Computing the Leafage of Chordal Graphs
Jonathan Sacramento
BSc Computer Science
2007/2008

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)________________________
Summary

This project investigates the open, graph-theoretic problem of computing the leafage of chordal graphs. Three graph algorithms are discussed in detail and implemented, namely a chordal and interval graph recognition algorithm and a clique tree construction algorithm. The first polynomial time algorithm for computing the leafage of chordal graphs is presented with a proof of its correctness and its time-complexity analysis.
Acknowledgements

I would like to thank my project supervisor Dr. Haiko Müller for coming up with the idea for the project and for his invaluable help and support throughout the year.

I would also like to thank my family for their support and for always reminding me that as long as I give my best I can rest happy with the results.
# Contents

1 Introduction ........................................... 1
   1.1 Aim ............................................. 1
   1.2 Objectives ...................................... 1
   1.3 Minimum Requirements .............................. 2
   1.4 Schedule ....................................... 3

2 Background Research .................................. 4
   2.1 Graph Theoretic Foundations ...................... 4
       2.1.1 Overview .................................. 4
       2.1.2 Basic Concepts .............................. 5
       2.1.3 Chordal Graphs ............................... 5
       2.1.4 Interval Graphs ............................... 6
       2.1.5 Clique Trees ................................ 6
   2.2 Background to the Problem ......................... 7
   2.3 Methodologies .................................... 9
       2.3.1 Overview .................................. 9
       2.3.2 Software Development ......................... 9
       2.3.3 Theoretical Research ......................... 10
   2.4 Appropriate Technologies .......................... 10
       2.4.1 Programming Languages ....................... 11
       2.4.2 Graph Modelling Libraries .................... 11

3 Graph Algorithm Implementations ..................... 13
   3.1 Chordal Graph Recognition Algorithm ............... 13
3.1.1 Analysis of Existing Algorithms ........................................... 13
3.1.2 Implementation of Algorithm ............................................. 14
3.1.3 Testing of Algorithm ....................................................... 17
3.2 Interval Graph Recognition Algorithm ..................................... 17
  3.2.1 Analysis of Existing Algorithms ........................................ 17
  3.2.2 Implementation of Algorithm .......................................... 18
  3.2.3 Testing of Algorithm .................................................... 20
3.3 Clique Tree Construction Algorithm ...................................... 20
  3.3.1 Analysis of Existing Algorithms ....................................... 20
  3.3.2 Implementation of Algorithm .......................................... 21
  3.3.3 Testing of Algorithm .................................................... 23

4 Algorithm for Computing the Leafage of Chordal Graphs ............. 24
  4.1 Overview ............................................................................ 24
  4.2 Design of the Algorithm .................................................... 24
  4.3 Algorithm for Computing the Leafage of Chordal Graphs ............ 27
  4.4 Proof of Correctness ......................................................... 31
  4.5 Time Complexity Analysis .................................................. 33

5 Evaluation ............................................................................. 34
  5.1 Overview ............................................................................ 34
  5.2 Evaluation of the Graph Algorithm Implementations ................ 34
  5.3 Evaluation of the Algorithm for Computing the Leafage of Chordal Graphs .................................................. 35

Bibliography ............................................................................ 37
Chapter 1

Introduction

1.1 Aim

The aim of the project is to develop an algorithm for computing the leafage of chordal graphs in polynomial time, and to provide a proof of correctness and time complexity analysis of the algorithm.

1.2 Objectives

There are two sets of objectives. The first set (project aim objectives) is composed of those tasks which, when completed, achieve the aim of the project. The second set (minimum requirement objectives) defines those tasks which are to be completed in order to meet certain minimum requirements, namely the implementation of the graph algorithms (see the following section).

The project aim objectives are to:

- Research chordal graphs, interval graphs, clique trees and any areas which stem from these.
- Research all prior work done on or related to the problem.
- Develop an algorithm for computing the leafage of chordal graphs.
• Provide a proof of correctness and time complexity analysis of the algorithm.

The **minimum requirement objectives** are to:

• Evaluate suitable programming languages for the implementation of graph algorithms.
• Evaluate suitable graph modelling libraries for representing graphs.
• Implement an algorithm for recognising chordal graphs.
• Implement an algorithm for recognising interval graphs.
• Implement an algorithm to construct a clique tree from a chordal graph.

### 1.3 Minimum Requirements

The minimum requirements are the measurable components against which the project is assessed. These are to:

• Implement an algorithm for recognising chordal graphs.
• Implement an algorithm for recognising interval graphs.
• Implement an algorithm to construct a clique tree from a chordal graph.
• Develop an algorithm to compute the leafage of chordal graphs.
• Prove the correctness of the algorithm and analyse its time complexity.

The possible extensions to the project include:

• If the *upper leafage* is defined as being the *maximum* number of leaves of a host tree, then a possible extension would be to compute or bound the upper leafage.
• Benchmarking framework for the developed algorithm.
• Optimization of the developed algorithm.
1.4 Schedule

Once the objectives and minimum requirements were agreed upon, the following schedule was developed. This schedule, in tabular form, lists the main tasks to be completed and the duration of the project in months. A shaded cell means that the corresponding task is to be completed by the end of that month.

<table>
<thead>
<tr>
<th>Task</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background Research</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implement Chordal Graph Recognition Algorithm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implement Interval Graph Recognition Algorithm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implement Clique Tree Construction Algorithm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Develop Algorithm for Computing the Leafage of Chordal Graphs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proof of Correctness and Time Complexity Analysis of Algorithm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.1: The project schedule
Chapter 2

Background Research

2.1 Graph Theoretic Foundations

2.1.1 Overview

This section serves two purposes, firstly it presents the background literature review carried out and secondly, it provides the prerequisite graph theoretic concepts needed to properly understand the nature of the problem, the project aim and the objectives. Basic concepts in graph theory are given first, followed by a definition and description of the main types of graphs relevant to the project. The background to the problem is presented next. At this point, the reader should have a clear understanding of the problem the project is aiming to solve.

The methodologies adhered to for both the software development and theoretical research phases are evaluated. Furthermore, an evaluation is carried out of suitable technologies to be used for the implementation of the graph algorithms.
2.1.2 Basic Concepts

An in-depth introduction to the study of graphs is provided by West [30]. Hereunder is a brief overview of the prerequisite knowledge on graphs required to understand the problem.

A graph $G$ is composed of a finite set $V$, the set of vertices, and a binary relation on $V$ which is denoted $E$, the set of edges. Graphs are usually represented pictorially, however, as it is much easier to understand by means of a diagram as opposed to complex mathematical symbols. Circles are used to represent vertices and lines between pairs of vertices represent the edges. A vertex is adjacent to another vertex if and only if there is an edge between them. A vertex is incident to the edge joining it to any of its adjacent vertices. The degree of a vertex $v$, $\text{Deg}(v)$, is the number of edges incident to $v$.

