The Ant Colony Optimisation for the Travelling Salesman Problem

Dylan Galea

November 16, 2018

Contents

1 Introduction

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

1.1 Some Graph Theory

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

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

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

When 2 vertices are joined by an edge, they are said to be adjacent. This is defined formally in definition 1.1.2 below.

Definition 1.1.2. *Given a graph G(V,E),* $\forall\ u,v \in V$, *u and v are said to be adjacent if* $\{u, v\} \in E$. [**?**]

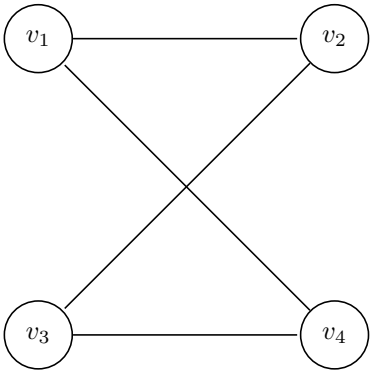
It is sometimes also required to know how many vertices are adjacent to a specific vertex in a graph.

Definition 1.1.3. *The degree of a vertex v in a graph G is the number of adjacent vertices to v in G* [**?**]. *The degree of a vertex v is denoted as deg(v).*

Any graph $G(V, E)$ can also be represented pictorially by drawing the vertices of G using circles, and by drawing the edges of G using lines between adjacent vertices. As a result, in this thesis a graph is sometimes given formally using sets or as a pictorial representation, assuming that one can be converted into another. Example 1.1.4 below depicts how a graph can be represented pictorially.

Example 1.1.4. Consider the graph *G(V,E)* such that V=$\{v_1, v_2, v_3, v_4\}$ and $E = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_1\}\}$.

Then G can be represented pictorially as :



Using definition 1.1.2, 2 adjacent vertices in G are v_1 and v_2. On the other hand, 2 non adjacent vertices in G are v_1 and v_3.
Using definition 1.1.3, the degree of every vertex in G is 2.

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

Definition 1.1.5. *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. [**?**]

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

Definition 1.1.6. *Given a graph G(V,E), a path in G joining any 2 vertices u,* $v \in V$, *is a sequence of vertices* $u = u_1$, u_2, ...,$u_n = v$ *in which no vertex is repeated and,* $\forall$ $0 < i < n$, $\{u_i, u_{i+1}\} \in E$. [**?**]

Definition 1.1.6 can now be used to define cycles and connectivity in a graph.

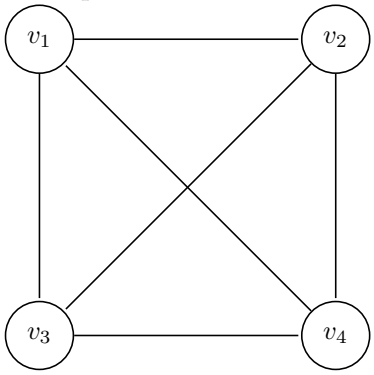
Definition 1.1.7. *A graph G(V,E) is said to be connected if* $\forall$ $u,v \in V$ $u \neq v$, *u and v are joined by a path.* [**?**]

Definition 1.1.8. *Given a graph G(V,E), a cycle in G is a path on* $n \geq 4$ *vertices, such that, the first vertex and the last vertex are equal.* [**?**]

By definitions 1.1.6, 1.1.8 above, it is clear that a cycle is a special instance of a path, with the only difference being that in a cycle, the first vertex and the last vertex are equal. Another thing worth mentioning is that, according to definitions 1.1.6 and 1.1.8, cycles and paths are sequences of vertices and not actual graphs. However, this is not the case because they can be represented easily as graphs. For example, given the path/cycle $u_1, u_2, ..., u_n$ a new graph $G(V, E)$ can be created such that, $V(G) = \{u_1, u_2, ..., u_n\}$ and $E(G) = \{\{u_i, u_{i+1}\} : \forall i, 0 < i < n\}$. For example, consider the cycle v1 v2 v3 v4 v1, then the graph depicted in example 1.1.4, is the graph representing this cycle. Such graphs are known as Cycle/Path graphs and are denoted by C_n/P_n

respecitvely, n being the number of vertices in the graph. Since this construction can be done, cycles/paths will be treated as both graphs and sequences. This will later be useful when defining Hamiltonian cycles. For better understanding of definitions 1.1.5, 1.1.6, 1.1.7 and 1.1.8, example 1.1.9 is constructed.

