School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

Final Report

Implementation and Performance comparison of Comparability graph recognition algorithms

Joseph Halfpenny

Submitted in accordance with the requirements for the degree of
BSc Computer Science

2022/2023

COMP3931 Individual Project

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	PDF file	Uploaded to Minerva (DD/MM/YY)
Link to online code repository	URL	Sent to supervisor and assessor (DD/MM/YY)
User manuals	PDF file	Sent to supervisor and assessor (DD/MM/YY)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) Joseph Halfpenny

Summary

The focus of the problem will be on two specific algorithms that identify whether a graph is a comparability graph or not, and the orientations of the edges that are produced when the algorithm is successful. The desired outcome is to have a set of tests in place that will help determine the performances for given graph types. The analysis of the results should allow for a recommendation as to which algorithm is best suited for a given graph type.

Acknowledgements

I would like to thank my supervisor, Dr Haiko Muller for meeting throughout the progression of my project and for his support during times of doubt.

I would like to thank my assessor, Dr. Sebastian Ordyniak for meeting with me to discuss the progress of my project.

Contents

Chapter 1

Introduction

1.1 Motivation and Context

This project was not a first choice but rather an undesirable project that was offered to me as a last resort. I had been told that this was a difficult project and that previous students who had been given the same project had not completed it to its entirety. I believe this project to be a body of work that was outside of my capability, but I have worked hard in order to try and achieve the project objectives.

1.2 Project Background

The researchers that deduced the algorithms used in this report have only formalised them theoretically, an implementation of these ideas have not yet been developed. The theoretical design and proof of these algorithms are sound but in practise the computer replication of the theory may vary due to limitations of machines and libraries.

1.3 Project Aim

The aim of this project is to develop the functionality of the two algorithms and to deduce and inform as to which of the two will perform best with a given graph. The analysis of the performances will be documented and provide evidence as to which is preferable for real world contexts.

1.4 Objectives

The objectives for this project are:

- The implementation of the two algorithms.
- Attain the time complexities stated in the research papers or explain the reasons for not achieving them.
- Determine the most efficient algorithm to use when executed on different graph types.

1.5 Deliverables

Upon successful completion of this project, there will be the following deliverables:

- Source code of the algorithms, which will be stored in a version-controlled repository.
- The project report.
- A directory of the performance results.

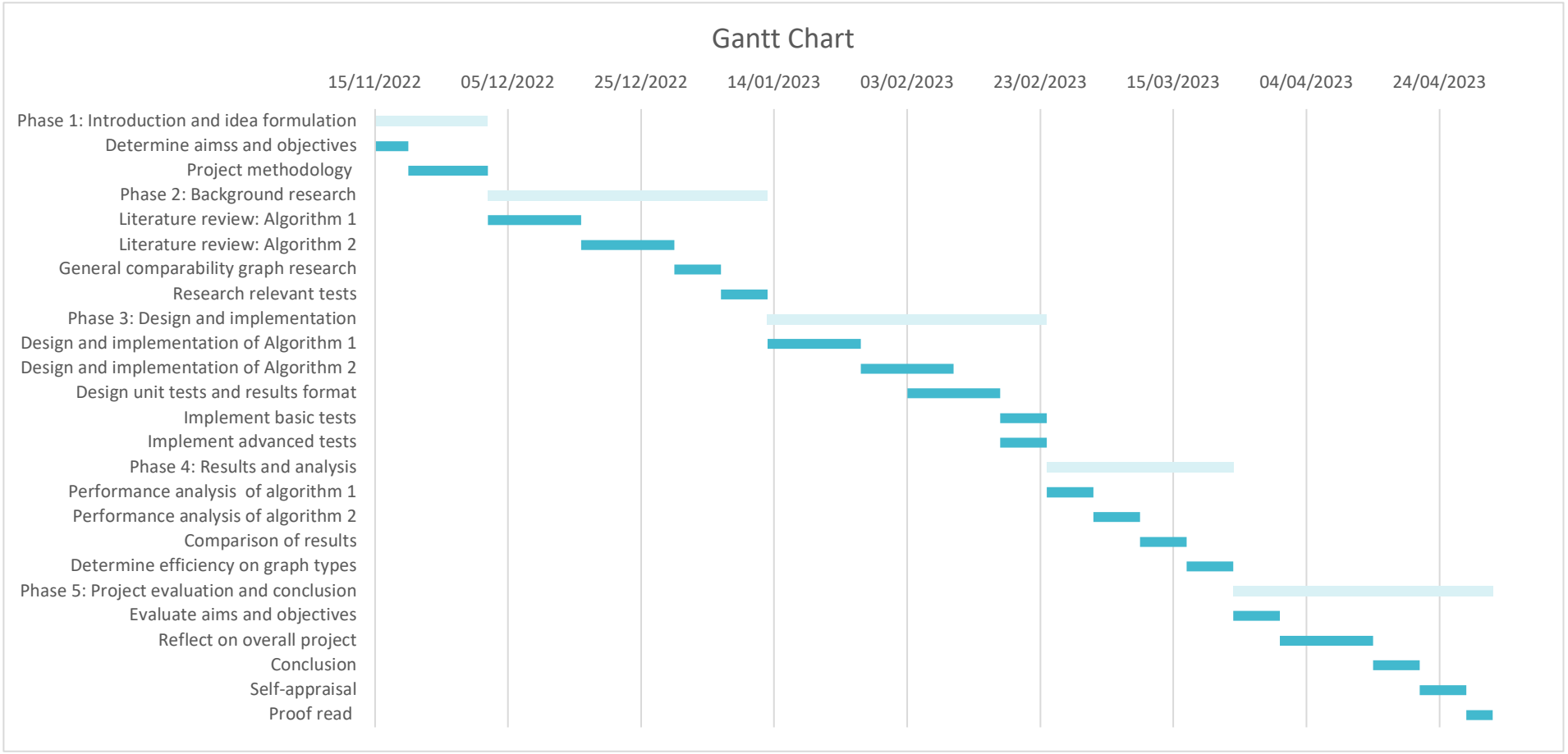
1.6 Project Plan

1.6.1 Project Methodology

The project follows a structure that is an iterative process of improvement, this is to ensure that a best possible versions of the algorithms are delivered. Deadline constraints will need to force incomplete iterations so that a more complete and comprehensive body of work can be delivered. The project will follow this methodology.

- Background research: This section includes the research necessary into the given papers by the Golumbic (1980) [1] and Gallai, Tibor (1967) [2], to understand the basic ideas and structures of each algorithm. A strong understanding of the algorithms will ensure that all future sections of the project can be discussed with clarity and a sound comprehension. Research into the properties of comparability graphs will be necessary to ensure relevant tests can be run and for the results produced to be analysed properly.
- Project Design: The design of the project will be stemmed from all the concepts learnt from the research section. The iterative design choices will be discussed with relation to the concepts needed in order to apply the theory into an implementation for both algorithms. The architecture of the testing phase will be discussed leading to the later section.
- Results and analysis: The results from the test will be analysed through the observation of the change in running time when varying graph input properties, this will give indication as to what graph types impact the performance.
- Evaluation and conclusion: Review the overall consensus of the project and the benefits and downfalls of the algorithms. Discuss any reflections on the development of aims and the quantifiable objectives completeness. Examine the challenges faced and the efforts made to overcome them. Conclude the overarching outcome from this project and a personal reflection of its progress.

1.6.2 Gantt Chart



Chapter 2

Background Research

2.1 Definition of comparability graphs

Comparability graphs are also known as transitively orientable graphs, interval graphs, and ordered graphs. For the purposes of this report we will use the use the terms comparability graph and transitively orientable graphs and follow the definition given by Kelly, David (1985) [6], the definitionprovided is as follows:

"The comparability graph of an ordered set P is the graph with vertex set X and edges xy whenever $x < y$ or $x > y$ in P."

We shall reiterate this definition in a more precise notation for further clarity. Let $G = (V, E)$ be an undirected graph. Let P be a non-reflexive partial ordering defined on a set E. If there exists a graph $G = (V, P)$ for which P follows this partial ordering, then G is known as a comparability graph.

The idea of a non-reflexive partial order is determined by the intersection of three relations. To illustrate these relations, let R be a relation on a set A. We say that R is a non-reflexive partial order if and only if R is non-reflexive, anti-symmetric and transitive, where:

The relation R on a set A is non-reflexive if and only if, $\forall x \in A$, $(x, x) \notin R$
i.e. no element is related to itself.
The relation R on a set A is anti-symmetric if and only if,$\forall (a, b) \in R$, $a \neq b$
and $(b, a) \notin R$
The relation R on a set A is transitive if and only if, $\forall (x, y) \in R$ and
$(y, z) \in R \Rightarrow (x, z) \in R$, for all $x, y, z \in A$

Together these relations define the properties of a comparability graph. The graph $G = (V, F)$ is said to have a transitive orientation if it conforms to these relations. The reversed transitive orientation of a set F is also a comparability graph.

2.2 Gamma rule

2.2.1 Papers definition of a comparability graph

In Golumbic's paper "Algorithmic graph theory and perfect graphs"[1] comparability graphs are defined as follows,

"An undirected graph $G = (V, E)$ is a comparability graph if there exists an orientation (V, F) of G satisfying:

$$F \cap F^{-}1 = \emptyset,\ F + F^{-}1 = E,\ F^2 \subseteq F\ "$$

where, $F \cap F^{-1} = \emptyset$ is the intersection of the oriented edges F with the set of edges of reversed orientation F^{-1}, this intersection must be empty.

$F + F^{-1} = E$ is the combination of the set of oriented edges F, with the set of edges F^{-1} with reversed orientation, these sets must result in the set of edges E with no orientation, due to the opposing orientations *cancelling* one another out.
$F^2 = \{ac|ab, bc,\ F \text{ for some vertex } b\}$, F^2 is the set of edges that are produced under the transitive closure. F^2 must be a proper subset of the oriented edges of F. Meaning not all edges E must be closed under the transitive property but those that can be, must be.

Other relations and theorems are required to form the algorithm used for determining whether a graph is a comparability graph or not, these include, the force direction rule, the partition of edges into implication classes, the G decomposition of a graph and finally the TRO theorem. These together give us a method for recognising comparability and returning their transitive orientation.

2.2.2 Force direction rule and implication class:

Given an undirected graph $G = (V, E)$ with the applications of the force direction rule the choice of orienting an arbitrary edge will force the direction of neighbouring edges. This forced direction continues to subsequent adjacent edges forming a set of edges known as an implication class. In order to retrieve these implication classes a formal definition of the force direction rule is needed. This relation is more strictly defined in Golumbic's (1980) paper as:

$$\text{"}ab\ \Gamma\ a'b' \text{ iff } \{ \text{ either } a = a' \text{ and } bb' \notin E \text{ or } b = b' \text{ and } aa' \notin E\}\text{"}$$

It must be understood that the relation Γ on E is not reflexive, meaning the direction of one orientation along an edge cannot force the orientation in the opposite direction along that same edge: $ab\ !\Gamma\ ba$, this would be a imply contradiction in orientation. When the arbitrary choice of the orientation of one edge implies the orientation of another, we say the edge is forced. We can say that ab forces ac whenever $ab\ \Gamma\ ac$.
Repeating this binary relation causes other edges to be subsequently forced by ab. From this repetition of the relation we can create implication classes. Two edges are said to be in the same implication class if and only if:

$$ab = a_0b_0\ \Gamma\ a_1b_1\ \Gamma\ \ldots\ \Gamma\ a_kb_k = cd \text{ with } k \leq 0.$$

This causes the set of edges E to be partitioned into subsets, these subsets are known as implication classes.
We will define the set containing all implication classes as $P(G)$
$P(G) = \{\hat{A}|A \in P(G)\}$, where $\hat{A} = A \cap A^{-1}$ is the symmetric closure of A. Meaning that all reversed orientations of the edges in the implication classes are included in $P(G)$.