The complement of a graph $G$, denoted $\bar{G}$, is the graph with the same vertex set, $V$, as $G$ but with edges joining each pair of unadjacent vertices in $G$. A graph, $K_n$, is complete if the set of vertices, $V$, is pairwise adjacent. A subgraph of a graph $G = (V,E)$ is any graph $H = (V',E')$ where $V' \subseteq V$ and $E' \subseteq E$. A clique, $K_r$, in a graph $G$ is a complete subgraph in $G$ of size $r$ (i.e. a complete subgraph comprising $r$ vertices). A maximal clique is a clique that cannot be enlarged by adding another vertex to it.

A tree is a connected, acyclic graph. A leaf in a tree is a vertex, $v$, with $\text{Deg}(v) = 1$. A spanning tree of a graph $G$ is a subgraph of $G$, spanning its vertex set $V$, that is also a tree.

A clique matrix is a matrix used to represent a graph in terms of its cliques. The graph’s vertices are placed on the rows of the matrix and the cliques are placed on the columns. If $v \in K$ then the cell representing the vertex and the clique will have a 1, otherwise the cell will hold a 0.

2.1.3 Chordal Graphs

Definition 1 An undirected graph $G = (V,E)$ is a chordal graph if every cycle of length greater than 3 has a chord.

There are several characterisations of chordal graphs. For example, "a graph $G$ is chordal if and only if every minimal vertex separator of $G$ is complete in $G$" is one given by Dirac [7]. Also, "a graph $G$ is chordal if and only if $G$ has a simplicial elimination ordering” Fulkerson [8]. However, the following
characterisation, given by Gavril [9], will be used as it follows from the problem definition,

"A graph is chordal if and only if it is the intersection graph of a family of subtrees of a host tree".

Golumbic and Biedl [12, 2] provide a good treatment of chordal graphs. Furthermore, a recognition algorithm is given in both, namely the lexicographic breadth first search (See Section 3.1).

### 2.1.4 Interval Graphs

Consider a set of intervals, such as numeric intervals on the real line (e.g. 4-5, 3-8, 1-6 etc.). An interval graph is constructed by representing each interval by a vertex, and connecting pairs of vertices if and only if their corresponding intervals intersect. In the above example, 4-5 intersects with 3-8 because the numbers 4 and 5 are contained in the interval 3-8. As with chordal graphs, Golumbic and Biedl [12, 2] also provide an indepth analysis of interval graphs and their properties. Corneil [6] introduces a recognition algorithm for interval graphs based on four sweeps of the lexicographic breadth first search (see Section 3.2).

As with chordal graphs, there are various characterisations of interval graphs. However, the definition given hereunder will be used throughout the document. Before providing a definition of an interval graph, it is necessary to define what an asteroidal triple is.

**Definition 2** An **asteroidal triple** is a triple of vertices, such that between any two vertices there exists a path which avoids the neighbourhood of the remaining vertex.

**Definition 3** An undirected graph $G = (V,E)$ is an **interval graph** if and only if it is a chordal graph and asteroidal triple free.

### 2.1.5 Clique Trees

Clique trees were introduced independently by Buneman [5], Gavril [9] and Walter [29]. However, West [30] provides a clear definition hereunder:

A tree $T$ is a clique tree of $G$ if there is a bijection between $V(T)$ and the maximal cliques of $G$ such that for each $v \in V(G)$ the cliques containing $v$ induce a subtree of $T$. 
However, Shibata [26] shows that a clique tree $T$ of a chordal graph $G$ can be constructed by first generating a weighted intersection graph of $G$; that is, a graph $K_G$ constructed by representing each maximal clique in $G$ by a vertex, connecting pairs of vertices if and only if the maximal cliques intersect, and assigning to each edge a weight given by the number of vertices shared by both maximal cliques. Thereafter, a clique tree of $G$ is generated by finding a maximum weight spanning tree of $K_G$; that is, a spanning tree where the sum of the edge weights is the largest of all spanning trees.

**Definition 4** A clique tree of a chordal graph $G$ is given as the maximum weight spanning tree of the weighted intersection graph $K_G$ of $G$.

Mckee and Blair et al. [22, 3] both explore the clique trees of chordal graphs and prove interesting properties, including "a connected graph $G$ is chordal if and only if there exists a tree for which the clique intersection property holds" etc. These provide a fertile collection of relationships between chordal graphs and their corresponding clique trees which can be explored further.

### 2.2 Background to the Problem

As mentioned in Section 2.1.3, a graph is chordal if and only if it is the intersection graph of a family of subtrees of a host tree. Thus, there is a tree $T$, which houses a family of subtrees, some of which may intersect (See Fig 2.1) With this information we can give a definition for the leafage of a chordal graph.

![Diagram](image.png)

**Figure 2.1:** A Subtree Representation on Host Tree $T$ and it’s corresponding Chordal Graph $G$
**Definition 5** The **leafage**, \( l(G) \), of a chordal graph \( G \) is the minimum number of leaves of the host tree in a subtree representation of \( G \).

The aim of the project is to develop an algorithm which, given a chordal graph \( G \), computes the leafage \( l(G) \) in polynomial time. Little work has been carried out explicitly on the leafage of chordal graphs, perhaps because there is currently little or no practical interest in doing so. The only paper published on the subject is by Lin et al. [21] titled "The Leafage of a Chordal Graph". This publication does not aim to provide an algorithmic solution for computing the leafage however, as it focuses on the mathematical properties of chordal graphs and many of it’s subclasses. The paper obtains upper and lower bounds on \( l(G) \) and computes it in certain special cases. Furthermore, the paper makes use of asteroidal sets (Kloks et al. [17] proves that computing the asteroidal number is NP-Complete for general graphs), an asteroidal set analogy for subtrees (asteroidal collections) and other structural properties of chordal graphs. The article goes on to suggest that certain ideas presented can be combined with the notion of a "dominator tree" to design a polynomial time algorithm in general.

There are various papers which touch on topics directly related to the leafage. Gavril and Shibata [10, 26], for example, show that minimal subtree representations correspond to the clique trees of chordal graphs. Before explaining why this is relevant, it should be mentioned that "every pairwise intersecting family of subtrees of a tree has a common vertex" [12]. Minimal subtree representations are defined by Lin et al. [21] as follows:

If \( Q \) is a clique in \( G \), any subtree representation \( f \) of \( G \) assigns some host vertex to all of \( Q \). If distinct vertices \( q,q' \) are assigned to cliques \( Q,Q' \) with \( Q \subset Q' \), then for \( v \in Q \) the entire \( q,q' \)-path in the host belongs to \( f(v) \). The first edge on this path may be contracted to smaller representation without changing the number of leaves. ... **Minimal subtree representations are those subtree representations having a bijection between the maximal cliques of \( G \) and the vertices of the host tree**
2.3 Methodologies

2.3.1 Overview

This sub-section briefly reviews some well-established software development methodologies following the outline of Bennet et al.[1], and discusses possible methodological systems suitable for theoretical research. A distinction is therefore made between methodologies applied to the software development aspect of the project and those adhered to when conducting theoretical research. A section is devoted to each, where an evaluation is conducted and the subsequent conclusions drawn.

