Universiteit Utrecht

Department of Information and Computing Sciences

Experimental Research and Algorithmic Improvements involving the Graph Parameter Boolean-width

Master Thesis
ICA-3507645

Author:
Frank J.P. van Houten
Utrecht University
Frankv@nhouten.com

Supervisor:
Prof. Dr. H.L. Bodlaender
Utrecht University
H.L.Bodlaender@uu.nl

July 2015

Abstract

In this thesis, we investigate numerous algorithms that make use of boolean decompositions. We provide a new algorithm for computing the representatives of linear decompositions. These representatives are the indices for a table storing partial solutions, which is used by dynamic programming algorithms. These algorithms are parameterized by the width of the boolean decomposition that is used as an input.

We present a new heuristic to compute linear boolean decompositions and experimentally evaluate it by comparing it to existing heuristics. The experimental evaluation shows that significant improvements can be made with respect to running time without increasing the width of the generated decompositions. Moreover, we consider reduction rules in order to reduce the running time needed to generate linear boolean decompositions. However, these reduction rules seem to describe degenerate graph classes that will not occur often in practical settings, meaning that the benefit of reducing vertices will be very marginal.

Boolean decompositions can be used to solve the class of locally checkable vertex subset problems. We evaluate an algorithm for solving these problems, showing that the algorithm is often up to several orders of magnitude faster compared to theoretical worst case bounds.

Contents

1 Introduction **1**
1.1 Boolean-width 2
1.2 Overview of the thesis 2

2 Preliminaries **3**
2.1 Graph Theory 3
2.2 Boolean-width 4
2.3 Properties and bounds 7
2.4 Running time analysis 8
2.5 Implementation details 8
2.5.1 Bitsets 8
2.5.2 Experimental setup 9

3 Introduction to boolean decompositions **10**
3.1 Neighborhood equivalence 10
3.2 Algorithms on boolean decompositions 11
3.2.1 Maximum independent set 12
3.2.2 Minimum dominating set 13

4 Linear boolean decompositions **16**
4.1 Linear boolean-width related to pathwidth 16
4.2 Algorithms parameterized by linear boolean-width 18
4.3 Computing representatives for linear decompositions 18
4.4 Exact linear decompositions 21

5 Heuristics for generating linear decompositions **23**
5.1 Generic form of the heuristics 23
5.1.1 Selecting the initial vertex 24
5.1.2 Pruning 24
5.1.3 Trivial cases 24
5.2 Score functions 25
5.2.1 Relative neighborhood 25
5.2.2 Least cut value 26
5.2.3 Incremental unions of neighborhoods 26
5.3 Experiments 28
5.4 Conclusion 31

6 Reduction rules for linear boolean-width **32**

6.1 Definitions . . . 33
6.2 Proving reduction rules . . . 34
6.3 Validity of general boolean-width reduction rules . . . 34
6.3.1 Islet rule . . . 35
6.3.2 Pendant rule . . . 35
6.3.3 Twin rule . . . 36
6.4 Validity of treewidth reduction rules . . . 36
6.5 New reduction rules . . . 38
6.5.1 Sequence rule . . . 38
6.5.2 Clique rule . . . 40
6.5.3 Other reduction rules . . . 41
6.6 Expanding linear decompositions using general reduction rules . . . 42
6.7 Conclusion . . . 43

7 Vertex subset problems 44
7.1 Definitions . . . 45
7.2 Bounds on the number of d-equivalence classes . . . 46
7.3 Solving (σ, ρ) problems . . . 47
7.4 Experiments . . . 50
7.5 Heuristics for minimizing the number of equivalence classes . . . 52
7.6 Conclusion . . . 52

8 Conclusion 54

Bibliography 56

A Linear boolean-width upper bounds on treewidthlib graphs 58

B Implemented algorithms 65

C (Paper) Practical algorithms for linear boolean-width 66

Chapter 1

Introduction

A graph is a structure used to model a set of objects, called vertices, and possible links between pairs of these objects, called edges. This representation can be used to solve many practical problems. A simple example is the problem of finding the shortest path between two vertices, which is solved continuously by GPS navigation systems. The problem of finding a shortest path can be solved relatively easily, but other problems are much harder and require more computation time. An example of such a problem is to find the maximum independent set in a graph. An independent set X is a subset of all vertices of the graph, such that no two vertices in X are connected by an edge. A trivial algorithm for finding the largest independent set in a graph with n vertices is to compute all 2^n possible subsets of vertices and check for each each subset if it is an independent set, while keeping track of the largest one seen so far. In practice this is not feasible as soon as n becomes a large number. Therefore we can make use of a divide and conquer approach, which splits the problem into multiple smaller problems. We then compute partial solutions, which we combine into a solution for the original problem.

One way to use the technique of divide and conquer is through the use of a dynamic programming algorithm on a decomposition tree of a graph. A decomposition tree of a graph is a derived structure that captures the necessary information on how to divide the graph. We can use a bottom up dynamic programming algorithm to compute partial solutions for each node of the decomposition tree. The running time of computing such a partial solution is bounded by a graph parameter that is associated to the complexity of the decomposition tree. Such a graph parameter is known as the width of the decomposition tree, and the width of a graph is the width of an optimal decomposition tree for that graph. Many NP-hard problems on graphs become easier if such a graph parameter is small. For instance, given a boolean decomposition of width k we can solve the maximum independent set problem in $O^*(2^k)$ time, compared to the $O^*(2^n)$ time needed for the trivial approach.

1.1 Boolean-width

Boolean-width is a recently introduced graph parameter [7]. The decomposition tree for boolean-width is a binary partition tree called a boolean decomposition. The decomposition tree is constructed in such a way that the number of distinct neighborhoods at each partition of the nodes of the decomposition tree is small. In this thesis we investigate generating boolean decompositions as well as using these decompositions in practical algorithms for solving problems on graphs.

Algorithms for computing boolean decompositions have been studied before in [24, 15, 18, 2]. In this thesis we mainly consider a special type of decomposition called a linear boolean decomposition. Linear boolean decompositions are easier to compute and the theoretical running times of algorithms using linear decompositions are lower than those using general ones.

Our contribution is inspired by Sharmin [18], who studied the practical aspects of boolean-width. We look into the benefits of using linear boolean decompositions over general ones, and give a new heuristic that can be used to compute linear boolean decompositions. We consider methods to speed up the computation of linear decompositions through the use of reduction rules, and we evaluate several algorithms that make use of boolean decompositions.

1.2 Overview of the thesis

In Chapter 2 we provide an overview of definitions that are used in this thesis. We also give a number of properties of boolean-width and discuss a few details on our implementation and on the experiments that we have conducted. In Chapter 3 we give an introduction to dynamic programming algorithms on boolean decompositions by explaining the algorithm of Bui-Xuan et al. [7] for solving the maximum independent set and minimum dominating set problems. In Chapter 4 we discuss the advantage of linear boolean decompositions and present a new algorithm to compute representatives using linear decompositions. In Chapter 5 we study a number of heuristics for generating linear boolean decompositions, including a new heuristic, and evaluate them experimentally. In Chapter 6 we discuss a number of reduction rules that can be used as preprocessing steps when generating linear boolean decompositions. The ideas for this chapter were obtained in collaboration with ten Brinke [21]. In Chapter 7 we study (σ, ρ) vertex subset problems and a dynamic programming algorithm that solves this class of problems. We experimentally compare the bounds of this algorithm to theoretical worst case bounds. Chapter 8 gives a summary of our results and possible topics for future research.

Additionally, in Appendix A we present results on linear boolean-width upper bounds for a large number of graphs. Appendix B contains a list of the most relevant algorithms that were implemented as part of this thesis project. In Appendix C we present the paper *Practical Algorithms for Linear Boolean-width*.

Chapter 2

Preliminaries

In this chapter we introduce to some basic terminology used throughout this thesis. We begin with a brief overview of graph theory that is essential for the concept of boolean-width. For a more extensive introduction to graph theory, we refer the reader to the book *Graph Theory* by Diestel [9]. In Section 2.2 we explain the concept of boolean decompositions and boolean-width, after which we continue to list a number of properties of boolean-width in Section 2.3. In Section 2.4 we explain how running time analysis is performed. In Section 2.5 we provide details about our implementations of the algorithms mentioned in this thesis, and on our machine configuration during the conducted experiments.

2.1 Graph Theory

Definition 2.1 (Graph)**.** A *graph* $G = (V(G), E(G))$ is a pair consisting of a set of *vertices* $V(G)$ and a set of *edges* $E(G)$. The *size* of a graph is equal to the cardinality of the set of vertices, i.e., $|V(G)|$. An edge $\{u, v\}$ is a pair of vertices u and v. If $\{u, v\} \in E(G)$, then u and v are adjacent to each other in G. The vertices u and v are called the *endpoints* of the edge $\{u, v\}$. In this thesis we only work with *simple graphs*. Edges in a simple graph are undirected, meaning the edges have no orientation, and each edge is a pair of two distinct vertices, which disallows self-loops in our graph. Furthermore, there is at most one edge between every pair of vertices in the graph.

Definition 2.2 (Bipartite graph)**.** A *bipartite graph* is a graph whose vertices can be divided into two disjoint sets X and Y, such that all edges have one endpoint in X and one endpoint in Y.

Definition 2.3 (Induced subgraph)**.** For a subset $X \subseteq V(G)$, we define the *induced subgraph* $G[X] = (X, E')$, where the set E' of edges consists of edges from $E(G)$ that only contain vertices in X, i.e., $\forall u, v \in X : \{u, v\} \in E' \leftrightarrow \{u, v\} \in E(G)$.

Definition 2.4 (Induced bipartite subgraph)**.** Two subsets $X, Y \subseteq V(G)$, for which $X \cap Y = \emptyset$, can induce a bipartite subgraph on a graph G. We denote this by $G[X, Y] = (X \cup Y, E')$, where $E' \subseteq E(G)$ consists of all edges in $E(G)$ that have one endpoint in X and one endpoint in Y, i.e., $\forall u \in X, v \in Y : \{u, v\} \in E' \leftrightarrow \{u, v\} \in E(G)$.

Definition 2.5 (Neighborhood)**.** The *neighborhood* of a vertex v in a graph G, also called the *open neighborhood* of v, is defined by $N_G(v) = \{u : \{u, v\} \in E(G)\}$, i.e., the set of vertices that are adjacent to v. The *degree* of a vertex v, $deg(v)$, is equal to $|N_G(v)|$. The *closed neighborhood* of a vertex v is denoted by $N_G[v] = N_G(v) \cup \{v\}$. For a subset $X \subseteq V(G)$, we define the neighborhood $N_G(X) = \bigcup_{v \in X} N_G(v)$.

Definition 2.6 (Vertex subset complement)**.** For a subset $A \subseteq V(G)$, we define the *complement* as the set of vertices $\overline{A} = V(G) \setminus A$.

Definition 2.7 (Cut of a graph)**.** Let $(A, \overline{A})$ be a partition of $V(G)$. Such a partition is called a *cut* of G. Each cut $(A, \overline{A})$ of G can induce a bipartite subgraph $G[A, \overline{A}]$.

Definition 2.8 (Neighborhood across a cut)**.** The *neighborhood across a cut* $(A, \overline{A})$ for a subset $X \subseteq A$ is defined as $N_G(X) \cap \overline{A}$.

Definition 2.9 (Twins)**.** Two vertices $u, v \in V(G)$ are *twins* if it holds that $N_G(u) \setminus v = N_G(v) \setminus u$. For a partition $(A, \overline{A})$ of $V(G)$, two vertices $u, v \in A$ are *twins across the cut* $(A, \overline{A})$ if $N_G(u) \cap \overline{A} = N_G(v) \cap \overline{A}$.

Definition 2.10 (Walk, path and cycle)**.** A *walk* in a graph G is a sequence of vertices, $v_1, \ldots, v_k$, such that $\{v_i, v_{i+1}\} \in E(G)$, for $i = 1$ to $k - 1$. A walk in which no vertex occurs twice is called a *path.* If $v_1 = v_k$ and all other vertices of the sequence are distinct, then it is called a *cycle.*

Definition 2.11 (Tree)**.** A graph $T = (V(T), E(T))$ is called a *tree* if T is connected and contains no cycles. We name the set $V(T)$ *nodes* to distinct a tree from a regular graph. A node $v \in V(T)$ is a *leaf* of T if $deg(v) \leq 1$ and is an *internal node* otherwise. A tree is a *rooted tree* if one node has been designated the root, in which case the edges have a natural orientation towards the root. On a path from a vertex v to the root node, the neighbor u of v on that path is called a *parent* of v. Additionally, v is a *child* of u. A *binary tree* is a rooted tree in which each node in $V(T)$ is either a leaf or has two children.

2.2 Boolean-width

Definition 2.12 (Decomposition tree)**.** A *decomposition tree* of a graph G is a pair (T, δ) where T is a full binary tree and $\delta : V(T) \to 2^{V(G)}$ is a function mapping nodes to subsets of $V(G)$. For the root node r of T, it holds that $\delta(r) = V(G)$. Furthermore, if nodes v and w are children of a node u, then $(\delta(v), \delta(w))$ is a partition of $\delta(u)$. For a decomposition (T, δ) let V_w denote the vertices contained in a node $w \in V(T)$., i.e., $V_w = \delta(w)$. A decomposition (T, δ) is a *full decomposition tree* if T has $|V(G)|$ leaves.

Note that in a full decomposition tree, it holds that for each vertex $v \in V(G)$, $\delta(l) = \{v\}$ for a unique leaf l of T. For each node $x \in V(T)$, we can define the cut $(V_x, \overline{V_x})$, which in turn can induce the bipartite subgraph $G[V_x, \overline{V_x}]$. Consider the graph G displayed in Figure 2.1a and the decomposition tree (T, δ) of G in Figure 2.2. Node x_4 has $V_{x_4} = \{a, b\}$ and induces the cut $(V_{x_4}, \overline{V_{x_4}}) = (\{a, b\}, \{c, d, e, f, g\})$. The bipartite subgraph that is induced by this cut is shown in Figure 2.1b.

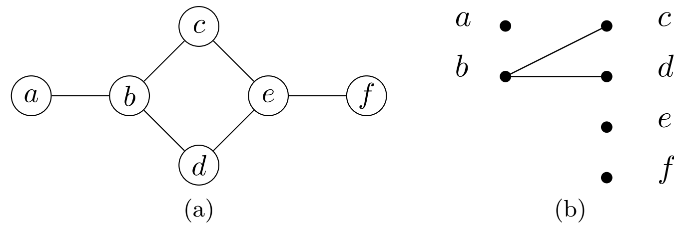


Figure 2.1: A graph G and the bipartite graph induced by node x_4 of the decomposition given in Figure 2.2, or by node x_5 of the linear decomposition given in Figure 2.3.

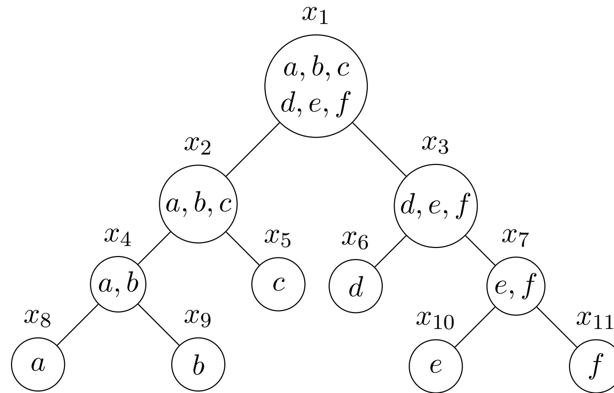


Figure 2.2: A decomposition tree (T, δ) of graph G from Figure 2.1a.

Definition 2.13 (Linear decomposition)**.** A *linear decomposition*, also referred to as a *caterpillar decomposition* [24], is a decomposition tree (T, δ) where T is a binary tree for which it holds that every internal node of T has at least one leaf as a child. We can define such a linear decomposition through a linear ordering $\pi = \pi_1, \ldots, \pi_{|V(G)|}$, which consists of all vertices in G, by letting δ map the i-th leaf of T to π_i.

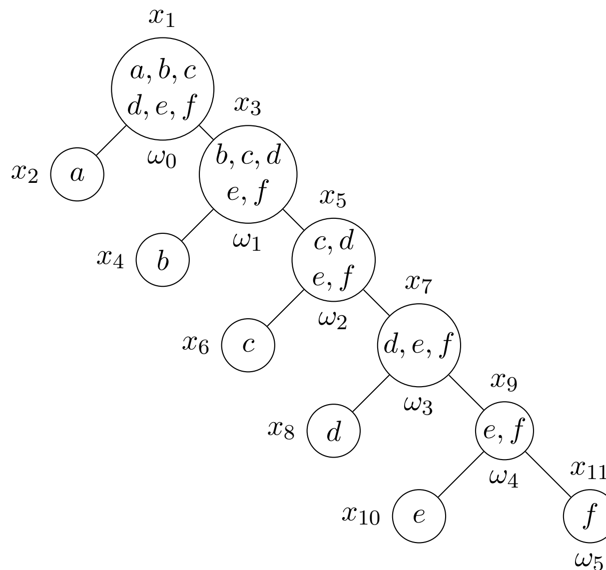


Figure 2.3: A linear decomposition tree of graph G from Figure 2.2 obtained through the linear ordering $\pi = (a, b, c, d, e, f)$.

As an example, let us define a linear decomposition through the linear ordering $\pi = (a, b, c, d, e, f)$ for the graph G displayed in Figure 2.2. We split off the vertices in the order they appear in π. This means that at the root node x_1 we have two child nodes c_1 and c_2 for which $\delta(c_1) = \pi_1$ and $\delta(c_2) = V(G) \setminus \pi_1$. The corresponding linear decomposition tree is shown in Figure 2.3.

For each internal node y, it is possible to directly construct the set of vertices $\delta(y)$ by looking at what vertices are previously split off in the tree. This motivates the following definition:

Definition 2.14 (ω-function)**.** Let $G = (V(G), E(G))$ be a graph and π be a linear ordering of $V(G)$. Let $\omega_i(\pi) : \pi \to 2^{V(G)}$ be a function mapping a linear ordering of vertices to the complement of the first i vertices appearing in that ordering.

$$\omega_i(\pi) = V(G) \setminus \bigcup_{j=1}^{i} \pi_j$$

Let (T, δ) be the linear decomposition tree constructed from π. Let y be the internal node with the i-th leaf of T as a child. If y has two leaf children, then let i be the one of lowest index. We obtain the set of vertices $\delta(y)$ in the following way:

$$\delta(y) = \omega_{i-1}(\pi) = V(G) \setminus \bigcup_{j=1}^{i-1} \pi_j$$

We omit the parameter π and write ω_i instead of $\omega_i(\pi)$ if π is clear from the context.

For the internal node x_5 from the linear decomposition tree of Figure 2.3, we can construct $\delta(x_5)$ given π. At node x_5 the vertices a and b are previously split off, which are π_1 and π_2 in our ordering respectively. Thus, $\delta(x_5) = \omega_2 = V(G) \setminus \{a, b\} = \{c, d, e, f\}$. This makes clear why the ω_i function takes the complement of the first i vertices, as $(\omega_i, \overline{\omega_i})$ now directly gives us a cut of our decomposition. For another example we refer to Figure 2.4.

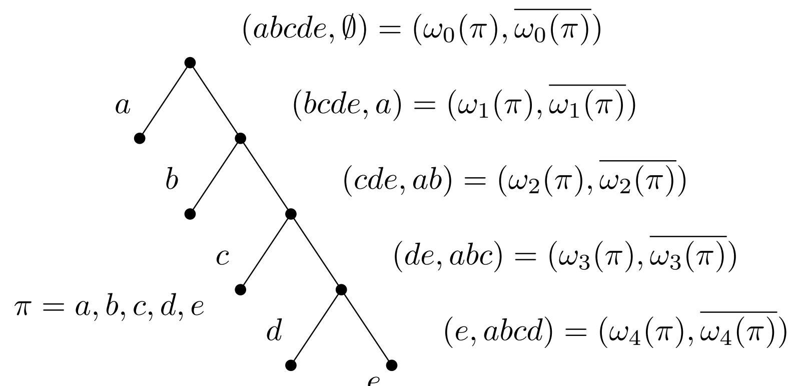


Figure 2.4: Simple linear decomposition of a graph with $V(G) = \{a, b, c, d, e\}$ and corresponding cuts described by the ω function.

Definition 2.15 (Boolean dimension of a cut)**.** Let $G = (V(G), E(G))$ be a graph and $A \subseteq V(G)$. Define the set of *unions of neighborhoods across a cut* $(A, \overline{A})$ as

$$\mathcal{UN}(A) = \left\{ N_G(X) \cap \overline{A} \,\middle|\, X \subseteq A \right\}.$$

The *boolean dimension of a cut* $(A, \overline{A})$ is a function bool-dim : $2^{V(G)} \to \mathbb{R}$.

$$\text{bool-dim}(A) = \log_2 |\mathcal{UN}(A)|.$$

For the cut induced by node x_4 in Figure 2.2, which is equal to the cut induced by node x_5 in Figure 2.3, we have that $\mathcal{UN}(\{a, b\}) = \{\emptyset, \{c, d\}\}$, resulting in the boolean dimension being $\log_2(|\{\emptyset, \{c, d\}\}|) = 1$.

Definition 2.16 (Boolean-width)**.** Let (T, δ) be a decomposition of a graph G. We define the *boolean-width* of (T, δ) as the maximum boolean dimension over all cuts of (T, δ).

$$\text{boolw}(T, \delta) = \max_{x \in V(T)} \text{bool-dim}(\delta(x))$$

The boolean-width of G is defined as the the minimum boolean-width over all possible full decompositions of G.

$$\text{boolw}(G) = \min_{full\ (T,\delta)\ of\ G} \text{boolw}(T, \delta)$$

Definition 2.17 (Linear boolean-width)**.** The *linear boolean-width* of a graph $G = (V(G), E(G))$ of size n is defined as the the minimum boolean-width over all linear decompositions of G.

$$\text{lboolw}(G) = \min_{linear\ (T,\delta)\ of\ G} \text{boolw}(T, \delta)$$

Note that given a linear ordering π of $V(G)$ we can always construct a unique linear boolean decomposition corresponding to π. Therefore we can define the linear boolean-width directly on π.

$$\text{lboolw}(\pi) = \max_{i=0}^{n} \text{bool-dim}(\omega_i(\pi))$$

We omit checking the boolean dimension for leaves since this is less than or equal to 1, which follows from the fact that if x is a leaf node and $\delta(x) = \{v\}$, then $\mathcal{UN}(\{v\}) = \{\emptyset, N_G(v)\}$.

The linear boolean-width of G can be defined as the minimum boolean-width over all possible permutations of $V(G)$.

$$\text{lboolw}(G) = \min_{permutation\ \pi\ of\ V(G)} \text{lboolw}(\pi)$$

2.3 Properties and bounds

There are a number of known properties and bounds for boolean-width that are used throughout this thesis. We give an overview of the most relevant ones below.

Property 2.18. *[12, Theorem 1.2.3] The number of unions of neighborhoods is symmetric for a cut* $(A, \overline{A})$*. This means that* $|\mathcal{UN}(A)| = |\mathcal{UN}(\overline{A})|$*.*

Property 2.19. *[7] For any graph* G*, it holds that* $0 \leq \text{boolw}(G) \leq |V(G)|$*.*

Property 2.20. *[24, Lemma 3.5.7] For any graph* $G = (V(G), E(G))$ *and a vertex* $v \in V(G)$*, it holds that* $\text{boolw}(G \setminus \{v\}) \leq \text{boolw}(G) \leq \text{boolw}(G \setminus \{v\}) + 1$*.*

Property 2.21. *[24, Theorem 3.5.5] For any cut* $(A, \overline{A})$ *of any graph* G*, it holds that* $|\mathcal{UN}(A)| = mis(G[A, \overline{A}])$*, where* $mis(G)$ *is the number of maximal independent sets in a graph* G*.*

Property 2.22. *[15, Proposition 2, Proposition 3] For any graph* G *of size* n*, a trivial upper bound on the boolean-width is* $n/3$*, whereas a trivial upper bound on the linear boolean-width is* $n/2$*.*

Property 2.23. *A linear boolean decomposition is a special case of general boolean decompositions, thus for any graph* G *it holds that* $\text{boolw}(G) \leq \text{lboolw}(G)$*.*

2.4 Running time analysis

We use big O and O^* to indicate the running time of algorithms. Let f and g be two functions, then

- $f(n) \in O(g(n))$ if there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
- $f(n) \in O^*(g(n))$ if $f(n) \in O(g(n) \cdot n^{O(1)})$.

A graph parameter is a number associated to a graph. Let k be a graph parameter of a graph G of size n, then we say that an algorithm is *parameterized by* k if there exists a function f such that the algorithm runs in $f(k) \cdot n^{O(1)}$. A problem solvable by such an algorithm is said to be *fixed parameter tractable* (FPT).

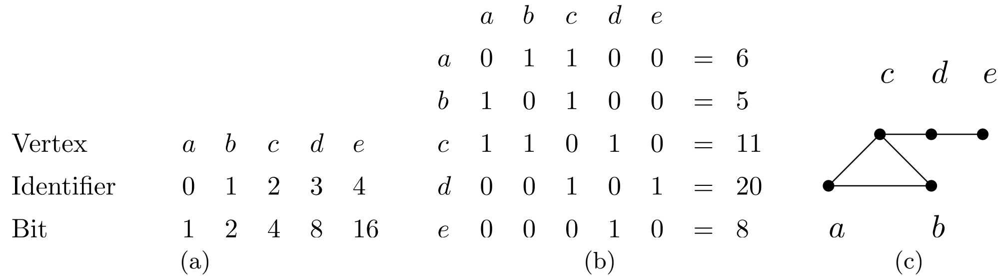
2.5 Implementation details

All algorithms implemented for this thesis project are implemented using the C# programming language and compiled using the *csc* compiler that comes with Visual Studio 12.0.

2.5.1 Bitsets

The algorithms in this thesis make extensive use of sets and set operations, which can be implemented efficiently through the use of bitsets. By using a mapping from vertices to bitsets that represent the neighborhood of a vertex, we can store the adjacency matrix of

a graph efficiently. Figure 2.5 provides an illustration of such a representation through bitsets. The idea is that each vertex receives a unique identifier, which is used to assign a unique bit, $2^{identifier}$, to each vertex. This way, we can add vertices together to construct sets of vertices, which we use to represent neighborhoods. Note that we simply store the integer that represents each neighborhood, and perform bitwise operations directly on these integers.



Vertex	a	b	c	d	e
Identifier	0	1	2	3	4
Bit	1	2	4	8	16

(a)

	a	b	c	d	e		
a	0	1	1	0	0	=	6
b	1	0	1	0	0	=	5
c	1	1	0	1	0	=	11
d	0	0	1	0	1	=	20
e	0	0	0	1	0	=	8

(b)

(c)

Figure 2.5: The identifier and corresponding power of 2 are displayed in 2.5a. The adjacency matrix in 2.5b is used to obtain an integer that represents the neighborhood of a vertex. The corresponding graph is displayed in 2.5c.

While the C# programming language does have a built-in BitArray class, it has some unwanted properties. For instance, if we want to perform an AND-operation given two bit arrays A and B, and assign the value to a new variable C, we lose the values of A or B. The operation of $A.And(B)$ is done in place, which is not a desired property when we work with set operations on neighborhoods of vertices and cuts of the graph. For this reason, we have built our own bitset class that uses 64-bit integers to represent sets of vertices. While we assume that bitset operations take $O(n)$ time and need $O(n)$ space, in practice this may come closer to $O(1)$ because we perform set operations on 64 vertices at once. If one assumes that these requirements are constant, several time and space bounds in this paper improve by a factor n.

2.5.2 Experimental setup

All experiments in this thesis were performed on a 64-bit Windows 8.1 computer, with a 2.20 GHz Intel Core i7-2670QM CPU and 6GB of RAM. The graphs used come from *Treewidthlib* [22], a collection of graphs that are used to benchmark algorithms using treewidth and related graph problems.

Chapter 3

Introduction to boolean decompositions

In this chapter, we provide an introduction to the practical use of boolean decompositions. We start in Section 3.1 by explaining the idea behind boolean-width. In Section 3.2 we give an overview of the algorithm by Bui-Xuan et al. [7] that uses boolean decompositions to solve the maximum independent set problem and minimum dominating set problem.

3.1 Neighborhood equivalence

The unions of neighborhoods of a set $A \subseteq V(G)$ is, as mentioned earlier, a collection of all distinct neighborhoods that occur across the cut $(A, \overline{A})$, i.e., for every set $X \subseteq V(G)$, the set $N_G(X) \cap \overline{A}$ is contained exactly once in the unions of neighborhoods. In order to illustrate how we incorporate these unions of neighborhoods in algorithms, we look at the maximum independent set problem (MIS).

Definition 3.1. (Independent set) Given a graph $G = (V(G), E(G))$ and a subset $X \subseteq V(G)$, X is called an *independent set* if $\forall v, w \in X : \{v, w\} \notin E(G)$. Alternatively, a subset $X \subseteq V(G)$ is an independent set if $\forall v \in X : |N_G(v) \cap X| = 0$. An independent set is called a *maximal independent set* if it cannot be made larger by adding vertices that are not yet contained in the set. An independent set is called a *maximum independent set* if it is the independent set of largest cardinality in the graph.

