Universiteit Utrecht

Department of Information and Computing Sciences

Variations on Boolean-width

Master Thesis
ICA-3677133

Author:
Chiel Ch.B. ten Brinke
Utrecht University
ctenbrinke@gmail.com

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

August 2015

Contents

Abstract

This thesis is about the graph parameter *boolean-width*[8] and several related parameters. In particular we investigate the restriction of boolean-width to linear decompositions, called linear boolean-width. Improving upon existing work, we give several new algorithms, both exact and heuristical, and test these experimentally. We also look at reduction rules for linear boolean-width. After that we consider cost variants of these parameters, which optimize a decomposition by means of minimizing the sum of all cut values in a decomposition rather than taking the maximum. We give a general non-trivial exact algorithm to compute decompositions with minimal f-cost, for any cut function f. Finally we evaluate these topics and give suggestions about what is worth further investigating.

Acknowledgments

I've been working together a lot with Frank van Houten, who has written his thesis about linear boolean-width [31]. Together we wrote a paper about this subject, under supervision of Hans Bodlaender, which has been accepted for the IPEC 2015 conference. The paper can be found at the end of the appendix.

With respect to my collaboration with Frank, I find it suitable to point out the parts in the results where he has done a critical part of the job. Firstly, we have been working together finding reduction rules for linear boolean-width, but Frank put it all into LaTeX, and made it a clean story. Secondly, Frank initially came up with the idea for the incremental-un-heuristic. This in turn inspired me to formulate the incremental-un-exact algorithm. Finally, both of us have an implementation of the algorithms that we used and tested, but since many of the experiments involve timing, we were forced to perform these on the same computer with the same implementation. We decided to do these timed experiments on Frank's implementation, while for some non-timed experiments we used mine.

I want to thank Frank van Houten for the pleasant collaboration, and Hans Bodlaender, who has been a great supervisor to me. Furthermore, I would like to thank my colleagues in the tree-width group for the good atmosphere in the room where we have all been writing our master theses.

Chapter 1

Introduction

Boolean-width is a recently introduced graph parameter [8]. Similarly to treewidth and other parameters, it measures some structural complexity of a graph. Many NP-hard problems on graphs become easier (i.e. Fixed Parameter Tractable) 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. This decomposition is bounded in size because the derivation of it relies on the graph parameter we are exploiting. Then a problem specific algorithm is applied to this derived decomposition, for instance dynamic programming or divide and conquer. However, computing an optimal decomposition tree is usually a hard problem in itself. A common approach to bypass this problem is to use heuristics to compute decompositions with a low boolean-width.

Algorithms for generating boolean decompositions have been studied before in [33, 22, 26, 2, 14]. In this thesis we investigate boolean-width and related parameters. In particular we investigate the restriction of boolean-width to linear decomposition, called linear boolean-width. Improving upon existing work, we give several new algorithms, both exact and heuristic, and test these experimentally. We also look at reduction rules for linear boolean-width. After that we consider cost variants of these parameters, which optimize a decomposition by means of minimizing the sum of all cut values in a decomposition rather than taking the maximum. We give a general non-trivial exact algorithm to compute decompositions with minimal f-cost, for any cut function f. Finally we evaluate these topics and give suggestions about what is worth further investigating.

My implementation of the algorithms that were used can be found on Github [29]. At the time of writing, the implementation of Frank van Houten is yet to be published, but should be on Github soon as well.

Chapter 2

Preliminaries

This chapter is dedicated to introducing the various mathematical concepts that we will be working with and to providing some examples to illustrate these definitions.

2.1 Graph Theory

In this subsection we briefly mention the definition of the graph concepts that we are working with.

Graph A graph G is a pair $V(G)$ called the vertices, and $E(G)$ called the edges where edges are unordered pairs of the vertices. We often will write $G = (V, E)$ when there is no ambiguity about which graph is considered. All graphs that we deal with in this thesis are finite, undirected, and simple.

Clique A graph $G = (V, E)$ is called a clique or a complete graph if for every pair of vertices $u, v \in V$ we have that $(u, v) \in E$.

Connected graph A graph $G = (V, E)$ is connected if for every pair of vertices $u, v \in V$ there exist a path from u to v. A graph that is not connected is called a disconnected graph.

Neighborhood For a graph $G = (V, E)$ and a vertex $v \in V$, the neighborhood of v, denoted $N(v)$, is the set of all vertices in G adjacent to v, i.e. the set $\{w \in V \mid (v, w) \in E\}$. The closed neighborhood is denoted $N[v] = N(v) \cup \{v\}$.

The neighborhood of a set $X \subseteq V$ is denoted $N(X) = \bigcup_{v \in X} N(v)$, and the closed neighborhood of X is denoted $N[X] = N(X) \cup X$. Denote with $N_G(X)$ the neighborhood of X in the graph G.

Twins For a graph $G = (V, E)$, two vertices $x, y \in V$ are twins if and only if $N(x) \setminus y = N(y) \setminus x$.

Vertex complement For a graph $G = (V, E)$ and $A \subseteq V$, the complement of A is denoted $A = V \setminus A$.

Complement graph The complement of a graph G, denoted $\overline{G}$, is the graph where $V(G) = V(\overline{G})$ and for any pair $u, v \in V(G)$ with $u \neq v$ we have $(u, v) \in E(G)$ if and only if $(u, v) \notin E(\overline{G})$.

Cut For a graph $G = (V, E)$ and $A \subseteq V$, the cut in G defined by A is the partition $(A, \overline{A})$ of V. The neighborhood of $X \subseteq A$ across $(A, \overline{A})$ is $N(X) \cap \overline{A}$. Two vertices $x, y \in A$ are twins across (A, A) if $N(x) \cap A = N(y) \cap A$.

Bipartite graph We say a graph $G = (V, E)$ is bipartite if there exist a subset $A \subseteq V$ such that every edge in E has one endpoint in A and the other in $\overline{A}$.

Induced bipartite subgraph For a graph $G = (V, E)$ and $A, B \subseteq V$ such that $A \cap B = \emptyset$. The bipartite graph induced by the two subsets is denoted $G[A, B] = (A \cup B, E')$ where $E' \subseteq E$ are the edges with one endpoint in A and one endpoint in B. Note that $A \subseteq V$ defines the induced bipartite subgraph $G[A, \overline{A}]$.

Tree A graph $T = (V, E)$ is called a tree if T is connected and contains no cycles. We name the set V nodes to distinct a tree from a regular graph. A node $v \in V$ 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 is either a leaf or has two children.

2.2 Running time analysis

With respect to the running time analysis and corresponding notation, we define the following. Let $G = (V, E)$ be a graph with n vertices. For functions f and g we say

- $f(n) \in O(g(n))$ if there exist c and n_0 such that for all $n > n_0$ we have $f(n) \leq c \cdot g(n)$.
- $f(n) \in O^*(g(n))$ if there exist a polynomial *poly*, such that $f(n) \in O(g(n) \cdot poly(n))$.
- $f(n) \in \tilde{O}(g(n))$ if there exist a polylogarithmic expression *logpoly*, such that $f(n) \in O(g(n) \cdot logpoly(n))$.

So in fact O, O^* and $\tilde{O}$ suppress constant, polynomial and polylogarithmic expressions, respectively, in asymptotic running times.

Let k be a parameter of G. When we measure the running time of an algorithm as a function of both n and k we call it a parameterized algorithm. A parameterized algorithm is called *fixed parameter tractable* (FPT) parameterized by k if there exists a function f and a polynomial function *poly* such that the algorithm finishes in time $f(k)poly(n)$.

The algorithms in this chapter 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 chapter improve by a factor n.

2.3 Decompositions of a graph

As mentioned in the introduction, we need a derived structure which captures the necessary information of a graph in order to exploit the parameter boolean-width. In the case of boolean-width, this is a binary partition tree, referred to as the decomposition tree. In other literature this is also referred to as "branch decomposition". Please note that for treewidth the derived structure is called “tree decomposition”, which is very different from the decomposition trees that we are talking about here.

Definition 2.1 (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 leaves of T and vertices of V. If a is a node and L are its leaves, we write $\delta(a) = \bigcup_{l \in L} \delta(l)$. So, 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 $\delta(w)$. Alternatively, we sometimes write V_w for $\delta(w)$.

Consider the example graph in Figure 2.1. An example of a decomposition of this graph can be found in Figure 2.3.

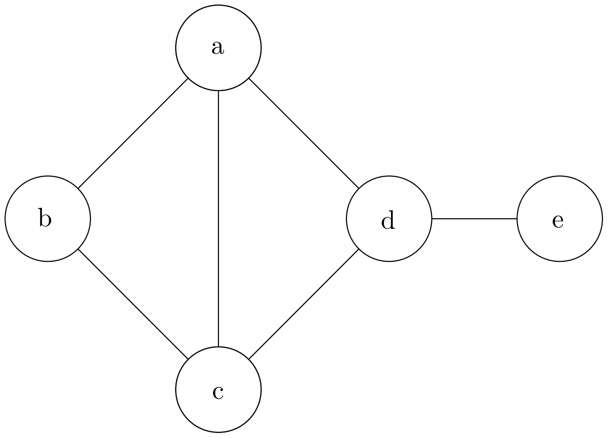


Figure 2.1: Example graph

Not every decomposition is useful. What makes a decomposition good? Please note that removing an edge in the decomposition tree results in two subtrees, inducing a partition $(A, \overline{A})$ of $V(G)$, which we will call a *cut*. Given a function that assigns a value to each cut, we want to find a decomposition that minimizes these values.

Definition 2.2 (cut function)**.** A symmetric function $f : 2^V \to \mathbb{R}_{\geq 0}$: $f(A) = f(\overline{A})$ for all $A \subset V(G)$ is called a *cut function*.

Definition 2.3 (f-width)**.** Let f be a cut function. The *f-width* of (T, δ) is the maximum value of $f(A)$ over all cuts $(A, \overline{A})$ of G, i.e. $\max_{(A,\overline{A})} f(A)$. The *$f$-width* of G is the minimum f-width over all possible decomposition trees of G.

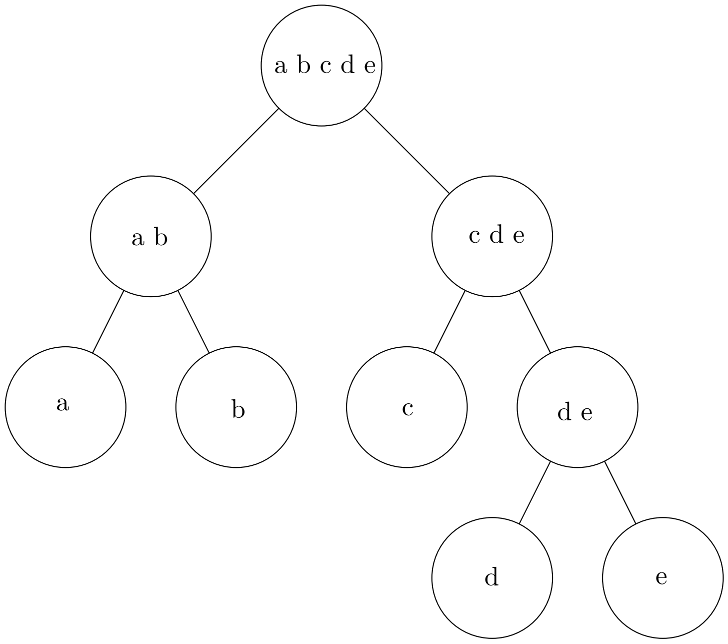


Figure 2.2: Example decomposition

Boolean-width is a graph parameter introduced by Bui-Xuan et al. [8] It minimizes the number of possible neighborhoods across a cut, for each cut in the decomposition.

Definition 2.4 (neighborhood across a cut)**.** Let G be a graph and $A \subseteq V(G)$. For any $X \subseteq A$ define the neighborhood of X across the cut $(A, \overline{A})$ as $\bigcup_{x \in X} N(x) \cap \overline{A}$.

Definition 2.5 (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 is symmetric for a cut $(A, \overline{A})$, i.e., $|\mathcal{UN}(A)| = |\mathcal{UN}(\overline{A}|)$ [15, Theorem 1.2.3].

Definition 2.6 (bool-dim)**.** Let G be a graph and A subset $V(G)$. Define the function bool-dim : $2^V \to \mathbb{N}, A \mapsto \log_2 |\mathcal{UN}(A)|$. The latter function is pronounced as *boolean dimension*.

Definition 2.7 (boolean-width)**.** Let G be a graph. Define the *boolean-width* of G, written $\mathrm{boolw}(G)$, using definition 2.3 with f being bool-dim.

In Figure 2.3, the bool-dim value for each cut has been printed. As one can see, for small graphs this is not very interesting, mostly because the two cuts of the root are always the same and because cuts at leaves always have value 1.

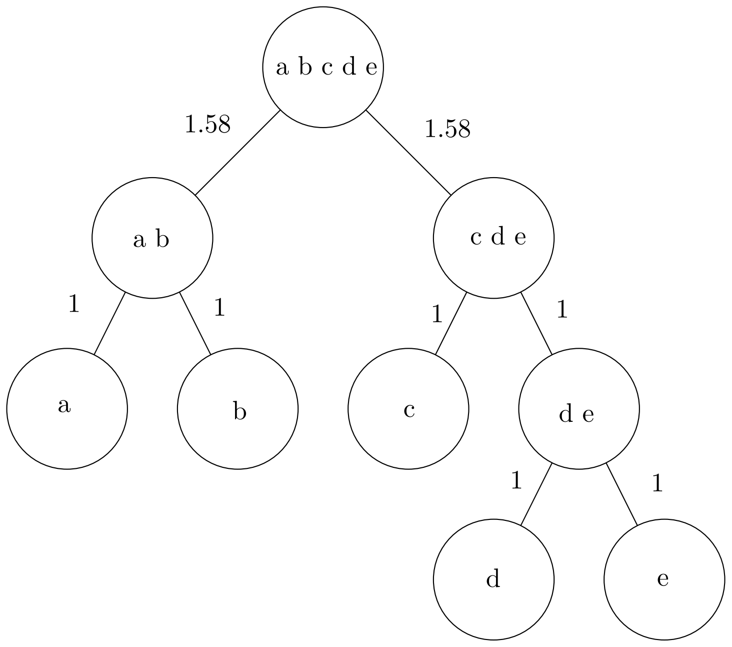


Figure 2.3: Boolean-width values for each cut of the example decomposition

For any graph G let $\mathcal{MIS}(G)$ be the collection of maximal independent sets in G. The following correspondence proves to be very useful. It was first pointed out by Nathann Cohen, as stated in Vatshelle's PhD thesis [33, Theorem 3.5.5].

Lemma 2.8. *Let $G = (V, E)$ be a graph and $A \subseteq V$. Then $|\mathcal{MIS}(G[A, \overline{A}])| = |\mathcal{UN}(A)|$.*

Proof. We show that there is a bijection between $\mathcal{UN}(A)$ and the maximal independent sets of $G[A, \overline{A}]$. For every $S \in \mathcal{UN}(A)$ we define $M(S) = \{v \in A : N(v) \cap A \subseteq S\}$ the unique maximal subset of A having S as neighborhood. Clearly $I = M(S) \cap A \setminus S$ is an independent set in $G[A, \overline{A}]$. Since

$M(S)$ is maximal, every vertex in $A \setminus I$ must have a neighbor in $A \setminus S$, hence no vertices in $A \setminus I$ can be added to I. Since $A \setminus I = S$, no vertices in A can be added to I. This shows that I is a maximal independent set of $G[A, \overline{A}]$ and hence $|\mathcal{UN}(A)| \leq |\mathcal{MIS}(G[A, \overline{A}])|$. For the other direction, for every maximal independent set I in $G[A, \overline{A}]$ we know that $N(I \cap A) = A \setminus I$, otherwise I would not be a maximal independent set. Hence $A \setminus I \in \mathcal{UN}(A)$, showing that there is a bijection between $\mathcal{UN}(A)$ and the maximal independent sets of $G[A, \overline{A}]$. □

2.4 Examples

In this section we give some trivial propositions to make the reader familiar with the introduced concepts.

Proposition 2.9. *Let $G = (V, E)$ be a graph with n vertices. If G is complete, i.e. a clique, then* $\operatorname{boolw}(G) = 1$.

Proof. Because G is complete we have that $N(x) = V \setminus \{x\}$ for any $x \in V$. From this it follows that $\mathcal{UN}(A) = \left\{\overline{A}\right\}$ and thus $\text{bool-dim}(A) = 1$ for any $\emptyset \neq A \subsetneq V$. Therefore we have $\operatorname{boolw}(G) = \max_{A \subseteq V} \text{bool-dim}(A) = 1$. □

A graph G is called maximal bipartite if no edge can be added without destroying its bipartiteness.

Proposition 2.10. *Let G be a graph with n vertices. If G is maximal bipartite, then* $\operatorname{boolw}(G) = 1$.