Example 1.1.9. Consider the graph *G(V,E)* below:



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

Another important graph theoretic concept is that of subgraphs.

Definition 1.1.10. *Given a graph G(V,E) and a graph H(V′,E′), H is a subgraph of G if* $V' \subseteq V$ *and* $E' \subseteq E$. [**?**]

After defining some important concepts, the next step is to extend definition 1.1.1 to define another class of graphs called weighted graphs. It must be noted that all definitions presented so far apply also to weighted graphs.

Definition 1.1.11. *Given a graph G(V,E), a weight function is a function* $f : E \mapsto \Re$ [**?**]. *The real numbers assigned to each edge are called weights.*

Definition 1.1.12. *A weighted graph is a graph G(V,E) with a weight function f* [**?**]. *This is denoted by the triple G(V,E,f) or G.*

According to Bondy and Murty [**?**], weighted graphs occur regularly in applied graph theory. For example, a railway network can be represented by a weighted graph were, the vertices are the set of towns in the railway network, and there are edges between 2 vertices in the graph if, there is a direct route

from one town to another, without visiting other towns in the process. The weight function would then represent the cost of travelling directly from one town to another. In addition to that, the shortest path between 2 towns in the network may be required. It is clear that in order to try and solve such problems, the total weight of a subgraph must first be defined.

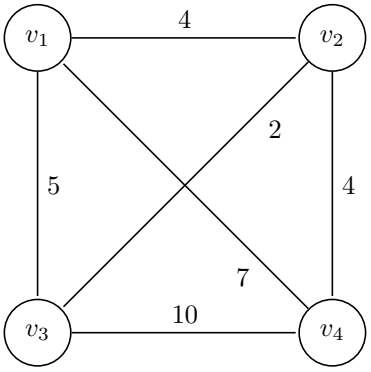
Definition 1.1.13. *Given a weighted graph G(V,E,f), the total weight of any subgraph H(V′,E′,f) of G is:*

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

. [**?**]

It is important to note that by definition 1.1.10, any weighted graph G is a subgraph of itself, therefore, it's weight can be calculated. This is highlighted in Example 1.1.14 below.

Example 1.1.14. Consider the weighted graph *G(V,E,f)* such that, *G(V,E)* is the graph in example 1.1.9 with weight function *f* such that,
$f(\{v_1, v_2\}) = 4$
$f(\{v_1, v_3\}) = 5$
$f(\{v_2, v_3\}) = 2$
$f(\{v_3, v_4\}) = 10$
$f(\{v_2, v_4\}) = 4$
$f(\{v_4, v_1\}) = 7$
Then by definition 1.1.12, the graph below is a weighted graph.



Also according to definition 1.1.10, the above graph is a subgraph of itself. Therefore it's weight can be calculated, were by definition 1.1.13, the weight of G is 32.

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

Definition 1.1.15. *A tree is a connected graph with no cycles.* [**?**]

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

Definition 1.1.16. *H(V′,E′) is a spanning subgraph of G(V,E) if H is a subgraph of G and* $V' = V$. [**?**]

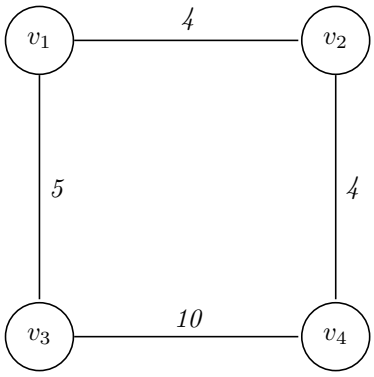
There are many spanning subgraphs, however the ones that are relevant to this thesis are spanning trees and spanning cycles, the latter mostly known as Hamiltonian cycles.

