Practical Aspects of the Graph Parameter Boolean-width

Sadia Sharmin

Dissertation for the degree of Philosophiae Doctor (PhD)

Department of Informatics
University of Bergen
Norway
June 2014

Acknowledgements

All praises to God almighty, the Lord of the World, who made me capable of going through this PhD journey.

First, I would like to thank everyone in the Algorithms group at the University of Bergen for upholding an excellent learning atmosphere. It has been a great experience to work with such nice and warmhearted people, who are experts in their fields and from whom I have learnt a lot.

I would like to acknowledge the tremendous help and support that I received from my advisors, Fredrik Manne and Jan Arne Telle. My greatest gratitude and appreciation goes to Fredrik Manne who advised me even with every single detail. His ideas, knowledge, and insights were always of great help for my advancement. I also thank him for his patience in reviewing so many inferior drafts, suggesting new ways of thinking, and encouraging me. The thesis would not have existed without him. It has been such an honor, privilege, and enjoyment to learn from and work with him.

Thanks to Martin Vatshelle, my friend and colleague, for helping me not only academically but also with many practical things. His nice and simple way of explaining and his appreciation always encouraged me. My ideas were always clearer after every discussion I had with him. And a special thanks to my office mates Reza and Naim who made my office a cozy place to be.

My sincere thank goes to the administrative staff in the Department of Informatics for easing many of the administrative activities. Thanks to the Norwegian Research Council for their financial support and allowing the freedom of research.

Finally, I am grateful to my parents, parents-in-law, sisters, and all my relatives for their continuous support and encouragement. I love them so much, and I would not have made it this far without them.

Last, but not least, my heartfelt thanks to my husband Aasif Ridwan Khan for the love, support, and endless sacrifices that he made throughout the last couple of years. None of the things I thanked for above would have been possible if he was not there standing by me. He had faith in me even when I didn't have faith in myself. This PhD journey has not been easy, both academically and personally. I believe we both have learned a lot about life during this journey and strengthened our commitment and determination to each other and to live life to the fullest. Special thanks to my son Ayman Arib for his love and patience. He has endured my busy schedules and stress and given me so much inspiration and motivation. Love you all.

Sadia Sharmin

June 2014

Bergen, Norway

I dedicate this thesis to my husband

Aasif Ridwan Khan

Contents

Acknowledgements **iii**

List of Figures **xi**

List of Tables **xiii**

I **xv**

1 Introduction **1**
1.1 Width parameters of graphs . . . 3
1.1.1 Computing width parameters . . . 3
1.1.2 Practical applications of width parameters . . . 4
1.1.3 Boolean-width . . . 4
1.1.4 Comparing boolean-width to other width parameters . . . 5
1.2 The need for good decompositions . . . 6
1.3 Our contribution . . . 6
1.4 Overview of the thesis . . . 7

2 Preliminaries **11**
2.1 Graph and problems on graph . . . 11
2.1.1 Runtime analysis . . . 12
2.2 Decompositions and boolean-width . . . 13
2.3 Properties of boolean-width . . . 15
2.4 Experimental setup . . . 16
2.4.1 Machine configuration . . . 16
2.4.2 Data Set . . . 16

3 A First Attempt at Generating Boolean Decompositions **19**
3.1 The algorithm . . . 20
3.1.1 Greedy initialization . . . 20
3.1.2 Local search . . . 21
3.2 Performance analysis . . . 23
3.2.1 Discussion and implementation details . . . 23
3.3 Experimental results . . . 24
3.3.1 Small grid graphs . . . 26
3.3.2 Rank-width upper bound vs boolean-width upper bound . . . 27
3.4 Conclusion . . . 28

4 Counting Maximal Independent Sets **29**
4.1 Previous algorithms for counting and enumerating MISs 29
4.2 A new algorithm 32
4.3 Experimental results 35
4.3.1 General graphs 35
4.3.2 Bipartite graphs 38
4.3.2.1 Random bipartite graphs 39
4.3.2.2 Real-world graphs 40
4.4 Conclusion 41

5 Speeding up the generation of Boolean Decompositions **43**
5.1 Reduction rules 43
5.1.1 Proof of correctness 44
5.1.2 Tree-width reductions that do not work 45
5.1.3 Implementation 46
5.2 Experimental results 46
5.2.1 Preprocessing 46
5.2.2 Speeding up Algorithm 1 48
5.3 Conclusion 50