For the undirected graph G in Figure 2.1, we can define the vertex set as $V = \{a, b, c, d, e\}$ and the edge set as $E = \{ab, ac, cd, de, ef, bf, ce, eb\}$. By repeatedly applying the gamma relation, we obtain the implication classes.

$\vec{ed} \Rightarrow \vec{ef}$	$A_1 = \{ed, ef\}$	$A_1^{-1} = \{de, fe\}$
$\vec{ce} \Rightarrow \vec{be}$	$A_2 = \{ce, be\}$	$A_2^{-1} = \{ec, eb\}$
$\vec{fb} \Rightarrow \vec{ab} \Rightarrow \vec{ac} \Rightarrow \vec{dc}$	$A_3 = \{fb, ab, ac, dc\}$	$A_3^{-1} = \{bf, ba, ca, cd\}$

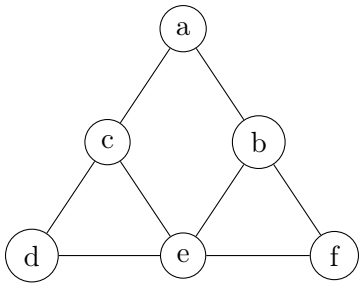
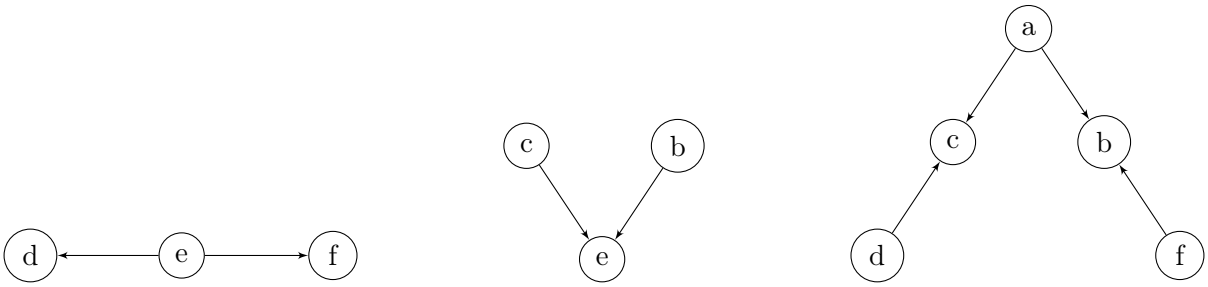


Figure 2.1



(a) (b) (c)

$\hat{A}_1 = \{ed, ef, de, fe\}$
$\hat{A}_2 = \{ce, be, ec, eb\}$
$\hat{A}_3 = \{fb, ab, ac, dc, bf, ba, ca, cd\}$

Figure 2.1 has been split into its corresponding implication classes (a), (b), (c), for each implication the arbitrary orientation of one edge is shown to force the next.

2.2.3 G-decomposition algorithm

The G-decomposition algorithm is a method for partitioning the set of undirected edges into their corresponding symmetric closure of the implication classes. We will be using the algorithm defined in Golumbic's (1980) paper [5]. In this algorithm B_i represents an implication class of $\hat{B}_i$, where $\hat{B}_i = A \cap A^{-1}$.

Algorithm 1: Decomposition Algorithm

Initially, let $i = 1$ and $E_1 = E$

Step(1): Arbitrarily pick an edge $e_i = x_i\ y_i \in E$
Step(2): Enumerate the implication class B_i of E_i containing $x_i\ y_i$.
Step (3): Define $E_{i+1} = E_i - \hat{B}_i$
Step (4): If $E_{i+1} = \emptyset$ then let $k = i$ and Stop; otherwise, increase i by 1 and go back to Step(1).

Every iteration reassigns the value of the E_i to the previous edge set E_{i-1} with the edges of B_{i-1} taken away from it, creating a sub-graph of the initial graph. The initial graph is split into its G-decomposition classes until all edges have been removed from the edge set E.

2.2.4 TRO Algorithm

In order to produce an algorithm for defining the transitive oriented set of edges of a graph G, another theorem must be introduced, the TRO theorem is also from Golumbi's (1980) paper [5]. The theorem states that the following statements are equivalent:

" i. $G = (V, E)$ is a comparability graph

ii. $A \cap A^{-1} = \emptyset$ for all implication classes A of E

iii. $B_i \cap B_i^{-1} = \emptyset$ for $i = 1, ..., k$, k represents the number of implication classes in the set E

iv. Every circuit of edge $v_1v_2, \ldots v_qv_1 \in E$ such that $v_{(q-1)}v_1, v_qv_2, v_{(i-1)}v_{(i+1)} \notin E$ for $(i = 2, \ldots, q-1)$ has even length. "

Through the combination of the G decomposition and the theorems shown in the TRO theorem, the following algorithm was developed by Golumbic [5]: The algorithm follows the

Algorithm 2: TRO Algorithm

Input: An undirected graph $G = (V, E)$.
Output: A transitive orientation F of edges of G, or a message that G is not a comparability graph

begin
Initialize: $i \leftarrow 1$; $E_i \leftarrow E$; $F \leftarrow \emptyset$
1. Arbitrarily pick an edge $x_i\ y_i \in E_i$
2. Enumerate the implication class B_i of E_i containing $x_i\ y_i$;
if $B_i \cap B_i^{-1} = \emptyset$ **then**
add B_i to F;
else
print "G is not a comparability graph";
STOP
end
3. Define $E_{i+1} \leftarrow E_i - \hat{B}_i$;
4. **if** $E_{i+1} = \emptyset$ **then**
$k \leftarrow i$; **output** F;
STOP;
else
$i \leftarrow i + 1$;
go to 1;
end
end

same process as the G-decomposition but incorporates a check of the intersection of the implication classes with their reversed orientation. This check ensures that there is no conflicting orientations being added to the set of directed edges F, if there is then the graph is not comparable. The rest of the algorithm follows the G decomposition principles previously covered.

If we apply the algorithm to the graph G shown in Figure 2.1 he following results are produced:

$E_1 = \{ab, ac, cd, de, ef, bf, ce, eb\}$

$B_1 = \{ed, ef\}$, $F = \{ed, ef\}$

$E_2 = \{ab, ac, cd, bf, ce, eb\}$

$B_2 = \{ce, be\}$, $F = \{ed, ef, ce, be\}$

$E_3 = \{ab, ac, cd, bf\}$

$B_3 = \{fb, ab, ac, dc\}$, $F = \{ed, ef, ce, be, fb, ab, ac, dc\}$
F is the set of transitively oriented edges, hence G is a comparbility graph.

2.3 Knotting graphs

2.3.1 Papers definition of a comparability graph

The general definition shown in T. Gallai's paper "Transitiv orientierbare Graphen" (1967) [2] is as follows: "A graph will be called transitively orientable if its edges can be directed in such a way that whenever ab and bc are arcs the ac is also an arc."
This definition mainly refers to the transitive nature of comparability graphs but also later refers to Ghouali-Houri (1967) paper [3] and Gilmore and Hoffman (1964) paper [4], stating that they "have given necessary and sufficient conditions for a graph to be transitively orientable"
A more precise definition is given by Gallai (1967) later on: "if G is transitively orientable and $ab, ac \in G$ and $bc \in G^{-1}$, then in any transitive orientation of G either both arcs ab,ac exist or both arcs ba, ca exist, i.e. the edges ab, ac must be directed the same way with respect to their common endpoint a." The definition covers the ideas with regards to the orientation of the edges, more specifically the need for transitivity and the effect of reversing the direction of particular edges will subsequently result in the reversing of others and that the orientation of edges is dependent on their common endpoint.

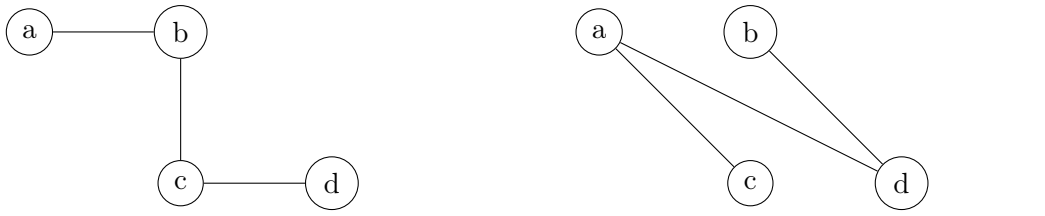
2.3.2 Relations needed to produce a knotting graph

For the production of a knotting graph two fundamental relations are required, the $\hat{}$ relation and the $\sim$ relation, which are both defined in T. Gallai's (1967) paper [2]. We will firstly discuss the $\hat{}$ relation which is defined as follows:

" The $\hat{}$ relation is reflexive and symmetric, and its transitive closure induces a partition of the edge set of G. The edges of this partition will be called edge classes of G. Two edge classes are in the same edge class if and only if there exists a sequence of edges $x_1y_1, \ldots, x_ny_n$ such that, $ab = x_1y_1, cd = x_ny_n$, and $x_iy_i \hat{} x_{i+1}y_{i+1}$ for $i = 1, \ldots, n-1$. If G is transitively orientable then, the choice of one direction for one given edge in an "edge class" E implies the direction for all the other edges in E. "
A more explicit definition suggests that two edges ab and cd are in the same edge class if and only if, there is a sequence of edges $x_1y_1, \ldots, x_ny_n$ such that, $ab = x_1y_1$, $cd = x_ny_n$, and $x_iy_i \hat{} x_{i+1}y_{i+1}$ for $i = 1, \ldots, n-1$. In order to clarify this relation an illustrated example using graph G from Figure 2.3 (a) is shown below.

$ab = x_1y_1 \qquad cd = x_3y_3$
$bc = x_2y_2 \qquad x_1y_1 \hat{} x_2y_2 \hat{} x_3y_3 = ab \hat{} bc \hat{} cd$

This relation with regards to transitive orientability suggests that for each edge class the choice of one edge's orientation implies the orientations for every other edge in that edge class. Each edge class has two possible transitive orientations. In order to apply transitive orientability to these classes we must introduce another relation.