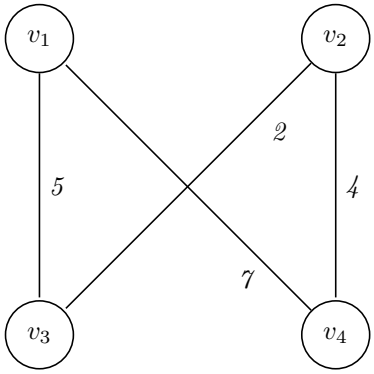
Definition 1.1.17. *A graph H is a spanning tree of G if H is a tree and H is a spanning subgraph of G.* [**?**]

Definition 1.1.18. *Given a graph G, c is a Hamiltonian cycle in G if c is a cycle and c is a spanning subgraph of G. Also, a graph that contains a Hamiltonian cycle is called a Hamiltonian graph.* [**?**]

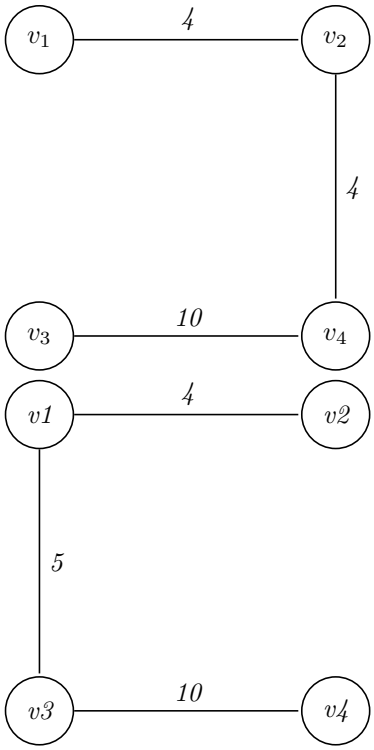
It is worth mentioning that definition 1.1.18 holds because, cycles can be represented by Cycle graphs due to the construction discussed earlier. What follows now is an example that illustrates better definitions 1.1.16, 1.1.17 and 1.1.18.

Example 1.1.19. Let G be the graph in example 1.1.14. Then, according to definition 1.1.16, the 2 graphs below are 2 spanning subgraphs of G because, they contain all the vertices of G and are subgraphs of G .





It must also be said that by definition 1.1.18, the above 2 graphs are Hamiltonian cycles in G because, they are spanning subgraphs of G and are Cycle sub-graphs of G. Since the above graphs are subgraphs of G, by definition 1.1.13, their weight can be calculated by summing up the weights of the edges. Thus, the Hamiltonian cycles above have weight 23 and 18 respectively.
Given the same graph G in example 1.1.14, the 2 graphs below are spanning trees of G of weight 18 and 19 respectively.



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

Having defined Hamiltonian cycles and spanning trees, it is natural to ask

whether there are necessary and sufficient conditions in a graph that guarantee it is Hamiltonian or that it contains a spanning tree as subgraph. In fact, theorem 1.1.20 gives a necessary and sufficient condition for a graph to have a spanning tree.

Theorem 1.1.20. *A graph G has a spanning tree* $\iff$ *it is connected.*

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

Theorem 1.1.20 confirms that for a graph to have a spanning tree, the graph must be connected and vice-versa. Thus, for spanning trees, the necessary and sufficient condition is connectivity. However, the same cannot be said about Hamiltonian cycles because, no necessary and sufficient conditions are known for a graph to be Hamiltonian. In fact, there are sufficient conditions for a graph to be Hamiltonian, however, these conditions are not necessary. According to Guichard [**?**], 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 [**?**], that these conditions are not necessary, because, there are Hamiltonian graphs that have few edges. For example, C_n has only $n - 1$ edges but is Hamiltonian. One such sufficient but not necessary condition for Hamiltonianicity is Ore's Theorem below.

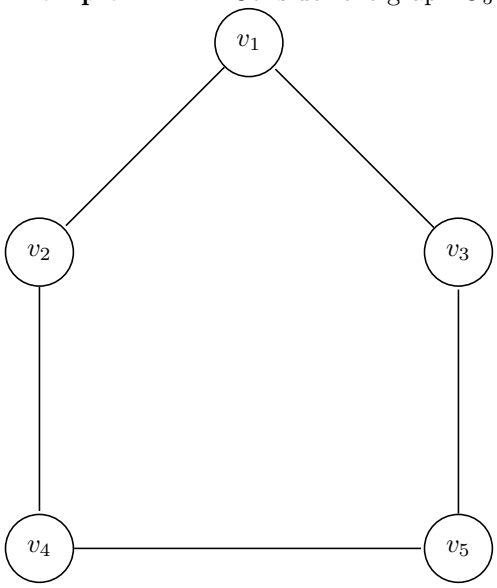
Theorem 1.1.21 (Ore's Theorem)**.** *Let G be a graph on* $n \geq 3$ *vertices such that if v and w are not adjacent in G* $\implies$ $deg(v) + deg(w) \geq n$*, then G is Hamiltonian.*

