aa/index.html`.

[16] M. Rajesh, S. Surya, and L. M. Murari. Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches. In Donald Davendra, editor, *Traveling Salesman Problem*, chapter 1. IntechOpen, Rijeka, 2010. URL: `https://doi.org/10.5772/12909`.

[17] S. S. Ray. *Graph Theory with Algorithms and its Applications: In Applied Science and Technology*. Springer, 2013.

[18] G. Reinelt. URL: `https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`.

[19] D. Rosenkrantz, R. Edwin Stearns, and P. M. Lewis II. An Analysis of Several Heuristics for the Traveling Salesman Problem. *SIAM J. Comput.*, Vol. 6:563–581, Sep 1977.

[20] E. Rowell. Know Thy Complexities! URL: `http://bigocheatsheet.com/`.

[21] The Oxford Math Center. Prim's Algorithm. URL: `http://www.oxfordmathcenter.com/drupal7/node/685`.

[22] V. V. Vazirani. *Approximation Algorithms*. Springer, 2 edition, 2003.