2.3.2 Software Development

The software development aspect of the project will adhere to an established methodology. Furthermore, the chosen methodology will have to be altered slightly as most are aimed at large-scale commercial software projects. The two most-suited methodologies are listed hereunder:

- **Unified Process[1]**
  The unified process is an iterative and incremental software development framework. It’s extensible nature allows for it to be customized to particular projects. The Unified Process divides the project into four phases (Inception, Elaboration, Construction and Transition) each of which requires particular emphasis on a set of tasks from the following, Modelling, Requirements Analysis, Design, Implementation, Testing and Deployment.

- **Waterfall Model[1]**
  It is well understood that nowadays the waterfall model is considered obsolete. However, for modularised implementations of this magnitude it should be considered. The waterfall model provides a sequential flow through each software development phase. These include, Requirements Analysis, Design, Implementation, Testing (validation), Integration, and Maintenance.

The implementation of an algorithm will require an initial analysis stage to establish inputs, outputs and general internal functions; A design stage where a holistic view of how the pseudocode is to be implemented in order to maximise it’s efficiency; An implementation stage where the algorithm is implemented in the appropriate programming language and a testing stage where the implemented algorithm is tested with a carefully selected range of inputs.
An iterative approach over the aforementioned stages seems the most effective means of implementing any algorithm. The first iteration would produce the basic ‘skeleton’ of the algorithm. The second iteration would build on the first, fixing any obvious bugs and performing extensive testing. The third iteration would provide a stable (or close to stable) implementation of the algorithm.

Such an approach closely resembles the Unified Process, although not completely. This method, however, seems the most appropriate.

### 2.3.3 Theoretical Research

The aim of the project is to carry out research in the area of chordal graphs for the purpose of developing an algorithm. Given the nature of research work, it is close to impossible to categorise the process as this would lead to improper scheduling of the task, i.e. ”the algorithm will be completed by such a date” etc. It is therefore impractical to adhere to a strict methodology as this would not be conducive to the research work being carried out.

However, the research will follow a generic strategy of reducing the problem in a “divide and conquer” fashion, ultimately aiming to piece together the sub-problems to successfully produce a solution. In this case, algorithms which solve the sub-problems in polynomial time will be pieced together to form a polynomial time algorithm in the general case.

### 2.4 Appropriate Technologies

This section evaluates technologies appropriate for the implementation of graph algorithms. The two main requirements are a:

1. suitable programming language
2. graph modelling library for building graph models in the chosen programming language
2.4.1 Programming Languages

In order to implement the algorithms a programming language is required. The criteria for evaluation is:

- Programmer ability in the language;
- Language versatility;
- Object-oriented; and
- The availability of third-party graph modelling libraries.

The following languages match most of the criteria: C++, Java and Python. The main weakness of C++ and Python is the availability of suitable third-party graph modelling libraries and, in the case of C++, programmer ability as Java and Python are the main languages taught at the School of Computing at the University of Leeds. All languages are extremely versatile, where Python would allow for a higher-level experience compared to Java and C++. Furthermore, the three candidate languages are object-oriented. Therefore, given the above, the programming language which matches the set criteria the most is Java. The implementation of the three graph algorithms will be carried out in Java.

2.4.2 Graph Modelling Libraries

A graph modelling library is required to facilitate the creation and visualisation of graphs and also to perform complex operations on graphs which would otherwise have to be programmed from scratch. As mentioned in Section 2.3.3 the main focus of the project is on the final algorithm produced, where the implementation of the three graph algorithms aid in the process of understanding the problem. The aim is to chose a free, stable, versatile and feature-rich library to make the process of graph modelling as efficient and effective as possible. The main graph modelling libraries for the chosen programming language, namely Java, are listed hereunder:

- yFiles[31]
  yFiles is an extensive Java class library that provides algorithms and components enabling the analysis, visualization, and the automatic layout of graphs, diagrams, and networks. yFiles also has a large amount of documentation available. However, yFiles is not free. In addition, the naming conventions used are not userfriendly.
LEDA [19]
LEDA is a C++ class library for efficient data types and algorithms. LEDA provides algorithmic in-depth knowledge in the field of graph- and network problems, geometric computations, combinatorial optimization and other. LEDA is implemented following the object-oriented approach. However, not only is LEDA not free (the more expensive option), but there are no Java bindings available which immediately exclude it from consideration.

uDraw [28]
uDraw (formally daVinci) is a graph modelling suite written in Java. It can either be used as a standalone application or can be integrated using its API. Although free, this suite is not specifically a Java class library and would therefore incur an unnecessary level complexity.

JGraphT [16, 15]
JGraphT is a free Java graph library that provides mathematical graph-theoretic objects and algorithms. JGraphT supports various types of graphs including directed, undirected graphs, graphs with weighted/ unweighted/ labeled or any user-defined edges. In addition Listenable graphs allow external listeners to track modification events. JGraphT is also designed to be simple and type-safe (via Java 5 generics).

The JGraphT API meets the selection criteria. Additionally, JGraphT can be easily integrated with another third-party library, namely the JGraph library, which enables visualisation and user interaction with the graph model. As described, the other libraries fail to meet the selection criteria.
Chapter 3

Graph Algorithm Implementations

3.1 Chordal Graph Recognition Algorithm

3.1.1 Analysis of Existing Algorithms

Fulkerson and Gross [8] provide an algorithmic characterisation of chordal graphs given as follows:

Theorem 3.1.1 Let $G$ be an undirected graph. The following statements are equivalent:

1. $G$ is chordal.
2. $G$ has a simplicial elimination ordering, and any simplicial vertex can start an elimination ordering.
3. Every minimal vertex separator induces a complete subgraph of $G$

Furthermore, this theorem makes use of the following lemma given by Dirac [7]:

Lemma 3.1.2 Every chordal graph $G$ has a simplicial vertex. Moreover, if $G$ is not a clique, then it has two nonadjacent simplicial vertices.

From the above, it can be deduced that, when constructing a simplicial elimination ordering, for each position in the ordering there is a choice of two simplicial vertices in a chordal graph. Leuker [20]
and Rose and Tarjan [24] both present an algorithm which uses the above to recognise chordal graphs in linear time. However, a collaboration between the three [25] lead to another algorithm which runs in time $O(|V| + |E|)$, where $V$ is the set of vertices and $E$ the set of edges in $G$ respectively. This algorithm, called the lexicographic breadth first search (or LBFS for short), keeps a list of vertices in lexicographic order; that is a dictionary order where, for example, 3871 < 396 and 453 < 4531. This is used to construct a simplicial elimination ordering for proving whether the graph is chordal or not.