(a) Undirected graph G (b) G^{-1}: complement graph of G

Figure 2.3

The $\sim$ relation notation on the edges $ab, ac \in G$ is written as:

$$ab \sim ac$$

this relation holds true if and only if, either $b = c$, or there exits a path W, where $W \in G^{-1}$ and $W = (x_0, ..., x_k)$ with $k \leq 1$, $x_0 = b$, $x_k = c$ and $ax_i \in G$, for $(i = 1, ..., k-1)$.
To clarify this relation we will reiterate the meaning of the notations. G^{-1} is the complement graph of G, W is the path from b to c in G^{-1}, k is the number of edges along the path W from b to c along the path W and ax_i is the path from a to x_i where $1 \leq i \leq k-1$.
When this relation is applied it results in edges being knotted at one common vertex, in this case vertex a.We will continue with the previous example of Figure 2.3, here Figure 2.3 (b) shows the complement graph G^{-1} of G. Let $E \in G$ and $E^{-1} \in G^{-1}$. $E = \{ab, bc, cd\}$, $E^{-1} = \{ac, ad, bd\}$. Let the $\sim$ relation be applied on E:

$ba \sim bc \Rightarrow b = b$, $W = x_0x_k$ where $x_k = c$, $x_0 = a$, $k = 1$ and $ac \in E^{-1}$.

The path $bx_i = bc$, $bc \in G$ with path length as 1, therefore ab and bc are knotted at b in G.

$cb \sim cd \Rightarrow c = c$, $W = x_0x_k$ where $x_k = c$, $x_0 = a$, $k = 1$ and $ac \in E^{-1}$.

The path $bx_i = bc$, $bc \in G$ with path length as 1, therefore bc and cd are knotted at c in G.

Hence, the edges ba and bc are knotted at b in G, and the edges cb and cd are knotted at c in G.

2.3.3 Application of the relations

It is shown by cases where $ab \sim ac$, $b \neq c$ and $ax_i \hat{} ax_{i+1}$ for $(i = 0, ..., k-1)$, that the knotted edges are in the same edge class, showing that $ab \hat{} ac$ implies $ab \sim ac$.
The $\sim$ relation with regards to transitive orientability declares that any two edges knotted at the same vertex must be oriented in the same direction with respect to their common endpoint vertex. If there are incoming edges oriented towards the common vertex, then this vertex is referred to as a sink, on the other hand if there are outgoing edges away from the common vertex, the vertex is referred to as a source. Applying this rule repeatedly for all edges of the graph allows for the transitive orientability of a graph to be determined.
The previous example is continued:

Defining orientation for $ba \sim bc$:

The arbitrary orientation of a to b implies that the edge bc must be oriented in the same direction with respect to the common vertex b. Therefore, $\vec{cb}$ is the implied orientation.

Defining orientation for $cb \sim cd$:

cb has already been oriented which determines that c is a source vertex. Implying that $\vec{cd}$ the orientation $\vec{cd}$ is the orientation of the edge.

The transitive orientability has been determined, the set of oriented edges $F = \{\vec{ab}, \vec{cb}, \vec{cd}\}$, meaning the graph G is a comparability graph.

Chapter 3

Design and implementation

Now that the background research has been covered, the design and implementation of the ideas explained shall now be discussed.

3.1 Design Overview

The goal of the system is to extract comparable data from the algorithms being tested, in order to do this the objectives must be made clear. The first is to implement the two fore-mentioned algorithms and the second is to implement a set of tests to identify correctness and to yield meaningful data for later analysis.

The overall architecture of the system will take as input an undirected graph $G = (V, E)$ and output the directed graph $G = (V, F)$ where F is the set of transitive orientable edges. Tests will be run on the output graph, checking for partial correctness by evaluating if the set of output edges produced are non-reflexive and antisymmetric. Measurements of the number of vertices, number of edges and the running time will be stored in a csv file for further processing. Different graph types will be generated and used as input graphs so that a variety of data can be gathered, which will in turn demonstrate the effectiveness of the algorithms on certain graph types.

The software being used is JGraphT which stores the edges of graphs as a set with the two vertices corresponding to the source vertex and target vertex. A new class to represent an edge is needed, called OrientableEdge, for which its use in the two algorithms will later be explained. This class extends the DefaultEdge class provided by the JGraphtT library and therefore bares a lot of the same functionality, with some additional features shown in tables 3.1 and 3.2 below.

Attributes	Type	Description
isOritented	private boolean	Default set to false, this is the value for whether an edge is oriented or not.
isForwardOriented	private boolean	Default set to false, this is the value for whether an edge is oriented in the forward direction.
isBackwardOriented	private boolean	Default set to false, this is the value for whether an edge is oriented in the backward direction.

Table 3.1: Attributes table.

Method	Parameters	Functionality
setForwardOrientation	DefaultDirectedGraph<*String, OrientableEdge*> graph, OrientableEdge edge	Calling this method will add the specified edge to the specified graph and the isOrientation and isForwardOrientation attributes are set to true.
setBackwardOrientaiton	DefaultDirectedGraph <*String, OrientableEdge*> graph, OrientableEdge edge	Calling this method will add the specified edge to the specified graph and the isOrientation and isForwardOrientation attributes are set to true.

Table 3.2: Method table.

3.2 Gamma Design and Implementation

3.2.1 Gamma Overview

The implementation carried out in this paper is a variation of the algorithm developed by Golumbic. The failed attempts to correctly separate the implication classes resulted in an implementation that applies the concepts of the force direction rule. "TRO Algorithm" shown in Golumbic's paper [5] was abandoned in order to obtain some partial results. The adaptation is correct for some graph cases but fails to identify comparability graphs in most. It therefore fails at distinguishing whether a given input graph is a comparability graphs or not, and should strictly not be used outside of the scope of this project

3.2.2 Basic rule implementation

The basic force direction rule was the core idea used in the development of this algorithm. The algorithm takes as input an undirected graph and returns the directed graph with transitive orientation if successful, else it returns an incorrect set which is later checked in the testing phase. The pseudocode provided below illustrate the steps taken by the force direction function.

Algorithm 1: Force Direction

Input: An undirected graph $G = (V_1, E)$.
Output: A directed graph $DG = (V_2, F)$, where F is the set of edges with transitive orientation.
begin
  Initialize: $V_2 \leftarrow V_1$;
  Arbitrarily pick an edge $e_0 \in E$, $e_0.setForwardDirection(DG, e_0)$;
  for $e_i \in E$ **do**
    if e_i *is oriented* **then**
      for $e_j \in E$ **do**
        if $e_i = e_j$ **then**
          continue;
        end
        if $e_i.SourceVertex = e_j.SourceVertex$ **and** $e_i.TargetVertex$ *to* $e_j.TargetVertex \notin E$ **then**
          if $e_i.isOrientedForward()$ **then**
            $e_j.setForwardOrientation(DG, e_j)$;
          else
            $e_j.setBackwardOrientation(DG, e_j)$;
          end
        end
        if $e_i.TargetVertex = e_j.TargetVertex$ **and** $e_i.SourceVertex$ *to* $e_j.SourceVertex \notin E$ **then**
          if $e_i.isOrientedForward()$ **then**
            $e_j.setForwardOrientation(DG, e_j)$;
          else
            $e_j.setBackwardOrientation(DG, e_j)$;
          end
        end
        if $e_i.SourceVertex = e_j.TargetVertex$ **and** $e_i.TargetVertex$ *to* $e_j.SourceVertex \notin E$ **then**
          if $e_i.isOrientedForward()$ **then**
            $e_j.setBackwardOrientation(DG, e_j)$;
          else
            $e_j.setForwardOrientation(DG, e_j)$;
          end
        end
        if $e_i.TargetVertex = e_j.SourceVertex$ **and** $e_i.SourceVertex$ *to* $e_j.TargetVertex \notin E$ **then**
          if $e_i.isOrientedForward()$ **then**
            $e_j.setForwardOrientation(DG, e_j)$;
          else
            $e_j.setBackwardOrientation(DG, e_j)$;
          end
        end
      end
    end
    while $!e_i.isOriented()$ **do**
      $e_i.setForwardOrientation(DG, e_i)$;
    end
  end
  return DG;
end

The initial for loop is iterating over the set of oriented edges which is, at first, the one arbitrary edge with forward orientation. This arbitrary edge causes other subsequent edges to be added to the set of oriented edges by forcing their direction. If no edge is forced due to the current oriented edge being iterated over, then another arbitrary edge is given a forward orientation. The nature of the edge sets indicates that we must check all four possible positions of the vertices within the two edges being compared. If we have $ab\Gamma a'b'$ then the following possibilities must be checked:

- $a == a'$ and $bb' \in E$, $\quad b == b'$ and $aa' \in E$.
- $a == b'$ and $ba' \in E$., $\quad b == a'$ and $ab' \in E$.

Once all edges have been oriented and added accordingly to the directed graph DG, the outer loop ends, and the directed graph is returned.
The time complexity of the algorithm is $(|E|^2)$ where E represents the edge set of the input graph. The inner loop and outer loop only apply to the edge set, this is where the majority of the computation is executed. Creating graph objects and adding vertices is of time complexity O(1) as they are represented as adjacency lists.

3.2.3 Improvements

The inability to implement the functionality for the separation of implication classes hindered the continuation of developing the TRO algorithm, which lead to the variation algorithm. The main challenge faced when trying to add edges to their correct implication class was finding a technique for relating all edges contained in the Γ-chains. Many previous efforts led to implication classes being incomplete. In spite of this, the variation algorithm functionality for recognising comparability graphs is partial, allowing for some test results to be collected. The main issue with the current version is that when an oriented edge has reached the end of the inner loop, the next unoriented edge is given a forward orientation without considering the direction of its surrounding edges; this leads to the algorithm failing in some cases. The time complexity in the Golumbic (2004) paper was not achieved. There is no mention in the paper of the time complexity with regards to the partitioning of the edges into their implication classes.

3.3 Knotting Design and Implementation

3.3.1 Knotting Overview

There is no mention of a complete algorithm for finding the transitive orientability of an undirected graph in Gallai's (1967) paper, therefore the implementation in this paper is a simple application of the relations previously explained in section 2.3. Little success was achieved for identifying comparability graphs, in most cases the algorithm fails to indicate the correct transitive orientability. Further understanding of the paper was required in order to obtain a complete and robust algorithm for detecting comparability graphs, due to time constraints and a lack of secondary source material a finalized and completely functional implementation was unsuccessful.

3.3.2 Basic rule Dseign and Implementation

The algorithm core functionality follows the concepts of the $\sim$ relation, it takes as input an undirected graph and returns a directed graph with its set of edges being transitively oriented. The orientations produced are sporadically faulty, leading to untrustworthy results.
Classes from the JGraphT library were required for finding the complement of the input graph as well as for graph traversal. In particular a complement graph generator and a method for finding Dijkstra shortest path was used. An explanation proceeding the pseudocode provided below clarifies their uses.