Proof. Let A and $\overline{A}$ be the two subsets forming the bipartite graph G. Because G is maximal bipartite we have that

$$N(x) = \begin{cases} \overline{A} & \text{if } x \in A \\ A & \text{if } x \in \overline{A} \end{cases} .$$

So we can take $(A, \overline{A})$ as the first cut in the decomposition, and randomly decompose the remaining part. Namely, $\text{bool-dim}(A) = 1$ and for any $\emptyset \neq B \subsetneq A$ we have $\text{bool-dim}(B) = 1$ as well. Therefore $\operatorname{boolw}(G) = \max_{A \subseteq V} \text{bool-dim}(A) = 1$. □

Proposition 2.11. *Let $G = (E, V)$ be a graph, $A \subseteq V$ and $x \in V$. Then the neighborhood of x across the cut $(A, \overline{A})$ in G is complementary to that in $\overline{G}$, i.e. $N_{\overline{G}}(x) \cap \overline{A} = \overline{N_G(x)} \cap \overline{A}$.*

Proof.

$$\begin{aligned} N_{\overline{G}}(x) \cap \overline{A} &= \left\{ y \,\middle|\, (x,y) \in E(\overline{G}) \right\} \cap \overline{A} \\ &= \{ y \mid (x,y) \notin E(G) \} \cap \overline{A} \\ &= \overline{\{ y \mid (x,y) \in E(G) \}} \cap \overline{A} \\ &= \overline{N_G(x)} \cap \overline{A}. \end{aligned}$$

□

However, for larger subsets the above result does not hold, i.e. it is not necessarily true that $N_{\overline{G}}(X) \cap \overline{A} = \overline{N_G(X)} \cap \overline{A}$, for $X \subseteq A \subseteq V$. For this reason, the boolean-width of a graph is in general not equal to the boolean-width of its complementary graph.

Chapter 3

Boolean-width

In this chapter we give brief introduction about existing algorithms concerning boolean-width. We will skip the heuristics developed by Sharmin [26], since those will be treated in Chapter 4.

3.1 Complexity

What can we say about the complexity of the problem of computing the boolean-width of a graph? Let us refer to this problem by $\mathcal{BOOLW}$. It is proven by Sigve Hortemo Sæther and Martin Vatshelle that $\mathcal{BOOLW}$ is NP-hard, though at the time of writing this result is not yet published [27]. So now that we know that $\mathcal{BOOLW}$ is at least as hard as any NP-complete problem, can we give a class of problems for which $\mathcal{BOOLW}$ is not harder? In other words, can we prove the membership of $\mathcal{BOOLW}$ of a certain complexity class?

Let C be some complexity class. If a problem is in NP^C, a non-deterministic Turing machine can decide it in polynomial time. So to prove the membership of a problem in a class NP^C, it suffices to show the existence of a polynomial size certificate that can be used to verify a solution in polynomial time on a deterministic Turing machine, using an oracle that decides problems in C.

Now let $G = (V, E)$ be a graph and let $f : \mathcal{P}(V) \to \mathbb{R}_{\geq 0}$ be some cut function. Suppose that to problem of computing $f(X)$ for some $X \subseteq V$ is member of C. We will show that the problem of computing the f-width of a graph is then member of NP^C.

Theorem 3.1. *If the problem of computing $f(X)$ for some $X \subseteq V$ is member of C, then computing the f-width of a graph of size n is a member of NP^C.*

Proof. A decomposition tree has $2n - 1 = O(n)$ nodes, where each node has to store sets of at most n vertices. Since all vertices can be labeled with a number less than n, only $O(\log n)$ space is needed to store a vertex. So the entire boolean decomposition only needs $O(n \cdot n \cdot \log n) = O(n^2 \log n)$ space, which is polynomial in n.

The decision problem we are facing is: given graph G and a number K, is the f-width of G less than K?

Suppose we are given a solution with a decomposition tree as certificate. We can compute f for the cut in each node in constant time using the $\#P$ oracle, and take their maximum M. This takes $O(n)$ time in total, which is polynomial in n. Now we can verify that the solution is correct by asserting that $M \leq K$. So we can verify a solution in polynomial time using the decomposition tree as the polynomial size certificate and by using the C-oracle to decide the problems in C. This completes the proof. □

Corollary 3.2. *The problem $\mathcal{BOOLW}$ is a member of the complexity class $NP^{\#P}$.*

Proof. It is known that counting the number of maximal independent sets in a graph is $\#P$-complete (even when restricted to chordal graphs) [19]. Therefore applying Theorem 3.1 with $C = \#P$ yields the required result. □

However, since $NP^{\#P}$ and NP are still different complexity classes, we are left to wonder for what complexity class C $\mathcal{BOOLW}$ is actually C-complete.

Open question 3.3. *For which complexity class C do we have that $\mathcal{BOOLW}$ is C-complete?*

3.2 Exact algorithms

Let $G = (V, E)$ be a graph of size n. In this section we look at exact algorithms to compute boolean-width of G. Trivially, we can establish a $O(3^n)$ algorithm. Consider the following recurrence relation.

$$\begin{aligned} w(\{v\}) &= 0 \\ w(A) &= \min_{\emptyset \neq B \subsetneq A} \{\max\{|\mathcal{UN}(B)|, |\mathcal{UN}(A \setminus B)|, w(B), w(A \setminus B)\}\} \end{aligned} \tag{3.1}$$

The final answer is then given by $\log_2 w(V)$.

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 [13]. Doing this for all A takes $O(2.7284^n)$ time. We solve recurrence relation (3.1) in a bottom-up fashion. For each iteration in the recurrence relation the minimum of $|2^A| - 1$ numbers has to be taken. Suppose $|A| = k$. Then this takes of course $O(2^k)$ time for each iteration. In solving the recurrence relation, k goes from 1 to n. Since there are $\binom{n}{k}$ subsets of size k, it takes

$$\sum_{k=1}^{n} \binom{n}{k} 2^k = O(3^n)$$

time to compute all values for boolw. The space requirements amount to $O(n \cdot 2^n)$, since bool-dim and boolw contain at most 2^n entries of integers of at most n bits.

The currently fastest known exact algorithm for boolean-width runs in $O^*(2.52^n)$ [33], but it is easy to translate this into a $O^*(2^{n+\text{boolw}(G)})$ algorithm. It makes use of the following algorithm by Oum [20].

Theorem 3.4. *Let V be a finite set with n elements. Let f be an integer-valued function defined on the subsets of V. Let $M = \max_{X \subseteq V} |f(X)|$. We assume that f is given by an oracle and each oracle call takes time $O(\alpha)$. Then we can compute the width of f on V exactly in time $O(2^n(n^3 \log n \log\log n \log M + \log^2 M + \alpha))$*

Theorem 3.5. *A boolean decomposition of minimum boolean-width for a graph G can be computed in $O(n^3 \cdot 2^{n+\text{boolw}(G)})$ 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 [10]. Given an upperbound K for the boolean-width, we can use the upperbound $2^K + 1$ as a limit for this algorithm, such that computing $\max(|\mathcal{UN}(A)|, K)$ takes at most $O(n^3 2^K)$ time. Namely, if the resulting boolean width is smaller than K, the decomposition only used cuts for which the upper bound was valid, so we are done. Otherwise, we report failure. By Theorem 3.4, we can now compute the boolean-width in $O(n^3 2^n 2^K)$.

To assure that K is indeed an upperbound, we perform an incremental search on K. Starting with $K = 1$, the procedure is executed boolw(G) times. The running time therefore is

$$O(\sum_{k=1}^{\lceil \text{boolw}(G) \rceil} n^3 2^{n+k}) = O(n^3 2^n (2^{\lceil \text{boolw}(G) \rceil + 1} - 1)) = O(n^3 2^{n+\text{boolw}(G)}).$$

The space requirements amount to $O(n \cdot 2^n)$, since the tables for $|\mathcal{UN}|$ and w contain at most 2^n entries of integers of at most n bits. □

It is known that for any cut function f that has the property that $f(A) \leq |A|$ for any $A \subseteq V$, the f-width is bounded by $\frac{n}{3}$. Namely, one can simply linearly decompose one third of the vertices in arbitrary order, after which one splits the remaining two third exactly in half. This makes sure that for any cut $(A, \overline{A})$ in the resulting decomposition it holds that $f(A) \leq \min(|A|, |\overline{A}|) \leq \frac{n}{3}$, since f is symmetric by definition. Thus, the f-width of this decomposition can be at most $\frac{n}{3}$, from the result follows. However, for boolean-width a slightly tighter bound has been found. Rabinovich et al. established that boolw$(G) \leq \frac{n}{3} - \frac{n}{672} + o(n)$ [22]. This makes the worst case performance of the algorithm in Theorem 3.5 equal to $O^*(2^{n+\frac{n}{3}-\frac{n}{672}+o(n)}) \subseteq O^*(2.5173^n)$.

3.2.1 Generating a boolean decomposition from a tree decomposition

It is possible to create a boolean decomposition D from a tree decomposition D' with the assertion that boolw$(D) \leq$ tw(D') [26, Section 6.2]. But can we also speed up the creation of an optimal boolean decomposition using a good tree decomposition? We propose a simple initial way to do this. Remember that a graph has degeneracy k if every subgraph has a vertex of degree at most k. Eppstein et al. established that for a k-degenerate graph, all maximal independent sets can be listed in $O^*(3^{k/3})$ time [11].

Theorem 3.6. *If a graph has tree-width k, then an optimal boolean-decomposition can be computed in $O^*(2^{n+0.528k})$.*

Proof. If a graph has treewidth k, then the degeneracy is at most k [24]. As stated above, the number of MIS can be computed in $O^*(3^{k/3})$. Using Theorem 3.4 the total running time becomes $O^*(2^n 3^{k/3}) \approx O^*(2^{n+0.528k})$. □

So if the boolean-width is more than about one half of the tree-width, this last algorithm is faster. It is probably not useful in practice, because it is likely that tree-width based algorithms are more suitable than boolean-width based algorithms as soon as the tree-width is almost as low as the boolean-width, since many application algorithms have a better running time in terms of tree-width.

3.3 Applications

What does a typical application of boolean-width look like? We consider two well-known problems: Maximum Independent Set and Minimum Dominating Set. Remember that an application algorithm assumes a decomposition of the graph, and solves the problem in a running time depending on the width of the decomposition. In both example algorithms, the sets of partial solutions attached to two siblings in the decomposition tree are merged to form again a set of partial solutions for the parent. Doing this recursively, starting with the leaves, should yield a set of solutions for the root of the decomposition tree, which in turn should give the final solution to the problem.

3.3.1 Computing Maximum Independent Set from a decomposition

The Maximum Independent Set problem is defined as finding the largest subset of vertices in the graph for which no two vertices are adjacent to each other. Let $(B, A \setminus B)$ be two children of a cut in the decomposition, as visualized in Figure 3.1. Let MIS_B contain all candidate maximum independent sets in the set B and similarly for $MIS_{A\setminus B}$. By this we mean that MIS_B contains exactly those independent subsets which could eventually be part of the final solution. All other subsets are ruled out by a reasoning which is yet to be explained. Given MIS_B and $MIS_{A\setminus B}$, we want to construct MIS_A. If two independent sets have the same neighborhood across $\overline{A}$ we only have to store the larger of them. Therefore, MIS_A is bounded by $|\mathcal{UN}(A)|$. We simply merge MIS_B and $MIS_{A\setminus B}$ into MIS_A by forming all possible combinations. This takes thus $O(|\mathcal{UN}(B)| \cdot |\mathcal{UN}(A \setminus B)|)$ time.

Since for any cut $(X, \overline{X})$ in the decomposition $|\mathcal{UN}(X)|$ is always bounded by the boolean-width of the decomposition that we are given, each merging step takes $O(n^2 4^k)$ time, and since there are only $O(n)$ node in the decomposition, the time for the entire algorithm is $O(n^3 4^k)$. See Algorithm 1 for corresponding pseudo code. In the pseudocode MIS_A is implemented as a mapping from neighborhoods across $(A, \overline{A})$ to the corresponding candidate independent set. This is to make it easy to rule out duplicate neighborhoods and only keep the largest candidate for each neighborhood.

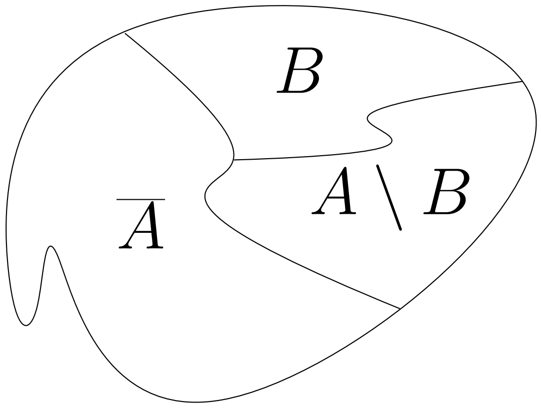


Figure 3.1: Venn diagram of a merge step in an algorithm.

Algorithm 1 Compute Maximum Independent Set given a decomposition D.

```
1:  procedure Compute-MIS-from-decomposition($V, D$)
2:      return Recursion($V, D$)[$\emptyset$]
3:  procedure Recursion($A, D$)
4:      $(B, D_1), (A \setminus B, D_2) \leftarrow$ pop the root from $D$
5:      $MIS_B \leftarrow$ Compute-MIS-from-decomposition($B, D_1$)
6:      $MIS_{A \setminus B} \leftarrow$ Compute-MIS-from-decomposition($A \setminus B, D_2$)
7:      $MIS_A \leftarrow$ mapping from all neighborhoods across $(A, \overline{A})$ to $\emptyset$
8:      for $I_1 \in MIS_B$ do                         ▷ Enumerate values, not keys
9:          for $I_2 \in MIS_{A \setminus B}$ do
10:             $I \leftarrow I_1 \cup I_2$
11:             if $I$ is independent set and $|MIS_A[N(I) \cap \overline{A}]| < |I|$ then
12:                 $MIS_A[N(I) \cap \overline{A}] \leftarrow I$
13:     return $MIS_A$
```

3.3.2 Computing Minimum Dominating Set from a decomposition

The Minimum Dominating Set problem is defined as finding the smallest subset S of vertices in the graph such that all vertices in the graph are adjacent to at least one vertex in S. Again let $(B, A \setminus B)$ be two children of a cut in the decomposition, as visualized in Figure 3.1. For $R \subseteq B$, let $MDS_B[R]$ denote all partially dominating sets in the set B which dominate everything in B, except R. Similarly define $MDS_{A\setminus B}[R]$. Note that these partial solutions are only useful if $R \subseteq N(\overline{A})$, since the remaining undominated vertices can never be dominated by vertices from $\overline{A}$. We want to merge MDS_B and $MDS_{A\setminus B}$ to form MDS_A, for all relevant R.

So let $R \subseteq N(\overline{A}) \cap A$. If two partially dominating sets in $MDS_A[R]$ have the same neighborhood across $\overline{A}$, we only have to store the smallest of them. Therefore, $MDS_A[R]$ is bounded by $|\mathcal{UN}(A)|$. So for each cut $(A, \overline{A})$ we have $|\mathcal{UN}(\overline{A})|$ possible subsets R, each of which corresponds to at most $|\mathcal{UN}(A)|$ possible partially dominating sets. Thus, $|MDS_A[R]|$ is bounded by $|\mathcal{UN}(A)| \cdot |\mathcal{UN}(\overline{A})|$. At each cut $(B, A \setminus B)$ in the decomposition, we merge $MDS_B[R]$ and $MDS_{A\setminus B}[R]$, for each $R \subseteq A$. Simply form all combinations and for all partially dominating sets with the same neighborhood across $(A, \overline{A})$ only keep the smallest. For each cut, this takes $O(|\mathcal{UN}(B)| \cdot |\mathcal{UN}(A \setminus B)| \cdot |\mathcal{UN}(\overline{A})|)$ time.

Since for any cut $(X, \overline{X})$ in the decomposition $|\mathcal{UN}(X)|$ is always bounded by the boolean-width of the decomposition that we are given, each merging step takes $O(n^2 8^k)$ time, and since there are only $O(n)$ node in the decomposition, the time for the entire algorithm is $O(n^3 8^k)$. See Algorithm 2 for corresponding pseudo code. In the pseudocode MDS_A is implemented as a mapping from undominated sets and neighborhoods across $(A, \overline{A})$ to the corresponding partially dominating set. This is to make it easy to rule out redundant solutions and only keep the smallest partially dominating set for each neighborhood.

The running times $O^*(4^k)$ and $O^*(8^k)$ were already established by Vatshelle [33]. We will come back to these applications in Chapter 6.

3.3.3 Vertex subset problems

The Maximum Independent Set and Minimum Dominating Set problems belong to a class of problems called the (σ, ρ) vertex subset problems, which

Algorithm 2 Compute Minimum Dominating Set given a decomposition D.

```
1: procedure Compute-MDS-from-decomposition($V, D$)
2:     return Recursion($V, D$)[$\emptyset, \emptyset$]
3: procedure Recursion($A, D$)
4:     $(B, D_1), (A \setminus B, D_2) \leftarrow$ pop the root from $D$
5:     $MDS_B \leftarrow$ Compute-MDS-from-decomposition($B, D_1$)
6:     $MDS_{A \setminus B} \leftarrow$ Compute-MDS-from-decomposition($A \setminus B, D_2$)
7:     $MDS_A \leftarrow$ mapping from all undominated $R \subseteq A$ and all neighbor-
   hoods across $(A, \overline{A})$ to $\emptyset$
8:     for $R \in \mathcal{UN}(\overline{A})$ do
9:         for $I_1 \in MDS_B[(R \setminus N(A \setminus B)) \cap B]$ do        ▷ Enumerate values
10:            for $I_2 \in MDS_{A \setminus B}[(R \setminus B) \cap (A \setminus B)]$ do
11:                $I \leftarrow I_1 \cup I_2$
12:                if $I$ dominates $A \setminus R$ and $|MDS_A[R, N(I) \cap \overline{A}]| > |I|$ then
13:                    $MDS_A[R, N(I) \cap \overline{A}] \leftarrow I$
14:    return $MDS_A$
```

were introduced by Telle [28]. 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.