To solve the MIS we first observe that for an independent set X, it holds that every subset $S \subseteq X$ is also an independent set. Now assume we are given a graph $G = (V(G), E(G))$ and a boolean decomposition (T, δ) of G. Let w be a node of T with child nodes a and b. Let Z be the maximum independent set of the graph. It follows that both $Z \cap V_w$ and $Z \cap \overline{V_w}$ are also independent sets. Thus if we have stored all independent sets $X \subseteq V_w$ and $Y \subseteq \overline{V_w}$, then $Z = X \cup Y$ for some subsets X and Y. This fact holds for every node $w \in V(T)$, shifting the problem to how to compute all independent sets $X \subseteq V_w$ and $Y \subseteq \overline{V_w}$. We can do this recursively since each independent set of $X \subseteq V_w$ can be

constructed through the union of an independent set $S \subseteq V_a$ and $T \subseteq V_b$, as illustrated in Figure 3.1. Note that this only works as long as the sets V_a and V_b are a partition of V_w, but this follows from the definition of boolean decompositions.

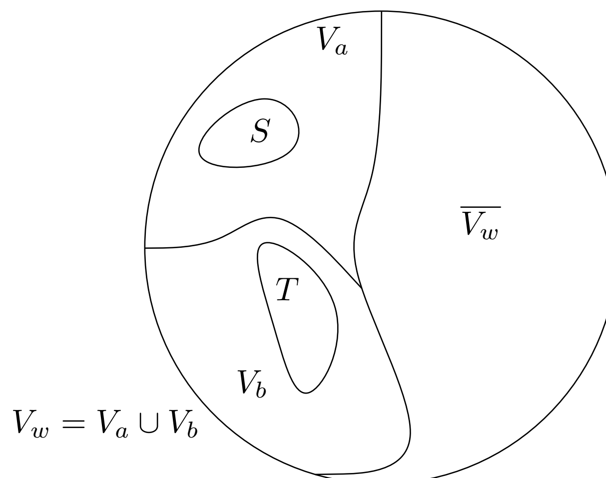


Figure 3.1: Partition of $V(G)$ at a node w.

The described approach leads us to a simple dynamic programming algorithm. We traverse the decomposition tree in a bottom-up fashion and combine the results at each step of the tree. Since at each node w we iterate over all independent sets in a and b, we want to bound the number of distinct sets stored. This is where we make use of equivalent neighborhoods across a cut. Note that if independent sets $S_1 \subseteq V_w$ and $S_2 \subseteq V_w$ have the same neighborhood across the cut $(V_w, \overline{V_w})$, i.e., $N_G(S_1) \cap \overline{V_w} = N_G(S_2) \cap \overline{V_w}$, then for every set $T \subseteq \overline{V_w}$ it holds that $S_1 \cup T$ is an independent set if and only if $S_2 \cup T$ is an independent set. For this reason it is sufficient to store only the largest independent set over all independent sets with the same neighborhood. A bound on the number of distinct possible neighborhoods across a cut is the boolean dimension of that cut.

Definition 3.2 (Neighborhood equivalence)**.** Let $G = (V(G), E(G))$ be a graph and $A \subseteq V(G)$. Two subsets $X, Y \subseteq A$ are said to be *neighborhood equivalent* with respect to $(A, \overline{A})$, denoted by $X \equiv_A Y$, if it holds that $N_G(X) \cap \overline{A} = N_G(Y) \cap \overline{A}$.

Definition 3.3 (Number of equivalence classes)**.** Let $G = (V(G), E(G))$ be a graph and $A \subseteq V(G)$. The number of equivalence classes of $\equiv_A$ is denoted $nec(\equiv_A)$. We define the number of equivalence classes of a decomposition (T, δ) as the maximum number of equivalence classes over all cuts of (T, δ), which we denote by $nec(T, \delta)$.

Proposition 3.4. *[24, Theorem 3.5.5] For any cut $(A, \overline{A})$, it holds that $|\mathcal{UN}(A)| = nec(\equiv_A)$.*

3.2 Algorithms on boolean decompositions

The strategy used to solve problems such as the maximum independent set problem is to first find a representative for each distinct neighborhood. These representatives are used as indices for our dynamic programming table, in which we store partial solutions to our problem.

Definition 3.5 (Representative)**.** Assume we are given a total ordering on the vertices of a graph G. Let (T, δ) be a boolean decomposition. For a node $w \in V(T)$, the *representative* of a set $X \subseteq V_w$ is a set $R \subseteq V_w$ such that R is the lexicographically smallest set for which $R \equiv_{V_w} X$ and $|R|$ is minimized. We denote the representative of a set X by $rep_{V_w}(X)$.

3.2.1 Maximum independent set

To solve the MIS problem we need to construct a list of all representatives and their corresponding neighborhoods for each cut $(V_w, \overline{V_w})$ induced by a node w of a decomposition (T, δ). Algorithm 1 can be used to compute such a list. We let LR_A be the list of representatives for the cut $(A, \overline{A})$, and LNR_A be the list of corresponding neighborhoods. Between each representative and corresponding neighborhood we create a pointer for quick access. Algorithm 1 was developed by Bui-Xuan et al. [7] and runs in $O(\text{boolw}(T, \delta) \cdot n^2 \cdot 2^{\text{boolw}(T,\delta)})$ time for a single node. Because there are $O(n)$ nodes in T, the total running time to construct such a list for every node is $O(\text{boolw}(T, \delta) \cdot n^3 \cdot 2^{\text{boolw}(T,\delta)})$. This algorithm can also be used to calculate the width of a decomposition by keeping track of the maximum number of representatives seen while computing all lists for all nodes of T.

Algorithm 1 Algorithm by Bui-Xuan et al. [7] for computing a list of representatives of a cut $(A, \overline{A})$ and their corresponding neighborhoods.

```
1:  function ComputeRepresentatives(Graph G, Subset A ⊆ V(G))
2:      LR_A, LNR_A ← {∅}
3:      LastLevel ← {∅}
4:      while LastLevel ≠ ∅ do
5:          NextLevel ← ∅
6:          for all R ∈ LastLevel do
7:              for all v ∈ A do
8:                  R' ← R ∪ {v}
9:                  N' ← N_G(R') ∩ Ā
10:                 if R' ≢^d_A R and N' ∉ LNR_A then
11:                     NextLevel ← NextLevel ∪ R'
12:                     LR_A ← LR_A ∪ R'
13:                     LNR_A ← LNR_A ∪ N'
14:                     Add pointers between R' and N'
15:         LastLevel ← NextLevel
16:     return LR_A and LNR_A
```

We define the following table that stores partial solutions of the MIS problem.

$$Tab_w[R] = \begin{cases} \max_{S \subseteq V_w} \{|S| : S \equiv_{V_w} R \text{ and } S \text{ is an IS of } G\}, \\ -\infty \text{ if no such set } S \text{ exists.} \end{cases}$$

For a leaf node l of T, we brute-force set these values for initialization purposes. While Bui-Xuan et al. mention that we should set these values to $Tab_l[\emptyset] = 0$ and $Tab_l[V_l] = 1$,

this is in fact incorrect if there are vertices of degree zero. A vertex v of degree zero will have $\{v\} \equiv_{V_w} \emptyset$, and $\{v\}$ is a valid independent set of size 1. Thus $Tab_l[\emptyset] = 1$. A simple solution to this is setting $Tab_l[\emptyset] = 0$ and $Tab_l[rep_{V_l}(V_l)] = 1$. Here, the latter will overwrite the former if needed.

After setting the values for the leaf nodes, we fill the table entries for all other nodes by combining the values of the child nodes as soon as they are set. For a node w with children a and b, the combine step is given in Algorithm 2.

Algorithm 2 Algorithm by Bui-Xuan et al. [7] for filling the table entries for a node w, used to solve the maximum independent set problem.

1: **procedure** CombineIS
2:     **for all** $R_w \in LR_{V_w}$ **do**
3:         $Tab_w[R_w] \leftarrow 0$
4:     **for all** $R_a \in LR_{V_a}$ **do**
5:         **for all** $R_b \in LR_{V_b}$ **do**
6:             **if** $R_a \cup R_b$ is an IS in $G[R_a, R_b]$ **then**
7:                 $R_w \leftarrow rep_{V_w}(R_a \cup R_b)$
8:                 $Tab_w[R_w] = \max(Tab_w[R_w], Tab_a[R_a] + Tab_b[R_b])$

Theorem 3.6. *[7, Theorem 5] Given a graph $G = (V(G), E(G))$ and a decomposition (T, δ), we can solve the maximum independent set problem on G in $O(\mathrm{boolw}(T, \delta) \cdot n^2 \cdot 2^{2\,\mathrm{boolw}(T,\delta)})$ time.*

3.2.2 Minimum dominating set

Definition 3.7. (Dominating set) Given a graph $G = (V(G), E(G))$ and a subset $X \subseteq V(G)$, X is called a *dominating set* if $X \cup N_G(X) = V(G)$. Alternatively, a set X is a dominating set if $\forall v \in V(G) \setminus X : |N_G(v) \cap X| \neq 0$. A dominating set is called a *minimum dominating set* if it is the dominating set of smallest cardinality in the graph.

The reason for looking at this problem in addition to the MIS problem is that the minimum dominating set problem (MDS) is slightly more complex. This is because a subset of V_w can be dominated by vertices in both V_w and $\overline{V_w}$. It follows that we also need to iterate over all representatives of $\overline{V_w}$ when we perform a combine step in our dynamic programming algorithm.

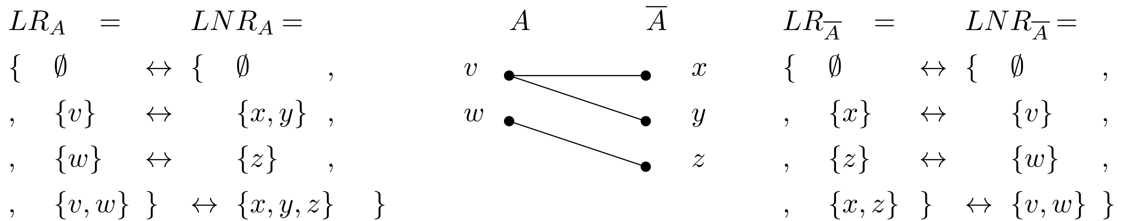


Figure 3.2: The lists LR_A, LNR_A $LR_{\overline{A}}$ and $LNR_{\overline{A}}$ for a cut $(A, \overline{A})$, together with the pointers between the elements of the lists.

We use Algorithm 1 to compute a list of representatives and corresponding neighborhoods for every node $w \in T$. Although the size of the unions of neighborhoods is

symmetric across a cut $(A, \overline{A})$, i.e., $|\mathcal{UN}(A)| = |\mathcal{UN}(\overline{A})|$, the actual sets of neighborhoods and representatives are not the same. We refer to Figure 3.2 for an example. For this reason, we need to compute all representatives and corresponding neighborhoods for both the cut $(V_w, \overline{V_w})$ and $(\overline{V_w}, V_w)$.

Definition 3.8. (Domination) Let $G = (V(G), E(G))$ be a graph and $A \subseteq V(G)$. For $X \subseteq A$, $Y \subseteq \overline{A}$ the pair (X, Y) dominates A if $A \setminus X \subseteq N_G(X \cup Y)$.

An alternative way of saying that (X, Y) dominates A is $\forall v \in A \setminus X : |N_G(v) \cap (X \cup Y)| \neq 0$. Simply put, we are dealing with a partial solution for A and keeping track of how many vertices of the dominating set are inside A itself. We use the following table to store such solutions.

$$Tab_w[R_w][R_{\overline{w}}] = \begin{cases} \min_{S \subseteq V_w}\{|S| : S \equiv_{V_w} R \text{ and } (S, R_{\overline{w}}) \text{ dominates } G\}, \\ \infty \text{ if no such set } S \text{ exists.} \end{cases}$$

For initialization, for a leaf node l of T, we brute-force set the following values, where we let R be the representative of $LR_{\overline{V_l}}$ with $N_G(R) = V_l$.

$$Tab_l[\emptyset][\emptyset] = \infty, Tab_l[V_l][\emptyset] = 1, Tab_l[V_l][R] = 1, Tab_l[\emptyset][R] = 0$$

In addition, we need a final check to see if V_l is a vertex of degree zero. If that is the case, then $Tab_l[\emptyset][\emptyset] = 1$, which will overwrite any previously stored values.

Algorithm 3 Algorithm by Bui-Xuan et al. [7] for filling the table entries for a node w, used to solve the minimum dominating set problem.

```
1: procedure CombineDS
2:     for all R_w ∈ LR_{V_w} do
3:         for all R_w̄ ∈ LR_{V̄_w} do
4:             Tab_w[R_w][R_w̄] ← ∞
5:     for all R_a ∈ LR_{V_a} do
6:         for all R_b ∈ LR_{V_b} do
7:             for all R_w̄ ∈ LR_{V̄_w} do
8:                 R_ā ← rep_{V̄_a}(R_b ∪ R_w̄)
9:                 R_b̄ ← rep_{V̄_b}(R_a ∪ R_w̄)
10:                R_w ← rep_{V_w}(R_a ∪ R_b)
11:                Tab_w[R_w][R_w̄] ← min(Tab_w[R_w][R_w̄],
                       Tab_a[R_a][R_ā] + Tab_b[R_b][R_b̄])
```

Similar to Algorithm 2, we set the table values for a node w with children a and b after the entries for a and b are set. The combine step for the dominating set problem is given in Algorithm 3.

Theorem 3.9. *[7, Theorem 7] Given a graph $G = (V(G), E(G))$ and a decomposition (T, δ), we can solve the minimum dominating set problem on G in $O(n^2 + \text{boolw}(T, \delta) \cdot n \cdot 2^{3\,\text{boolw}(T,\delta)})$ time.*

Note that Algorithm 2 and Algorithm 3 are very similar. In fact, the MIS and MDS problems are contained in a class of problems called (σ, ρ) vertex subset problems, which are all solvable using boolean decompositions. In Chapter 7 we discuss this class of problems and give an overview of the algorithm by Bui-Xuan et al. [8] for solving them.

Chapter 4

Linear boolean decompositions

In this chapter we explain why linear decompositions can be more desirable than general decompositions in practical applications. Recall from Definition 2.13 that linear boolean decompositions can be defined through a linear ordering of the vertices of a graph. For this reason, it is often easier to find a linear decomposition - we just need to find a linear ordering of the vertices of the graph that gives us a low boolean-width. While linear boolean decompositions are a special case of general boolean decompositions, on certain graph classes, such as caterpillar trees and cliques, the linear boolean-width is equal to the boolean-width. This can be observed by noting that, while constructing a linear ordering for a clique or caterpillar tree, there is always a next vertex for the ordering such that the number of unions of neighborhoods does not increase. However, for a simple graph such as a tree, where the boolean-width is 1, finding the linear boolean-width turns out to be a difficult problem. We therefore investigate a new bound on the linear boolean-width in terms of a different graph parameter, *pathwidth*, in Section 4.1. An advantage of linear decompositions is that algorithms using linear decompositions have a lower time complexity than those using general decompositions, which we show in Section 4.2 and 4.3. In Section 4.4 we briefly discuss the topic of how to compute exact linear boolean decompositions.

4.1 Linear boolean-width related to pathwidth

Treewidth [17] is a graph parameter used in a wide range of applications [3]. Similar to boolean-width, treewidth is a value associated with a decomposition of a graph called a *tree decomposition*. In this section, we relate linear boolean decompositions to a special case of tree decompositions called *path decompositions*.

Definition 4.1 (Tree decomposition and treewidth)**.** Given a graph $G = (V(G), E(G))$, a tree decomposition of G is a pair (T, δ), where T is a tree and $\delta : V(T) \to 2^{V(G)}$ is a function mapping nodes to subsets of vertices of $V(G)$. A node $x \in V(T)$ is called a *bag* of the tree decomposition. A tree decomposition (T, δ) satisfies the following properties.

(i) $\bigcup_{x \in V(T)} \delta(x) = V(G)$.

(ii) $\forall\{u,v\} \in E(G) : \exists x \in V(T)$ such that $u, v \in \delta(x)$.

(iii) For $x, y, z \in V(T)$, if y is on the path of T between x and z, then $\delta(x) \cap \delta(z) \subset \delta(y)$.

The *treewidth* of a tree decomposition, denoted $tw(T, \delta)$, is equal to $\max_{x \in V(T)} |\delta(x)| - 1$, i.e., the size of the largest bag minus one. The treewidth of a graph, denoted $tw(G)$, is equal to the smallest treewidth over all possible tree decompositions of G.

It is known that for any graph G, it holds that $\mathrm{boolw}(G) \leq tw(G) + 1$ [24, Theorem 4.2.8]. By restricting the decomposition tree of a tree decomposition to be a path, we obtain a *path decomposition*. The width of such a path decomposition is called the *pathwidth* [16], or *pw* for short. Because T is a path, we can define T through a sequence of subsets $X_1, \ldots, X_n$ of G, where X_i denotes the vertices contained in bag i.

We present a new bound on linear boolean-width in terms of pathwidth by explaining a method of construction that gives us a linear boolean decomposition of a graph G from a path decomposition of G. Recall that a linear boolean decomposition can be defined through a linear ordering $\pi = \pi_1, \ldots, \pi_{|V(G)|}$ of $V(G)$. The idea is that, given a path decomposition $X_1, \ldots, X_n$, we select vertices one by one from a subset X_i and append them to the linear ordering π. We then move on to X_{i+1}. For shorthand notation we denote $\chi_i = \bigcup_{j=1}^{i} X_i$.

Theorem 4.2. *For any graph G it holds that* $\mathrm{lboolw}(G) \leq \mathrm{pw}(G) + 1$.

Proof. Let $S_i = \{u \,|\, u \in \chi_i : N_G(u) \cap \overline{\chi_i} \neq \emptyset\}$. For each $u \in S_i$, it holds that $\exists j > i : \exists w \in X_j$ for which $\{u, w\} \in E(G)$. By definition of a path decomposition, we know that there is a subset X_j with $u, w \in X_j$, and since all subsets containing a certain vertex are subsequent in the path decomposition, it follows that $u \in X_i$ and $u \in X_{i+1}$, implying that $S_i \subseteq X_i$ and $S_i \subseteq X_{i+1}$. By definition, the unions of neighborhoods of χ_i can only consist of neighborhoods of subsets of S_i. It follows that $|\mathcal{UN}(\chi_i)| = 2^{\text{bool-dim}(\chi_i)} \leq 2^{|S_i|} \leq 2^{|X_i|} \leq 2^{\mathrm{pw}(G)+1}$. What remains to be shown is that while appending vertices one by one from a subset X_{i+1}, the number of unions of neighborhoods will not exceed $2^{|X_{i+1}|}$ at any point. For each vertex $v \in X_{i+1}$ there are two possibilities; if $v \in S_i$, then appending v to the linear ordering will not increase the boolean dimension, since v's neighborhood was already an element of the unions of neighborhoods constructed so far; if $v \notin S_i$, then it is possible that v will contribute a new neighborhood to the unions of neighborhoods, which will cause factor 2 increase in the worst case. There are at most $|X_{i+1} \setminus S_i|$ such vertices, and because $S_i \subseteq X_{i+1}$, it follows that $|X_{i+1} \setminus S_i| = |X_{i+1}| - |S_i|$. We conclude that at any point during construction it holds that

$$\mathcal{UN}(\chi_{i+1}) = 2^{\text{bool-dim}(\chi_{i+1})} \leq 2^{|S_i|} \cdot 2^{|X_{i+1}| - |S_i|} = 2^{|X_{i+1}|} \leq 2^{\mathrm{pw}(G)+1}$$

□

4.2 Algorithms parameterized by linear boolean-width

While linear decompositions might have a higher lower bound on the boolean-width of the allowed decompositions than general decompositions, the time complexities for algorithms parameterized by linear boolean-width are lower than those parameterized by boolean-width. A few examples of running times are displayed in Table 4.1.

Problem	Parameterized by	
	Boolean-width	Linear boolean-width
Maximum Independent Set	$O^*(2^{2\,\text{boolw}})$	$O^*(2^{\text{lboolw}})$
Minimum Dominating Set	$O^*(2^{3\,\text{boolw}})$	$O^*(2^{2\,\text{lboolw}})$
Maximum Induced Matching	$O^*(4^{3\,\text{boolw}^2})$	$O^*(4^{2\,\text{lboolw}^2})$

Table 4.1: Running time complexities for solving different problems using algorithms parameterized by (linear) boolean-width [7, 15, 8].

The fact that algorithms using linear decompositions are a factor two faster than those using general decompositions comes from the following observation. We note that in Algorithm 3, for each node w of a tree T, three for-loops are used to iterate over all representatives of $\overline{V_w}$, V_a and V_b, with a and b being the child nodes of w. In the case of linear decompositions, we use the fact that the number of representatives for the leaf child of w is bounded. Assume b is a leaf node, then we only iterate over $\emptyset$ and V_b. Since this holds for every internal node, we get a maximum number of $2^{\text{lboolw}} \cdot 2^{\text{lboolw}} \cdot 2$ combinations that we iterate over in contrast to $2^{\text{boolw}} \cdot 2^{\text{boolw}} \cdot 2^{\text{boolw}}$ for general decompositions. Thus in general, if we have that $2^{2\,\text{lboolw}} < 2^{3\,\text{boolw}}$, then the theoretical upper bound on linear boolean-width is lower than boolean-width.

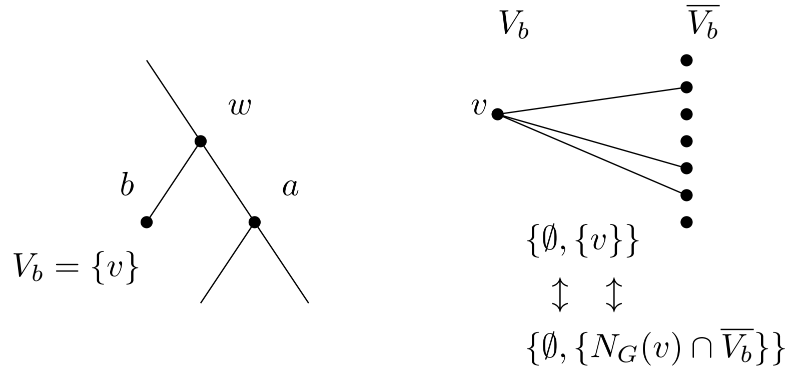


Figure 4.1: Representatives and neighborhoods of the cut $(V_b, \overline{V_b})$, where b is a leaf of a linear decomposition.

4.3 Computing representatives for linear decompositions

Linear boolean decomposition give us - besides a lower running time for algorithms parameterized by linear boolean-width - a faster way to construct representatives. Since calculating the width of a decomposition can be done by keeping track of the maximum number of representatives seen while constructing all representatives, we can judge the quality of a linear decomposition faster than that of a general decomposition. In order

to compute the representatives, we exploit the fact that at each step only one vertex changes sides in the bipartite graph induced by a cut.

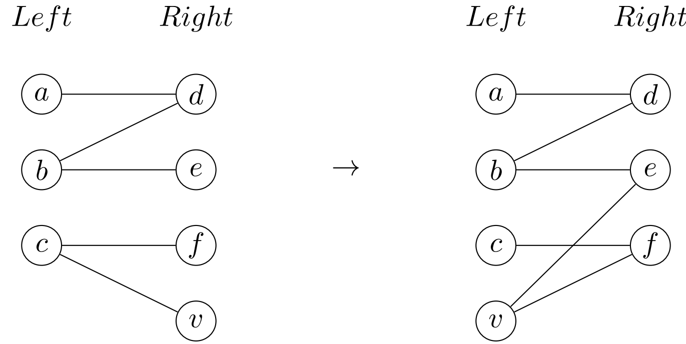


Figure 4.2: Possible changes in the bipartite graph corresponding to a cut after performing an iteration of Algorithm 4.

Using Algorithm 4 we can compute all lists LR_A and LNR_A. Note that in the algorithm we let $LastLevel$ and $NextLevel$ be bidirectional hash maps of representatives and corresponding neighborhoods. By observing what changes in the bipartite graph induced by a cut when one vertex moves from *Right* to $Left$ we can determine the correctness of the algorithm. See Figure 4.2 for an example of a vertex v being processed by the algorithm.

Theorem 4.3. *Let* $G = (V(G), E(G))$ *be a graph and* π *a linear ordering of* $V(G)$*. Using Algorithm 4 we can compute the list of representatives and their corresponding neighborhoods for each cut* $(\omega_i, \overline{\omega_i})$ *in* $O(n^2 \cdot 2^{\text{lboolw}(\pi)})$ *time.*

Proof. Assume that we are processing vertex v at position $i+1$ of a given linear ordering π and that all representatives for the cut $(\omega_{i-1}, \overline{\omega_{i-1}})$ are computed correctly. We want to compute all representatives of the cut $(\omega_{i-1} \cup \{v\}, \overline{\omega_{i-1}} \setminus \{v\}) = (\omega_i, \overline{\omega_i})$. Recall from Definition 2.14 that at step i of the algorithm, it holds that $Left = \overline{\omega_i}$ and $Right = \omega_i$. We first show that each pair of a representative and corresponding neighborhood is a valid pair for the new cut. There are two ways in which the algorithm constructs representatives. The first way is by retrieving the neighborhood of a representative R of step i and removing v from it. This constructs the set $N_G(R \setminus \{v\}) \cap Right$, which is a valid neighborhood across the cut. The second way captures how v interacts with each neighborhood of step i: The algorithm constructs new neighborhoods by adding v to an existing representative, i.e., $R \cup \{v\}$, and obtains the corresponding neighborhood by adding $N_G(v) \cap Right$ to $N_G(R) \cap Right$, which is also a valid neighborhood across the cut. Note that if we encounter multiple sets that give us the same neighborhood across the cut, we only store the actual representative of these sets. This shows that if a pair of a representative and corresponding neighborhood is contained in the map, then it is a valid pair for cut $(\omega_i, \overline{\omega_i})$.

What remains to be shown is that all representatives are contained in the map. Assume for contradiction that there is a representative R that is not contained in the map. If $v \notin R$, then R was a representative at step $i-1$ and we should have encountered it during our iteration over all previous representatives. If R is not contained in the map

after encountering it, then it means we found a set R' for which $R' \equiv_{\omega_i} R$, contradicting that R should be contained in the map. If $v \in R$ then $R \setminus \{v\}$ was not a representative at step $i-1$, or else we would have encountered R during this iteration. Let $R'' = rep_{\omega_{i-1}}(R \setminus \{v\})$. At step i we will construct the set $R'' \cup \{v\}$, for which $R'' \cup \{v\} \equiv_{\omega_i} R$. Again, this contradicts that R is a valid representative, since $R'' \cup \{v\}$ will either have a lower cardinality or be lexicographically smaller than R.

In order to determine the running time, we note that the algorithm iterates over all elements of the ordering π, of which there are exactly n. For each element of the sequence, the algorithm iterates over all representatives that belong to the previous node x of the decomposition, which are at most $nec(\equiv_{V_x}) \leq nec(T, \delta) = 2^{\text{lboolw}(\pi)}$. For each representative, we retrieve the current neighborhood across the cut from the map in $O(n)$ amortized time. Directly computing $N_G(R) \cap Right$ is also possible but would take $|R|$ set operations of $O(n)$ each. Inserting the representative and its neighborhood using Algorithm 5 can be done by hashing the neighborhood in $O(n)$ time and checking for containment in the hash map in $O(1)$ amortized time. Checking for containment can also be done in $O(\log(nec(\equiv_{V_w}) \cdot n)$ time by using a balanced binary search tree. Validating that the representative is lexicographically smaller than the previously stored one also takes $O(n)$ time. It follows that the total running time is $O(n^2 \cdot 2^{\text{lboolw}(\pi)})$.

□

Algorithm 4 Algorithm for constructing all representatives and corresponding neighborhoods for all cuts of a linear decomposition.

```
 1: procedure ComputeLinearRepresentatives(Graph G, Linear ordering π)
 2:     LastLevel ← {∅, ∅}
 3:     Left ← ∅
 4:     Right ← V(G)
 5:     LR_Left, LR_Right ← LastLevel.Representatives
 6:     LNR_Left, LNR_Right ← LastLevel.Neighborhoods
 7:     i ← 1
 8:     while i ≤ |V(G)| do
 9:         NextLevel ← ∅
10:         v ← π_i
11:         Left ← Left ∪ {v}
12:         Right ← Right \ {v}
13:         for all R ∈ LastLevel.Representatives do
14:             N ← LastLevel.GetNeighborhood(R) \ {v}
15:             NextLevel.Update(R, N)
16:             R' ← R ∪ {v}
17:             N' ← N ∪ (N_G(v) ∩ Right)
18:             NextLevel.Update(R', N')
19:         LastLevel ← NextLevel
20:         LR_Left ← LastLevel.Representatives
21:         LNR_Left ← LastLevel.Neighborhoods
22:         i ← i + 1
```

Note that this gives us all the representatives and neighborhoods for all cuts $(\omega_i, \overline{\omega_i})$. In order to get all representatives for all cuts $(\overline{\omega_i}, \omega_i)$ we can simply invert the linear ordering that represents our decomposition and run the algorithm a second time.

