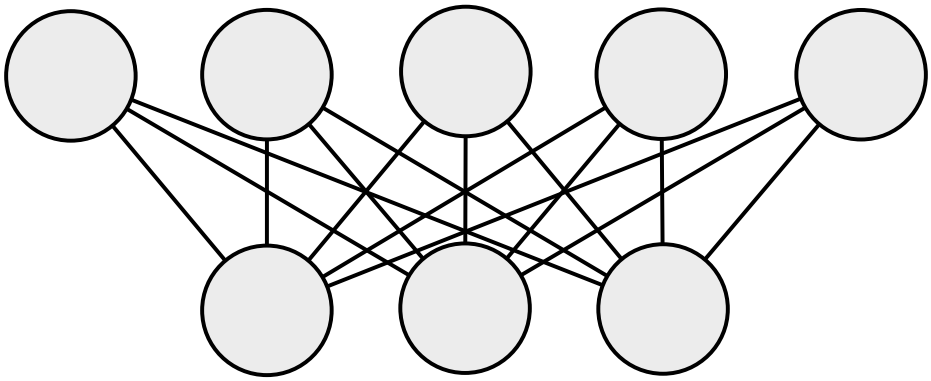
Leading Edge Triangulation of Bayesian Networks

DAT3, FALL SEMESTER 2010
GROUP D306A
DEPARTMENT OF COMPUTER SCIENCE
AALBORG UNIVERSITY

18TH OF DECEMBER 2010



Title: Leading Edge Triangulation of Bayesian Networks
Theme: Modelling
Project period: Dat3, fall semester 2010
Project group: d306a
Group members:

Jonas Finnemann Jensen

Lars Kærlund Østergaard

Filipe Emanuel dos Santos Albuquerque Barroso

Jeppe Thaarup

Rasmus Emma Hassig Schøn

Supervisor: Ricardo Gomes Lage

Special thanks to: Thorsten Jørgen Ottosen, for assistance and inspiration.

Abstract:

In this report we propose two improvements to reduce the search space for state-of-the-art optimal triangulation algorithms, with respect to the total table size criterion.
The improvements, we propose, exploit properties of triangulated graphs and can be applied to any triangulation algorithm, searching in the space of all elimination orders.
This report also covers the basis for inference in Bayesian networks and introduces the problem of triangulation. We examine heuristics, minimal and optimal methods for solving this problem.
Finally, we compare the methods discussed and show that it is possible to achieve considerable improvements in the efficiency of optimal methods.

Copies: 9
Number of pages: 77
Appendices: 3
Completion date: 17th of December, 2010

Preface

For readers with a basic understanding for Bayesian networks and how this relates to the problems of triangulation, chapters 6-9 will probably be most interesting, as this is where our contributions and work is presented. An efficient C++ implementation of the algorithms presented in this report should accompany this report on a CD, and be available for download at `http://jopsen.dk/blog/2010/12/triangulation-project/` along with a digital version of this report.

Figures and tables are enumerated in the same fashion after what number the given figure or table is in the current chapter. e.g x.y where x is the chapter and y is the number of the figure in the chapter, so the third figure in chapter 2 would have the number 2.3.

Definitions, theorems, corollary are enumerated after what number they are in the report as a whole, e.g. definition 20 will also be the 20th definition in the report and corollary 2 will be the 2nd corollary in the report.

Algorithms, or pseudo code, are enumerated like definitions. These are written in their own environment with a headline of what algorithm it is what it is called and then the pseudo code is written on enumerated lines.

All references to the bibliography, citation, are written in parentheses; internal references in the report are just by number with no parentheses.

The following gives an overview of the chapters in this report.

Chapter 1 contains the project description and problem statement.
Chapter 2 contains basic theory of Bayesian networks along with the definition and idea of triangulation.
Chapter 3 contains a presentation of minimal methods and their pseudo-code implementation.
Chapter 4 contains a presentation of greedy heuristic methods and their pseudo-code implementation.
Chapter 5 contains a presentation of the basic optimal methods and their pseudo-code implementation.
Chapter 6 introduces an optimization technique for optimal methods by reducing expansions using pivot cliques.
Chapter 7 introduces an optimization technique for optimal methods by predicting coaliscing by using transposition of perfect elemination orders.
Chapter 8 introduces an optimization technique for optimal methods by maximal prime subgraph decomposition.
Chapter 9 introduces an optimization technique for optimal methods by reducing expansions using graph symmetry.
Chapter 10 contains a comparison of the methods and their different optimizations.
Chapter 11 contains a discussion of the previous chapters, along with future work.
Chapter 12 contains the conclusion of the problem statement.

Contents

Contents i

Figures iii

1 Introduction 1

2 Bayesian Networks and the Problem of Triangulation 3
- 2.1 Tools from Probability Theory . . . 3
- 2.2 Bayesian Networks . . . 5
  - 2.2.1 Inference in Bayesian Networks . . . 5
  - 2.2.2 The Chain Rule for Bayesian Networks . . . 6
  - 2.2.3 Moral Graph . . . 7
  - 2.2.4 Elimination . . . 8
- 2.3 Triangulation . . . 8
  - 2.3.1 Triangulated graphs . . . 11
  - 2.3.2 Minimum, Minimal and Optimal Triangulations . . . 11

3 Minimal Methods 13
- 3.1 LB-Triang . . . 13
- 3.2 Maximal Cardinality Search (MCS-M) . . . 14
- 3.3 Recursive Thinning . . . 17

4 Greedy Heuristic Methods 19
- 4.1 Generic Greedy Algorithm . . . 19
  - 4.1.1 Min-fill . . . 20
  - 4.1.2 Min-width . . . 20
  - 4.1.3 Min-weight . . . 21

5 Searching for Optimal Solutions 23
- 5.1 Optimal Search Algorithms . . . 23
- 5.2 Clique Maintenance . . . 24
  - 5.2.1 Finding Maximal Cliques . . . 24
  - 5.2.2 Finding New Maximal Cliques After Adding/Removing Edges . . . 25
  - 5.2.3 Incremental Update . . . 26
- 5.3 Best First Search for Optimal Triangulations . . . 27
- 5.4 Depth-First Search . . . 27

6 Reducing Expansions with Pivot Cliques 31
- 6.1 The Pivot Clique Selection Algorithm . . . 32
- 6.2 Pivot Selection Criteria . . . 33
- 6.3 Evaluation of the pivot strategies . . . 35

7 Coalescence Prediction using Transposition of PEOs **39**
7.1 The Transposition Oracle for PEOs 39
7.2 Coalescence Prediction 40
7.3 Best First Search with Oracle 41
7.4 Summary 43

8 Maximal Prime Subgraph Decomposition **45**
8.1 Finding Decompositions 45
8.2 Exploiting Decomposition 48
8.3 Best First Search with Maximal Prime Subgraph Decomposition 48

9 Reducing Expansion Using Graph Symmetry **51**
9.1 Defining Node Equivalence 51
9.2 Finding Node Equivalence 52
9.3 Conclusion on Exploitation of Graph Symmetry 52

10 Comparison of Triangulation Methods **55**
10.1 Minimal Methods 55
10.2 Greedy Heuristics 56
10.3 Optimal 56

11 Discussion **61**
11.1 Future Work 62

12 Conclusion **63**

A Best First Search with Decomposition, Coalescence Prediction and Pivot **65**

B Comparison of Pivot Selection Strategies **67**

C Implementation **75**
C.1 Overview 75

Bibliography **76**

List of Figures

1 The diagonal is a fill edge. . . . v
2 The nodes $\{E,F,C\}$ is a connected component . . . v

2.1 A bayesian network . . . 5
2.2 Different kinds of connections in Bayesian networks. . . . 6
2.3 Graph (b) is the moralized undirected version of graph (a) . . . 7
2.4 The same domain graph, but in (b) X has been eliminated . . . 8
2.5 A non-minimal triangulation produced by the elimination ordering C,B,D,A . . . 10
2.6 A clique . . . 11

3.1 A minimal triangulation produced by the LB-Triang algorithm, where $\alpha = \{A,B,C,D,E\}$ 14
3.2 The MCS-M algorithm run on the example graph. Numbers in parentheses denote $w(u)$. Dark grey represents a numbered node, with corresponding number written below, and light grey points out that the weight for a given node is incremented. This example produces the minimal eliminaton ordering $\alpha = (E,D,C,B,A)$. (pg.292 Berry et al., 2004, fig.5) . . . 16
3.3 A minimal triangulation produced by the elimination order starting with B . . . 17
3.4 A non-minimal triangulation produced by the elimination order C,B,D,A . . . 18

5.1 The search tree for all elimination orders of a graph with 3 nodes. . . . 23

6.1 Selecting the clique with the largest intersection. . . . 33

7.1 The search tree with oracle coalesce prediction. . . . 41

8.1 Graph with simplicial nodes that admits decomposition. . . . 47

9.1 A graph with symmetry. . . . 51

10.1 Graphs showing the greedy heuristic algorithms and their tts on graphs with 20 nodes (top) and 30 nodes (bottom), including the optimal solution (bfs/dfs) for comparison. . . . 58

C.1 The graph G . . . 75

Notation

Graph:

Let $G = (V, E)$ be an undirected graph, consisting of a set of nodes V and a set of edges E. The nodes of a graph are given by $\mathcal{V}(G)$ and the edges of a graph are defined by $\mathcal{E}(G)$.

Fill edges:

Fill edges are the edges added during a triangulation process. In the triangulation $G' = (V, E \cup T)$, the fill edges T are added to the set of edges. In figures, fill edges are illustrated as dashed lines, e.g. in figure 1 the diagonal is a fill edge.

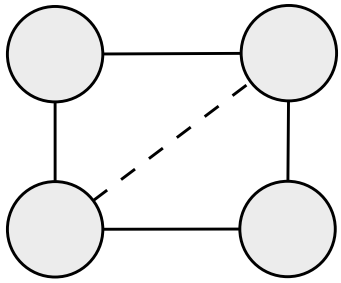


Figure 1: The diagonal is a fill edge.

Neighbours:

$nb(x, G)$ denotes the set of neighbours of a given node $x \in \mathcal{V}(G)$. Likewise, $nb(S, G)$ denotes the set of neighbours of the set $S \subseteq \mathcal{V}(G)$.

Family:

$fa(x, G)$ yields the family of node $x \in \mathcal{V}(G)$, which is $nb(x, G) \cup x$. Similarly, $fa(S, G)$ contains the family of all nodes in this set S, where $S \subseteq \mathcal{V}(G)$. More precisely, the family of a set of nodes is $\bigcup_{s_i \in S} nb(s_i, G) \cup s_i$.

Subgraph:

For a set of nodes $W \subseteq \mathcal{V}(G)$, the subgraph induced by W is $G[W] = (W, \mathcal{E}(W))$, where $\mathcal{E}(W) = (W \times W) \cap \mathcal{E}(G)$.

Connected component:

A subgraph in which nodes are connected.

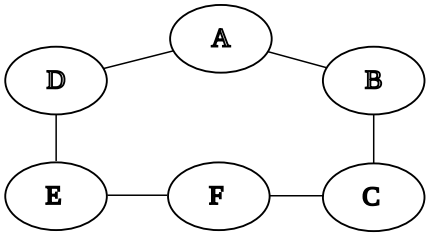


Figure 2: The nodes $\{E, F, C\}$ is a connected component

Clique:

A clique C is a set of nodes, s.t. $C \subseteq \mathcal{V}(G)$ and there is an edge between each distinct pair of nodes from C, i.e. $G[C]$ is a complete subgraph. All maximal cliques of G are denoted by $\mathcal{C}(G)$.

Minimal seperator:

Given a graph $G = (V,E)$, a subset $S \subset \mathcal{V}(G)$ is a *seperator* of the graph if and only if $V(G) \setminus S$ is not connected. S is an (a,b)-*seperator* if and only if $\exists (a,b) \in G$ where the nodes a and b is in different connected components of $\mathcal{V}(G) \setminus S$. If there is no proper subset of S that is also an (a,b)-seperator then S is a minimal (a,b)-seperator.(Berry et al., 2010)

Density:

The density of a graph $G = (V,E)$ is defined as

$$D = \frac{2|\mathcal{E}(G)|}{|\mathcal{V}(G)| \cdot (|\mathcal{V}(G)| - 1)} \tag{1}$$

.

Probability of a variable For each variable, X, there is a associated probability, $P(X = x)$, where $P \in [0;1]$, denoting the probability that X will be in a certain state, x. This will be denoted as $P(X)$.

CHAPTER 1

Introduction

Reasoning under uncertainty is a task which applies to many domains, such machine intelligence, medicine, manufacturing, finance and agriculture. Typically, one may be interested in determining the respective probabilities of number of outcome for a given event. The probabilities of these outcomes typically interact with other events, as well as the introduction of evidence. Bayesian networks can be used as tools to model this kind of relationship. With a Bayesian network it is possible to find the conditional probability of any event occurring. This property enables inference in Bayesian networks.

In practice, inference in Bayesian networks can be accomplished by computing a joint probability table, from which the probable states of all variables, given some evidence can be calculated. Unfortunately, the size of the joint probability table grows exponentially with the number of variables in the network. Thus, inference in Bayesian networks quickly becomes intractable.

Nevertheless, it is possible to find an elimination order of the variables in a Bayesian network that exploits the independence between variables, to reduce the resulting total table size needed during inference. A method called triangulation can be used to find good elimination orders. Triangulation is closely related to elimination orders, since a graph has a perfect elimination order if and only if it is triangulated.

There are many ways to triangulate a graph, but we are interested only in the one that gives the smallest joint probability table. However, it is NP-hard to find an optimal elimination order, so it is important to investigate the accuracy and efficiency of heuristic methods to triangulate a graph.

Problem Statement

In this project, we will investigate heuristic methods for triangulating Bayesian networks. In addition, we will examine exact methods for finding optimal solutions, so that we may compare heuristic algorithms to these. Furthermore, we will attempt to improve the efficiency of optimal search methods.

This will be done on the basis of the following hypothesis:

It is possible to improve the efficiency of optimal search algorithms for triangulation of Bayesian networks.

Specifically we wish to investigate the following in this project:

- How heuristic methods for triangulation compare to each other in terms of deviation from the optimal solution.
- How can optimal solutions be found more efficiently?

Through this investigation we will acquire knowledge about the problem of triangulation in order to find improvements and optimizations for optimal triangulation of Bayesian networks.

CHAPTER 2

Bayesian Networks and the Problem of Triangulation

A Bayesian network is used as a probabilistic graphical model for simulating reasoning about problems of uncertainty. For instance, it can be used to evaluate the risks associated with some decision or comparing the odds of a number of wagers. Bayesian networks are a tool for performing inference or belief updating, which would otherwise be impractical or unfeasible to do manually.

Inference in Bayesian networks is the process of using evidence about events to determine the certainty of other events occurring. In practice Bayesian networks can be applied in various domains, where decisions are based on a set of variables and where probabilities are needed to asses some real-world problem; e.g. diagnosing an illness based on a number of possible symptoms, making a decision whether or not to test for the presence of oil before drilling, deciding to test milk or produce for contamination and creating artificial intelligence in computer games, etc. The rest of this chapter will contain a brief introduction to the definitions, tools and methods forming the basis for Bayesian networks and triangulation.

Section 2.1 covers background material from probability theory, which forms the grounding for Bayesian networks.

Section 2.2 has a brief introduction to Bayesian networks and the components that make up a Bayesian network, including definitions and methods for how evidence may be propagated in such a model. Moreover section 2.2.1 deals with inference and its use in Bayesian networks and section 2.2.4 presents variable elimination.

Finally, section 2.3, is about triangulation. Specifically, the process and purpose of triangulation with regard to inference in Bayesian networks.

Jensen and Nielsen (2007) provides a more thorough exposition on Bayesian Networks and how to perform reasoning with them.

2.1 Tools from Probability Theory

A Bayesian network exploits formulas and definitions found in probability theory. Therefore the following section introduces notation and definitions from this area.

The *sample space* of a given process, for which the outcome is uncertain, is the set of all possible *outcomes* of the process if and only if the outcomes are mutually exclusive. A subset of some sample space is called an *event*. In other words an event may contain different outcomes, e.g for a lottery with numbers ranging from 1-90, the sample space, s, consists of all outcomes, which in this case is the set of numbers $s = \{1,2,3,\ldots 88,89,90\}$. An outcome, o, may be any one number, e.g. $o = 12$ and an event, e, could, for instance, be all numbers in the sample space greater than 88, namely $e = \{89,90\}$. So, the set e is a subset of s, $e \subset s$.

The *domain* of a variable X is the set of possible states: $\mathrm{dom}(X) = \{x_1,\ldots,x_n\}$. We consider a set of variables $\{A_1,\ldots,A_n\}$ over a *sample space* $\mathcal{S}$.

To ensure consistent reasoning, for each variable A, it is required that the set of possible states $dom(A)$ are mutually exhaustive and mutually exclusive, i.e. there is no outcome which is not in the sample space, $a \notin \mathcal{S}$, and there is no outcome that implies $x = y$ and $x = z$ for all $y \neq z$, respectively.

The *joint* probability table $P(A,B)$ holds the probabilities for all events in A given some event in B. It follows that the size of such a probability table is $|A| \cdot |B|$, thus when the number of variables in

a joint probability table grows, the size of the probability table grows exponentially. The *conditional* probability table $P(A|B)$ is the probabilities for all events in A given the occurance of some event in B.

In the following two probability tables, the probability of having a disease A given some symptom B is used as an example. The table 2.1 lists the probability of having the disease A with or without the presense of symptom B.

Table 2.1: Conditional Probability Table, $P(A|B)$

	b_{true}	b_{false}
a_{true}	0.4	0.9
a_{false}	0.6	0.1

Table 2.2 represents the probabilities of having disease A and symptom B. This table shows that the probability of having disease A and symptom B, which is 15%.

Table 2.2: Joint Probability Table, $P(A,B)$

	b_{true}	b_{false}
a_{true}	0.15	0.25
a_{false}	0.45	0.15

Marginalization is the process of removing a variable from a joint probability table, e.g. to get the probability of symptom B from table 2.2 regardless of whether or not you have disease A. To get $P(B)$ variable A must be marginalized out of table 2.2. This may be expressed in the following formula $P(B) = \sum_A P(A,B)$ resulting in the following $P(B) = (0.15 + 0.45, 0.25 + 0.15) = (0.60, 0.40)$. Using marginalization of a joint probability table the probability of any variable in the joint probability table can be found. In this case the probability of having symptom B is 60%.

Fundamental Rule

The fundamental rule is used to calculate the probability of observing two events, a and b, from the probability of a and b given b.

$$P(a|b)P(b) = P(a \cap b) \tag{2.1}$$

The fundamental rule can be reformulated in different ways, where one leads to the next rule, namely Bayes' rule (Jensen and Nielsen, 2007, pg.5). Note the fundamental rule can also be generalized and applied to probability tables of variables.

Bayes' Rule

Bayes rule relates the probability of A given B to the probability of B given A, granted that the probability of B is not 0. Bayes' rule has the following form (Jensen and Nielsen, 2007).

$$P(A|B) = \frac{P(A|B)P(A)}{P(B)} \tag{2.2}$$

Here $P(A)$ and $P(B)$ is the prior probability of A and B respectively. The probability $P(A)$ is prior in the sense that it does not take any information, about B or anything else, into account. Bayes' rule can be used to update probability tables and compute the probability of A given B, using statistics about the prior probabilities of A and B, and information about the occurrence of B given A.

The Chain Rule

The chain rule allows domains with uncertainty to be represented. This rule can be used to calculate the probability $P(A_i)$ or $P(A_i|e)$ of some variable, A_i, in the universe of variables $\mathcal{U} = \{A_1, \ldots, A_n\}$, given the joint probability table $\mathcal{P}(\mathcal{U}) = P(A_1, \ldots, A_n)$. The general chain rule for probability distributions is

$$\mathcal{P}(\mathcal{U}) = P(A_n \mid A_1, \cdots, A_{n-1})P(A_{n-1} \mid A_1, \cdots, A_{n-2}) \cdots P(A_2 \mid A_1)P(A_1) \tag{2.3}$$

2.2 Bayesian Networks

A Bayesian network is a type of causal network. It is a directed acyclic graph consisting of a set of vertices representing variables and a set of edges, which are the causal relationships between these events. The direction of the edges indicate causal impact between events. Variables represent sample spaces consisting of, in this case, a finite set of states and each variable is always in one of its states.

A Bayesian network is:

- a set of variables, each with a finite set of mutually exclusive states,
- a set of directed edges between variables, such that the variables and edges form a directed acyclic graph and
- a conditional probability table $P(A|B_1, \ldots, B_n)$ for each variable A with parents $B_1, \ldots, B_n$.

An example of a Bayesian network can be seen in figure 2.1.

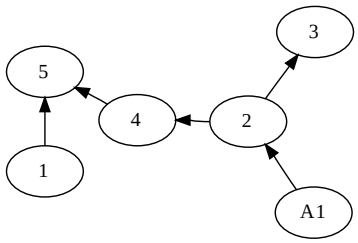


Figure 2.1: A simple bayesian network

2.2.1 Inference in Bayesian Networks

Inference is the process of belief updating given evidence in a Bayesian network. Evidence is introduced to a variable in a Bayesian network in order to *instantiate* the variable; such as setting a node to a given state, which in turn may have an impact on the probabilities of the other variables. When evidence is introduced it can cause a change in the probability tables of the network. These tables store the events and variables with their respective probabilities of being in a given state. Inference in Bayesian networks is generally NP-hard (Jensen and Nielsen, 2007, pg.45).

The table size is the product of the number of states of each variable. The total table size is just a measure for how much memory is required in order to store the probability table while performing inference.

There are rules for how evidence may be transmitted between variables in a Bayesian network. There are three different types of connections with their respective rules for evidence propagation, namely serial, diverging and converging connections. These rules are used to determine if two nodes are so-called *d-separated*. d-separation is explained in the following.

d-Separation

d-Separation reflects the relationship between two nodes and is used to encode the dependencies and indepencies between variables in the network. When two nodes are d-separated from each other, it means that if either node receives evidence this cannot propagate to the other node. To determine if two nodes are d-separated the connections between them are examined; These are either serial, diverging and converging. The opposite of d-separated is d-connected.