Definition 3.7 ((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 3.8 (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 3.9 (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 3.10 (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 $|\mathcal{UN}(A)| = nec(\equiv_A^1)$ [33, Theorem 3.5.5] Also note that $X \equiv_A^d Y$ if and only if $N_A^d(X) = N_A^d(Y)$.

There has been developed a general algorithm for this entire class by Bui-Xuan et al. [1] The running time of the algorithm for solving these problems is $O(n^4 \cdot nec_d(T, \delta)^3)$, 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 4.5 we investigate how close the value of $nec_d(T, \delta)$ comes to any of the theoretical bounds.

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 3.11. *[1, Lemma 5]* $nec(\equiv_A^d) \leq 2^{d \cdot k^2}$.

Lemma 3.11 shows us that we can solve (σ, ρ) problems in $O^*(8^{dk^2})$. In next chapters we come back to this.

Chapter 4

Linear Boolean-width

In this chapter, 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.

4.1 Introduction

Algorithms for computing boolean decompositions have been studied before in [33, 22, 26, 14], but in this chapter we study the specific case of linear boolean decompositions, which are considered in [2, 22, 26]. 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. [1], but this can be improved to $O^*(nec_d(T, \delta)^2)$ for linear decompositions. Here, $nec_d(T, \delta)$ is the number of d-neighborhood equivalence classes, i.e., the maximum size of the dynamic programming table, as defined in Section 3.3.3. This can also be easily observed in the algorithms for Maximum Independent Set and Minimum Dominating Set that are described in Section 3.3. Namely, when working with a linear decomposition, always one of the branches in a cut is bounded by 1 in size, which implies that the number of unions of neighborhoods across the cut is bounded by two. Therefore the entire running time improves by a factor 2^k.

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 [26, 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 compared to the theoretical worst case bounds.

4.2 Preliminaries

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

Definition 4.1 (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.

An example of a linear decomposition can be found in Figure 4.1. This is in fact a possible decomposition of the example graph in Figure 2.1 that we encountered before.

Definition 4.2 (linear boolean-width)**.** The boolean-width of G is defined as the 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.

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

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

The restriction to linear decompositions makes linear boolean-width really a different parameter. There are graphs for which the boolean-width is low, but the linear boolean-width is very high. For instance, we will see in

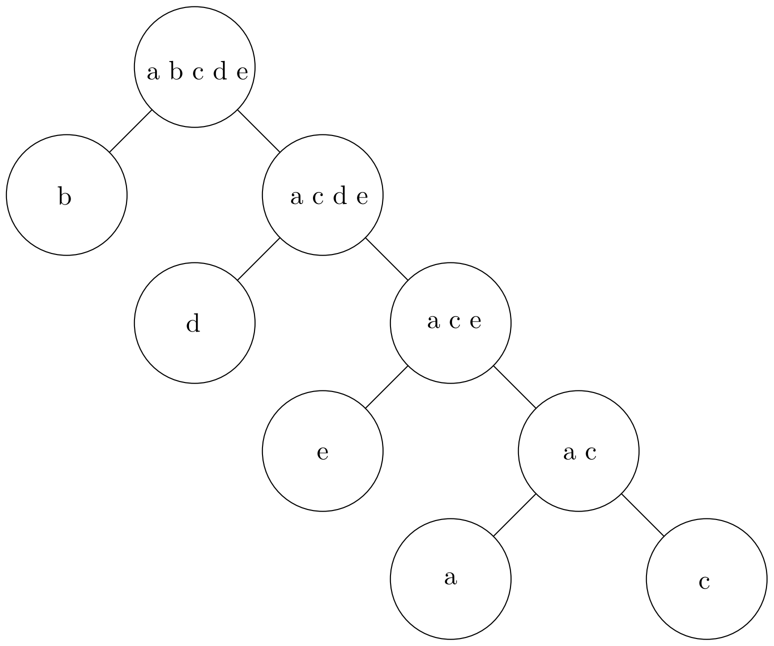


Figure 4.1: Example of a linear decomposition

Chapter 5 that trees have boolean-width 1. A corresponding decomposition can be constructed in linear time. However, trees do generally have a very high linear-boolean width. It is not known whether there is a polynomial time algorithm to compute the linear boolean-width of trees.

Open question 4.3. *Does there exists a polynomial time algorithm to compute linear boolean-width on trees?*

It is also an open problem whether computing linear boolean-width on a general graph is NP-hard, although it is strongly suspected.

Open question 4.4. *Is computing linear boolean-width NP-hard?*

In this chapter we assume that the graphs of concern are 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.

4.3 Exact Algorithms

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

$$\begin{aligned} w(\{v\}) &= 0 \\ w(A) &= \min_{v \in A}\{\max\{|\mathcal{UN}(A \setminus \{v\})|, w(A \setminus \{v\})\}\} \end{aligned} \tag{4.1}$$

The linear boolean-width of the graph G is now given by $\log_2(w(V))$. 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 4.5)
- With adaptation of the exact algorithm for boolean-width by Vatshelle [33], a running time of $O(n^3 \cdot 2^{n+\text{lboolw}(G)})$ is achieved. (See Theorem 4.6)

Theorem 4.5. *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 [13]. Doing this for all A takes $O(2.7284^n)$ time.

We solve recurrence relation (4.1) 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 $O(n^3 2^{n+\text{boolw}(G)})$, see Theorem 3.5. Theorem 4.6 is a direct adaptation to linear boolean-width.

Theorem 4.6. *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 $O(n2^n)$ 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 [10]. 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 (4.1). This can be solved in $O(n \cdot 2^n)$ time by the same reasoning as in Theorem 4.5. 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. □

4.3.1 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 4. The observation is that once we know a union neighborhood of some cut A, then it should not be hard to construct the union of neighborhoods in $A \cup \{v\}$. We refer to this as “incrementing the union of neighborhoods”. In the following it is important that the $\mathcal{UN}$ sets are implemented as hashmaps, which will only save distinct neighborhoods.

Algorithm 3 Compute $\mathcal{UN}(X \cup \{v\})$ given $\mathcal{UN}(X)$.

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

Lemma 4.7. *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 $\mathcal{U}$, 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 $\mathcal{U}$. Assume for contradiction that S is a valid neighborhood not contained in $\mathcal{U}$. 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 $\mathcal{U}$, 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 $\mathcal{U}$, also contradicting our assumption. It follows that a neighborhood is contained in the set $\mathcal{U}$ if and only if it is a valid neighborhood across the cut $(X \cup \{v\}, \overline{X} \setminus \{v\})$.

The 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 size 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 $O(n \cdot 2^k)$. The space requirements amount to $O(n \cdot 2^k)$, for storing $\mathcal{U}$ which contains at most $O(2^k)$ sets of size at most $O(n)$. □

The reason for computing $T_{\mathcal{UN}}(Y)$ separately in a depth-first fashion is to keep the space requirements low. If it were computed on the fly while solving the recurrence relation, it would be evaluated in a breadth-first fashion, which would increase the space requirements to $O(n2^{n+K})$.

Lemma 4.8. *Given a graph $G = (V, E)$ of size n and an integer K, Algorithm 4 computes the linear boolean width, if it is at most $\log K$, in $O(nK2^n)$ time using $O(n2^n)$ 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 $T_{\mathcal{UN}}$ is always computed when necessary, since there may be X such that $T_{\mathcal{UN}}(X)$ is not computed, while $T_{\mathcal{UN}}(X) \leq K$. Suppose that $X \subseteq V$ of size i occurs in an optimal decomposition and $T_{\mathcal{UN}}(X)$ 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

Algorithm 4 Return lboolw(G), if it is smaller than $\log K$, otherwise return ∞.

```
 1: procedure INCREMENTAL-UN-EXACT($G, K$)
 2:     $T_{\mathcal{UN}}(\emptyset) \leftarrow 0$
 3:     COMPUTE-COUNT-UN($G, K, T_{\mathcal{UN}}, \emptyset, \{\emptyset\}$)
 4:
 5:     $w(X) \leftarrow \infty$, for all $X \subseteq V$
 6:     $w(\emptyset) \leftarrow 0$
 7:
 8:     for $i \leftarrow 0, \ldots, |V| - 1$ do
 9:         for $X \subseteq V$ of size $i$ do
10:             for $v \in V \setminus X$ do
11:                 $Y \leftarrow X \cup \{v\}$
12:                 if $w(X) \leq K$ then
13:                     $w(Y) \leftarrow \min(w(Y), \max(T_{\mathcal{UN}}(Y), w(X)))$
14:
15:     return $\log_2(w(V))$
16:
17: procedure COMPUTE-COUNT-UN($G, K, T_{\mathcal{UN}}, X, \mathcal{UN}_X$)
18:     for $v \in V \setminus X$ do
19:         $Y \leftarrow X \cup \{v\}$
20:         if $T_{\mathcal{UN}}(Y)$ is not defined then
21:             $\mathcal{UN}_Y \leftarrow$ INCREMENT-UN($G, X, \mathcal{UN}_X, v$)
22:             $T_{\mathcal{UN}}(Y) \leftarrow |\mathcal{UN}_Y|$
23:             if $T_{\mathcal{UN}}(Y) \leq K$ then
24:                 COMPUTE-COUNT-UN($G, K, T_{\mathcal{UN}}, Y, \mathcal{UN}_Y$)
```

implies that $T_{\mathcal{UN}}(X_j) \leq K$ 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 $T_{\mathcal{UN}}(X)$ must be computed, contradiction. Thus we conclude that $T_{\mathcal{UN}}$ 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 4.7.

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, $T_{\mathcal{UN}}(X)$ will be defined and line 20 prevents further calls with the same 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 4.7), 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 $O(n \cdot 2^{n+K})$.

For the space requirements, we observe that the tables $T_{\mathcal{UN}}$ 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 4.9. *Given a graph G, Algorithm 4 can be used to compute* $\text{lboolw}(G)$ *in $O(n2^{n+\text{lboolw}(G)})$ time using $O(n2^n)$ space.*

Proof. Iteratively double K in Algorithm 4, starting with $K = 1$, until it returns a number that is not ∞. By Lemma 4.8 this will take $O(\sum_{\log K=1}^{\text{lboolw}(G)} n \cdot 2^{n+\log K}) = O(n \cdot 2^{n+\text{lboolw}(G)+1}) = O(n2^{n+\text{lboolw}(G)})$ and take $O(n2^n)$ space. □

This new algorithm improves upon the time in Theorem 4.6 by a factor n^2, while the space requirements stay the same. Since the tightest known upperbound for linear boolean-width is $\frac{n}{2} - \frac{n}{143} + O(1)$ [22], 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.

4.3.2 Applying the same technique for boolean-width

If we apply a similar technique for the exact algorithms for boolean-width, we indeed get a new working algorithm as well. But it appears not to work so great for boolean-width as it does for linear boolean-width, since unions of neighborhoods are not simply incremented, but formed by combining two unions of neighborhoods. Moreover, it is not as easy to implement, since the subset convolution algorithm as to be integrated as well. See Appendix A for the corresponding pseudo code.

Remark 4.10. Given a graph G, Algorithm 9 can be used to compute $\operatorname{boolw}(G)$ in $O^*(2^{n+2\operatorname{boolw}(G)})$ time.

4.4 Heuristics

4.4.1 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 [26, Chapter 8], and a similar approach is shown in Algorithm 5.

Algorithm 5 Greedily generate an ordering based on the score function and the given starting vertex.

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

At any point in the algorithm we denote the set of all vertices 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 $Candidates \subseteq Right$, 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 $Left, Right$.

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 search. However, the fact that an arbitrary vertex is used for the first BFS search 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 later on 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.

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.

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.

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 4.11 generalizes results by Sharmin [26], 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 4.11. *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. $\square$

Preliminary experiments show that the order of pruning, limiting the candidate set and adding more trivial cases is very dependent on properties of a graph. For instance, taking the distance 2 neighborhood decreases the running time significantly for large graphs, but sometimes results in decompositions of higher width caused by a random difference in tie-breakers. Therefore we cannot give an optimal configuration in the general case; we simply tried to find a good configuration depending on the available information of the graphs uses as input.

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 [26] $\frac{|External(v)|}{|Internal(v)|}$ 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 4.12, 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 4.12. *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.

Least Cut Value Heuristic

The LeastCutValue heuristic by Sharmin [26] 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} [16] algorithm on BG, with the time of CCM_{IS} being exponential in n.

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 6. 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.

Algorithm 6 Greedy heuristic that incrementally keeps track of the Unions of Neighborhoods.

```
1:  procedure INCREMENTAL-UN-HEURISTIC($G, init$)
2:      $Decomposition \leftarrow (init)$
3:      $Left, Right \leftarrow \{init\}, V \setminus \{init\}$
4:      $\mathcal{UN}_{Left} \leftarrow \{\emptyset, N(init) \cap Right\}$
5:      while $Right \neq \emptyset$ do
6:          $Candidates \leftarrow$ set returned by candidate set strategy
7:          if there exists $v \in Candidates$ belonging to a trivial case then
8:              $chosen \leftarrow v$
9:              $\mathcal{UN}_{chosen} \leftarrow$ INCREMENT-UN($G, Left, \mathcal{UN}_{Left}, v$)
10:         else
11:             for all $v \in Candidates$ do
12:                 $\mathcal{UN}_v \leftarrow$ INCREMENT-UN($G, Left, \mathcal{UN}_{Left}, v$)
13:                 if $chosen$ is undefined or $|\mathcal{UN}_v| < |\mathcal{UN}_{chosen}|$ then
14:                     $chosen \leftarrow v$
15:                     $\mathcal{UN}_{chosen} \leftarrow \mathcal{UN}_v$
16:         $Decomposition \leftarrow Decomposition \cdot chosen$
17:         $Left \leftarrow Left \cup \{chosen\}$
18:         $Right \leftarrow Right \setminus \{chosen\}$
19:         $\mathcal{UN}_{Left} \leftarrow \mathcal{UN}_{chosen}$
20:     return $Decomposition$
```

Theorem 4.13. *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 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 $O(n \cdot 2^{|\mathcal{UN}_{Left}|})$ time by

Lemma 4.7. By definition $|\mathcal{UN}_{Left}|$ never exceeds the boolean-width k of the resulting decomposition, and 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 $O(n \cdot 2^k)$. □

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.

4.5 Experiments

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 C# programming language and compiled our programs using the *csc* compiler that comes with Visual Studio 12.0. For readability purposes only parts of the experiment data have been put inline. For all experiment data please refer to Section 4.7.

4.5.1 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 [12] 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 $N^2(Left) \cap Right$, which avoids that we exclude certain optimal solutions, as opposed to Sharmin [26], who restricted this set to $N(Left) \cap Right$. However, this does not affect the results significantly.

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 4.2. 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.

Because of feasibility limitations, the Incremental-UN-exact algorithm is not used for the graphs in Figure 4.3. While the optimal values are now unknown, it is clear that Incremental-UN-heuristic outperforms all other heuristics. Interestingly enough, RelativeNeighborhood$_3$ peers with Incremental-UN-heuristic as soon as the edge probability exceeds 0.4. Moreover, RelativeNeighborhood and 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 RelativeNeighborhood$_3$ when dense graphs are encountered.

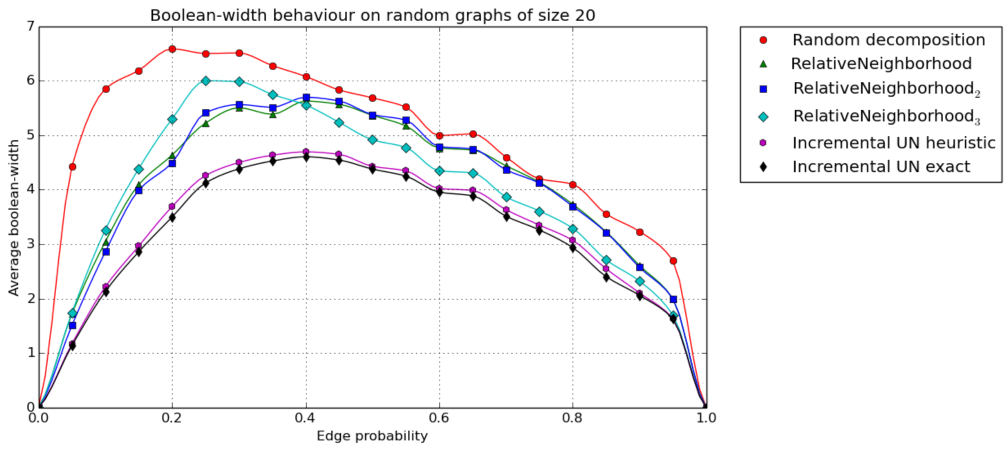


Figure 4.2: Performance of different heuristics on random generated graphs consisting of 20 vertices, with varying edge probabilities, in terms of linear boolean-width.