Algorithm 1: Knotting Algorithm

```
Input: An undirected graph G = (V1, E).
Output: A directed graph DG = (V2, F), where F is the set of edges with transitive orientation.
begin
    Initialize: V2 ← V1; for e1 ∈ E do
        for e2 ∈ E do
            if e1 = e2 and e1 and e2 not oriented then
                e2.setForwardOrientation(DG, e2);
            end
            if e2 is not oriented and e1.SourceVertex = e2SourceVertex then
                W ← path P1 ∈ E from e1.TragetVertex to e2.TargetVertex
                if |W| = 0 then
                    continue;
                else
                    Vw ← vertex set of W
                    for v ∈ Vw do
                        Q ← any path P2 from e1.SourceVertex to v
                        if 1 ≤ Q ≤ |W| − 1 then
                            if e1.isForwardOriented() then
                                e2.setForwardOrientation(DG, e2)
                            else
                                e2.setBackwardOrientation(DG, e2)
                            end
                        else
                            continue;
                        end
                    end
                end
            end
            if e2 is not oriented and e1.SourceVertex = e2TargetVertex then
                W ← path P1 ∈ E from e1.TragetVertex to e2.SourceVertex
                if |W| = 0 then
                    continue;
                else
                    Vw ← vertex set of W
                    for v ∈ Vw do
                        Q ← any path P2 from e1.SourceVertex to v
                        if 1 ≤ Q ≤ |W| − 1 then
                            if e1.isForwardOriented() then
                                e2.setBackwardOrientation(DG, e2)
                            else
                                e2.setForwardOrientation(DG, e2)
                            end
                        else
                            continue;
                        end
                    end
                end
            end
```

```
        if e2 is not oriented and e1.TargetVertex = e2SourceVertex then
            W ← path P1 ∈ E from e1.SourceVertex to e2.TargetVertex
            if |W| = 0 then
                continue;
            else
                Vw ← vertex set of W
                for v ∈ Vw do
                    Q ← any path P2 from e1.TargetVertex to v
                    if 1 ≤ Q ≤ |W| − 1 then
                        if e1.isForwardOriented() then
                            e2.setBackwardOrientation(DG, e2)
                        else
                            e2.setForwardOrientation(DG, e2)
                        end
                    else
                        continue;
                    end
                end
            end
        end
        if e2 is not oriented and e1.TargetVertex = e2TargetVertex then
            W ← path P1 ∈ E from e1.SourceVertex to e2.SourceVertex
            if |W| = 0 then
                continue;
            else
                Vw ← vertex set of W
                for v ∈ Vw do
                    Q ← any path P2 from e1.TargetVertex to v
                    if 1 ≤ Q ≤ |W| − 1 then
                        if e1.isForwardOriented() then
                            e2.setForwardOrientation(DG, e2)
                        else
                            e2.setBackwardOrientation(DG, e2)
                        end
                    else
                        continue;
                    end
                end
            end
        end
    end
    return DG
  end
end
```

This algorithm orients edges in the forward direction when they are shown to be equivalent. For scenarios where the edges being compared are not equal, the $\sim$ relation checks are enforced. A complement graph generator provided by JGraphT was used for finding the complement edge set of the input graph, in addition to this, Dijkstra shortest path algorithm was utilised to traverse this edge set, retrieve the path W and search the edge set of the input graph to find a path Q from the current vertex to a vertex in W.
Building upon the idea of edges being knotted at a common vertex, this implementation applies the concept of sink and source vertices to orient the edges according to their transitive orientation; these notions are mentioned in section 2.3.3. All combinations of source and target vertices must be considered, denoting that the orientations with respect to the $\sim$ relation, are as follows: We assume that the orientation for vertices a and b are from left to right $\vec{a}b$,

- ab ac, results in a being a source vertex.

- *ab ca*, results in *a* being a source vertex.
- *ba ac*, results in *a* being a sink vertex.
- *ba ca*, results in *a* being a sink vertex.

When all the edges have been considered the outer loop exits and the directed graph is returned.

3.3.3 Improvements

The choice a forward orientation when edges are seen to be equivalent is incorrect, in order to eliminate this feature a method for comparing each sequence of knotted edges should be considered. A solution to this problem was not found as the grouping of 'edges classesd' was hindered by the rigidity of the edge set notation. An improvement would be to use the outer loop to iterate over the set of edges in the directed graph and remove edges from the undirected edge set list once oriented. This would reduce the number of edges needed to be compared as only oriented edges and non-oriented edge will be compared. Unfortunately, this is not possible as the mutation of sets during iteration is forbidden with the potential to lead to inconsistencies in the resulting graph. Overall, the system in place not effective and fails to be a consistent algorithm for detecting comparability graphs meaning there is significant need for improvements.

3.4 Testing

3.4.1 Testing Overview

The overarching testing architecture follows a systematic format. Firstly, the input and output formats are defined, secondly basic tests are run, proceeding this, general test are run with a variety of parameters. The outputs of these basic tests are checked manually for correctness. The process is repeated with improvements applied after each iteration, for which further edge test cases are introduced. The results are documented as csv files with measurements required for analysis and to determine the correctness of the results. For reliable analysis of the properties of the algorithms, the measurements were carefully chosen.

3.4.2 Measurements

The measurements chosen for the analysis of the results and to identify potential improvements in the implementation are as shown below. These measurement also correspond to the headers of the csv output files:
Algorithm run – define the algorithm tested.
Type of graph – by testing graph types pattern recognition can be developed.
Number of input vertices – needed for determining effect on the running time.
Number of input edges – needed for determining effect on the running time.
Number of output vertices – check if it matches the number of input vertices.
Number of output edges – check if it matches the number of input edges.
Reflexivity – check if the graph is reflexive.

Symmetric - check if the graph is symmetric.
Running time in ms – running time needed to identify the complexity.
Successful output – see the output graph has the same number of edges as the input graph and check the the correctness of the graph.

3.4.3 Graph Types

To draw conclusions as to scenarios that will provide better efficiency for one of the two given algorithms, multiple graph types have to be tested. Each test follows a similar design. The design:

- Create a generator for the graph type
- Apply the algorithm and measure the running time
- Check if the number of edges produced is the same
- Check reflexive, symmetric, transitive, to give some indication of correctness
- Write measurement results to relevant csv file
- Repeat and vary the number of vertices and edges

Chapter 4

Results and Analysis

4.1 Overview of the results

The analysis of the results for each algorithm will be divided into three section, the first section will discuss the performance when run on the basic tests, the second will cover performance when dealing with general cases of different graph types and finally the scalability of each algorithm will be reviewed. Initially the basic test results will be discussed, this will cover graphs that were manually created with a known outcome that can be easily verified. The names and shapes of the graphs are as follows:

- Triangle graph: a triangle shaped graph with 3 vertices and 3 edges (comparability graph)
- Square graph: a graph shaped as a square shown in [5] (comparability graph)
- Extended triangle graph: shown in [5] as a triangle graph with each point having a vertex protruding from it. (not a comparability graph)
- House graph: A graph similar to the square graph but with a triangle section connected (comparability graph)
- Star graph: shown in [2] (comparability graph)
- Four triangles: this graph is shown in [10] slides and constitutes of 3 triangles sharing common vertices, known as not being a comparability graph.
- Alternate four triangle: same as the four-triangle graph but with one edge removed (comparability graph)

The next set of graphs are the general graphs, and they make use of some of the properties of comparability graph discussed in 2.1.1, some of these graph types are known to be comparability graphs. The graph types being tested are as follows: bipartite, complete, grid graph, grid variation, variation, symmetric graphs, reflexive graphs, empty graphs, singleton graphs. The bipartite graphs will be tested with partitions ranging from: 1/2, 1/4, 1/10, 1/100, where partition represents the divide in the number of vertices
This variation in graph types will allow for analysis and determining which algorithm is preferable under given conditions.
It is important to note that the time complexities stated have not been achieved and are only specific to the adaptations in this paper.
The system used to conduct these experiments was a MacBook Pro, the system is as follows:

- OS: macOS Monterey
- CPU: 1.4 GHz Quad-Core Intel Core i5
- GPU: Intel Iris Plus Graphics 645 1536 MB

4.2 Gamma results and analysis

4.2.1 Basic test results

All tests were past at a very low running time relative to later tests, and are all shown to be correct by manually checking the output. Table 4.1 shows the results, the success column indicates success and returns transitive orientation when a comparability graph is input otherwise it returns false.

Graph Type	Running Time	Success
Basic triangle	0.3	true
Basic square graph	0	true
Basic extended triangle graph	0.3	false
Basic house graph	0.2	true
Basic star graph	0.1	true
Basic kite graph	0	true
Basic four triangle graph	0	false
Basic alternate four triangle graph	0.3	true

Table 4.1: Gamma: Basic graphs results.

Results of varying vertices:
The effect on the number of vertices is negligible, tests with 0 to 1000 vertices were run and the average running time was 0.0007792, with no indication that an increase in the number of vertices had a direct effect on the running time. Therefore, any further analysis of the results relating to gamma implementation will not take the number of vertices into consideration.
Results of varying edges:
By observing that the main body of computation in the algorithm is the iteration over the edge sets, it is evident that the number of edges does have a direct impact on the running time. This was shown by generating random graphs all with 32 vertices and varying the number of vertices from 0 to the maximum value of 496, this showed a direct quadratic increase in the execution time.

4.2.2 General cases

Grid graphs:
Grid graph NxN: Grid graphs with equal row and column values ranging from 2 to 25 showed a near direct quadratic growth when considering the number of input edges relative to the running time. It can be observed that as the number of edges double the running time increases by a factor of 4; this supports the notion of a time complexity that is of order $O(n^2)$ where n is the number of edges.

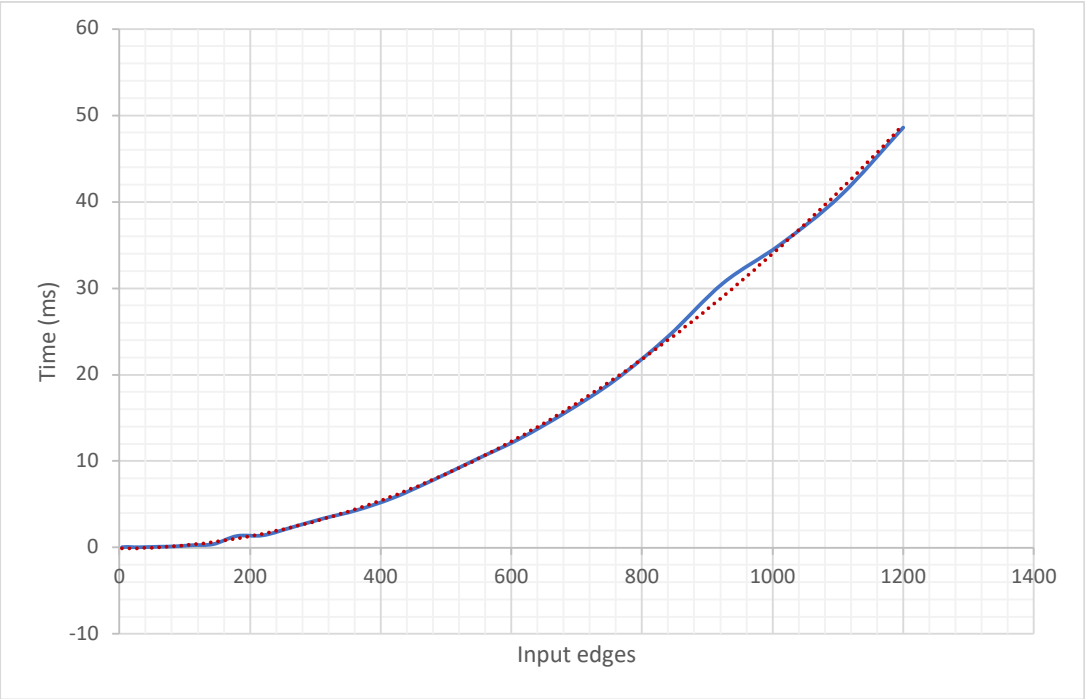


Figure 4.1: Grid graph NxN:
Average running time with respect to the number of edges.