There exists another algorithm developed by Tarjan [27] and presented in an unpublished set of notes called the maximum cardinality search (or MCS for short). Refer to Golumbic p87 [12] for a brief description. However, because the interval graph recognition algorithm also makes use of the LBFS algorithm, it is LBFS that will be implemented as opposed to MCS.

### 3.1.2 Implementation of Algorithm

The graph algorithm to be implemented for the recognition of chordal graphs is the lexicographic breadth first search. For the sake of completeness the algorithm has been reproduced below (this version has been taken from Golumbic [12]).

**Algorithm 1 Lexicographic Breadth First Search($G$)**

begin
1: assign the label $\emptyset$ to each vertex
2: for $i \leftarrow n$ to 1 do
3: pick an unnumbered vertex $v$ with largest label; \quad $\sigma(i) \leftarrow v$;
5: for each unnumbered vertex $w \in \text{Adj}(v)$ do
6: add $i$ to label($w$);
7: end for
8: end for
end

The above algorithm takes as input a graph $G$ and any vertex $v \in V$ and generates a simplicial elimination ordering. However, it is still necessary to verify the ordering to ascertain whether the $G$ is indeed chordal or not. This procedure is carried out by another algorithm which takes as its input an ordering $\sigma$ of $V$ and outputs "true" if $\sigma$ is a simplicial elimination ordering or "false" otherwise. This algorithm, also taken from Golumbic [12], is reproduced below.
Algorithm 2 Perfect(σ)
begin
1: for all $v \in V$ do
2: $A(v) \leftarrow \emptyset$;
3: end for
4: for $i \leftarrow 1$ to $n - 1$ do
5: $v \leftarrow \sigma(i)$;
6: $X \leftarrow \{x \in \text{Adj}(v) | \sigma^{-1}(v) < \sigma^{-1}(x)\}$;
7: if $X = \emptyset$ then
8: go to line 8
9: end if
10: $u \leftarrow \sigma(\min\{\sigma^{-1}(x) | x \in X\})$;
11: concatenate $X - \{u\}$ to $A(u)$;
12: if $A(v) - \text{Adj}(u) \neq \emptyset$ then
13: return "false"
14: end if
15: end for
16: return "true"
end

Before any code was written careful consideration was given to the design of the application. The main issues included the:

- Logical structure of the program.
- Data structures to be used when operating internally on the graph.
- Type of graph visualisation to use.
- Type of user interaction.

It was decided that the application would be built as a Java applet, given that in this form the program is self-contained and can be easily distributed, via the web or otherwise. Java applets have a method, `init()`, which is automatically called when the applet is run. It is in `init()` that the LBFS algorithm will be placed, to be run when the `mouseClicked()` event is triggered. All other methods will be called from `init()`, namely `positionVertexAt()` and `perfect()` which position a vertex at a particular point on the frame and performs the verification of the ordering produced respectively.
When the button is clicked, the algorithm is run on the graph created by the user. If the graph is chordal, then the applet returns a simplicial elimination ordering of the graph to the user via a JOptionPane. If the graph is not chordal however, the user is notified of this. It should be noted that the applet prints out useful information if run from a terminal or command prompt.

Given that JGraphT and would provide the necessary data structures for representing graphs internally, the only issue was how to efficiently store these to effectively operate on them. It was decided that the collections classes provided by Java, such as ArrayLists, Vectors, Sets and Maps, were to be used over custom data structures written from scratch. This decision was made on the basis of the wealth of efficient methods already available for these data structures. Also, in terms of graph visualisation, the applet would contain a blank, white frame where the user can create a graph through defined interactions, namely a right – click on the frame to create a vertex and a left – click on the frame to specify a pair of edges to join. Furthermore, a button at the bottom of the frame would, when clicked, perform the LBFS algorithm on the graph created. The screenshot above illustrates the look and feel of the applet.
3.1.3 Testing of Algorithm

A thorough testing strategy was designed to check whether the implementation properly recognises chordal graphs. For this, a selection of graphs was prepared. This selection included chordal graphs and non-chordal graphs alike. Please refer to Appendix B for screenshots illustrating:

- the output when the graph is chordal
- the output when the graph is not chordal
- the output printed to the terminal

The algorithm returned the correct output in every test case. The applet is included on the CD attached to this report. The source code is also available to be checked.

3.2 Interval Graph Recognition Algorithm

3.2.1 Analysis of Existing Algorithms

The majority of interval graph recognition algorithms are based on the following characterisation presented by Gilmore and Hoffman [11]:

**Theorem 3.2.1** A graph is an interval graph if and only if its maximal cliques can be linearly ordered in such a way that for every vertex in the graph the maximal cliques to which it belongs occur consecutively in the linear order.

Such implementations run in $O(|V|^3)$ if not cleverly structured. Booth and Leuker [4], however, presented an algorithm which employs $PQ\text{-trees}$ to produce the linear order on the set of maximal cliques. PQ-trees are complicated data structures and therefore make the algorithms which employ them increasingly complex (refer to Hsu et al. [14] for an in-depth discussion of PQ-trees).

An attempt was made to improve on Booth and Leuker’s work by Korte and Möhring [18], where they introduce a close variant of PQ-trees which they call $MPQ\text{-trees}$. Thereafter, Corneil et al. [6] and Habib et al. [13] have developed algorithms which don’t depend on PQ-trees, but instead, make use of the LBFS algorithm. The former runs four sweeps of the LBFS algorithm, whereas the latter uses LBFS to create a clique tree of the chordal graph and then manipulate the clique tree to form a cliquepath if
the graph is interval). The algorithm chosen to be implemented is the one given by Corneil et al. This is due mainly because a modified version of the LBFS algorithm (see Section 3.1) can be used.

### 3.2.2 Implementation of Algorithm

The interval graph recognition algorithm to be implemented is the four-sweep algorithm based on LBFS presented by Corneil et al. [6]. This algorithm is based on the following theorem given by Olariu [23].