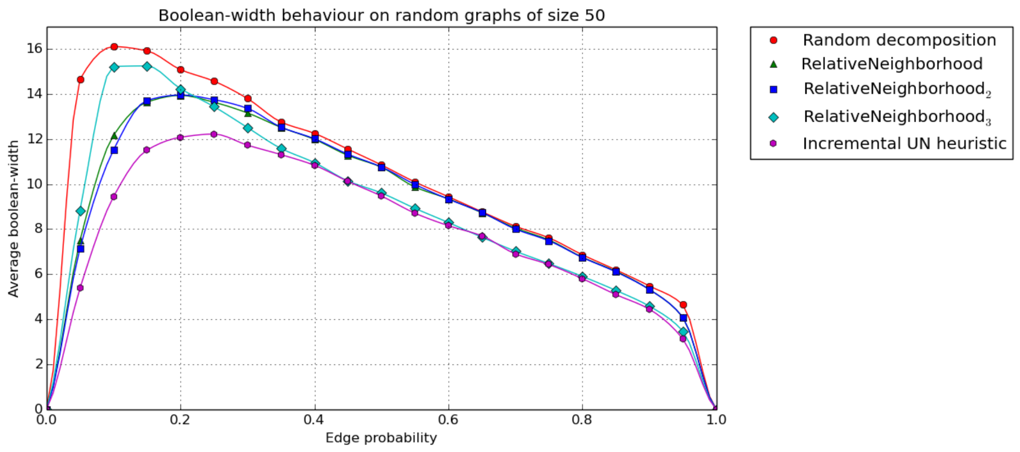


Figure 4.3: Performance of different heuristics on random generated graphs consisting of 50 vertices, with varying edge probabilities.

4.5.2 Comparing heuristics on real world graphs

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* [30], 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 4.4 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 search respectively. The n-IUN heuristic uses all possible start vertices. For all other heuristics we obtained the start vertex through performing a double BFS search. In Table 4.1 and 4.2 we present the results of our experiments.

Table 4.1: Linear boolean-width of the decompositions returned by different heuristics.

Graph	$\|V\|$	Edge Density	Relative	LeastCut	IUN	2-IUN	n-IUN
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
david	87	0.11	9.38	6.27	6.27	6.27	5.86
celar04-pp	114	0.08	11.67	7.27	7.27	7.27	7.27
1bkb-pp	127	0.18	16.81	9.98	9.98	9.53	9.53
miles1500	128	0.64	8.17	5.58	5.58	5.58	5.29
celar10-pp	133	0.07	10.32	11.95	11.95	7.64	6.91
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
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

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

Table 4.2: Time in seconds of the heuristics used to find linear boolean decompositions.

Graph	$\|V\|$	Edge Density	Relative	LeastCut	IUN	2-IUN	n-IUN
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
david	87	0.11	0.02	3.15	0.04	0.06	1.62
celar04-pp	114	0.08	0.04	5.73	0.14	0.23	9.85
1bkb-pp	127	0.18	0.06	198.05	1.14	4.18	107.32
miles1500	128	0.64	0.06	44.57	0.10	0.14	7.05
celar10-pp	133	0.07	0.06	8.93	1.96	4.72	18.43
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
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

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 three cases while n-IUN gives a better decomposition in 10 out of 12 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 4.2 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.

4.5.3 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 4.3. 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, 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 $ntc = \max\limits_{w \in T} ntc(V_w)$ and $\min ntc = \max\limits_{w \in T} \min(ntc(V_w), ntc(\overline{V_w}))$.

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 $\log_2(nec_2(T', \delta')) = 6.75$.

Table 4.3: Results of using the algorithm by Bui-Xuan et al. [1] 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)
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
david	5.86	9.37	68.63	22.19	44.61	34	919
celar04-pp	7.27	11.15	105.61	28.53	65.74	-	-
1bkb-pp	9.53	-	181.47	52.30	98.49	-	-
miles1500	5.29	9.30	55.87	34.87	49.69	8	4038
celar10-pp	6.91	10.34	95.41	25.36	59.70	50	10179
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
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	-	-

Another interesting observation with respect to applicability of decompositions can be done in Table 4.8, which is placed in the next section. Here the linear boolean-width heuristics are compared to boolean-width heuristics by Sharmin [26]. We know that for sigma-rho problems with $d = 1$ the worst case running times are $O(2^{2lbw})$ for linear boolean-width and $O(2^{3bw})$ for boolean-width. We observe that this value is usually lower for linear boolean-width, which suggests that with the current heuristics it is better to use linear boolean decompositions for applications.

4.6 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 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. Although the timed results are not entirely comparable to Sharmin's observed data, we can still see that our new heuristic for linear boolean decompositions surpasses heuristics for tree shaped decompositions in both time and quality.

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., for a cut $(A, \overline{A})$ $nec_d(A) \neq nec_d(\overline{A})$, 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. We will consider new reduction rules for linear boolean-width in Chapter 5. While most reduction rules introduced by Sharmin [26] 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 [31]. Another topic of research is to compare the performance of vertex subset algorithms parameterized by boolean-width to algorithms parameterized by treewidth [32].

4.7 Obtained data

This section contains the full set of tables containing the experiment data.

Table 4.4: Linear boolean-width of the decompositions returned by the heuristics described in Section 4.4, with *Candidates* = *Right*. For 2-IUN we use two start vertices; one is obtained through a single BFS search, while the other is obtained through a double BFS search. The n-IUN heuristic uses all n start vertices, and all other heuristics use start vertices obtained through performing a double BFS.

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

Continued on next page

Table 4.4 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	Relative	LeastCut	IUN	2-IUN	n-IUN
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

Table 4.5: Time in seconds of the heuristics used to find the linear boolean decompositions of which the boolean-width is displayed in Table 4.4.

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

Continued on next page

Table 4.5 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	Relative	LeastCut	IUN	2-IUN	n-IUN
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

Table 4.6: Results of using the algorithm by Bui-Xuan et al. [1] for solving (σ, ρ) problems on graphs, using decompositions obtained using the IUN heuristic using all starting vertices. The columns UB indicate theoretical upperbounds on the number of equivalence classes, with $UB_1 = 2^{d \cdot \text{boolw}^2}$, $UB_2 = (d+1)^{\min ntc}$ and $UB_3 = ntc^{d \cdot \text{boolw}}$, with $ntc = \max_{w \in T} ntc(V_w)$ and $\min ntc = \max_{w \in T} \min(ntc(V_w), ntc(\overline{V_w}))$.

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	-	-

Continued on next page

Table 4.6 – *Continued from previous page*

Graph	boolw	$\log_2(nec)$	$\log_2(UB_1)$	$\log_2(UB_2)$	$\log_2(UB_3)$	MIM	Time (s)
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 4.7: Width of linear boolean decompositions found with the IUN heuristic using the start vertices returned by performing a double BFS, and with $candidates = N^2(Left) \cap Right$ in order to decrease the computation time. The values of the two others heuristics are taken from [26]. Missing entries are caused by a lack of internal memory which is caused by the $O(n \cdot 2^k)$ space requirement, with k being the linear boolean-width of the computed decomposition. The last column indicates the time of the IUN heuristic.

Graph	$\|V\|$	Edge Density	LeastUnc	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

Table 4.8: Linear boolean-width upperbounds that are obtained through using the IUN heuristic with all starting vertices and *candidates* = *Right*. The *tw* column gives an upperbound on the treewidth, while the *bw* column gives an upperbound on the boolean-width, which values are taken from [26]. 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 $2^{2lbw} > 2^{3bw}$.

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
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

Continued on next page

Table 4.8 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	tw	bw	lbw	lbw/bw
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
BN_16-pp-015	34	0.28	11	3.58	4.39	1.23

Continued on next page

Table 4.8 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	*tw*	*bw*	*lbw*	*lbw/bw*
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

Continued on next page

Table 4.8 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	tw	bw	lbw	lbw/bw
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
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

Continued on next page

Table 4.8 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	*tw*	*bw*	*lbw*	*lbw/bw*
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
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

Continued on next page

Table 4.8 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	*tw*	*bw*	*lbw*	*lbw/bw*
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
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

Continued on next page

Table 4.8 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	*tw*	*bw*	*lbw*	*lbw/bw*
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 4.8 – *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

Continued on next page

Table 4.8 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	tw	bw	lbw	lbw/bw
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

Chapter 5

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 4. 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 5.1, 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.

We start this chapter with a number of definitions in Section 5.1. In Section 5.2 we explain how reduction rules can be found and proven to be correct. In Section 5.3 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 5.4 we look at reduction rules for treewidth and

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

Figure 5.1: Steps for applying reduction rules.

check their validity for boolean-width. In Section 5.5 we provide a number of new reduction rules. In Section 5.6 we present ways to combine properties of linear decompositions with reduction rules for general decompositions.

5.1 Definitions

Definition 5.1 (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 5.2 (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 the property of boolean-width 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 5.3 (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 5.4 (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 5.5 (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.

5.2 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 5.6. *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 5.6.

In order to disprove existing reduction rules, we make use of the following observation that follows directly from Definition 5.2.

Observation 5.7. *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 $\text{lboolw}(\pi) = \text{lboolw}(\pi')$.

5.3 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 [26, 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.

5.3.1 Islet rule

Lemma 5.8. *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. □

5.3.2 Pendant rule

Lemma 5.9. *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 5.2. 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. □

5.3.3 Twin rule

Lemma 5.10. *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 5.3. 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 5.2, of which the boolean-width is approximately 1.58. If $\{c, x\} \in E(G)$,

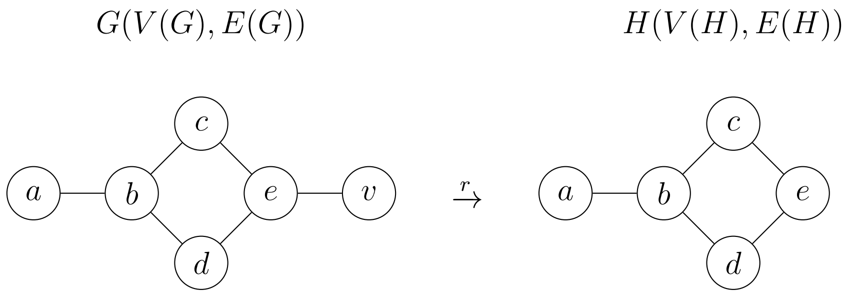


Figure 5.2: Counterexample to the pendant rule. The linear ordering (a, b, e, c, d) of $V(H)$ would have no safe expansion method to obtain a linear ordering of $V(G)$.

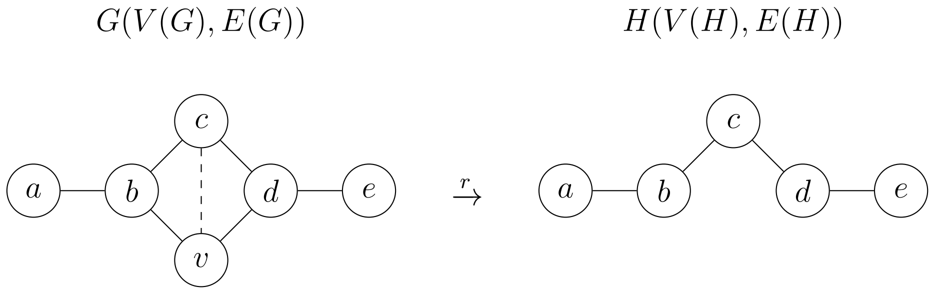


Figure 5.3: Counterexample to the twin rule. The linear ordering (a, b, c, d, e) would have no safe expansion method to obtain a linear ordering of $V(G)$.

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 $\text{lboolw}(H) < \text{lboolw}(G)$. $\square$

Following from Lemma 5.8, 5.9 and 5.10, 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 5.11. *The islet rule is a safe reduction rule for linear boolean-width.*

5.4 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 [26], 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 5.12. (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 5.4 for an example. Note that vertices of degree 2 are always almost simplicial by definition.

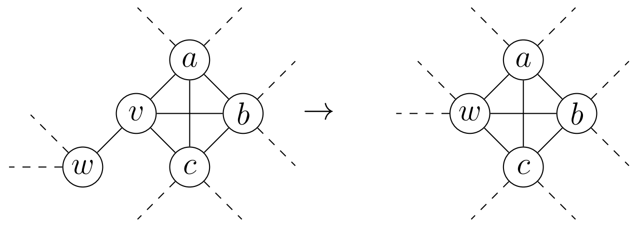


Figure 5.4: Application of the almost simplicial reduction rule for tree-width.

We present a counterexample to the almost simplicial rule for (linear) boolean-width, shown in Figure 5.5. 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.

Definition 5.13. (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.

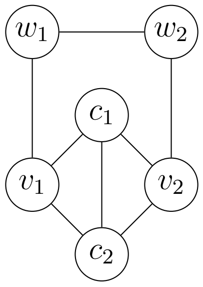


Figure 5.5: Counterexample to the almost simplical vertex rule.

Proposition 5.14. *[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 5.6 and Figure 5.7 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.

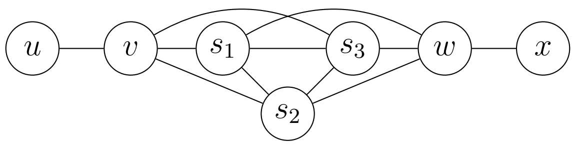


Figure 5.6: Counterexample to the clique separation rule.

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 [26]. This property

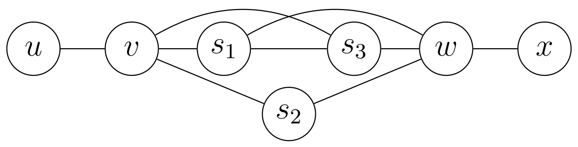


Figure 5.7: Counterexample to the almost clique separation rule.

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.

5.5 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 5.6 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 $(A \cup \{v\}, A \setminus \{v\})$.

5.5.1 Sequence rule

Definition 5.15 (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 5.8.

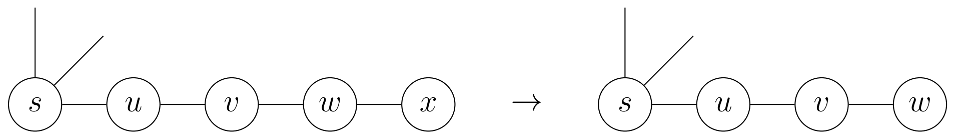


Figure 5.8: The sequence rule applied to a graph G.

Lemma 5.16. *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 5.15. 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 5.17 (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 5.15. We first construct the decomposition π'' for which $\text{lboolw}(\pi'') \leq \text{lboolw}(\pi')$ by applying the rearrangement technique described in Lemma 5.16 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 $\text{bool-dim}(\omega''_{i-1}) = \text{bool-dim}(\omega_i)$, 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 $\text{lboolw}(\pi'') = \text{lboolw}(\pi)$.

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 $\text{bool-dim}(\omega''_i) = \text{bool-dim}(\omega_{i-1})$, 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 $\text{lboolw}(\pi'') = \text{lboolw}(\pi)$.

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

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.

5.5.2 Clique rule

Definition 5.18 (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 $V(H) = V(G) \setminus \{c_1, \dots, c_n\}$ and $E(H) = \{\{x, y\} \mid \{u, v\} \in E(G) \wedge u \neq c_1 \vee \dots \vee c_n\}$. An example is illustrated in Figure 5.9, where the separator S is the singleton $\{s\}$.

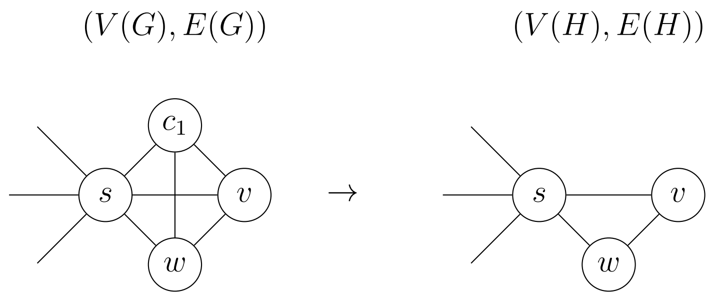


Figure 5.9: Clique rule applied to a graph G.

Theorem 5.19 (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 $s_1, \dots, s_{|S|}, v, w, c_1, \dots, c_n \in C \subseteq V(G)$ as described in Definition 5.18. 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 $c_1, \dots, c_n$ directly after v, meaning $\pi_{i+j} = c_j$ for $j = 1, \dots, n$. 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 bool-dim$(\omega_{i+j})$ = bool-dim(ω_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 lboolw(π) = lboolw(π'). We conclude that we can always construct a linear boolean decomposition π of G from π' of H while keeping the boolean-width equal. $\square$

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 $\text{lboolw}(\pi) \leq \text{lboolw}(\pi')$.

5.5.3 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.

5.6 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 4 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.

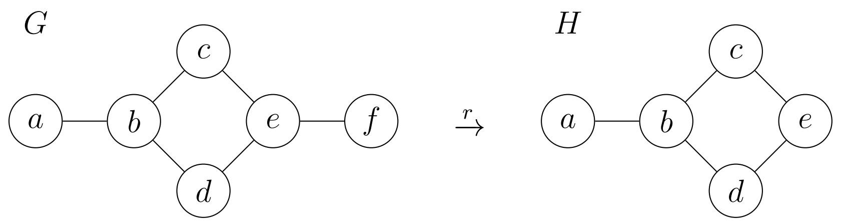
1. Start with a graph $G = (V(G), E(G))$.