Grid graph 4xN: The running time showed a general quadratic behaviour but with some fluctuation in time values as the input edges increased. This was due to range of edges being relatively small in comparison to other graph tests, this caused increased inconsistency in the running time as the algorithm may have performed in best cases and worst cases but did not have enough input data to normalize to an overall trend.
Grid graph 40xN:
With a large number of input edges the quadratic trend was evident little to no variation from the trend line was displayed, showing that a change in the row number does not affect the time complexity but an augmentation in the number of edges allows for more telling results.
Grid variation NxN: The removal of one edge critically affected the correctness of the algorithm, as the number of output edges was greater than input edges. In all of these cases the algorithm failed and was shown to add an excess number of orientations that contradicted one another. Despite this failure in correctness the running time still shows a quadratic relationship with regards to the number of input edges. The same can be observed with the 4xN and the 40xN graphs but with a greater inconsistency in the execution time. This means that the algorithm 3.1 cannot reliably be used to find the transitive orientation of graphs of this type.
Bipartite graphs:
Bipartite 1/2 and 1/3:
An even partition in the number of input vertices showed cohesive execution time when considering the number of edges and returning a successful transitive orientation for all manually checked graphs with a time complexity of $O(n^2)$. The same can be said for a graph with a partition of 1/3 and 2/3 and the 1/100 and 99/100 but with a less uniform average running times.

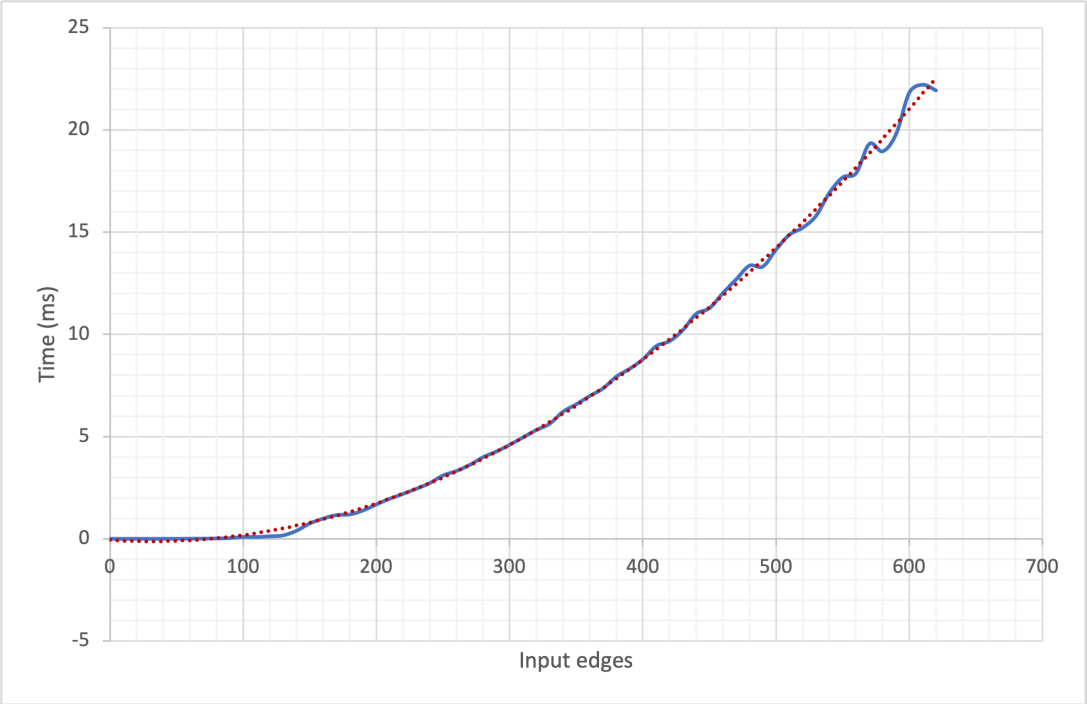


Figure 4.2: Bipartite 1/2 partition:
Average running time with respect to the number of edges.

Bipartite 1/4 and 1/10:
Bipartite graphs with a partition of 1/4,2/4 and 1/10, 9/10 did not show signs of regularity when executed. The correctness for identifying a comparability graph was uniform but the running time did not show any evident trends, except for a general increase after a given number of vertices. For a 1/4 partition the increase was after 150 edges were computed, and for the 1/10 partition 70 edges were needed to be considered before a sharp rise in the running time is displayed.

Complete graphs: Observing the performance of the algorithm's execution on complete graphs a quadratic relation was less evident but still present. Large deviations from the trend line created inconsistencies in the correlation between the running time and the number of input edges, explaining the imperfect quadratic relations. This can be observed in particular when 2016 edges were input and a spike in the running time was recorded, the sudden increase can be explained as being a worst-case scenario where many comparisons are computed to produce the output graph. Despite the ambiguity of the complexity all executions upon complete graphs returned a correct identification of comparability graphs when manually checked.

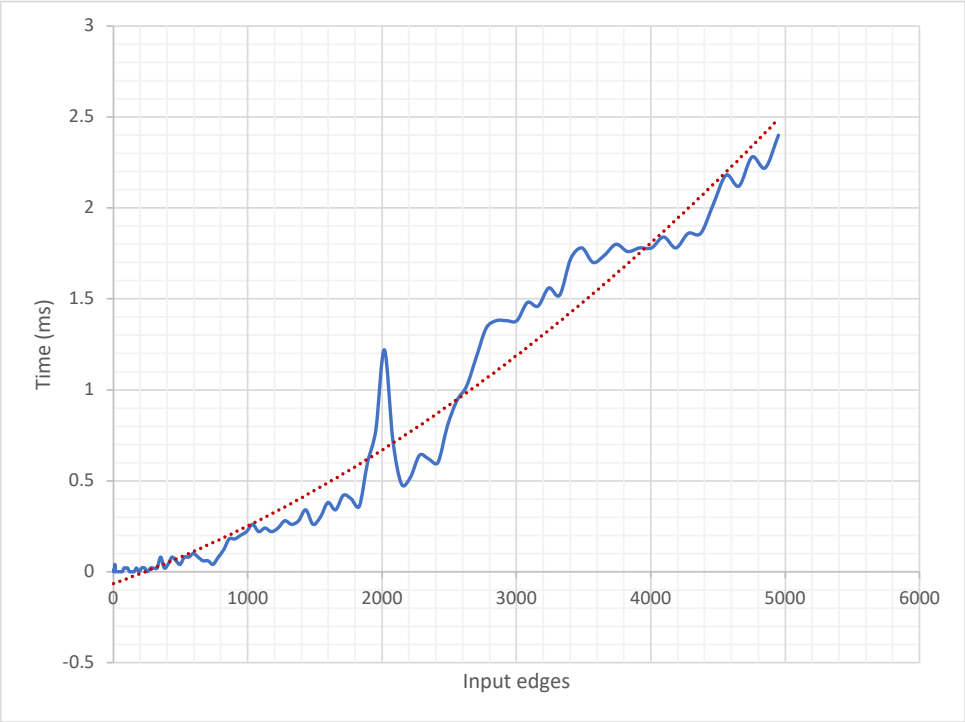


Figure 4.3 Complete graph: Average running time with respect to the number of edges.

Random, reflexive and symmetric graphs: The random graphs generated for testing random graphs, reflexive graphs and symmetric graphs are all identical graphs (the same seed was used) except for the change of one edge that will alter their reflexive or symmetric nature. Negligible discrepancy in the running time was found between the three graph types. The algorithm failed to recognise the difference in the one edge. Incomplete correctness was presented again, this could be observed by the addition of too many output edges in comparison to input edges, making the algorithm ineffective for identification of comparability graphs when presented with a random graph. The functionality fails when exceeding 400 edges leading to a significant drop in the running time. All graphs displayed the same results shown below:

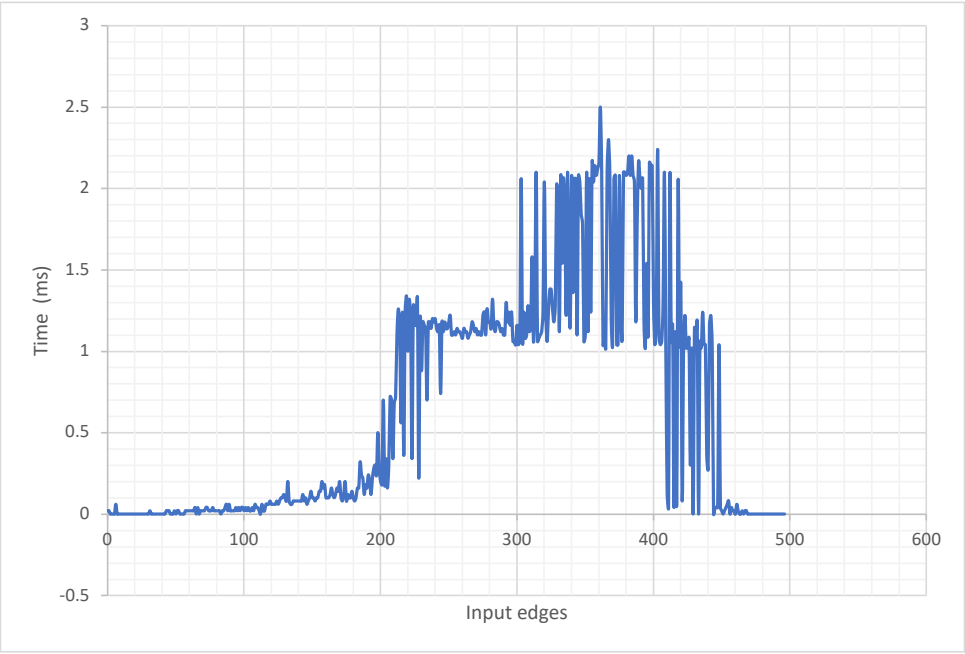


Figure 4.4: Random graph: average running time with respect to the number of edges.

Scalability:
Despite large fluctuation in results, the scalability is consistent and does not deviate from a quadratic relation.

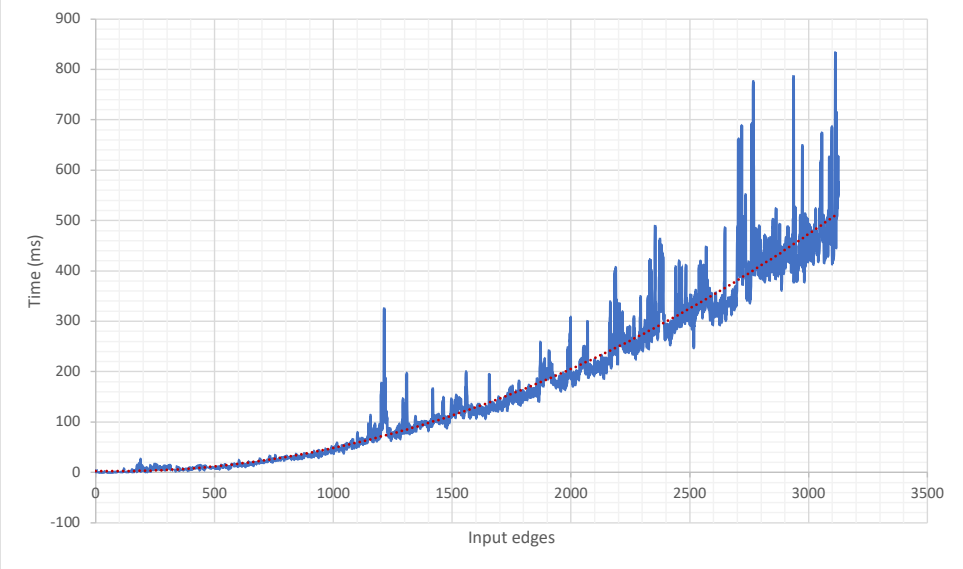


Figure 4.5: Random graph: average running time with respect to the number of edges.