Algorithm 5 Procedure that updates the representative of a neighborhood. If we did not encounter this neighborhood before we automatically add it, otherwise a check if performed to see if the new representative R is lexicographically smaller than R'.

1: **procedure** UPDATE(Representative R, Neighborhood N)
2: **if** $N \in Neighborhoods$ **then**
3: $R' \leftarrow GetRepresentative(N)$
4: **if** R is a representative of R' **then**
5: $Remove(R', N)$
6: $Insert(R, N)$
7: **else**
8: $Insert(R, N)$

Even if a binary search tree is used to store the lists of representatives we still achieve a lower running time than the $O(\text{boolw}(T, \delta) \cdot n^3 \cdot 2^{\text{boolw}(T,\delta)})$ used by Algorithm 1. Moreover, the approach of reusing sets of a previous cut turns out to be very helpful when finding linear decompositions, which we will elaborate on in Chapter 5.

4.4 Exact linear decompositions

When using boolean decompositions for applications, we want to use decompositions of low boolean-width. It would be ideal if we could use the optimal decomposition, but finding the optimal decomposition is believed to be NP-hard [24, Section 6.1]. The current best algorithm for finding an exact decomposition runs in $O^*(2^n \cdot 2^{n/3}) = O^*(2.52^n)$ [24, Lemma 6.1.2], which makes generating decompositions unfeasible as soon as the number of vertices in a graph becomes large. For linear decomposition, we present a simple algorithm that runs in $O^*(2.7284^n)$.

For any boolean decomposition, it holds, by definition, that the boolean-width is equal to the highest boolean dimension over all cuts induced by the nodes of the decomposition. Thus for a given subset A, any decomposition that contains the cut $(A, \overline{A})$ has at least a boolean-width of bool-dim(A). This leads us to a simple exact algorithm that for any subset $A \subseteq V(G)$, keeps track of the maximum boolean dimension found so far over all paths leading to this subset. This value is either the boolean dimension of the cut $(A, \overline{A})$ itself or the boolean dimension of a cut that was encountered previously in the decomposition tree. For linear decompositions, we extend our decomposition with one vertex at a time, thus for a subset A we only need to consider all subsets of A where we remove one vertex from A. This gives us the following recurrence relation:

$$P(\emptyset) = 1$$

$$P(A) = \max(|\mathcal{UN}(A)|, \min_{a \in A} P(A \setminus \{a\}))$$

A simple implementation of this recurrence relation is given in Algorithm 6. The algorithm computes the boolean dimension of a cut on the fly using an algorithm for counting the number of maximal independent sets in the bipartite graph induced by the cut $(A, \overline{A})$. For each subset of A, of which there are $O(2^n)$, we eventually recurse. Counting the number of maximal independent sets can be done in $O(1.3642^n)$ time [10], making the total running time $O^*(2^n \cdot 1.3642^n) = O^*(2.7284^n)$.

Algorithm 6 Procedure for computing an optimal linear decomposition. $Tab[A]$ will contain the value of the best boolean decomposition that can be obtained by decomposing $V(G)$ up until A.

```
1: procedure ExactLinear(Subset A ⊆ V(G))
2:     opt ← ∞
3:     for all a ∈ A do
4:         if Tab[A \ {a}] = null then
5:             ExactLinear(A \ {a})
6:         opt ← min(opt, Tab[A \ {a}])
7:     Tab[A] ← max(MaximalIS(G[A, Ā]), opt)
```

There are easier ways to calculate the boolean dimension of a cut, as we will see in Section 5.2.3. This easier method also allows us to stop a search when the boolean dimension exceeds a certain threshold, giving us a way to make an exact algorithm with running time depending on the boolean-width of a graph. For further information we refer the reader to Appendix C, or [21].

Since linear boolean decompositions are often used when finding general decompositions is too time consuming, it seems counter-intuitive to use exact algorithms for finding optimal linear decompositions. This is why in the next chapter we focus on heuristics to construct linear decompositions of low boolean-width.

Chapter 5

Heuristics for generating linear decompositions

The running time of algorithms parameterized by boolean-width is dependent on the boolean-width of the input decomposition. For this reason, we want to be able to compute a 'good' decomposition of a graph, with 'good' meaning a decomposition of low boolean-width. Obviously, the optimal decomposition of a graph is very desirable, but it is often not feasible to compute because of the required exponential time. In this chapter we show different heuristics for finding linear boolean decompositions of low width, which is a common approach used to bypass the difficulty of finding optimal decompositions. We compare these heuristics by both the time required to generate a decomposition and the width of the generated decomposition.

5.1 Generic form of the heuristics

Recall that a linear ordering of the vertices of a graph has a one-to-one correspondence to a linear decomposition of that graph. The strategy when using a heuristic is to find a sequence of the vertices in such a way that the resulting decomposition will be of low linear boolean-width. A simple way to accomplish this is to start the sequence with some vertex and then append a currently unselected vertex to the sequence by some selection criteria. This approach is used in the heuristics introduced by Sharmin [18] and is shown in Algorithm 7.

At any point in the algorithm, we denote the set of all vertices that are contained in the ordering by $Left$ and the remaining vertices by *Right*. While *Right* is not empty, we choose a vertex from a candidate set $\mathit{Candidates} \subseteq \mathit{Right}$ based on a set of trivial cases or, if no trivial case applies, by making a local greedy choice using a score function that indicates the quality of the current state of the cut $(Left, \mathit{Right})$.

Algorithm 7 Greedily generate an ordering based on the score function and the given starting vertex.

1: **function** GenerateVertexOrdering(Graph G, Score function f, Vertex *init*)
2: $\quad$ $Decomposition \leftarrow (init)$
3: $\quad$ $Left \leftarrow \{init\}$
4: $\quad$ $Right \leftarrow V(G) \setminus \{init\}$
5: $\quad$ **while** $Right \neq \emptyset$ **do**
6: $\quad\quad$ $Candidates \leftarrow$ set returned by candidate set strategy
7: $\quad\quad$ **if** there exists $v \in Candidates$ belonging to a trivial case **then**
8: $\quad\quad\quad$ $chosen \leftarrow v$
9: $\quad\quad$ **else**
10: $\quad\quad\quad$ $chosen \leftarrow \underset{v \in Candidates}{\operatorname{argmin}} (f(G, Left, Right, v))$
11: $\quad\quad$ $Decomposition \leftarrow Decomposition \cdot \{chosen\}$
12: $\quad\quad$ $Left \leftarrow Left \cup \{chosen\}$
13: $\quad\quad$ $Right \leftarrow Right \setminus \{chosen\}$
14: **return** *Decomposition*

5.1.1 Selecting the initial vertex

Selecting a good initial vertex can be of great influence on the quality of the decomposition. Sharmin [18] proposes a double breadth first search (BFS) in order to select the initial vertex. This is done by initiating a BFS, starting at an arbitrary vertex, after which a vertex of the last level of the BFS is selected. This process is then repeated by using the found vertex as a starting point for the second BFS. However, using an arbitrary vertex for the first BFS already influences the boolean-width of the computed decomposition. During our experiments we noticed that performing a single BFS sometimes gave better results. Because applications are a lot more expensive in terms of running time compared to computing a decomposition, we propose to use all possible starting vertices when trying to find a good decomposition.

5.1.2 Pruning

Starting from multiple initial vertices allows us to do some pruning. If we notice that the score of the decomposition currently being constructed exceeds the score of the best decomposition found so far, we can immediately stop and move to the next initial vertex. For this reason, it is wise to start with the most promising initial vertices, which can be obtained through the double BFS method, and try all other initial vertices afterwards.

5.1.3 Trivial cases

A vertex is chosen to be the next vertex in the ordering if it can be guaranteed to be an optimal choice by means of a trivial case.

Lemma 5.1. *Let $X \subseteq Left$. If $\exists v \in Right$ such that $N_G(v) \cap Right = N_G(X) \cap Right$, then choosing v will not change the boolean-width of the resulting decomposition.*

Proof. The choice for v will not change the unions of neighborhoods in any way, which means that $\mathcal{UN}(Left) = \mathcal{UN}(Left \cup \{v\})$. Thus for any vertex in $Right \setminus \{v\}$, it will hold that it will interact in the exact same with with $\mathcal{UN}(Left)$ as it would with $\mathcal{UN}(Left \cup \{v\})$, resulting in the boolean dimension of the computed ordering being the same. □

Lemma 5.1 generalizes results by Sharmin [18]. The two trivial cases given by her are sub cases of our lemma, namely $X = \emptyset$ and $X = \{u\}$ for all $u \in Left$. Note that we can add a wide range of trivial cases by varying X, such as $X = Left$ and $\forall u, w \in Left : X = \{u, w\}$, but this will increase the complexity of the algorithm.

5.2 Score functions

We now present the three score functions used in our experiments. While there exist many more possible selection criteria, these three provided us with the best results judging by the boolean-width of the computed decompositions. The first two heuristics, the RelativeNeighborhood and LeastCutValue heuristic, are both designed by Sharmin [18], while the third heuristic, the Incremental-UN heuristic, is a new addition.

5.2.1 Relative neighborhood

The idea of the RelativeNeighborhood heuristic is to minimize the ratio between vertices that are not in $N_G(Left) \cap Right$ and vertices that are. In order to minimize this we define for a cut $(Left, Right)$ and a vertex v the following two sets.

$$Internal(v) = (N_G(v) \cap N_G(Left)) \cap Right$$
$$External(v) = (N_G(v) \setminus N_G(Left)) \cap Right$$

In the original formulation by Sharmin [18], $\frac{|External(v)|}{|Internal(v)|}$ is used as a score function. However, if we use $\frac{|External(v)|}{|Internal(v)|+|External(v)|} = \frac{|External(v)|}{|N_G(v) \cap Right|}$ we get the same ordering without having an edge case for dividing by zero. Furthermore, in contrast to Sharmin, we note that we can compute these sets directly by performing set operations, instead of having to check for each vertex $w \in N_G(v)$ if $w \in N_G(Left) \cap Right$ or not. We will refer to this heuristic by RelativeNeighborhood.

One can easily see that the running time of this algorithm is $O(n^3)$ for a single initial vertex and the required space amounts to $O(n)$. Note, however, that this algorithm only gives us a decomposition. If we need to know the corresponding boolean-width, we need to compute it afterwards, which would require an additional $O(n^2 \cdot 2^k)$ time using Algorithm 4, where k is the boolean-width of the decomposition.

5.2.2 Least cut value

The LeastCutValue heuristic by Sharmin [18] greedily selects the next vertex $v \in Right$ that will have the smallest boolean dimension across the cut $(Left \cup \{v\}, Right \setminus \{v\})$. The intuition behind this approach is that by selecting the vertex that will induce the lowest boolean dimension for the next cut, we should end up with a good decomposition overall. The next vertex is obtained by constructing the bipartite graph $BG = G[Left \cup \{v\}, Right \setminus \{v\}]$ for each $v \in Right$, and counting the number of maximal independent sets of BG. The number of maximal independent sets of BG is equal to the boolean dimension of that cut (see Property 2.21).

We count the number of maximal independent sets by using the CCM_{IS} algorithm developed by Manne and Sharmin [13]. The advantage of the CCM_{IS} algorithm over algorithms that generate all maximal independent sets is that we are not interested in the sets themselves, but solely in the number of sets. As an example of how this can speed up the counting process, note that if a graph has two connected components A and B, the total number of maximal independent sets in $G[A \cup B]$ is $mis(G[A]) \cdot mis(G[B])$.

To determine the running time, we note that for each vertex in $Right$, of which there are $O(n)$, we calculate the boolean dimension of the cut using the CCM_{IS} algorithm. If the graph never gets disconnected, then generating all maximal independent sets would be the same as counting them. Since there are $O(3^{n/3})$ maximal independent sets [14], the runtime of the CCM_{IS} algorithm is exponential in n [13].

5.2.3 Incremental unions of neighborhoods

The approach of the LeastCutValue heuristic of constructing a bipartite graph and then calculating the number of maximal independent sets is a computationally expensive approach. Moreover, we do a lot of unnecessary work, because even for cuts that do not occur in the final decomposition we calculate the number of maximal independent sets.

A more efficient way to compute the boolean dimension of each cut is by reusing the neighborhoods of the previous cut. This approach is similar to Algorithm 4, where we use the representatives of the previous cut to compute the representatives of the current cut. Our new algorithm is displayed in Algorithm 8, which can be used to compute $\mathcal{UN}(X \cup \{v\})$ given $\mathcal{UN}(X)$. In the following it is important that the $\mathcal{UN}$ sets are implemented as hash maps, which will only save distinct neighborhoods.

Algorithm 8 Compute $\mathcal{UN}(X \cup \{v\})$ given $\mathcal{UN}(X)$.

1: **function** Increment-UN(Graph G, Subset $X \subseteq V(G)$, Unions of neighborhoods $\mathcal{UN}_X$, Vertex v)
2: $\quad U \leftarrow \emptyset$
3: $\quad$ **for** $S \in \mathcal{UN}_X$ **do**
4: $\quad\quad U \leftarrow U \cup \{S \setminus \{v\}\}$
5: $\quad\quad U \leftarrow U \cup \left\{(S \setminus \{v\}) \cup (N_G(v) \cap (\overline{X} \setminus \{v\}))\right\}$
6: $\quad$ **return** U

Lemma 5.2. *The procedure Increment-UN is correct and runs in $O(n \cdot |\mathcal{UN}_X|)$ time using $O(n \cdot |\mathcal{UN}_X|)$ space.*

Proof. For proof by induction, assume that all unions of neighborhoods for the cut $(X, \overline{X})$ saved inside the set $\mathcal{UN}_X$ are computed correctly. For each neighborhood in $\mathcal{UN}_X$, we only perform two actions to obtain new neighborhoods. The first action is removing v, since v cannot be in any neighborhood of $X \cup \{v\}$. The second operation is adding $N_G(v)$ to an existing neighborhood, which also results in a valid new neighborhood across the cut. It is clear that if a neighborhood is added to U, then it is a valid neighborhood across the cut $(X \cup \{v\}, \overline{X} \setminus \{v\})$. We now show that all valid neighborhoods of the cut $(X \cup \{v\}, \overline{X} \setminus \{v\})$ are contained in U. Assume for contradiction that S is a valid neighborhood not contained in U. By definition, there is a set R for which $N_G(R) \cap (\overline{X} \setminus \{v\}) = S$. If $v \notin R$, then $N_G(R) \cap \overline{X} \in \mathcal{UN}_X$, meaning that we add $N_G(R) \cap (\overline{X} \setminus \{v\})$ to U, contradicting our assumption. If $v \in R$, then $N_G(R \setminus \{v\}) \cap \overline{X} \in \mathcal{UN}_X$. During the algorithm we construct $(N_G(R \setminus \{v\}) \cup N_G(v)) \cap (\overline{X} \setminus \{v\})$, which is equal to $N_G(R) \cap (\overline{X} \setminus \{v\})$. This means that $N_G(R) \cap (\overline{X} \setminus \{v\})$ is added to U, also contradicting our assumption. It follows that a neighborhood is contained in the set U if and only if it is a valid neighborhood across the cut $(X \cup \{v\}, \overline{X} \setminus \{v\})$.

The running time is determined by the number of sets S saved in $\mathcal{UN}_X$. The number of unions of neighborhoods that we iterate over does not exceed 2^k, where k is the boolean dimension of $\mathcal{UN}_X$. The set operations that are performed for each S take at most $O(n)$ time. This results in the total time for this algorithm to be $O(n \cdot |\mathcal{UN}_X|)$. The space requirements amount to $O(n \cdot |\mathcal{UN}_X|)$ for storing U, which contains at most $O(|\mathcal{UN}_X|)$ sets of size at most $O(n)$. □

We use Algorithm 8 to compute the sizes of all candidate cuts and select the one that gives the lowest boolean dimension. The full heuristic that we obtain, called the Incremental Unions of Neighborhoods heuristic (IUN heuristic for short), is shown in Algorithm 9. Note that we need to add some additional bookkeeping when a trivial case occurs, since the unions of neighborhoods should remain up to date at all times.

Theorem 5.3. *The* Incremental-UN-heuristic *procedure runs in $O(n^3 \cdot 2^k)$ time using $O(n \cdot 2^k)$ space, where k is the boolean-width of the resulting linear decomposition.*

Proof. The running time is determined by the number of sets saved in $\mathcal{UN}_{Left}$. The worst case consisting of $Candidates = Right$ will result in at most n iterations and calls to Increment-UN. This call takes $O(n \cdot |\mathcal{UN}_{Left}|)$ time by Lemma 5.2. By definition $|\mathcal{UN}_{Left}|$ never exceeds 2^k, where k is the boolean-width of the resulting decomposition. Because we need to make n greedy choices to process the entire graph, we conclude that the total time for this algorithm is $O(n^3 \cdot 2^k)$. For the space requirements we observe that all structures in the algorithm require $O(n)$ space, except for the unions of neighborhoods. Since there are only stored two of them at any time and they require at most $O(n \cdot 2^k)$ space, the total space requirements amount to $O(n \cdot 2^k)$. □

Algorithm 9 Greedy heuristic that incrementally keeps track of the Unions of Neighborhoods.

```
 1: function INCREMENTAL-UN-HEURISTIC(Graph G, Vertex init)
 2:     Decomposition ← (init)
 3:     Left, Right ← {init}, V(G) \ {init}
 4:     UN_Left ← {∅, N_G(init) ∩ Right}
 5:     while Right ≠ ∅ do
 6:         Candidates ← set returned by candidate set strategy
 7:         if there exists v ∈ Candidates belonging to a trivial case then
 8:             chosen ← v
 9:             UN_chosen ← INCREMENT-UN(G, Left, UN_Left, v)
10:         else
11:             for all v ∈ Candidates do
12:                 UN_v ← INCREMENT-UN(G, Left, UN_Left, v)
13:                 if |UN_v| < |UN_chosen| then
14:                     chosen ← v
15:                     UN_chosen ← UN_v
16:         Decomposition ← Decomposition · chosen
17:         Left ← Left ∪ {chosen}
18:         Right ← Right \ {chosen}
19:         UN_Left ← UN_chosen
20:     return Decomposition
```

A useful property of Algorithm 9 is that the running time is output sensitive. It follows that if a decomposition is not found within reasonable time, then the decomposition that would have been generated is not useful for practical algorithms.

5.3 Experiments

In order to get an idea of how the IUN heuristic performs compared to existing heuristics, we judge them by both the boolean-width of the generated decomposition and the time needed for computation. As mentioned in Section 2.5.2 the graphs in our experiments come from *Treewidthlib* [22], and are the same set of graphs used by Sharmin in her experiments [18, Chapter 8].

We ran the three different heuristics mentioned in Section 5.2 with $Candidates = Right$ and with an additional two variations on the IUN heuristic by varying the set of starting vertices. The first variation, named 2-IUN, has two starting vertices which are obtained through a single and double BFS respectively. The n-IUN heuristic uses all possible starting vertices. For all other heuristics we obtained the starting vertex through performing a double BFS search. In Table 5.1 and 5.2 we present the results of our experiments.

It is expected that the IUN heuristic and LeastCutValue heuristic give the same linear boolean-width, since both these heuristics greedily select the vertex that minimizes the boolean dimension. The RelativeNeighborhood heuristic performs worse than all other heuristics in nearly all cases. While the difference might not seem very large,

Table 5.1: Linear boolean-width of the decompositions returned by different heuristics, with $Candidates = Right$.

Graph	$\|V\|$	Edge Density	Relative	LeastCut	IUN	2-IUN	n-IUN
alarm	37	0.10	3.32	3.00	3.00	3.00	3.00
barley	48	0.11	5.70	5.91	5.91	4.70	4.58
pigs-pp	48	0.12	10.35	7.13	7.13	7.13	6.64
BN_100	58	0.17	15.84	11.56	11.56	10.86	10.86
eil76	76	0.08	8.86	8.33	8.33	8.33	8.33
david	87	0.11	9.38	6.27	6.27	6.27	5.86
1jhg	101	0.17	12.86	8.67	8.67	8.49	8.41
1aac	104	0.25	20.29	12.40	12.40	12.40	12.33
celar04-pp	114	0.08	11.67	7.27	7.27	7.27	7.27
1a62	122	0.21	18.92	11.68	11.68	11.28	11.14
1bkb-pp	127	0.18	16.81	9.98	9.98	9.53	9.53
1dd3	128	0.17	16.61	9.98	9.98	9.90	9.90
miles1500	128	0.64	8.17	5.58	5.58	5.58	5.29
miles250	128	0.05	7.95	7.13	7.13	5.39	4.58
celar10-pp	133	0.07	10.32	11.95	11.95	7.64	6.91
anna	138	0.05	12.65	8.67	8.67	8.51	7.94
pr152	152	0.04	12.69	11.19	11.19	10.36	8.29
munin2-pp	167	0.03	15.17	9.61	9.61	9.61	7.61
mulsol.i.5	186	0.23	7.55	5.29	5.29	5.29	3.58
zeroin.i.2	211	0.16	7.92	4.46	4.46	4.46	3.81
boblo	221	0.01	19.00	4.32	4.32	4.32	4.00
fpsol2.i-pp	233	0.40	5.58	6.07	6.07	5.78	4.81
munin4-wpp	271	0.02	13.04	9.27	9.27	9.27	7.61

note that algorithms parameterized by boolean-width are exponential in the width of a decomposition. The 2-IUN heuristic outperforms IUN in 11 cases while n-IUN gives a better decomposition in 20 out of 23 cases, which shows that a good initial vertex is of great influence on the width of the decomposition.

Looking at the running times displayed in Table 5.2 for computing each decomposition, we see that the RelativeNeighborhood heuristic is significantly faster. This is to be expected because of the $O(n^3)$ time, compared to the exponential time for all other heuristics. An interesting comparison we can make is the difference between the IUN heuristic and LeastCutValue heuristic. While both of these heuristics give the same decomposition, IUN is significantly faster. Additionally, even 2-IUN and n-IUN are often faster than the LeastCutValue heuristic.

In addition to these experiments, we ran the IUN heuristic on a number of large graphs that were previously dismissed by Sharmin as being too computationally expensive for a heuristic that greedily minimizes the boolean dimension [18]. In order to limit the running time, we only use a single starting vertex which is obtained through a double BFS search, and let the set $Candidates$ be equal to $N_G(Left \cup N_G(Left)) \cap Right$. Vertices not contained in this set will definitely be a worse option than any vertex in this set, as they will not have any shared neighbors. The results are displayed in Table 5.3. Note that the values of the two others heuristics are taken from [18]. Missing

entries are caused by a lack of internal memory which is caused by the $O(n \cdot 2^k)$ space requirement.

Table 5.2: Time in seconds of the heuristics used to find the linear boolean decompositions of which the boolean-width is displayed in Table 5.1.

Graph	$\|V\|$	Edge Density	Relative	LeastCut	IUN	2-IUN	n-IUN
alarm	37	0.10	< 0.01	0.02	< 0.01	< 0.01	0.06
barley	48	0.11	< 0.01	0.18	0.01	0.02	0.16
pigs-pp	48	0.12	< 0.01	0.76	0.02	0.04	0.52
BN_100	58	0.17	< 0.01	25.10	0.41	1.24	17.17
eil76	76	0.08	0.02	5.00	0.13	0.29	8.35
david	87	0.11	0.02	3.15	0.04	0.06	1.62
1jhg	101	0.17	0.03	24.46	0.21	0.48	14.75
1aac	104	0.25	0.04	754.54	5.66	11.81	375.31
celar04-pp	114	0.08	0.04	5.73	0.14	0.23	9.85
1a62	122	0.21	0.06	585.95	3.10	11.57	376.26
1bkb-pp	127	0.18	0.06	198.05	1.14	4.18	107.32
1dd3	128	0.17	0.07	117.21	0.92	2.74	91.19
miles1500	128	0.64	0.06	44.57	0.10	0.14	7.05
miles250	128	0.05	0.02	0.56	0.05	0.10	1.24
celar10-pp	133	0.07	0.06	8.93	1.96	4.72	18.43
anna	138	0.05	0.06	20.81	0.22	0.57	19.95
pr152	152	0.04	0.10	50.74	1.76	5.66	120.06
munin2-pp	167	0.03	0.11	3.81	0.80	3.37	30.21
mulsol.i.5	186	0.23	0.09	37.88	0.13	0.27	8.80
zeroin.i.2	211	0.16	0.06	18.70	0.09	0.11	5.85
boblo	221	0.01	0.29	3.39	0.28	0.56	46.22
fpsol2.i-pp	233	0.40	0.18	189.11	0.36	0.74	56.63
munin4-wpp	271	0.02	0.61	57.87	1.98	6.66	367.37

For more results, we refer to Appendix A, Table A.1, where we have used the IUN heuristic on a large number of graphs, comparing the obtained linear boolean-width of the decomposition to the currently known lowest boolean-width and treewidth for that graph.

Table 5.3: Width of linear boolean decompositions found with the IUN heuristic using the starting vertices returned by performing a double BFS, and with $candidates = N_G(Left \cup N_G(Left)) \cap Right$ in order to decrease the computation time. The last column indicates the time need by the IUN heuristic.

Graph	$\|V\|$	Edge Density	LeastUncommon	Relative	IUN	Time (s)
link-pp	308	0.02	34.81	28.68	17.44	610.09
diabetes-wpp	332	0.01	8.58	18.58	5.32	1.53
link-wpp	339	0.02	35.00	29.03	16.79	374.04
celar10	340	0.02	20.81	15.00	10.17	1.83
celar11	340	0.02	19.54	14.70	10.80	1.88
rd400	400	0.01	34.73	21.32	17.01	1,007.03
diabetes	413	0.01	29.32	19.32	-	-
fpsol2.i.3	425	0.10	15.87	8.92	7.67	2.11
pigs	441	0.01	24.04	18.00	12.39	20.08
celar08	458	0.02	24.95	15.00	10.17	2.12
d493	493	0.01	20.29	48.10	16.73	708.57
homer	561	0.01	36.22	28.49	-	-
rat575	575	0.01	16.48	37.23	-	-
u724	724	0.01	18.72	50.09	-	-
inithx.i.1	864	0.05	11.98	7.22	6.81	7.31
munin2	1003	< 0.01	31.25	12.13	11.91	61.17
vm1084	1084	< 0.01	15.21	48.95	-	-
BN_24	1819	< 0.01	4.91	2.32	2.58	610.72
BN_25	1819	< 0.01	4.64	2.32	2.58	601.41
BN_23	2425	< 0.01	8.48	3.17	2.58	1,808.29
BN_26	3025	< 0.01	6.98	2.32	3.58	4,532.83

5.4 Conclusion

In this chapter we presented a new greedy heuristic for finding linear boolean decompositions, called the IUN heuristic. The heuristic has a running time that is significantly lower than the previous best heuristic and finds a decomposition in output sensitive time. This means that if a decomposition is not found within reasonable time, then the decomposition that would have been generated is not useful for practical algorithms. Because of the improved running time we can run the heuristic multiple times for different starting vertices, resulting in significantly better decompositions compared to existing heuristics. Furthermore, we have generalized the trivial cases that might occur during the execution of a heuristic, which makes it easier to detect choices that are locally optimal. Additional investigation in obtaining even faster heuristics can be proven to be worthwhile, for instance by finding a good approximation algorithm for the size of the unions of neighborhoods. Combining properties of the IUN heuristic and the RelativeNeighborhood heuristic might also lead to better decompositions, as they make use of complementary features of a graph. Another approach for obtaining good decompositions could be a branch and bound algorithm that makes use of trivial cases that are used in the heuristics.

Chapter 6

Reduction rules for linear boolean-width

Finding decompositions of low boolean-width is a difficult problem in itself, which is why we have looked at heuristics in Chapter 5. To ease this process, we can make use of preprocessing steps in order to speed up the computation of decompositions. The idea behind these preprocessing steps is that we apply a certain number of reduction rules to our input graph. These rules remove certain vertices and all edges incident to these vertices from the graph. By doing this we obtain a new graph called the reduced graph, which can be used as input for a heuristic. We make sure that the linear boolean-width of the reduced graph is equal to the linear boolean-width of the original input graph. We then reverse the applied reduction rules to expand the decomposition of the reduced graph into a decomposition of the original graph of equal boolean-width. A schematic overview is presented in Figure 6.1, where at each step the linear boolean-width is required to remain unchanged. Because the reduced graph will have less vertices, it will speed up the computation of a decomposition, since there are less options when deciding the next vertex for a linear ordering of the vertices of the graph that is being constructed by a heuristic.

$$\text{Graph } G \xrightarrow{\textit{reducing}} \text{Graph } H \xrightarrow{\textit{decomposing}} \pi' \text{ of } H \xrightarrow{\textit{expanding}} \pi \text{ of } G$$

Figure 6.1: Steps for applying reduction rules.

We start this chapter with a number of definitions in Section 6.1. In Section 6.2 we explain how reduction rules can be found and proven to be correct. In Section 6.3 we investigate known reduction rules for boolean decompositions. The rules do not automatically apply to linear boolean decompositions, since linear decompositions are more restrictive in their tree construction. In Section 6.4 we look at reduction rules for treewidth and check their validity for boolean-width. In Section 6.5 we provide a number of new reduction rules. In Section 6.6 we present ways to combine properties of linear decompositions with reduction rules for general decompositions.