6 Generating a Boolean Decomposition from a Tree Decomposition **51**
6.1 Generating a tree decomposition 51
6.2 Computing a boolean decomposition from a tree decomposition 52
6.2.1 Making nodes of the tree disjoint 53
6.2.2 Placing every vertex in a leaf node 54
6.2.3 Making a parent node contain every vertex in its children 55
6.2.4 Making the decomposition binary 55
6.2.5 Time complexity 56
6.3 Experimental results 56
6.4 Conclusion 59

7 Exact and Random Boolean Decompositions **61**
7.1 Exact boolean decomposition 61
7.1.1 Experimental results 62
7.2 Random boolean decompositions 64
7.2.1 Experiments with random graphs 65
7.2.2 Experiments with real world graphs 66
7.3 Conclusion 67

8 Generating Caterpillar Decompositions **69**
8.1 Algorithms for generating caterpillar decompositions 69
8.1.1 Selecting the first vertex 70
8.1.2 Trivial cases 71
8.1.3 Least Uncommon Neighbors 72
8.1.4 Relative Neighborhood 72
8.1.5 Greedy 73
8.1.6 Time complexity and implementation 74
8.2 Experimental results 74

8.3 Other orderings 79
8.4 Conclusion 79

9 Maximum Independent Set using Caterpillar Decompositions 81
9.1 Using a Linear Decomposition to solve the ISP 82
9.1.1 Proof of correctness 84
9.1.2 Time complexity and implementation 84
9.2 Experimental results 85
9.3 Conclusion 88

10 Maximum Independent Set using Branch-and-Bound 91
10.0.1 Background 91
10.1 Our algorithm 92
10.1.1 Preprocessing/reduction rules 92
10.1.2 The recursive algorithm 93
10.1.3 Vertex ordering 97
10.1.4 Experimental results 98
10.1.5 Conclusion 101

11 Conclusion 103
11.1 Open problems 103

Bibliography 105

II Papers 113

Paper I 115

Appendix- Paper I 131

Paper II 139

List of Figures

1.1 A partition of the vertices in G. 4

2.1 A graph G and a possible boolean decomposition of G. 13
2.2 A caterpillar decomposition of G from Figure 2.1. 15

3.1 Ratio (tree-width divided by boolean-width) versus edge density for all the 300 graphs for which heuristically computed upper bounds are known. 24
3.2 Improvement of boolean-width upper bound as the local search progresses over time, for the graph eil51.tsp ($V(G)$=51, $E(G)$=140). 25
3.3 Improvement of boolean-width upper bound as the local search progresses over time, for the graph miles1500 ($V(G)$=128, $E(G)$=5198). 25
3.4 Upper-bound on boolean-width of square grids. 27
3.5 Comparison of experimental rank-width upper bounds with boolean-width upper bounds. 27

4.1 A possible execution of Algorithm 8. 34
4.2 Relative performance of TomitaMis compared to the algorithm by Gaspers et al.. 36
4.3 Relative performance of different CCMis algorithms. 37
4.4 Comparing time for CCMis and UN(A) with respect to average degree for bipartite graphs $BG = (30, 30, E)$. 39
4.5 Comparing time for CCMis and UN(A) with respect to the number of MISs for bipartite graphs $BG = (30, 30, E)$. 40

5.1 Reversing the reduction rules by adding the reduced vertex v. 44
5.2 Simplicial vertices $s_1, s_2, \ldots, s_6$. 45

6.1 A graph G and a corresponding tree decomposition of $tw(G) = 2$. 52
6.2 For $\forall Y \subseteq S$ boolean dimension of $cut(A \cup Y, B \cup (S \setminus Y)) \leq |S|$. 53
6.3 Starting from Figure 6.1 after making the bags disjoint. 54
6.4 Starting from Figure 6.1 after making all bags leaves. 54
6.5 Starting from Figure 6.1 after copying every vertex to its parent. 55
6.6 Starting from Figure 6.1 a binary boolean decomposition tree of G with $boolw(G) = \log(3)$=1.58. 56

7.1 Exact *rw* and *boolw* for a set of small real world graphs 63
7.2 Boolean-width of random decompositions on random graphs with edge probability 0.5. 65
7.3 Boolean-width of random decompositions on random graphs. 65

7.4 Boolean-width of random graph with $n = 20$ vertices using exact and random decompositions. 66

8.1 A graph G. 71
8.2 One stage of the selection process 72

9.1 A graph G 83
9.2 Logarithm (base 2) of the time required for dynamic programming for different orderings 87

10.1 Degree 2 vertex reduction 93

List of Tables

3.1 Results for selected graphs 26

4.1 Description for benchmark real world graphs from TreewidthLIB. 36
4.2 CPU time(sec) for benchmark real world graphs from TreewidthLIB . . . 38
4.3 CPU time(sec) for bipartite graphs generated from the graphs listed in Table 4.1 40