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.

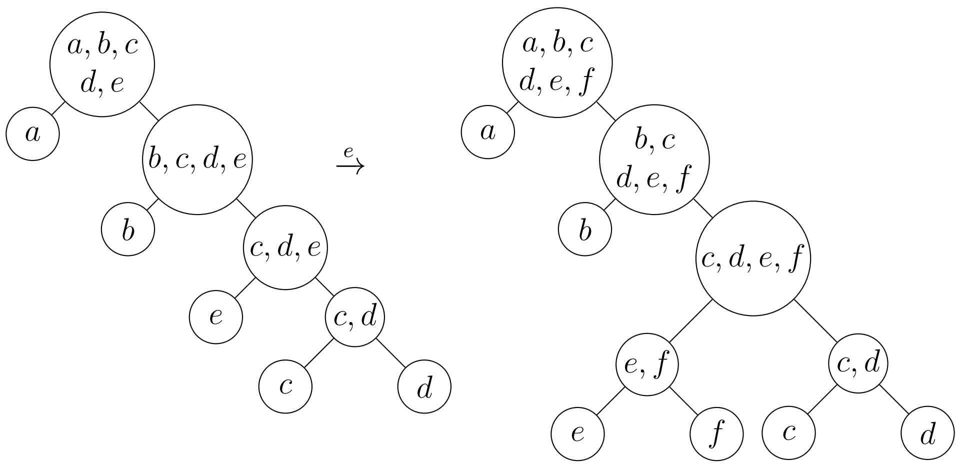
3. Use a heuristic on H to obtain a linear decomposition (T', δ').

4. Expand (T', δ') to a decomposition (T, δ) of G using expansion methods described in [26, 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 Section 3.3. 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 5.10 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 5.10: Application of the pendant rule.

5.7 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 6

Cost of decompositions

A large number of width measures of decompositions of graphs has been studied, e.g. rank-width, carving-width, branch-width, matching-width and boolean-width. These are defined as the maximum over all cuts defined by the decomposition of a function of the cut. If we instead take the sum of these values, we obtain cost variants, that may better reflect the running time of dynamic programming algorithms using the decompositions. Compare this to the notion of tree-cost, introduced by Bodlaender et al., which better reflects the time of algorithms using tree decompositions [7]. In this chapter, we give an exact algorithm for a large class of cut-functions f to compute a decomposition with minimum cost with respect to f. In the remaining of this chapter, let $G = (V, E)$ be a graph and let $n = |V|, m = |E|$.

Definition 6.1 (f-cost)**.** Let f be a cut function. The *f-cost* of (T, δ) is the sum of $f(A)$ over all cuts $(A, \overline{A})$ of G, i.e. $\sum_{(A,\overline{A})} f(A)$. The *$f$-cost* of G is the minimum f-cost over all possible decomposition trees of G.

It is an open problem whether computing f-cost is NP-hard for many functions f.

Open question 6.2. *Is computing boolean-cost NP-hard?*

6.1 Exact algorithm for f-cost

This problem is defined on the graph G and a function $f : \mathcal{P}(V) \to \mathbb{N}$ as the solution to the following recurrence relation. As you can see it is similar to the f-width problem, except that we sum the values instead of taking the maximum.

$$\begin{aligned} c(f,\{v\}) &= 0 \\ c(f,A) &= \min_{\emptyset \neq B \subsetneq A} \{f(B) + f(A \setminus B) + c(f,B) + c(f, A \setminus B)\} \end{aligned} \tag{6.1}$$

A decomposition of minimum f-cost can be computed in $O(3^n + \alpha)$ time using dynamic programming, where $O(\alpha)$ is the time needed to compute f. This is done as follows. As a preprocessing step we compute all values of f in $O(\alpha)$ time. Now we solve the recurrence relation in Equation 6.4 bottom up. For each iteration in the recurrence relation the minimum of $|2^X| - 1$ numbers has to be taken. Suppose $|X| = k$. Then this takes of course $O(2^k)$ time for each iteration. In solving the recurrence relation, $|X|$ goes from 1 to n. Since there are $\binom{n}{k}$ subsets of size k, it takes

$$\sum_{k=1}^{n} \binom{n}{k} 2^k = O(3^n)$$

time to compute all $c(f, X)$. Adding both running times yields $O(3^n + \alpha)$.

We will improve on this trivial algorithm by using a concept called subset convolution. The technique that we use is partially inspired by the f-width exact algorithm by Oum [20].

Definition 6.3. Let R be a semi ring. Let V be a set of size n and let $f, g : \mathcal{P}(V) \to R$ be two functions. The subset convolution $f * g$ is defined as

$$(f * g)(X) = \sum_{Y \subseteq X} f(Y) g(X \setminus Y)$$

As one can see, it is easily possible to compute this convolution in $O(3^n)$ semi ring operations by direct evaluation for each subset. Björklund et al.[3] have developed a faster algorithm to compute convolutions for when R is a ring.

Theorem 6.4. *[3, Theorem 1] Let V be a set of size n and let $f, g : \mathcal{P}(V) \to R$ be two functions where R is a ring. Then the subset convolution $f * g$ can be computed in $O(n^2 2^n)$ ring operations.*

From this, they have derived a similar result for the min-sum semi ring, which we will use to establish our new exact algorithm.

Theorem 6.5. *[3, Theorem 3] Let V be a set of size n and let $f, g : \mathcal{P}(V) \to R$ be two functions where R is the min-sum semi ring. Then the subset convolution $f * g$ can be computed in $\tilde{O}(M2^n)$ time, provided that the range of the functions f, g is $\{-M, -M+1, \ldots, M\}$.*

In order to be able to apply the subset convolution technique, we rewrite the recurrence relation to the following one. Note that these are equivalent.

$$\begin{aligned} c(f, \{v\}) &= f(\{v\}) \\ c(f, A) &= f(A) + \min_{\emptyset \neq B \subsetneq A} \{c(f, B) + c(f, A \setminus B)\} \end{aligned} \tag{6.2}$$

With this definition, $f(V)$ can always taken to be 0, since this value will be present in any decomposition while it is not really a cut.

Lemma 6.6. *Given a graph $G = (V, E)$, a cut-function f and an integer K. If $c(f, V) \leq K$ the value $c(f, V)$ can be computed in $\tilde{O}(n2^nK)$ time.*

Proof. To make sure the numbers involved in our computations do not get too big, we work with a bounded versions of f and c. Let $\tilde{f}(X) = \min(f(X), K+1)$ and $\tilde{c}(\tilde{f}, X) = \min(c(\tilde{f}, X), K+1)$. We define the following family of functions. For all $1 \leq i \leq n$ and $A \subseteq V$ such that $|A| \leq i$ define

$$g_i(A) = \begin{cases} \tilde{c}(\tilde{f}, A) & : 1 \leq |A| \leq i \\ K + 1 & : \text{otherwise} \end{cases}$$

We show how to compute g_n. We find that g_{i+1} can be directly derived from $g_i * g_i$, where $*$ is convolution in the min-sum semiring. If $|A| > i+1$ or $|A| = 0$ then $g_{i+1}(A) = K+1$. If $|A| = 1$, $g_{i+1}(A) = \tilde{f}(A)$. Suppose $2 \leq |A| \leq i+1$. Then $(g_i * g_i)(A) = \min_{B \subseteq A} \{g_i(B) + g_i(A \setminus B)\}$. Because we know that $g_i(A) = K+1$ if for the critical B it would hold that $|B| = 0$ or $|B| = i+1$ we can safely rewrite this to

$$\begin{aligned} (g_i * g_i)(A) &= \min_{\emptyset \neq B \subsetneq A} \{g_i(B) + g_i(A \setminus B)\} \\ &= \min_{\emptyset \neq B \subsetneq A} \left\{\tilde{c}(\tilde{f}, B) + \tilde{c}(\tilde{f}, A \setminus B)\right\} \end{aligned}$$

Thus, we can derive g_i by $g_{i+1}(A) = \min((g_i * g_i)(A) + \tilde{f}(A), K+1)$. Furthermore, $g_1(A) = \tilde{f}(A)$. Finally, we note that $\tilde{c}(\tilde{f}, V) = g_n(V)$. So if $c(f, V) \leq K$, then it is given by $\tilde{c}(\tilde{f}, V)$. Conversely, if $\tilde{c}(\tilde{f}, V) \leq K$, then it equals $c(f, V)$. Thus, if $\tilde{c}(\tilde{f}, V) \leq K$ we return it, otherwise report failure.

By Theorem 6.5 we know that computing a convolution in the min-sum semiring can be done in $\tilde{O}(2^n M)$ time, provided that the range of the input functions is $\{-M, -M+1, \ldots, M\}$. Because we work with the bounded functions $\tilde{f}$ and $\tilde{c}$ we see that the values of g_i never exceed $K+1$. Thus every convolution can be computed in $\tilde{O}(2^n K)$ time. Since there are n convolutions carried out in total, the running time $\tilde{O}(n 2^n K)$ follows. $\square$

Theorem 6.7. *Given a graph $G = (V, E)$ and a cut-function f, the f-cost of G can be computed in $\tilde{O}(n2^n k)$, where k is the resulting cost value.*

Proof. Start with $K = 1$ and double its value while the procedure described in Lemma 6.6 reports failure. The resulting running time will be $\sum_{\log K=0}^{\lceil \log k \rceil} \tilde{O}(n2^n K) \subseteq \tilde{O}(n2^n K)$. $\square$

6.2 Investigating cost variants of various width parameters

We apply the result from Theorem 6.7 to the cost variants of the width parameters branch-width [23], carving-width [25], matching-width [18], module-width [9], maximum-matching-width [33], maximum-induced-matching-width [33], rank-width [21] and boolean-width [33]. We give a brief definition of the corresponding cut functions, but for a more thorough treatment we refer to the respective publications. Let $G = (V, E)$ be a graph and $n = |V|, m = |E|$.

The *branch-cost* can be defined as the cost of mid on E, where mid is defined as follows. For $X \subseteq E$ let $\mathrm{mid}(X)$ be the number of vertices meeting an edge in X as well as an edge in $E \setminus X$.

The *carving-cost* can be defined as the cost of η on V, where $\eta(X)$ is the number of edges of G having one end in X and the other end in $V \setminus X$.

The *matching-cost* can be defined as the cost of π on V, where π is defined as follows. Suppose G has at least one perfect matching. For a $X \subseteq V$, let $\pi(X)$ be the maximum $|\delta(X) \cap M|$ over all perfect matchings M, where $\delta(X)$ is the set of all edges having one end in X and the other end in $V \setminus X$.

The *module-cost* can be defined as the cost of ntc on V, where $\mathrm{ntc}(X)$ is the size of the twin class partition of X. A twin class partition of X is a partition of X such that for all $x, y \in X$ we have x and y in the same partition class if and only if $N(x) \cap X = N(y) \cap X$.

The *maximum-matching-cost* can be defined as the cost of mm on V, where $\mathrm{mm}(X)$ is the size of a maximum matching in $G[X, V \setminus X]$.

The *maximum-induced-matching-cost* can be defined as the cost of mim on V, where $\text{mim}(X)$ is the size of a maximum induced matching in $G[X, V \setminus X]$.

The *rank-cost* can be defined as the cost of $2^{\text{cut-rank}}$ on V, where $\text{cut-rank}(X) = \log_2 |\{(\triangle_{y \in Y} N(y)) \cap (V \setminus X) \mid Y \subseteq X\}|$. Since f is required to be integer valued, we consider $f = 2^{\text{cut-rank}}$ instead.

The *boolean-cost* can be defined as the cost of $2^{\text{bool-dim}}$ on V, where $\text{bool-dim}(X) = \log_2 |\left\{(\bigcup_{y \in Y} N(y)) \cap (V \setminus X) \,\middle|\, Y \subseteq X\right\}|$. Since f is required to be integer valued, we consider $f = 2^{\text{bool-dim}}$ instead.

Corollary 6.8. *Given a graph G, the branch-cost, carving-cost, matching-cost, module-cost, maximum-matching-cost, maximum-induced-matching-cost, rank-cost and boolean-cost can be computed in $O^*(2^m)$, $O^*(2^n)$, $O^*(2^n)$, $O^*(2^n)$, $O^*(2^n)$, $O^*(2.784^n)$, $O^*(k2^n)$ and $O^*(k2^n)$ time respectively, where k is the corresponding cost value.*

Proof. For branch-cost it holds that $\alpha \in O(m2^m)$ and since all values are bounded by m, k can never exceed nm, from which the result follows. The precomputations α for carving-cost, matching-cost, module-cost and maximum-matching-cost take all polynomial time per subset, so $O^*(2^n)$ in total. Because all function values are polynomially bounded by n, k can never grow exponential in terms of n and so the result follows. For maximum-induced-matching-cost $\alpha \in O^*(2.784^n)$ [33]. Since all values are bounded by n, k can never exceed n^2, from which the result follows. For rank-cost the precomputations take polynomial time in n per subset, so $\alpha \in O^*(2^n)$. Since for rank-cost the bound on the value k is exponential in n, the result follows. Finally, for boolean-cost $\alpha \in O^*(k2^n)$ [33]. Since for boolean-cost the bound on the value k is exponential in n, the result follows. □

6.3 Applications

Corollary 6.8 should mostly be considered as a theoretical result. Many applications do not linearly depend on their f-cost value, and to make use of the tighter bound that minimum cost decompositions can provide one has to consider adapted functions. However, this adapted function may not be a cut function. In the following we give an example of that. Since we are mostly concerned with boolean decompositions in this thesis, we restrict ourselves to applications for boolean-cost. This should already clearly illustrate how minimum cost decomposition may need an adapted function in order to be tighter than minimum width decompositions and how this may not be a cut function. In that case our f-cost algorithm is useless, since it relies on f being a cut function.

Again consider the problem Maximum Independent Set from Section 3.3 and the corresponding algorithm. Remember that the time-critical part of that algorithm is the merge of two partial solutions. At each cut $(B, A \setminus B)$ in the decomposition, we merge MIS_B and $MIS_{A\setminus B}$ to form MIS_A by making all possible combinations. This takes $O(|\mathcal{UN}(B)| \cdot |\mathcal{UN}(A \setminus B)|)$ time.

In the following we refer to the boolean-cost and the linear boolean-cost of G as boolc(G) and lboolc(G) respectively. Now if we were working on a decomposition with minimized boolean-cost, we wouldn't have a better upperbound guarantee on the size of $|\mathcal{UN}(X)|$ for any X occurring in the decomposition compared to a decomposition with minimized boolean-width. Namely, the upperbounds we have for $|\mathcal{UN}(X)|$ are simply boolc(G) and $2^{\text{boolw}(G)}$ respectively. So since $\text{boolc}(G) \geq 2^{\text{boolw}(G)}$ the notion of boolean-cost doesn't help us here to improve the running time.

Having another look at the Maximum Independent Set algorithm, we see that we actually want to minimize $|\mathcal{UN}(A \setminus B)| \cdot |\mathcal{UN}(B)|$ over all nodes in the decomposition. However, this is not a cut function because this value depends on two subsets instead of one. So the recurrence to solve becomes

$$\begin{aligned} c(f, \{v\}) &= f(\{v\}) \\ c(f, A) &= \min_{\emptyset \neq B \subsetneq A} \{f(B, A \setminus B) + c(f, B) + c(f, A \setminus B)\} \end{aligned} \tag{6.3}$$

with $f(X, Y) = |\mathcal{UN}(X)| \cdot |\mathcal{UN}(Y)|$. How can we solve this fast? In general we can not, for these kind of functions, since 3^n function values are needed, i.e. for each pair $B \subseteq A \subseteq V$ f can have a unique value. But maybe it is possible to exploit the special form $f(A, B) = |\mathcal{UN}(A)| \cdot |\mathcal{UN}(B)|$ in the above recurrence. This remains open for now.

Open question 6.9. *Is it possible to solve Equation 6.4 in time faster than $O(3^n)$ if $f(X, Y) = |\mathcal{UN}(X)| \cdot |\mathcal{UN}(Y)|$?*

The same question can be asked for any other vertex subset problem, like Minimum Dominating Set, for which $f(X, Y) = |\mathcal{UN}(X)| \cdot |\mathcal{UN}(Y)| \cdot |\mathcal{UN}(X \cup Y)|$.

6.3.1 Linear boolean-cost

What if we restrict ourselves to linear boolean decompositions? I.e. does minimizing linear boolean-cost make sense? Yes, it does if our application is Maximum Independent Set. Since we can safely assume our graph is connected (otherwise we can decompose each component independently and glue the resulting decompositions together), we have that $|\mathcal{UN}(\{y\})| = 2$, for any $y \in V$. Thus, we always have either $|\mathcal{UN}(B)| \leq 2$ or $|\mathcal{UN}(A \setminus B)| \leq 2$ for each node in the decomposition, since the decomposition is linear. So the recurrence then becomes

$$\begin{aligned} c(f, \{v\}) &= f(\{v\}) \\ c(f, A) &= \min_{b \in A} \{2 \cdot |\mathcal{UN}(A \setminus \{b\})| + c(f, \{b\}) + c(f, A \setminus \{b\})\} \end{aligned} \tag{6.4}$$

If we embed this equation into the Incremental-UN-Exact algorithm which is described in Algorithm 4, we get a $O^*(2^n \cdot \text{lboolc}(G))$ algorithm. Looking at the Maximum Independent Set algorithm as described in Algorithm 1, we see that its running time actually equals $O(n^2 \, \text{lboolc}(G))$, if we use a linear decomposition with minimal boolean-cost. If we compare this to the linear boolean-width variant, we would have an algorithm of $O(n^2 2^{\text{lboolw}(G)})$ time. Since $\text{lboolc}(G) \leq n \cdot 2^{\text{lboolw}(G)}$, we have indeed tightened the worst case asymptotic running time bound.