6.1 Definitions

Definition 6.1 (Reduction rule)**.** A rule r is called a *reduction rule* if it can do the following: Given a graph $G = (V(G), E(G))$, r can derive a reduced graph $H = (V(H), E(H))$ by removing a certain number of vertices and all edges incident to these vertices. We denote $G \xrightarrow{r} H$ to indicate that H is obtained by reducing G using rule r.

Definition 6.2 (Safe reduction rule)**.** Let $G = (V(G), E(G))$ be a graph and let r be some reduction rule. Let $H = (V(H), E(H))$ and $G \xrightarrow{r} H$. Then r is called *safe* if $\text{lboolw}(G) = \text{lboolw}(H)$.

Note that by applying a reduction rule to a graph G it always holds that $\text{lboolw}(H) \leq \text{lboolw}(G)$. This follows from Property 2.20 of boolean-width which states that removing vertices cannot increase the boolean-width of a graph.

In order to revert the changes made by a safe reduction rule, there should be a reverse operation on a linear decomposition obtained from the reduced graph that gives us a valid linear decomposition of the original graph.

Definition 6.3 (Expansion method)**.** Let $G = (V(G), E(G))$ be a graph and let r be some reduction rule. Let $H = (V(H), E(H))$ and $G \xrightarrow{r} H$. A method e is called an *expansion method for rule* r if e can construct a linear decomposition (T, δ) of G for each linear decomposition (T', δ') of H We denote this by $(T', \delta') \xrightarrow{e} (T, \delta)$.

Since there is a bijection between a linear ordering π and a linear decomposition (T, δ) it is sufficient if a rule e can construct a linear ordering π of $V(G)$ out of a linear ordering π' of $V(H)$, denoted by $\pi' \xrightarrow{e} \pi$.

Definition 6.4 (Safe expansion method)**.** Let e be some expansion method for a reduction rule r. Let $G = (V(G), E(G))$ and $H = (V(H), E(H))$ be graphs with $G \xrightarrow{r} H$. Then e is called *safe* if for all linear decompositions (T', δ') of H it holds that if $(T', \delta') \xrightarrow{e} (T, \delta)$ then $\text{lboolw}(T, \delta) \leq \text{lboolw}(T', \delta')$.

Even though adding vertices cannot decrease the boolean-width, we do not require strict equality for expansion methods. Rather, we allow for the obtained decomposition of G to be of an even lower boolean-width than the decomposition of H on which we are expanding. The reason for this is that we might reorder a decomposition π' of H into a new decomposition π'' of H, after which we expand into a decomposition π of G with $\text{lboolw}(\pi) = \text{lboolw}(\pi'') \leq \text{lboolw}(\pi')$. We consider this reordering to be part of the expansion method.

Definition 6.5 (Position in a linear decomposition)**.** Let π be a linear ordering of the vertices of a graph. Let v and w be two distinct vertices contained in π. Let i be the position of v and j be the position of w in π, i.e., $\pi_i = v$ and $\pi_j = w$. We use $v <_\pi w$ to denote $i < j$, meaning that v appears before w in π.

6.2 Proving reduction rules

In order for a reduction rule to be valid and safe, we need to show that both the reduction and expansion step will not change anything to the linear boolean-width of a decomposition. This motivates the following lemma:

Lemma 6.6. *Let $G = (V(G), E(G))$ be a graph and let r be some reduction rule. Let $H = (V(H), E(H))$ and $G \xrightarrow{r} H$. Let e be the expansion method of r. If e is a safe expansion, then r is a safe reduction from G to H.*

Proof. Let π' be the optimal linear ordering of $V(H)$ with $\text{lboolw}(\pi') = \text{lboolw}(H)$. We know that there is some linear ordering π of $V(G)$ for which $\pi' \xrightarrow{e} \pi$. Since e is safe, it holds by definition that $\text{lboolw}(\pi) \leq \text{lboolw}(\pi')$. We know that by adding vertices to a decomposition, the boolean-width cannot decrease, so we can conclude that $\text{lboolw}(\pi') = \text{lboolw}(\pi) = \text{lboolw}(G)$. Thus $\text{lboolw}(H) = \text{lboolw}(G)$, which makes r a safe reduction rule. □

To prove the safeness of a reduction rule it is sufficient to show that the expansion method of a reduction rule is safe for all linear decompositions of H to linear decompositions of G. The safeness of the reduction rule follows from Lemma 6.6.

In order to disprove existing reduction rules, we make use of the following observation that follows directly from Definition 6.2.

Observation 6.7. *For a reduction rule r and two graphs G and H, if $G \xrightarrow{r} H$ and $\text{lboolw}(H) < \text{lboolw}(G)$, then r is not a safe reduction rule.*

In other words, it suffices to show that after applying the reduction the resulting graph has a lower linear boolean-width than the original graph. We can also conclude this from the fact that there is no safe expansion method for this reduction rule, since for the linear ordering π',with $\text{lboolw}(\pi') = \text{lboolw}(H)$, there will be no linear ordering π of $V(G)$ with $\text{lboolw}(\pi) = \text{lboolw}(\pi')$.

6.3 Validity of general boolean-width reduction rules

A starting point for finding reduction rules for linear boolean decompositions is the validating or disproving of known reduction rules for general boolean decompositions. We focus on the three reduction rules described in [18, Chapter 5]. In this section we show that two of the three rules that hold for boolean decompositions do not hold when applied to linear boolean decompositions. We consider the following three rules.

1. **Islet rule.** If $deg(v) = 0$ then v can be removed.
2. **Pendant rule.** If $|E(G)| > 1$ and $deg(v) = 1$ then v can be removed.
3. **Twin rule.** If $|E(G)| > 1$ and $N_G(u) = N_G(v)$ or $N_G[u] = N_G[v]$ then u can be removed.

6.3.1 Islet rule

Lemma 6.8. *Let $G = (V(G), E(G))$ be a graph and let r be the islet rule. Let $H = (V(H), E(H))$ and $G \xrightarrow{r} H$. We can construct a safe expansion method e for r.*

Proof. Let v be the vertex of degree 0 that has been removed from G. By definition of a degree 0 vertex it holds that $N_G(v) = \emptyset$. Let π' be any linear ordering of $V(H)$. For any value of i, it holds that $\emptyset \in \mathcal{UN}(\omega_i(\pi'))$. It follows that at any position in π', we can insert v without increasing the boolean dimension, hence the linear boolean-width of any linear ordering π where we have inserted v at any position will be equal to the linear boolean-width of π', and π is a valid linear ordering of $V(G)$. Thus inserting v at any position in π' to obtain π is a safe expansion method. □

6.3.2 Pendant rule

Lemma 6.9. *Let $G = (V(G), E(G))$ be a graph and let r be the pendant rule. Let $H = (V(H), E(H))$ and $G \xrightarrow{r} H$. Applying the pendant rule can result in* $\text{lboolw}(H) < \text{lboolw}(G)$.

Proof. Assume we are constructing a decomposition of graph H pictured in Figure 6.2. Any optimal decomposition will have a boolean-width of 1, for instance the decomposition obtained from the linear ordering $\pi' = (a, b, e, c, d)$. If we obtained this graph by applying the pendant rule to some initial graph, then a possible initial graph would be graph G. However, any optimal decomposition from an ordering π of G will have at least one cut $(\omega_i, \overline{\omega_i})$ where $|\mathcal{UN}(\omega_i)| = 3$, for instance $\pi = (a, b, e, c, d, v)$, resulting in a boolean-width of $\log_2(3) \approx 1.58$. Thus $\text{lboolw}(H) < \text{lboolw}(G)$, meaning we cannot reduce a graph using the pendant rule without assuring the boolean-width does not change. □

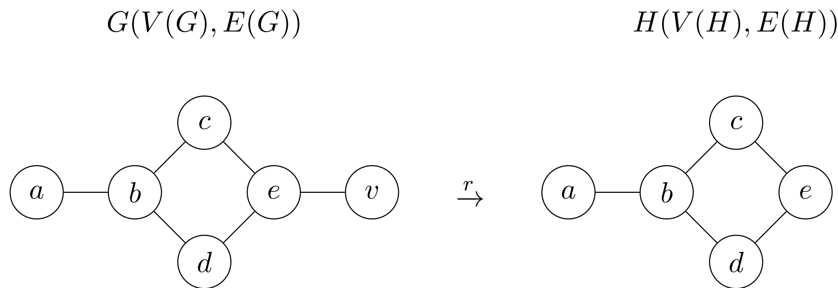


Figure 6.2: Counterexample to the pendant rule. The linear ordering (a, b, e, c, d) of $V(H)$ would have no safe expansion method to obtain a linear ordering of $V(G)$.

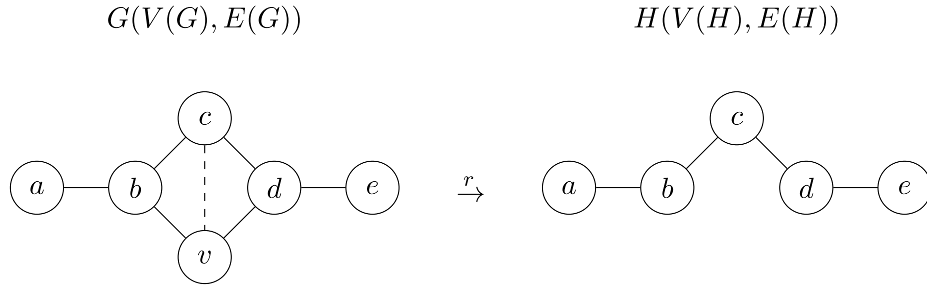


Figure 6.3: Counterexample to the twin rule. The linear ordering (a, b, c, d, e) would have no safe expansion method to obtain a linear ordering of $V(G)$.

6.3.3 Twin rule

Lemma 6.10. *Let $G = (V(G), E(G))$ be a graph and let r be the twin rule. Let $H = (V(H), E(H))$ and $G \xrightarrow{r} H$. Applying the twin rule can result in* $\text{lboolw}(H) < \text{lboolw}(G)$.

Proof. Assume we are constructing a linear decomposition of graph H pictured in Figure 6.3. Any optimal linear decomposition will have a maximum boolean-width of 1, for instance by using the linear ordering $\pi' = (a, b, c, d, e)$. If the twin rule is used to obtain graph H, then it is possible that graph G was our original input graph. If $\{c, v\} \notin E(G)$, then we obtain the same graph as in Figure 6.2, of which the boolean-width is approximately 1.58. If $\{c, x\} \in E(G)$, then it also holds that we cannot construct a linear decomposition in which we will not reach $|\mathcal{UN}(\omega_i)| = 3$ for some cut $(\omega_i, \overline{\omega_i})$. This means that regardless of $\{c, x\} \in E(G)$ or not, $\text{lboolw}(G) \approx 1,58$, thus $\text{lboolw}(H) < \text{lboolw}(G)$. □

Following from Lemma 6.8, 6.9 and 6.10, we conclude that only the islet rule can be applied on graphs in order to reduce the number of vertices when finding an optimal linear boolean decomposition.

Theorem 6.11. *The islet rule is a safe reduction rule for linear boolean-width.*

6.4 Validity of treewidth reduction rules

There are multiple preprocessing rules known for treewidth [6, 4, 5], that often work well in practice. While it has been shown that removing simplicial vertices cannot be used as a preprocessing step for boolean-width [18], for a number of other treewidth reduction rules it is still an open problem whether they are valid (linear) boolean-width. In this section we investigate several of these known rules for treewidth.

Definition 6.12. (Almost simplicial vertex) A vertex v is called *almost simplicial* if all neighbors of v except one form a clique.

For treewidth there exists a rule to reduce almost simplicial vertices. This is done by taking the single neighbor w of v that is not in the clique, after which we remove v and connect w to every vertex in the clique, see Figure 6.4 for an example. Note that vertices of degree 2 are always almost simplicial by definition.

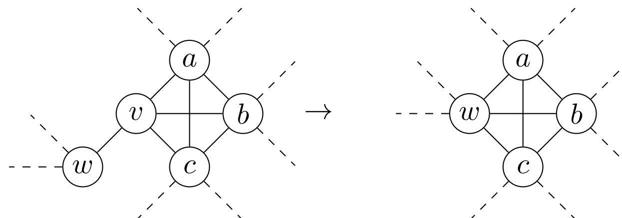


Figure 6.4: Application of the almost simplicial reduction rule for tree-width.

We present a counterexample to the almost simplicial rule for (linear) boolean-width, shown in Figure 6.5. If we remove the almost simplicial vertices v_1 and v_2, then the remaining graph will be a clique. The boolean-width of a clique is 1, whereas the boolean-width of the original graph is larger than 1. Therefore we cannot remove almost simplicial vertices without assuring that the boolean-width does not decrease.

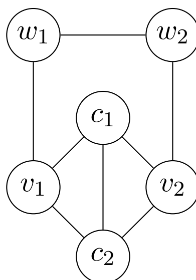


Figure 6.5: Counterexample to the almost simplical vertex rule.

Definition 6.13. (Separator) Let $G = (V(G), (E(G))$ be a graph. Let $S \subseteq V(G)$. The set S is called a separator of G if $G[V(G) \setminus S]$ has more than one connected component. S is called a *minimal separator* if there is no proper subset of S that is also a separator.

For treewidth we have the following property for separators that are also a clique.

Proposition 6.14. *[5] Given a clique separator S and a graph G, the treewidth of G is equal to the maximum over all connected components Z of $G[V(G) \setminus S]$ of the treewidth of $G[Z \cup S]$.*

If a graph has a clique separator then it is possible to compute tree decompositions for parts of the graph. Afterward, the decompositions for these parts can be merged together. If one wants to quickly find a bound on the treewidth, then it is sufficient to check the components induced by these clique separators. A similar property also holds for minimal *almost clique separators*. An almost clique separator is a separator in which all vertices minus one form a clique. It has been shown that reducing minimal almost clique separators is safe for treewidth [5].

Unfortunately, linear boolean-width does have the property that we can determine the linear boolean-width by looking at separate connected components. We refer to Figure 6.6 and Figure 6.7 for a counterexample for clique separators and almost clique

separators respectively. In both cases, the graph induced by a connected component together with the separator $\{s_1, s_2, s_3\}$ will have a linear boolean-width that is strictly lower than the linear boolean-width of the original graph.

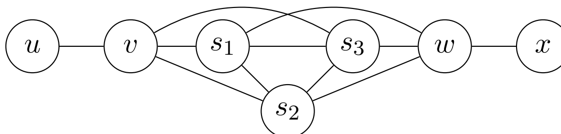


Figure 6.6: Counterexample to the clique separation rule.

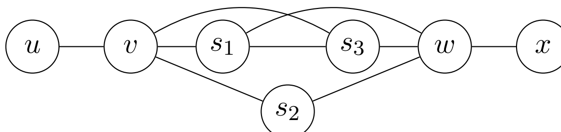


Figure 6.7: Counterexample to the almost clique separation rule.

The boolean-width of a graph H, with H being a minor of a graph G, is not always smaller than the boolean-width of G itself [18]. This property does hold for treewidth and is used as a building block for most reduction rules, which lead us to believe that most treewidth reduction rules do not hold for boolean-width. Furthermore, this result shows that it is harder to find valid reduction rules for boolean-width, since contracting edges is often an unsafe operation when reducing graphs.

6.5 New reduction rules

In this section we present two new reduction rules that are valid for linear boolean-width. We also present a few ideas for other reduction rules for which we omit the proof of correctness. The approach we take for proving these reduction rules is to show that there is a safe expansion method for every decomposition of a reduced graph, after which it follows from Lemma 6.6 that the reduction rule is safe.

When computing a decomposition, a choice is made at every step of which vertex should be selected at that point. We use the terminology of 'making a choice' in our proofs; we investigate the change in the boolean dimension cuts when a vertex gets chosen at that position, i.e., we consider the influence of a vertex v on all the unions of neighborhoods of $(A, \overline{A})$ and all other cuts when v gets chosen for the cut $(A \cup \{v\}, A \setminus \{v\})$.

6.5.1 Sequence rule

Definition 6.15 (Sequence rule)**.** Let $G = (V(G), E(G))$ be a graph. Let $s, u, v, w, x \in V(G)$ such that all are distinct. Let $\{s\}$ be a separator of the graph. Let $N_G(u) = \{s, v\}$, $N_G(v) = \{u, w\}$, $N_G(w) = \{v, x\}$ and $N_G(x) = \{y\}$. Applying the *sequence rule* to G removes the vertex x and the edge $\{w, x\}$ from the graph.

To clarify, if r is the sequence rule and $G \xrightarrow{r} H$, then $V(H) = V(G) \setminus \{x\}$ and $E(H) = E(G) \setminus \{w, x\}$. An example of the sequence rule is illustrated in Figure 6.8.

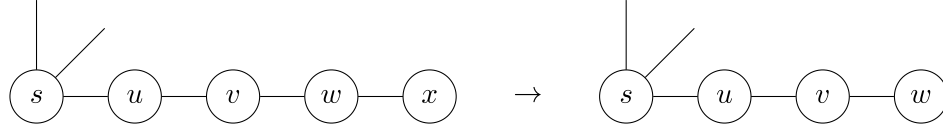


Figure 6.8: The sequence rule applied to a graph G.

Lemma 6.16. *Let $G = (V(G), E(G))$ be a graph. Let r be the sequence rule and let $G \xrightarrow{r} H$. For any linear decomposition π' of H we can construct a decomposition π'' of H for which* $\text{lboolw}(\pi'') \leq \text{lboolw}(\pi')$.

Proof. Let $s, u, v, w \in V(H)$ be vertices described as in Definition 6.15. Let y be defined as the vertex of the set $\{u, v, w\}$ that has the lowest index in π', with $\pi_i = y$. We construct π'' by copying π' up until position i, leaving the boolean dimension for all cuts up until i unchanged.

Assume $s <_{\pi'} y$. If $y = w$, then we can see that the boolean dimension can increase more than when $y = u$. For π'' we therefore choose to put vertex u at position i, after which we directly insert v and w. When v gets inserted, every neighborhood that has v in it, which are only neighborhoods that have u as a representative, are replaced with neighborhoods with w and v being the representative. Once w is chosen, the boolean dimension decreases and the boolean dimension of all later cuts will remain unchanged. Thus for π'', we have $\pi''_i = u, \pi''_{i+1} = v$ and $\pi''_{i+2} = w$, which will possibly result in a lower boolean width.

Assume $y <_{\pi'} s$. Regardless of which vertex y represents, the influence on the boolean dimension is the same. Thus a valid choice for step i for π'' would be vertex w. We now apply the same reasoning as in the previous case. After w we can directly insert v and u without increasing the boolean dimension of any cut prior to step i. Furthermore, the boolean dimension for all cuts after w, v and u also remains unchanged or will decrease. Thus for π'', we let $\pi''_i = w, \pi''_{i+1} = v$ and $\pi''_{i+2} = u$.

Both cases lead to $\text{lboolw}(\pi'') \leq \text{lboolw}(\pi')$. □

Theorem 6.17 (Sequence rule)**.** *The sequence rule is a safe reduction rule for linear boolean-width.*

Proof. Let $G = (V(G), E(G))$ be a graph and let r be the sequence rule. Let $G \xrightarrow{r} H$. We show that a safe expansion method exists for r. Let π' be any linear boolean decomposition of H and let $s, u, v, w, x \in V(G)$ be vertices as described in Definition 6.15. We first construct the decomposition π'' for which $\text{lboolw}(\pi'') \leq \text{lboolw}(\pi')$ by applying the rearrangement technique described in Lemma 6.16 to π'. In order to construct a decomposition π of G, we distinguish between two different cases. For both these cases let $\pi''_i = w$.

Assume $s <_{\pi''} w$. We construct π by copying π'' and inserting x directly after w. This will not influence the boolean-dimension of any later cuts, since $N_G(x) \cap \overline{\omega_{i+1}} = \emptyset$. Any cuts before step i will also remain unchanged, since w is the only neighbor of x. Only the cut $(\omega_i, \overline{\omega_i})$ could possibly have an increased boolean dimension, but it can

be observed that bool-dim(ω''_{i-1}) = bool-dim(ω_i), i.e., the boolean dimension when v is chosen in π'' is equal to the boolean dimension of when w is chosen in π. It follows that lboolw(π'') = lboolw(π).

Assume $w <_{\pi''} s$. We construct π by copying π'' and inserting x directly in front of w. Because w is the only neighbor of x, no cuts before step $i-1$ have a change in boolean dimension. Both w in π'' and x in π have one neighbor across their respective cut, which gives us bool-dim(ω''_i) = bool-dim(ω_{i-1}), i.e., the boolean dimension when w is chosen in π'' is equal to the boolean dimension of when x is chosen in π. The boolean dimension when w is chosen in π gives us the same situation as when v is chosen in π''. Since the remainder of the decompositions are equal, we can conclude that the boolean dimension remains the same across all later cuts. It follows that lboolw(π'') = lboolw(π).

In summary, lboolw(π) = lboolw(π'') and lboolw$(\pi) \leq$ lboolw(π'), meaning that we can safely insert x using this expansion method. □

Note that we can reduce any path starting in a separator and ending in a pendant vertex, with the other vertices of the path being of degree 2, to a path of length 4 by applying the sequence rule multiple times.

6.5.2 Clique rule

Definition 6.18 (Clique rule)**.** Let $G = (V(G), E(G))$ be a graph. Let $S = \{s_1, \ldots, s_{|S|}\}$ be a minimal separator of G. Let $s_1, \ldots, s_{|S|}, v, w, c_1, \ldots, c_n \in C \subseteq V(G)$ such that all are distinct members of the same clique C of size $|S| + n + 2$. Let $v, w, c_1, \ldots, c_n$ have no other neighbors besides vertices in the clique. Applying the *clique rule* to G removes all vertices $c_1, \ldots, c_n$ and incident edges from G.

In other words, if r is the clique rule and $G \xrightarrow{r} H$, then $V(H) = V(G) \setminus \{c_1, \ldots, c_n\}$ and $E(H) = \{\{x, y\} \mid \{u, v\} \in E(G) \wedge u \neq c_1 \vee \cdots \vee c_n\}$. An example is illustrated in Figure 6.9, where the separator S is the singleton $\{s\}$.

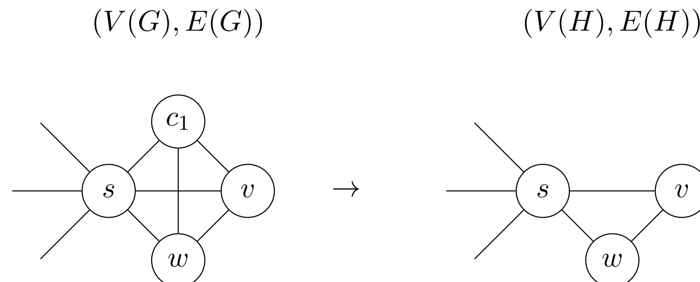


Figure 6.9: Clique rule applied to a graph G.

Theorem 6.19 (Clique rule)**.** *The clique rule is a safe reduction rule for linear boolean-width.*

Proof. Let $G = (V(G), E(G))$ be a graph and let r be the clique rule. Let $G \xrightarrow{r} H$. We show that a safe expansion method exists for r.

Assume we are given a decomposition π' of H and that we have identified the vertices $s_1, \ldots, s_{|S|}, v, w, c_1, \ldots, c_n \in C \subseteq V(G)$ as described in Definition 6.18. Let $\pi'_i = v$. Without loss of generality, it holds that $v <_{\pi'} w$. To construct π we copy π' and leave the order of vertices unchanged. We expand π by inserting all vertices $c_1, \ldots, c_n$ directly after v, meaning $\pi_{i+j} = c_j$ for $j = 1, \ldots, n$. A change in boolean dimension can only happen for neighborhoods where a neighbor of a vertex c_j suddenly is a unique representative for this vertex, while it was not a unique representative before. However, since $\forall s, s' \in S : N_G[s] \cap C = N_G[s'] \cap C$ and because each vertex in S already is a representative of v, it follows for all cuts before step i the boolean dimension does not change. Because $N_G[v] = N_G[w] = N_G[c_j]$, every vertex c_j will not contribute a neighborhood to the union of neighborhoods. Thus for every cut $(\omega_{i+j}, \overline{\omega_{i+j}})$, it holds that $\text{bool-dim}(\omega_{i+j}) = \text{bool-dim}(\omega_i)$, meaning the expansion method has no influence on any later cuts either. In summary, no neighbor of c_j will have an increase of the boolean dimension at their corresponding cut, which results in $\text{lboolw}(\pi) = \text{lboolw}(\pi')$. We conclude that we can always construct a linear boolean decomposition π of G from π' of H while keeping the boolean-width equal. □

Additionally, we can apply the same rearrangement technique as we did with the sequence rule by assuring that vertex w gets chosen directly after v in π'. This can have a positive effect on the boolean-width, resulting in $\text{lboolw}(\pi) \leq \text{lboolw}(\pi')$.

6.5.3 Other reduction rules

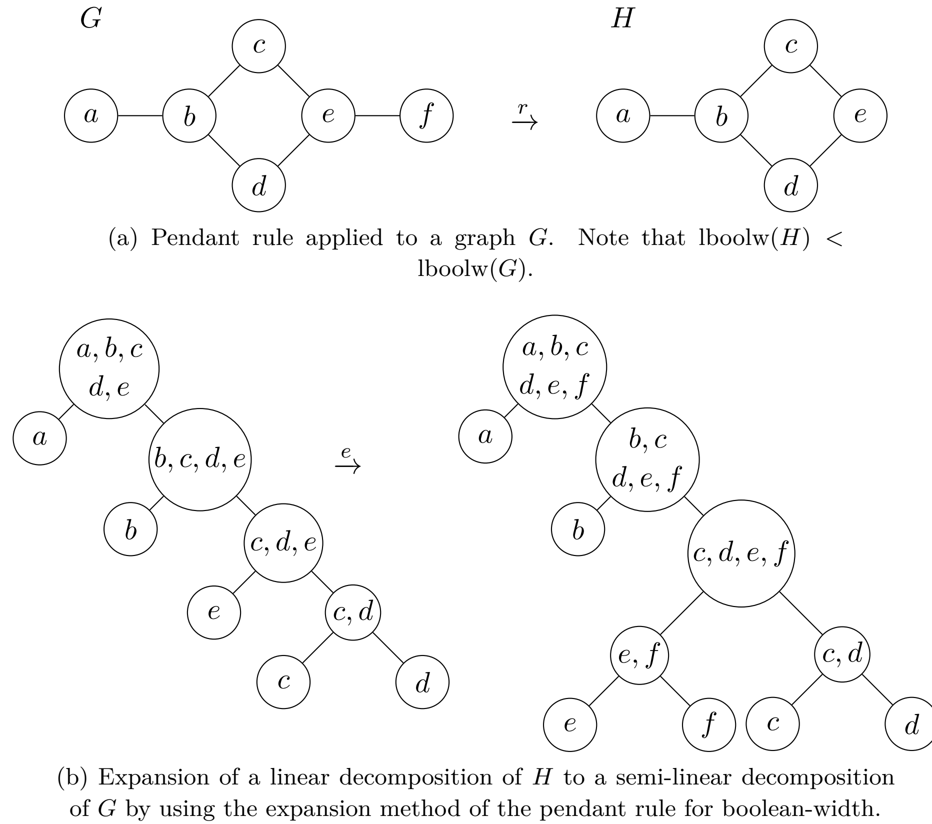
The technique to find reduction rules is to first identify the stage at which a choice for a vertex is guaranteed to be optimal for a decomposition. We then see if we can expand this sequence of optimal choices to a larger sequence while keeping the boolean dimension unchanged. A graph structure that exhibits this property is for instance a *caterpillar tree*. A caterpillar tree is a tree in which all vertices are within distance 1 of a central path. We believe that any caterpillar tree that is separated from the rest of the graph can be reduced through the sequence rule, since we make a sequence of optimal choices when applying the sequence rule. For structures such as caterpillar trees or cliques, the optimal choice for a linear decomposition is very obvious, which led us to the reduction rules of the previous section. Another interesting question is if we can shorten a path of arbitrary length of degree 2 vertices to a fixed length. For a path between two separators it also holds that, once we have chosen a vertex of that path, we consider vertices neighboring to the chosen vertex to be the next vertex for our decomposition. All in all, we believe that the two reduction rules from the previous section can be expanded upon and more cases can be found with additional research.

6.6 Expanding linear decompositions using general reduction rules

The reason for choosing linear decomposition over general decompositions is that linear decompositions make dynamic programming algorithms easier and result in a lower theoretical running time. Furthermore, using the heuristics presented in Chapter 5 it is much easier to construct a linear decomposition. However, as can be seen in the previous section, the reduction rules for linear boolean-width are far from practical, in contrast to the reduction rules that hold for boolean-width. Therefore we propose to combine the best of both worlds.

1. Start with a graph $G = (V(G), E(G))$.
2. Apply reduction rules for boolean-width (islet, pendant and twin rule) on G to obtain a reduced graph H. Since the reduction rules that are valid for linear boolean-width are sub cases of the reduction rules for boolean-width, we do not need to apply them.
3. Use a heuristic on H to obtain a linear decomposition (T', δ').
4. Expand (T', δ') to a decomposition (T, δ) of G using expansion methods described in [18, Chapter 5]. By definition of reduction rules we know that $\text{boolw}(T, \delta) = \text{lboolw}(T', \delta')$.