4.3 Knotting results and analysis

4.3.1 Basic test results

The algorithm fails to recognise graphs that are not comparability graphs but does find the correct transitive orientation when the input graph is a comparability graph. Table 4.2 shows success in all cases, even for graphs that are not transitively orientable. We must therefore assume that for all cases, other than for graphs that are known to be comparability graphs, the output is incorrect meaning the only use of the algorithm is to return the transitive orientation of a comparability graph.

Graph Type	Running Time	Success
Basic triangle	2.72900023	true
Basic square graph	0.59817996	true
Basic extended triangle graph	0.50666667	false
Basic house graph	0.56666667	true
Basic star graph	0.65333333	true
Basic kite graph	0.7	true
Basic four triangle graph	1.41867667	true
Basic alternate four triangle graph	0.70785815	true

Table 4.2: Knotting: Basic graphs results.

Result of varying the vertices:

When varying the vertices, there is clear indication that an increase in the number of vertices has a quadratic effect on the running time. The number of edges is kept at 1 while the number of vertices varies from 0 to 100 for randomly generated graphs. A quadratic relation is shown meaning that the complexity with respect to the number of vertices is $O(n^2)$ where n is the number of vertices.

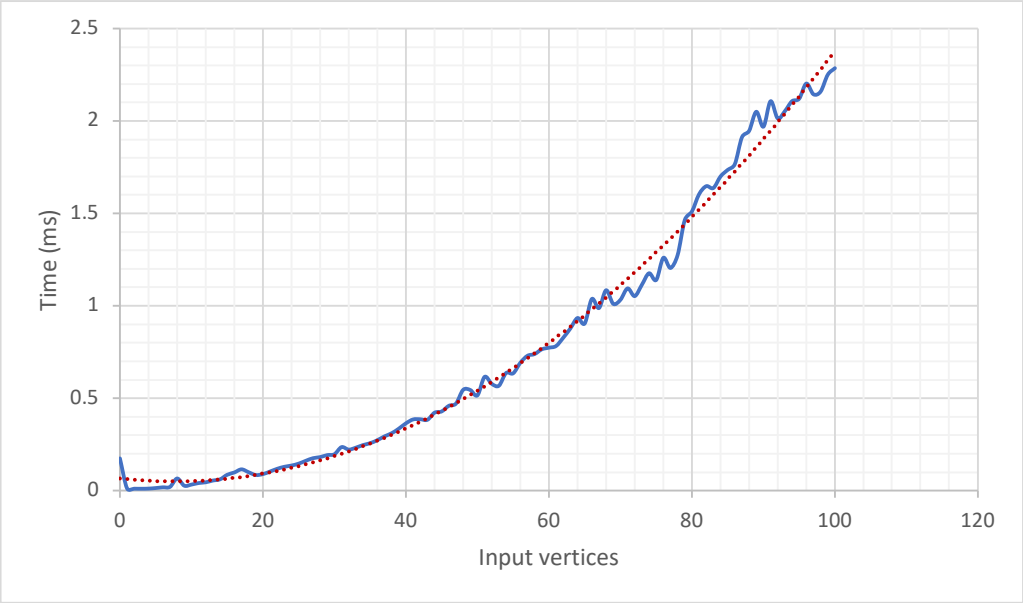


Figure 4.6: Random graph E=1: Average running time with respect to the number of vertices.

Results of varying the edges:

The time complexity with respect to edges is also shown to follow a quadratic curve, in figure 4.6 the number of vertices are kept at 16. We can now determine that both vertex and edge number has a direct impact on the running time.

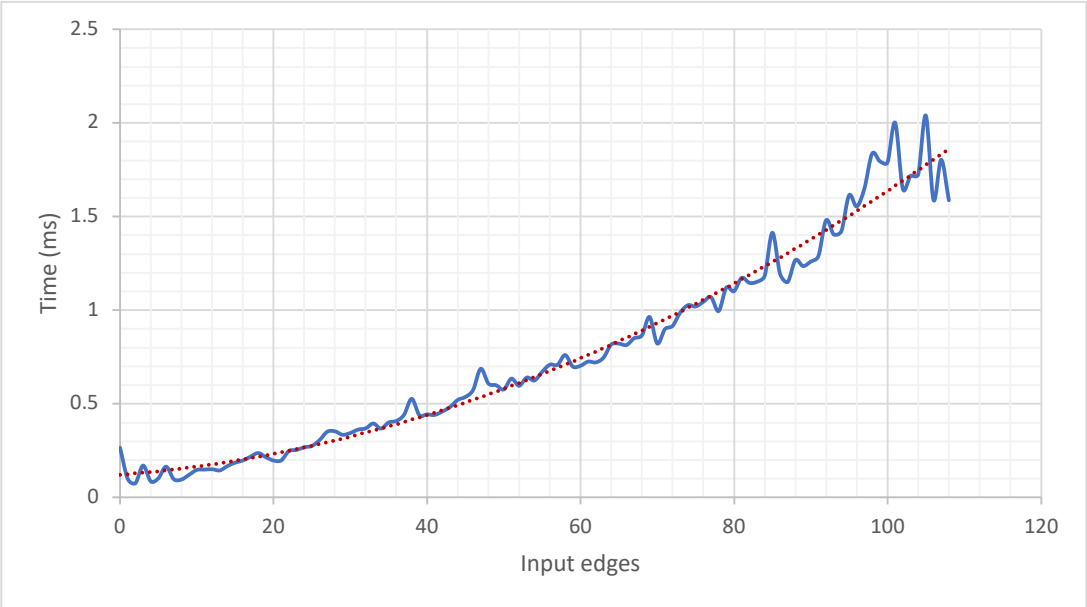


Figure 4.7: Random graph V=16: Average running time with respect to the number of edges.

Bipartite partitions 1/2, 1/3, 1/4 and 1/100:

For cases where the input graph is a bipartite graph with an even partition of the vertices there is a general trend supporting the quadratic nature previously established, figure 4.6 below

displays the average running time increasing by a factor of 4 when the number of vertices is doubled. This relation is shown in partitions of 1/4 and 1/100, emphasising that a different partition of the vertices does not affect adversely affect the time complexity.

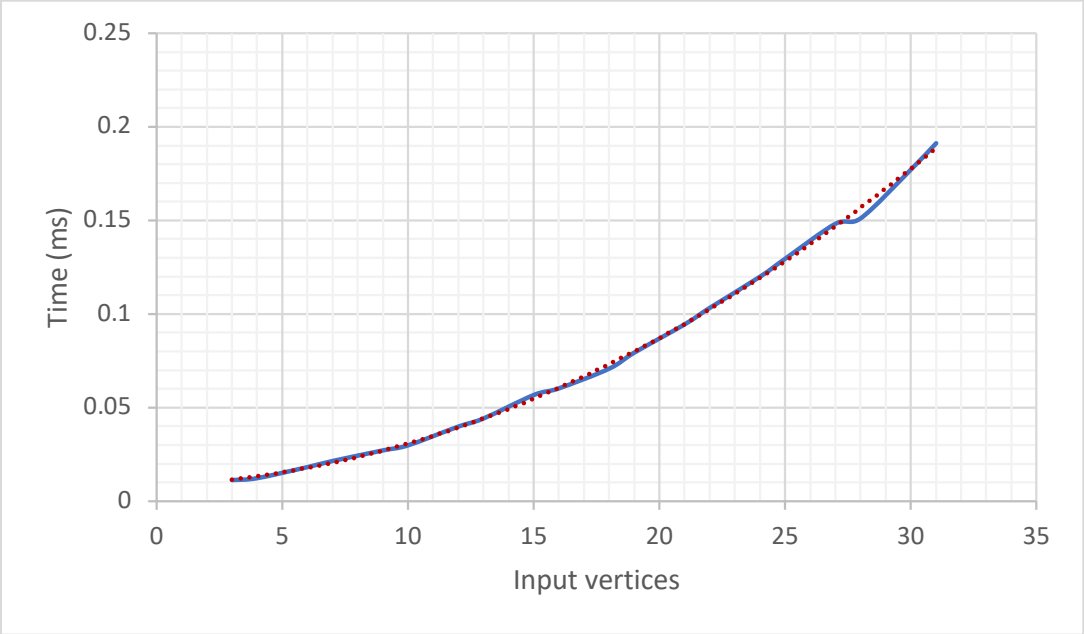


Figure 4.8: Bipartite 1/3 partition: Average running time with respect to the number of vertices

Linear complexity is noticed in all cases if only the input edges are varied. There is an improvement in the time complexity when compared to random graphs being input, therefore the algorithm shows indication of better performance when the input graph is a bipartite graph.

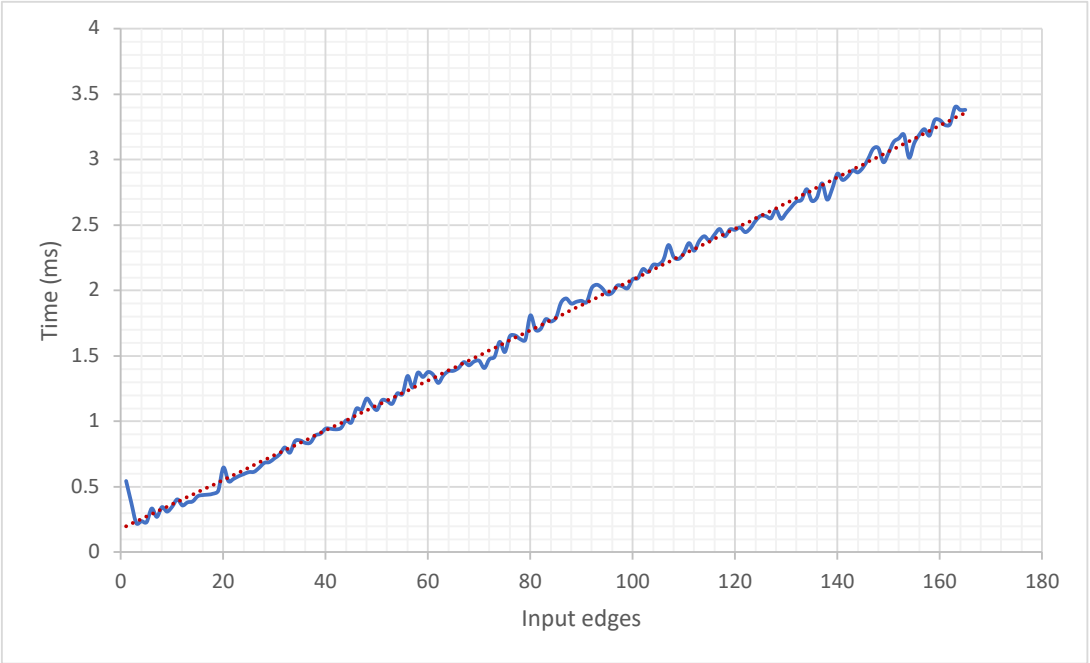


Figure 4.9: Bipartite 1/3 partition: Average running time with respect to the number of edges

It is worth noting that for the number of vertices in a 1/10 partition, the figure 4.10 shows a large spike in running time at 60 input vertices, this is considered an anomaly in the data and if we ignore this sudden rise, the results show promise for supporting the idea of a quadratic

trend. The surge can be explained as a worst-case scenario with significant impact on the time of execution, showing the flaws in the robustness of the algorithm.