Proof. Suppose that $G(V, E)$ is a graph satisfying all the conditions in the theorem statement but is not Hamiltonian. Then since G is not Hamiltonian and K_n is Hamiltonian, G must be a subgraph of K_n having fewer edges than K_n. Therefore, add edges to G between non adjacent vertices to obtain a subgraph $H(V', E')$ of K_n such that adding an edge to H would create a subgraph of K_n which is Hamiltonian. Let $u, v \in V'$ be 2 non-adjecent vertices in H. Since by construction G is a subgraph of H, u, v must be non-adjcent in G. Therefore $deg(u) + deg(v) \geq n$ in both G and H. Since adding an edge to H creates a resultant graph that is Hamiltonian, then, adding an edge between u and v creates a Hamiltonian graph. Therefore, in H there must be a path joining u

and v containing all the vertices of H. Let the path be $u = v_1, v_2, ..., v_n = v$. Now suppose $deg(v_1) = \alpha$ in H. Now $\forall i, 1 < i < n$, if there is an edge between u_1 and u_i in H, then there must not be an edge between u_{i-1} and u_n because, $u_1, u_i, u_{i+1}, ..., u_n, u_{i-1}, u_{i-2}, ..., u_1$ would be a Hamiltonian cycle in H, thus H would be Hamiltonian. Therefore, $deg(u_n) \leq n - 1 - \alpha$
$\implies$ $deg(u_1) + deg(u_n) \leq \alpha + n - 1 - \alpha$ in H
$\implies$ $deg(u_1) + deg(u_n) \leq n - 1$ in H
$\implies$ $deg(u_1) + deg(u_n) < n$ in G since G is a subgraph of H.
This contradicts the assumption that $deg(u_1 = u) + deg(u_n = v) \geq n$ in G
Therefore, G must be Hamiltonian. [**?**] □

It is important to note that the above proof uses the fact that K_n is Hamiltonian. This is true because any C_n is always a subgraph of K_n. Therefore any C_n that spans K_n is a subgraph of K_n. Example 1.1.22 below is a counter example to show why Ore's theorem gives a sufficient but not necessary condition.

Example 1.1.22. Consider the graph C_5 *below.*



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

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

1.2 Some Computational Theory

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

Definition 1.2.1. *An optimisation problem is a problem were the goal is to find the best solution out of all possible solutions.* [**?**]

Definition 1.2.1 indicates that when solving an optimisation problem, each solution has an associated value. The aim of the algorithm solving this optimisation problem would then be to find the solution with the best (min/max) value. Another type of problems are decision problems.

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

In [**?**] it is argued that, given an optimisation problem, one can transform this optimisation problem into a decision problem such that, the decision problem is easier to reason about than the optimisation problem. Therefore, if some optimisation problem is easy to solve, the decision problem version is easy to solve aswell [**?**]. 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 [**?**]. Before going into harder computational theory, an example is given to undestand how an optimisation problem can be transformed into a decision problem.

Example 1.2.3. Two optimisation problems are:
1. The shortest path problem
2. The minimum weight spanning tree problem
Given a connected weighted graph G, the shortest path problem is the task of finding a path P between 2 vertices in G such that, the weight of P is the minimum amongst all paths joining these 2 vertices in G [**?**]. 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 [**?**]. The shortest path problem is an optimsation problem because, from all the possible paths in the graph, the best path is chosen. In this case the best path is the path with minimum weight. Similarly, using the same reasoning, the minimum weight spanning tree problem is an optimisation problem
The decision problems related to these optimisation problems are the following:
1. Given a connected weighted graph G, 2 vertices u and v, and an integer k, does a path exist from u to v in G of weight at most k?
2. Given a connected weighted graph G, and an integer k , does G contain a spanning tree of weight at most k?
The above 2 problems are decision problems because the answer to these problems is either a yes or a no.

Reasoning about computational problems is usally done by looking at properties belonging to the algorithms that solve these problems. One property that is sometimes looked at is called the running time of an algorithm. This is defined in definition 1.2.4 below.