> **Definition 1** Let $G = (V, E)$ be a directed graph, then two nodes $A, B \in \mathcal{V}(G)$, $A \neq B$ are d-separated if for all paths between A and B there exists an intermediate variable $X \in \mathcal{V}(G)$ such that: *(i)* the connection is either serial or diverging with X having recieved evidence or *(ii)* the connection is converging, and neither X nor any of the descendants of X have received evidence.

In the following the three different kinds of connections are described.

Serial Connections

Two nodes have a serial connection if and only if there exists a sequence of directed edges connecting them. In figure 2.2 A and E are in a serial connection, since there is an edge from A to B and from B to E, but A and C are not in a serial connection because the direction of the edges connecting them changes. In figure 2.2 evidence can only be transmitted between A and E if B is not instantiated.

Diverging Connections

In figure 2.2 B and D have a diverging connection. Evidence may be transmitted between B and D unless the intermediate variable C is instantiated.

Converging Connections

In figure 2.2 A and C have a converging connection, evidence may be transmitted between A and C unless the intermediate variable B or any descendant of B is instantiated.

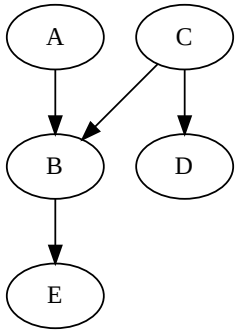


Figure 2.2: Different kinds of connections in Bayesian networks.

2.2.2 The Chain Rule for Bayesian Networks

The chain rule equation for Bayesian networks is slightly different from the general chain rule given earlier, yet it specifies the same operation. This is due to the fact that a Bayesian network specifies a unique joint probability distribution, which is given by all conditional probability tables present in

the Bayesian network. In addition, the chain rule for Bayesian networks demonstrates that a Bayesian network provides a compact representation of a joint probability distribution.

In equation 2.4 the chain rule for Bayesian networks is presented. $\mathcal{U} = \{A_1, \ldots, A_n\}$ denotes the universe of variables in a Bayesian network and A is some variable in $\mathcal{U}$. $pa(A)$ denotes the parents of the variable A.

$$P(\mathcal{U}) = \prod_{A \in \mathcal{U}} P(A \mid \text{pa}(A)) \tag{2.4}$$

Equation 2.5 shows the chain rule for when evidence is introduced into the Bayesian network.

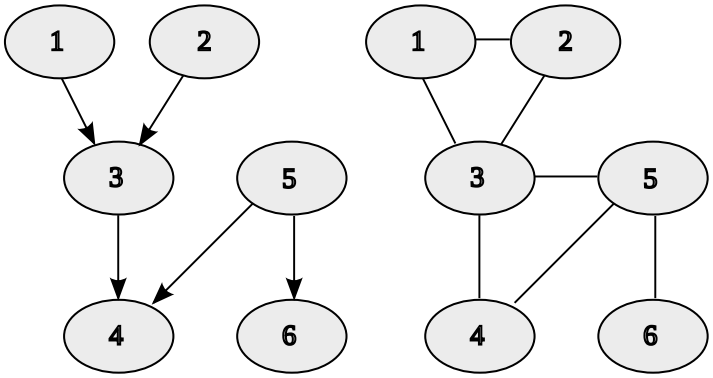
$$P(\mathcal{U} \mid \mathbf{e}) = \prod_{A \in \mathcal{U}} P(A \mid \text{pa}(A)) \prod_i \mathbf{e}_i \tag{2.5}$$

The chain rule for Bayesian networks enables us to calculate $P(A)$ and $P(A|e)$ for any $A \in \mathcal{U}$, given the joint probability table $P(\mathcal{U}) = \{A_1, \ldots, A_n\}$. The application of the chain rule is to marginalize variables in $\mathcal{U}$ until we are left with the variable we seek, including the variables where evidence has been introduced.

This enables us to calculate the probability of the remaining event. This is referred to as the process of variable elimination, which strongly depends on the order for which the variables are marginalized, namely the elimination order. Moreover, $P(\mathcal{U})$ grows exponentially with the number of variables in the Bayesian network, which underlines the importance of choosing an elimination order which gives a small joint probability table.

2.2.3 Moral Graph

A moral graph or domain graph for a Bayesian network, can be obtained by all connecting pairs of nodes that have a common child and removing the direction of all edges. Figure 2.3 shows a Bayesian network and it is moral graph.



(a) A directed graph (b) An undirected graph

Figure 2.3: Graph (b) is the moralized undirected version of graph (a)

The new links that are added between parents are called *moral links*. The moral graph is also called the *domain graph* for a Bayesian network and it is used for determining an elimination order. In the following section elimination is discussed.

2.2.4 Elimination

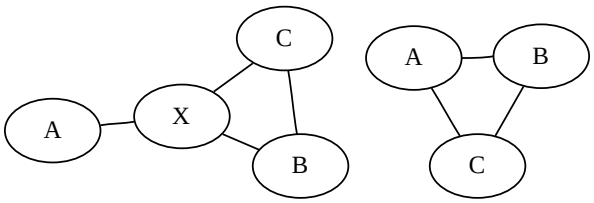
An elimination order σ is a sequence (ordered tuple) of variables signifying the order in which they are marginalized out

$$\sigma = (A_1, \ldots, A_n), A_i \in \mathcal{U} \tag{2.6}$$

where all variables $A \in \mathcal{U}$ must appear exactly once and the target variable appears last. Note, for n distinct variables, there are $(n-1)(n-2)\ldots 1 = (n-1)!$ different elimination orders ending with a given variable.

When calculating a potential within a Bayesian network we can use the chain rule. Instead of computing the joint probability table for all variables (which may not even be tractable, as it grows exponentially) we marginalize out variables during application of the chain rule until we are left with the desired variable. This is possible because marginalization is commutative, thus the order in which the variables in the graph are marginalized is irrelevant.

When variable A is eliminated we will be working with all the variables that are adjacent to A in the domain graph. This means that in the graph in which A has been eliminated, all neighbours of A are pairwise linked. If a poor elimination order is chosen the size of the intermediate joint probability table can grow intractably large.



(a) A domain graph

(b) Same domain graph but with X eliminated

Figure 2.4: The same domain graph, but in (b) X has been eliminated

In figure 2.4b two new links have been added, these links are called *fill-ins*. In this example, when eliminating X a new that was not present from the start is introduced. In order to avoid new domains we seek to avoid fill-ins. The less fill-ins the better, as an elimination order with no fill-ins requires less space (as it does not introduce new domains) than an elimination order that adds fill-ins. An elimination order that does not introduce fill-ins is called a *perfect elimination order*. There can be more than one perfect elimination order for any given graph. Finding this elimination order is closely tied to triangulation which will be described in the following section.

2.3 Triangulation

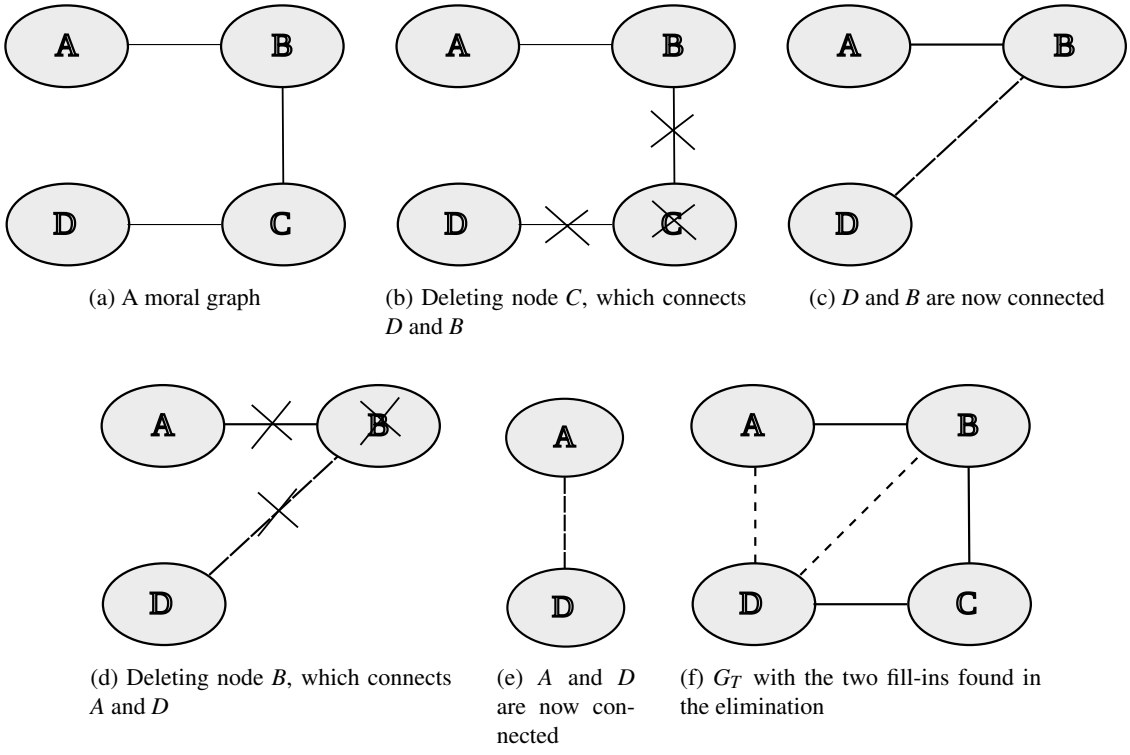
An undirected graph is called a triangulated graph or chordal graph if it has a perfect elimination order. For a triangulated graph it holds that every cycle consisting of at least four nodes has a chord. The process of triangulating a graph may introduce fill-in edges, to ensure that cycles of length greater than 3 have a chord. If and only if this condition holds, the graph has a perfect elimination order.

The elimination order for the triangulation is used to create a junction tree, for the triangulated graph. It is important to state that any triangulated graph can yield an elimination order that ends with any variable and that if one variable has a perfect elimination order then all variables have one. So, triangulation and finding an elimination order are intrinsically the same.

Example of a triangulation

The figures 2.5(a) – (f) shows an example of how to triangulate a moral graph. The elimination order (C,B,D,A) is far from minimal, there are many viable orders in this graph, but for the sake of a better example, an order with multiple fill-ins is chosen. In figure 2.5e, eliminating the two remaining nodes is arbitrary and can be done without introducing new fill-ins. The triangulated graph G_T in figure 2.5f is created by adding the two fill-ins found in the elimination process of the moral graph.

Triangulation of the graph in the example of figure 2.5 is not even necessary as it already has two perfect elimination orders, namely $\{A,B,C,D\}$ and $\{D,C,B,A\}$, and is therefore already a triangulated graph.



(a) A moral graph

(b) Deleting node C, which connects D and B

(c) D and B are now connected

(d) Deleting node B, which connects A and D

(e) A and D are now connected

(f) G_T with the two fill-ins found in the elimination

Figure 2.5: A non-minimal triangulation produced by the elimination ordering C,B,D,A

2.3.1 Triangulated graphs

A graph G is *complete* if all pairs of vertices $(A,B) \in G$ are pair-wise connected. The nodes $V(G)$ of a complete graph G is a *complete set*. A set of vertices $U \subseteq V(G)$ is complete in G if $G'[U]$ is complete. If U is a complete set and no other complete set B exists such that $U \subset B$, then U is a *maximal* complete set also known as a *clique*. A clique of $|V| = k$ nodes is a *k-clique*. We will usually not bother with the distinction between complete graphs and complete sets.

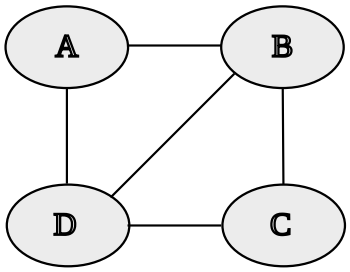


Figure 2.6: A graph with the 3-clique sets $\{A,D,B\}$ and $\{D,B,C\}$

In figure 2.6 there are many complete subgraphs; $\{A,B\},\{A,D,B\}$ etc., but only two maximal complete sets ($\{A,D,B\}$ and $\{B,D,C\}$), as all other complete sets are subgraphs of these. Note that the graph itself is not complete.

A node x is *simplicial* if and only if the node itself and its neighbors form a clique; in other words $fa(x)$ is a clique. The first node of an elimination order is always simplicial, therefore a triangulted graph will always contain at least one simplicial node.

> **Theorem 1** A triangulated graph $G = (V,E)$ that is not complete, with $|\mathcal{V}(G)| > 2$, will always have two non-adjacent simplicial nodes.

The following is a proof for theorem 1, it is based on a proof from (Koski and Noble, 2009, pg.128).

Proof Given a graph $G = (V,E)$ which is not complete, and that theorem 1 is true for all graphs that have less nodes then G. Consider two non-adjacent nodes α and β. Two subgraphs, $G[A]$ and $G[B]$, are created with the minimal (a,b)-separator for α and β denoted S. $G[A]$, is denoted as the largest connected component of $\mathcal{V}(G) \setminus S$ and $G[B] = \mathcal{V}(G) \setminus A \cup S$, so $\alpha \in G[A]$ and $\beta \in G[B]$.

By induction one of the two following cases is true: 1. $G[A \cup S]$ is complete. 2. $G[A \cup S]$ is not complete and it has two non-adjacent simplicial nodes. Since $G[S]$ is complete, at least one of the two simplicial nodes is in the subgraph $G[A]$, which in return means that this node is also simplicial in G; since none of the neighbors of $G[A]$ is in $G[B]$. If $G[A \cup S]$ is complete, then any node in $G[A]$ is a simplicial node of G. In both cases, there exists a simplicial node of G in $G[A]$. This whole deduction can similarly be used for the subgraph $G[B]$, in other words, there is a simplicial node in $G[B]$. A simplicial node in $G[A]$ and a simplicial node in $G[B]$ are non-adjacent, since they are separated by the minimal (a,b)-separator S and also are not in $G[S]$. Which proves that if G is not complete, G have two non-adjacent simplicial nodes.

2.3.2 Minimum, Minimal and Optimal Triangulations

In this report three different criteria which can be applied to the triangulated graphs sought after. They are listed in the following.

Definition 2 (Minimum triangulation) A triangulation F of a graph $G = (V, E \cup T)$ is minimum if and only if $\nexists F'$ of graph G where F' contains less edges than F, $F' < F$.(Berry et al., 2010)

Definition 3 (Minimal triangulation) A triangulation F of graph $G = (V, E \cup T)$, is minimal if and only if $\nexists F' \subset F$, such that $G' = (V, E \cup F')$ where F' is also a triangulation. (Berry et al., 2010)

Definition 4 (Optimal triangulation) A triangulation F of graph $G = (V, E \cup T)$, is optimal if and only if $\nexists F'$ of graph G which has a smaller table size than F

In this report we seek optimal triangulations, since it gives the best indication of the tractability of the joint probability table. (Ottosen and Vomlel, 2010b)

Summary

A Bayesian network is a probabilistic graphical model. A Bayesian network enables inference by the means of variable elimination, which can be done via the triangulated domain graph of the Bayesian network and associated perfect elimination order. The task of triangulation can be approached in many ways, however, since finding an optimal triangulation is NP-hard, the triangulation step is a major technical barrier for more widespread adoption of Bayesian networks, even though the basics are well understood.

It is still an open problem to find and optimize triangulation methods and algorithms that solve the problem more efficiently. Hence, a number of different triangulation algorithms exist each with its respective optimality criterion, e.g. minimal triangulation methods seek to find an elimination order that introduces the least fill-ins, whereas optimal triangulation focuses on creating the smallest joint probability table of the network. So, to improve triangulation, given that a number of methods already exist, the goal will most likely be to find more precise heuristic methods and more efficient optimal methods; which will in return mean Bayesian networks can be adopted easier. The motivation behind the search for better triangulation methods is to make Bayesian networks adoptable in more fields by allowing a lot more complex domains to be modelled. The methods and algorithms which already exist for triangulation will be discussed in further detail starting with minimal methods in chapter 3, continuing with greedy heuristic methods in chapter 4 and then optimal methods in chapter 5.

CHAPTER 3

Minimal Methods

The purpose of this chapter is to present methods for finding the minimal triangulations of a graph. Also, a method called recursive thinning which removes redundant fill-ins from non-minimal triangulated graphs is discussed.

A minimal triangulation has the property, by definition 3, that there exists no proper subset of the fill-in edges for which the graph still is triangulated. I.e. the graph is no longer triangulated if a fill-in edge were to be removed. The total table size obtained by a minimal method will, in general, not fare well against optimal methods, nor greedy heuristics for that matter. In return minimal methods are fast and can be applied to other problems, such as graph decomposition.

3.1 LB-Triang

The following algorithm, developed by (Berry, 1999), focuses on not requiring a precomputed minimal ordering of the nodes of a given graph, like previous traditional efficient algorithms have relied on.

Algorithm 1 LB-Triang

Input: A graph $G = (V,E)$, an ordering α on $\mathcal{V}(G)$.
Output: A minimal triangulation $H = (\mathcal{V}(G), \mathcal{E}(G) \cup F)$ of G.

```
1: H = G
2: F = ∅
3: for all vertices x in V(G) taken in order α do
4:     for all connected components C ∉ nb(x,H) do
5:         Make nb(C,G) into a clique by adding fill-in set F'
6:         F = F ∪ F'
7:         H = (V(G), E(G) ∪ F)
8:     end for
9: end for
```

The LB-Triang algorithm, shown in algorithm 1, triangulates a given graph $G = (V,E)$ from an ordering α of the nodes in the graph. The algorithm starts by looking at the first node x in the ordering, for which it finds the neighbors $nb(x,H)$ in the updated graph H, which contains the fill-ins as well as the original graph G. The algorithm then creates a set of all nodes that are not in the family of the current node x. Then we find subsets (connected components) $C \notin nb(x,H)$ in which nodes are connected in this set. For all the connected components it adds fill-ins between the components's neighboring nodes that are in the family of x in the original graph ($nb(C,G)$), making them into a clique. We then continue to the next node in the ordering, this goes on for all nodes in the ordering. Depending on the structure of the graph, the most fill edges are created at the first or second node. The algorithm does not halt when the graph is triangulated, it continues until all nodes have been iterated over; which is time-consuming and inefficient.

In figure 3.1 node A is chosen for the first iteration, hence the ordering α starts with A, connected non-neighbors of A, the connected components $C \notin nb(A,H)$ is found to be $\{\{E,C\}\}$. E and C's neighbors in the family of A is $\{D,B\}$ and therefore fill-in $\{D,B\}$ is added. For the second iteration B is investigated; connected non-neighbors of B, $C \notin nb(B,H)$ is found to be $\{\{E\}\}$ and E's neighbors within the family of

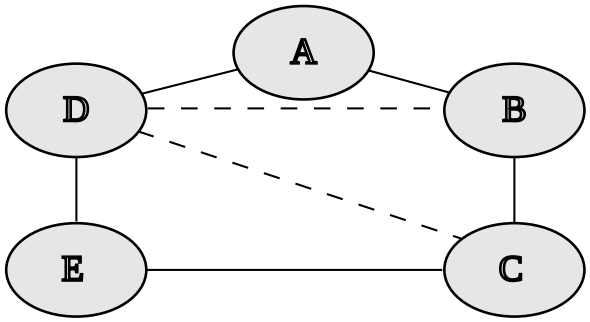


Figure 3.1: A minimal triangulation produced by the LB-Triang algorithm, where $\alpha = \{A, B, C, D, E\}$

B is $\{D, C\}$ and therefore fill-in $\{D, C\}$ is added. The algorithm keeps iterating over the rest of the nodes, but no further fill-ins will be added for any of them, and the graph now has a minimal triangulation.

3.2 Maximal Cardinality Search (MCS-M)

The MCS algorithm is based on a result that for recognizing chordality of a graph when performing a lexicographic breadth-first search (Lex-BFS) it is sufficient to simply maintain and compare the number of processed neighbours for each node, rather than maintaining a list of processed neighbours for each node. Hence, the name maximum cardinality search (MCS). MCS-M, which was developed by Berry et al. (2004), is an extension of MCS in the sense that so-called fill paths are also considered. In addition, MCS-M guarantees minimal triangulations, whereas MCS does not.

This method produces a minimal triangulation by generating an ordering α of the nodes in the graph and the set of fill-ins F, such that G has a perfect elimination ordering. The ordering α is essentially a reversed elimination order.

Integer weights are maintained for each node in the graph. A weight is the cardinality of the already processed neighbours of a node. In other words, the node which is adjacent or for which there is a path to the highest number of numbered nodes is selected in each iteration. The algorithm is described in 2.

The algorithm iterates over all n vertices in G. The first node is choosen arbitrarily, since each node has its initial weight $w(v) = 0$. At each step i a node v is assigned a number and the weight of all unnumbered nodes $u_1, \cdots, u_k$ for which there exists a path $\rightsquigarrow$ between u and v such that $\forall x_i \in \rightsquigarrow$: $\nexists \alpha(x_i) \wedge w(x_i) < w(u)$ for $1 \leq i \leq k$ (i.e. each node is unnumbered and has weight which is strictly less than $w(u)$ and $w(v)$, of course, since v was the node with greatest inital weight) are added to a set S, which is the set of nodes on the fill path of v.

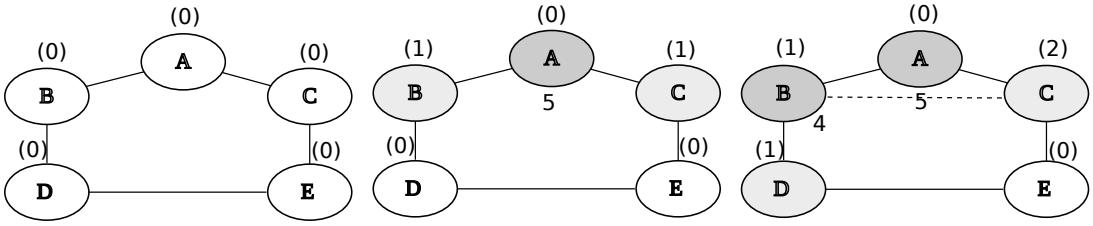
Subsequently all nodes $s \in S$ receive the weight $w(s) = w(s) + 1$ and if $(s, v) \notin E$ then $F = F \cup (s, v)$. Before next iteration v is assigned a number $\alpha(v) = i$. Once all nodes have been processed a reversed minimal elimination ordering a and the set of fill-ins F have been produced. (Berry et al., 2004)

An example of the algorithm running on a graph is shown in figure 3.2.