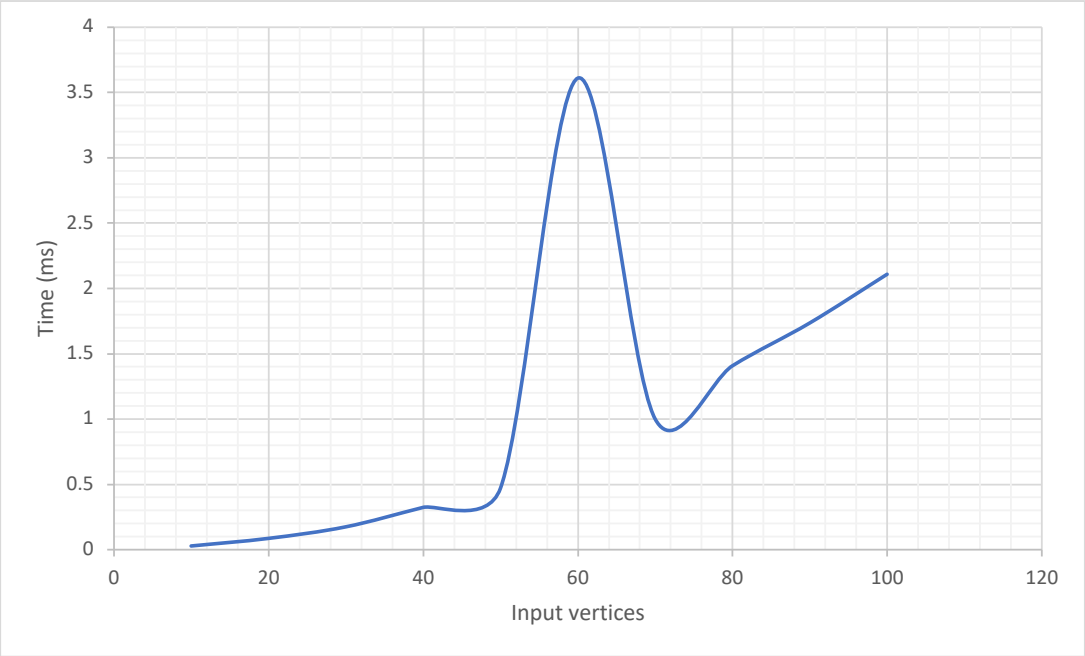


Figure 4.10: Bipartite 1/10 parittion: Average running time with respect to the number of vertices.

Complete graphs:

For a complete graph the number of edges grows quadratically with the number of vertices, while the number of vertices grows linearly, meaning that the ratio V/E approaches an infinitesimally small value as the graph size increases. The running time is shown to have a cubic tendency with regards to the ratio of vertices over edges.

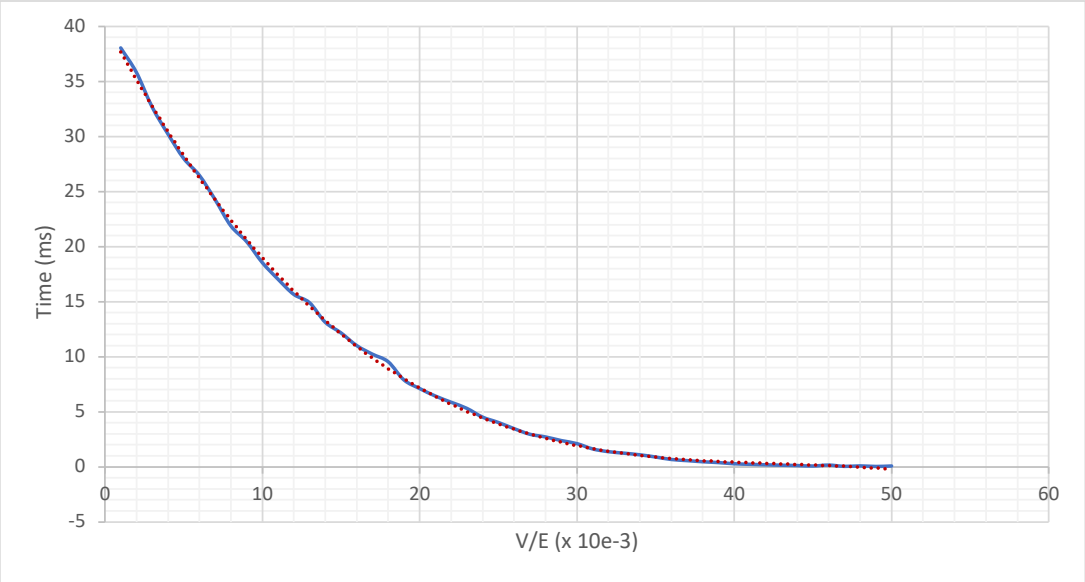


Figure 4.11: Complete graph: Average running time with respect to the ratio of number vertices/edges.

Grid graphs:

Every grid graph followed the same trend regardless of the variation in one edge or not. Indication that if the number of vertices is just under 6/10 of the number of edges, then the running time is dramatically augmented, this can be explained by the algorithms need to run more checks along paths created in a denser graph.

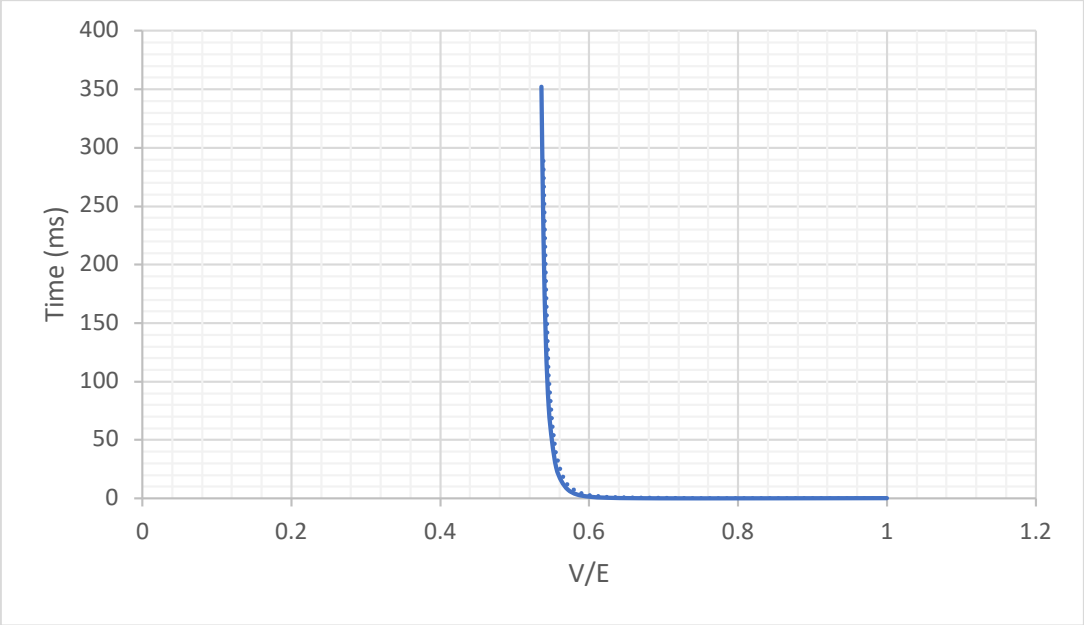


Figure 4.12: Grid graph NxN: Average running time with respect to the ratio of number vertices/edges.

Scalability:

If the final anomalies shown in figure 4.13 are not considered the number of input edges cause an exponential growth of the running time of the algorithm. This algorithm scalability is shown to be extremely poor when run on random graphs with a greater number of edges than 350, for graphs under this value the performance is reasonable. The number of vertices and its quadratic trend, as well as the number of edges and its exponential effect on the running time, definitively highlight the inefficiency of the implemented algorithm when executed on large graphs.

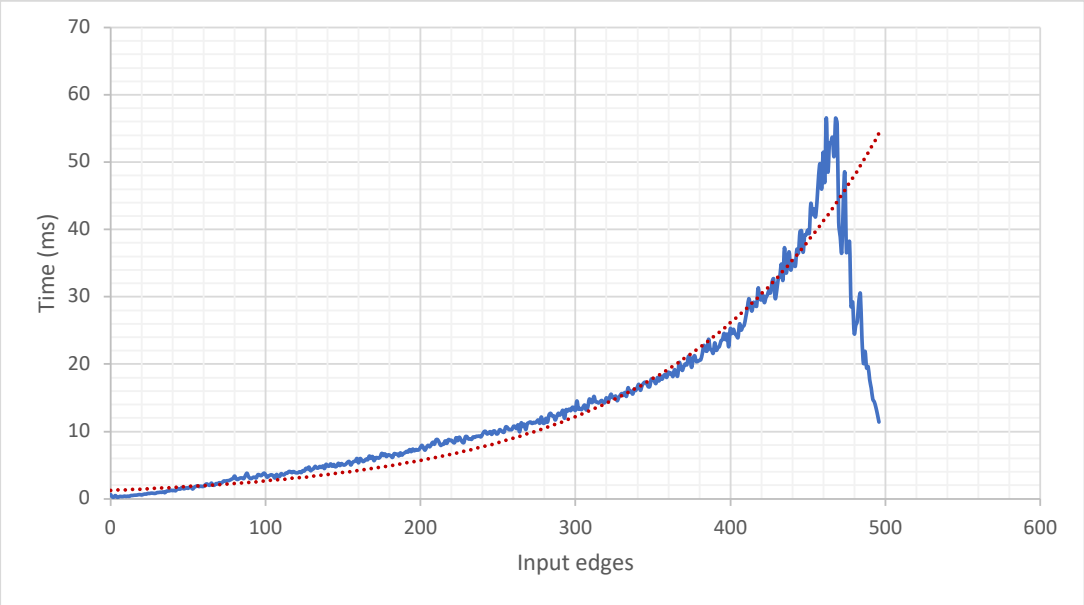


Figure 4.13: Random graph: Average running time with respect to the number of edges.