Note that the decomposition that we end up with at step 4 is not a linear decomposition. However, this decomposition does exhibit linear properties, and therefore we call them *semi-linear* decompositions. Consider the dynamic programming algorithm for solving the dominating set problem as described in Chapter 3. The advantage of using a linear decomposition is that instead of a $O(2^{3k})$ algorithm we can get a $O(2^{2k})$ algorithm, with k being the boolean-width of a decomposition. This follows from the fact that at each combine step of two nodes of the linear decomposition we know that one of the two nodes is a leaf node. This bounds the number of representatives that can occur for this node by two; the empty set and the neighborhood of the vertex itself. When working with semi-linear decompositions we have added vertices to a linear decomposition in a way that guaranteed the boolean-width to remain equal. Moreover, the number of representatives at each cut remains unchanged, which means that even though the decompositions is not linear anymore, at each combine step we can still guarantee that the number of representatives in one of the child nodes is bounded by two. We can conclude that algorithms on semi-linear decompositions have the same theoretical running time as linear decompositions. We refer to Figure 6.10 for an example of the construction of such a decomposition.



(a) Pendant rule applied to a graph G. Note that $\text{lboolw}(H) < \text{lboolw}(G)$.

(b) Expansion of a linear decomposition of H to a semi-linear decomposition of G by using the expansion method of the pendant rule for boolean-width.

Figure 6.10: Application of the pendant rule.

6.7 Conclusion

The reduction rules described in this chapter occur in very specific cases, which we believe do not happen often in practical applications. What these rules do show is that preprocessing for linear boolean-width is much harder than for boolean-width or treewidth. This can be a barrier when trying to find decompositions for very large graphs where preprocessing is a necessity. Therefore we suggest the approach of using reduction rules for general boolean decompositions, after which a heuristic for linear decompositions can be used on the reduced graph. We believe that this will give very good bounds in practice, but additional research is required to verify our intuition. For this reason we propose more investigation into what makes certain rules valid and if there are more practical rules that can be applied for boolean-width.

Chapter 7

Vertex subset problems

In this chapter we look at how to solve a large class of vertex subset problems, called (σ, ρ) vertex subset problems, which were introduced by Telle [19]. This class of problems consists of finding a (σ, ρ)-set of maximum or minimum cardinality and contains well known problems such as the maximum independent set, the minimum dominating set, and the maximum induced matching problem. It was shown by Telle and Proskurowski [20] that they are solvable using treewidth in $O^*(2^{O(tw)})$. Van Rooij et al. later improved upon this by by using subset convolutions [23]. Gerber and Kobler [11] provided an algorithm for solving vertex subset problems using clique-width as a parameter in $O^*(2^{2^{poly(cw)}})$. Boolean decompositions can be used to efficiently solve these problems, with the running time exponential in the boolean-width of the decomposition, and polynomial in the input size. The main motivation to use boolean-width as a parameter for solving these problems is that boolean-width is considered to be a practical parameter, which can lead to easy to implement algorithms and easy to find decompositions of low boolean-width.

In Section 7.1 we start by defining the properties of vertex subset problems and give a number of definitions that aid in describing the algorithm by Bui-Xuan et al. [8] for solving these problems. The running time of this algorithm is $O^*(nec_d(T, \delta)^3)$ [8], where $nec_d(T, \delta)$ is the number of equivalence classes of a problem specific equivalence relation, which can be bounded in terms of boolean-width. In Section 7.2 we give a number of theoretical bounds on the number of equivalence classes. The algorithm itself and implementation details are shown in Section 7.3. In Section 7.4 we conduct a number of experiments on a selection of graphs, such as testing the feasibility of solving vertex subset problems and how close the value of $nec_d(T, \delta)$ approaches any of the theoretical bounds.

σ	ρ	d	Standard terminology	max	min
{0}	$\mathbb{N}$	1	Independent set	NPC	P
$\mathbb{N}$	$\mathbb{N}^+$	1	Dominating set	P	NPC
{0}	$\mathbb{N}^+$	1	Independent Dominating set	NPC	NPC
$\mathbb{N}^+$	$\mathbb{N}^+$	1	Total Dominating set	P	NPC
{0, 1}	{1}	2	Weakly Perfect Dominating set	NPC	NPC
$\mathbb{N}$	{1}	2	Perfect Dominating set	P	NPC
{1}	{1}	2	Total Perfect Dominating set	NPC	P
{0}	{1}	2	Efficient Dominating set or Perfect Code	NPC	NPC
{0}	{0, 1}	2	Strong Stable set or 2-Packing	NPC	P
$\mathbb{N}$	{0, 1}	2	Nearly Perfect set	P	P
{0, 1}	{0, 1}	2	Total Nearly Perfect set	NPC	NPC
{1}	$\mathbb{N}$	2	Induced Matching	NPC	P
{1}	$\mathbb{N}^+$	2	Dominating Induced Matching	NPC	NPC
$\mathbb{N}$	$\geq d$	d	d-Dominating set	P	NPC
{d}	$\mathbb{N}$	d+1	Induced d-Regular Subgraph	NPC	?

Table 7.1: A selection of vertex subset problems taken from [19, 8], together with the corresponding values for σ, ρ and $d(\sigma, \rho)$, with $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. The max and min column indicate the complexity of finding such a (σ, ρ)-set with maximum or minimum cardinality, with P, NPC and ? indicating Polytime, NP-Complete and unknown respectively.

7.1 Definitions

Definition 7.1 ((σ, ρ)-set)**.** Let $G = (V(G), E(G))$ be a graph. Let σ and ρ be finite or co-finite subsets of $\mathbb{N}$. A subset $X \subseteq V(G)$ is called a (σ, ρ)-set if the following holds:

$$\forall v \in V(G) : |N_G(v) \cap X| \in \begin{cases} \sigma & \text{if } v \in X, \\ \rho & \text{if } v \in V(G) \setminus X. \end{cases}$$

We refer to Table 7.1 for an overview of some vertex subset problems.

In order to confirm if a set X is a (σ, ρ)-set, we have to count the number of neighbors a vertex $v \in V(G)$ has in X. Let us consider the case where we are interested in an independent set, which is equivalent to checking if a set X is a $(\{0\}, \mathbb{N})$-set. Any vertex $v \in X$ cannot have a vertex in X as its neighbor, since if $v \in X$, then $|N_G(v) \cap X|$ should be 0. If v turns out to have a neighbor in X, then it is irrelevant if v has more than one neighbor in X, since there is no difference between a vertex having one, two or n neighbors for this problem instance; any number of neighbors invalidates X being a $(\{0\}, \mathbb{N})$-set.

We capture the property of only having to count up until a certain number of neighbors in the function $d : 2^{\mathbb{N}} \to \mathbb{N}$, which is defined as follows.

Definition 7.2 (d-function)**.** Let $d(\mathbb{N}) = 0$. For every finite or co-finite set $\mu \subseteq \mathbb{N}$, let $d(\mu) = 1 + \min(\max_{x \in \mathbb{N}} x : x \in \mu, \max_{x \in \mathbb{N}} x : x \notin \mu)$. Let $d(\sigma, \rho) = \max(d(\sigma), d(\rho))$.

Definition 7.3 (d-neighborhood)**.** Let $G = (V(G), E(G))$ be a graph. Let $A \subseteq V(G)$ and $X \subseteq A$. The *d-neighborhood* of X with respect to A, denoted by $N_A^d(X)$, is a multiset of vertices from $\overline{A}$, where a vertex $v \in \overline{A}$ occurs $\min(|N_G(v) \cap X|, d)$ times in $N_A^d(X)$. A d-neighborhood can be represented as a vector of length $|\overline{A}|$ over $\{0, 1, \ldots, d\}$.

Definition 7.4 (d-neighborhood equivalence)**.** Let $G = (V(G), E(G))$ be a graph and $A \subseteq V(G)$. Two subsets $X, Y \subseteq A$ are said to be *d-neighborhood equivalent* with respect to $(A, \overline{A})$, denoted by $X \equiv_A^d Y$, if it holds that $\forall v \in \overline{A} : \min(|N_G(v) \cap X|, d) = (|N_G(v) \cap Y|, d)$. The number of equivalence classes of a cut $(A, \overline{A})$ is denoted by $nec(\equiv_A^d)$. The number of equivalence classes of a decomposition (T, δ) is defined as $\max(nec(\equiv_A^d), nec(\equiv_{\overline{A}}^d))$ over all cuts $(A, \overline{A})$ of (T, δ), which we denote by $nec_d(T, \delta)$.

Note that $N_A^1(X) = N(X) \cap \overline{A}$. It can then be observed that $|\mathcal{UN}(A)| = nec(\equiv_A^1)$ [24, Theorem 3.5.5]. Furthermore, we have that $X \equiv_A^d Y$ if and only if $N_A^d(X) = N_A^d(Y)$.

We now generalize the notion of a representative to allow us to define representative for sets that have equal d-neighborhoods. Note that for $d = 1$, the definition is equal to the definition of a representative given in Definition 3.5.

Definition 7.5 (Representative)**.** Let (T, δ) be a boolean decomposition. For a node $w \in V(T)$, the *representative* of a set $X \subseteq V_w$ is a set $R \subseteq V_w$ such that R is the lexicographically smallest set for which $R \equiv_{V_w}^d X$ and $|R|$ is minimized. We denote the representative of a set X by $rep_{V_w}^d(X)$.

7.2 Bounds on the number of d-equivalence classes

In Chapter 3 we only had to store the optimal solution among solutions with equal neighborhoods. In the case of vertex subset problems we use a similar approach, only now we store partial solutions with distinct d-neighborhoods. This means that a bound on the number of d-equivalence classes of a decomposition is important in order to analyze the running time of algorithms in terms of boolean-width. While it is still an open problem whether there is a tight upper bound in terms of boolean-width on the number of d-equivalence classes on a decomposition [8], there are a number of (trivial) upper bounds known. We present an overview of the most relevant bounds together with a brief explanation as to why they are valid, for which we make use of a *twin class partition* of a graph.

Definition 7.6 (Twin class partition)**.** Let $G = (V(G), E(G))$ be a graph and let $A \subseteq V(G)$. The *twin class partition* of A is a partition of A such that $\forall x, y \in A$, x and y are in the same partition class if and only if $N_G(x) \cap \overline{A} = N_G(y) \cap \overline{A}$. The number of partition classes of A is denoted by $ntc(A)$.

For all bounds listed below, let $G = (V(G), E(G))$ be a graph of size n and let d be a non-negative integer. Let $(A, \overline{A})$ be a cut of a decomposition (T, δ) of G, and let $k = \text{bool-dim}(A)$.

Proposition 7.7. $nec(\equiv_A^d) \leq 2^n$.

Proof. This is a trivial bound, since for every equivalence class there is a set R that is the representative of all sets $X \subseteq V(G)$ with $R \equiv_A^d X$. The number of subsets of A is $2^{|A|} \leq 2^n$, thus there are less than 2^n possible distinct representatives. □

Proposition 7.8. *[1, Lemma 1]* $nec(\equiv_A^d) \leq 2^{d \cdot k^2}$.

Proposition 7.9. *[24, Lemma 5.2.2]* $nec(\equiv_A^d) \leq (d+1)^{\min(ntc(A), ntc(\overline{A}))}$.

Proposition 7.10. $nec(\equiv_A^d) \leq ntc(A)^{d \cdot k}$.

Proof. In order to prove this bound, we make use of a graph parameter called *maximum induced matching-width* [2]. We denote the maximum matching-width of A by $mim(A)$. It has been shown that for a graph G and for any subset $A \subseteq V(G)$, it holds that $mim(A) \leq$ bool-dim(A)[24, Theorem 4.2.10]. From [24, Lemma 5.2.3], we know that $nec(\equiv_A^d) \leq ntc(A)^{d \cdot mim(A)}$, thus we can conclude that $nec(\equiv_A^d) \leq ntc(A)^{d \cdot k}$. □

7.3 Solving (σ, ρ) problems

In this section we show the algorithm by Bui-Xuan et al. [8] for solving (σ, ρ) problems using a boolean decomposition (T, δ) of a graph G. While we refer the reader to the original paper for an in-depth explanation of the correctness of the algorithm, we do provide an overview of the main pseudo-code and present details on our implementation. The main idea is to use a dynamic programming algorithm to store partial solutions to the (σ, ρ) problem that we are solving, similar to the way we solved the maximum independent set and minimum dominating set problems in Chapter 3. For (σ, ρ) problems we make use the following definition given by Bui-Xuan et al.

Definition 7.11 (σ, ρ-domination)**.** Let $G = (V(G), E(G))$ be a graph, let $A \subseteq V(G)$ and $\mu \subseteq \mathbb{N}$. A subset $X \subseteq V(G)$ μ-dominates A if $\forall v \in A : |N_G(v) \cap X| \in \mu$. For $X \subseteq A$, $Y \subseteq \overline{A}$ the pair (X, Y) σ, ρ-dominates A if $(X \cup Y)$ σ-dominates X and $(X \cup Y)$ ρ-dominates $A \setminus X$.

In a bottom-up traversal of the nodes of T, we construct solutions for a node w by iterating over all values saved for the child nodes a and b of w. Using their representatives, we construct new representatives for the union of these sets, while keeping track of the size of the optimal set that fulfills the constraint of σ, ρ-dominating V_w. The optimal size depends on if we are finding a set of maximum or minimum cardinality, which we indicate through the function *opt*. The sizes are saved in a table which is defined as follows.

$$Tab_w[R_x][R_y] = \begin{cases} opt_{S \subseteq V_w}\{|S| : S \equiv_{V_w}^d R_x \text{ and } (S, R_y)\ \sigma, \rho\text{-dominates } V_w\}, \\ -\infty \text{ if no such set } S \text{ exists and } opt = max, \\ +\infty \text{ if no such set } S \text{ exists and } opt = min. \end{cases}$$

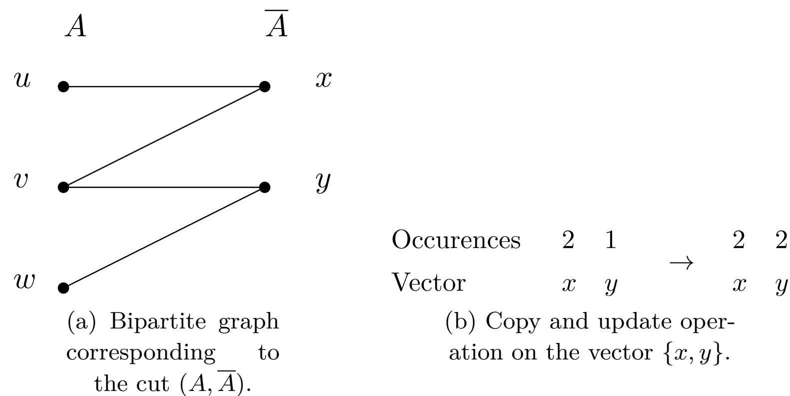
The goal is to fill this table with the correct values, after which the solution to our problem will be saved at $Tab_{V(G)}[\emptyset][\emptyset]$. The first step is to compute a structure that,

given a set A, returns a bidirectional map of all representatives together with their corresponding d-neighborhoods across the cut $(A, \overline{A})$. For simplicity we make use of two lists: one of all representatives LR_A^d, and one of all corresponding d-neighborhoods LNR_A^d and create pointers between the entries of the elements of these lists. We compute the representatives by applying Algorithm 10 to V_w and $\overline{V_w}$, for all nodes $w \in V(T)$.

Algorithm 10 Algorithm by Bui-Xuan et al. [8] for computing a list of representatives of a cut $(A, \overline{A})$ and their corresponding d-neighborhoods.

```
1: function ComputeRepresentatives(Graph G, Subset A ⊆ V(G), Integer d)
2:     LR_A^d, LNR_A^d ← {∅}
3:     LastLevel ← {∅}
4:     while LastLevel ≠ ∅ do
5:         NextLevel ← ∅
6:         for all R ∈ LastLevel do
7:             for all v ∈ A do
8:                 R' ← R ∪ {v}
9:                 N' ← N_A^d(R')
10:                if R' ≢_A^d R and N' ∉ LNR_A then
11:                    NextLevel ← NextLevel ∪ R'
12:                    LR_A^d ← LR_A ∪ R'
13:                    LNR_A^d ← LNR_A ∪ N'
14:                    Add pointers between R' and N'
15:        LastLevel ← NextLevel
16:    return LR_A^d and LNR_A^d
```

Algorithm 10 runs in $O(n^2 \cdot nec(\equiv_A^d))$, if a hash map is used in order to store the representatives. This gives us amortized $O(n)$ time to look up if a d-neighborhood is already contained in LNR in contrast to the $O(\log(nec(\equiv_A^d)) \cdot n)$ time needed for the binary search approach used by Bui-Xuan et al. Note that this algorithm is very similar to Algorithm 1 of Chapter 3.



(a) Bipartite graph corresponding to the cut $(A, \overline{A})$.

(b) Copy and update operation on the vector $\{x, y\}$.

Figure 7.1: Construction of d-neighborhoods using Algorithm 10. d-Neighborhoods can be represented as a vector of length $|\overline{A}|$ over $\{0, 1, \ldots, d\}$, which is displayed on the right. Assume that we have $R = \{u, v\}$ and $N_A^2(R) = \{x, x, y\}$. If we construct $R' = R \cup \{w\}$, then we only need to increase the values of $N_G(w) \cap \overline{A}$ by 1, as long as it does not exceed d. Thus, in this case we end up with $N_A^2(R) = \{x, x, y, y\}$. Note that even if $\{w, x\}$ would be an edge, $N_A^2(R)$ would still be equal to $\{x, x, y, y\}$, since the number of occurrences is capped at $d = 2$

.

The construction of $N' = N^d_A(R')$ can be done in $O(n)$ by copying the d-neighborhood vector of R and updating the entries for all vertices in $N_G(v) \cap \overline{V_w}$. We refer to Figure 7.1 for an example of this approach of copying and updating the vector. In our implementation we perform the check of $R' \not\equiv^d_A R$ by checking if $N^d_A(R') = N^d_A(R)$. Note, however, that this check can lead to invalid values. If $v \in R$, then $N^d_A(R \cup \{v\})$ is never equal to $N^d_A(R)$ after applying a copy and update operation. A simple solution to this problem is to iterate only over the vertices in $A \setminus R$ instead of A.

For initialization purposes we start by setting the table values for the leaf nodes of the decomposition. For a leaf node l, we know that V_l only consists of one single vertex, by definition of boolean decompositions. Therefore we also know that there are only two possible representatives, i.e., $\emptyset$ and V_l, and at most two d-neighborhoods, namely $\emptyset$ and $N_G(V_l)$. Bui-Xuan et al. propose to brute-force set the values for every representative R with respect to $\equiv^d_{V(G)\setminus\{v\}}$ through the following methods:

- If $|N_G(v) \cap R| \in \sigma$, then $Tab_l[\{v\}][R] = 1$.
- If $|N_G(v) \cap R| \in \rho$, then $Tab_l[\emptyset][R] = 0$.

However, similar to the leaf cases in Chapter 3, we need an additional check for vertices of degree zero. This check is omitted by Bui-Xuan et al. If a vertex v has degree zero, but $|N_G(v) \cap R| \in \sigma$, which in this case comes down to $0 \in \sigma$, then we should set $Tab_l[\emptyset][\emptyset] = 1$, because $(V_l, \emptyset)$ σ, ρ-dominates V_l, but $\emptyset \equiv^d_{V_l} V_l$.

In Algorithm 11 we present the pseudo-code that shows how to fill all entries of Tab for a node w, with children a and b. The lists of representatives and corresponding d-neighborhoods for every set V_w and $\overline{V_w}$ for every node $w \in V(T)$ are computed beforehand. Note that this algorithm is basically a generic approach of Algorithm 3 for solving the minimum dominating set problem, as seen in Chapter 3.

Algorithm 11 Algorithm by Bui-Xuan et al. [8] for computing the size of a (σ, ρ) set of minimum or maximum cardinality. This algorithm shows how all table entries for a node w of the decomposition tree are filled.

1: **procedure** COMBINE
2: **for all** $R_w \in LR^d_{V_w}$ **do**
3: **for all** $R_{\overline{w}} \in LR^d_{\overline{V_w}}$ **do**
4: $Tab_w[R_w][R_{\overline{w}}] \leftarrow \begin{cases} \infty & \text{if } opt = \min, \\ -\infty & \text{if } opt = \max. \end{cases}$
5: **for all** $R_a \in LR^d_{V_a}$ **do**
6: **for all** $R_b \in LR^d_{V_b}$ **do**
7: **for all** $R_{\overline{w}} \in LR^d_{\overline{V_w}}$ **do**
8: $R_{\overline{a}} \leftarrow rep^d_{\overline{V_a}}(R_b \cup R_{\overline{w}})$
9: $R_{\overline{b}} \leftarrow rep^d_{\overline{V_b}}(R_a \cup R_{\overline{w}})$
10: $R_w \leftarrow rep^d_{V_w}(R_a \cup R_b)$
11: $Tab_w[R_w][R_{\overline{w}}] \leftarrow opt(Tab_w[R_w][R_{\overline{w}}], Tab_a[R_a][R_{\overline{a}}] + Tab_b[R_b][R_{\overline{b}}])$

Theorem 7.12. *Given a graph $G = (V(G), E(G))$ and a decomposition (T, δ) we can solve any (σ, ρ) vertex subset problem on G in $O(n^3 \cdot nec_d(T, \delta)^3)$ time, with $d = d(\sigma, \rho)$.*

Proof. We need to compute all entries for the table for all nodes w of T by applying Algorithm 11 a total of $O(n)$ times. The for-loops of Algorithm 11 each take $O(nec(T, \delta))$ at most, thus $O(nec(T, \delta)^3)$ in total. For every triplet the construction of a new representative can be done in $O(n^2)$. For instance, if we need to compute $R_{\overline{a}}$, then we start by constructing $N^d_{\overline{V_a}}(R_b \cup R_{\overline{w}})$ by iterating over all vertices in $\overline{V_a}$ and counting the number of neighbors each of these vertices has in $R_b \cup R_{\overline{w}}$. We can use $N^d_{\overline{V_a}}(R_b \cup R_{\overline{w}})$ to find the corresponding representative in $LR_{\overline{V_a}}$ in $O(n)$ amortized time, giving us a total running time of $O(n^3 \cdot nec(T, \delta)^3)$, improving upon the $O(n^4 \cdot nec(T, \delta)^3)$ time given by Bui-Xuan et al. □

7.4 Experiments

We have used linear decompositions in order to compute the size of the maximum induced matching (MIM) of a selection of graphs for which the results are presented in Table 7.2. These are the same graphs that were used in Chapter 5. The decomposition that we work on is obtained through running the IUN heuristic over all possible starting vertices. The maximum induced matching problem is defined as finding the largest $(\{1\}, \mathbb{N})$ set, with $d(\{1\}, \mathbb{N}) = 2$. The choice for the MIM problem is arbitrary, as any vertex subset problem with $d = 2$ will have the same number of equivalence classes and therefore all require the same running time when computing a solution. We provide the computed value of $nec_d(T, \delta)$, together with the most relevant theoretical upper bounds presented in Section 7.2. Note that we take the logarithm of each value, since we find this value easier to interpret and compare to other graph parameters. We let $UB_1 = d \cdot \text{boolw}^2$, $UB_2 = \log_2((d + 1)^{\min ntc})$ and $UB_3 = \log_2(ntc^{d \cdot \text{boolw}})$, with $ntc = \max_{w \in V(T)} ntc(V_w)$ and $\min ntc = \max_{w \in V(T)} \min(ntc(V_w), ntc(\overline{V_w}))$.

The column MIM displays the size of the MIM of the graph, while the time column indicates the time in seconds needed to compute the solution. Missing values for nec and MIM are caused by a lack of internal memory, which can be explained from the fact that the space requirement for the algorithm used to compute the MIM is $O^*(nec_d(T, \delta)^2)$.

An interesting observation that we can make, for instance by looking at the graphs ZEROIN.I.2 and BOBLO, is that a lower boolean-width does not automatically imply a lower number of equivalence classes. This leads us to the question of whether this can happen for decompositions of the same graph - or even decompositions of the same width. In Table 7.3 we present our findings on the graph BARLEY. We can see that a lower boolean-width is not a guarantee for a lower number of equivalence classes. What may be even more unfortunate is that for this instance a boolean decomposition of width 4.81 gave both the best and the worst decomposition in terms of the number of equivalence classes. This means that if we use a heuristic that tries to minimize the boolean-width we might still end up with a 'bad' decomposition for vertex subset problems, even though there might be a much better decomposition of the same boolean-width.

Graph	boolw	$\log_2(nec)$	UB_1	UB_2	UB_3	MIM	Time (s)
alarm	3.00	4.32	18.00	7.92	13.93	18	< 1
barley	4.58	7.00	42.04	12.68	27.51	22	3
pigs-pp	6.64	10.31	88.28	19.02	49.17	22	1147
BN_100	10.86	-	235.93	36.45	105.53	-	-
eil76	8.33	12.63	138.81	22.19	65.10	-	-
david	5.86	9.37	68.63	22.19	44.61	34	919
1jhg	8.41	13.53	141.58	41.21	81.75	-	-
1aac	12.33	-	304.08	72.91	141.25	-	-
celar04-pp	7.27	11.15	105.61	28.53	65.74	-	-
1a62	11.14	-	248.09	60.23	121.61	-	-
1bkb-pp	9.53	-	181.47	52.30	98.49	-	-
1dd3	9.90	-	196.11	52.30	103.17	-	-
miles1500	5.29	9.30	55.87	34.87	49.69	8	4038
miles250	4.58	7.24	42.04	15.85	31.72	52	37
celar10-pp	6.91	10.34	95.41	25.36	59.70	50	10179
anna	7.94	11.94	125.98	33.28	75.48	-	-
pr152	8.29	12.76	137.45	22.19	63.13	-	-
munin2-pp	7.61	11.82	115.97	19.02	54.60	-	-
mulsol.i.5	3.58	6.11	25.70	14.26	24.80	46	22
zeroin.i.2	3.81	6.58	28.99	20.60	28.18	30	59
boblo	4.00	6.17	32.00	9.51	20.68	148	41
fpsol2.i-pp	4.81	8.07	46.22	22.19	36.61	46	934
munin4-wpp	7.61	12.13	115.97	19.02	57.98	-	-

Table 7.2: Results of using the algorithm by Bui-Xuan et al. [8] for solving (σ, ρ) problems on graphs, using decompositions obtained through the IUN heuristic using all starting vertices.

$\text{boolw}(T, \delta)$	$\log_2(nec(T, \delta))$
4.58	7
4.64	7.51
4.7	7.15
4.7	7.39
4.81	6.75
4.81	12.13
4.91	7.92
4.91	8.44
4.91	10.82
5	7.57
5	7.83
5	8.75
5.09	8.86
5.17	7.67
5.32	8.99

Table 7.3: Boolean-width and number of equivalence classes for a number of decompositions of the graph BARLEY.

7.5 Heuristics for minimizing the number of equivalence classes

We have seen that a low boolean-width of a decomposition does not automatically result in a low number of equivalence classes for other values than $d = 1$. Therefore it may be worthwhile to investigate heuristics that focus on minimizing the number of equivalence classes. A starting point that we investigated was to modify the IUN heuristic of Chapter 5, such that it will keep track of the number of equivalence classes instead of the unions of neighborhoods. Recall that the IUN heuristic selects a vertex out of a set of not-yet processed vertices (called *Right*), and adds the one that minimizes the boolean dimension to the set of processed vertices (called *Left*). In order to determine which vertex v we select we generate $\mathcal{UN}(Left \cup \{v\})$ out of $\mathcal{UN}(Left)$. Because the size of the unions of neighborhoods is symmetric across a cut $(A, \overline{A})$ there is no need to compute $|\mathcal{UN}(Right \setminus \{v\})|$; it follows directly from $|\mathcal{UN}(Left \cup \{v\})|$.

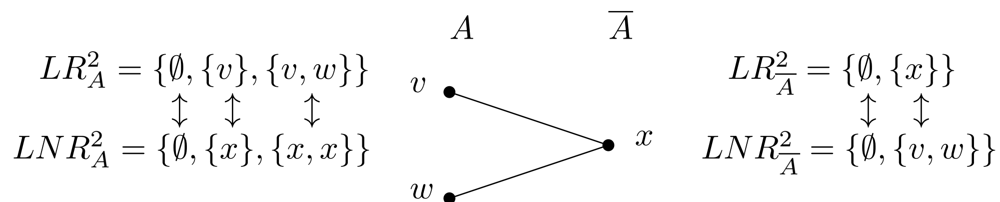


Figure 7.2: The lists LR^d_A and LNR^d_A for a cut $(A, \overline{A})$ for $d = 2$, together with the pointers between the elements of the lists.

Unfortunately, the number of equivalence classes is not symmetric. We refer to Figure 7.2 for a simple illustration of this fact. What follows is that if we would apply the same approach of choosing a vertex that minimizes the number of equivalence classes across a cut, then we need to compute both $nec(\equiv^d_{Left \cup \{v\}})$ and $nec(\equiv^d_{Right \setminus \{v\}})$, for each vertex v. While there is a simple way to obtain $nec(\equiv^d_{Left \cup \{v\}})$ out of all equivalence classes of *Left*, similar to Algorithm 8 of Chapter 5, there is currently no way of obtaining $nec(\equiv^d_{Right \setminus \{v\}})$ out of the equivalence classes of *Right*. Thus we have to construct $LR^d_{Right \setminus \{v\}}$ using Algorithm 10 for each potential cut. Apart from an extra factor $O(n)$, this leads to the heuristic not being output sensitive. In practice this renders the heuristic useless because of the running time.