Algorithm 2 MCS-M

Input: A Graph $G = (V, E)$.
Output: A minimal elimination ordering α of G and the corresponding triangulated graph $H = G^+_\alpha$.

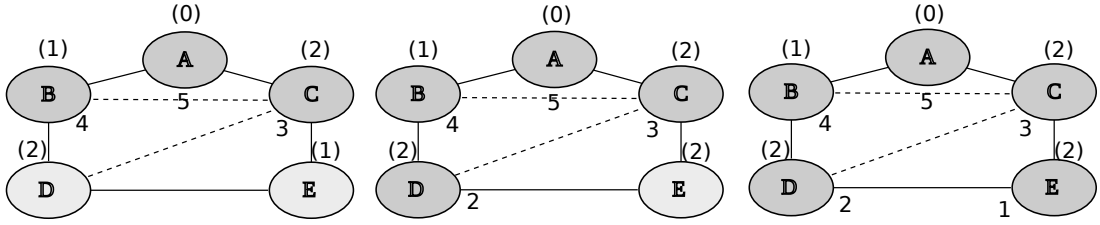
$F = \emptyset$
$R = \mathcal{V}(G)$ ▷ R is the set of unnumbered nodes.
for all nodes $v \in \mathcal{V}(G)$ **do**
    $w(v) = 0$
    for $i = n$ **downto** 1 **do**
        Choose an unumbered node v s.t. $\arg\max_{v \in R} w(v)$
        $S = \emptyset$
        for all unnumbered nodes $u \in \mathcal{V}(G)$ **do**
            if $\exists uv \in E(G)$ or a path $u, x_1, x_2, \cdots, x_k, v$ in G through unnumbered nodes s.t. $w(x_i) < w(u)$ for $1 \leq i \leq k$ **then**
                $S = S \cup \{u\}$
            end if
        end for
        for all nodes $u \in S$ **do**
            $w(u) = w(u) + 1$
            if $uv \notin \mathcal{E}(G)$ **then**
                $F = F \cup \{uv\}$
            end if
        end for
    end for
    $R = R \setminus \{v\}$
end for
return $H = (\mathcal{V}(G), \mathcal{E}(G) \cup F)$



(a) The initial graph. All weights are zero.

(b) $v = A$. A is numbered and the weights $w(B)$ and $w(C)$ are incremented.

(c) $v = B$. A fill path through $(B \to D \to E \to C)$ is found. $w(D)$ and $w(C)$ are incremented.



(d) $v = C$. A second fill path $(C \to E \to D)$ is found. $w(E)$ and $w(D)$ are increased by one.

(e) $v = D$. $w(E)$ is updated.

(f) $v = E$. E just receives a number and the algorithm terminates.

Figure 3.2: The MCS-M algorithm run on the example graph. Numbers in parentheses denote $w(u)$. Dark grey represents a numbered node, with corresponding number written below, and light grey points out that the weight for a given node is incremented. This example produces the minimal eliminaton ordering $\alpha = (E, D, C, B, A)$. (pg.292 Berry et al., 2004, fig.5)

3.3 Recursive Thinning

After finding an elimination ordering, one might find that there is a subset $T' \subset T$, such that T is the corresponding triangulation of a graph G, and T' is also a triangulation of G. Where the total table size of T' is no worse than T, and often significantly better than T.

In order to develop and design an algorithm that removes redundant fill-ins, and thereby making it minimal, the following Theorem is proposed by Kjaerulff (1990).

Theorem 2 Let $G = (V,E)$ be a graph and $G' = (V, E \cup T)$ be triangulated. Then T is minimal if and only if each edge in T is a unique chord of a 4-cycle in G'.

An equivalent proposal of theorem 2 is provided by the following corollary.

Corollary 1 Let $G = (V,E)$ be a graph and $G' = (V, E \cup T)$ be triangulated. Then T is minimal if and only if for each edge $\{v,w\} \in T$ there is a pair of distinct vertices $\{x,y\} \subseteq \text{nb}(v,G') \cap \text{nb}(w,G')$ such that $\{x,y\} \notin E \cup T$.

Figure 3.3 illustrates the properties of corollary 1; the graph has a minimal triangulation, because for the fill $\{A,C\} \in T$ there is no pair of adjacent nodes that are common neighbours of A and C.

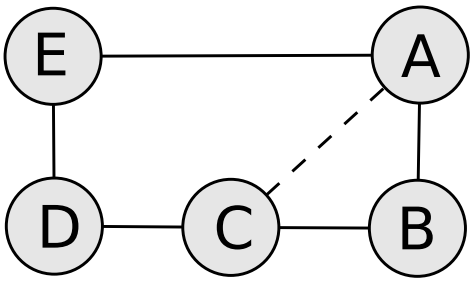


Figure 3.3: A minimal triangulation produced by the elimination order starting with B

A redundant fill-in, $e = \{v,w\}$ can only be a subset of a single clique C, as a fill-in which is a subset of more than one clique infers that the graph is not triangulated, which contradicts the redundancy of e. When the reduntant fill-in e is removed, C splits into two new cliques; $C_1 = C \setminus \{v\}$ and $C_2 = C \setminus \{w\}$, which weights sum is typically less than the weight of C and never worse than that.

A triangulation T may become minimal by dropping the redundant fill-ins that fulfil the conditions of corollary 1. However, it is important to run sweeps through the graph more than once, and therefore the algorithm is made recursive. The following example illustrates why it is important to run through the graph at least more than once. In figure 3.4 the fill-in $\{B,D\}$ cannot be removed, as it has a pair of non-adjacent neighbours ($\{A,C\}$). The fill-in $\{A,D\}$ can however be removed as they only have one common neighbour, i.e., B. After $\{A,D\}$ has been deemed redundant and removed from T, $\{B,D\}$ can be removed as it no longer has a pair of non-adjacent neighbours.

The following algorithm proposed by Kjaerulff (1990) is based on the previous discussion.

The algorithm works by finding fill-ins without common neighbours that are non-adjacent, removing it from the set of fill-ins T as well as the original triangulated graph G and recursivly running the algorithm again with the new input to remove new candidates.

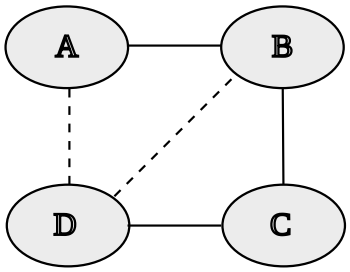


Figure 3.4: A non-minimal triangulation produced by the elimination order C,B,D,A

Algorithm 3 Recursive Thinning

```
1: function THIN(T, G = (V, E ∪ T), R)        (initially R = T)
2:     R' = {e1 ∈ T | ∃e2 ∈ R: e1 ∩ e2 ≠ ∅}
3:     T' = {{v, w} ∈ R' | G(nb(v, G) ∩ nb(w, G)) is complete}
4:     if T' ≠ ∅ then
5:         return Thin(T \ T', G = (V, E ∪ T \ T'), T')
6:     else
7:         return T
8:     end if
9: end function
```

Summary

Minimal methods have fast execution time. They provide a quick way of obtaining a minimal triangulations, but do not guarentee triangulations of optimal table size. This goes to show that minimal triangulations in general do not lend themselves to triangulation with total table size as the optimality criterion, as shown in chapter 10. Nevertheless, minimal triangulations are useful for the task of finding decompositions of a graph.

In the following chapter other strategies for computing triangulations, namely greedy heuristic methods, are discussed.

CHAPTER 4

Greedy Heuristic Methods

This chapter covers greedy heuristic methods for producing triangulated graphs. The reason why greedy heurstic methods can be employed for the task of triangulation, is that some of them may yield relatively good approximations to an optimal solution in a fraction of the time required by optimal search methods.

However, since these methods rely on a local greedy search, mistakes may accumulate throughout execution, leading to a less than optimal elimination order. Moreover, each heuristic works better on some graphs rather than others. Nevertheless, using heuristic methods to compute an initial upper bound for optimal search methods provides the opportunity to discard non-optimal branches by the means of upper bound pruning right from the beginning.

The greedy methods described in this chapter generally follow the same pattern and can therefore be integrated into one algorithm. The heuristics only differ in the way the cost is computed, as well as the function each specific method seeks to minimize.

4.1 Generic Greedy Algorithm

The heuristics discussed in the following seek to produce a minimal triangulation based on some local optimization criterion. Algorithm 4 shows the generic greedy algorithm. The subscript X denotes the name of the applied optimization criterion. For instance, Greedy$_{MinFill}$ indicates that ComputeCost uses the minimum number of fill-ins introduced after eliminating a node to choose the best candidate for the elimination order.

The *depth* parameter in the signature of algorithm 4 indicates the look-ahead depth which should be applied to a given heuristic. The role of this is to have the ability to configure the heuristic algorithms to search deeper into the problem graph and potentially choose better elimination orders based on more informed paths found from the look-ahead searches.

Algorithm 4 Greedy

```
1:  function GREEDY_X(G, depth)
2:      F = ∅
3:      R = V(G)                                   ▷ R is the set of non-eliminated nodes.
4:      while R ≠ ∅ do
5:          minCost = ∞
6:          best = ⊥
7:          for each node v ∈ R do
8:              cost_v = COMPUTECOST_X(G, R, v, depth)
9:              if cost_v < minCost then
10:                 minCost = cost_v
11:                 best = v
12:             end if
13:         end for
14:         F = F ∪ ELIMINATENODE(v, R)            ▷ Note: sets R = R \ {v}.
15:     end while
16:     return T = (V(G), E(G) ∪ F)
17: end function
```

4.1.1 Min-fill

The first cost function covered is minimum fill (min-fill). Min-fill is a heuristic strategy which produces a triangulated graph by successively eliminating nodes which lead to the fewest fill-ins. Specifically, each node v_i in the elimination order $\sigma = (v_1, v_2, v_3, \cdots, v_n)$ is greedily chosen such that the number of fill edges $|F_i|$ indtroduced at each step by eliminating v_i is the smallest. The minimum fill of a graph $G = (V, E)$ is $|\mathcal{E}(G) - \mathcal{E}(G_T)|$ over all triangulations G_T of G. (Ottosen and Vomlel, 2010b) Since this method makes use of a local greedy heuristic estimate, it is not guarenteed to find the minimum number of fill-ins whose inclusion renders the graph triangulated. The general problem of finding the minimum number of fill-ins required in order to make a graph triangulated is NP-complete, which was shown by (Yannakakis, 1981) by reduction from the optimal linear arrangement problem.

The ComputeCost function for min-fill is shown in algorithm 5.

For each node $u \in \mathcal{V}(G)$ the algorithm iterates the neighbour set $nb(u, G)$, and greedily selects a node $v \in nb(u, G)$ which introduces fewest fill-ins to the triangulated graph $G_T = (\mathcal{V}(G), \mathcal{E}(G) \cup F)$ after elimination. By augmenting the algorithm with k look-ahead steps it is possible to consider a path of nodes.

Algorithm 5 Min-fill

```
1:  function COMPUTECOST_MinFill(G, R, n, depth)        ▷ R is the set of remaining nodes
2:      cost = COUNTFILLINS(G, n, R)        ▷ Finds the number of fill-ins introduced by eliminating n
3:      R' = R \ {n}
4:      if depth > 1 and R' ≠ ∅ then
5:          minCost = ∞
6:          for each node v ∈ R' do
7:              cost_v = COMPUTECOST_MinFill(G, R', v, depth − 1)
8:              if cost_v < minCost then
9:                  minCost = cost_v
10:             end if
11:         end for
12:         cost = cost + minCost
13:     end if
14:     return cost
15: end function
```

4.1.2 Min-width

The minimum width (min-width) criterion requires the triangulated graph to have minimum treewidth, which is the size of the largest clique minus one. The algorithm checks the degree $\delta(v)$ of each node $v \in \mathcal{V}(G)$ and the node with the lesser degree is removed first. The degree of a node is the number incident edges to the node. (Ottosen and Vomlel, 2010b)

The cost function for min-width is shown in algorithm 6.

The algorithm goes through all n vertices in $\mathcal{V}(G)$ determining their degree. While there are still remaining nodes, each remaining node is examined and the one with the least degree is eliminated. After a node is eliminated the degree $d(u)$ of each $u \in nb(n, G)$ is recomputed. (Ottosen and Vomlel, 2010b)

Algorithm 6 Min-width

```
1: function COMPUTECOST_MinWidth(G, R, n, depth)        ▷ R is the set of remaining nodes
2:     cost = | nb(n, G) ∩ R |                          ▷ Cost is the width of the potential clique.
3:     ...                                              ▷ Identical to algorithm 5 lines 3-13.
4:     return cost
5: end function
```

4.1.3 Min-weight

The minimum weight (min-weight) criterion states that a triangulated graph must have minimum table size. Each node $v \in \mathcal{V}(G)$ has a weight $w(v)$ associated to it, which corresponds to the number of states $sp(X)$ of the respective variable X in a Bayesian network. (Ottosen and Vomlel, 2010b)

Min-weight is shown in algorithm 7. The min-weight heuristic minimizes the function $f(C_u) = w(C_u)$, where C_u is the family of each node $u \in \mathcal{V}(G)$ and $w(C_u)$ is the weight of the node u. The algorithm iterates through all nodes $u \in \mathcal{V}(G)$ and calculates the weight of the family $fa(u)$ of each u, which is the table size. Essentially, this algorithm minimizes the weight of the cliques that are being created by calculating their weight using $w(C_u) = \prod_{j \in C_u} c(j)$.

Algorithm 7 Min-weight

```
1: function COMPUTECOST_MinWeight(G, R, n, depth)       ▷ R is the set of remaining nodes
2:     clique = {nb(n, G) ∩ R} \ {n}
3:     cost = TABLESIZE(clique)                         ▷ The table size of the potential clique introduced.
4:     ...                                              ▷ Identical to algorithm 5 lines 3-13.
5:     return cost
6: end function
```

Summary

As mentioned earlier the heuristics described above may produce good or bad triangulations depending on the graph. Therefore it makes sense to compare their accuracy on the same set of graphs. Benchmarks have been performed and results are discussed in chapter 10.

Greedy heuristic methods are fast, but are not guaranteed to always lead to the best solution, since their search space is much smaller than the space searched by optimal methods. This may be a problem if the elimination order for some Bayesian network produced from e.g. min-fill turns out to be intractable, which is not unthinkable. (Ottosen and Vomlel, 2010b)

In the next chapter optimal search methods are covered. These methods are guaranteed to find an elimination order which yields the optimal table size, however this comes at the cost of exponential asymptotic complexity, due to the NP-hardness of finding a elimination order of minimum total table size. Because of the inherent difficulty of exponential complexity, it is important to explore methods that reduce the runtime and/or memory requirements by getting rid of non-optimal branches.

CHAPTER 5

Searching for Optimal Solutions

This chapter presents methods for finding the optimal triangulation of a graph, where the optimality criterion is the total table size.

5.1 Optimal Search Algorithms

In this chapter we consider two different algorithms for computing optimal elimination orders, namely depth-first search and best-first search. These algorithms find an optimal triangulation by searching the space of all elimination orders. What sets these two algorithms apart is the strategy by which this search space is explored. Moreover, either method has its pros and cons with respect to space and time complexity. Still, both methods have the property that they permit certain enhancements, such as upper bound pruning and coalescing, which enable us to increase their efficiency.

Definition 5 A partially triangulated graph G_T of a graph $G = (V,E)$ is a subgraph $G[T]$ with a perfect elimination order, where $T = \mathcal{V}(G) \setminus R$ and $R \neq \emptyset$. We say T is the set of eliminated nodes and R is the set of remaining nodes in G. Also, $T \cup R = \mathcal{V}(G)$ and $T \cap R = \emptyset$.

When running either of these search algorithms, an initial upper bound or seed value is computed with a heuristic method, such as min-fill. This upper bound is used to reduce the search space. Since the optimality criterion is total table size, the upper bound is simply instantiated with the total table size of the solution found with the heuristic method.

Furthermore, in Ottosen and Vomlel (2010a) a lower bound on the total table size is also computed using the maximal cliques of a partially triangulated graph. Note, finding the maximal cliques of a graph is NP-complete. So, in order to avoid constructing the maximal cliques and computing the total table size of every elimination order, an approximated table size of a partial elimination order is computed. This approximation is used to avoid expansion of some elimination orders, i.e. an elimination orders with approximated total table size larger than the upper bound. Note, the approximation must be a lower bound for this work.

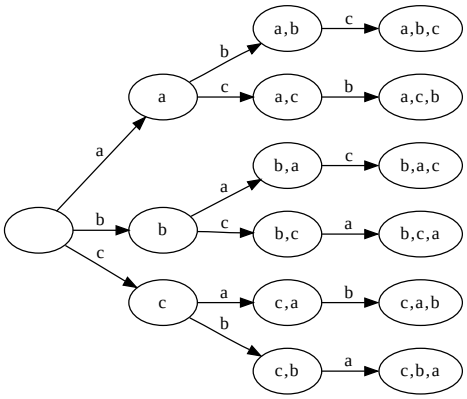


Figure 5.1: The search tree for all elimination orders of a graph with 3 nodes.

Now, the closer to the optimal solution this initial upper bound is, the more the efficiency of the algorithm is increased, since more unpromising branches will be pruned with a tighter bound. Again, since we consider optimal search algorithms and ensured that the approximated total table size is a lower bound, the resulting elimination order will never be worse (in terms of total table size) than the elimination order found initially by the heuristic value.

In figure 5.1 the tree illustrates the space of all elimination orders of any graph with 3 nodes a,b,c. Each node in represents a computation step and a distinct partial elimination order. Notice, that if a lower bound on the total table size of step (a), where node a has been eliminated, is larger than the total table

size of some complete elimination order, there is no need to explore the successor steps, (a,b), (a,c) and their respective successor steps, (a,b,c) and (a,c,b). Basically, successor branches are discarded.

Additional reduction of the search space is possible by the means of coalescing. This method is possible due the result, known as the Invariance theorem (theorem 3). It basically states that the resulting subgraph $G[V \setminus Y]$ induced by applying any elimination order containing the same subset Y is exactly the same, no matter what order each node in Y is eliminated.

The Invariance theorem and proof given in Darwiche (2009)[p.236] are reproduced in the following.

> **Theorem 3 (Invariance Theorem)** If σ_1 and σ_2 are two partial elimination orders containing the same set of nodes $Y \subset \mathcal{V}(G)$, then applying these will lead to the identical subgraphs, G_{σ_1} and G_{σ_2}.

Proof We need to show that two nodes a and b of $\mathcal{V}(G) \setminus Y$, which are non-adjacent in the initial graph G, are adjacent in graph G_{σ_1} if and only if there exists a path $a, x_1, \ldots, x_m, b$ which connects a and b in graph G, and $\forall x_i : x_i \in \sigma_1$. In other words, the set of edges introduced between nodes in G_{σ_1} by eliminating up to σ_1 are the same, regardless of the order each $x_i \in \sigma_1$ is eliminated. This ensures that G_{σ_1} and G_{σ_2} are the exact same subgraphs.

Let $G = G_1, \ldots, G_n = G_{\sigma_1}$ be the sequence of graph transformations generated by eliminating up to σ_1 in G. Suppose there is a path $\rho = (a, x_1, \ldots, x_m, b)$ connecting nodes a and b in the graph G_1 in the sequence. Let G_i be the last graph in the sequence of transformations which preserves the path ρ. Graph G_{i+1} is induced by eliminating some node x_j from the path ρ. Eliminating x_j introduces an edge between the two nodes x_{j-1} and x_{j+1} on the path ρ, if there is not already an edge. Consequently, G_{i+1} still maintains a path ρ' connecting a and b, where all internal nodes are in σ_1. Therefore, nodes a and b stay connected by a path ρ' after elimination of x_j. Also, a and b are adjacent in graph G_{σ_1}.

We now assume that nodes a and b are adjacent in graph G_{σ_1}, but non-adjacent in G. Let G_i be the first graph in the sequence in which a and b are adjacent. Graph G_i is the result of eliminating some variable x_j where $\{a,b\} \subseteq nb(x_j)$ in graph G_{i-1}. This implies that nodes a and b are connected by a path ρ where each internal node x_i is in σ_1.

By repeated argument on the edges (a,x_j) and (b,x_j), it follows that a and b must be connected by a path ρ where each internal node is in σ_1. □

This result shows that the search space of all possible elimination orders contains many replicated parts. This knowledge can be applied to optimal search algorithms with the benefit of avoiding having to solve identical subgraphs in the search tree. In practice it requires that an algorithm keeps track of the subgraphs seen so far, so it is possible to look them up and perform coalescing. (Darwiche, 2009)

5.2 Clique Maintenance

In Ottosen and Vomlel (2010a) the problem of finding all of maximal cliques is reduced by computing the maximal cliques of G_2 using the maximal cliques of G_1, where the only difference between G_1 and G_2 is a set of edges. Typically, these are the fill-ins introduced when eliminating a node in a step from the partially triangulated graph G_1 to G_2.

5.2.1 Finding Maximal Cliques

The Bron-Kerbosch algorithm can be used to find the maximal cliques of a graph. It operates with three disjoint sets R, P and X. The set P are the prospective nodes; nodes that may be used in a maximal clique. The set X is the excluded nodes; nodes that may not be used in a maximal clique, this set is used

to avoid reporting the same maximal cliques more than once. The set R is the nodes that are currently in the clique.

The algorithm works by recursively calling itself, with $R \cup \{v\}$ as R, $P \cap nb(v,G)$ as P and $X \cap nb(v,G)$ as X, for all $v \in P$. During execution the set R, which is the current clique being grown, is expanded by one node (v is added), while P is reduced to only the neighbours of v that were previous prospective nodes. All nodes in R are connected to all nodes in $R \cup P$, and when $P = \emptyset$, R is a maximal clique, and if $X = \emptyset$ the maximal clique R will not have been reported before.

Bron-Kerbosch with pivot, see algorithm 8, only performs the recursive call for all prospective nodes $v \in P \setminus nb(v,G)$ that are not neighbours of some pivot node p. The Bron-Kerbosch algorithm, with and without pivot, is presented detailed in (Bron and Kerbosch, 1973) where Bron-Kerbosch with pivot selection is refered to as "Version 2". Different pivot selection strategies are discussed in Cazals and Karande (2008), however, we just choose the first node in $P \cup X$, as it is done in Ottosen and Vomlel (2010a).