**Theorem 3.2.2** A graph \( G \) is an interval graph if and only if there exists a linear order \( \prec \)
on the set of vertices such that for every choice of vertices \( u, v, w \), with \( u \prec v \) and \( v \prec w \) then \( uv \in E \) implies \( uw \in E \)

On the first sweep, LBFS is used (see Section 3.1.2), where in line 3, an arbitrary vertex \( v \) with the largest label is selected. However, there might be various vertices with the largest label. The three subsequent sweeps deterministically select this vertex. Firstly, there are two different variants of LBFS. The first variant of LBFS, \( \text{LBFS}^+ \), requires a previous sweep of LBFS. In this case, in line 3 vertex \( v \) is chosen to be the last vertex in the previous sweep; that is, if in line 3 there are various tied vertices, then the one which appears last in the previous sweep is chosen. Two runs of \( \text{LBFS}^+ \) are performed. The second variant of LBFS, \( \text{LBFS}^* \), requires two previous LBFS runs (namely, the two previous \( \text{LBFS}^+ \) sweeps). This variant however, chooses two vertices in line 3. The vertices chosen are the last vertex in the first (of the two previous runs) sweep and the last vertex in the second sweep. To select a vertex from the two it is necessary to introduce some terminology. Below is the explanation presented in [6]:

"Call the set of tied vertices [in line 3] \( S \)... We say that vertex \( x \) in \( S \) flies (F) if there is a vertex \( y \) such that \( y \) occurs after \( S \) and \( xy \in E \). Further, \( x \) is said to neighbour fly (NF) if it does not fly but has a neighbour in \( S \) that flies. Finally, vertex \( x \) is said to be OK if it neither flies nor neighbour flies. \( \text{LBFS}^* \) chooses between the two [selected] vertices, \( a \) and \( b \), by referring to the following table."

Thus, the implementation will be composed of three different algorithms to perform the recognition of the graph, namely LBFS, \( \text{LBFS}^+ \) and \( \text{LBFS}^* \). A first sweep of LBFS will be made, followed by two sweeps of \( \text{LBFS}^+ \) followed finally by a single sweep of \( \text{LBFS}^* \). This implementation follows the same design principles as the chordal graph recognition algorithm (see Section 3.1.2) except for the modularisation of the three different algorithms. Finally, the \text{perfect}() algorithm will verify the
ordering to make sure the graph is an interval graph. The screenshot above illustrates the look and feel of the applet.

Figure 3.2: Interval Graph Recognition Applet
3.2.3 Testing of Algorithm

A thorough testing strategy was designed to check whether the implementation properly recognises interval graphs. For this, a selection of graphs was prepared. This selection included interval graphs and non-interval graphs alike. Please refer to Appendix C for screenshots illustrating:

- the output when the graph is interval
- the output when the graph is not interval
- the output printed to the terminal

The algorithm returned the correct output in every test case. The applet is included on the CD attached to this report. The source code is also available to be checked.

3.3 Clique Tree Construction Algorithm

3.3.1 Analysis of Existing Algorithms

Clique trees have various characterisations although, as mentioned in Section 2.1.5, the maximum-weight spanning tree definition will be assumed for the purpose of the project. Thus, it is necessary, given a chordal graph \( G \), to extract a clique tree. Blair et al. [3] discusses various algorithms for doing so.

Prim’s maximum-weight (minimum-weight) spanning tree algorithm is an example of a classic algorithm being employed for this purpose. It assumes that a weighted clique graph has been created from the chordal graph as it requires weighted edges in order to function. Prim’s algorithm works my creating a subtree of a clique tree and slowly adding edges (representing cliques) until the subtree becomes the clique tree.

Another algorithm used for generating a clique tree is maximum cardinality search (MCS) briefly discussed in Section 3.1.1. As MCS creates a simplicial elimination ordering of a chordal graph, then it can easily detect the cliques of the graph in the process. However, Prim’s algorithm will be chosen for the implementation because of its simplicity.
3.3.2 Implementation of Algorithm

Prim’s maximum-weight spanning tree algorithm requires a weighted clique graph of the chordal graph to be created prior to being run. Thus, the implementation will be a three-part process. The first part will take as input the graph created by the user and check whether it’s a chordal graph using the LBFS. If the graph is indeed a chordal graph then the second part will create from it a weighted clique graph, by representing each maximal clique by a vertex and connecting a pair of vertices if the two maximal cliques they represent intersect. In addition each edge is assigned a weight given by the number of vertices in the intersection of the two cliques. Finally, the third part will perform Prim’s maximum-weight spanning tree algorithm on the weighted clique graph to create a clique tree of the chordal graph.

The algorithm as presented by Blair et al. [3] has, for the sake of completeness, been reproduced below.

Note the following notation: $W_G$ represents the weighted clique graph, $E_T$ represents the set of edges making up the weighted clique graph, $\hat{K}$ represents the set of cliques in the subtree created thus far.

\begin{algorithm}
\textbf{Algorithm 3} Maximum-Weight Spanning Tree($W_G$)
\begin{algorithmic}
\State $E_T \leftarrow \emptyset$;
\State choose $K \in W_G$; $\hat{K} \leftarrow \{K\}$;
\For {$r \leftarrow$ to $m$}
\State choose cliques $K \in \hat{K}$ and $K' \in W_G - \hat{K}$ for which $|K \cup K'|$ is maximum;
\State $E_T \leftarrow E_T \cup \{(K, K')\}$;
\State $\hat{K} \leftarrow \hat{K} \cup \{K'\}$;
\EndFor
\end{algorithmic}
\end{algorithm}

The design issues involved in this implementation were different from the previous two algorithms. In this case a graph is to be returned as opposed to a simple boolean "true" or "false" answer. Therefore, it was decided to have the same applet setup as the previous two implementations, where the user is allowed to create a graph on the frame. However, once the graph is created, if it’s chordal another applet window will pop-up with the clique tree drawn. The only issue is the fact that when the frame with the clique tree is shown, the vertices of the clique tree are not positioned in any way; that is, the clique tree is not drawn appropriately and user intervention is required in order to move the vertices and view the clique graph properly. This, however, is a problem which hasn’t been rectified as it was decided that continuing with the rest of the project was a better use of time. After all this implementation forms part
of the minimum requirements as a means of better understanding the problem and not as an end-product which forms part of the aim of the project. The following two screenshots (see next page also) illustrate the look and feel of the applet and the issue with the drawing of the clique tree.

Figure 3.3: Clique Tree Construction Algorithm with drawn Clique Tree
3.3.3 Testing of Algorithm

A thorough testing strategy was designed to check whether the implementation properly recognises chordal graphs. For this, a selection of graphs was prepared. This selection included chordal graphs and non-chordal graphs alike. Please refer to Appendix D for screenshots illustrating:

- the output when the graph is chordal
- the output when the graph is not chordal
- the output printed to the terminal

The algorithm returned the correct output in every test case. The applet is included on the CD attached to this report. The source code is also available to be checked.
Chapter 4

Algorithm for Computing the Leafage of Chordal Graphs

4.1 Overview

This chapter presents the results of the research work conducted with the aim of developing an algorithm for computing the leafage of chordal graphs in polynomial time. Firstly, the design of the algorithm is discussed, where related graph-theoretic concepts are described. Also, the functioning of the algorithm is presented with a guided example of its operation. Lastly, the algorithm is proved correct and a time complexity analysis given.