Definition 1.2.4. *The running time of an algorithm is the number of steps taken by the algorithm to execute, given a particular input. The running time is represented by the function* $T : \mathbb{N} \mapsto \mathbb{N}$ *were the domain represents the size of the input, and the co domain represents the amount of steps taken by the algorithm to execute given a particular input size.* [**?**], [**?**]

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

Example 1.2.5. Consider an algorithm that performs the addition of 2 n-bit integers. Then if the addition of 2 bits takes a computational steps, the addition of 2 $n-$bit integers will take $a \times n$ steps. Therefore the running time of this algorithm for an input of size n is $T(n) = a \times n$. [**?**], [**?**]

Expressing the running time in this way seems pretty easy to understand, however it has problems. Typically, the running time of an algorithm is fixed for any given input, however, the running time function is not defined on the actual inputs but on the input sizes. This means that since the same input size might represent different inputs, the running time may be different for the same input size n, as this depends on which input is given. Therefore, different running time functions could represent the same algorithm, depending on which input of size n is given. To solve this problem, the worst-case running time is usually used when reasoning about the running time of an algorithm, implicitly choosing the running time function that gives the worst-case running time. The worst-case running time is the longest running time for any input of size n. This would solve the problem because, the worst-case running time of an algorithm is an upper bound to all other running times for any input of size n, therefore, this assures that the algorithm will never get a longer running time for any input of size n. This avoids making guesses about the running times and avoids hoping that it would never get worse. [**?**], [**?**]

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 the running time of a particular algorithm would need a different amount of computational steps on different computers. For example the addition of 2 bits as seen in example 1.2.5, might take a different number of computational steps on different computers resulting into different running time functions. To get around this problem, the algorithms are analyzed asymptotically. This means making the input size increase without bound so that in the running time function, only the order of growth of the function is relevant. This can only be done because as the input size increases, the lower-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 lower order terms and multiplicative constants of the function can be ignored. [**?**], [**?**]

Therefore in a running time function, the highest order term is the most relevant term because it gives the order of growth of the function, i.e. how much will the running time increase as the input size grows without bound. For example, given the running time function $T(n) = an^2 + bn + c$, what remains in the asymptotic analysis of this running time function is n^2. It is important to note that the order of growth of different running time functions for the same algorithm will not be different. This happens because the order of growth of a function is dependent on the input size. For example, the n-bit addition algorithm in example 1.2.5 may have a running time function $T(n) = a \times n$ on one computer, and a running time function $T(n) = b \times n$ on another computer. The running time functions may differ only in the multiplicative constants because, these depend on the computer's architecture. However, the order of growth is n^2 for both. [**?**], [**?**]

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

Definition 1.2.6. *Let $f : \mathbb{N} \mapsto \mathbb{N}$ and $g : \mathbb{N} \mapsto \mathbb{N}$ be 2 monotonic functions. Then f(n) = O(g(n)) if $\exists$ $c > 0$ and $n_0 > 0$ such that $f(n) \leq cg(n)$ $\forall$ $n \geq n_0$.* [**?**]

Definition 1.2.6 implicitly states that when writing $f(n) = O(g(n))$, $g(n)$ is an upperbound to $f(n)$ as the input size n grows without bound [**?**]. 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 $g(n)$. The following is a continuation to example 1.2.4.

Example 1.2.7. By example 1.2.4, the algorithm that performs 2 n-bit addition has a running time function *T(n)= an.* Therefore, by definition 1.2.6, *T(n) = O(n)* because, since a is a natural number $\exists$ $z \geq a$ such that, $an \leq zn$. Therefore at each step, the order of growth of *T(n)* is linear with respect to the input size n.

It is important to note that if $T(n) = O(g(n))$ then $T(n) = O(h(n))$ if $h(n)$ grows faster than $g(n)$ as $n \to \infty$. For example, in example 1.2.6 $T(n) = O(n^2)$ aswell since, n^2 is an upperbound to n as $n \to \infty$. However, unless otherwise stated it can be assumed that whenever $T(n) = O(g(n))$, $g(n)$ is a tight asymptotic bound to $T(n)$. This means that, $T(n) = cg(n)$, were c is a constant factor [**?**].

More definitions will now be introduced were the aim is to represent discussed concepts without having to write a lot of text.