Theorem 6.10. *Given a graph G and a minimal linear boolean-cost decomposition, the Maximum Independent Set problem can be solved in $O(n^2 \, \text{lboolc}(G))$ time.*

Unfortunately, this approach does not work for general vertex subset problems.

6.3.2 Conclusion

We have seen a faster than trivial exact algorithm for computing a decomposition of a graph with minimum cost with respect to f, where f ranges over a large class of cut functions. There is room for some further investigation. It would be relevant to investigate for which width parameters and to what extent the cost variants allow to reflect the running time of applications more closely. Moreover, improvements may be made in the polynomial factor of the algorithm, possibly by looking at specific properties of cut functions.

If we restrict ourselves to boolean-cost, we see that minimizing cost rather than width is generally not very useful for solving vertex subset problems. An exception to this is Maximum Independent Set, for which the use of optimal linear boolean-cost decompositions would actually tighten the asymptotic running time. For the other vertex subset problems it would be interesting to investigate possibilities for faster than trivial algorithms minimizing the actual cost of solving a vertex subset problem from a decomposition D, i.e. $\sum_{(A,B)\in D} nec_d(A) \cdot nec_d(B) \cdot nec_d(A \cup B)$, where d depends on the problem being solved. As we have seen, the problem with this is that these values can not be represented by cut functions, as they depend on more than one subset. A topic for further research would be to investigate possibilities to bypass this problem by somehow making use of the fact that these functions have overlapping information in their values.

Regarding heuristics minimizing cost instead of width, we can say that these would probably be the same. Namely, heuristics usually try to minimize the cut value for each individual cut in a greedy manner. So heuristics do generally no distinguish between cost and width.

Chapter 7

Conclusion

Starting with the facts, we have seen that linear boolean decompositions are easier to generate compared to tree shaped boolean decompositions. Not only are the exact algorithms faster, the linear heuristics also seem to surpass their tree shaped counterparts in generation speed. And even though the linear heuristics are restricted in the sense that the solution space is smaller, the applications were most of the cases faster to solve using linear decompositions, since their asymptotic running times are better for linear decompositions.

The other conclusion that we have drawn from the experiments is that the theoretical upperbounds for *nec* in terms of boolean-width is in practice far away from the actual value of *nec*. This encourages to start looking for ways to optimize decompositions using *nec* instead of boolean-width. Since there are several problems to be solved before this can be done, this remains a topic for further research.

For linear boolean-width, we have looked at reduction rules. We found that it is not easy to find and prove reduction rules that are broad enough to be useful. However, we put forward that using reduction rules for general, tree shaped, decompositions in combination with a heuristic for linear decompositions may be a fruitful approach. Namely, although the resulting decompositions will not be linear, they still will exhibit certain linear properties. Additional research is required to verify our intuition.

We have also seen a new non trivial exact algorithm to minimize the so called f-cost over decompositions, for arbitrary cut functions f. Looking at applications in the boolean-width domain, we could conclude that the applicability of this result is minimal for this particular area. An interesting research topic would be to determine for which functions f and for which corresponding applications minimizing f-cost over decompositions would actually tighten the worst case running time bound.

Bibliography

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

[2] 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.

[3] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Fourier meets möbius: Fast subset convolution. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*, STOC '07, pages 67–74, New York, NY, USA, 2007. ACM.

[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] Hans L. Bodlaender and Fedor V. Fomin. Tree decompositions with small cost. *Discrete Appl. Math.*, 145(2):143–154, January 2005.

[8] 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.

[9] Binh-Minh Bui-Xuan, Jan Arne Telle, and Martin Vatshelle. Feedback vertex set on graphs of low cliquewidth. In *Combinatorial Algorithms*, pages 113–124. Springer, 2009.

[10] 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.

[11] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park, editors, *Algorithms and Computation*, volume 6506 of *Lecture Notes in Computer Science*, pages 403–414. Springer Berlin Heidelberg, 2010.

[12] P. Erdös and A. Rényi. On random graphs. *Publicationes Mathematicae 6: 290-297*, 1959.

[13] 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.

[14] 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.

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

[16] 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.

[17] Robert T. Moenck. Practical fast polynomial multiplication. In *Proceedings of the Third ACM Symposium on Symbolic and Algebraic Computation*, SYMSAC '76, pages 136–148, New York, NY, USA, 1976. ACM.

[18] Serguei Norine. *Matching structure and Pfaffian orientations of graphs.* PhD thesis, Citeseer, 2005.

[19] Yoshio Okamoto, Takeaki Uno, and Ryuhei Uehara. Linear-time counting algorithms for independent sets in chordal graphs. In Dieter Kratsch, editor, *Graph-Theoretic Concepts in Computer Science*, volume 3787 of *Lecture Notes in Computer Science*, pages 433–444. Springer Berlin Heidelberg, 2005.

[20] Sang-il Oum. Computing rank-width exactly. *Information Processing Letters*, 109(13):745–748, 2009.

[21] Sang-il Oum and Paul Seymour. Approximating clique-width and branch-width. *Journal of Combinatorial Theory, Series B*, 96(4):514–528, 2006.

[22] 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.

[23] Neil Robertson and Paul D Seymour. Graph minors. x. obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52(2):153–190, 1991.

[24] Neil Robertson and P.D Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49 – 64, 1984.

[25] Paul D. Seymour and Robin Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.

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

[27] Sigve Hortemo SÃęther and Martin Vatshelle. Personal communication.

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

[29] Ch. B. ten Brinke. https://github.com/chiel92/boolean-width. Algorithms for computing boolean-width of graphs and related things.

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

[31] F. J. P. van Houten. Experimental research and algorithmic improvements involving the graph parameter boolean-width. Master's thesis, Utrecht University, The Netherlands, 2015.

[32] 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.

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

[34] Joachim Von Zur Gathen and Jürgen Gerhard. *Modern computer algebra.* Cambridge university press, 2013.

Appendices

Appendix A

Alternative exact algorithm for boolean-width

In this chapter we describe the exact algorithm for computing boolean-width in the same fashion as our new linear boolean-width exact algorithm, as referred to in Chapter 4. The proof of correctness is very similar to that of the linear variant, and for the correctness of the Möbius Transform functions we refer to Björklund et al. [3]. As one can see, the running time does not work out so well as it does for linear boolean-width. Therefore, Vatshelle's approach, as described in Chapter 3 remains the fastest algorithm.

Consider the following function.

$$g_i(X) = \begin{cases} 1 & : 1 \leq |X| \leq i < n \wedge w(f, X) \leq K \wedge f(X) \leq K \\ 1 & : |X| = i = n \wedge w(f, X) \leq K \\ 0 & : \text{otherwise} \end{cases} \tag{A.1}$$

This function will be used and recursively constructed in Algorithm 9.

Remark A.1. Given a graph $G = (V, E)$ of size n and a integer K, the procedure Decide-bool-width (Algorithm 9) decides whether there is a boolean decomposition with width at most K in $O(n^3 2^{n+2K})$.

Algorithm 7 Compute Möbius Transform

```
procedure MOBIUS-TRANSFORM($f, V[, r]$)
    # If $r$ is given, perform a ranked transform
    # Subsets that do not have cardinality $r$ are not counted
    if $r$ then
        for $X \subseteq V$ do
            if $|X| = r$ then
                $\hat{f}_0(X) \leftarrow f(X)$
            else
                $\hat{f}_0(X) \leftarrow 0$
    else
        $\hat{f}_0 \leftarrow f$

    for $i \in V$ do
        for $X \subseteq V$ do
            if $i \in X$ then
                $\hat{f}_i(X) \leftarrow \hat{f}_{i-1}(X) + \hat{f}_{i-1}(X \setminus \{i\})$
            else
                $\hat{f}_i(X) \leftarrow \hat{f}_{i-1}(X)$
    return $\hat{f}_1, \ldots, \hat{f}_n$
```

Algorithm 8 Compute Ranked Möbius Inversion

```
procedure RANKED-MOBIUS-INVERSION($f \circledast g, V[, r]$)
    $(f * g)_0 \leftarrow (f \circledast g)$

    for $i \in V$ do
        for $X \subseteq V$ do
            if $i \in X$ then
                $(f * g)_i(r, X) \leftarrow (f * g)_{i-1}(r, X) - (f * g)_{i-1}(r, X \setminus \{i\})$
            else
                $(f * g)_i(r, X) \leftarrow (f * g)_{i-1}(r, X)$

    for $X \subseteq V$ do
        $(f * g)(X) \leftarrow (f * g)_n(|X|, X)$
    return $(f * g)$
```

Algorithm 9 Decide boolean-width

```
procedure DECIDE-BOOLW($V, K$)
    boolw($X$) $\leftarrow$ *No*, for all $X \subseteq V$
    boolw($\emptyset$) $\leftarrow$ *Yes*
    bool-dim($X$) $\leftarrow$ *No*, for all $X \subseteq V$
    bool-dim($\emptyset$) $\leftarrow$ *Yes*
    $\mathcal{UN}(\emptyset) \leftarrow \emptyset$

    for $i \leftarrow 1 \ldots |V|$ do
        $\hat{g}_i(i, \ldots) \leftarrow$ MOBIUS-TRANSFORM($g_i, V, i$)
        for $X \subseteq V$ do
            $(\hat{g}_i \circledast \hat{g}_i)(i+1, X) \leftarrow \sum_{0 \le j \le i+1} \hat{g}_i(j, X) \hat{g}_i(i+1-j, X)$
        $(g_i * g_i) \leftarrow$ RANKED-MOBIUS-INVERSION($\hat{g}_i \circledast \hat{g}_i, V, i$)

        $\hat{h}_1, \ldots, \hat{h}_n \leftarrow$ MOBIUS-TRANSFORM($g_i * g_i, V$)
        for $X \subseteq V$ do
            if $\hat{h}_n(X) > 0$ then
                $Y \leftarrow$ Binary search in $\hat{h}$
                $\mathcal{UN}(X) \leftarrow$ COMBINE-UN($Y, X \setminus Y$)
                if $|\mathcal{UN}(X)| \le K$ then
                    bool-dim($X$) $\leftarrow$ *Yes*

        for $X \subseteq V$ do
            if $|X| \le i+1 \wedge (g_i * g_i)(X) > 0 \wedge$ bool-dim($X$) then
                $g_{i+1}(X) = 1$
            else
                $g_{i+1}(X) = 0$

    for $X \subseteq V$ do
        if $g_n(X)$ then
            boolw($X$) $\leftarrow$ *Yes*

    return boolw($V$)
```

Algorithm 10 Combined union of neighborhoods

```
1: procedure Combine-UN($A, B$)
2:     $U_A \leftarrow \emptyset$
3:     $U_B \leftarrow \emptyset$
4:     for $S \in \mathcal{UN}(A)$ do
5:         $U_A \leftarrow U_A \cup \{S \setminus B\}$
6:     for $S \in \mathcal{UN}(B)$ do
7:         $U_B \leftarrow U_B \cup \{S \setminus A\}$
8:
9:     $U \leftarrow \emptyset$
10:    for $S_A \in U_A$ do
11:        for $S_B \in U_B$ do
12:            $U \leftarrow U \cup \{S_A \cup S_B\}$
13:    return $U$
```

Appendix B

Unfinished Tales about Multisubset Convolution

In the following we conjecture a new algorithm for computing multisubset convolutions, with a proposed technique which should still be worked out.

Preliminaries

Let N be a set with n elements. Without loss of generality, we can relabel N to be $\{1, \dots, n\}$. Let $m : N \to \mathbb{N}$ be a multiplicity function on N. The pair $M = (N, m)$ defines a multiset. Given a multiset M, we denote the corresponding multiplicity function by m_M and the corresponding set containing the distinct elements in M by N_M. If there is no ambiguity we simply refer to m and N respectively. We generalize the normal set operators to multisets. Let $A = (N_A, m_A), B = (N_B, m_B)$ be two multisets. Then

$$\begin{aligned}
A \uplus B &= (N_A \cup N_B, m_A + m_B) \\
A \cup B &= (N_A \cup N_B, \max(m_A, m_B)) \\
A \cap B &= (N_A \cap N_B, \min(m_A, m_B)) \\
A \setminus B &= (N_A \setminus N_B, \max(0, m_A - m_B)) \\
A \subseteq B &\Leftrightarrow \forall a \in N_A : m_A(a) \leq m_B(a)
\end{aligned}$$

For readability purposes we will sometimes write m_i for $m(i)$. We can also write a subset of $A \subseteq M$ as a tuple of multiplicities $(m_A(1), \dots, m_A(n))$.

Multisets literals are written like $\{a, b^2, c^3\}$, where a, b, c are elements and the upper right number denotes the corresponding multiplicity. If M is a multiset, we denote by $M_{\leq i}$ the subset of M containing only elements that are smaller than i.

Definition B.1. For a multiset $M = (N, m)$, a commutative ring R and two functions $f, g : \mathcal{P}(M) \to R$, the convolution $f * g$ is defined as

$$(f * g)(A) = \sum_{B \subseteq A} f(B) g(A \setminus B)$$

An equivalent notation, using multiplicities as parameters instead of multisubsets, would be as follows.

$$(f * g)(a_1, \ldots, a_n) = \sum_{b_1=0}^{a_1} \cdots \sum_{b_n=0}^{a_n} f(b_1, \ldots, b_n) g(a_1 - b_1, \ldots, a_n - b_n)$$

As one can see, multisubset convolution is equivalent to discrete convolution over multiple variables. Trivially, $f * g$ can be computed by explicit evaluation of $(f * g)(A)$ for all $A \subseteq M$, which would take $O(\sum_{A \subseteq M} |\mathcal{P}(A)|) \subseteq O(|\mathcal{P}(M)|^2)$ time. In the next sections we investigate faster methods.

Discrete Convolution using Kronecker's Trick

In this section, we will see a faster-than-trivial algorithm for computing the multisubset convolution of two functions. In the remainder of this section, let $M = (N, m)$ be a multiset, R a commutative ring and $f, g : \mathcal{P}(M) \to R$ two functions. It is a well known fact that computing a singlevariate discrete convolution $f' * g'$ in R with domain size d, defined as $(f' * g')(k) = \sum_{l=0}^{k} f'(l) g'(k - l)$, is equivalent to multiplying two polynomials in R with maximal degree d. Namely, in the product of $f'(0) + f'(1)x + \cdots + f'(d)x^d$ and $g'(0) + g'(1)x + \cdots + g'(d)x^d$, the coefficients of x^a are exactly $\sum_{b=0}^{a} f(b)g(a-b)$. This means that we can apply any polynomial multiplication algorithm to obtain the singlevariate discrete convolution. However, note that our original multisubset convolution problem is a discrete convolution in multiple variables. To address this, we use a well known technique that is often referred to as Kronecker's Trick. Kronecker's Trick is used to convert a multivariate polynomial multiplication problem into singlevariate one, e.g. as demonstrated by Moenck [17]. The above similarity between multiplying polynomials and computing a convolution suggests that this trick also applies to multivariate convolution problems, which indeed is the case. Since this procedure will be part of the final hybrid algorithm, we demonstrate it here in terms of our multisubset convolution problem.

Define the mapping

$$\varphi : \{0, \ldots, 2m_1\} \times \cdots \times \{0, \ldots, 2m_n\} \to \{0, \ldots, m_n \prod_{i=1}^{n-1} (2m_i + 1)\},$$

$$(a_1, \ldots, a_n) \mapsto a_1 + (2m_1 + 1)a_2 + (2m_1 + 1)(2m_2 + 1)a_3 + \cdots + \prod_{i=1}^{n-1} (2m_i + 1)a_n$$

Lemma B.2. *φ is bijective and has an inverse φ^{-1}.*

Proof. For every element $a \in \text{codomain}(\varphi)$, it is readily seen that a corresponding origin $(a_1, \ldots, a_n) \in \text{domain}(\varphi)$ can be constructed by iteratively dividing a with remainder by $\prod_{j=1}^{i}(2m_j + 1)$, with i running from $n-1$ to 1, where the division results yield a_n to a_1 subsequently. □

The mapping φ should be seen as the converter between the domain of the original multivariate convolution, and the derived singlevariate convolution. Furthermore, define

$$f' : \{0, \ldots, m_n \prod_{i=1}^{n-1} (2m_i + 1)\} \to R, a \mapsto \begin{cases} f(\varphi^{-1}(a)) & \text{if } \varphi^{-1}(a) \in \text{domain}(f) \\ 0 & \text{if otherwise} \end{cases}$$

The function g' is defined similarly.