4.2 Design of the Algorithm

Every chordal graph $G$ has a set of clique trees to which it corresponds. Furthermore, there is a correspondence between clique trees of $G$ and minimal subtree representations of the host tree $T$. Thus, the leafage, $l(G)$, is the minimum number of leaves in a clique tree. A naive approach to computing the leafage would be to count the leaves of each clique tree of $G$. Such an approach would run in order
$O(K^{K-2})$ as it would have to go through all maximum-weight spanning trees of the weighted clique graph of $G$ (see [30] p82).

However, consider the fact that each chordal graph already has its defined leafage; that is, the clique tree with the minimum number of leaves. If, for example, the leafage is three and four respectively, then the corresponding clique trees would be of the form illustrated in Figs 4.1 and 4.2 below.

From Fig 4.1 it can be observed that with leafage three there is usually a clique in the centre with three branches stemming from it. Leafage four increases the number of clique tree configurations of which Fig 4.2 is an example. However, the clique tree in Fig 4.2 can be modified, without affecting the leafage, into a star configuration, where all branches stem from a central clique. This idea can be generalised to say that every clique tree can be modified into a star configuration without affecting the leafage. This modification can be performed through \textit{contraction} of the path between the vertices (cliques) from...
which the branches stem. The design of the algorithm is centered around modifying chordal graphs in order to create such a star configuration.

Star configurations of clique trees have a single clique in the centre from which branches stem. These branches are paths of consecutive cliques. Thus, there must be at least one clique in a chordal graph which corresponds to the central clique in the star configuration, this clique is denoted $k_c$ (see Lemma 4.4.1). Moreover, if there are various of these cliques then they can be contracted as aforementioned. Also, as each branching in a clique tree is a path of consecutive cliques it is obvious that these are exactly the interval graphs (recall Theorem 3.2.1 from Section 3.2.1). This last observation could form part of verification after the process of contraction.

The algorithm therefore would perform as follows given a chordal graph $G$:

1. Search $G$ for $k_c$;
2. If there are more than one then choose one and contract between the pairs of candidate cliques;
3. Check that each branch forms an interval graph;
4. Count the number of branches of $k_c$ and return this as the leafage of $G$;

As an example consider the chordal graph in Fig 4.3. If one were to manually work out the maximum-weight spanning tree with the minimum number of leaves then they would end up with Fig 4.4. The subtlety is with the clique on the right in Fig 4.4. It can be seen that both edges can be chosen (as each have a weight of 1), but only one will lead to the minimum number of leaves. Thus, as the diagram shows the top edge is chosen over the bottom. Counting the number of leaves on the clique tree (the clique tree is marked with a red line) gives the leafage as three.

The algorithm would first recognise the shaded maximal cliques in Fig 4.3 as they have the same number of branches. Thereafter, the one chosen to be $k_c$ would be modified to cater for the branches of the other. Say the left-most clique is $k_c$, then it would be modified to have an extra vertex to which branch stemming from the other clique is attached. Finally, each branching of $k_c$ is checked to see if it’s an interval graph. In this case they are, so leafage three is returned.
The algorithm is presented in pseudocode in the following section and is divided into two parts for the sake of clarity. The first part locates the $k_c$ maximal clique and the second performs the necessary modifications and contractions between cliques. The algorithm is presented in a way which makes clear the type of implementation employed; that is, a less generic style is used. For example, the first part of the algorithm is performed on the clique matrix of the chordal graph (Recall Section 2.1.2).

### 4.3 Algorithm for Computing the Leafage of Chordal Graphs

The algorithm is presented in two parts. The first part locates the $k_c$ maximal clique in $G$. The second part uses this to perform the necessary contractions between cliques to produce the star configuration required.
ComputeLeafage() Part 1

- **Idea:** Locate the maximal clique, $k_c$, in $G$ which cuts the graph into the maximum number of components by searching the clique matrix for specific patterns.

- **Input:** Clique matrix representation of $G$.

- **Output:** The maximal clique $k_c$, if it exists or the leafage, $l(G)$, if $G$ is a star or $|K| \leq 2$.

**Algorithm 4 ComputeLeafage Part1($G$)**

begin
1: $k_c \leftarrow 0$; $cliqueStack \leftarrow 0$; $currentCounter \leftarrow 0$; $maxCounter \leftarrow 0$; $flag \leftarrow 0$;
2: if $|K| = 1$ then
3: return leafage 1; end ComputeLeafage();
4: end if
5: for each column in the clique matrix do
6: for each row in the clique matrix do
7: if cell = 1 and all cells in the row hold a 1 then
8: return leafage 2; end ComputeLeafage();
9: end if
10: if cell = 1 and there is another 1 on the row then
11: $currentCounter++$;
12: else
13: $flag \leftarrow 1$; break from inner for-loop;
14: end if
15: end for
16: if $flag = 1$ then
17: $flag \leftarrow 1$; go to next column in clique matrix;
18: end if
19: if $flag = 1$ then
20: if $currentCounter = 0$ then
21: continue outer for-loop;
22: end if
23: if $currentCounter > maxCounter$ then
24: $maxCounter \leftarrow currentCounter$; add clique to $cliqueStack$;
25: end if
26: if $currentCounter = maxCounter$ then
27: add clique to $cliqueStack$;
28: end if
29: end if
30: $currentCounter \leftarrow 0$;
31: end for
32: $k_c \leftarrow$ last clique added to the $cliqueStack$;
end
ComputeLeafage() Part 2

- **Idea:** If there are other maximal cliques in $G$ with the same number of branches as $k_c$, then, through a process of contraction, create a *clique star* from which the leafage can be computed directly.

- **Input:** Clique matrix representation of $G$ and all data structures from Part 1.

- **Output:** A clique star with the modified $k_c$ maximal clique as the root clique, with each vertex hosting a branch of cliques, and the leafage, which corresponds to the number of branches.
Algorithm 5 ComputeLeafage Part2\((G)\)