Definition 1.2.8. *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)).* [**?**]

Definition 1.2.8 does not specify that the running time function of an algorithm must be the worst case running time function. However, recall that when discussing running time functions it was said that it can be assumed that the worst-case running time function is taken. Therefore, it can be assumed that the worst-case running time function is taken when saying that an algorithm has time complexity $O(g(n))$. 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 time $O(g(n))$ this means the problem can be solved by an algorithm that has a worst-case time complexity $O(g(n))$

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 [**?**] and [**?**].
In table 1, the time-complexities are ordered in ascending order of growth [**?**].

Table 1: Time Complexities and Terminology Used.

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

This means that for example, constant time algorithms have a smaller order of growth than logarithmic time algorithms. It must be noted that the higher the order or growth, the more steps the algorithm needs to execute and therefore, the less efficient it is. In general, problems that can be solved at worst by polynomial time algorithms are tractable problems [**?**]. On the other hand, problems that require superpolynomial (exponential/factorial) time algorithms to be solved are intractable problems [**?**]. In general tractable problems are considered to be easy to solve, whereas intractable problems are hard to solve.

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

Example 1.2.9. In example 1.2.7, *T(n) = O(n).* Therefore according to definition 1.2.8, the n-bit addition algorithm has a time complexity of *O(n).* Therefore by table 1, the n-addition algorithm has a linear time complexity, or simply is a linear-time algorithm. As a result, this algorithm is tractable.

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 by a polynomial time algorithm and others which cannot. This seems to indicate that there are several classes of problems. One of these classes is the class P defined below. [**?**]

Definition 1.2.10. *The class P consists of all the problems that can can be solved in polynomial time.* [**?**]

Therefore according to definition 1.2.10, the class P consists of tractable problems. One problem in this class is the $n-$bit addition problem defined in example 1.2.5. The $n-$ bit addition problem is in the class P because, it can be solved by an algorithm having a worst-case linear time complexity. Apart from the class P, there is also the class NP. The following definition helps in defining the class NP.

Definition 1.2.11. *A problem can be verified in polynomial time if given a solution to the problem, the solution can be checked by an algorithm if it is correct or not in polynomial time.* [**?**]

The following example explains the concept of polynomial time verification.

Example 1.2.12. To explain polynomial time verification, 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 [**?**]. A solution to the Hamiltonian cycle problem is a cycle $S = v_1$, ..., v_n of $|V| = n$ vertices. Therefore to confirm that the Hamiltonian cycle problem is verifiable in polynomial time, S must be checked if it is a Hamiltonian cycle or not in polynomial time. 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 takes at worst *O(n)* time because it must check all edges in the cycle. Hence the checker algorithm takes polynomial time to execute. As a result, the Hamiltonian cycle problem can be verified in polynomial time.

Now the class NP can be defined.

Definition 1.2.13. *The class NP consists of all decision problems that can be verified in polynomial time.* [**?**]

It is important to note that according to definition 1.2.13, for a problem A to be in the class NP, A must be a decision problem. This will not limit the discussion because as discussed before, every optimization problem can be transformed into a related decision problem by specifying a bound on the solution to be optimized (see example 1.2.3). Reasoning about decision problems is easier because the related decision problem is not harder than the original optimisation problem. As a result, one can prove that an optimisation problem

cannot be solved in polynomial time by showing that the related decision problem cannot be solved in polynomial time. [**?**]

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

Definition 1.2.14. *Suppose that A and B are two decision problems. A reduction algorithm is an algorithm that transforms any instance (input) α of A into an instance β of B with the following properties:*

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

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

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

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

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 P 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 P can be used to transform any instance α of A into an instance β of B were again α can be solved in polynomial time as discussed in the previous paragraph, contradicting 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 computational time) 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.

Definition 1.2.15. *The decision problem P is in NPC if it is in NP and it is as hard as any problem in NP.* [**?**]

In other words, definition 1.2.15 states that for a problem P to be in NPC, then P must be in NP and that any problem in NP can be transformed using a reduction algorithm into P. ..describe how to proof that a problem is in npc using geeks source, P=NP problem using other papers in cormen bookand ready

2 The Travelling Salesman Problem

2.1 Heuristics

2.2 The Ant Colony Algorithm

3 Experimental Data

4 Conclusion