7.6 Conclusion

The algorithm by Bui-Xuan et al. [8] is a relatively easy to implement algorithm for solving (σ, ρ) vertex subset problems. The space requirement of the algorithm turns out to be the main bottleneck, but we can still solve a number of problems on graphs with real applications. Furthermore, our experimental evaluation shows that the algorithm is much faster, up to several orders of magnitude, compared to theoretical worst case bounds.

An important observation that we made is that if $\text{boolw}(T, \delta) < \text{boolw}(T', \delta')$, then there is no guarantee that $nec_d(T, \delta) < nec_d(T', \delta')$. While in general it holds that

minimizing boolean-width results in a low number of equivalence classes, we think that it can be worthwhile to focus on minimizing $nec_d(T, \delta)$ instead of the boolean-width when solving vertex subset problems. However, the number of equivalence classes is not symmetric, i.e., $nec(\equiv^d_A) \neq nec(\equiv^d_{\overline{A}})$ does not always holds for a cut $(A, \overline{A})$, which makes it harder to develop fast heuristics that focus on minimizing nec_d. This follows from the need of such a heuristic to keep track of both the equivalence classes of A and $\overline{A}$ to make the choice that is locally optimal. We therefore propose additional investigation in getting a more realistic theoretical upper bound in terms of boolean-width, which can be of great importance to find a more clear correlation between boolean-width and the number of equivalence classes. This could possibly lead to more insight into why some decompositions have a low number of equivalence classes compared to decompositions of the same width.

Chapter 8

Conclusion

In this thesis we showed numerous algorithms that make use of boolean decompositions and experimentally verified the applicability of these algorithms on a number of graphs. We gave motivation for using linear boolean decompositions over general boolean decompositions. The running time of algorithms parameterized by boolean-width is theoretically lower when using linear decompositions. Additionally, we can compute the width of linear decompositions faster than for general ones.

In order to obtain decompositions that can be used in practical settings, we introduced a new heuristic for generating linear boolean decompositions. This heuristic is several orders of magnitude faster than the previous best heuristic. Furthermore, it returned decompositions of lower width than any previously known heuristic.

Future research topic 1 - A good approximation algorithm instead of calculating the exact boolean dimension of a cut can be a good alternative in heuristics that greedily minimize the boolean-width.

We found reduction rules in order to reduce the running time needed to generate linear boolean decompositions. However, these reduction rules seem to describe degenerate graph classes that will not occur often in practical settings, meaning that the benefit of reducing vertices will be very marginal. We believe that utilizing the properties of linear boolean decompositions instead of focusing on new reduction rules for them is a more worthwhile area of research.

Future research topic 2 - The use of reduction rules for general boolean decompositions in combination with heuristics for generating linear boolean decompositions can lead to improvements in known upper bounds on the boolean-width of a lot of practical graphs. This leads to the question if there are more reduction rules for general boolean decompositions.

We investigated the theoretical upper bound for solving (σ, ρ) vertex subset problems using an algorithm parameterized by boolean-width. We implemented this algorithm to solve the minimum induced matching problem on a select number of graphs. This led to the conclusion that the space requirement of the algorithm is often a bottleneck when solving (σ, ρ) problems.

Future research topic 3 - A better theoretical upper bound on the number of equivalence classes in terms of boolean-width could lead to better understanding of the relation between the two.

Future research topic 4 - A heuristic that focuses on minimizing the number of equivalence classes for a decomposition rather than minimizing the boolean-width could most likely lead to more (σ, ρ) problems being solvable on practical graphs.

Future research topic 5 - Since solving (σ, ρ) problems is FPT using other graph parameters, it would be interesting to see if these other parameters lead to practical algorithms and how the execution time compares to the algorithm parameterized by boolean-width.

Bibliography

[1] I. Adler, B.-M. Bui-Xuan, Y. Rabinovich, G. Renault, J. A. Telle, and M. Vatshelle. On the boolean-width of a graph: Structure and applications. In *Graph Theoretic Concepts in Computer Science*, volume 6410 of *Lecture Notes in Computer Science*, pages 159–170. Springer Berlin Heidelberg, 2010.

[2] R. Belmonte and M. Vatshelle. Graph classes with structured neighborhoods and algorithmic applications. *Theoretical Computer Science*, 511(0):54 – 65, 2013.

[3] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–23, 1993.

[4] H. L. Bodlaender, A. M. C. A. Koster, F. van den Eijkhof, and L. C. van der Gaag. Pre-processing for triangulation of probabilistic networks. In *Proceedings of the 17th Conference on Uncertainty in Arti Intelligence*, pages 32–39. Morgan Kaufmann, 2001.

[5] H. L. Bodlaender and A. M.C.A. Koster. Safe separators for treewidth. *Discrete Mathematics*, 306(3):337 – 350, 2006.

[6] H. L. Bodlaender and B. van Antwerpen-de Fluiter. Reduction algorithms for graphs of small treewidth. *Information and Computation*, 167(2):86 – 119, 2001.

[7] B.-M. Bui-Xuan, J. A. Telle, and M. Vatshelle. Boolean-width of graphs. In *Parameterized and Exact Computation*, volume 5917 of *Lecture Notes in Computer Science*, pages 61–74. Springer Berlin Heidelberg, 2009.

[8] B.-M. Bui-Xuan, J. A. Telle, and M. Vatshelle. Fast dynamic programming for locally checkable vertex subset and vertex partitioning problems. *Theoretical Computer Science*, 511:66–76, 2013.

[9] R. Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, August 2005.

[10] S. Gaspers, D. Kratsch, and M. Liedloff. On independent sets and bicliques in graphs. In *Graph-Theoretic Concepts in Computer Science*, volume 5344 of *Lecture Notes in Computer Science*, pages 171–182. Springer Berlin Heidelberg, 2008.

[11] M. U. Gerber and D. Kobler. Algorithms for vertex-partitioning problems on graphs with fixed clique-width. *Theoretical Computer Science*, 299(1-3):719–734, April 2003.

[12] K. H. Kim. *Boolean matrix theory and its applications (Monographs and textbooks in pure and applied mathematics).* Marcel Dekker, 1982.

[13] F. Manne and S. Sharmin. Efficient counting of maximal independent sets in sparse graphs. In *Experimental Algorithms*, volume 7933 of *Lecture Notes in Computer Science*, pages 103–114. Springer Berlin Heidelberg, 2013.

[14] J. W. Moon and L. Moser. On cliques in graphs. *Israel Journal of Mathematics*, 3(1):23–28, 1965.

[15] Y. Rabinovich, J. A. Telle, and M. Vatshelle. Upper bounds on boolean-width with applications to exact algorithms. In *Parameterized and Exact Computation*, volume 8246 of *Lecture Notes in Computer Science*, pages 308–320. Springer International Publishing, 2013.

[16] N. Robertson and P. D. Seymour. Graph minors. I. Excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39 – 61, 1983.

[17] N. Robertson and P. D. Seymour. Graph minors. X. Obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52(2):153 – 190, 1991.

[18] S. Sharmin. *Practical Aspects of the Graph Parameter Boolean-width.* PhD thesis, University of Bergen, Norway, 2014.

[19] J. A. Telle. Complexity of domination-type problems in graphs. *Nordic Journal of Computing*, 1(1):157–171, 1994.

[20] J. A. Telle and A. Proskurowski. Algorithms for vertex partitioning problems on partial k-trees. *SIAM Journal on Discrete Mathematics*, 10(4):529–550, 1997.

[21] C. B. ten Brinke. Variations on boolean-width. Master's thesis, Utrecht University, The Netherlands, 2015.

[22] Treewidthlib. http://www.staff.science.uu.nl/∼bodla101/treewidthlib/. A benchmark for algorithms for treewidth and related graph problems.

[23] J. M. M. van Rooij, H. L. Bodlaender, and P. Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In *Algorithms - ESA 2009*, volume 5757 of *Lecture Notes in Computer Science*, pages 566–577. Springer Berlin Heidelberg, 2009.

[24] M. Vatshelle. *New width parameters of graphs.* PhD thesis, University of Bergen, Norway, 2012.

Appendix A

Linear boolean-width upper bounds on treewidthlib graphs

Table A.1 contains linear boolean-width upper bounds that are obtained through using the IUN heuristic on all possible starting vertices and *candidates* = *Right* on a large selection of graphs from treewidthlib [22]. The *tw* column gives a known upper bound on the treewidth, while the *bw* column gives an upper bound on the boolean-width, of which the values are taken from [18]. Cursive graph names marked with an asterisk indicate the graphs for which, in theory, the linear boolean decomposition will give a higher bound on the running time than the boolean decomposition, i.e., graphs for which $2^{2lbw} > 2^{3bw}$.

Table A.1

Graph	$\|V\|$	Edge Density	*tw*	*bw*	*lbw*	*lbw*/*bw*
celar06-pp-003	4	0.5	2	1	1	1.00
*diabetes-pp-001**	6	0.8	4	1	1.58	1.58
*munin3-pp-001**	7	0.81	5	1	1.58	1.58
*munin3-pp-002**	7	0.81	5	1	1.58	1.58
celar06-pp-000	8	0.43	3	1	1	1.00
diabetes-pp-002	8	0.61	4	2.32	2.32	1.00
mainuk-pp	9	0.78	6	1.58	1.58	1.00
rl5934-pp-001	10	0.44	4	2.81	3.17	1.13
fl3795-pp-001	10	0.44	4	2.81	3	1.07
fl3795-pp-003	10	0.44	4	2.81	3	1.07
fl3795-pp-002	10	0.44	4	2.81	3.17	1.13
pathfinder-pp-001	11	0.58	5	2.58	3.32	1.29
myciel3	11	0.36	5	3	3.46	1.15
pcb3038-pp-001	11	0.4	5	3	2.81	0.94
fl3795-pp-004	11	0.42	4	3	3.46	1.15
pathfinder-pp	12	0.65	6	2.58	2.81	1.09
celar11-pp-002	13	0.59	7	2.81	3.17	1.13
celar04-pp-001-000	15	0.74	9	1.58	2	1.27

Continued on next page

Table A.1 – *Continued from previous page*

Graph	$\lvert V\rvert$	Edge Density	tw	bw	lbw	lbw/bw
weeduk	15	0.47	7	1.58	1.58	1.00
fungiuk	15	0.34	4	2	1.58	0.79
pcb3038-pp-002	15	0.3	5	3	2.81	0.94
mildew-wpp	15	0.3	4	2.58	3.32	1.29
celar04-pp-001	16	0.78	10	1.58	2	1.27
celar06-pp	16	0.84	11	1.58	1.58	1.00
celar10-pp-001	16	0.51	8	3	3.46	1.15
celar09-pp-001	16	0.51	8	3	3.17	1.06
celar08-pp-002	16	0.51	8	3	3.32	1.11
celar07-pp-002	16	0.45	7	3	3.32	1.11
barley-pp-001	16	0.42	7	3.32	3.32	1.00
celar11-pp-004	16	0.36	6	3.17	3.58	1.13
munin2-pp-005	16	0.3	5	3	3.58	1.19
munin2-pp-006	16	0.3	5	3	3.58	1.19
munin2-pp-003	16	0.3	5	3.17	3.7	1.17
munin2-pp-004	16	0.3	5	3.17	3.7	1.17
munin2-pp-007	17	0.35	7	3.46	3.58	1.03
munin2-pp-011	17	0.35	7	3.46	3.58	1.03
munin2-pp-010	17	0.35	7	3.46	3.81	1.10
munin2-pp-008	17	0.35	7	3.46	3.58	1.03
munin2-pp-009	18	0.31	6	3.46	3.81	1.10
munin2-pp-012	18	0.31	6	3.46	3.81	1.10
celar01-pp-002	19	0.65	10	2	2.32	1.16
celar02-pp	19	0.67	10	2	2	1.00
celar05-pp-001	19	0.66	11	2	2.32	1.16
celar11-pp-001	19	0.65	10	2	2.32	1.16
fl3795-pp-005	19	0.22	4	3.32	3.58	1.08
water-pp-001	21	0.45	9	3.81	4.09	1.07
anna-pp	22	0.64	12	3.46	3.81	1.10
water-pp	22	0.42	9	4.17	4.32	1.04
water-wpp	22	0.42	9	4.17	4.32	1.04
munin4-pp-001	23	0.26	8	3.58	4	1.12
munin4-pp-002	23	0.26	8	3.58	4	1.12
myciel4	23	0.28	10	5	5.49	1.10
BN_29	24	0.18	5	2	2.32	1.16
BN_28	24	0.18	5	2	2.32	1.16
queen5_5	25	0.53	18	5.29	5.67	1.07
barley-pp	26	0.24	7	3.7	3.46	0.94
fl3795-pp-006	26	0.16	5	3.81	4.17	1.09
david-pp	29	0.47	13	4.09	4.32	1.06
barley-wpp	29	0.2	7	3.81	3.58	0.94
pcb3038-pp-003	29	0.12	5	4.32	4.75	1.10
celar02-wpp	30	0.33	10	2.81	2.58	0.92
water	32	0.25	9	4.39	4.75	1.08

Continued on next page

Table A.1 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	*tw*	*bw*	*lbw*	*lbw/bw*
BN_16-pp-015	34	0.28	11	3.58	4.39	1.23
celar06-wpp	34	0.28	11	3	3.17	1.06
BN_16-pp-014	34	0.28	11	3.81	4.86	1.28
1bx7-pp	34	0.31	11	4.7	4.39	0.93
mildew	35	0.13	4	3	3.32	1.11
queen6_6	36	0.46	25	7.65	8.08	1.06
alarm	37	0.1	4	2.58	3	1.16
celar03-pp-001	38	0.34	14	5.81	6.11	1.05
*munin4-pp-003**	38	0.16	8	3.58	5.39	1.51
munin4-pp-004	38	0.16	8	4.17	5.39	1.29
celar08-pp-001	39	0.38	16	5.09	5.21	1.02
oesoca	39	0.09	3	2.32	3	1.29
1bx7	41	0.24	11	4.91	4.75	0.97
oesoca42	42	0.08	3	2.32	3.17	1.37
celar07-pp-001	45	0.32	16	5.46	5.86	1.07
celar01-pp-001	47	0.25	15	5.88	6.36	1.08
celar05-pp-002	47	0.25	15	6.07	5.83	0.96
myciel5	47	0.22	19	8.12	6.49	0.80
1ubq-pp	47	0.16	12	5.95	8.79	1.48
pigs-pp-001	47	0.12	9	5.95	7.07	1.19
1brf-pp	48	0.36	22	7.01	7.25	1.03
1rb9	48	0.37	22	6.77	7.17	1.06
celar11-pp-003	48	0.23	15	5.73	4.58	0.80
*mainuk**	48	0.18	7	3.58	6.49	1.81
barley	48	0.11	7	4	3.7	0.93
pigs-pp	48	0.12	9	5.7	6.64	1.16
1brf	49	0.35	22	7.01	7.3	1.04
queen7_7	49	0.4	35	10.36	10.97	1.06
1kth-pp	51	0.33	20	7.06	5.86	0.83
1i07-pp	51	0.28	15	5.55	7.18	1.29
eil51.tsp	51	0.11	9	5.78	5.78	1.00
1igq-pp	52	0.37	23	6.74	7.45	1.11
1kth	52	0.32	20	7.04	6.87	0.98
1g6x	52	0.31	19	6.89	7.21	1.05
1igq	54	0.35	23	6.89	7.61	1.10
zeroin.i.1-pp	54	0.89	46	1.58	1.58	1.00
1e0b-pp	55	0.33	24	7.69	8.32	1.08
munin4-pp-006	55	0.11	8	4.32	5.17	1.20
munin4-pp-005	55	0.11	8	4.39	5.17	1.18
1j75	56	0.36	27	8.51	8.94	1.05
1k61-pp	56	0.37	26	8.02	8.37	1.04
1sem-pp	56	0.37	26	8.09	8.5	1.05
1bbz-pp	56	0.35	25	8.18	8.36	1.02
1bf4-pp	57	0.39	26	7.63	7.79	1.02

Continued on next page

Table A.1 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	tw	bw	lbw	lbw/bw
1cka	57	0.38	27	8.55	8.87	1.04
1sem	57	0.36	26	8.32	8.66	1.04
zeroin.i.2-pp	57	0.69	32	2.81	3.32	1.18
zeroin.i.3-pp	57	0.69	32	3	3.32	1.11
1bbz	57	0.34	25	8.3	8.36	1.01
1oai-pp	57	0.32	22	7.94	8.28	1.04
1jo8	58	0.37	27	8.46	8.73	1.03
1oai	58	0.32	22	7.87	8.15	1.04
celar01-pp-003	58	0.19	15	6.97	6.89	0.99
1g2b-pp	59	0.37	28	8.5	8.99	1.06
1igd-pp	59	0.36	25	7.66	7.9	1.03
1kq1-pp	59	0.35	27	8.63	8.94	1.04
1pwt-pp	59	0.38	29	8.85	9.24	1.04
1i07	59	0.23	15	5.52	5.93	1.07
1k61	60	0.33	26	8.32	8.81	1.06
1kq1	60	0.34	27	8.79	8.89	1.01
1ku3-pp	60	0.33	23	7.46	7.53	1.01
1e0b	60	0.29	24	8.13	8.42	1.04
knights8_8-pp	60	0.09	16	10.77	11.3	1.05
1gut-pp	61	0.33	22	7.19	7.54	1.05
1i2t	61	0.35	27	8.38	9.03	1.08
1igd	61	0.34	25	7.75	7.9	1.02
1pwt	61	0.36	29	8.81	9.27	1.05
1ku3	61	0.32	23	7.53	7.61	1.01
1g2b	62	0.34	28	8.72	9.05	1.04
1fr3-pp	62	0.32	21	7.16	7.29	1.02
celar04-pp-002	62	0.17	16	6.86	7.26	1.06
1bf4	63	0.34	26	7.9	8.09	1.02
1r69	63	0.35	30	9.12	9.51	1.04
munin1-pp-001	63	0.09	11	5.58	6.43	1.15
1gcq-pp	64	0.36	30	8.95	9.38	1.05
queen8_8	64	0.36	45	13.16	14.05	1.07
1a8o	64	0.27	25	9.11	9.3	1.02
knights8_8	64	0.08	16	11.06	11.64	1.05
1fjl	65	0.29	26	7.9	8.49	1.07
1c9o	66	0.34	29	8.75	8.88	1.01
1hg7	66	0.33	29	8.81	9.13	1.04
1ezg	66	0.25	23	8.33	7	0.84
1en2-pp	66	0.21	17	7.46	8.54	1.14
munin1-pp	66	0.09	11	5.58	6.43	1.15
1c4q	67	0.34	31	9.45	9.71	1.03
1fse	67	0.33	27	8.58	8.75	1.02
1kw4	67	0.3	28	9.39	5.73	0.61
1gut	67	0.28	22	7.47	7.36	0.99

Continued on next page

Table A.1 – *Continued from previous page*

Graph	$\lvert V \rvert$	Edge Density	*tw*	*bw*	*lbw*	*lbw/bw*
1fr3	67	0.28	21	7.29	7.47	1.02
1b67-pp	67	0.25	16	6.61	9.61	1.45
1gcq	68	0.33	30	9.36	9.65	1.03
1ail-pp	68	0.28	24	8.11	8.33	1.03
1d3b-pp	68	0.3	25	8.54	5.78	0.68
1b67	68	0.25	16	6.61	8.52	1.29
1c75	69	0.29	30	9.88	8.31	0.84
1ail	69	0.27	24	8.07	9.68	1.20
1d3b	69	0.29	25	8.44	8.53	1.01
1en2	69	0.2	17	7.24	7	0.97
1cc8	70	0.34	32	9.35	9.63	1.03
1dj7-pp	70	0.3	27	8.12	8.22	1.01
1i27-pp	70	0.3	27	8.67	8.82	1.02
1l9l	70	0.29	29	9.26	10	1.08
1ljo-pp	71	0.31	30	8.92	9.02	1.01
1dp7-pp	71	0.3	27	9.21	9.15	0.99
graph03-pp-001	71	0.11	20	12.53	12.24	0.98
1mgq-pp	72	0.31	28	8.98	9.08	1.01
1i27	73	0.28	27	8.78	9.06	1.03
mulsol.i.1-pp	73	0.83	50	2.32	2.58	1.11
1dj7	73	0.28	27	9.66	8.22	0.85
1ldd	74	0.31	32	9.6	9.73	1.01
1ljo	74	0.29	30	8.88	9.06	1.02
1mgq	74	0.3	28	8.91	9.06	1.02
huck	74	0.11	10	2.81	3.32	1.18
1ubq	74	0.08	12	6.61	7.75	1.17
1ig5	75	0.29	33	10.45	10.64	1.02
1dp7	76	0.27	27	9.01	9.3	1.03
celar10-pp-002	76	0.15	16	7.25	6.58	0.91
celar08-pp-003	76	0.15	16	7.41	6.58	0.89
celar09-pp-002	76	0.15	16	7.46	6.58	0.88
1iqz	77	0.29	33	10	10.1	1.01
1qtn-pp	77	0.25	24	8.56	8.33	0.97
*munin3-pp-003**	79	0.09	7	4.17	12.73	3.05
graph03-pp	79	0.1	20	12.99	5.61	0.43
sodoku-elim1	80	0.28	45	9.47	12	1.27
*jean**	80	0.08	9	3.91	6.54	1.67
celar05-pp	80	0.13	15	7.2	4.58	0.64
sodoku	81	0.25	45	9	12.7	1.41
celar03-pp	81	0.13	14	6.19	6.11	0.99
graph03-wpp	84	0.09	20	12.74	12.92	1.01
1fk5	85	0.23	31	10.76	10.1	0.94
1aba	85	0.25	29	10.13	10.81	1.07
graph01-pp-001	85	0.09	24	13.4	13.66	1.02

Continued on next page

Table A.1 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	tw	bw	lbw	lbw/bw
1ctj-pp	86	0.25	33	10.78	11.07	1.03
1ctj	87	0.25	33	10.74	11.04	1.03
1ptf	87	0.3	38	11.21	10.86	0.97
1qtn	87	0.21	24	9.15	8.97	0.98
david	87	0.11	13	5.32	5.86	1.10
graph05-pp-001	87	0.1	24	12.68	13.31	1.05
1awd	89	0.28	38	10.8	11.13	1.03
celar03-wpp	89	0.11	14	6.17	6.49	1.05
celar05-wpp	89	0.11	15	7.52	6.54	0.87
graph01-pp	89	0.08	24	14.62	13.96	0.95
munin1-wpp	90	0.05	11	7.23	7.58	1.05
1jhg-pp	91	0.19	25	8.34	8.41	1.01
graph05-pp	91	0.1	24	13.84	13.49	0.97
celar07-pp	92	0.12	16	6	6	1.00
a280.tsp-pp	92	0.06	14	8.23	7.38	0.90
*kroE100.tsp-pp**	92	0.06	10	6.48	14.84	2.29
1g2r-pp	93	0.26	37	11.87	11.51	0.97
graph01-wpp	93	0.07	24	14.69	11.41	0.78
1czp	94	0.27	38	11.47	11.6	1.01
1g2r	94	0.25	37	12.17	14.19	1.17
graph05-wpp	94	0.09	24	14.38	13.18	0.92
1c5e	95	0.26	36	11.06	10.83	0.98
myciel6	95	0.17	35	13.4	7.86	0.59
homer-pp	95	0.17	31	14.61	13.88	0.95
kroA100.tsp-pp	95	0.06	10	7.61	6.58	0.86
celar11-pp	96	0.1	15	6.64	5.98	0.90
munin3-pp	96	0.07	7	4.32	5.86	1.36
celar07-wpp	97	0.01	16	6	7.17	1.20
*kroC100.tsp-pp**	97	0.06	10	6.94	11.97	1.72
1plc	98	0.25	35	11.28	11.1	0.98
1lkk-pp	99	0.24	34	11	10.84	0.99
1d4t-pp	99	0.23	35	11.88	6.58	0.55
celar11-wpp	99	0.1	15	7.17	4.91	0.68
1i0v	100	0.24	41	12.21	12.47	1.02
celar02	100	0.06	10	3.32	4.91	1.48
*celar06**	100	0.07	11	3.81	14.85	3.90
graph05	100	0.08	24	13.7	13.36	0.98
graph01	100	0.07	24	14.61	14.21	0.97
graph03	100	0.07	20	13.29	8.41	0.63
1erv	101	0.25	41	12.26	12.44	1.01
1jhg	101	0.17	25	8.87	11.97	1.35
1iib-pp	102	0.27	40	11.98	11.76	0.98
1d4t	102	0.22	35	12.87	10.31	0.80
1iib	103	0.26	40	12.62	11.79	0.93

Continued on next page

Table A.1 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	*tw*	*bw*	*lbw*	*lbw/bw*
1b0n	103	0.19	32	10.81	11.17	1.03
1lkk	103	0.22	34	11.89	13.56	1.14
1aac	104	0.25	41	12.29	12.33	1.00
1bkf-pp	105	0.23	36	11.1	11.4	1.03
1bkf	106	0.23	36	11.69	11.44	0.98
1bkr	107	0.24	44	14.4	13.75	0.95
1rro	107	0.23	43	15.36	3.58	0.23
1f9m	109	0.23	45	14.27	13.56	0.95
*pathfinder**	109	0.04	6	3.32	10.83	3.26
celar04-pp	110	0.09	16	7.29	7.27	1.00
1fs1	114	0.21	34	13.79	7.36	0.53
celar04-wpp	116	0.07	16	7.95	11.1	1.40
1gef-pp	117	0.22	43	12.93	13.35	1.03
1gef	119	0.21	43	13.6	13.35	0.98
mulsol.i.5-pp	119	0.36	31	3	3	1.00
1a62-pp	120	0.21	37	14.7	11.14	0.76
1a62	122	0.21	37	13.62	9.68	0.71
1dd3-pp	124	0.17	31	14.6	9.25	0.63
ch130.tsp-pp	125	0.05	12	8.67	9.53	1.10
1bkb-pp	127	0.18	30	15.55	9.9	0.64
miles1500	128	0.64	77	4.86	5.29	1.09
1dd3	128	0.17	31	11.68	4.58	0.39
miles500	128	0.14	22	9.42	7.04	0.75
*miles250**	128	0.05	9	4.95	9.61	1.94
1bkb	131	0.17	30	14.53	6.91	0.48
celar10-pp	133	0.07	16	9.08	7.7	0.85
anna	138	0.04	12	6.67	7.25	1.09
celar09-wpp	142	0.06	16	8.49	7	0.82
celar01-pp	157	0.07	15	7.39	7	0.95
celar01-wpp	158	0.06	15	7.09	7.61	1.07
munin2-pp	167	0.03	7	5.49	6.91	1.26
mulsol.i.3	184	0.23	32	4.95	3.58	0.72
mulsol.i.4	185	0.23	32	4.81	3.58	0.74
mulsol.i.5	186	0.23	31	4.95	3.58	0.72
mulsol.i.2	188	0.22	32	4.81	3.58	0.74
celar08-wpp	190	0.05	16	9.64	11.48	1.19
mulsol.i.1	197	0.2	50	4	4.17	1.04
zeroin.i.3	206	0.17	32	5.39	3.81	0.71
zeroin.i.1	211	0.19	50	3.7	3.32	0.90
zeroin.i.2	211	0.16	32	5.39	3.81	0.71
fpsol2.i.1-pp	233	0.4	66	4.91	4.81	0.98

Appendix B

Implemented algorithms

We present an overview of the most important algorithms that were implemented as part of this thesis project.

- Algorithm 4 of Chapter 5 for computing representatives using linear boolean decompositions.
- Algorithms for computing optimal boolean decompositions using exact algorithms, explained in Chapter 4 and Appendix C.
- Algorithm for solving the maximum independent set problem by Sharmin [18, Chapter 9], using linear decompositions.
- Algorithms for solving the minimum dominating set problem, counting the number of independent sets and counting the number of dominating sets in a graph by Bui-Xuan et al. [7], modified to work only on linear decompositions.
- All heuristics described in Chapter 5, with additional scoring functions such as the *least uncommon neighbors* [18] and *max cardinality search* (Appendix C).
- Multiple algorithms for counting the number of maximal independent sets in a graph, such as the CCM_{IS} algorithm by Manne and Sharmin [13].
- Algorithm 11 of Chapter 7 for solving (σ, ρ) vertex subset problems by Bui-Xuan et al. [8].

Appendix C

(Paper) Practical algorithms for linear boolean-width

Practical Algorithms for Linear Boolean-width*

Chiel B. ten Brinke[1], Frank J. P. van Houten[1], and Hans L. Bodlaender[1]

1 **Department of Computer Science, Utrecht University**
PO Box 80.089, 3508 TB Utrecht, The Netherlands
`CtenBrinke@gmail.com, Frankv@nhouten.com, H.L.Bodlaender@uu.nl`

Abstract

In this paper, we give a number of new exact algorithms and heuristics to compute linear boolean decompositions, and experimentally evaluate these algorithms. The experimental evaluation shows that significant improvements can be made with respect to running time without increasing the width of the generated decompositions. We also evaluated dynamic programming algorithms on linear boolean decompositions for several vertex subset problems. This evaluation shows that such algorithms are often much faster (up to several orders of magnitude) compared to theoretical worst case bounds.