Lemma B.3. *$f * g = (f' * g') \circ \varphi$*

Proof. Let $(a_1, \ldots, a_n) \in \text{domain}(f)$ arbitrary but fixed and let $a = \varphi(a_1, \ldots, a_n)$. We have to show that $(f' * g')(a) = (f * g)(a_1, \ldots, a_n)$, i.e.

$$\sum_{b=0}^{a} f'(b)g'(a-b) = \sum_{b_1=0}^{a_1} \cdots \sum_{b_n=0}^{a_n} f(b_1, \ldots, b_n)g(a_1 - b_1, \ldots, a_n - b_n)$$

Claim: $f'(b)g'(a-b)$ is counted in $(f' * g')(a)$ iff $f(b_1, \ldots, b_n)g(a_1 - b_1, \ldots, a_n - b_n)$ is counted in $(f * g)(a_1, \ldots, a_n)$. *Proof*: Trivially, if $(b_1, \ldots, b_n) \subseteq (a_1, \ldots, a_n)$ then also $\varphi(b_1, \ldots, b_n) \le \varphi(a_1, \ldots, a_n)$. Conversely, if $b \le a$ it may happen that $(b'_1, \ldots, b'_n) = \varphi^{-1}(b) \not\subseteq \varphi^{-1}(a) = (a'_1, \ldots, a'_n)$. Suppose this is the case. Then it must be that there is some i such that $b'_i > m_i$

or $(a'_i - b'_i) > m_i$. Namely, if not, $b'_i + (a'_i - b'_i) = a_i$, for all i, which implies $\varphi^{-1}(b) \subseteq \varphi^{-1}(a)$. But that means that $\varphi^{-1}(a) \notin \text{domain}(f)$ or $\varphi^{-1}(a-b) \notin \text{domain}(g)$. By definition of f' and g' this implies that $f'(b)$ or $g'(a-b)$ is 0, such that $f'(b)g'(a-b)$ is not counted. □

Lemma B.3 tells us that the multivariate convolution $f*g$ can be computed by evaluating the singlevariate convolution $f' * g'$. This leads to the following result.

Theorem B.4. *Let $d = m_n \prod_{i=1}^{n-1}(2m_i + 1)$. Then $f * g$ can be computed in $O(T(d))$ ring operations, where $T(d)$ is the number of operations needed to multiply two singlevariate polynomials in R of degree at most d.*

Proof. By Lemma B.3, $f * g$ is given by $(f' * g') \circ \varphi$. Note that the size of the domains of $\varphi, \varphi^{-1}, f', g'$ is $d = m_n \prod_{i=1}^{n-1}(2m_i + 1)$, such that computing $\varphi, \varphi^{-1}, f', g'$ takes at most $O(d)$ time. Computing a singlevariate convolution $f' * g'$ in R with domain size d is equivalent to multiplying two polynomials in R with maximal degree d. □

Multiplying two polynomials of degree at most d in R can be done in $O(d \log d \log \log d)$ time using a variant of the well known Schönhage-Strassen algorithm, as described by Von zur Gathen & Gerhard [34, Chapter 8, Theorem 8.23]. This immediately gives the following corollary.

Corollary B.5. *$f * g$ can be computed in $O(d \log d \log \log d)$ ring operations, where $d = m_n \prod_{i=1}^{n-1}(2m_i + 1)$.*

Adapting Fast Subset Convolution to Multisubset Convolution

It appears that one cannot modify the fast subset convolution algorithm of Björklunds et al. to work with multisets, which is what the author tried to do on his first attempt. Due to lack of time, this remains a conjecture, although we would like to give a brief sketch of a strategy that may work.

Conjecture B.6. For two functions $f, g : \mathcal{P}(M) \to R$, the convolution $f * g$ can be computed in $O(|M|^2 \cdot |\mathcal{P}(M)|)$ ring operations.

The proposed solution is as follows.

Consider the *group algebra* of the direct product group

$$G = \mathbb{Z}_{m_1+1} \times \mathbb{Z}_{m_2+1} \times \cdots \times \mathbb{Z}_{m_n+1}.$$

The group structure of G naturally adds/subtracts element multiplicities (as is desirable with multisets), and *ranking* (cf. ranked Möbius transform) enables one to eliminate unwanted wrapping modulo $m_i + 1$ for each i.

To sum up, use the group algebra of G and with fast convolution (that is, the algebra product) executed via the standard sequence: FFT (on G) $\rightarrow$ pointwise product $\rightarrow$ inverse FFT (on G). Similarly to ranked transforms in the Möbius (also called zeta) setting, ranking the transforms by the size of the input multisets enables detection and elimination of wrapping (=the sum of two multiplicities in at least one coordinate i exceeds m_i and hence wraps around modulo $m_i + 1$, which can be detected by observing that the total multiplicity in the result does not match the sum of sizes of the inputs). If this strategy works out, the running time would be roughly the same as what is stated in the conjecture. To execute the FFTs one needs to be careful to have sufficient roots of unity available in the underlying coefficient field. Thus it may be necessary to assume that the functions f, g take values over an appropriate *field* F in place of the ring R, e.g. over the complex numbers will certainly do.

Hybrid Algorithm

In this section we compare the two previously presented algorithms and try to construct a hybrid algorithm which takes the best of both worlds. In the remainder of this section, let $M = (N, m)$ be a multiset, R a commutative ring and $f, g : \mathcal{P}(M) \rightarrow R$ be two functions.

Comparing the two algorithms

In Corollary B.5 we established that the convolution $f * g$ can be computed in $O(d \log d \log \log d)$ time, where $d = m_n \prod_{i=1}^{n-1}(2m_i + 1)$. Suppose k is the maximal multiplicity that occurs in M. Then we can bound d by $k(2k+1)^{n-1} \leq (2k+1)^n$. So the running time is then bounded by

$$O((2k+1)^n n \log k \log(n \log k)) \tag{B.1}$$

In Conjecture B.6 we stated that the convolution $f * g$ can be computed in $O(|M|^2\mathcal{P}(M))$. We can bound $|M|$ by nk and $\mathcal{P}(M)$ by $(k+1)^n$. This results in a running time of

$$O(n^2k^2(k+1)^n) \tag{B.2}$$

Let us compare these running times in different settings. For both k and n being unbounded, we see that the running time in Equation B.2 is asymptotically faster than Equation B.1.

Suppose k is bounded and n is unbounded. Then Equation B.1 reduces to $O((2k+1)^n n \log n)$. Equation B.2 reduces to $O(n^2(k+1)^n)$. As you can see, the latter is asymptotically faster than the former.

Suppose k is unbounded and n is bounded. Then Equation B.1 reduces to $O(k^n \log k \log\log k)$. Equation B.2 reduces to $O(k^{n+2})$. As you can see, now the former is asymptotically faster than the latter.

Thus, neither of the two algorithms outperforms the other in all of the cases. In order to create an algorithm that performs strong in all cases, we combine the two algorithms into an allround hybrid algorithm.

Constructing a hybrid algorithm

In constructing the hybrid algorithm, we use the algorithm from Section B to compute the convolution over a derived ring that has the original convolution operator as multiplication. This latter convolution operator is then evaluated using the conjectured multisubset convolution algorithm as stated in Conjecture B.6. The algorithm is given and proved in Conjecture B.8. To make this possible, we first need the following observation.

Lemma B.7. *Let F be a set of functions $M \to R$ closed under $*$ (multisubset convolution over M) and $+$ (function addition). Then F is a commutative ring under these operations.*

Proof. First we note that F is an abelian group under function addition, since function addition is associative, commutative, has an identity $I^+(X) = 0$ for all $X \subseteq M$, and for every $f \in F$ there exists a $(-f) \in F$.

Second, we prove that F is a commutative monoid under integer convolution $*$.

$$\begin{aligned}
(f*(g*h))(X) &= \sum_{Y\subseteq X} f(Y)\cdot (g*h)(X\setminus Y) \\
&= \sum_{Y\subseteq X} f(Y)\cdot \sum_{Z\subseteq (X\setminus Y)} g(Z)h((X\setminus Y)\setminus Z) \\
&= \sum_{Y\subseteq X}\sum_{Z\subseteq (X\setminus Y)} f(Y)g(Z)h((X\setminus Y)\setminus Z) \qquad (\cdot \text{ distributes over } +) \\
&= \sum_{Y\uplus Z\uplus W = X} f(Y)g(Z)h(W)
\end{aligned}$$

Since f, g and h can be arbitrarily swapped in the last expression due to the associativity and commutativity of normal multiplication, $*$ is associative and commutative. Furthermore, F has a multiplicative identity as well

$$I^*(X) = \begin{cases} 1 & \text{if } X = \emptyset \\ 0 & \text{if otherwise} \end{cases}$$

because

$$(f*I^*)(X) = \sum_{Y\subseteq X} f(Y)I^*(X\setminus Y) = f(X)I^*(\emptyset) = f(X)$$

Finally, we note that $*$ distributes over $+$, since

$$\begin{aligned}
(f*(g+h))(X) &= \sum_{Y\subseteq X} f(Y)\cdot (g+h)(X\setminus Y) \\
&= \sum_{Y\subseteq X} f(Y)g(X\setminus Y) + f(Y)h(X\setminus Y) \qquad (\cdot \text{ distributes over } +) \\
&= \sum_{Y\subseteq X} f(Y)g(X\setminus Y) + \sum_{Y\subseteq X} f(Y)h(X\setminus Y) \\
&= (f*g+f*h)(X)
\end{aligned}$$

asserting that F indeed fulfills the axioms of a commutative ring. □

In the following, we denote partial function application by putting the partial arguments in subscript. For example, if $f:\mathbb{Z}\times\mathbb{Z}\to\mathbb{Z}$ is the function defined by $f(a,b) = a+b$, then $f_1:\mathbb{Z}\to\mathbb{Z}$ is the function defined by $f_1(b) = 1+b$. Because one of the sub algorithms is still a conjecture, the following result is also named conjecture, although the proof is valid and complete provided that the sub algorithms are correct.

Conjecture B.8. Divide M into $M_1 = (N_1, m_1)$ and $M_2 = (N_2, m_2)$. Suppose $n_1 = |N_1|, n_2 = |N_2|$ and k_1 and k_2 are the maximum multiplicities occurring in m_1 and m_2 respectively. Furthermore, suppose that n_1 is bounded by some constant. Then $f*g$ can be computed in $O({k_1}^{n_1}(k_2+1)^{n_2}{n_2}^2{k_2}^2 \log k_1 \log\log k_1)$ ring operations.

Proof. Denote by $*_{M_2}$ the convolution operator which only convolutes over M_2. Let F be a set of functions $M_2 \to R$ closed under $*_{M_2}$ and $+$ (function addition). By Lemma B.7, F is a commutative ring under these operations.

Without loss of generality we can assume that N_1 consists of the first n_1 elements in N. Let $(a_1, \ldots, a_n) = A \subseteq M$, and write $A_1 = A \cap M_1, A_2 = A \cap M_2$. We can split the convolution as follows.

$$\begin{aligned}
(f * g)(a_1, \ldots, a_n) &= \sum_{b_1=0}^{a_1} \cdots \sum_{b_n=0}^{a_n} f(b_1, \ldots, b_n) g(a_1 - b_1, \ldots, a_n - b_n) \\
&= \sum_{b_1=0}^{a_1} \cdots \sum_{b_{n_1}=0}^{a_{n_1}} \sum_{Y_2 \subseteq A_2} f_{b_1,\ldots,b_{n_1}}(Y_2) g_{a_1-b_1,\ldots,a_{n_1}-b_{n_1}}(A_2 \setminus Y_2) \\
&= \sum_{b_1=0}^{a_1} \cdots \sum_{b_{n_1}=0}^{a_{n_1}} (f_{b_1,\ldots,b_{n_1}} *_{M_2} g_{a_1-b_1,\ldots,a_{n_1}-b_{n_1}})(A_2)
\end{aligned}$$

The last expression can be recognized as convolution in the commutative ring F. By Corollary B.5 this can be computed in $O((2k_1+1)^{n_1} n_1 \log k_1 \log(n_1 \log k_1))$ ring operations in F. Since by definition n_1 is bounded, this reduces to $O({k_1}^{n_1} \log k_1 \log\log k_1)$ ring operations in F.

How expensive are ring operations in F in terms of ring operations in R? The multiplication $*_{M_2}$ in F, takes $O({n_2}^2{k_2}^2(k_2+1)^{n_2})$ ring operations in R, by Conjecture B.6. Function addition in F takes at most $O(n_2k_2)$ time, since the domain of functions in F is of size at most n_2k_2. Since $*_{M_2}$ is the most expensive operation, the result follows. □

Paper

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. 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, 7], but in this paper we study the specific case of linear boolean decompositions, which are considered in [1, 10, 12]. 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 $O^*(nec^3)$ due to a dynamic programming algorithm by Bui-Xuan et al. [3], but this can be improved to $O^*(nec^2)$ for linear decompositions. Here, nec is the number of d-neighborhood 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.

Conference title on which this volume is based on.

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 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 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.

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

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

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

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

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 is a function mapping a subset of vertices A to the linear boolean-width of the induced subgraph $G[A,\overline{A}]$.

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

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 19, Appendix A.2)
- 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 20, Appendix A.2)

3.1 Improving the running time

We present a faster and easier way to precompute for all cuts $A \subseteq V$ the value $|\mathcal{UN}(A)|$, 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: procedure INCREMENT-UN($G, X, \mathcal{UN}_X, v$)
2:     $\mathcal{U} \leftarrow \emptyset$
3:     for $S \in \mathcal{UN}_X$ do
4:         $\mathcal{U} \leftarrow \mathcal{U} \cup \{S \setminus \{v\}\}$
5:         $\mathcal{U} \leftarrow \mathcal{U} \cup \{(S \setminus \{v\}) \cup (N(v) \cap (\overline{X} \setminus \{v\}))\}$
6:     return $\mathcal{U}$
```

▶ **Lemma 6** (Appendix A.3). *The procedure Increment-UN is correct and runs in $O(n \cdot |\mathcal{UN}_X|)$ time using $O(n \cdot |\mathcal{UN}_X|)$ space.*

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

```
procedure INCREMENTAL-UN-EXACT($G, K$)
    $T_{\mathcal{UN}}(\emptyset) \leftarrow 0$
    COMPUTE-COUNT-UN($G, K, T_{\mathcal{UN}}, \emptyset, \{\emptyset\}$)

    $P(X) \leftarrow \infty$, for all $X \subseteq V$
    $P(\emptyset) \leftarrow 0$

    for $i \leftarrow 0, \dots, |V| - 1$ do
        for $X \subseteq V$ of size $i$ do
            for $v \in V \setminus X$ do
                $Y \leftarrow X \cup \{v\}$
                if $P(X) \leq K$ then
                    $P(Y) \leftarrow \min(P(Y), \max(T_{\mathcal{UN}}(Y), P(X)))$

    return $\log_2(P(V))$

procedure COMPUTE-COUNT-UN($G, K, T_{\mathcal{UN}}, X, \mathcal{UN}_X$)
    for $v \in V \setminus X$ do
        $Y \leftarrow X \cup \{v\}$
        if $T_{\mathcal{UN}}(Y)$ is not defined then
            $\mathcal{UN}_Y \leftarrow$ INCREMENT-UN($G, X, \mathcal{UN}_X, v$)
            $T_{\mathcal{UN}}(Y) \leftarrow |\mathcal{UN}_Y|$
            if $T_{\mathcal{UN}}(Y) \leq K$ then
                COMPUTE-COUNT-UN($G, K, T_{\mathcal{UN}}, Y, \mathcal{UN}_Y$)
```

▶ **Theorem 7** (Appendix A.4). *Given a graph G, Algorithm 2 can be used to compute* lboolw(G) *in $O(n \cdot 2^{n+\text{lboolw}(G)})$ time using $O(n \cdot 2^n)$ space.*

This new algorithm improves upon the time in Theorem 20 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.

```
procedure GenerateVertexOrdering($G, ScoreFunction, init$)
    $Decomposition \leftarrow (init)$
    $Left \leftarrow \{init\}$
    $Right \leftarrow V \setminus \{init\}$
    while $Right \neq \emptyset$ do
        $Candidates \leftarrow$ set returned by candidate set strategy
        if there exists $v \in Candidates$ belonging to a trivial case then
            $chosen \leftarrow v$
        else
            $chosen \leftarrow \underset{v \in Candidates}{\operatorname{argmin}} (ScoreFunction(G, Left, Right, v))$
        $Decomposition \leftarrow Decomposition \cdot \{chosen\}$
        $Left \leftarrow Left \cup \{chosen\}$
        $Right \leftarrow Right \setminus \{chosen\}$
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 8 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 8** (Appendix A.5). *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.*

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 9, 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 9** (Appendix A.6). *The mapping $\frac{a}{b} \mapsto \frac{a}{a+b}$ is order preserving.*

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.