The maximal cliques of a graph $G = (V,E)$ can be found by initially invoking the Bron-Kerbosch algorithm (See algorithm 8) as such BRONKERBOSCH($G, \emptyset, V, \emptyset$). It is also possible to find the maximal cliques in a subgraph $G[S]$ induced by the nodes $S \subseteq V$ by calling the algorithm with the arguments BRONKERBOSCH($G, \emptyset, S, \emptyset$).

Algorithm 8 Bron-Kerbosch with pivot

```
 1: function BronKerbosch(G, R, P, X)
 2:     if P = ∅ and X = ∅ then
 3:         return {R}                                  ▷ Report R as maximal clique.
 4:     else
 5:         C = ∅
 6:         p = n, where n ∈ P ∪ X                      ▷ Pivot selection.
 7:         for all v ∈ P \ nb(p,G) do
 8:             P = P \ {v}
 9:             C = C ∪ BronKerbosch(G, R ∪ {v}, P ∩ nb(v,G), X ∩ nb(v,G))
10:             X = X ∪ {v}
11:         end for
12:         return C
13:     end if
14: end function
```

When implementing algorithm 8 it is worth noting that each maximal clique will only be reported/returned once on line 3. Thus, the union operation in line 9 will only operate on disjoint sets. In a practical implementation this means that a reference to an array could be given as parameter and R could be added to this array on line 3. As a result, avoiding a potentially expensive union operation, reducing the number of dynamic memory allocations.

5.2.2 Finding New Maximal Cliques After Adding/Removing Edges

In Ottosen and Vomlel (2010a) the new maximal cliques in $G' = (V, E \cup F)$ after adding (or removing) a set of edges F to $G = (V,E)$ are computed by calling Bron-Kerbosch like this BRONKERBOSCH($G', \emptyset, fa(I,G'), \emptyset$) and consider the reported cliques that intersect I, where $I = \{v,u \mid \{u,v\} \in F\}$ is the set of nodes to which a new edges in F is attached. This is possible because new cliques must appear in the family of I, and all maximal cliques in $G[fa(I,G')]$ that intersect with I are also maximal cliques in G'. And as mentioned in the section about Bron-Kerbosch it can also be used to find maximal cliques in a subgraph, such as $G[fa(I,G')]$.

In algorithm 9, this approach has been taken a little further by including the intersection test with I into the Bron-Kerbosch algorithm. This allows us to prune further in line 6, as suggested in Ottosen and Vomlel (2010a), and only report maximal cliques in G' on line 4. In a practical implementation this means that we will never allocate memory for cliques that are not maximal in G'. And we maintain the good implementation properties from the Bron-Kerbosch, i.e. that the union operation in line 11 only operates on disjoint sets.

Algorithm 9 Algorithm finding new maximal cliques after adding/removing edges

```
1:  function FINDNEWCLIQUES(G, R, P, X, I)
2:      if P = ∅ and X = ∅ then
3:          if R ∩ I ≠ ∅ then
4:              return {R}                          ▷ Report R as a new maximal clique.
5:          end if
6:      else if R ∩ I ≠ ∅ or P ∩ I ≠ ∅ then         ▷ Extra pruning.
7:          C = ∅
8:          p = n, where n ∈ P ∪ X
9:          for all v ∈ P \ nb(p, G) do
10:             P = P \ {v}
11:             C = C ∪ FINDNEWCLIQUES(G, R ∪ {v}, P ∩ nb(v, G), X ∩ nb(v, G), I)
12:             X = X ∪ {v}
13:         end for
14:         return C
15:     end if
16:     return ∅
17: end function
```

To find the new maximal cliques that appear in $G' = (V, E \cup F)$ after adding edges F to $G = (V, E)$ FINDNEWCLIQUES$(G', \emptyset, fa(I, G'), \emptyset, I)$ is called, where $I = \{u, v \mid \{u, v\} \in F\}$. This will yield the maximal cliques that intersect I, which includes all the new maximal cliques. To update the old maximal clique set $\mathcal{C}(G)$ to find the maximal cliques of G' all cliques that intersect I are removed and the maximal cliques found by calling FINDNEWCLIQUES$(G', \emptyset, fa(I, G'), \emptyset, I)$ are added.

5.2.3 Incremental Update

Algorithm 10 computes the maximal clique set C' of a graph $G' = (V, E \cup F)$ using a graph $G = (V, E)$ and the maximal clique set C of this graph. This is done by removing cliques that intersect with some node for which a new edge has been added. Subsequently, the set of maximal cliques that appear around the edges to which a new edge have been added are computed, using algorithm 9. Notice that in practical implementation it is often useful to also maintain total table size while adding/removing cliques.

Algorithm 10 Algorithm updating the cliques set when adding edges

```
1: function UPDATECLIQUES(G, G', C)
2:     F = E(G') \ E(G)
3:     I = {v, u | {u, v} ∈ F}
4:     C' = {X ∈ C | X ∩ I = ∅}
5:     C' = C' ∪ FINDNEWCLIQUES(G', ∅, fa(I, G'), , I)
6:     return C'
7: end function
```

5.3 Best First Search for Optimal Triangulations

Best-first search (BFS) is an optimal search algorithm, which finds the solution by expanding the most promising nodes first, given some rule for prioritizing these nodes. In contrast to the breadth-first search algorithm, which continuously expands all nodes in the order by which they were enqueued, chooses to expand the most promising successor.

For this reason, BFS requires to maintain a frontier of promising expanding nodes in memory. So, to adapt this algorithm to the searching problem of finding optimal elimination orders, a structure called step is used. In most litterature this is known as a node, but for disambiguation, we will refer to these as steps and solely be referring to a vertex, representing a variable in a Bayesian network, as a node.

Each step represents a partial elimination order by storing information about the configuration of the graph, remaining nodes and maximal cliques. In pseudocode these attributes are accessed by: $s.G$, $s.R$, $s.C$, $s.tts$; where $s.G$ is the current graph configuration, $s.R$ is the remaining missing nodes, $s.C$ are the cliques of the graph used in clique maintenance and finally $s.tts$ is the current table size.

The BFS algorithm included in this report is the algorithm developed by Ottosen and Vomlel (2010b). Likewise, our implementation makes use of hash map for coalescing. As mentioned earlier, this coalescing map is used to prune unnecessary expansion of steps leading to the same resulting subgraph, yet with table size worse than the current upper bound. Again, the benefit of this is increased efficiency, by avoiding computation of the same subproblems in the search graph.

Pseudocode for BFS is shown in algorithm 11. Here, an start step is created, with the initial graph, the remaining nodes, which at this point are all nodes of the graph, except already simplicial nodes. BFS uses the greedy minimum fill algorithm to compute an initial upper bound and then find the cliques. The cliques, can then be used to determine the initial table size of the graph. This initial table size is also a measure of the best solution found so far.

The step is then added to a priority queue. While this queue is non-empty, a step is dequeued and expanded. Expanding a step corresponds to generating successor steps for all the remaining nodes not in the partial elimination order of the parent step. New fill-ins are introduced and the set of remaining nodes is recomputed after elimination of a node, as well as any simplicial nodes. Table size and the affected maximal cliques are recomputed.

To sum up: A branch is abandoned if the table size of the current successor step is larger than the current best. If there are no more nodes remaining in the step, this means a solution or goal step has been found. Finally, a branch is abandoned if it coalesces with a better partial elimination order in the hash map.

After pruning the coalescing map is updated. All steps that have the same set of remaining nodes as the step currently being expanded are removed from the queue. At the end of each iteration the successor step is enqueued with its associated table size for prioritization.

5.4 Depth-First Search

Depth-first search (DFS) can also be used to find elimination orders with optimal total table size. DFS is a simple uninformed search method, which expands a path as deep as possible, by always expanding the first child step encountered, backtracking if a goal step or a leaf step is found. In other words, steps are expanded in a last-in-first-out manner.

DFS is advantageous in terms of memory requirements, since it does not maintain a frontier of nodes, unlike BFS. The search space of all elimination orders forms a tree structure of size $O(n!)$, where $n = |\mathcal{V}(G)|$. Exploring this tree in a depth-first manner requires $O(n)$ space and in general $\Theta(n!)$ time, since deeper steps are expanded first and the height of the tree is at most n.

The running time with coalescing is $O(n!)$ rather than $\Theta(n!)$ without this enhancement. However, coalescing requires $O(2^n)$ space, but with much smaller hidden constants than with best-first search.

Algorithm 12 lists pseudocode for DFS as presented by Ottosen and Vomlel (2010b). The code is similar to that of best-first search, shown in listing 11. One thing that is immediately apparent is the lack of a priority queue, where instead EXPANDSTEP calls itself recursively until it encounters a step that can be pruned or a goal step in line 21. If this goal step is better than a solution found so far, the best solution is updated.

Three global variables are used, namely $best_T$, which stores the best triangulation found so far and $best_{tts}$ is its associated total table size, and lastly there is map, which is the coalescing map initialized in line 10. In line 12 the best solution is returned.

(Darwiche, 2009; Ottosen and Vomlel, 2010b)

Summary

In this chapter we have introduced the searching problem of computing an elimination order of optimal table size. We have discussed optimal search algoritms and how they can be adapted to search for optimal triangulations. Later in this report we will see that the efficiency of both algorithms be improved further, by exploiting certain properties of elimination orders and triangulated graphs.

Algorithm 11 Best First Search

```
1: function BestFirstSearch(G)
2:     s = CreateStep()                                  ▷ Create an empty step structure.
3:     s.G = G
4:     s.R = V(G) \ FindSimplicials(G)
5:     s.C = BronKerbosch(G, ∅, V(G), ∅)
6:     s.tts = TableSize(s.C)
7:     map = CreateHashMap()                             ▷ Initialize an empty hash-map
8:     G_minfill = Greedy_MinFill(G, 1)
9:     C_minfill = BronKerbosch(G_minfill, ∅, V(G), ∅)
10:    best_tts = TableSize(C_minfill)                   ▷ Use minfill as upperbound.
11:    Enqueue(Q, s)                                     ▷ Q is priority queue of open steps
12:    while Q ≠ ∅ do
13:        n = Dequeue(Q)
14:        if n.R = ∅ then
15:            return n.G
16:        end if
17:        for all v ∈ n.R do
18:            m = CreateStep()
19:            m.G = IntroduceFillins(n.G, n.R, v)
20:            m.R = n.R \ FindSimplicials(m.G[n.R])
21:            m.C = UpdateCliques(n.G, m.G, n.C)
22:            m.tts = TableSize(m.C)
23:            if m.tts ≥ best_tts then                  ▷ Upperbound pruning
24:                continue
25:            else if m.R = ∅ then
26:                best_tts = m.tts                      ▷ Update upperbound
27:            end if
28:            if map(m.R) ≤ m.tts then                  ▷ Prune using hash-map
29:                continue
30:            end if
31:            map(m.R) = m.tts
32:            RemoveFromQueue(Q, m.R)                   ▷ Remove step q ∈ Q where q.R = m.R
33:            Enqueue(Q, m)
34:        end for
35:    end while
36: end function
```

Algorithm 12 Depth-First Search

```
1: function DFS(G)
2:     s =CreateStep()                                  ▷ Create an empty step structure.
3:     s.G = G
4:     s.R = V(G)\FindSimplicials(G)
5:     s.C =BronKerbosch(G, ∅, V(G), ∅)
6:     s.tts =TableSize(s.C)
7:     best_T =Greedy_MinFill(G, 1)                     ▷ best_T is a global variable.
8:     C_minfill =BronKerbosch(best_T, ∅, V(G), ∅)
9:     best_tts = TableSize(C_minfill)                  ▷ best_tts is a global variable.
10:    map = CreateHashMap()                            ▷ Create global hash-map.
11:    ExpandStep(s)
12:    return best_T
13: end function
14: function ExpandStep(n)
15:    for v ∈ n.R do
16:        m = CreateStep()
17:        m.G = IntroduceFillins(n.G, n.R, v)
18:        m.R = n.R\ FindSimplicials(m.G[n.R])
19:        m.C = UpdateCliques(n.G, m.G, n.C)
20:        m.tts = TableSize(m.C)
21:        if m.R = ∅ then
22:            if m.tts < best_tts then                 ▷ Update upper bound.
23:                best_tts = m.tts
24:                best_T = m.G
25:            end if
26:        else
27:            if m.tts ≥ best_tts then
28:                continue
29:            end if
30:            if map(m.R) ≤ m.tts then                 ▷ Prune using hash-map.
31:                continue
32:            end if
33:            map(m.R) = m.tts
34:            ExpandStep(m)                            ▷ Recursive call.
35:        end if
36:    end for
37: end function
```

CHAPTER 6

Reducing Expansions with Pivot Cliques

In this section we exploit a well known fact about triangulated graphs to reduce the number of successor steps generated when expanding a step in the optimal search algorithms. This should reduce number of steps generated and thus provide a performance improvement. The idea we introduce here chooses a clique for which successor steps will not be generated. We call this clique for a pivot clique and prove that any triangulation, including the optimal triangulation, can be obtained when reducing expansion using pivot clique.

> **Theorem 4** Let $G = (V,E)$ be a incomplete graph containing at least 3 nodes. For any partial elimination order $\sigma = (x_1, x_2, x_3, \ldots, x_{i-1})$ of G there exists at least two non-adjacent nodes x_i and x_j, such that the same triangulated graph G_T can be obtained regardless of whether x_i or x_j is eliminated next.

Proof Elimination of a node, introduction of fill-ins, corresponds to rendering it simplicial at the time of elimination in the resulting triangulated graph. In this graph there are always at least two non-adjacent simplicial nodes, this follows from theorem 1. Consequently, there are always at least two non-adjacent nodes that can be made simplicial. Thus, there must exist nodes x_i and x_j that are non-adjacent such that the same triangulated graph can be obtained, regardless of whether x_i or x_j is eliminated next. □

This knowledge about partial elimination orders can be applied directly to DFS and BFS with only minor changes to the algorithms, as shown in algorithm 13 and explain further. Recall that a step in both of these algorithms represents a partial elimination order. And it follows from theorem 4 that there are always at least two non-adjacent nodes leading to any triangulation, including the optimal triangulation.

As said and seen in algorithm 13, the changes required to BFS are minor. It is only required to reduce the expansion of the steps using the pivot strategy choosen to apply to BFS. Similar changes may be applied to DFS so that it can use a pivot strategy.

Algorithm 13 Best First Search with Pivot

```
 1: function BESTFIRSTSEARCH-PIVOT(G)
 2:     s =CREATESTEP()
 3:     Insert line 3-10 from algorithm 11
 4:     ENQUEUE(Q,s)
 5:     while Q ≠ ∅ do
 6:         n =DEQUEUE(Q)
 7:         Insert line 14-16 from algorithm 11
 8:         X = n.R\SELECTPIVOT(G_n,R_n,C_n)        ▷ Reduce expansion set X with pivot
 9:         for all v ∈ X do
10:             m =CREATESTEP()
11:             Insert line 18- 32 from algorithm 11
12:             ENQUEUE(Q,m)
13:         end for
14:     end while
15: end function
```

Suppose we run BFS on some graph G, where initially $|\mathcal{V}(G)| > 3$. Let n be the step which is expanded and $k = |n.R|$, a successor step m_i is generated for each node $u_i \in n.R,\ 1 \leq i \leq k$, where $n.R$ denotes nodes not eliminated in step n, i.e. remaining nodes. There are two non-adjacent simplicial nodes leading to an optimal triangulation. So, it is not necessary create a successor step m_i for every node $u_i \in n.R,\ 1 \leq i \leq k$.

Now we choose some singleton subset $C_p = \{v\}$ of $n.R$ and instead only create a successor step m_i for every node $u_i \in n.R \setminus C_p,\ 1 \leq i \leq l$, where $l = |n.R \setminus C_p|$, at most one of the two nodes leading to any solution will be excluded, including the optimal solution.

In fact, further removal is possible since we know that the two nodes are non-adjacent we can choose C_p to be any clique in G, because two non-adjacent nodes cannot both be in the same clique. So, rather than having the possibility to remove a single node at a step it can be generalized to an entire clique. This is fairly convenient as cliques are already maintained and therefore readily available. Because of this choosing a clique, which is the largest subset of $n.R$ is trivial and does not require much additional computation.

6.1 The Pivot Clique Selection Algorithm

Pivot selection requires that an additional set of nodes is maintained, namely the set which will be expanded in the successor step m_i, denoted X_m. It is important to note that the algorithm alters the set which is expanded $X_m = n.R \setminus C_p$, rather than set of remaining nodes $n.R$. If nodes were removed from $n.R$ information about the graph could easily be lost, since nodes may have overlapping cliques.

Algorithm 14 shows how a pivot clique is selected. Here, the strategy is to select the largest intersecting clique c.

According to theorem 1 we require for correctness that there at least three nodes in the remaining graph, or rather $|n.R| > 3$. However, the algorithm does not require a check for this condition as shown in line 4. This is due to the fact that three remaining nodes would become simplicial and simply removed (as done in BFS and DFS). Subsequently, $m.R = \emptyset$ and the branch terminates, since all nodes have been eliminated from G.

The complexity of algorithm 14 is linear in the number of cliques $|C|$ w.r.t. the for-loop in line 5 and intersection (line 6) is linear in the number of bits of each clique.

Algorithm 14 MaxSize Pivot Selection

```
 1: function SELECTPIVOT_MaxSize(G, R, C)
 2:     max = 0                                  ▷ Max. cardinality of intersection.
 3:     pivot = ∅
 4:     if |R| ≥ 3 then                          ▷ R is the set of remaining nodes.
 5:         for c_i ∈ C do                       ▷ C is the set of maximal cliques.
 6:             if |c_i ∩ R| > max then
 7:                 max = |c_i ∩ R|
 8:                 pivot = c_i
 9:             end if
10:         end for
11:     end if
12:     return pivot                             ▷ The largest intersecting clique.
13: end function
```

6.2 Pivot Selection Criteria

There are a number of other ways by which some pivot clique can be selected for removal. In algorithm 14 the largest remaining clique is always chosen. Yet, depending on the input graph there are possibly better criteria for selecting a pivot clique such that it reduces by highest number of expansions. In addition, pivot selection could potentially be improved by using tie-breaking rules.

Selecting the largest clique yields benefits in general, since it potentially causes the fewest number of expansions in the successor step. Figure 6.1 illustrates pivot selection. Initially node A is eliminated, inducing the fill-in $\{F,C\}$, which forms the clique $P = \{C,F,G\}$. Moreover, the set of remaining nodes is now $R = \{A,B,C,D,E,F,G,H\} \setminus \{A\}$. The clique P is chosen as the pivot, since it is the largest clique which intersects with the set of remaining nodes. Now each node in $R \setminus P$ is expanded and new pivot cliques are potentially chosen in subsequent expansions. Note that initially every edge in the graph were the largest intersecting cliques, so any of these could have formed a pivot clique.

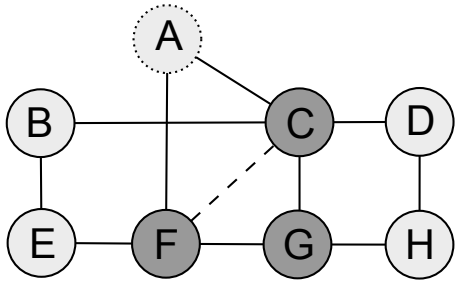


Figure 6.1: Selecting the clique with the largest intersection.

Another strategy is to choose a clique of minimum width and break ties by selecting the largest of such cliques. Here the minimum width is the cardinality of the family of the selected clique intersected with the set of remaining nodes. The idea behind such a strategy is to choose the clique that is most likely to lead to an optimal solution. This way we may generate more children, but we are less likely to generate as many optimal solutions in the long run. After all we are only interested in one optimal solutions at termination.

Here is a list of some of the pivot selection strategies we have tested. They all revolve around the idea of excluding as much as possible or excluding as many steps leading to a potential optimal solution as possible.