1998 ACM Subject Classification G.2.2 [Discrete Mathematics]: Graph Theory — Graph algorithms; F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems — Computations on discrete structures

Keywords and phrases graph decomposition, boolean-width, heuristics, exact algorithms, vertex subset problems

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Boolean-width is a recently introduced graph parameter [2]. Similarly to treewidth and other parameters, it measures some structural complexity of a graph. Many NP-hard problems on graphs become easy if some graph parameter is small. We need a derived structure which captures the necessary information of a graph in order to exploit such a small parameter. In the case of boolean-width, this is a binary partition tree, referred to as the decomposition tree. However, computing an optimal decomposition tree is usually a hard problem in itself, because of the required exponential running time. A common approach to bypass this problem is to use heuristics to compute decompositions with a low boolean-width.

Algorithms for computing boolean decompositions have been studied before in [17, 10, 12, 1, 7], but in this paper we study the specific case of linear boolean decompositions. Linear decompositions are easier to compute and the theoretical running time of algorithms for solving practical problems is lower on linear decompositions than on tree shaped ones. For instance, vertex subset problems can be solved in $O^*(nec_d(T,\delta)^3)$ due to a dynamic programming algorithm by Bui-Xuan et al. [3], but this can be improved to $O^*(nec_d(T,\delta)^2)$ for linear decompositions. Here, $nec_d(T,\delta)$ is the number of equivalence classes, i.e., the maximum size of the dynamic programming table.

* The research of the third author was partially funded by the Networks programme, funded by the Dutch Ministry of Education, Culture and Science through the Netherlands Organisation for Scientific Research.

© Chiel B. ten Brinke, Frank J. P. van Houten and Hans L. Bodlaender;
licensed under Creative Commons License CC-BY
Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We first give an exact algorithm for computing optimal linear boolean decompositions, improving upon existing algorithms, and subsequently investigate several new heuristics through experiments, improving upon the work by Sharmin [12, Chapter 8]. We then study the practical relevance of these algorithms in a set of experiments by solving an instance of a vertex subset problem, investigating the number of equivalence classes $nec_d(T, \delta)$ compared to the theoretical worst case bounds.

2 Preliminaries

A *graph* $G = (V, E)$ of size n is a pair consisting of a set of n *vertices* V and a set of *edges* E. The *neighborhood* of a vertex $v \in V$ is denoted by $N(v)$. For a subset $A \subseteq V$ we denote the neighborhood by $N(A) = \bigcup_{v \in A} N(v)$. In this paper we only consider simple, undirected graphs and assume we are given a total ordering on the vertices of a graph G. For a subset $A \subseteq V$ we denote the *complement* by $\overline{A} = V \setminus A$. A partition $(A, \overline{A})$ of V is called a *cut* of the graph. Each cut $(A, \overline{A})$ of G induces a bipartite subgraph $G[A, \overline{A}]$.

The *neighborhood across a cut* $(A, \overline{A})$ for a subset $X \subseteq A$ is defined as $N(X) \cap \overline{A}$.

▶ **Definition 1** (Unions of neighborhoods). Let $G = (V, E)$ be a graph and $A \subseteq V$. We define the set of *unions of neighborhoods across a cut* $(A, \overline{A})$ as

$$\mathcal{UN}(A) = \left\{ N(X) \cap \overline{A} \,\middle|\, X \subseteq A \right\}.$$

The number of unions of neighborhoods $\#\mathcal{UN}$ is symmetric for a cut $(A, \overline{A})$, i.e., $\#\mathcal{UN}(A) = \#\mathcal{UN}(\overline{A})$ [8, Theorem 1.2.3]. Furthermore, for any cut $(A, \overline{A})$ of a graph G it holds that $\#\mathcal{UN}(A) = \#\mathcal{MIS}(G[A, \overline{A}])$, where $\#\mathcal{MIS}(G)$ is the number of maximal independent sets in G [17, Theorem 3.5.5].

▶ **Definition 2** (Decomposition tree). A *decomposition tree* of a graph $G = (V, E)$ is a pair (T, δ), where T is a full binary tree and δ is a bijection between the nodes of T and subsets of vertices of V. For the root node r of T it holds that $\delta(r) = V$. Furthermore, if nodes a and b are children of a node w, then $(\delta(a), \delta(b))$ is a partition of $\delta(w)$. For a decomposition (T, δ) let V_w denote the vertices contained in a node $w \in T$, i.e., $V_w = \delta(w)$.

In this paper we consider a special type of decompositions, namely *linear decompositions*.

▶ **Definition 3** (Linear decomposition). A *linear decomposition*, or *caterpillar decomposition*, is a decomposition tree (T, δ) where T is a full binary tree and for which each internal node of T has at least one leaf as a child. We can define such a linear decomposition through a linear ordering $\pi = \pi_1, \ldots, \pi_n$ of the vertices of G by letting δ map the i-th leaf of T to π_i.

▶ **Definition 4** (Boolean-width). Let $G = (V, E)$ be a graph and $A \subseteq V$. The *boolean dimension of* A is a function bool-dim : $2^V \to \mathbb{R}$.

$$\text{bool-dim}(A) = \log_2 \#\mathcal{UN}(A).$$

Let (T, δ) be a decomposition of a graph G. We define the *boolean-width* of (T, δ) as the maximum boolean dimension over all cuts induced by nodes of (T, δ).

$$\text{boolw}(T, \delta) = \max_{w \in T} \text{bool-dim}(\delta(w))$$

The boolean-width of G is defined as the minimum boolean-width over all possible full decompositions of G, while the *linear boolean-width* of a graph $G = (V(G), E(G))$ of size n is defined as the the minimum boolean-width over all linear decompositions of G.

$$\text{boolw}(G) = \min_{(T,\delta)\ of\ G} \text{boolw}(T, \delta)$$

$$\text{lboolw}(G) = \min_{linear\ (T,\delta)\ of\ G} \text{boolw}(T,\delta)$$

It is known that for any graph G it holds that $\text{boolw}(G) \leq treewidth(G)+1$ [17, Theorem 4.2.8]. The linear variant of treewidth is called *pathwidth* [11], or pw for short.

▶ **Theorem 5.** *For any graph G it holds that* $\text{lboolw}(G) \leq \text{pw}(G) + 1$.

Proof. We give a method of construction that gives us a linear boolean decomposition of a graph G from a path decomposition of G. Recall that a linear boolean decomposition can be defined through a linear ordering $\pi = \pi_1, \ldots, \pi_n$ of V. The idea is that given a path decomposition $X_1, \ldots, X_n$ we select vertices one by one from a subset X_i and append them to the linear ordering π, after which we move on to X_{i+1}. For shorthand notation we denote $\chi_i = \bigcup_{j=1}^{i} X_i$.

Let $S_i = \{u \,|\, u \in \chi_i : N_G(u) \cap \overline{\chi_i} \neq \emptyset\}$. For each $u \in S_i$, it holds that $\exists j > i\ \exists w \in X_j$ for which $\{u, w\} \in E(G)$. By definition of a path decomposition, we know that there is a subset X_j with $u, w \in X_j$, and since all subsets containing a certain vertex are subsequent in the path decomposition, it follows that $u \in X_i$ and $u \in X_{i+1}$, implying that $S_i \subseteq X_i$ and $S_i \subseteq X_{i+1}$. By definition, the unions of neighborhoods of χ_i can only consist of neighborhoods of subsets of S_i. It follows that $|\mathcal{UN}(\chi_i)| = 2^{\text{bool-dim}(\chi_i)} \leq 2^{|S_i|} \leq 2^{|X_i|} \leq 2^{\text{pw}(G)+1}$. What remains to be shown is that while appending vertices one by one from a subset X_{i+1}, the number of unions of neighborhoods will not exceed $2^{|X_{i+1}|}$ at any point. For each vertex $v \in X_{i+1}$ there are two possibilities; if $v \in S_i$, then appending v to the linear ordering will not increase the boolean dimension, since v's neighborhood was already an element of the unions of neighborhoods constructed so far; if $v \notin S_i$, then it is possible that v will contribute a new neighborhood to the unions of neighborhoods, which will cause factor 2 increase in the worst case. There are at most $|X_{i+1} \setminus S_i|$ such vertices, and because $S_i \subseteq X_{i+1}$, it follows that $|X_{i+1} \setminus S_i| = |X_{i+1}| - |S_i|$. We conclude that at any point during construction it holds that

$$\mathcal{UN}(\chi_{i+1}) = 2^{\text{bool-dim}(\chi_{i+1})} \leq 2^{|S_i|} \cdot 2^{|X_{i+1}|-|S_i|} = 2^{|X_{i+1}|} \leq 2^{\text{pw}(G)+1}$$

◀

The algorithms in this paper make extensive use of sets and set operations, which can be implemented efficiently by using bitsets. By using a mapping from vertices to bitsets that represent the neighborhood of a vertex we can store the adjacency matrix of a graph efficiently. We assume that bitset operations take $O(n)$ time and need $O(n)$ space, even though in practice this may come closer to $O(1)$. If one assumes that these requirements are constant, several time and space bounds in this paper improve by a factor n.

In this paper we assume that the graph G is connected, since if the graph consists of multiple connected components we can simply compute a linear decomposition for each connected component, after which we glue them together, in any arbitrary order.

3 Exact Algorithms

We can characterize the problem of finding an optimal linear decomposition by the following recurrence relation, in which P contains partial solutions.

$$\begin{aligned} P(\{v\}) &= \#\mathcal{UN}(\{v\}) = \begin{cases} 1 & \text{if } N(v) = \emptyset \\ 2 & \text{if } N(v) \neq \emptyset \end{cases} \\ P(A) &= \min_{v \in A}\{\max\{\#\mathcal{UN}(A), \#\mathcal{UN}(A \setminus \{v\})\}\} \end{aligned} \quad (1)$$

The boolean-width of the graph G is now given by $\log_2(P(V))$. Adaptation of existing techniques lead to the following algorithms for linear boolean-width, upon we hereafter improve:

- With dynamic programming a running time of $O(2.7284^n)$ is achieved. (See Theorem 6)
- With adaptation of the exact algorithm for boolean-width by Vatshelle [17], a running time of $O(n^3 \cdot 2^{n+\text{lboolw}(G)})$ is achieved. (See Theorem 7)

▶ **Theorem 6.** *A linear boolean decomposition of minimum boolean-width can be computed in $O(2.7284^n)$ time using $O(n \cdot 2^n)$ space.*

Proof. As a preprocessing step we compute for all cuts $A \subseteq V$ the values $\#\mathcal{UN}(A)$ by computing $\#\mathcal{MIS}(G[A, \overline{A}])$. Computing $\#\mathcal{MIS}$ for any graph can be done in $O(1.3642^n)$ time [6]. Doing this for all A takes $O(2.7284^n)$ time.

We solve recurrence relation (1) in a bottom-up fashion. For each iteration, the minimum of $|A|$ numbers has to be taken. Suppose $|A| = k$, then this takes $O(k)$ time for each iteration. When solving the recurrence relation, $|A|$ goes from 1 to n. Since there are $\binom{n}{k}$ subsets of size k, it takes $\sum_{k=1}^{n} \binom{n}{k} k = O(n \cdot 2^{n-1}) = O(n \cdot 2^n)$ time to compute all values for lboolw.

Because the preprocessing step of computing bool-dim is the bottleneck, the total time is $O(2.7284^n)$. The space requirements amount to $O(n \cdot 2^n)$, since bool-dim and lboolw contain at most 2^n entries of integers of at most n bits. ◀

The currently fastest known exact algorithm for boolean-width runs in $O^*(2^{n+K})$ [17], where K is a known upperbound for the boolean-width of the current graph. By performing a binary search on K, we can achieve an output sensitive asymptotic running time. Theorem 7 is a direct adaptation to linear boolean-width.

▶ **Theorem 7.** *A linear boolean decomposition of minimum boolean-width for a graph G can be computed in $O(n^3 \cdot 2^{n+\text{lboolw}(G)})$ time using $O(n \cdot 2^n)$ space.*

Proof. As a preprocessing step we compute for all cuts $A \subseteq V$ the values $\#\mathcal{UN}(A)$, using a polynomial time delay algorithm, which lists maximal independent sets in $G[A, \overline{A}]$ with at most $O(n^3)$ time in between two results [4]. We can use the upperbound K as a limit for this algorithm, such that computing $\max(\#\mathcal{UN}(A), K)$ takes at most $O(n^3 K)$ time.

Now consider relation (1). This can be solved in $O(n \cdot 2^n)$ time by the same reasoning as in Theorem 6. This results in a total running time of $O(n^3 \cdot 2^{n+\text{lboolw}(G)})$ by binary search on K. The space requirements amount to $O(n \cdot 2^n)$, since the tables bool-dim and lboolw contain at most 2^n entries of integers of at most n bits. ◀

3.1 Improving the running time

We present a faster and easier way to precompute for all cuts $A \subseteq V$ the values $\#\mathcal{UN}(A)$, which results in a new algorithm displayed in Algorithm 2. In the following it is important that the $\mathcal{UN}$ sets are implemented as hashmaps, which will only save distinct neighborhoods.

Algorithm 1 Compute $\mathcal{UN}(X \cup \{v\})$ given $\mathcal{UN}(X)$.

1: **function** Increment-UN($G, X, \mathcal{UN}_X, v$)
2: $\quad U \leftarrow \emptyset$
3: $\quad$ **for** $S \in \mathcal{UN}_X$ **do**
4: $\quad\quad U \leftarrow U \cup \{S \setminus \{v\}\}$
5: $\quad\quad U \leftarrow U \cup \big\{(S \setminus \{v\}) \cup (N(v) \cap (\overline{X} \setminus \{v\}))\big\}$
6: $\quad$ **return** U

▶ **Lemma 8.** *The procedure Increment-UN is correct and runs in* $O(n \cdot |\mathcal{UN}_X|)$ *time using* $O(n \cdot |\mathcal{UN}_X|)$ *space.*

Proof. For proof by induction, assume that all unions of neighborhoods for the cut $(X, \overline{X})$ saved inside the set $\mathcal{UN}_X$ are computed correctly. For each neighborhood in $\mathcal{UN}_X$ we only perform two actions to obtain new neighborhoods. The first action is removing v, since v cannot be in any neighborhood of $X \cup \{v\}$. The second operation is adding $N(v)$ to an existing neighborhood, which also results in a valid new neighborhood across the cut. It is clear that if a neighborhood is added to U, then it is a valid neighborhood across the cut $(X \cup \{v\}, \overline{X} \setminus \{v\})$. We now show that all valid neighborhoods of the cut $(X \cup \{v\}, \overline{X} \setminus \{v\})$ are contained in U. Assume for contradiction that S is a valid neighborhood not contained in U. By definition, there is a set R for which $N(R) \cap (\overline{X} \setminus \{v\}) = S$. If $v \notin R$, then $N(R) \cap \overline{X} \in \mathcal{UN}_X$, meaning that we add $N(R) \cap (\overline{X} \setminus \{v\})$ to U, contradicting our assumption. If $v \in R$, then $N(R \setminus \{v\}) \cap \overline{X} \in \mathcal{UN}_X$. During the algorithm we construct $(N(R \setminus \{v\}) \cup N(v)) \cap (\overline{X} \setminus \{v\})$, which is equal to $N(R) \cap (\overline{X} \setminus \{v\})$. This means that $N(R) \cap (\overline{X} \setminus \{v\})$ is added to U, also contradicting our assumption. It follows that a neighborhood is contained in the set U if and only if it is a valid neighborhood across the cut $(X \cup \{v\}, \overline{X} \setminus \{v\})$.

The running time is determined by the number of sets S saved in $\mathcal{UN}_X$. The number of unions of neighborhoods that we iterate over does not exceed 2^k, where k is the boolean dimension of $\mathcal{UN}_X$. The set operations that are performed for each S take at most $O(n)$ time. This results in the total time for this algorithm to be $O(n \cdot |\mathcal{UN}_X|)$. The space requirements amount to $O(n \cdot |\mathcal{UN}_X|)$, for storing U which contains at most $O(|\mathcal{UN}_X|)$ sets of size at most $O(n)$. ◀

▶ **Lemma 9.** *Given a graph* $G = (V, E)$ *of size* n *and an integer* K*, Algorithm 2 computes the linear boolean width, if it is at most* $\log K$ *in* $O(n \cdot K \cdot 2^n)$ *time using* $O(n \cdot 2^n)$ *space.*

Proof. Consider the first part of procedure Incremental-UN-exact, where the call to the procedure Compute-count-UN is made. It may not be immediately clear that $\#\mathcal{UN}$ is always computed when necessary, since there may be X such that $\#\mathcal{UN}(X)$ is not computed, while $\#\mathcal{UN}(X) \leq K$. Suppose that $X \subseteq V$ of size i occurs in an optimal decomposition and $\#\mathcal{UN}(X)$ has not been computed. Since we are dealing with linear decompositions, there exists an ordering $v_1, \ldots, v_i$ of X such that for all $1 \leq j \leq i$, the set $X_j = \bigcup_{0 \leq j' \leq j} v_{j'}$ also occurs in the optimal decomposition. Obviously this implies that $\#\mathcal{UN}(X_j) \leq K$ for all j. But this means that for all these X_j the if-statement on line 23 evaluates to true. But that means that $\#\mathcal{UN}(X)$ must be computed, contradiction. Thus we conclude that $\#\mathcal{UN}$ is computed correctly throughout the algorithm. The second part of procedure Incremental-UN-exact simply solves the recurrence in a bottom-up dynamic programming fashion. Finally, the procedure Increment-UN is correct by Lemma 8.

We now analyze the running time. Consider the procedure Compute-count-UN. We observe that the procedure can only be called once for each $X \subseteq V$, because as soon as the call is made, $\#\mathcal{UN}(X)$ will be defined and line 20 prevents further calls with equal X. At every call the for-loop has to make at most n iterations, thus we obtain $O(n \cdot 2^n)$ iterations in total. If line 20 evaluates false, the body of the for-loop takes constant time. If line 20 evaluates true, the call to Increment-UN takes $O(n \cdot 2^K)$ time (by Lemma 8), as $|\mathcal{UN}_X| \leq K$ (otherwise by line 23 the call to Compute-count-UN would not have been made). Because line 20 only returns true at most $O(2^n)$ times, the time of Compute-count-UN amounts to $O(n \cdot 2^{n+K})$. Consider the rest of the code in Incremental-UN-exact. The three outer for-loops account for $n \cdot 2^n$ executions of the inner code block, which take $O(1)$ time, resulting in $O(n \cdot 2^n)$ time in total. Thus, in total the time amounts $O(n \cdot 2^{n+K})$.

Algorithm 2 Return $\text{lboolw}(G)$, if it is smaller than $\log K$, otherwise return ∞.

```
 1: function INCREMENTAL-UN-EXACT(G, K)
 2:     #UN(∅) ← 0
 3:     COMPUTE-COUNT-UN(G, K, #UN, ∅, {∅})
 4:
 5:     P(X) ← ∞, for all X ⊆ V
 6:     P(∅) ← 0
 7:
 8:     for i ← 0, ..., |V| − 1 do
 9:         for X ⊆ V of size i do
10:             for v ∈ V \ X do
11:                 Y ← X ∪ {v}
12:                 if P(X) ≤ K then
13:                     P(Y) ← min(P(Y), max(#UN(Y), P(X)))
14:
15:     return log_2(P(V))
16:
17: procedure COMPUTE-COUNT-UN(G, K, #UN, X, UN_X)
18:     for v ∈ V \ X do
19:         Y ← X ∪ {v}
20:         if #UN(Y) is not defined then
21:             UN_Y ← INCREMENT-UN(G, X, UN_X, v)
22:             #UN(Y) ← |UN_Y|
23:             if #UN(Y) ≤ K then
24:                 COMPUTE-COUNT-UN(G, K, #UN, Y, UN_Y)
```

For the space requirements, we observe that the tables $\#\mathcal{UN}$ and S are of size at most 2^n storing numbers of n bits. Moreover, the recursion of COMPUTE-COUNT-UN can be at most n deep, so only n unions of neighborhoods have to be stored, which are at most of size $n \cdot 2^K$. Since $O(n \cdot 2^K) \subseteq O(n \cdot 2^{n/2}) \subsetneq O(n \cdot 2^n)$, the total space requirements amount to $O(n \cdot 2^n)$. ◀

▶ **Theorem 10.** *Given a graph G, Algorithm 2 can be used to compute* $\text{lboolw}(G)$ *in* $O(n \cdot 2^{n+\text{lboolw}(G)})$ *time using* $O(n \cdot 2^n)$ *space.*

Proof. Iteratively double K in Algorithm 2, starting with $K = 1$, until it returns a number that is not ∞. By Lemma 9 this will take $O(\sum_{\log K=1}^{\text{lboolw}(G)} n \cdot 2^{n+\log K}) = O(n \cdot 2^{n+\text{lboolw}(G)+1}) = O(n \cdot 2^{n+\text{lboolw}(G)})$ and take $O(n \cdot 2^n)$ space. ◀

This new algorithm improves upon the time in Theorem 7 by a factor n^2, while the space requirements stay the same. Since the tightest known upperbound for linear boolean-width is $\frac{n}{2} - \frac{n}{143} + O(1)$ [10], this algorithm can be slower than dynamic programming, since $O(2^{n+\frac{n}{2}-\frac{n}{143}+O(1)}) = O(2.8148^{n+O(1)}) \supsetneq O(2.7284^n)$, but this is very unlikely to happen in practice.

4 Heuristics

4.1 Generic form of the heuristics

The goal when using a heuristic is to find a linear ordering of the vertices in a graph in such a way that the decomposition that corresponds to this ordering will be of low boolean-width. A basic strategy to accomplish this is to start the ordering with some vertex and then by some selection criteria append a new vertex to the ordering that has not been appended yet. This strategy is used in heuristics introduced by Sharmin [12, Chapter 8], and a similar approach is shown in Algorithm 3.

Algorithm 3 Greedily generate an ordering based on the score function and the given starting vertex.

1: **procedure** GENERATEVERTEXORDERING($G, ScoreFunction, init$)
2: $\quad Decomposition \leftarrow (init)$
3: $\quad Left \leftarrow \{init\}$
4: $\quad Right \leftarrow V \setminus \{init\}$
5: $\quad$ **while** $Right \neq \emptyset$ **do**
6: $\quad\quad Candidates \leftarrow$ set returned by candidate set strategy
7: $\quad\quad$ **if** there exists $v \in Candidates$ belonging to a trivial case **then**
8: $\quad\quad\quad chosen \leftarrow v$
9: $\quad\quad$ **else**
10: $\quad\quad\quad chosen \leftarrow \underset{v \in Candidates}{\text{argmin}} (ScoreFunction(G, Left, Right, v))$
11: $\quad\quad Decomposition \leftarrow Decomposition \cdot \{chosen\}$
12: $\quad\quad Left \leftarrow Left \cup \{chosen\}$
13: $\quad\quad Right \leftarrow Right \setminus \{chosen\}$
14: **return** $Decomposition$

At any point in the algorithm we denote the set of all vertices contained in the ordering by $Left$, and the remaining vertices by $Right$. While $Right$ is not empty, we choose a vertex from a candidate set $Candidates \subseteq Right$, based on a set of trivial cases, and, if no trivial case applies, by making a local greedy choice using a score function that indicates the quality of the current state $Left, Right$.

4.1.1 Selecting the initial vertex

Selecting a good initial vertex can be of great influence on the quality of the decomposition. Sharmin proposes to use a double breadth first search (BFS) in order to select the initial vertex. This is done by initiating a BFS, starting at an arbitrary vertex, after which a vertex of the last level of the BFS is selected. This process is then repeated by using the found vertex as a starting point for the second BFS. However, the fact that an arbitrary vertex is used for the first BFS already influences the boolean-width of the computed decomposition. During our experiments we noticed that performing a single BFS sometimes gave better results. But since we will see in Chapter 5 that applications are a lot more expensive in terms of running time, it is wise to use all possible starting vertices when trying to find a good decomposition.

4.1.2 Pruning

Starting from multiple initial vertices allows us to do some pruning. If we notice during the algorithm that the score of the decomposition that is being constructed exceeds the score of the best decomposition found so far, we can stop immediately and move to the next initial vertex. For this reason, it is wise to start with the most promising initial vertices (e.g. obtained by the double BFS method), and after that try all other initial vertices.

4.1.3 Candidates

The most straightforward choice for the set *Candidates* is to take *Right* entirely. However, we may do unnecessary work here, since vertices that are more than 2 steps away from any vertex in $Left$ cannot decrease the size of $\mathcal{UN}$. This means that they should never be chosen by a greedy score function, which means that we can skip them right away. By this reasoning, the set of *Candidates* can be reduced to $N^2(Left) \cap Right = N(Left \cup N(Left)) \cap Right$. Especially for larger sparse graphs, this can significantly decrease the running time.

4.1.4 Trivial cases

A vertex is chosen to be the next vertex in the ordering if it can be guaranteed that it is an optimal choice by means of a trivial case. Lemma 11 generalizes results by Sharmin [12], since the two trivial cases given by her are subcases of our lemma, namely $X = \emptyset$ and $X = \{u\}$ for all $u \in Left$. Note that we can add a wide range of trivial cases by varying X, such as $X = Left$ and $\forall u, w \in Left : X = \{u, w\}$, but this will increase the complexity of the algorithm.

▶ **Lemma 11.** *Let $X \subseteq Left$. If $\exists v \in Right$ such that $N(v) \cap Right = N(X) \cap Right$, then choosing v will not change the boolean-width of the resulting decomposition.*

Proof. The choice for v will not change the unions of neighborhoods in any way, which means that $\mathcal{UN}(Left) = \mathcal{UN}(Left \cup \{v\})$. Thus, for any vertex in $Right \setminus \{v\}$ it will hold that it will interact in the exact same with with $\mathcal{UN}(Left)$ as it would with $\mathcal{UN}(Left \cup \{v\})$, resulting in the boolean dimension of the computed ordering being the same. ◀

4.1.5 Relative Neighborhood Heuristic

For a cut $(Left, Right)$ and a vertex v define

$$Internal(v) = (N(v) \cap N(Left)) \cap Right$$
$$External(v) = (N(v) \setminus N(Left)) \cap Right$$

In the original formulation by Sharmin [12] $\frac{|External(v)|}{|Internal(v)|}$ is used as a score function. However, if we use $\frac{|External(v)|}{|Internal(v)|+|External(v)|} = \frac{|External(v)|}{|N(v)\cap Right|}$ we get the same ordering by Lemma 12, without having an edge case for dividing by zero. Furthermore, in contrast to Sharmin's proposal of checking for each vertex $w \in N(v)$ if $w \in N(Left) \cap Right$ or not, we can compute these sets directly by performing set operations. We will refer to this heuristic by RelativeNeighborhood.

▶ **Lemma 12.** *The mapping $\frac{a}{b} \mapsto \frac{a}{a+b}$ is order preserving.*

Proof. Suppose $\frac{a}{b} \leq \frac{c}{d}$. Then $ad - bc \leq 0$. Now we see that

$$\frac{a}{a+b} - \frac{c}{c+d} = \frac{a(c+d) - c(a+b)}{(c+d)(a+b)} = \frac{ac + ad - ac - bc}{(c+d)(a+b)} = \frac{ad - bc}{(c+d)(a+b)} \leq 0$$

Thus $\frac{a}{a+b} \leq \frac{c}{c+d}$. ◀

Two variations on this heuristic can be obtained through the score functions $\frac{|External(v)|}{|N(v)|}$ and $1 - \frac{|Internal(v)|}{|N(v)|}$, which work slightly better for sparse random graphs and extremely well for dense random graphs respectively. We will refer to these two variations by RelativeNeighborhood$_2$ and RelativeNeighborhood$_3$.

One can easily see that the running time of these three algorithms is $O(n^3)$ and the required space amounts to $O(n)$. Notice however that this algorithm only gives us a decomposition. If we need to know the corresponding boolean-width we need to compute it afterwards, for instance by iteratively applying Increment-UN on the vertices in the decomposition, and taking the maximum value. This would require an additional $O(n^2 \cdot 2^k)$ time and $O(n \cdot 2^k)$ space, where k is the boolean-width of the decomposition.

4.1.6 Least Cut Value Heuristic

The LeastCutValue heuristic by Sharmin [12] greedily selects the next vertex $v \in Right$ that will have the smallest boolean dimension across the cut $(Left \cup \{v\}, Right \setminus \{v\})$. This vertex is obtained by constructing the bipartite graph $BG = G[Left \cup \{v\}, Right \setminus \{v\}]$ for each $v \in Right$, and counting the number of maximal independent sets of BG using the CCM_{IS} [9] algorithm on BG, with the time of CCM_{IS} being exponential in n.