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

```
procedure Incremental-UN-Heuristic($G, init$)
    $Decomposition \leftarrow (init)$
    $Left, Right \leftarrow \{init\}, V \setminus \{init\}$
    $\mathcal{UN}_{Left} \leftarrow \{\emptyset, N(init) \cap Right\}$
    while $Right \neq \emptyset$ do
        $Candidates \leftarrow$ set returned by candidate set strategy
        if there exists $v \in Candidates$ belonging to a trivial case then
            $chosen \leftarrow v$
            $\mathcal{UN}_{chosen} \leftarrow$ Increment-UN($G, Left, \mathcal{UN}_{Left}, v$)
        else
            for all $v \in Candidates$ do
                $\mathcal{UN}_v \leftarrow$ Increment-UN($G, Left, \mathcal{UN}_{Left}, v$)
                if $chosen$ is undefined or $|\mathcal{UN}_v| < |\mathcal{UN}_{chosen}|$ then
                    $chosen \leftarrow v$
                    $\mathcal{UN}_{chosen} \leftarrow \mathcal{UN}_v$
        $Decomposition \leftarrow Decomposition \cdot chosen$
        $Left \leftarrow Left \cup \{chosen\}$
        $Right \leftarrow Right \setminus \{chosen\}$
        $\mathcal{UN}_{Left} \leftarrow \mathcal{UN}_{chosen}$
    return $Decomposition$
```

▶ **Theorem 10** (Appendix A.7). *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.*

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 11** ((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 12** (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 13** (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 $N^d_A(X)$, is a multiset of vertices from $\overline{A}$, where a vertex $v \in \overline{A}$ occurs $\min(d, |N(v) \cap X|)$ times in $N^d_A(X)$. A d-neighborhood can be represented as a vector of length $|\overline{A}|$ over $\{0, 1, \ldots, d\}$.

▶ **Definition 14** (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 $X \equiv^d_A Y$, 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 $nec(\equiv^d_A)$. The number of equivalence classes $nec_d(T, \delta)$ of a decomposition (T, δ) is defined as $\max(nec(\equiv^d_A), nec(\equiv^d_{\overline{A}}))$ over all cuts $(A, \overline{A})$ of (T, δ).

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

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 15** (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 $ntc(A) \leq \min(n, 2^{\text{bool-dim}(A)})$ [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 $k = \text{bool-dim}(A) = nec(\equiv^1_A)$.

▶ **Lemma 16.** *[3, Lemma 5]* $nec(\equiv^d_A) \leq 2^{d \cdot k^2}$.

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

▶ **Lemma 18** (Appendix A.8)**.** $nec(\equiv^d_A) \leq ntc(A)^{d \cdot k}$.

By Lemma 16 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

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

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 $N^2(Left) \cap Right$, which avoids that we exclude certain optimal solutions, as opposed to Sharmin [12], who restricted this set to $N(Left) \cap Right$. However, this does not affect the results significantly.

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.

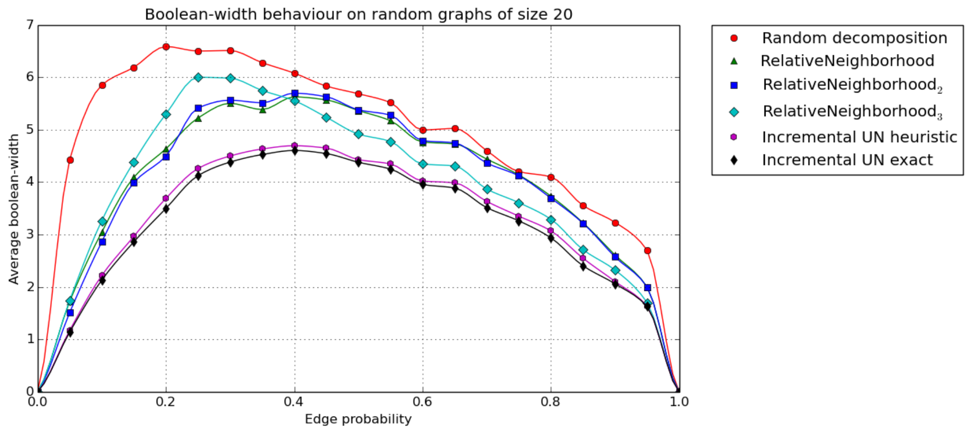


Figure 1 Performance of different heuristics on random generated graphs consisting of 20 vertices, with varying edge probabilities, in terms of linear boolean-width.

6.2 Comparing heuristics on real-world graphs

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.

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

Graph	$\|V\|$	Edge Density	Relative	LeastCut	IUN	2-IUN	n-IUN
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
david	87	0.11	9.38	6.27	6.27	6.27	5.86
celar04-pp	114	0.08	11.67	7.27	7.27	7.27	7.27
1bkb-pp	127	0.18	16.81	9.98	9.98	9.53	9.53
miles1500	128	0.64	8.17	5.58	5.58	5.58	5.29
celar10-pp	133	0.07	10.32	11.95	11.95	7.64	6.91
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

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
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
david	87	0.11	0.02	3.15	0.04	0.06	1.62
celar04-pp	114	0.08	0.04	5.73	0.14	0.23	9.85
1bkb-pp	127	0.18	0.06	198.05	1.14	4.18	107.32
miles1500	128	0.64	0.06	44.57	0.10	0.14	7.05
celar10-pp	133	0.07	0.06	8.93	1.96	4.72	18.43
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

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, note that algorithms parameterized by boolean-width are exponential in the width of a decomposition. The 2-IUN heuristic outperforms IUN in three cases while n-IUN gives a better decomposition in 11 out of 13 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, 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 $ntc = \max\limits_{w \in T} ntc(V_w)$ and $\min ntc = \max\limits_{w \in T} \min(ntc(V_w), ntc(\overline{V_w}))$.

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, because of the $O^*(nec_d(T,\delta)^2)$ space requirement. An interesting observation we can make by looking at the graphs zeroin.i.2 and boblo, is that a lower boolean-width does not imply a lower number of equivalence classes. We even encountered this for decompositions of the same graph: 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 $log_2(nec_2(T',\delta')) = 6.75$.

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)
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
david	5.86	9.37	68.63	22.19	44.61	34	919
celar04-pp	7.27	11.15	105.61	28.53	65.74	-	-
1bkb-pp	9.53	-	181.47	52.30	98.49	-	-
miles1500	5.29	9.30	55.87	34.87	49.69	8	4038
celar10-pp	6.91	10.34	95.41	25.36	59.70	50	10179
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	-	-

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 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., for a cut $(A,\overline{A})$ $nec_d(A) \neq nec_d(\overline{A})$, 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.

A Omitted proofs

A.1 Proof of Theorem 5

▶ **Claim**. 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 $S_i = \{u \,|\, u \in \chi_i : N(u) \cap \overline{\chi_i} \neq \emptyset\}$. For each $u \in S_i$ it holds that $\exists j > i\ \exists w \in X_j$ for which $\{u, w\} \in E$. 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, thus 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

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

◀

A.2 Adaptation of existing exact algorithms

Straighforward dynamic programming leads to the following result.

▶ **Theorem 19.** *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 20 is a direct adaptation to linear boolean-width.

▶ **Theorem 20.** *A linear boolean decomposition of minimum boolean-width for a graph G can be computed in $O(n^3 \cdot 2^{n+\mathrm{lboolw}(G)})$ 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 $O(n^3 \cdot K)$ time.

Now consider relation (1). This can be solved in $O(n \cdot 2^n)$ time by the same reasoning as in Theorem 19. This results in a total running time of $O(n^3 \cdot 2^{n+\mathrm{lboolw}(G)})$ 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. ◀

A.3 Proof of Lemma 6

▶ **Claim.** 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 $\mathcal{U}$, 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 $\mathcal{U}$. Assume for contradiction that S is a valid neighborhood not contained in $\mathcal{U}$. 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 $\mathcal{U}$, 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 $\mathcal{U}$, also contradicting our assumption. It follows that a neighborhood is contained in the set $\mathcal{U}$ if and only if it is a valid neighborhood across the cut $(X \cup \{v\}, \overline{X} \setminus \{v\})$.

The 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 $|\mathcal{UN}_X|$. 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 $\mathcal{U}$ which contains at most $O(|\mathcal{UN}_X|)$ sets of size at most $O(n)$. ◀

A.4 Proof of Theorem 7

▶ **Claim.** Given a graph G, Algorithm 2 can be used to compute $\mathrm{lboolw}(G)$ in $O(n \cdot 2^{n+\mathrm{lboolw}(G)})$ 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 21 this will take $O(\sum_{\log K=1}^{\mathrm{lboolw}(G)} n \cdot 2^{n+\log K}) = O(n \cdot 2^{n+\mathrm{lboolw}(G)+1}) = O(n \cdot 2^{n+\mathrm{lboolw}(G)})$ and take $O(n \cdot 2^n)$ space. ◀

▶ **Lemma 21.** *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 $T_{\mathcal{UN}}$ is always computed when necessary, since there may be X such that $T_{\mathcal{UN}}(X)$ is not computed, while $T_{\mathcal{UN}}(X) \leq K$. Suppose that $X \subseteq V$ of size i occurs in an optimal decomposition and $T_{\mathcal{UN}}(X)$ 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 $T_{\mathcal{UN}}(X_j) \leq K$ 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 $T_{\mathcal{UN}}(X)$ must be computed, contradiction. Thus we conclude that $T_{\mathcal{UN}}$ 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 6.

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, $T_{\mathcal{UN}}(X)$ 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 6), 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 $O(n \cdot 2^{n+K})$.

For the space requirements, we observe that the tables $T_{\mathcal{UN}}$ 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 $O(n \cdot 2^n)$. ◀

A.5 Proof of Lemma 8

▶ **Claim**. Let $X \subseteq \mathit{Left}$. If $\exists v \in \mathit{Right}$ such that $N(v) \cap \mathit{Right} = N(X) \cap \mathit{Right}$, 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 $\mathcal{UN}(\mathit{Left}) = \mathcal{UN}(\mathit{Left} \cup \{v\})$. Thus, for any vertex in $\mathit{Right} \setminus \{v\}$ it will hold that it will interact in the exact same with with $\mathcal{UN}(\mathit{Left})$ as it would with $\mathcal{UN}(\mathit{Left} \cup \{v\})$, resulting in the boolean dimension of the computed ordering being the same. ◀

A.6 Proof of Lemma 9

▶ **Claim**. 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 $\frac{a}{a+b} \leq \frac{c}{c+d}$. ◀

A.7 Proof of Theorem 10

▶ **Claim.** 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 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 $O(n \cdot 2^{|\mathcal{UN}_{Left}|})$ time by Lemma 6. 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 $O(n \cdot 2^k)$. ◀

A.8 Proof of Lemma 18

▶ **Claim.** $nec(\equiv_A^d) \leq ntc(A)^{d \cdot k}$.

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 $mim(A) \leq$ bool-dim(A) [17, Theorem 4.2.10]. From [17, Lemma 5.2.3] we know that $nec(\equiv_A^d) \leq ntc(A)^{d \cdot mim(A)}$, thus $nec(\equiv_A^d) \leq ntc(A)^{d \cdot k}$. ◀

B Figures and Tables

B.1 Figures

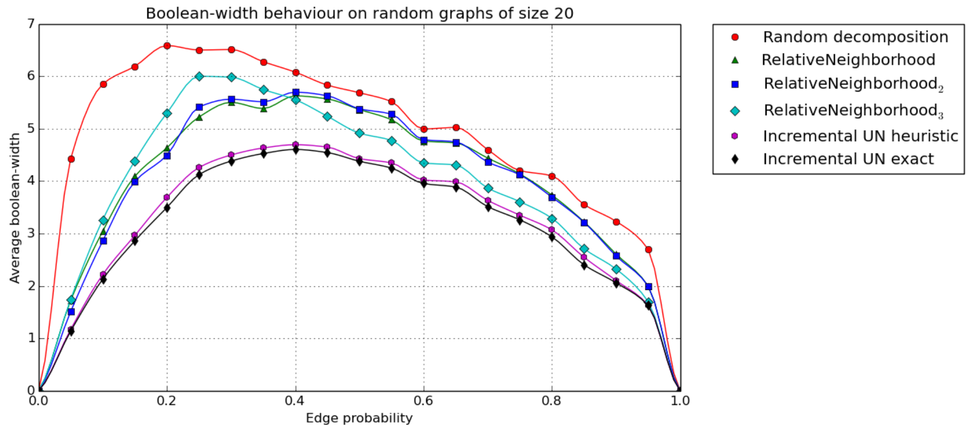


Figure 2 Performance of different heuristics on random generated graphs consisting of 20 vertices, with varying edge probabilities, in terms of linear boolean-width.

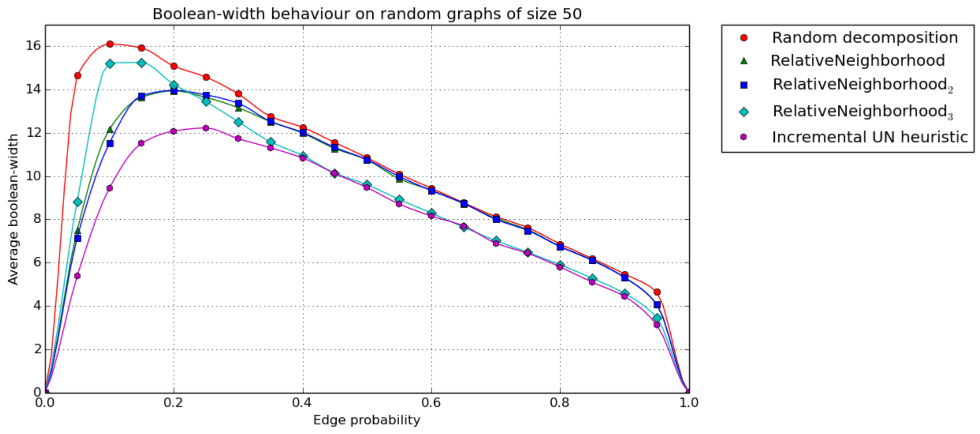


Figure 3 Performance of different heuristics on random generated graphs consisting of 50 vertices, with varying edge probabilities. Because of feasibility limitations, the INCREMENTAL-UN-EXACT algorithm is only used for the in Figure 3. While the optimal values are now unknown, it is clear that INCREMENTAL-UN-HEURISTIC outperforms all other heuristics. Interestingly enough, RELATIVENEIGHBORHOOD$_3$ peers with INCREMENTAL-UN-HEURISTIC as soon as the edge probability exceeds 0.4. Moreover, RELATIVENEIGHBORHOOD and 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 RELATIVENEIGHBORHOOD$_3$ when dense graphs are encountered.

B.2 Tables

Table 4 Linear boolean-width of the decompositions returned by the heuristics described in Section 4, with *Candidates* = *Right*. For 2-IUN we use two start vertices; one is obtained through a single BFS search, while the other is obtained through a double BFS search. The n-IUN heuristic uses all n start vertices, and all other heuristics use start vertices obtained through performing a double BFS.

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

Table 5 Time in seconds of the heuristics used to find the linear boolean decompositions of which the boolean-width is displayed in Table 4.

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

Continued on next page

Table 5 – *Continued from previous page*

Graph	$\lvert V\rvert$	Edge Density	Relative	LeastCut	IUN	2-IUN	n-IUN
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

Table 6 Results of using the algorithm by Bui-Xuan et al. [3] for solving (σ, ρ) problems on graphs, using decompositions obtained using the IUN heuristic using all starting vertices. The columns UB indicate theoretical upperbounds on the number of equivalence classes, with $UB_1 = 2^{d\cdot\text{boolw}^2}$, $UB_2 = (d+1)^{\min ntc}$ and $UB_3 = ntc^{d\cdot\text{boolw}}$, with $ntc = \max_{w\in T} ntc(V_w)$ and $\min ntc = \max_{w\in T} \min(ntc(V_w), ntc(\overline{V_w}))$.

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

Continued on next page

Table 6 – *Continued from previous page*

Graph	boolw	$\log_2(nec)$	$\log_2(UB_1)$	$\log_2(UB_2)$	$\log_2(UB_3)$	MIM	Time (s)
munin4-wpp	7.61	12.13	115.97	19.02	57.98	-	-

Table 7 Width of linear boolean decompositions found with the IUN heuristic using the start vertices returned by performing a double BFS, and with $candidates = N^2(Left) \cap Right$ in order to decrease the computation time. The values of the two others heuristics are taken from [12]. Missing entries are caused by a lack of internal memory which is caused by the $O(n \cdot 2^k)$ space requirement, with k being the linear boolean-width of the computed decomposition. The last column indicates the time of 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

Table 8 Linear boolean-width upperbounds that are obtained through using the IUN heuristic with all starting vertices and $candidates = Right$. The *tw* column gives an upperbound on the treewidth, while the *bw* column gives an upperbound on the boolean-width, which values are taken from [12]. 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 $2^{2lbw} > 2^{3bw}$.

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

Continued on next page

Table 8 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	*tw*	*bw*	*lbw*	*lbw/bw*
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
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

Continued on next page

Table 8 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	*tw*	*bw*	*lbw*	*lbw*/*bw*
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
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

Continued on next page

Table 8 – *Continued from previous page*

Graph	$\lvert V\rvert$	Edge Density	*tw*	*bw*	*lbw*	*lbw/bw*
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
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

Continued on next page

Table 8 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	*tw*	*bw*	*lbw*	*lbw*/*bw*
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
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

Continued on next page

Table 8 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	*tw*	*bw*	*lbw*	*lbw*/*bw*
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
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

Continued on next page

Table 8 – *Continued from previous page*

Graph	$\|V\|$	Edge Density	*tw*	*bw*	*lbw*	*lbw*/*bw*
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
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