\begin{algorithm}
\textbf{begin}
\begin{align*}
1: \text{finalCliqueStack} & \gets 0; \text{leafage} \gets 0; \text{extraLeafageCounter} \gets 0; \\
2: \textbf{for all} \ k \in \text{cliqueStack} \ \textbf{do} \\
3: \quad \textbf{if} \ \text{the number of branches in} \ k \ \text{equals to those in} \ k_c \ \textbf{then} \\
4: \quad \text{finalCliqueStack} \gets k; \\
5: \quad \textbf{end if} \\
6: \textbf{end for} \\
7: \textbf{if} \ \text{finalCliqueStack} = \emptyset \ \textbf{then} \\
8: \quad \textbf{for all} \ \text{branches} \ s \ \text{stemming from} \ k_c \ \textbf{do} \\
9: \quad \quad \textbf{if} \ s \ \text{is not an interval graph} \ \textbf{then} \\
10: \quad \quad \quad \text{extraLeafageCounter} \gets (\text{extraLeafageCounter} + \text{ComputeLeafagePart1}(s)) - 1; \\
11: \quad \quad \quad \text{remove} \ s; \\
12: \quad \quad \quad \text{leafage}++; \\
13: \quad \quad \textbf{end if} \\
14: \quad \textbf{end for} \\
15: \quad \text{return} \ \text{leafage}; \ \text{end ComputeLeafage();} \\
17: \textbf{else} \\
18: \quad \text{finalCliqueStack} \gets k_c \\
19: \quad \textbf{while} \ k_c \ \text{is not the only clique with the largest number of branches in} \ \text{finalCliqueStack} \ \textbf{do} \\
20: \quad \quad \text{choose} \ a, b \in \text{finalCliqueStack} \ \text{where} \ \exists \ d \in \text{finalCliqueStack} : d \not\in a, b\text{-path}; \\
21: \quad \quad \text{compute the distances of each clique from} \ k_c, \ \text{let} \ a \ \text{be the closest and} \ b \ \text{the furthest;} \\
22: \quad \quad \textbf{for all} \ \text{branches} \ s \ \text{stemming from} \ b \ \textbf{do} \\
23: \quad \quad \quad \textbf{if} \ s \ \text{is not an interval graph} \ \textbf{then} \\
24: \quad \quad \quad \quad \\text{extraLeafageCounter} \gets (\text{extraLeafageCounter} + \text{ComputeLeafagePart1}(s)) - 1; \\
25: \quad \quad \quad \quad \text{remove} \ s; \\
26: \quad \quad \quad \text{end if} \\
27: \quad \quad \textbf{end for} \\
28: \quad \quad \textbf{if} \ \text{each clique is on a separate sub-branch} \ \textbf{then} \\
29: \quad \quad \quad \text{modify} \ a, \ \text{to have} \ |a| + (|b| - 1) \ \text{vertices}; \\
30: \quad \quad \quad \text{attach to the extra} \ |b| - 1 \ \text{vertices in} \ a \ \text{the branches of clique} \ b \ \text{which don’t include the clique} \\
31: \quad \quad \quad \text{path to} \ k_c; \\
32: \quad \quad \quad \text{remove clique} \ b \ \text{and the clique path upto but not including the vertex which joins both} \ a \ \text{’s} \\
33: \quad \quad \quad \text{and} \ b \ \text{’s clique path to} \ k_c; \\
34: \quad \quad \quad \textbf{else} \\
35: \quad \quad \quad \quad \text{modify} \ a, \ \text{to have} \ |a| + (|b| - 1) - 1 \ \text{vertices;} \\
36: \quad \quad \quad \quad \text{attach to the extra} \ |b| - 1 \ \text{vertices in} \ a \ \text{the branches of clique} \ b \ \text{which don’t include the clique} \\
37: \quad \quad \quad \quad \text{path to} \ a; \\
38: \quad \quad \quad \quad \text{remove clique} \ b \ \text{and the clique path upto clique} \ a; \\
39: \quad \quad \quad \textbf{end if} \\
40: \quad \quad \text{leafage} \gets |k_c|; \\
41: \quad \quad \text{leafage} \gets \text{leafage} + \text{extraLeafageCounter}; \\
42: \quad \text{Return} \ \text{leafage}; \\
43: \quad \textbf{end while} \\
44: \quad \textbf{end if} \\
\textbf{end}
\end{align*}
\end{algorithm}
4.4 Proof of Correctness

Lemma 4.4.1 Let $G$ be a chordal graph, connected but not complete, that is not a star and $|K| > 2$, then there is a maximal clique, $k_c$, in $G$ which, if removed, cuts the graph into the largest number of components.

Proof Assume that $G$ is a connected, chordal graph, where $G$ is not a star and $|K| > 2$. Suppose $G$ has no maximal clique, $k_c$, which cuts the graph into the largest number of components. Then there are at least a pair of cliques $u, v \in K$, with no path between them, because if such a path existed then removing it would cut the graph into the largest number of components. However, this contradicts the fact that $G$ is connected. Therefore, such a maximal clique does exist in $G$.

Theorem 4.4.2 Let $G$ be a chordal graph, connected but not complete, that is not a star and $|K| > 2$, Part 1 of the algorithm correctly locates the $k_c$ maximal clique in $G$ which, if removed, cuts $G$, into the maximum number of components.

Proof By Lemma 0.1, the situation where there is no $k_c$ maximal clique, $G$ is not a star and $|K| \leq 2$ cannot occur. By lines 2, 3, 8 and 9, if $G$ is a star or $|K| \leq 2$, the algorithm will return the leafage directly. Assuming that $G$ has at least one maximal clique, then the algorithm examines the graph’s clique matrix, placing cliques on the cliqueStack if and only if it has more branches than the previous clique.

Thus, when the algorithm terminates, the last clique pushed onto the cliqueStack will be the maximal clique with the largest number of branches in $G$. Suppose that this wasn’t the case. Then there would be another maximal clique in $G$ with a larger number of branches. However, on terminating, the algorithm has gone through all cliques in $G$ and such a clique would have been recognised and pushed onto the cliqueStack. This contradicts our assumption and so the last clique on the cliqueStack is the clique with the largest number of branches in $G$.

Theorem 4.4.3 Let $G$ be a chordal graph, connected but not complete, that is not a star and $|K| > 2$. Assume the data structures used in Part 1, Part 2 of the algorithm correctly constructs, through contraction, a clique star from which the leafage, $l(G)$, can be easily computed.

Proof Assuming that $G$ is a connected, chordal graph, where $G$ is not a star and $|K| > 2$, then from Part 1, $G$ can either have a single $k_c$ maximal clique with no other maximal clique having the same number
of branches, or a single $k_c$ maximal clique and various other maximal cliques with the same number of branches as $k_c$.

If the latter is the case, then all cliques with the same number of branches as $k_c$, including $k_c$, are pushed onto the finalCliqueStack. The algorithm proceeds to choose pairs of cliques from the finalCliqueStack and contracting as required until $k_c$ is the only clique with the largest number of branches. At each step in the loop the branches of the clique furthest from $k_c$ is checked if they form an interval graph. If not the algorithm recursively calls itself to compute the leafage of that branch and removes the branch from $G$.

**Claim 4.4.4** On termination, each branch stemming from $k_c$ is an interval graph. Also, the leafage incurred by sub-branches of $G$ which are not interval graphs is correctly computed.

**Proof** Suppose, when the algorithm terminates, there is a branch stemming from $k_c$ which is not an interval graph. Then, at some point a pair of cliques were selected, with the clique furthest from $k_c$ having a non-interval graph attached to one of it’s vertices. However, by lines 19 and 20 such a branching would have been recognised and subsequently removed. This contradicts the assumption. Hence, when the algorithm terminates, all branches stemming from $k_c$ are interval graphs.