4.1.7 Incremental Unions of Neighborhoods Heuristic

Generating a bipartite graph and then calculating the number of maximal independent sets is a computational expensive approach. A different way to compute the boolean dimension of each cut is by reusing the neighborhoods from the previous cut, similarly to Incremental-UN-exact. We present a new algorithm, called the Incremental-UN-heuristic, in Algorithm 4. A useful property of this algorithm is that the running time is output sensitive. It follows that if a decomposition is not found within reasonable time, then the decomposition that would have been generated is not useful for practical algorithms.

▶ **Theorem 13.** *The* Incremental-UN-heuristic *procedure runs in $O(n^3 \cdot 2^k)$ time using $O(n \cdot 2^k)$ space, where k is the boolean-width of the resulting linear decomposition.*

Proof. The running time is determined by the number of sets saved in $\mathcal{UN}_{Left}$. The worst case consisting of $Candidates = Right$ will result in at most n iterations and calls to Increment-UN. This call takes $O(n \cdot |\mathcal{UN}_{Left}|)$ time by Lemma 8. By definition $|\mathcal{UN}_{Left}|$ never exceeds 2^k, where k is the boolean-width of the resulting decomposition. Because we need to make n greedy choices to process the entire graph, we conclude that the total time for this algorithm is $O(n^3 \cdot 2^k)$ For the space requirements we observe that all structures in the algorithm require $O(n)$ space, except for the unions of neighborhoods. Since there are only stored two of them at any time and they require at most $O(n \cdot 2^k)$ space, the total space requirements amount to $O(n \cdot 2^k)$. ◀

Algorithm 4 Greedy heuristic that incrementally keeps track of the Unions of Neighborhoods.

```
1: procedure INCREMENTAL-UN-HEURISTIC(G, init)
2:     Decomposition ← (init)
3:     Left, Right ← {init}, V \ {init}
4:     UN_Left ← {∅, N(init) ∩ Right}
5:     while Right ≠ ∅ do
6:         Candidates ← set returned by candidate set strategy
7:         if there exists v ∈ Candidates belonging to a trivial case then
8:             chosen ← v
9:             UN_chosen ← INCREMENT-UN(G, Left, UN_Left, v)
10:        else
11:            #UN_chosen ← ∞
12:            for all v ∈ Candidates do
13:                UN_v ← INCREMENT-UN(G, Left, UN_Left, v)
14:                if |UN_v| < #UN_chosen then
15:                    chosen ← v
16:                    UN_chosen ← UN_v
17:                    #UN_chosen ← |UN_v|
18:        Decomposition ← Decomposition · chosen
19:        Left ← Left ∪ {chosen}
20:        Right ← Right \ {chosen}
21:        UN_Left ← UN_chosen
22:    return Decomposition
```

4.1.8 Unsuccessful ideas

- First Improvement — Preliminary experiments pointed out that it not only gave worse results in terms of boolean-width, but it also increased the time needed to compute a decomposition, which can be explained by the output sensitivity of the INCREMENTAL-UN-HEURISTIC. In other words, even though the best improvement strategy takes more time to determine the next vertex for a single iteration, it is worthwhile to put effort in finding a good cut, as it also decreases the time for future cuts.
- Lookaheads — This technique does not only look at the change of $\mathcal{UN}$ resulting from choosing a candidate v, but also recursively considers the changes of the algorithm after v has been chosen, up to a fixed depth. With each level of depth added, the time complexity increases with a factor n, but experiments turned out that the benefits were only marginal.
- Minimal Neighborhood Cover — This heuristic tries to minimize the number of neighborhoods in $Left$ that are needed to cover the neighborhood of the vertex to be chosen.
- Max Cardinality Search — This heuristics selects vertices in such an order that at each step the vertex with most neighbors in $Left$ is chosen. In practice this heuristic performed similar to other already known polynomial heuristics.

5 Vertex subset problems

Boolean decompositions can be used to efficiently solve a class of vertex subset problems called (σ, ρ) vertex subset problems, which were introduced by Telle [13]. This class of problems consists of finding a (σ, ρ)-set of maximum or minimum cardinality and contains

well known problems such as the maximum independent set, the minimum dominating set and the maximum induced matching problem. The running time of the algorithm for solving these problems is $O(n^4 \cdot nec_d(T,\delta)^3)$ [3], where $nec_d(T,\delta)$ is the number of equivalence classes of a problem specific equivalence relation, which can be bounded in terms of boolean-width. In Section 6 we investigate how close the value of $nec_d(T,\delta)$ comes to any of the theoretical bounds.

5.1 Definitions

▶ **Definition 14** ((σ,ρ)-set)**.** Let $G=(V,E)$ be a graph. Let σ and ρ be finite or co-finite subsets of $\mathbb{N}$. A subset $X \subseteq V$ is called a (σ,ρ)-set if the following holds

$$\forall v \in V : |N(v) \cap X| \in \begin{cases} \sigma & \text{if } v \in X, \\ \rho & \text{if } v \in V \setminus X. \end{cases}$$

In order to confirm if a set X is a (σ,ρ)-set we have to count the number of neighbors each vertex $v \in V$ has in X, where it suffices to count up until a certain number of neighbors. As an example, when we want to confirm if a set X is an independent set, which is equivalent to checking if X is a $(\{0\},\mathbb{N})$-set, it is irrelevant if a vertex v has more than one neighbor in X. We capture this property in the function $d : 2^{\mathbb{N}} \to \mathbb{N}$, which is defined as follows:

▶ **Definition 15** (d-function)**.** Let $d(\mathbb{N}) = 0$. For every finite or co-finite set $\mu \subseteq \mathbb{N}$, let $d(\mu) = 1 + \min(\max_{x\in\mathbb{N}} x : x \in \mu, \max_{x\in\mathbb{N}} x : x \notin \mu)$. Let $d(\sigma,\rho) = \max(d(\sigma), d(\rho))$.

▶ **Definition 16** (d-neighborhood)**.** Let $G=(V,E)$ be a graph. Let $A \subseteq V$ and $X \subseteq A$. The *d-neighborhood* of X with respect to A, denoted by $N_A^d(X)$, is a multiset of vertices from $\overline{A}$, where a vertex $v \in \overline{A}$ occurs $\min(d, |N(v) \cap X|)$ times in $N_A^d(X)$. A d-neighborhood can be represented as a vector of length $|\overline{A}|$ over $\{0,1,\ldots,d\}$.

▶ **Definition 17** (d-neighborhood equivalence)**.** Let $G=(V,E)$ be a graph and $A \subseteq V$. Two subsets $X, Y \subseteq A$ are said to be *d-neighborhood equivalent* with respect to A, denoted by $X \equiv_A^d Y$, if it holds that $\forall v \in \overline{A} : \min(d, |X \cap N(v)|) = \min(d, |Y \cap N(v)|)$. The number of equivalence classes of a cut $(A,\overline{A})$ is denoted by $nec(\equiv_A^d)$. The number of equivalence classes $nec_d(T,\delta)$ of a decomposition (T,δ) is defined as $\max(nec(\equiv_A^d), nec(\equiv_{\overline{A}}^d))$ over all cuts $(A,\overline{A})$ of (T,δ).

Note that $N_A^1(X) = N(X) \cap \overline{A}$. It can then be observed that $\#\mathcal{UN}(A) = nec(\equiv_A^1)$ [17, Theorem 3.5.5] Also note that $X \equiv_A^d Y$ if and only if $N_A^d(X) = N_A^d(Y)$.

5.2 Bounds on the number of equivalence classes

We present a brief overview of the most relevant bounds that are currently known, for which we make use of a *twin class partition* of a graph.

▶ **Definition 18** (Twin class partition)**.** Let $G=(V,E)$ be a graph of size n and let $A \subseteq V$. The *twin class partition* of A is a partition of A such that $\forall x, y \in A$, x and y are in the same partition class if and only if $N(x) \cap \overline{A} = N(y) \cap \overline{A}$. The number of partition classes of A is denoted by $ntc(A)$ and it holds that $ntc(A) \le \min(n, 2^{\text{bool-dim}(A)})$ [2].

For all bounds listed below, let $G=(V,E)$ be a graph of size n and let d be a non-negative integer. Let $(A,\overline{A})$ be a cut induced by any node of a decomposition (T,δ) of G, and let $k = \text{bool-dim}(A) = nec(\equiv_A^1)$.

▶ **Lemma 19.** *[3, Lemma 5]* $nec(\equiv^d_A) \leq 2^{d \cdot k^2}$.

▶ **Lemma 20.** *[17, Lemma 5.2.2]* $nec(\equiv^d_A) \leq (d+1)^{\min(ntc(A), ntc(\overline{A}))}$.

▶ **Lemma 21.** $nec(\equiv^d_A) \leq ntc(A)^{d \cdot k}$.

Proof. We make use of a graph parameter called *maximum induced matching-width* [1]. Let $mim(A)$ denote the maximum matching-width of A. It has been shown that for a graph G and for any subset $A \subseteq V$ it holds that $mim(A) \leq \text{bool-dim}(A)$ [17, Theorem 4.2.10]. From [17, Lemma 5.2.3] we know that $nec(\equiv^d_A) \leq ntc(A)^{d \cdot mim(A)}$, thus $nec(\equiv^d_A) \leq ntc(A)^{d \cdot k}$. ◀

By Lemma 19 we conclude that we can solve (σ, ρ) problems in $O^*(8^{dk^2})$. This shows that applications are more computationally expensive than using heuristics to find a decomposition.

6 Experiments

6.1 Comparing Heuristics on random graphs

We will look at the performance of heuristics on randomly generated graphs, for which we used the Erdös-Rényi-model [5] to generate a fixed set of random graphs with varying edge probabilities. By using the same set of graphs for each heuristic, we rule out the possibility that one heuristic can get a slightly easier set of graphs than another.

In these experiments we start a heuristic once for each possible initial vertex, so n times in total. For the RelativeNeighborhood heuristic we select the best decomposition based upon the sum of the score function for all cuts, since computing all actual linear boolean-width values would take $O(n^3 \cdot 2^k)$ time, thereby removing the purpose of this polynomial time heuristic. For the set *Candidates* we take $N^2(Left) \cap Right$, as opposed to Sharmin [12], who restricted this set to $N(Left) \cap Right$.

We let the edge probability vary between 0.05 and 0.95 with steps of size 0.05. For each edge probability value, we generated 20 random graphs. The result per edge probability is taken to be the average boolean-width over these 20 graphs, which are shown in Figure 1. It can be observed that the Incremental-UN-heuristic procedure performs near optimal. Furthermore we see that the RelativeNeighborhood variants perform somewhere in between the optimal value and the value of random decompositions.

In Figure 2 we show the performance of different heuristics on random generated graphs consisting of 50 vertices, with varying edge probabilities. Because of feasibility limitations, the Incremental-UN-exact algorithm is only used for the graphs in Figure 1. While the optimal values are now unknown, it is clear that Incremental-UN-heuristic outperforms all other heuristics. Interestingly enough, $\text{RelativeNeighborhood}_3$ peers with Incremental-UN-heuristic as soon as the edge probability exceeds 0.4. Moreover, RelativeNeighborhood and $\text{RelativeNeighborhood}_2$ do not perform better than a random decomposition generator after the edge probability exceeds 0.4. We also observe that the highest boolean-width values are reached when the edge probability is around 0.1–0.2, indicating that the size of the graphs has an influence on the edge-probability-boolean-width-curve. Also note that it seems that dense random graphs have lower linear boolean-width than sparse graphs. Therefore it may be profitable to use $\text{RelativeNeighborhood}_3$ when dense graphs are encountered.

6.2 Comparing heuristics on practical graphs

The experiments in this section are performed on a 64-bit Windows 7 computer, with a 3.40 GHz Intel Core i5-4670 CPU and 8GB of RAM. We implemented the algorithms using the

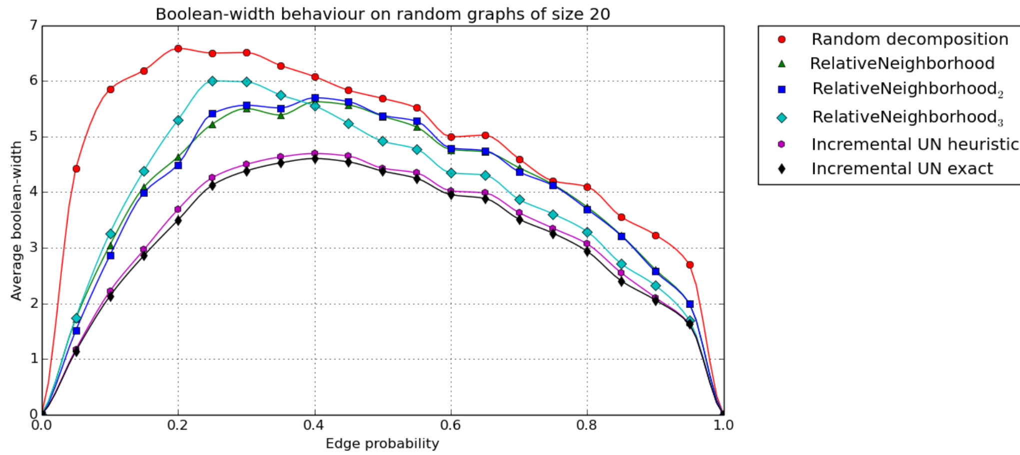


Figure 1 Performance of different heuristics on random generated graphs consisting of 20 vertices, with varying edge probabilities, in terms of linear boolean-width.

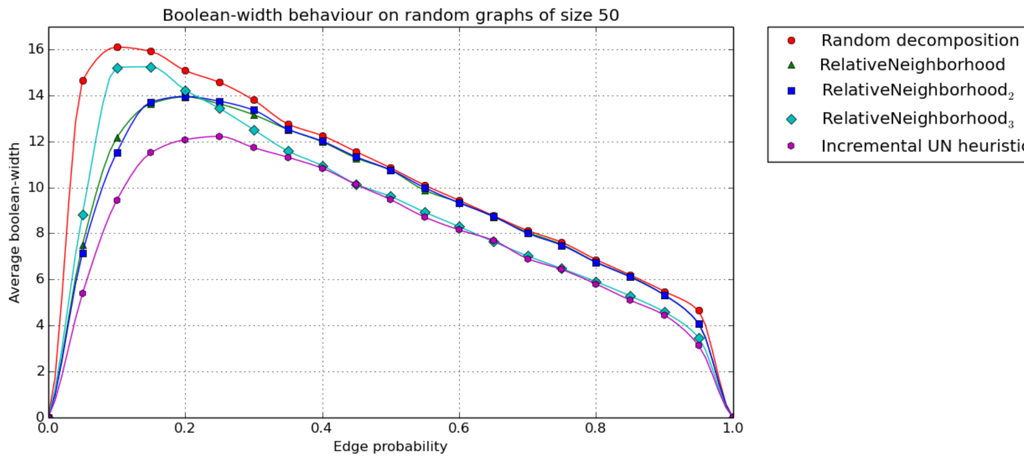


Figure 2 Performance of different heuristics on random generated graphs consisting of 50 vertices.

C# programming language and compiled our programs using the *csc* compiler that comes with Visual Studio 12.0.

In order to get an idea of how the INCREMENTAL-UN-HEURISTIC compares to existing heuristics we compare them by both the boolean-width of the generated decomposition and the time needed for computation. We cannot compare the heuristics to the optimal solution, because computing an exact decomposition is not feasible on these graphs. The graphs that were used come from *Treewidthlib* [14], a collection of graphs that are used to benchmark algorithms using treewidth and related graph problems.

We ran the three different heuristics mentioned in Section 4 with $Candidates = Right$ and with an additional two variations on the INCREMENTAL-UN-HEURISTIC (IUN) by varying the set of start vertices. The first variation, named 2-IUN, has two start vertices which are obtained through a single and double BFS respectively. The n-IUN heuristic uses all possible start vertices. For all other heuristics we obtained the start vertex through performing a double BFS. In Table 1 and 2 we present the results of our experiments.

It is expected that the IUN heuristic and LEASTCUTVALUE heuristic give the same

Table 1 Linear boolean-width of the decompositions returned by different heuristics.

Graph	$\|V\|$	Edge Density	Relative	LeastCut	IUN	2-IUN	n-IUN
alarm	37	0.10	3.32	3.00	3.00	3.00	3.00
barley	48	0.11	5.70	5.91	5.91	4.70	4.58
pigs-pp	48	0.12	10.35	7.13	7.13	7.13	6.64
BN_100	58	0.17	15.84	11.56	11.56	10.86	10.86
eil76	76	0.08	8.86	8.33	8.33	8.33	8.33
david	87	0.11	9.38	6.27	6.27	6.27	5.86
1jhg	101	0.17	12.86	8.67	8.67	8.49	8.41
1aac	104	0.25	20.29	12.40	12.40	12.40	12.33
celar04-pp	114	0.08	11.67	7.27	7.27	7.27	7.27
1a62	122	0.21	18.92	11.68	11.68	11.28	11.14
1bkb-pp	127	0.18	16.81	9.98	9.98	9.53	9.53
1dd3	128	0.17	16.61	9.98	9.98	9.90	9.90
miles1500	128	0.64	8.17	5.58	5.58	5.58	5.29
miles250	128	0.05	7.95	7.13	7.13	5.39	4.58
celar10-pp	133	0.07	10.32	11.95	11.95	7.64	6.91
anna	138	0.05	12.65	8.67	8.67	8.51	7.94
pr152	152	0.04	12.69	11.19	11.19	10.36	8.29
munin2-pp	167	0.03	15.17	9.61	9.61	9.61	7.61
mulsol.i.5	186	0.23	7.55	5.29	5.29	5.29	3.58
zeroin.i.2	211	0.16	7.92	4.46	4.46	4.46	3.81
boblo	221	0.01	19.00	4.32	4.32	4.32	4.00
fpsol2.i-pp	233	0.40	5.58	6.07	6.07	5.78	4.81
munin4-wpp	271	0.02	13.04	9.27	9.27	9.27	7.61

linear boolean-width, since both these heuristics greedily select the vertex that minimizes the boolean dimension. The RelativeNeighborhood heuristic performs worse than all other heuristics in nearly all cases. While the difference might not seem very large, note that algorithms parameterized by boolean-width are exponential in the width of a decomposition. The 2-IUN heuristic outperforms IUN in 11 cases while n-IUN gives a better decomposition in 20 out of 23 cases, which shows that a good initial vertex is of great influence on the width of the decomposition.

Looking at the times displayed in Table 2 for computing each decomposition we see that the RelativeNeighborhood heuristic is significantly faster. This is to be expected because of the $O(n^3)$ time, compared to the exponential time for all other heuristics. The interesting comparison that we can make is the difference between the IUN heuristic and LeastCutValue heuristic. While both of these heuristics give the same decomposition, IUN is significantly faster. Additionally, even 2-IUN and n-IUN are often faster than the LeastCutValue heuristic.

6.3 Vertex subset experiments

We have used the linear decompositions given by the n-IUN heuristic to compute the size of the maximum induced matching (MIM) in a selection of graphs, of which the results are presented in Table 3. The maximum induced matching problem is defined as finding the largest $(\{1\}, \mathbb{N})$ set, with $d(\{1\}, \mathbb{N}) = 2$. The choice for the MIM problem is arbitrary,

Table 2 Time in seconds of the heuristics used to find linear boolean decompositions.

Graph	$\|V\|$	Edge Density	Relative	LeastCut	IUN	2-IUN	n-IUN
alarm	37	0.10	< 0.01	0.02	< 0.01	< 0.01	0.06
barley	48	0.11	< 0.01	0.18	0.01	0.02	0.16
pigs-pp	48	0.12	< 0.01	0.76	0.02	0.04	0.52
BN_100	58	0.17	< 0.01	25.10	0.41	1.24	17.17
eil76	76	0.08	0.02	5.00	0.13	0.29	8.35
david	87	0.11	0.02	3.15	0.04	0.06	1.62
1jhg	101	0.17	0.03	24.46	0.21	0.48	14.75
1aac	104	0.25	0.04	754.54	5.66	11.81	375.31
celar04-pp	114	0.08	0.04	5.73	0.14	0.23	9.85
1a62	122	0.21	0.06	585.95	3.10	11.57	376.26
1bkb-pp	127	0.18	0.06	198.05	1.14	4.18	107.32
1dd3	128	0.17	0.07	117.21	0.92	2.74	91.19
miles1500	128	0.64	0.06	44.57	0.10	0.14	7.05
miles250	128	0.05	0.02	0.56	0.05	0.10	1.24
celar10-pp	133	0.07	0.06	8.93	1.96	4.72	18.43
anna	138	0.05	0.06	20.81	0.22	0.57	19.95
pr152	152	0.04	0.10	50.74	1.76	5.66	120.06
munin2-pp	167	0.03	0.11	3.81	0.80	3.37	30.21
mulsol.i.5	186	0.23	0.09	37.88	0.13	0.27	8.80
zeroin.i.2	211	0.16	0.06	18.70	0.09	0.11	5.85
boblo	221	0.01	0.29	3.39	0.28	0.56	46.22
fpsol2.i-pp	233	0.40	0.18	189.11	0.36	0.74	56.63
munin4-wpp	271	0.02	0.61	57.87	1.98	6.66	367.37

any vertex subset problem with $d = 2$ will have the same number of equivalence classes and therefore they all require the same time when computing a solution. We present the computed value of $nec_d(T, \delta)$, together with theoretical upperbounds, since for $d = 2$ a tight upperbound in terms of boolean-width is not known. Note that we take the logarithm of each value, since we find this value easier to interpret and compare to other graph parameters. We let $UB_1 = 2^{d \cdot \text{boolw}^2}$, $UB_2 = (d+1)^{\min ntc}$ and $UB_3 = ntc^{d \cdot \text{boolw}}$, with $ntc = \max_{w \in T} ntc(V_w)$ and $\min ntc = \max_{w \in T} \min(ntc(V_w), ntc(\overline{V_w}))$.

The column MIM displays the size of the MIM in the graph, while the time column indicates the time needed to compute this set. Missing values for nec and MIM are caused by a lack of internal memory. The reason for this is that the space requirement for the algorithm used to compute the MIM is $O^*(nec_d(T, \delta)^2)$. An interesting observation that we can do, for instance by looking at the graphs zeroin.i.2 and boblo, is that a lower boolean-width does not automatically imply a lower number of equivalence classes. We even encountered this for two decompositions (T, δ) and (T', δ') of the same graph. For instance, for the graph barley we observed $\text{boolw}(T, \delta) = 4.58$ and $\text{boolw}(T', \delta') = 4.81$, while $log_2(nec_2(T, \delta)) = 7.00$ and $log_2(nec_2(T', \delta')) = 6.75$.

7 Conclusion

We have presented a new heuristic and a new exact algorithm for finding linear boolean decompositions. The heuristic has a running time that is several orders of magnitude faster

Table 3 Results of using the algorithm by Bui-Xuan et al. [3] for solving (σ, ρ) problems on graphs, using decompositions obtained using the n-IUN heuristic.

Graph	boolw	$\log_2(nec)$	$\log_2(UB_1)$	$\log_2(UB_2)$	$\log_2(UB_3)$	MIM	Time (s)
alarm	3.00	4.32	18.00	7.92	13.93	18	< 1
barley	4.58	7.00	42.04	12.68	27.51	22	3
pigs-pp	6.64	10.31	88.28	19.02	49.17	22	1147
BN_100	10.86	-	235.93	36.45	105.53	-	-
eil76	8.33	12.63	138.81	22.19	65.10	-	-
david	5.86	9.37	68.63	22.19	44.61	34	919
1jhg	8.41	13.53	141.58	41.21	81.75	-	-
1aac	12.33	-	304.08	72.91	141.25	-	-
celar04-pp	7.27	11.15	105.61	28.53	65.74	-	-
1a62	11.14	-	248.09	60.23	121.61	-	-
1bkb-pp	9.53	-	181.47	52.30	98.49	-	-
1dd3	9.90	-	196.11	52.30	103.17	-	-
miles1500	5.29	9.30	55.87	34.87	49.69	8	4038
miles250	4.58	7.24	42.04	15.85	31.72	52	37
celar10-pp	6.91	10.34	95.41	25.36	59.70	50	10179
anna	7.94	11.94	125.98	33.28	75.48	-	-
pr152	8.29	12.76	137.45	22.19	63.13	-	-
munin2-pp	7.61	11.82	115.97	19.02	54.60	-	-
mulsol.i.5	3.58	6.11	25.70	14.26	24.80	46	22
zeroin.i.2	3.81	6.58	28.99	20.60	28.18	30	59
boblo	4.00	6.17	32.00	9.51	20.68	148	41
fpsol2.i-pp	4.81	8.07	46.22	22.19	36.61	46	934
munin4-wpp	7.61	12.13	115.97	19.02	57.98	-	-

than the previous best heuristic and finds a decomposition in output sensitive time. This means that if a decomposition is not found within reasonable time, then the decomposition that would have been generated is not useful for practical algorithms. Running the new heuristic once for every possible starting vertex results in significantly better decompositions compared to existing heuristics.

We have seen that if $\text{lboolw}(T, \delta) < \text{lboolw}(T', \delta')$, then there is no guarantee that $nec(T, \delta) < nec(T', \delta')$. While in general it holds that minimizing boolean-width results in a low value of number of equivalence classes, we think that can be worthwhile to focus on minimizing the nec_d instead of the boolean-width when solving vertex subset problems. However, the number of equivalence classes is not symmetric, i.e., $nec(\equiv^d_A) \neq nec(\equiv^d_{\overline{A}})$ does not always holds for a cut $(A, \overline{A})$, which makes it harder to develop fast heuristics that focus on minimizing nec_d since we need to keep track of both the equivalence classes of A and $\overline{A}$.

Further research can be done in order to obtain even better heuristics and better upperbounds on both the linear boolean-width and boolean-width on graphs. For instance, combining properties of the INCREMENTAL-UN-HEURISTIC and the RELATIVENEIGHBORHOOD heuristic might lead to better decompositions, as they make use of complementary features of a graph. Another approach for obtaining good decompositions could be a branch and bound algorithm that makes us of trivial cases that are used in the heuristics. To decrease the time needed by the heuristics one can investigate reduction rules for linear boolean-width. While most reduction rules introduced by Sharmin [12] for boolean-width do

not hold for linear boolean-width, they can still be used on a graph after which we can use our heuristic on the reduced graph. Although the resulting decomposition after reinserting the reduced vertices will not be linear, the asymptotic running time for applications does not increase [15]. Another topic of research is to compare the performance of vertex subset algorithms parameterized by boolean-width to algorithms parameterized by treewidth [16].

References

1. R. Belmonte and M. Vatshelle. Graph classes with structured neighborhoods and algorithmic applications. *Theoretical Computer Science*, 511(0):54 – 65, 2013. Exact and Parameterized Computation.
2. B.-M. Bui-Xuan, J. A. Telle, and M. Vatshelle. Boolean-width of graphs. In *IWPEC 2009*, volume 5917 of *LNCS*, pages 61–74. Springer, 2009.
3. B.-M. Bui-Xuan, J. A. Telle, and M. Vatshelle. Fast dynamic programming for locally checkable vertex subset and vertex partitioning problems. *Theoretical Computer Science*, 511:66–76, 2013.
4. V. M.F. Dias, C. M.H. de Figueiredo, and J. L. Szwarcfiter. On the generation of bicliques of a graph. *Discrete Applied Mathematics*, 155(14):1826 – 1832, 2007.
5. P. Erdös and A. Rényi. On random graphs. *Publicationes Mathematicae 6: 290–297*, 1959.
6. S. Gaspers, D. Kratsch, and M. Liedloff. On independent sets and bicliques in graphs. In *Proc. WG 2008*, volume 5344 of *LNCS*, pages 171–182. Springer, 2008.
7. E. M. Hvidevold, S. Sharmin, J. A. Telle, and M. Vatshelle. Finding good decompositions for dynamic programming on dense graphs. In *IWPEC 2012*, volume 7112 of *LNCS*, pages 219–231. Springer, 2012.
8. K. H. Kim. *Boolean matrix theory and its applications (Monographs and textbooks in pure and applied mathematics)*. Marcel Dekker, 1982.
9. F. Manne and S. Sharmin. Efficient counting of maximal independent sets in sparse graphs. In *Experimental Algorithms*, volume 7933 of *LNCS*, pages 103–114. Springer, 2013.
10. Y. Rabinovich, J. A. Telle, and M. Vatshelle. Upper bounds on boolean-width with applications to exact algorithms. In *IWPEC 2013*, volume 8246 of *LNCS*, pages 308–320. Springer, 2013.
11. N. Robertson and P. D. Seymour. Graph minors. I. Excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39 – 61, 1983.
12. S. Sharmin. *Practical Aspects of the Graph Parameter Boolean-width*. PhD thesis, University of Bergen, Norway, 2014.
13. J. A. Telle. Complexity of domination-type problems in graphs. *Nordic Journal of Computing*, 1(1):157–171, 1994.
14. Treewidthlib. http://www.staff.science.uu.nl/∼bodla101/treewidthlib/. A benchmark for algorithms for treewidth and related graph problems.
15. F. J. P. van Houten. Experimental research and algorithmic improvements involving the graph parameter boolean-width. Master's thesis, Utrecht University, The Netherlands, 2015.
16. J. M. M. van Rooij, H. L. Bodlaender, and P. Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In *Algorithms - ESA 2009*, volume 5757 of *LNCS*, pages 566–577. Springer, 2009.
17. M. Vatshelle. *New width parameters of graphs*. PhD thesis, University of Bergen, Norway, 2012.