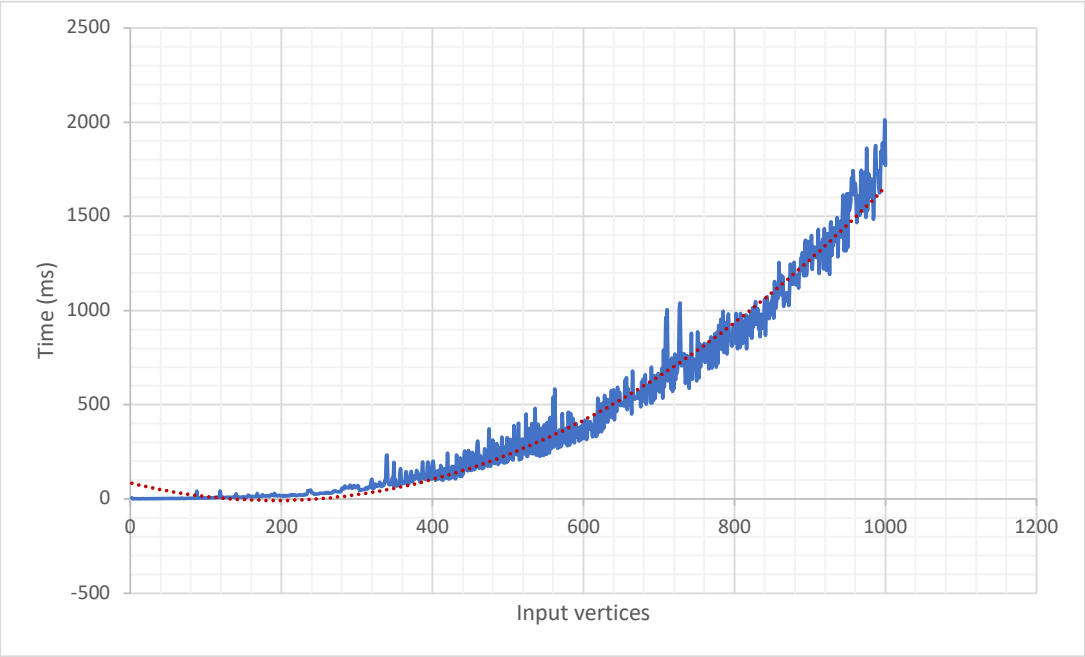


Figure 4.14: Random graph: Average running time with respect tot the number of vertices.

Chapter 5

Project Evaluation and Conclusions

The process of evaluation will review the objectives and aims of the project, discuss the limitations and mitigations taken to allow for a more complete body work.

5.1 Meeting Aims and Objectives:

By referring to sections 1.3 and 1.4 the aims and objectives can be assessed.
The first objective was the implementation of the two algorithms, unfortunately this objective was not achieved due to limitations in knowledge and coding capabilities causing consequential effects to the proceeding sections of the project. Despite this failed objective some mitigations were put in place so that quantifiable data could be collected for an analysis to be carried out. These mitigations were the variations of the forementioned algorithms, neither of these variations are an accurate representation of the theoretical ideas discussed but they did allow for transitive orientations to be delivered in some cases. This is not a desirable outcome, and these algorithms should not be used outside of this research project.
The second objective was achieved to a standard below that originally intended. Initially the stated time complexities were believed to be attained but for a different complexity were implied by explaining the variation in implementation and explicitly discussed later in section 3.4.
The third aim was achieved; a range of input cases were produced based on known properties of comparability graphs, the performances of the implementations were examined so that explanations could be drawn and the running time of each were backed up with quantifiable data.
The fourth objective for determining which is better was easily distinguished, the incompleteness of the algorithms and limited correctness of their output showed that neither were suitable for use other than for research but overall, the gamma implementation showed clear signs of being a more promising algorithm.
The main aim of delivering two usable algorithms and determining which is preferable in real world contexts was not completed, in retrospect this project appeared to be realisable but revealed to be a complex body of work that required greater understanding and expertise in the subject area other than the algorithms 2 module studied in second year.

5.2 Planning, Project Management and Methodology

The project plan inspired promise in the realisation of the project but every phase of the plan was delayed because of arising complexities in the general problem, perhaps a less rigidly structured approach may have allowed for more flexibility for mistakes and further research. The research phase was heavily extended over time because of implementations being unsuccessful despite the basis of the ideas being well understood. Overall, the project management appeared to be sound when starting but the need to learn many new aspects with

regards to graph theory hindered the management of later phases of the project, resulting in the forced delivery of incomplete algorithms and analysis of data that did not yield any critical new ideas.

5.3 Challenges and limitations

With only a brief introduction to graph theory from the Algorithms 2 module studied in second year, many challenges were faced when learning new material. The limited basic literature regarding comparability graphs meant referring back to the initial papers [1] and [2] as the main sources for understanding the problem, this proved to be challenging as most aspects and relations were initially a new concept and required further research. Similar issues were face when learning to use JGraphT library. The library at first proved to be challenging as the documentation available is limited in its description of classes and how they are created, therefore a lot of trial and error was needed to come to grips with it. Appropriate testing for graph algorithms as well as determining relevant measurements was a completely novel area of the subject; this resulted in more research which subsequently delayed other aspects of the project.

5.4 Conclusion

The inability to implement a method for determining the ‘implication classes’ or ‘edge classes’ proved to be the greatest bottleneck in the project’s development. Despite this failure, a conclusion can be drawn as to which algorithm showed more promise when considering the ease of theoretical understanding, implementation and better time complexity. The analysis was carried out on an incorrect variation of the algorithms and there is evident indication that the variation gamma rule algorithm is preferable for all graph types. Due to the limitations and challenges faced this is the only plausible conclusion that can be deduced, but it must be said that there is no certainty or indication that this conclusion can be extended upon a correct implementation of the algorithm.

References

[1] A. Chan, F. Dehne, and R. Taylor. Cgmgraph/cgmlib: Implementing and testing cgm graph algorithms on pc clusters and shared memory machines. *The International Journal of High Performance Computing Applications*, 19(1):81–97, 2005.

[2] T. Gallai. Transitiv orientierbare graphen. *Acta Mathematica Hungarica*, 18(1-2):25–66, 1967.

[3] A. Ghouila-Houri. Sur la généralisation de la notion de commande d'un système guidable. *Revue française d'informatique et de recherche opérationnelle*, 1(4):7–32, 1967.

[4] P. C. Gilmore and A. J. Hoffman. A characterization of comparability graphs and of interval graphs. *Canadian Journal of Mathematics*, pages 539–548, 1964.

[5] M. C. Golumbic. *Algorithmic graph theory and perfect graphs*. Elsevier, 2004.

[6] D. Kelly. Comparability graphs. *Graphs and Order: The Role of Graphs in the Theory of Ordered Sets and Its Applications*, pages 3–40, 1985.

[7] P. A. Kumar. Csl851: Algorithmic graph theory fall 2013 lecture 3, 2013. Available at: `https://www.cse.iitd.ac.in/~naveen/courses/CSL851/arpit.pdf`, version 1.6.0.

[8] D. Michail, J. Kinable, B. Naveh, and J. V. Sichi. Jgrapht—a java library for graph data structures and algorithms. *ACM Trans. Math. Softw.*, 46(2), May 2020.

[9] S. D. Nikolopoulos and L. Palios. Algorithms for p4-comparability graph recognitionand acyclic p4-transitive orientation. *Algorithmica*, 39(2):95–126, 2004.

[10] W. T. Trotter. Math 3012 applied combinatorics lecture 15, 2015. Available at: `https://bpb-us-w2.wpmucdn.com/sites.gatech.edu/dist/8/179/files/2015/11/3012-Lecture-15.pdf`.

[11] E. W. Weisstein. Graph theory. From MathWorld—A Wolfram Web Resource. Last visited on 15/04/2023.

[12] R. Wiśniewski. *Perfect Graphs and Comparability Graphs*, pages 31–48. Springer International Publishing, Cham, 2017.

[5] [2] [4] [6] [3] [1] [7] [10] [8] [11] [9] [12]

Appendix A

Self-appraisal

A.1 Critical self-evaluation

The main objectives and aims of this project were not achieved, the project as a whole can be considered a failure. This was a difficult task to undertake, it is to my understanding that previous students had taken on this task and were unfortunately not able to complete it successfully either, it is clear that it required a very good base foundation and understanding of graph theory, which I believe is beyond the introduction of the graph algorithms presented in the Algorithms 2 module.
The work accomplished here is not to a standard I would have liked despite the regular meetings and hours put into the project. The objectives were all only partly complete as the main bottleneck of the project, implementing the algorithms caused the following objectives incompleteness. In retrospect, the objectives should have been chosen to have less critical dependency upon one another, this way the project's continuity would have suffered less.
The literature review was well executed, examples have been shown to express my understanding of the topic, this was one aspect of the project that was successful.
The implementation phase was where the most difficulty was encountered, the trials to find a method of distinguishing the "implication classes" or "edge classes" were met with frustration and little success. Many different approaches were explored, where one step in the right direction then created multiple further problems until the approach was found to be impossible to implement due to the rigidity of the sets being used. This would mean more research was required creating a build-up of backlogged tasks and a collection of code that only proved partially successful. Time constraints meant that the analysis phase of the project required a deliverable which forced the decision to have variation algorithms as an alternative.
The final phase of the project was a success as a stand-alone analysis of results although the resulting data was not strictly correct it was still able to reveal trends and patterns which could be examined. Ideally more in-depth analysis could have been delivered but limited time did not allow for further expansion of ideas.
Overall, the body of work presented in this report is underwhelming in consideration to the work put in. Perhaps a more seasoned coder would have found more success, despite the failures in the objectives I was still able to learn a lot from the experience.

A.2 Personal reflection and lessons learned

Organisation and time management was initially well structured but delays in implementation due to the need of further research into the field meant that the final outcomes of the project suffered. The choice of continuing onto the analysis of performance without a correct implementation of the algorithms greatly impacted the objectives achievability but I believe it was the right decision as it led to a far greater appreciation as to how testing and performance

analysis is carried out with regards to graph algorithms. This entire process was something I had never done to such depth and has now installed confidence in my ability to replicate such a procedure but with improvements and a stronger sense of direction. On the other hand, my confidence with regards to writing code has regressed because all of the unsuccessful attempts at creating a correctly functioning algorithm despite having understood the problem well. This is not to say that I did not gain a more comprehensive grasp of the Java language, although I am aware that more can be learnt to better the functionality of the code written.

A.3 Legal, social, ethical and professional issues

A.3.1 Legal issues

The BCS's code of conduct (BCS, 2020) considers legal issues as "having regard for the legitimate right of Third Parties". All references to other work has been provided and cited throughout the paper as to not infringe upon misuse of intellectual property. No other legal issues appear in this paper.

A.3.2 Social issues

No personal data of any kind was used in this project and testing did not include the need for participants, therefore this report poses no social issues.

A.3.3 Ethical issues

No concerns regarding privacy or harm are present in this report, meaning ethical issues are not relevant.

A.3.4 Professional issues

No professional issues are present as there is solely one contributor to the project.

Appendix B

External Material

Libraries used:

JGraphT: jgrapht-core:1.5.1, link to github repository: https://github.com/jgrapht/jgrapht

Pseudocode:
Pseudocode extracts are cited from [5] and [2].

Microsoft Excel was used for data analysis and graph creation.

Appendix C

Source code and repositories

Link to source repository:
`https://gitlab.com/joseph_halfpenny/final-year-project.git`
Results and analysis excel files are included in the repository.