DynamicWidthSize:

The clique with minimum width is chosen as the pivot if the average width of the graph G is larger than the minimum width of G plus the number of remaining nodes. Otherwise the largest clique is chosen.

$$DynamicWidthSize(G) = \begin{cases} MinWidth(G) & \text{, average(width(G))} > min-width(G) + |R| \\ MaxSize(G) & \text{, otherwise} \end{cases}$$

DynamicWidthSizePk:

The clique is with minimum width chosen as the pivot if the number of remaining nodes is less the total number of nodes divided by k, where $k > 1$. Otherwise the largest clique is chosen as the pivot.

$$DynamicWidthSizePk(G) = \begin{cases} MinWidth(G) & \text{, } |R| < \frac{|\mathcal{V}(G)|}{k} \\ MaxSize(G) & \text{, otherwise} \end{cases}$$

First:

Chooses the first clique in the set C as the pivot. $c_1 \in C$

Last:

Chooses the last clique in the set C as the pivot. $c_n \in C : n = |C|$

MaxFill:

Choose the clique which adds the most fill-ins as the pivot. $\arg\max_{c \in C} \text{countFillins}(c)$

MaxFillFamily:

Choose the pivot clique that has a node, whose family adds the most fill-ins. $\arg\max_{c \in C} \text{countFillins}(n) : n \in fa(c)$

MaxSize:

Chooses the largest clique as the pivot. $\arg\max_{c \in C} size(c)$

MaxSizeMaxFill:

Chooses the pivot clique that has the largest size, breaking ties by choosing the clique which adds more fill-ins.

$$MaxSizeMaxFill(G) = \begin{cases} c_i & , size(c_i) > size(c_j) : \forall c_i, c_j \in \mathcal{C}(G) \wedge c_i \neq c_j \\ c_i & , size(c_i) = size(c_j) : \forall c_i, c_j \in \mathcal{C}(G) \wedge c_i \neq c_j \wedge \\ & \quad countFill-ins(c_i) > countFill-ins(c_j) \end{cases}$$

MaxSizeMaxWidth:

Chooses the pivot clique that has the largest size, breaking ties by choosing the clique that has the largest width.

$$MaxSizeMaxWidth(G) = \begin{cases} c_i & , size(c_i) > size(c_j) : \forall c_i, c_j \in \mathcal{C}(G) \wedge c_i \neq c_j \\ c_i & , size(c_i) = size(c_j) : \forall c_i, c_j \in \mathcal{C}(G) \wedge c_i \neq c_j \wedge width(c_i) > width(c_j) \end{cases}$$

MaxSizeMinFill:

Chooses the pivot clique that has the largest size, breaking ties by choosing the clique that adds the least number of fill-ins.

$$MaxSizeMinFill(G) = \begin{cases} c_i & , size(c_i) > size(c_j) : \forall c_i, c_j \in \mathcal{C}(G) \wedge c_i \neq c_j \\ c_i & , size(c_i) = size(c_j) : \forall c_i, c_j \in \mathcal{C}(G) \wedge c_i \neq c_j \wedge \\ & \quad countFill-ins(c_i) < countFill-ins(c_j) \end{cases}$$

MaxSizeMinFillFamily:

Chooses the pivot clique that has the largest size, breaking ties by choosing F; the clique with a node whose family adds the least number of fill-ins.

$$MaxSizeMinFillFamily(G) = \begin{cases} c_i & , size(c_i) > size(c_j) : \forall c_i, c_j \in \mathcal{C}(G) \wedge c_i \neq c_j \\ c_i & , size(c_i) = size(c_j) : \forall c_i, c_j \in \mathcal{C}(G) \wedge c_i \neq c_j \wedge c_i isF \end{cases}$$

MaxSizeMinWidth:

Chooses the pivot clique that has the largest size, breaking ties by choosing the clique that has minimum width.

$$MaxSizeMinWidth(G) = \begin{cases} c_i & , size(c_i) > size(c_j) : \forall c_i, c_j \in \mathcal{C}(G) \wedge c_i \neq c_j \\ c_i & , size(c_i) = size(c_j) : \forall c_i, c_j \in \mathcal{C}(G) \wedge c_i \neq c_j \wedge width(c_i) < width(c_j) \end{cases}$$

MaxWidth:

Chooses the the clique that has the maximum width of amount all remaining cliques. $\arg\max_{c \in C} MaxWidth(c)$

Middle:
Always chooses the clique in the middle as the pivot. $c_m \in C : m = \frac{|C|}{2}$

MinFill:
Chooses the clique that adds the least number of fill-ins as the pivot. $\arg\min_{c \in C} countFill - ins(c)$

MinFillFamily:
Chooses the pivot clique that has a node, whose family adds the least fill-ins. $\arg\min_{c \in C} countFill - ins(n) : n \in fa(c)$

MinFillSingle:
Chooses the pivot clique, which has a node that introduces the least fill-ins. $\arg\min_{c \in C} countFill - ins(c)$

MinSizeMaxFillFamily:
Chooses the pivot clique with the smallest size, breaking ties by choosing Q; the clique with a node whose family adds the most fill-ins.

$$MinSizeMaxFillFamily(G) = \begin{cases} c_i & , size(c_i) < size(c_j) : \forall c_i, c_j \in \mathcal{C}(G) \wedge c_i \neq c_j \\ c_i & , size(c_i) = size(c_j) : \forall c_i, c_j \in \mathcal{C}(G) \wedge c_i \neq c_j \wedge c_i isQ \end{cases}$$

MinWidth:
Chooses a pivot clique of minimum width. $\arg\min_{c \in C} |fa(c) \cap R|$, where C is the set of cliques and R is the set of remaining nodes.

MinWidthMaxSize:
Chooses the pivot clique of the smallest width, breaking ties by choosing the clique that has maximum size.

$$MaxSizeMaxFill(G) = \begin{cases} c_i & , width(c_i) < width(c_j) : \forall c_i, c_j \in \mathcal{C}(G) \wedge c_i \neq c_j \\ c_i & , width(c_i) = width(c_j) : \forall c_i, c_j \in \mathcal{C}(G) \wedge c_i \neq c_j \wedge size(c_i) < size(c_j) \end{cases}$$

ROT13:
Select some random pivot clique.

6.3 Evaluation of the pivot strategies

We believe that the pivot clique often ends up being the same in many steps, since eliminating a single node only causes minor changes to the graph at each step.

In general for sparse graphs the *MaxSize* strategy works well. We suppose this is because there are many equally good pivot cliques and many optimal solutions, i.e. many which intersect with the same number of remaining nodes. In other words, most of them are equally good pivot candidates.

In contrast, for denser graphs the *MinWidth* strategy has a better payoff. Opposite to sparse graphs, the clique sizes vary more and as a consequence not all largest pivot cliques are equally good.

From the tables: table.6.1, table.6.2, table.6.3, table.6.4 and table.6.5 It is possible to see that both *MaxSize* and *MinWidth* are the best strategies for finding the pivot clique. Used alone or in combination with other strategy.

But between *MaxSize* and *MinWidth*, in our tests in graphs of size 30 (table.6.1, table.6.2 and table.6.3), it is possible to see that *MinWidth* is better than *MaxSize* in denser graphs while *MaxSize* is better than *MinWidth* in sparser graphs. This difference gets more significant in larger graphs as we can see in the table.6.5 where *MinWidth* is more than two times faster than *MaxSize* in a sparse graph of size 60.

This happens because, in sparse graphs, the size of each cliques is more significant, this is, the standard deviation of the sizes of the cliques is large and so *MaxSize* strategy is more effective, since this strategy chooses the clique with the larger size. While in denser graphs, it is the width of each clique that is more significant and so *MinWidth* strategy is more efficient.

To get the best of both strategies; *MaxSize* and *MinWidht*, we combine the two in the *MaxSizeMinWidth* and *MinWidthMaxSize* and in fact these to strategies worked out as aspected and the results are shown in the following tables.

For tables containing all strategies, see appendix B.

	Running time in sec		
strategy	Dense	Mid	Sparse
MaxSize	0.46	77.91	743.38
MaxSizeMaxWidth	0.45	79.21	781.70
MaxSizeMinWidth	0.45	79.21	736.80
MinWidth	0.38	76.34	1132.76
MinWidthMaxSize	0.37	75.93	1107.40

Table 6.1: Table of best average results for DFS with Pivot in the different graph densities of size 30

	Running time in sec		
strategy	Dense	Mid	Sparse
MaxSize	0.12	25.19	272.48
MaxSizeMaxWidth	0.12	25.00	272.51
MaxSizeMinWidth	0.11	24.25	265.96
MinWidth	0.12	26.85	410.65
MinWidthMaxSize	0.11	25.04	397.92

Table 6.2: Table of best average results for DFS with Pivot and Oracle in the different graph densities of size 30

	Running time in sec		
strategy	Dense	Mid	Sparse
MaxSize	0.45	76.06	743.72
MaxSizeMaxWidth	0.49	76.84	4767.58
MaxSizeMinFill	0.51	80.24	785.95
MaxSizeMinWidth	0.48	76.51	779.95
MinWidth	0.40	74.29	881.27

Table 6.3: Table of best average results for BFS with Pivot in the different graph densities of size 30

Running time in sec			
strategy	Dense	Mid	Sparse
MaxSize	0.43	77.89	794.24
MaxSizeMinFill	0.50	82.15	801.37
MaxSizeMaxWidth	0.48	78.89	780.65
MaxSizeMinWidth	0.45	78.14	784.50
MinWidth	0.36	75.56	916.45

Table 6.4: Table of best results for BFS with Pivot and Oracle in the different graph densities of size 30

Running time in sec	
strategy	time
MaxSize	27.12
MinWidth	11.26
MinWidthMaxSize	7.09
First	9.95

Table 6.5: Table of best results for BFS with Pivot and Oracle for dense graphs of size 60

CHAPTER 7

Coalescence Prediction using Transposition of PEOs

In this chapter we present a set of theorems for predicting coalescing, which we used in our own method dubbed "BFS-Oracle". Our work is inspired by a transposition "oracle" for perfect elimination orders presented in Chandran et al. (2003). Proofs of our theorems and correctness of our algorithms do not follow from the results presented in Chandran et al. (2003). But our theory, lemma, theorem, and corollary, presented in section 7.2, is inspired by the work presented in Chandran et al. (2003). We propose a method for predicting coalescence for an arbitrary number of nodes that are pairwise non-adjacent.

7.1 The Transposition Oracle for PEOs

In this section a lemma, theorem, corollary and corresponding proofs from Chandran et al. (2003, section 2) are reproduced for completeness, but using our notation for convenience. They seek to determine whether two elements in a perfect elimination order for a given chordal graph can be exchanged. In contrast to the results we present in the following section, and suggests that our work can be extended.

In this section we denote a perfect elimination order for a chordal graph, G, as a mapping $\sigma : \{i \in \mathbb{N} \mid i \leq |\mathcal{V}(G)|\} \mapsto \mathcal{V}(G)$. This is basically a mapping from order of elimination to respective nodes.

We denote the set of nodes with a higher position in the elimination order σ than i as $\mathcal{V}_\sigma(i,G) = \{v \mid v = \sigma(j) \wedge j > i\}$.

For convenience we write $nb_\sigma(i,v,G)$ to denote the neighbour-set of v with a position in σ higher than i, i.e. $nb_\sigma(i,v,G) = nb(v,G) \cap \mathcal{V}_\sigma(i,G)$.

> **Lemma 1** If b follows a in a perfect elimination order σ, $\sigma(i) = a$ and $\sigma(i+1) = b$, for a chordal graph G, where a and b are adjacent then $nb_\sigma(i+1,a,G) \subseteq nb_\sigma(i+1,b,G)$.

Proof Since b follows a in σ, $nb_\sigma(i,a,G) = nb_\sigma(i+1,a,G) \cup \{b\}$ must be true. It is also trivial to see that the neighbors of a with a higher position in the elimination order than a, denoted $nb_\sigma(i,a,G)$ must form a clique, otherwise a is not simplicial at the time of elimination, which it must be in a perfect elimination ordering σ for a chordal graph G. When $nb_\sigma(i+1,a,G) \cup \{b\}$ forms a clique, then b is adjacent to all members of $nb_\sigma(i+1,a,G)$, thus $nb_\sigma(i+1,a,G) \subseteq nb_\sigma(i+1,b,G)$. □

> **Theorem 5** If b follows a in a perfect elimination order $\sigma = (\ldots a,b,\ldots)$, $\sigma(i) = a$ and $\sigma(i+1) = b$, for a chordal graph G, then a and b can be transposed in σ, so that $\sigma' = (\ldots b,a,\ldots)$ with $\sigma'(i) = b$ and $\sigma'(i+1) = a$ is also a perfect elimination ordering for G, if and only if a and b are non-adjacent or $nb_\sigma(i+1,a,G) = nb_\sigma(i+1,b,G)$.

Proof σ' is a perfect elimination order for G if every $nb'_\sigma(j,v,G)$ where $\sigma'(j) = v$ forms a clique. If for some j and $nb'_\sigma(j,v,G)$ where $\sigma'(j) = v$ is not a clique, then v is not simplicial in G when it is eliminated, which it must be in order for σ' to be a perfect elimination order for G.

To complete this proof we need to show that $nb'_\sigma(i,b,G)$ and $nb'_\sigma(i+1,a,G)$ are cliques. We know that σ is a perfect elimination order, which implies that $nb_\sigma(j,v,G)$ where $\sigma(j) = v$ is a clique. Consequently this proves that $nb'_\sigma(j,v,G)$ where $\sigma'(j) = v$ is a clique for all $j \notin \{i,i+1\}$.

If a and b are non-adjacent in G, then $nb'_\sigma(i+1,a,G) = nb_\sigma(i,a,G)$ and $nb'_\sigma(i,b,G) = nb_\sigma(i+1,b,G)$ are both cliques, because elimination of b cannot change the neighbour-set of a if they are non-adjacent in G.

If a and b are adjacent in G and $nb_\sigma(i+1,a,G) = nb_\sigma(i+1,b,G)$, then we know that $nb'_\sigma(i+1,a,G) = nb_\sigma(i+1,b,G)$ because $\mathcal{V}''_\sigma(i+1,G) = \mathcal{V}'_\sigma(i+1,G)$, which shows that $nb'_\sigma(i+1,a,G)$ is a clique. We know that $nb'_\sigma(i,b,G) = nb_\sigma(i+1,b,G) \cup \{a\}$ and since $nb_\sigma(i+1,b,G) = nb'_\sigma(i+1,a,G)$, it follows that $nb'_\sigma(i,b,G) = nb'_\sigma(i+1,a,G) \cup \{a\}$ is a clique because we have already shown that $nb'_\sigma(i+1,a,G)$ is a clique. Therefore $nb'_\sigma(i+1,a,G) \cup \{a\}$ is a clique.

If a and b are adjacent in G and $nb_\sigma(i+1,a,G) \subset nb_\sigma(i+1,b,G)$ then $nb'_\sigma(i,b,G)$ is not a clique, because $a \in nb'_\sigma(i,b,G)$ and a is non-adjacent to at least one node in $nb_\sigma(i+1,b,G)$ since $nb_\sigma(i+1,a,G) \subset nb_\sigma(i+1,b,G)$. □

Corollary 2 If b follows a in a perfect elimination order $\sigma = (\ldots a,b\ldots)$, $\sigma(i) = a$ and $\sigma(i+1) = b$, for a chordal graph G, then a and b can be transposed in σ, so that $\sigma' = (\ldots b,a\ldots)$, $\sigma'(i) = b$ and $\sigma'(i+1) = a$, if and only if a and b is non-adjacent or $|nb_\sigma(i,a,G)| = |nb_\sigma(i+1,b,G)| + 1$.

Proof If a and b are non-adjacent the result follows from theorem 5. If a and b are adjacent it follows from lemma 1 that $nb_\sigma(i+1,a,G) = nb_\sigma(i+1,b,G)$ if $|nb_\sigma(i+1,a,G)| = |nb_\sigma(i+1,b,G)|$ because $nb_\sigma(i+1,a,G) \subset nb_\sigma(i+1,b,G)$ which means that $nb_\sigma(i+1,a,G)$ cannot have elements not in $nb_\sigma(i+1,b,G)$, so if both are of the same size they must be equal. We also know that $nb_\sigma(i+1,a,G) \cup \{b\} = nb_\sigma(i,a,G)$ and $nb_\sigma(i+1,b,G) = nb_\sigma(i,b,G)$, which means we can say that $|nb_\sigma(i+1,a,G)| = |nb_\sigma(i,a,G)| - 1$ and $|nb_\sigma(i+1,b,G)| = |nb_\sigma(i,b,G)|$, which in turn proves that the equation $|nb_\sigma(i,a,G)| = |nb_\sigma(i+1,b,G)| + 1$ is equivalent to $|nb_\sigma(i+1,a,G)| = |nb_\sigma(i+1,b,G)|$. □

7.2 Coalescence Prediction

From theorem 1, which states that a non-complete triangulated graph with more than two nodes always has at least two non-adjacent simplicial nodes, it follows that there exists more than one perfect elimination order for any triangulated graph. This result can also be obtained from corollary 2, which suggests under some conditions the order in which some nodes are eliminated does not matter.

In this section we propose a method for predicting coalescence. This work is inspired by corollary 2 (Chandran et al., 2003), but as previously mentioned the correctness of our results do not follow from 2. Rather, we present independent proofs for our results.

Lemma 2 Let G be a graph, then a set of nodes $Z \subset \mathcal{V}(G)$ is pairwise non-adjacent $\{a,b\} \notin \mathcal{E}(G)$ and $\forall a,b \in Z$, if and only if Z is also pairwise non-adjacent in any partially triangulated graph of G, where Z has been eliminated, $\{a,b\} \notin \mathcal{E}(G_T)$ and $\forall a,b \in Z$, where G_T is a partial triangulation obtained by elimination of Z in any order.

Proof Elimination of $a \in Z$ can only introduce fill-ins to nodes in $fa(a,G)$ and since a is non-adjacent to all other nodes in Z, $fa(a,G) \cap Z = \emptyset$, it follows that elimination of a cannot introduce a fill-in to any node in Z. Thus, if nodes in Z are pairwise non-adjacent, then so are nodes in G_T after elimination of Z. The other direction is trivial to prove as elimination only introduces fill-ins, thus if nodes in Z are pairwise non-adjacent in G_T, where Z has been eliminated, Z is also pairwise non-adjacent in G. □

Theorem 6 Let G be a graph and $Z \subset \mathcal{V}(G)$, where $\{a,b\} \notin \mathcal{E}(G)$ and $\forall a,b \in Z$, be a set of pairwise non-adjacent nodes. The set of fill-ins F introduced by elimination of Z is the same regardless of the order in which Z is eliminated.

Proof Let F_a denote the fill-ins introduced by elimination of $a \in Z$, then $F = \bigcup_{i \in Z} F_i$ is the set fill-ins introduced by elimination of all nodes in Z. The set of fill-ins F_a introduced by elimination of $a \in Z$ can only be changed by the elimination of some $b \in Z$ if *(i)* b is a neighbour of a, *(ii)* F_b introduces a new neighbour to a or *(iii)* if there exists some fill-in $\{c,d\} \in F_b$, where $c,d \in fa(a,G)$.

(i) is not the case since all nodes in Z are pairwise non-adjacent, and it follows from lemma 2 that this property is maintained during elimination, $b \in Z$ cannot be a neighbour of a.

In the proof of lemma 2 we also showed that no elimination of $b \in Z$ can introduce a fill-in such that $a \in Z$ gets a new neighbour, thus (ii) cannot occur.

So, the only way, (iii), for which F_a can be changed by the elimination of some node $b \in Z$ is if b introduces a fill-in, f, between two neighbors of a. However, such a change would only move f from F_a to F_b, which means that the fill-in f would still be in $F = \bigcup_{i \in Z} F_i$. Thus, the order in which nodes in Z are eliminated does not change the set of fill-ins added. □

Corollary 3 Let G be a graph, and $Z \subset \mathcal{V}(G)$, where $\{a,b\} \notin \mathcal{E}(G)$ and $\forall a,b \in Z$, be a set of pairwise non-adjacent nodes. Then the set of triangulated graphs of G that can be obtained from the partial triangulation G_T, where all the nodes in Z have been eliminated, are the same regardless of what order Z was eliminated in.

Proof It follows from theorem 6 that the set of fill-ins, F, introduced to G by the elimination of Z, is the same regardless of the order in which Z is eliminated. And since the partial triangulation $G_T = (\mathcal{V}(G), \mathcal{E}(G) \cup F)$, is produced by adding F to G, G_T must also be the same regardless of what order the nodes in Z were eliminated. □

Corollary 2 suggests that theorem 6 could be extended to also cover some adjacent nodes, a and b, in G if $|nb(a,G')| = |nb(b,G')|$, where fill-ins for a have been introduced in G'. But in pratical applications this is rarely the case, and contrary to theorem 6 this property might be harder to generalize.

Corollary 3 can be used to predict coalescence when searching in the space of all elimination orders. In figure 7.1 the search tree for all triangulations of a graph G, with nodes $a,b,c \in \mathcal{V}(G)$, where a and b are non-adjacent and c is adjacent to a and b, is illustrated. Notice, that the elimination order (b,a) is never explored. In this small example coalescence prediction does not benefit much, but when there are more nodes this will give substantial improvements in efficiency, which is apparent in the test results.

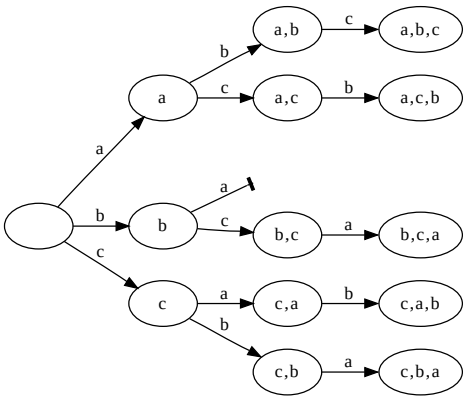


Figure 7.1: The search tree with oracle coalesce prediction.

7.3 Best First Search with Oracle

Corollary 3 can be exploited when searching in the space of all elimination orders. We already have coalescing, which

ensures that only one of the partial elimination orders (a,b) and (b,a) will be expanded. So, corollary 3 can only be used to predict coalescing. This means that we will reduce the number of successor steps generated by never inserting steps into the queue, which would coalesce in any case.

In algorithm 15 we extend the step structure with a set, X, the expansion set. So, $m.X$ is the expansion set of step m. The expansion set is the nodes for which successor steps should be generated. During expansion of a step we maintain a set called E_x, the expendable set, which is initialized in line 9. The expendable set E_x contains the nodes for which successor steps need not be generated, unless they are in $fa(v,m.G)$, i.e. the family of the node v just eliminated. We add a node, a, to the expendable set, E_x, *(i)* if it was not in the expansion set of the step currently being expanded $a \notin n.X$ or *(ii)* if some successor step m, where $v = a$, has already been generated.

Initially, the expansion set, $s.X = s.R$, will be the set of remaining nodes, $s.R$. When the initial step, s, is expanded, the expendable set, E_x, will be initialized to $\emptyset$ in line 9. Thus, when the successor step m_v, where v is eliminated is generated, a node a can only be in E_x if some other successor step, m_a, where a was eliminated and $v \notin E_x$ have been generated. If this is the case and $a \notin fa(v,G[m.R])$, it follows from corollary 3 that we are not required to explore both steps with the partial elimination orders (a,v) and (v,a), respectively. And since a step, m_a, with partial elimination order (a) and $v \in m_a.X$, there is no need to include a in the expansion set $m_v.X$.

When we are not expanding the initial step s there will be cases where E_x is initialized to some non-empty set in line 9. However, a node a will only be in E_x initially for the expansion of step n with partial elimination order $(p_1,\ldots,p_k)$, if a is not in $n.X$. This implies that a is non-adjacent to $p_j,\ldots,p_k$, for some $j < k$. And thus, by corollary 3 there is no need to explore a in the next expansion of successor step m, where v was eliminated if $a \notin fa(v,G[m.R])$. Because this will coalesce with some other step m' with the partial elimination order $(p_1,\ldots,a,p_j,\ldots,p_k)$, which must have been generated with $v \in m'.X$.

Algorithm 15 Best First Search with Oracle

```
1:  function BESTFIRSTSEARCH-ORACLE(G)
2:      s =CREATESTEP()
3:      Insert line 3-10 from algorithm 11
4:      s.X = s.R                          ▷ Initial expansion set contains all remaining nodes
5:      ENQUEUE(Q,s)
6:      while Q ≠ ∅ do
7:          n =DEQUEUE(Q)
8:          Insert line 14-16 from algorithm 11
9:          E_x = n.R \ n.X                ▷ Initialize X to nodes that need not be expanded
10:         for all v ∈ n.X do             ▷ Notice that we use the expansion set n.X here
11:             E_x = E_x ∪ {v}            ▷ Add v to node that can be excluded X if not in fa(v)
12:             Insert line 18- 32 from algorithm 11
13:             m.X = m.R \ (E_x \ fa(v,m.G[m.R]))   ▷ Reduction of the expansion set for m
14:             ENQUEUE(Q,m)
15:         end for
16:     end while
17: end function
```

By analyzing algorithm 15 one can see that if $|fa(v,m.G[m.R])|$ on line 13, is 0, this should cut the running in half. However, it stands to reason that $|fa(v,m.G[m.R])| \neq 0$, but compared to $|m.R|$ it is fair to say that $|fa(v,m.G[m.R])|$ is close 0, especially for sparse graphs. And then it should not be unreasonable to expect a speedup of around 50 percent, which is also what practical application shows.