However, in line 19, if a particular branch is not recognised as being an interval graph (i.e. it is chordal), then the algorithm is called recursively to operate on the given branch. The whole process starts again, however when the recursive call terminates it returns the leafage of the given branch to the parent instance of the algorithm; that is, the leafage of the branch minus 1, as the branch is actually connected to the rest of the graph. This value is stored in extraLeafageCounter and is updated whenever a non-interval graph sub-branch is encountered. When the algorithm terminates, the extraLeafageCounter is added to the leafage to give the overall leafage for $G$.

**Claim 4.4.5** The leafage, $l(G)$, is equal to the number of branches stemming from $G$.

**Proof** Once all the candidate maximal cliques have been contracted and their branches attached to the modified $k_c$, then $G$ will form a star configuration with $k_c$ as the central clique. This corresponds to the clique tree of $G$ with the minimum leaves because of the correlation between $G$ and its clique tree. Thus, the number of branches is equal to the leafage. Suppose this is not the case. Then, there would
still be another candidate maximal clique in $G$. However, such a clique would have been recognised by line 19 as it would be in the $finalCliqueStack$. This contradicts our assumption. Hence, the number of branches stemming from $k_c$ at the end of the algorithm corresponds to the leafage.

4.5 Time Complexity Analysis

The time complexity analysis of each part of the algorithm will be carried out.

Part 1

In the first part of the algorithm, lines 5 and 6 dominate the rate of growth as they contain the nested for-loop which for every clique in the graph, checks every vertex. Thus, in the worst-case scenario this has a running time of $O(|K| \times |V|)$ where $K$ is the set of cliques and $V$ the set of vertices.

Thus, part 1 of the algorithm runs in polynomial time.

Part 2

In the second part of the algorithm, lines 19 and 22 dominate the rate of growth in the worst-case scenario. In this case, in the worst case all cliques could be candidate cliques in line 19 so this would need to run $|K|$ times. Each time two cliques would be chosen, where the clique furthest from $k_c$ will have $|B|$ branches on which the algorithm calls itself recursively. Thus, the running time is $O((|K| \times |B|^{|K|})$. This running time, although not very efficient is polynomial in time.

Thus, the total running time is indeed polynomial in time.
Chapter 5

Evaluation

5.1 Overview

This evaluation aims to measure the overall success of the project. Certain criteria is required in order to properly evaluate the solution against the initial problem. These are defined and justified in their relevant section.

The aim of the project was to develop an algorithm which computes the leafage of chordal graphs in polynomial time, to prove its correctness and to analyse its time complexity. Furthermore, in addition to the project aim, several other minimum requirements were set with the purpose of aiding in the understanding of the problem.

5.2 Evaluation of the Graph Algorithm Implementations

The criteria chosen to evaluate the three graph algorithm implementations is both the test strategy (see Sections 3.1.3, 3.2.3 and 3.3.3) and how much the process increased my understanding of the problem. The former evaluates the quality of the implementation from a software development perspective. The latter evaluates the purpose of the implementation.
From the test strategy sections and screenshots in the appendices it is clear that the graph algorithm implementations have been successful. Moreover, the whole implementation process, which lasted three months, provided invaluable insight into the internals of the actual algorithms. Furthermore, the implementations required extensive background research which also aided in developing a solid foundation of knowledge for the theoretical research work. There is only one aspect of the implementation which was not up to standard, this is the drawing of the clique tree on the frame. It requires the user to move the vertices around the frame to make the graph easy to see. This, however, wasn’t a priority and was disregarded, although perhaps future work could include software upgrades.

The implementation of the algorithms also added a software engineering aspect to the project, which otherwise would have been purely theoretical. This provided for a good balance between theory and practice. Overall, therefore, it seems that the graph algorithm implementations were a success.

5.3 Evaluation of the Algorithm for Computing the Leafage of Chordal Graphs

The developed algorithm will be measured against a naive algorithmic solution to the problem, and also against the minimum requirements.

The minimum requirements stated that the algorithm had to run in polynomial time. By the time complexity analysis (see Section 4.3) this is true. A naive approach to the problem would be to generate every clique tree for a chordal graph and count the number of leaves for each. Such an approach would run in order $O(K^{K-2})$ as it would have to go through all maximum-weight spanning trees of the weighted clique graph of the chordal graph. The developed algorithm runs, in the worst case, in order $O((|K| \times |B|)^{|K|})$ which is theoretically worse than the naive approach. However, the naive algorithm has to go through all clique trees every time, and so has a running time of $O(K^{K-2})$. This means that in the majority of cases the developed algorithm out-performs the naive approach.

Additionally, it should be mentioned that the developed algorithm is the first polynomial time algorithm to compute the leafage of a chordal graph. It is hoped that more work will be done on this in the future.
so as to increase the efficiency of the algorithm. Overall, the developed algorithm has been more of a success than a failure.
Bibliography


Appendix A

When deciding on the project topic it was clear to me that I wanted to undertake a theoretical project, namely one involving graph theory. This is mainly because of the great love for the subject and the vocation to be involved in research relevant to my field of study. I have thoroughly enjoyed the experience of being able to solve an open problem in my subject and also of working with leaders in their fields. There have been good and bad times throughout the project lifetime. Most notably, I have at times felt a lack of inspiration which has made it difficult to work on a problem, especially when conducting the theoretical research. Many dead-ends lowered my morale at times, and this impeded me from giving my best. By the same token, carrying out research was the most satisfying experience, as when results are achieved there is a huge self satisfaction.

The worst part of the project was when, nearing the end, when I thought all was well and done, I came accross what I thought were a few counter examples to my correctness proofs which took me quite a while to prove otherwise. This was very annoying as I felt all the work I had done was for nothing. There is no way that these problems can be avoided as this is the nature of theoretical reseach. One can work forever on some project which is later proved totally incorrect. But this doesn’t mean that the the work done is useless and fruitless, quite the contrary. Personally, I have no regrets whatsoever about the project. Even if I would have found a counter example, or indeed if my proofs are broken at a later date I wouldn’t have changed anything as I have learned invaluable lessons as a consequence of the project.
Appendix B

Figure 5.1: Chordal Graph Recognition Algorithm Recognising a Chordal Graph
Figure 5.2: Chordal Graph Recognition Algorithm Rejecting a Chordal Graph

Figure 5.3: Chordal Graph Recognition Algorithm Output to Terminal
Appendix C

Figure 5.4: Chordal Graph Recognition Algorithm Recognising a Chordal Graph
Figure 5.5: Chordal Graph Recognition Algorithm Rejecting a Chordal Graph

Figure 5.6: Chordal Graph Recognition Algorithm Output to Terminal
Figure 5.7: Clique Construction Algorithm Generating a Clique Tree
Figure 5.8: Clique Construction Algorithm Rejecting a Chordal Graph
Figure 5.9: Clique Construction Algorithm Output to Terminal