Moreover, it is possible to apply this improvement to the DFS algorithm presented earlier. A compar-

ison pf depth first search and best first search, with and without this improvement, is presented in chapter 10, along with a comparison of all the other improvements. It is also worth noting that this improvement, while it does not reduce the search space (without coalescing) to all chordal graphs only, it does provide a reduction of the search space, which might also be useful when searching for a triangulation with another optimality criterion, such as minimum treewidth or minimum number of fill-ins.

7.4 Summary

The improvement proposed in this chapter provide a significant speedup. Whilst it may be possible to extend theorem 6 to cover some adjacent nodes, these improvements do not reduce the search space from all elimination orders into just the set of all distinct triangulations, which the results in Chandran et al. (2003) suggest would be a significant reduction. This indicates that research into other kinds of search algorithms, which only consider the space of triangulations might lead to further improvement.

CHAPTER 8

Maximal Prime Subgraph Decomposition

Maximal prime subgraph decomposition (MPD) essentially boils down to finding ways to decompose a graph into subgraphs such that each subgraph can be triangulated independently. In other words, it introduces the divide-and-conquer strategy to the realm of triangulation. This may yield several advantages, such as parallelization, incremental triangulation and smaller problem sizes.

That being said, not all graphs admits decomposition, so it is not guaranteed to yield benefits in all cases. In addition, computing an MPD introduces extra overhead, which must also be taken into account. For this reason, it would be useful being able to efficiently discern whether it is worth computing an MPD or if a graph even allows decomposition.

Olesen and Madsen (2002) present methods for decomposing a Bayesian network into an MPD. This algorithm is used in Flores and Gamez (2003) to decompose a graph during triangulation. However, their work shows that there is not many decompositions to find, so we will attempt decomposition during each step of the triangulation process in algorithm 11. For this purpose we present a minimalistic and practical method for quickly finding decompositions in graphs without simplicial nodes, for which the maximal cliques are known.

8.1 Finding Decompositions

We say that graph $G = (V, E)$ admits decomposition if some clique, C, splits the graph, such that $G[V \setminus C]$ is disconnected. When we say that two sets of nodes A and B in $G[V \setminus C]$ are disconnected, we shall assume that both A and B are non-empty, it also implies that A and B are disjoint, and that all paths from a node in A to a node in B in the graph G passes through at least one node in C.

Definition 6 (Separator) The clique C in G is a separator if $G[V \setminus C]$ is disconnected, and we say that G admits decomposition with respect to the separator C.

Definition 7 (Prime Graph) A prime graph is a graph that does not admit decomposition, i.e. has no separators.

Definition 8 (Part) We say that A and B are two parts of G with respect to a separator C if A and B are two disjoint and disconnected sets of nodes in $G[V \setminus C]$.

The methods presented in Tarjan (1985) and Olesen and Madsen (2002) initially use a minimal triangulation to find separators. The method in Olesen and Madsen (2002) the junction tree for some minimal triangulation is created and separators are merged until all separators are complete in the original graph, the result is called an MPD-junction tree. It is possible to show that the leaves of an MPD-junction tree can be eliminated first, which is a great property for triangulation.

In Flores and Gamez (2003) the method from Olesen and Madsen (2002) is used for finding decomposition after which the subgraphs are triangulated. However, in Flores and Gamez (2003) it is concluded that most of the subgraphs found by decomposition are initially simplicial. This leads us to

focus on finding decomposition in graphs without simplicial nodes. In reality such decompositions are not common, so we will not maintain an MPD-junction tree, but rather just find the smallest leaf of such a tree.

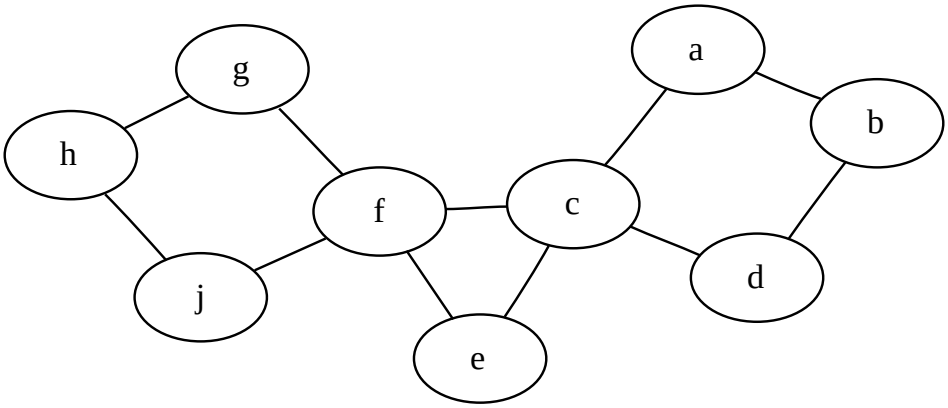
Lemma 3 If a graph admits decomposition, the separator must be a subset of one of the maximal cliques of the graph.

Proof By definition 6, a separator must be a clique, and since all cliques are subsets of maximal cliques, so must a separator. If a separator is not a subset of a maximal clique, it is not a clique. □

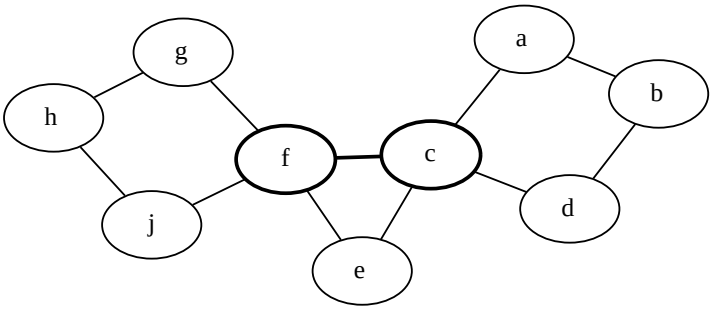
Theorem 7 A graph G without simplicial nodes admits decomposition if and only if some maximal clique C in G is a separator for G.

Proof (Theorem 7) Let $G = (V,E)$ be a graph without simplicial nodes, which has a separator C, where C is not a maximal clique in G, such that $G[V \setminus C]$ contains at least two disconnected sets of nodes A and B (A and B are parts of G with respect to the separator C).

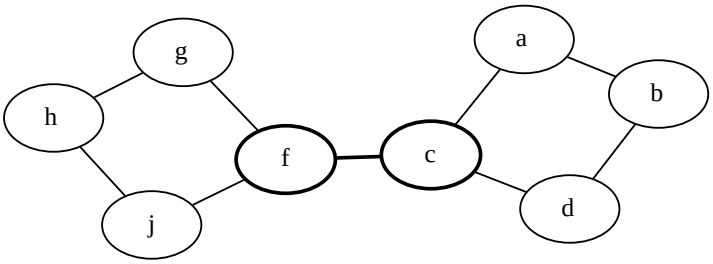
Then if $A \cup C$ is a clique in G, G contains the simplicial node A, which gives a contradiction. So, we assume that $A \cup C$ is not a clique, which implies that expanding C to be a maximal clique will not make A a subset of C. The same argument holds for B, thus when there is a non-maximal clique separator for a graph G, which has no simpicial nodes, then there is also a maximal clique separator for G. □



(a) The initial graph



(b) Separator that splits the graph into three parts



(c) Elimination of node 'e'

Figure 8.1: Graph with simplicial nodes that admits decomposition.

Figure 8.1 illustrates a graph that admits decomposition. The clique $S = \{f, c\}$ is a separator that splits the graph into three parts, $A = \{h, g, j\}$, $B = \{a, b, d\}$ and $C = \{e\}$. However, $C \cup S$ is a clique, so C is simplicial, as explained in the proof above. If we remove e from the graph G on figure 8.1, both $S_1 = \{f\}$ and $S_2 = \{c\}$ will be separators for G. But as shown in the proof, expanding the separator up to $S = \{f, c\}$ will always be possible as long as the separated parts are not simplicial.

By theorem 7 we can find any decomposition permitted by a graph without simplicial nodes by determining if the some maximal clique is a separator. Moreover, if we wish to check if elimination of some node in the graph introduces a separator, it is enough to test if one of the new cliques is a separator, since elimination preserves paths and does not remove maximal cliques, rather, they are merged.

8.2 Exploiting Decomposition

The fact that some graphs admit decomposition can be exploited to reduce the number of elimination orders that need to be explored. In fact, as stated and proved below, we can say that a part of a graph (see definition 8) can be eliminated first.

> **Theorem 8** Let $G = (V,E)$ be an incomplete graph with at least three nodes, for which $C = \{c_1,\ldots,c_k\}$ is a separator, such that $G[V \setminus C]$ contains two disconnected sets of nodes $A = \{a_1,\ldots,a_i\}$ and $B = \{b_1,\ldots,b_j\}$. Then an elimination order $\sigma = (a_1,\ldots,a_i,b_1,\ldots,b_j,c_1,\ldots,c_k)$ exists for every triangulation G_T of G.

Proof From theorem 4 it follows that any clique can be eliminated last. Thus, we can say that there exists an elimination order of the form $\sigma' = (n_1,\ldots,n_{i+j},c_1,\ldots,c_k)$ where $n_1,\ldots,n_{i+j} \in A \cup B$ for every triangulation of G. However, since there are no paths between A and B in $G[V \setminus C]$, it follows that elimination of any node in $A \cup B$ cannot create such a path. That is, the order in which nodes from A and B are eliminated with respect to one another does not matter. Thus, an elimination order of the form $(a_1,\ldots,a_i,b_1,\ldots,b_j,c_1,\ldots,c_k)$ must exist for every triangulation of G. □

It is fairly obvious that theorem 8 can be exploited to limit the exploration of elimination orders. If the graph $G = (V,E)$ decomposes with respect to C, such that two sets of nodes A and B are disconnected in $G[V \setminus C]$, then there is no need to explore elimination orders where nodes from $C \cup B$ have been eliminated before all the nodes in A have been eliminated. This provides a significant reduction in the number of elimination orders that needs to be explored.

8.3 Best First Search with Maximal Prime Subgraph Decomposition

The idea for implementing best first search (algorithm 11) with prime subgraph decomposition is to look for decompositions every time a node is eliminated from the graph. Then decompositions are used to limit the number of expansions to be explored, as is possible from theorem 8.

Algorithm 16 takes a graph G without simplicial nodes, a set of maximal cliques, separator candidates, C, and the smallest part admitted using any maximal clique not in C as the separator. The algorithm returns the smallest part of the graph.

This is done by iterating over the set of separator candidates (line 2). Then finding the remaining nodes, R, when this separator is removed (line 3). Once this is done, we choose some node $n \in R$ and find all nodes, I, in $G[R]$ that can be reached from n. This is done using the REACHABLE(...) function, which can be implemented efficiently using breath first search. Then on line 7 we test if $G[R]$ is disconnected, i.e. $I \neq \mathcal{V}(G) \setminus c$, and the cardinality of the part, I, found is smaller than S. If both of these tests hold, then $S = I$. On line 10 we remove the part, I, just found from R, such that when we choose another $n \in R$, provided that $G[R]$ was disconnected, another part will be found. This is carried out until all parts of the graph have been found. Consequently, the smallest part, S, has been found.

Algorithm 17 lists the pseudo code for best first search with maximal prime decomposition. In this algorithm the step structure has been extended with a set representing the subgraph part S. $n.S$ denotes the smallest part of the remaining graph for step n, w.r.t. some separator. Note, that the separator is not in $n.S$, so this set can be used for expansion on line 9. This follows from theorem 8. This is also why the algorithm looks for the smallest part of the graph, so that expansion is limited as much as possible. The subgraph part set, $m.S$, is computed by calling DECOMPOSE(...) on the remaining graph, including the new maximal cliques as separator candidates and smallest part admitted using another maximal clique as input. Note, that in cases where all nodes of a smallest part have been eliminated the algorithm searches for decompositions with all maximal cliques as separator candidates. It should also be noted

Algorithm 16 Find Smallest Prime Subgraph using Maximal Cliques

```
1: function DECOMPOSE(G, C, S)
2:     for all c ∈ C do
3:         R = V(G) \ c                    ▷ Remaining nodes, e.g. without the separator candidate
4:         repeat
5:             Let n be a node in R
6:             I = REACHABLE(G[R], n)                    ▷ Find reachable nodes from n
7:             if I ≠ V(G) \ c ∧ |I| < |S| then          ▷ If G[R] is disconnected and |I| < |S|
8:                 S = I
9:             end if
10:            R = R \ I                ▷ Remove the part we just found, so other parts can be found.
11:        until R ≠ ∅
12:    end for
13:    return S                                  ▷ Return smallest subgraph without separator.
14: end function
```

that the set of fill-ins added, the set of nodes with new fill-ins, as well as the set of new cliques are not expensive to find, as they are computed in algorithm 10, and thus for a practical implementation UPDATECLIQUES(...) would require modification.

Algorithm 17 Best First Search with Decomposition

```
1: function BESTFIRSTSEARCH-MPD(G)
2:     s = CREATESTEP()
3:     Insert line 3-10 from algorithm 11
4:     s.S = DECOMPOSE(s.G[s.R], s.C, s.R)          ▷ Attempting initial decomposition
5:     ENQUEUE(Q, s)
6:     while Q ≠ ∅ do
7:         n = DEQUEUE(Q)
8:         Insert line 14-16 from algorithm 11
9:         for all v ∈ n.S do                              ▷ Limit expansion to n.S
10:            Insert line 18- 32 from algorithm 11
11:            if n.S ∩ m.R = ∅ then
12:                m.S = DECOMPOSE(m.G[m.R], m.C, m.R)
13:            else
14:                F = E(m.G) \ E(n.G)                    ▷ The set of fill-ins added.
15:                I = {v, u | {u, v} ∈ F}                ▷ The set of nodes with new fill-ins.
16:                NC = C ∈ m.C | C ∩ I ≠ ∅                          ▷ Find new cliques
17:                m.S = DECOMPOSE(m.G[m.R], NC, n.S ∩ m.R)   ▷ Find new decompositions
18:            end if
19:            ENQUEUE(Q, m)
20:        end for
21:    end while
22: end function
```

The improvements presented in chapter 6, about reduction of an expansion set with a pivot clique, cannot always be combined with maximal prime decomposition as the proof of theorem 8 relies on the ability to eliminate nodes from one clique last. However, reduction of the expansion set using a pivot clique, can still be done when $n.S = n.R$, i.e. no decomposition have been found. Practical

application shows that decomposition does not happen very often, and some dense graph does not admit decomposition during elimination.

The coalescing prediction presented in chapter 7 can be combined with algorithm 17. A pseudo implementation of algorithm 17 combined with coalesce prediction and pivot, when there is no decomposition, can be seen in appendix A. And with coalesce prediction and pivot selection, where there is no decomposition, decomposition actually starts to payoff. For more information see chapter 10 where the methods are compared.

Summary

The use of decomposition in algorithm 17 is relatively simple and minimalistic. The approach focuses on exploiting decomposition with as little overhead as possible. Similar work in Flores and Gamez (2003) indicates that the number of non-simplicial decompositions is usually low. In addition, unpublished work by Thorsten J. Ottosen indicates that the overhead of building and using an MPD-junction tree as introduced in Olesen and Madsen (2002) is too expensive.

Even with the rather small overhead incurred by algorithm 17, decomposition does not yield any advantages by itelf, even for graphs which admit an MPD. Rather it should be combined with coalescence prediction and pivot selection. Nonetheless, the concept does have value, since it opens up for more approaches to the problem of triangulation. It is also likely that algorithm 16 could be implemented more efficiently.

It might also be possible to develop some heuristic to reduce the number of times the graph is tested for decomposition. As it would not be a serious problem if a decompositions is missed once in a while, if such a heuristic dramatically reduces the number of fruitless decomposition checks. It is also likely that some efficient method for disproving the existence of a decomposition could be found.

CHAPTER 9

Reducing Expansion Using Graph Symmetry

In this chapter we will introduce the idea of using graph symmetry for reducing the number of successor steps generated. We will not reach any performance improvements, but show that graph symmetry is not very common.

9.1 Defining Node Equivalence

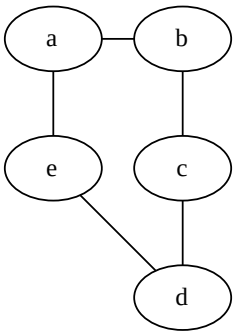


Figure 9.1: A graph with symmetry.

If we consider figure 9.1 we will notice that we will get the same kind of fill-ins and remaining graph regardless of what node is eliminated first. It simply does not matter whether we eliminate a, b, c, d or e first, it will introduce one fill-in and the remaining graph will have four node connected in a cycle without a chord. We say that the nodes are equivalent, which leads us to the following definition.

Definition 9 (Node equivalence) Two nodes a and b in a graph G are equivalent if exchanging their labels in a graph G' where all other nodes are unlabeled does not change anything.

If we study the search algorithms for optimal elimination orders, BFS and DFS, they will both generate a successor step for each node in the graph on figure 9.1. And as the nodes are equivalent, following definition 9, successor steps of these successor steps will also be generated. Coalescing will ensure of that successor steps will only be generated for one of the steps (a, b) and (b, a). But coalescing will not be every effective and the queue will have no effect as all the successor steps are equivalent and thus have the same total table size. This leads us to the following hypothesis.

We can address the shortcomings for the queue and coalescing map, by finding equivalent nodes before generating successor steps, and only generate one successor step for each set of equivalent nodes, thus limiting the number of computation steps and improving performance.

We will later show that in practical application this has a very limited effect as the number of equivalent nodes is fairly low in large graphs, and the cost of finding equivalent nodes is far greater than the few extra expansions required.

9.2 Finding Node Equivalence

Algorithm 18 can be used to find unique nodes. The idea behind the algorithm is to create a table with a row for each node and a column for each iteration. During first iteration the entry for node v is $|nb(v, G)|$. In the following iterations the entry for node v is a 2-tuple of the previous entry for v and the set of previous entries for the neighbours of v. After atmost $|\mathcal{V}(G)|$ iterations, any two nodes with the same entry in the last iteration are equivalent.

The algorithm described starts by saying that any two nodes with different number of neighbours cannot be equivalent. Then the algorithm starts to propogate the number of neighbours throughout the graph. Eventually, entry $|\mathcal{V}(G)|$ for node v will hold a unique representation of the graph seen from v's point of view. That is, that the entire graph can be reconstructed from entry $|\mathcal{V}(G)|$ for node v, but it will be unlabeled.

The approach described above requires a lot of memory as it will eventually hold $|\mathcal{V}(G)|^2$ more or less complete representations of the graph. However, since we're only interested in whether or not two entries differ, we can assign a unique number N to each entry, and keep a value table V_{tb}, where we associate values with unique numbers. The value table can be reset after each iteration, and we only need access to the previous iteration, so we need only maintain two columns of the table.

This means that algorithm 18 can be implemented quite efficiently, and with more than $O(|\mathcal{V}(G)|)$ memory in $O(|\mathcal{V}(G)|^3)$ time. A pratical implementation of the algorithm can also be done without dynamic allocations. And as illustrated in algorithm 18 we can also terminate the algorithm as soon as we can conclude that all nodes are unique.

Algorithm 19 shows how algorithm 18 can be used to limit the expansion of algorithm 11. This is done by only generating successor steps for the set of unique nodes. That is, if the unlabeled remaining graph after elimination of a is the same as the unlabeled remaining graph after elimination of b, then only one successor step is generated for a and b.

9.3 Conclusion on Exploitation of Graph Symmetry

Equivalence between nodes, as defined in defintion 9, does not occur often. If we consider graphs with only a few nodes, it can be somewhat hard to construct graphs without node equivalence. However, when the number of nodes is larger and the graphs are dense, node equivalence can be rare.

Pratical implementation and evaluation of algorithm 19, shows that node equivalence does not occur often. And even for sparse graphs the overhead for identifying equivalence is higher than the benefit of reduction in the number of expansions.

In chapter 10, where comparisons are presented, algorithm 19 has a speedup constant of 0.74, which means that it is about 26% slower (on sparse graphs). If algorithm 18 is combined with depth first search, algorithm 12, then it is only about 3% slower (on sparse graphs). Nevertheless, we can conclude that there is not much node equivalence to find, and finding it does not provide a significant reduction in the number of expansions.

It might be possible that some heuristic can be used to reduce the number of times FINDUNIQUENODES(...) is called, or that another algorithm can be used to find equivalence, perhaps it is possible to create an algorithm that can update the equivalence information from the previous step. However, it is unlikely to payoff, as the number of equivalent nodes is fairly low.

Algorithm 18 Find unique nodes

```
1: function FINDUNIQUENODES(G)
2:     unique = 0
3:     for all v ∈ V(G) do                                   ▷ First iteration
4:         tb_cur(v) = |nb(v,G)|
5:     end for
6:     repeat                                                ▷ All the following iterations
7:         tb_prev = tb_cur                                  ▷ Swap previous and current tables
8:         V_tb = ∅
9:         unique_prev = unique
10:        unique = 1
11:        for all v ∈ V(G) do
12:            V = ∅
13:            for all n ∈ nb(v,G) do
14:                V = V ∪ {tb_prev(n)}
15:            end for
16:            V = (tb_prev(v), V)
17:            N = 0
18:            for all (n,v) ∈ V_tb do
19:                if v = V then                             ▷ Check if V is in V_tb
20:                    N = n
21:                end if
22:            end for
23:            if N = 0 then                                 ▷ Insert V in V_tb
24:                N = unique                                ▷ Find a unique number
25:                unique = unique + 1
26:                V_tb = V_tb ∪ {(N,V)}
27:            end if
28:            tb_cur(v) = N
29:        end for
30:        if unique = |V(G)| then
31:            return V(G)                                   ▷ Conclude that all nodes are unique
32:        end if
33:        Δunique = unique − 1 − unique_prev
34:    until Δunique = 0
35:    unique_nodes = V(G)
36:    for v ∈ unique_nodes do
37:        for n ∈ unique_nodes do
38:            if tb_cur(v) = tb_cur(n) then
39:                unique_nodes = unique_nodes \ {n}         ▷ Conclude that n and v are equivalent
40:            end if
41:        end for
42:    end for
43:    return unique_nodes
44: end function
```

Algorithm 19 Best First Search with Unique Node

```
1: function BESTFIRSTSEARCH-UNIQUE(G)
2:     s =CREATESTEP()
3:     Insert line 3-10 from algorithm 11
4:     ENQUEUE(Q,s)
5:     while Q ≠ ∅ do
6:         n =DEQUEUE(Q)
7:         Insert line 14-16 from algorithm 11
8:         for all v ∈FINDUNIQUENODES(n.G[n.R]) do
9:             Insert line 18- 32 from algorithm 11
10:            ENQUEUE(Q,m)
11:        end for
12:    end while
13: end function
```

CHAPTER 10

Comparison of Triangulation Methods

This chapter will compare the results of some of the algorithms described in chapters 3 through 8.

Minimal triangulation algorithms will be compared based on number of edges added and the total table size of the triangulated graph. Since these algorithms are relatively fast and produce minimal triangulations, yet do not guarantee optimal table size, they are reviewed on how far they deviate from the optimal total table size. Moreover, these algorithms will be compared with respect to the average number of fill-in edges added.

The comparison of greedy heuristic algorithms will be based on the average total table size of the triangulated graphs, both among the different greedy algorithms and how far from the optimal solution their results are on average.

Both greedy heuristic and minimal triangulation algorithms are relatively fast, therefore it is not interesting to compare them based on their running time.

Optimal algorithms guarantee the optimal solution and therefore always produce the same total table size, regardless of any enhancements to improve runtime efficiency. Due to this, the various strategies and pruning implementations are compared based on their respective running times.

All results were produced on the same 1.66GHz dual core laptop with 1.5GB RAM running x86 Ubuntu 10.04.

All algorithms were run on graphs of three different sizes; graphs consisting of 20, 30 and 40 nodes. There are 28 different graphs of 20 nodes, 73 different graphs of 30 nodes and 21 different graphs of 40 nodes.

The test graphs range in density from $11 - 39\%$.

10.1 Minimal Methods

This section relates to the methods described in chapter 3.

LBTriang and *MCS-M* have been used on the same graphs which have been divided into 3 categories; graphs with 20, 30 and 40 nodes. Table 10.1 contains the average results for running the LBTriang and MCS-M algorithms on the same graphs from each category. In the data provided, running time has been omitted as these algorithms have negligible runtime on the test graphs. The focus of these algorithms is finding the triangulation which does not have a smaller subset of fill-ins. For comparison with other algorithms, average total table size of the triangulated graph has also been included.

# of nodes	Average edges		Average tts	
	LBTriang	MCS-M	LBTriang	MCS-M
20	90	98	2418	8546
30	226	252	1014959	7716582
40	381	445	155034778	3124739955

Table 10.1: Average edges after triangulation and tts for the minimal algorithms

While LBTriang does not rely on the ordering of the nodes (it does require an ordering; but produces the same amount of fill-ins for different orderings on the same graph), the MCS-M algorithm relies heavily on the ordering of nodes. This may explain why LBTriang pulls ahead and produces, on average, about 11% less fill-ins compared to MCS-M.

10.2 Greedy Heuristics

This section covers the results from the greedy heuristic methods described in chapter 4. When considering greedy heuristics, time consumption is not particularly interesting, since they are fast in terms of time complexity. Due to this, the running times of the algorithms has been omitted in the following data. In this case, the relevant metric is the average divergence from optimal total table size. Recall that *minfill* searches for a triangulation with the least amount of fill-ins, *minwidth* searches for a triangulation with the lowest treewidth, *minwidth 3LA* is the same, but with a look-ahead of $k = 3$.

# of nodes	Average tts			
	minfill	minwidth	minwidth 3LA	optimal (bfs/dfs)
20	3252	4094	2596	2076
30	5044	5097	11870	5028

Table 10.2: Average tts for the three greedy heuristic algorithms

Table 10.2 shows the average total table size for three greedy heuristic algorithms. The same graphs of 20 and 30 nodes were used for the algorithms. Total table size of the optimal solution is also provided for comparison. Figure 10.1 shows the results for the algorithms on graphs with 20 and 30 nodes.

On graphs with 20 nodes, it appears that applying lookaheads yields triangulations of smaller *tts*, although increasing the lookahead may introduce more overhead and cause more divergence. As soon as the number of nodes in the graphs starts to grow, 3LA appears to be doing exactly this and makes bad choices, which introduce more fill-ins. This tendency is also found on graphs with 40 nodes but was not included in the data, due to the fact that some of the graphs of 40 nodes were intractable with an optimal algorithm, and therefore has no basis to be compared against.

In general min-fill seems to be the better choice as it produces the best results, since thet are closest to optimal in a short amount of time.

10.3 Optimal

This section relates to the methods described in chapter 5.

Tables 10.3-10.4 contains the average running time of the BFS and DFS algorithms with different pruning strategies and the use of coalescence prediction. The standard BFS and DFS algorithms are also included for comparison. The algorithms have been run on the same graphs of 30 nodes with different densities (sparse 39%, medium 22-32% and dense 11-16%). According to section 6.3 *MaxSize* is on average the best pivot strategy, and is therefore the pivot strategy of choice for comparisons.

As seen in the tables below, all optimizations, except *MPD*, improve the running time of both BFS and DFS. It seems the MaxSize pivot strategy improves both BFS and DFS equally, whereas *Oracle* improves DFS more. Table 10.5 shows speed-up constants for BFS- and DFS-MPD-PO-MaxSize, compared to BFS and DFS implementation with no optimizations on the different density of graphs.

	Running time in sec		
Density	high (39%)	mid (22-32%)	low (11-16%)
BFS	0.10	4.02	78.37
Unique	0.11	5.51	105.43
Oracle	0.04	1.62	41.51
MPD	0.10	4.14	82.27
Pivot-MaxSize	0.07	2.81	49.18
Pivot-MinWidth	0.06	2.72	60.67
MPD-Oracle	0.04	1.73	41.82
MPD-Pivot-MaxSize	0.07	3.03	55.55
MPD-Pivot-MinWidth	0.06	2.86	62.34
Oracle-Pivot-MaxSize	0.07	0.94	28.77
Oracle-Pivot-MinWidth	0.06	0.99	29.15
MPD-Oracle-Pivot-MaxSize	0.02	1.03	30.34
MPD-Oracle-Pivot-MinWidth	0.02	1.04	28.85

Table 10.3: Average running time for BFS with various implementations of pruning strategies and coalescence prediction, including combinations of these.

	Running time in sec		
Density	high (39%)	mid (22-32%)	low (11-16%)
DFS	0.10	4.34	97.73
Unique	0.10	5.04	99.29
Oracle	0.04	1.72	37.59
MPD	0.10	4.54	100.05
Pivot-MaxSize	0.06	2.86	49.29
Pivot-MinWidth	0.06	2.79	73.92
MPD-Oracle	0.04	1.86	39.89
MPD-Pivot-MaxSize	0.07	3.02	51.45
MPD-Pivot-MinWidth	0.06	2.87	73.97
Oracle-Pivot-MaxSize	0.02	0.92	17.92
Oracle-Pivot-MinWidth	0.02	0.98	26.92
MPD-Oracle-Pivot-MaxSize	0.02	1.03	18.91
MPD-Oracle-Pivot-MinWidth	0.02	1.05	28.00

Table 10.4: Average running time for DFS with various implementations of pruning strategies and coalesce prediction, and combinations of these.

	high (39%)	mid (22-32%)	low (11-16%)
BFS	0.50	3.90	2.58
DFS	0.50	4.21	5.17

Table 10.5: Speed-up constants for BFS- and DFS-MPD-PO-MaxSize, compared to BFS and DFS implementation with no optimisations on the different graph densities.

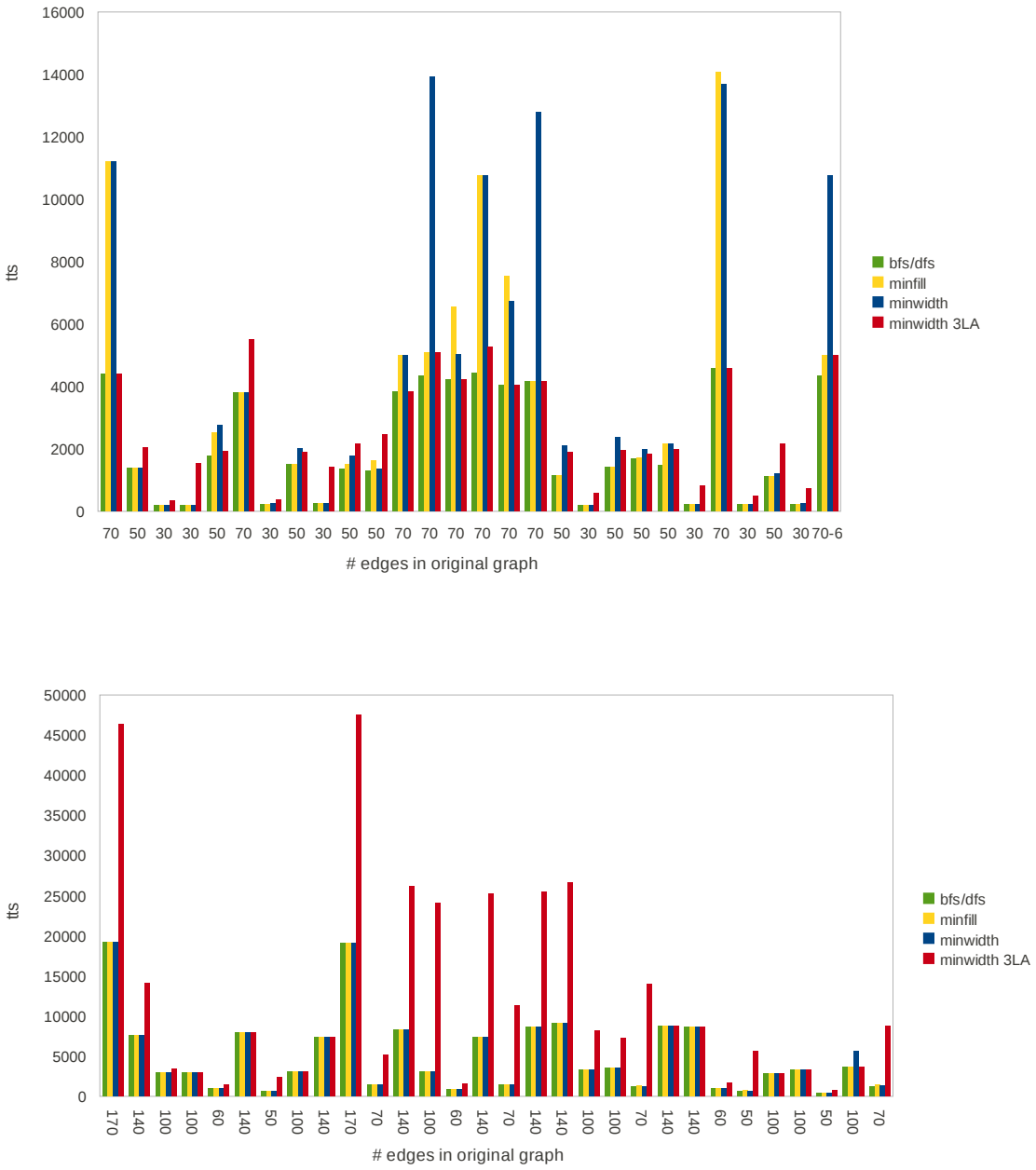


Figure 10.1: Graphs showing the greedy heuristic algorithms and their tts on graphs with 20 nodes (top) and 30 nodes (bottom), including the optimal solution (bfs/dfs) for comparison.

Summary

The minimal methods produce results far from the optimal solution with regard to total table size. They are however, guaranteed to produce a minimal set of fill-ins, and can be used to find the maximal prime decomposition of a graph. In conclusion, minimal methods should not be used for triangulation if total table size should be small.

Greedy heuristics, especially min-fill, are relatively close to getting the optimal results in far less running time. In fact, if one should use a simple triangulation method, min-fill should be considered as the first obvious choice. For this approach to work the triangulation must be allowed to be non-optimal in some of cases, since min-fill can produce non-optimal results on some graphs. A way remedy for this could, of course, be to idenify all the graphs for which min-fill is non-optimal, and triangulate these graph with an optimal search method. This, however, introduces the problem identifying graphs, for which min-fill produces non-optimal results, without first running an optimal method. Look-aheads can in some cases be used to solve the problem of min-fill not being optimal on some graphs.

However, results presented earlier in this chapter, show the tendency that too many look-aheads (perhaps compared to the size of the graph investigated) can lead to worse results than with no look-aheads. So, one should not just choose an arbitrary number of look-aheads, as this can cause further deviation from the optimal table size. Apply too few look-aheads may simply not give any improvements whatsoever. It therefore seems that there is a balance between too few and too many look-aheads. The best number of look-aheads might be effected by the number of nodes, graph density, number of edges, cliques, etc of a given graph. Nevertheless, since min-fill is cheap, one could perhaps run min-fill several times on the same graph with a dynamically adjusted number of look-aheads and then select the best solution out of these.

The optimal methods show improvements in efficiency with each optimization added to the algorithm. The biggest runtime gain appears to be produced with the inclusion of *oracle*. The best speed-up achieved is about five times as fast as the original algorithm. MPD, on the other hand, does not improve in all inclusions, this is because MPD does not provide a speed-up for all graphs, only select few.

In conclusion one would, if possible, always use the method which produces the optimal solution in the shortest amount of time and with the least space requirement. Since we do not have an algorithm which has all benefits and none of the downsides, a trade-off will have to be accepted. For faster methods which allow a varying degree of error compared to the optimal solution in total table size, one should use greedy heuristics; they will on average produce an acceptable result. If however, this trade-off cannot be allowed, one should seek to use optimal solutions with as many pruning and coalescing optimisations aspossible to speed-up the process. Also, some thought should be put into choosing the correct pivot strategy, with respect to the classification of the graphs to be triangulated. A discussion of which strategy to choose was presented earlier in section 6.3.

CHAPTER 11

Discussion

In this report we have proposed two new methods to improve the efficiency of optimal/exact search algorithms. First, we have implemented a method which reduces the number of states explored in the search space of elimination orders, namely pivot cliques. Secondly, a technique to predict coalescence of elimination orders can be applied in conjunction with the previously mentioned improvement.

We have also presented, Maximal prime subgraph decomposition (MPD), and showed how it may improve the efficiency of optimal search for the case of sparse graphs, when used in conjunction with the two other methods. Note that using pivot cliques is only possible when decomposition in the graph is not exploited, since the clique separators must be preserved. Still, using pivot cliques enables MPD to have a fall-back strategy when the graph is not decomposable.

One trend that is immediately apparent is how well min-fill in general yields relatively good triangulations, compared to the other heuristic methods we have experimented with. Also, in a large number of graphs, min-fill found optimal solutions. Despite this fact we uncovered quite a few graphs where min-fill did not find optimal triangulations. In general min-fill was worst on graphs of twenty nodes by a deviation from the optimal solution by about 30 percent. Whereas, for all test graphs consisting of thirty nodes min-fill on average only deviated slightly (about 0.5 %) from the optimal solution. Also, min-fill with lookaheads improved the average result for graphs consisting of twenty nodes, which indicates that it may be benefitial to experiment more with min-fill-LA(k).

It can be discussed whether the graphs are representative of the types of graphs encountered in a practical real-world setting. Tests have solely been conducted on bipartite graphs, which are generally some of the hardest graphs to triangulate. This difficulty arises from the large number of combinations of fill-ins in sparse graphs (Ottosen and Vomlel, 2010b). We were able to confirm this observation from the results of our experiments on graphs of varying densities. From the results in chapter 10 there is a clear tendency that sparse graphs take the longest time to triangulate.

Despite our efforts to find methods for improving the efficiency of BFS and DFS we still encountered sparse test graphs that were intractable, even though the graphs only consisted of forty nodes. Again, we believe the reason for this difficulty is due to the sheer number of possible combinations sparse graphs give rise to, which eventually exhausts all available memory.

In any case the test results do show that it is possible to achieve considerable improvements in efficiency with minor changes to the optimal methods. E.g. the best DFS method exhibited an average speed-up of a factor five for graphs consisting of thirty nodes. This is clearly an improvement over the unmodified version of DFS. More powerful hardware could provide further insight into the speed-up factor for larger inputs.

Regardless of the optimality criterion, be it minimum total table size, minimum treewidth or minimum number of fill-ins, optimal triangulation is still NP-hard. Due to this it is indeed possible that no general algorithm for finding optimal triangulations running in polynomial time exists. However, we have seen that the optimal search methods can exhibit good average case speed for denser graphs.

Among the poorest performing methods for finding triangulations of small table size were surely the minimal methods, MCS-M and LB-Triang. It is clear that minimality of a triangulation does not equate to small table size. These methods can, however, be applied for other tasks. For instance, Olesen and Madsen (2002) discusses the usage of these methods for admitting a MPD.

Work on Heuristics for A*

A few other attempts at improving optimal search have been made. We have been working on a heuristic function we call three-way merge. This function essentially estimates the change in table size when eliminating a node, and this estimate is then used to reorder the priority queue. The change in table size is then associated with the set of nodes that have been eliminated, and stored. And whenever, a new step is generated, we lookup the table size change associated with the largest subset of remaining nodes, not adjacent to the set of eliminated nodes. That is, we merge the table size change of some other step, with the table size of the step just generated, in order to get a lower bound on the final total table size.

We have been working on a complicated data structure for storing and making these kinds of queries. And we have tested different ways in which the table size change from two different steps could be merged. We believe that it is possible to get a lower bound on the final total table size for a partial triangulation, using another partial triangulation of some of the remaining nodes. We did not succeed in exploiting this to provide any significant speedup. And due to a lack of ressources this work was never formalized, though we are fairly confident that it is possible to prove correctness.

11.1 Future Work

One possible avenue to pursue could be min-fill-LA(k) with dynamic look-aheads, as mentioned during the discussion of our test results. By running several passes of min-fill-LA(k) with increasing look-aheads and finally choosing the best solution, it might be possible to improve the accuracy of this heuristic. This is similar to the idea of re-triangulation discussed in (Flores and Gamez, 2007), where several different methods may be applied to the same graph, where the best solution among these is used.

Another open question is how to determine how difficult a certain graph is to triangulate. More specifically, whether it is feasible to run a heuristic method on the graph and find a solution of acceptable table size. As mentioned previously, sparse graphs are among the most challenging, so the density could, perhaps, be used, among other properties, as a measure for difficulty. Still, our test data shows that min-fill is able to produce optimal results for some sparse graphs. Obviously, one way to know for sure whether min-fill is non-optimal on some graph is to run an exact method on it and compare the results. Yet, as mentioned before, it is not always feasible to search for the optimal solution, even for relatively small graphs.

Our work in chapter 7 suggests that research into searching in the space of triangulations might be worth while. As the space of all elimination orders is much larger than the space of all triangulations, as there are many elimination orders that yields the same triangulation. Furthermore, it is possible that such a search space could be reduced to the space of minimal triangulations or, perhaps, even smaller. Yet, compared to BFS and DFS presented in Ottosen and Vomlel (2010b) and improved here, research in searching the space of all triangulations would likely require a whole new approach.

CHAPTER 12

Conclusion

We have shown that it is possible to improve the efficiency of optimal triangulation methods substantially. This has been achieved by reducing the search space of elimination orders by the means of pivot cliques and in advance pruning of redundant subproblems with coalescence prediction by transposition in partial elimination orders. However, the optimal methods are still, in general, too computationally demanding for use in larger problem sizes. Even in cases where the problem size is very small some sparse graphs remain intractable, where dense graphs can be computed in a matter seconds.

Triangulation with respect to other optimality criteria, namely miniumum treewidth and minimum fill-in, can also benefit from the improvements proposed in this report, since these optimizations exploit general properties of elimination orders and triangulated graphs.

We have compared heuristic methods and deduced that the min-fill heuristic generally gives the best results, which in many cases only slightly deviate from the optimal solution, making min-fill the most reliable greedy heuristic we have experimented with. However, we can conclude that there are also cases where min-fill gives results that are far from optimal. Consequently, the choice of a greedy heuristic over an optimal method can potentially render inference in the triangulated graph intractable.

APPENDIX A

Best First Search with Decomposition, Coalescence Prediction and Pivot

The pseudo code for a combination of all the improvements to algorithm 11 can be found in algorithm 20.

Algorithm 20 Best First Search with MPD, Oracle and Pivot

```
1:  function BESTFIRSTSEARCH-MPD-PIVOT-ORACLE(G)
2:      s =CREATESTEP()
3:      Insert line 3-10 from algorithm 11
4:      s.S =DECOMPOSE(s.G[s.R], s.C, s.R)              ▷ Attempt initial decomposition
5:      s.X = s.R                                       ▷ Set initial expansion set
6:      ENQUEUE(Q, s)
7:      while Q ≠ ∅ do
8:          n =DEQUEUE(Q)
9:          Insert line 14-16 from algorithm 11
10:         E_x = n.R \ n.X
11:         if n.S = n.R then
12:             X = n.X \ SELECTPIVOT(G_n, R_n, C_n)    ▷ Reduce expansion set X with pivot
13:         else
14:             X = n.X ∩ n.S                           ▷ Reduce expansion set X with decompostion
15:         end if
16:         for all v ∈ X do
17:             E_x = E_x ∪ {v}
18:             m =CREATESTEP()
19:             Insert line 18- 32 from algorithm 11
20:             m.X = m.R \ (E_x \ fa(v, m.G[m.R]))     ▷ Reduce expansion set by coalesce prediction
21:             if n.S ∩ m.R = ∅ then                   ▷ Find decompositions
22:                 m.S =DECOMPOSE(m.G[m.R], m.C, m.R)
23:             else
24:                 F = E(m.G) \ E(n.G)                 ▷ The set of fill-ins added.
25:                 I = {v, u | {u, v} ∈ F}             ▷ The set of nodes with new fill-ins.
26:                 NC = C ∈ m.C | C ∩ I ≠ ∅            ▷ Find new cliques
27:                 m.S =DECOMPOSE(m.G[m.R], NC, n.S ∩ m.R)
28:             end if
29:             ENQUEUE(Q, m)
30:         end for
31:     end while
32: end function
```

APPENDIX B

Comparison of Pivot Selection Strategies

	Running time in sec				
strategy	time	Pruned by upperbound	Coalescings	Node expansions	Solutions found
First	0.42	8763	4770	1270	9704
MaxFill	0.63	6637	5438	1321	9342
MinFill	0.57	9156	4393	1259	9917
MaxSizeMinFill	0.51	6539	4794	1311	9847
MaxSize	0.46	6578	5057	1314	9633
Middle	0.44	7807	5086	1313	9944
MaxSizeMaxWidth	0.45	6594	5223	1321	9579
Last	0.48	7715	4616	1283	10152
MaxWidth	0.44	7638	5245	1321	9856
MinWidthMaxSize	0.37	7965	4149	1189	9035
MaxSizeMinWidth	0.45	6563	4938	1311	9699
MinWidth	0.38	8238	4170	1189	9290
MinFillSingle	0.64	7715	4616	1283	10152
MaxSizeMaxFill	0.49	6637	5438	1321	9342

Table B.1: Table of averages results of DFS with Pivot strategies on graphs of size 30 with high density and only Pivot

	Running time in sec			
strategy	time	Pruned by upperbound	Coalescings	Node expansions
MaxSizeMaxFill	0.49	2239	19111	1331
MinFill	0.61	7770	18211	1269
First	0.41	4822	18526	1280
ROT13	0.48	4288	19103	1323
MinWidthMaxSize	0.36	4178	17132	1199
MaxFill	0.61	2239	19111	1331
MaxSizeMinFill	0.51	2210	18941	1321
MaxSizeMinWidth	0.48	2208	18941	1321
MinFillSingle	0.63	5141	18658	1291
Last	0.48	5141	18658	1291
MinWidth	0.4	4516	17322	1199
Middle	0.45	4237	19102	1323
MaxWidth	0.45	3345	19326	1331
MaxSizeMaxWidth	0.47	2239	19111	1331
MaxSize	0.45	2226	18999	1324

Table B.2: Table of averages results of BFS with Pivot strategies on graphs of size 30 with high density and only Pivot

	Running time in sec				
strategy	time	Pruned by upperbound	Coalescings	Node expansions	Solutions found
MaxWidth	0.15	3808	1011	1321	2857
MinFillFamily	0.25	4889	897	1269	2411
MaxSizeMinFillFamily	0.21	4160	1031	1317	2589
MaxFill	0.3	2695	1013	1321	2471
Last	0.14	3263	811	1131	2319
MinSizeMaxFillFamily	0.14	2665	1016	1321	2651
DynamicWidthSize	0.22	3639	1072	1314	2916
ROT13	0.13	3804	939	1281	2464
MaxSizeMaxFill	0.15	2687	1013	1321	2471
Middle	0.16	3731	969	1292	2605
MaxSize	0.12	2720	1052	1314	2504
MinFill	0.3	4412	810	1161	2217
MaxSizeMinFill	0.16	2784	871	1298	2289
MaxSizeMinWidth	0.11	2769	935	1311	2368
MinWidthPrime	0.15	4336	807	1125	2309
MinWidthMaxSize	0.11	3610	755	1113	2098
MaxFillFamily	0.22	2797	1024	1321	2569
DynamicWidthSize4	0.28	2735	1017	1314	2347
MaxSizeMaxWidth	0.12	2677	1012	1321	2471
DynamicWidthSize3	0.22	2735	1017	1314	2347
DynamicWidthSize2	0.20	2735	1017	1314	2347
First	0.15	4794	981	1260	2541
MinWidth	0.12	4336	807	1125	2309

Table B.3: Table of averages results of DFS with Pivot strategies and Oracle on graphs of size 30 with low density and with Oracle

	Running time in sec			
strategy	time	Pruned by upperbound	Coalescings	Node expansions
MaxSizeMaxFill	0.53	2239	23655	1331
DynamicWidthSize4	0.55	2226	23529	1324
MinFill	0.57	7770	22539	1269
MinFillFamily	0.59	7770	22539	1269
MaxWidth	0.46	3345	23655	1331
MaxSize	0.43	2226	23529	1324
Last	0.48	5141	22935	1291
MinFillSingle	0.68	5141	22935	1291
Middle	0.47	4237	23511	1323
MaxSizeMaxWidth	0.48	2239	23655	1331
First	0.41	4822	22738	1280
MinSizeMaxFillFamily	0.49	2239	23655	1331
MaxSizeMinFill	0.5	2210	23475	1321
DynamicWidthSize	0.64	7238	24503	1324
MaxFillFamily	0.64	2239	23655	1331
MaxFill	0.64	2239	23655	1331
DynamicWidthSize2	0.58	2226	23529	1324
MaxSizeMinWidth	0.45	2208	23475	1321
MaxSizeMinFillFamily	0.56	8657	24986	1331
MinWidthPrime	0.41	4516	21279	1199
ROT13	0.46	4288	23511	1323
MinWidth	0.36	4516	21279	1199
DynamicWidthSize3	0.58	2226	23529	1324

Table B.4: Table of averages results of BFS with Pivot strategies and Oracle on graphs of size 30 with high density and with Oracle

	Running time in sec				
strategy	time	Pruned by upperbound	Coalescings	Node expansions	Solutions found
First	94.57	2957248	2813448	338399	68831
MaxFill	101.97	1888455	3140642	350469	66397
MinFill	92.94	2398420	2288882	267629	71632
MaxSizeMinFill	80.95	1683323	2835783	325154	64178
MaxSize	77.91	1693957	2943829	333001	66835
Middle	93.27	2613762	3069941	357595	68616
MaxSizeMaxWidth	79.21	1747602	2980308	339450	66647
MinWidthMaxSize	75.93	2279821	2233112	262416	67935
MaxSizeMinWidth	77.18	1666085	2905895	328277	67063
MinWidth	76.34	2298139	2236562	262222	69329
MinFillSingle	110.76	2601253	3029122	348499	69228
MaxSizeMaxFill	84.2	1696910	3040431	338929	66239
MaxWidth	93.54	2481254	3089313	358543	68992
Last	92.4	2601253	3029122	348499	69228

Table B.5: Table of averages results of DFS with Pivot strategies on graphs of size 30 with low density and only Pivot

	Running time in sec			
strategy	time	Pruned by upperbound	Coalescings	Node expansions
ROT13	88.72	*2450327*	3050831	323020
MinWidthMaxSize	73.94	2328952	2341986	248767
MaxFill	95.89	1527726	3152026	319786
MaxSizeMinFill	80.24	1533012	2953280	313484
MaxSizeMinWidth	76.51	1483241	3021471	314999
MinFillSingle	102.69	2254358	3023492	314973
Last	85.65	2254358	3023492	314973
MinWidth	74.29	2368462	2347384	248752
MaxWidth	86.27	2161485	3077945	322404
MaxSizeMaxWidth	76.84	1496465	3050171	318068
MaxSize	76.06	1480475	3040127	316074
Middle	87.52	2373255	3049457	322990

Table B.6: Table of averages results of the BFS with Pivot strategies on graphs of size 30 with middle density and only Pivot

	Running time in sec				
strategy	time	Pruned by upperbound	Coalescings	Node expansions	Solutions found
MinFillFamily	43.47	1535826	298641	291536	14804
MaxSizeMinFillFamily	32.54	1415472	293823	295697	13844
MaxFill	43.84	958145	369953	349221	13732
Last	28.97	1230196	319646	311527	13802
MinSizeMaxFillFamily	27.26	927099	344429	335213	13863
DynamicWidthSize	36.42	928696	334500	328066	15839
ROT13	34.77	1498604	354793	344650	14913
MaxSizeMaxFill	28.76	889857	349552	335253	13901
Middle	33.97	1445284	367713	348704	15452
MaxSize	25.19	937181	342310	331212	14549
MinFill	37.79	1155463	233561	221309	13192
MaxSizeMinFill	28.06	916003	301391	312296	11399
MaxSizeMinWidth	24.25	898586	323622	321431	13748
MinWidthPrime	26.9	1150589	235413	222949	13814
MinWidthMaxSize	25.04	1075895	225830	212019	12858
MaxFillFamily	38.71	1006362	356547	347995	14138
DynamicWidthSize4	35.49	906967	332874	328054	13920
MaxSizeMaxWidth	25.0	916717	338395	331770	14407
DynamicWidthSize3	35.43	906967	332874	328054	13920
DynamicWidthSize2	35.44	907662	332919	328117	14048
First	35.04	1563824	329001	305828	15182
MinWidth	26.85	1150273	235411	222913	13814
MaxWidth	36.48	1471726	385134	354372	16339

Table B.7: Table of averages results of DFS with Pivot strategies and Oracle on graphs of size 30 with low density and with Oracle

	Running time in sec			
strategy	time	Pruned by upperbound	Coalescings	Node expansions
MaxSizeMaxFill	82.17	1434788	5258381	316261
DynamicWidthSize4	89.04	1474217	5167256	313583
MinFill	92.16	2528025	4264121	256110
MinFillFamily	92.33	2528025	4264121	256110
MaxWidth	88.09	2161500	5369108	322408
MaxSize	77.89	1480491	5251751	316078
Last	87.57	2254371	5303421	314976
MinFillSingle	104.59	2254371	5303421	314976
Middle	89.64	2373273	5357469	322994
MaxSizeMaxWidth	78.89	1474858	5213151	315982
First	90.85	2849198	5061711	307197
MinSizeMaxFillFamily	81.43	1465140	5320756	317668
MaxSizeMinFill	82.15	1498407	5097970	311281
DynamicWidthSize	83.08	1548101	5159230	313611
MaxFillFamily	97.68	1527741	5414066	319790
MaxFill	97.58	1527741	5414066	319790
DynamicWidthSize2	83.97	1477153	5167840	313644
MaxSizeMinWidth	78.14	1475912	5139159	312261
MaxSizeMinFillFamily	85.25	2701557	4676234	292346
MinWidthPrime	75.42	2369538	4157237	248803
ROT13	90.61	2450344	5373440	323024
MinWidth	75.56	2368470	4156983	248754
DynamicWidthSize3	84.14	1474217	5167256	313583

Table B.8: Table of averages results of BFS with Pivot strategies and Oracle on graphs of size 30 with middle density and with Oracle

	Running time in sec				
strategy	time	Pruned by upperbound	Coalescings	Node expansions	Solutions found
First	5317.96	71585305	52048917	8443839	3535
MaxFill	1026.09	38719360	58919426	9013814	1177
MinFill	1311.46	74381155	55039812	8401765	2607
MaxSizeMinFill	768.4	34178861	46055260	7747145	1061
MaxSize	743.38	34468282	48784492	7998432	1119
Middle	1180.78	66873153	64913829	10143968	2672
MaxSizeMaxWidth	781.7	35200519	51577953	8332463	1140
MinWidthMaxSize	1107.4	68944771	50852162	7841613	2358
MaxSizeMinWidth	736.8	34349340	47164548	7870728	1140
MinWidth	1132.76	70577778	52828735	8132579	2669
MinFillSingle	1336.0	66378895	63147347	9850241	2691
MaxSizeMaxFill	821.25	34532424	52025785	8286244	1153
MaxWidth	5041.87	50459925	61864165	9664030	2939
Last	1165.78	66381948	63148537	9850385	2693

Table B.9: Table of averages results of DFS with Pivot strategies on graphs of size 30 with low density and only Pivot

	Running time in sec			
strategy	time	Pruned by upperbound	Coalescings	Node expansions
MaxFill	861.44	17949422	33506690	4481155
MaxSizeMinFill	785.95	18769934	29705853	4381607
MaxSizeMinWidth	779.95	18636014	30235576	4421025
MinFillSingle	1102.75	31019423	34883926	4762073
MaxSizeMaxFill	776.05	17405192	31444146	4403790
MinFill	995.24	35983582	29727960	4034372
First	898.75	35849080	28874633	4181233
ROT13	1091.42	36767835	34301991	4821484
Last	4939.43	31019688	34884237	4762074
MinWidth	881.27	34762218	28888966	3959451
MaxWidth	913.04	23170166	34060084	4657757
MaxSizeMaxWidth	4767.58	17807001	31213382	4423257
MaxSize	743.72	18277437	30678698	4415874
Middle	5020.19	31364159	34913545	4824088

Table B.10: Table of averages results of BFS with Pivot strategies on graphs of size 30 with low density and only Pivot

	Running time in sec				
strategy	time	Pruned by upperbound	Coalescings	Node expansions	Solutions found
MaxWidth	394.52	24709305	8074570	9129468	1796
MinFillFamily	8924.39	38520926	9289215	9457339	1762
MaxSizeMinFillFamily	4678.44	30407646	7224730	7755949	1697
MaxFill	443.66	16726826	7555264	8747818	274
Last	398.78	28009470	7571393	8519474	1122
MinSizeMaxFillFamily	296.74	16366255	6432613	8112495	282
DynamicWidthSize	272.4	15644756	5948497	7608502	264
ROT13	494.37	35786528	8658140	9160352	1517
MaxSizeMaxFill	312.75	15641255	6529133	8046354	257
Middle	448.07	31300573	8694720	9522782	1436
MaxSize	272.48	16465445	6202197	7821182	298
MinFill	561.83	32250205	7252935	7863668	1270
MaxSizeMinFill	299.5	15957379	5636776	7456938	264
MaxSizeMinWidth	265.96	15889689	5800496	7538846	274
MinWidthPrime	438.34	31831357	7318109	7787925	1328
MinWidthMaxSize	397.92	28750556	6443310	6899062	1033
MaxFillFamily	4690.84	19522223	7357486	8768935	912
DynamicWidthSize4	273.11	15644756	5948497	7608502	283
MaxSizeMaxWidth	272.51	15866785	6293000	7963155	307
DynamicWidthSize3	274.06	15646979	5948631	7608732	690
DynamicWidthSize2	307.61	18397678	6258390	7718068	1847
First	403.47	29963338	6794708	7398879	1643
MinWidth	410.65	29614279	6810762	7198398	1209

Table B.11: Table of averages results of DFS with Pivot strategies and Oracle on graphs of size 30 with low density and with Oracle

	Running time in sec			
strategy	time	Pruned by upperbound	Coalescings	Node expansions
MaxSizeMaxFill	797.48	17598069	77035091	4317980
DynamicWidthSize4	789.75	18401762	76958686	4352422
MinFill	1061.42	35983598	72120689	4034392
MinFillFamily	1063.41	35983598	72120689	4034392
MaxWidth	957.23	23170174	86441546	4657779
MaxSize	794.24	18277455	78814503	4415896
Last	4959.54	31019703	89911013	4762096
MinFillSingle	1161.49	31019703	89911013	4762096
Middle	9164.69	31364177	89107466	4824110
MaxSizeMaxWidth	780.65	17916215	76991759	4330038
First	943.0	35849098	76006461	4181255
MinSizeMaxFillFamily	821.12	17762273	80597637	4448849
MaxSizeMinFill	801.37	18708215	75350459	4293245
DynamicWidthSize	4950.06	18401749	76958654	4352422
MaxFillFamily	937.3	17949430	83716559	4481177
MaxFill	933.66	17949430	83716559	4481177
DynamicWidthSize2	810.48	19942127	77351564	4384465
MaxSizeMinWidth	784.5	18628869	76102389	4324477
MaxSizeMinFillFamily	888.96	34284858	70497662	3976047
MinWidthPrime	999.23	36735835	74517939	4147813
ROT13	1131.1	36767852	88789196	4821506
MinWidth	916.45	34762234	71517770	3959470
DynamicWidthSize3	790.0	18405468	76960783	4352524

Table B.12: Table of averages results of BFS with Pivot strategies and Oracle on graphs of size 30 with low density and with Oracle

Running time in sec	
strategy	time
MaxSize	27.12
MinFill	26.80
MinFillSingle	39.25
MaxSizeMaxFill	28.77
MaxSizeMinFill	28.94
MaxSizeMinWidth	27.02
MaxSizeMaxWidth	27.07
MaxFill	41.62
MaxWidth	27.47
MinWidth	11.26
MinWidthPrime	11.25
MinWidthMaxSize	7.09
Last	26.20
First	9.95
Middle	27.25
DynamicWidthSize	30.20
DynamicWidthSize2	27.11
DynamicWidthSize3	27.12
DynamicWidthSize4	27.13
MaxFillFamily	35.11
MinFillFamily	20.93
MinSizeMaxFillFamily	27.88
MaxSizeMinFillFamily	23.321
ROT13	27.437

Table B.13: Table of averages results of BFS with Pivot strategies and Oracle on graphs of size 60 with high density and with Oracle

APPENDIX C

Implementation

This appendix covers the implementation of the algorithms described in previous chapters.

C.1 Overview

All programming has been done in C++. To obtain maximal efficiency C++ templates are used to parameterize the input graphs and various methods with a compile-time defined static size. This size is set to the number of nodes in the input graph.

The program itself is a simple command-line based application, which outputs results to the terminal.

Graph Representation

Graphs are represented by an adjacency $n \times n$ matrix, where n is the number of nodes in the graph. For example the complete graph $G = (\{a,b,c\},\{ab,ac,bc\})$ shown in figure C.1 has the form in equation C.1.

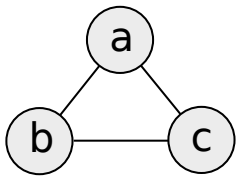


Figure C.1: The graph G

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \tag{C.1}$$

Since the graph is undirected the matrix is symmetrical across the diagonal, so if $\forall(i,j):\ (i,j) = (j,i)$ does not hold for all pairs, the graph is not consistent. The diagonal contains zeros, since we do not allow cycles in a graph.

With this representation it is easy to compute the neighbors of a given node. E.g. The first row in the matrix corresponds to the neighbor-set of node a, which is $nb(a) = [0\ 1\ 1] = \{b,c\}$, since the last two indices correspond to nodes b and c, respectively.

The storage requirements for this representation are low, since the matrix is represented by an array of n bitsets (or rather, bitfields), which each contain n bits.

Moreover, it is easy to perform many set and graph-related operations on this data structure efficiently, since it is merely a low-level abstraction of bits.

Encapsulation of Triangulation Methods

In order to have the ability to plug-in an unrestricted number of algorithm implementations into the program an abstract base class named `TriangulationStrategy` is introduced. Every implementation

inherits from this class and overloads the virtual method `run()`, which invokes the concrete algorithm and returns the output graph.

In addition, the base class contains methods for counting statistics and reporting running progress.

Representing Subgraphs and Eliminated Nodes

Remaining nodes in a graph are represented by a bitfield, which contains nodes $[0-n]$ in the same order as the graph. Initially this bitfield contains only 1s, which essentially means that no nodes have been eliminated yet. Every time a node is eliminated from the graph, the bit position corresponding to the node is set to 0. E.g. the bitfield $[1\ 1\ 0]$ indicates that node c has been eliminated from the graph shown in figure C.1.

Subgraphs are easy to represent using this approach. Often the subgraph induced from eliminating a given node is needed. For instance, when the number of fill-ins or maximal cliques must be computed with respect to a set of already eliminated nodes. So, eliminated nodes are omitted by applying the bitfield of remaining nodes as an AND-mask on the neighbor-set of a node.

E.g. Let $G_e = [1\ 1\ 0]$ be the bitfield where node c is eliminated and $nb(a) = [0\ 1\ 1]$ is the neighbor-set of a from earlier, then the operation $G_e \wedge nb(a) = [0\ 1\ 0]$ yields the non-eliminated neighbors of node a, namely $\{b\}$.

Bibliography

Anne Berry. A wide-range efficient algorithm for minimal triangulation. *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, 1999.

Anne Berry, Jean R. S. Blair, Pinar Heggernes, and Barry W. Peyton. Maximum cardinality search for computing minimal triangulations of graphs. *Algorithmica*, 2004.

Anne Berry, Romain Pogorelcnik, and Geneviéve Simonet. An introduction to clique minimal separator decomposition. *Algorithms*, 2010.

Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16:575–577, September 1973. ISSN 0001-0782.

F. Cazals and C. Karande. Note: A note on the problem of reporting maximal cliques. *Theor. Comput. Sci.*, 407:564–568, November 2008. ISSN 0304-3975.

L. S. Chandran, L. Ibarra, F. Ruskey, and J. Sawada. Generating and characterizing the perfect elimination orderings of a chordal graph. *Theoretical Computer Science*, 307(2):303 – 317, 2003. ISSN 0304-3975. doi: DOI:10.1016/S0304-3975(03)00221-4.

Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 1 edition, April 2009. ISBN 978-0521884389.

M. Julia Flores and José A. Gamez. Triangulation of bayesian networks by retriangulation. *International Journal of Intelligent Systems*, 18:153 – 164, 2003. doi: doi:10.1002/int.10079.

M. Julia Flores and José A. Gamez. A review on distinct methods and approaches to perform triangulation for bayesian networks. 2007.

Finn V. Jensen and Thomas D. Nielsen. *Bayesian Networks and Decision Graphs*. Springer, 2 edition, 2007. ISBN 978-0-387-68281-5.

Uffe Kjaerulff. Triangulation of graphs - algorithms giving small total state space. 1990.

Timo Koski and John M. Noble. *Bayesian Networks an Introduction*. John Wiley & Sons. Ltd, 1 edition, 2009. ISBN 978-0-470-74304-4.

Kristian G. Olesen and Anders L. Madsen. Maximal prime subgraph decomposition of bayesian networks. *IEEE TRANSACTIONS ON SYSTEMS, MAN AND CYBERNETICS*, 32, Feburary 2002.

Thorsten J. Ottosen and Jiri Vomlel. Honour thy neighbour - clique maintenance in dynamic graphs. 2010a.

Thorsten J. Ottosen and Jiri Vomlel. All roads lead to rome - new search methods for optimal triangulation. 2010b.

Robert E. Tarjan. Decomposition by clique separators. *Discrete Mathematics*, 55(2):221 – 232, 1985. doi: DOI:10.1016/0012-365X(85)90051-2.

Mihalis Yannakakis. Computing the minimum fill-in is np-complete. *Society for Industrial and Applied Mathematics*, 1981.