5.1 Effect of reduction rules when applied to graph instances from probabilistic networks 47
5.2 Effect of reduction rules when applied to graph instances generated from other sources than probabilistic networks 48
5.3 Comparison of greedy initialization for original and preprocessed graphs . 49
5.4 Comparison of the time required for greedy initialization in Algorithm 1 using UN(A) and CCMis 49
5.5 Greedy initialization using CCMIS on large graphs 49

6.1 Comparison of boolean-width upper bound obtained from Algorithm 9 and tree-width upper bound listed in TreewidthLIB 57
6.2 Comparison of boolean-width upper bounds obtained from Algorithm 1 and Algorithm 9 58

7.1 Exact *rw*, *boolw*, and *tw* for a set of small real world graphs 62
7.2 Exact *cw*, *rw*, and *boolw* for a set of named graphs 64
7.3 Comparison of boolean-width upper bounds obtained from random decompositions, Algorithm 1, and Algorithm 9 67

8.1 Running time of the heuristics on small graphs 75
8.2 Linear boolean-width upper bounds given by the heuristics on small graphs 76
8.3 Running time of the heuristics on large graphs 77
8.4 Linear boolean-width upper bounds given by the heuristics on large graphs 77
8.5 Comparing linear boolean-width upper bounds of graphs with exact clique-width 78

9.1 Comparing running times for Algorithm 14 using different orderings . . . 86
9.2 Comparing running times for Algorithm 14 using different orderings . . . 86
9.3 Comparing running times for Algorithm 14 using different orderings for the 2nd DIMACS graphs 88

10.1 Comparing Algorithm 15 with the algorithm of McCreesh et al. 99

10.2 Running times for Algorithm 15 on a set of graphs from the 2nd DIMACS challenge 100
10.3 Comparing running times of Algorithm 15 and Algorithm 14 101

Part I

Chapter 1

Introduction

Many natural computational problems are NP-complete or NP-hard. As far as we know these problems are computationally intractable, meaning that they lack polynomial time algorithms. Still these tasks can be solved given sufficient time, but in practice this might take too long to be useful.

Discovering that a problem is NP-complete or NP-hard provides a compelling reason to stop searching for an efficient algorithm for it. But as such problems still need to be solved quickly a number of methods have been developed to deal with these. This includes various techniques such as heuristics that trade optimality, completeness, accuracy, or precision for speed. Simulated annealing, tabu search, and ant colony optimization are other well known metaheuristics to search for the global optimum of a given function in a large search space. Approximation algorithms provide solutions with guaranteed error bounds in polynomial time. This approach is increasingly being used for large real world problems, but not all approximation algorithms are suitable for practical applications.

While heuristics and approximation algorithms can in many cases give solutions of sufficient quality, there are settings where an optimal solution is required. For these settings techniques such as branch-and-bound search algorithms can be used to go through the solution space looking for the best solution. Other methods include Integer linear programs and corresponding solvers, such as CPLEX.

Dynamic programming is yet another method for solving complex problems by breaking them down into simpler subproblems, that is applicable to problems exhibiting the properties of overlapping subproblems and optimal substructure. Common to these approaches is that they might use an exponential amount of time and thus be impractical in many settings.

If this is the case it might still be possible to simplify or restrict the problem. Although a problem can be hard in general, there are situations in which it is possible to quantify when an instance may be easier than the worst case. Many of the problem instances that occur naturally have a hidden structure and taking advantage of this can make our task easier. Parameterized Complexity is one such approach that extracts and exploits the power of this hidden structure of the input instances to solve hard problems. In parameterized algorithms every problem instance comes with a relevant secondary measurement k, called a *parameter*. The parameter indicates how difficult the instance

is. Typically the larger the parameter value the harder it is to solve the problem and for small values of the parameter, the problems can be solved efficiently. This eventually brings about the notion of fixed parameter tractability (FPT). A problem is fixed parameter tractable (FPT) if an instance of size n with parameter k can be solved in $f(k)n^{O(1)}$ time where f is a computable function independent of n. Thus for small k it is feasible to solve these problems even on large input instances.

Many important computationally hard problems can be modeled as graph problems. Parameterized algorithms for solving hard problems on graphs can be easier when the parameters are small, i.e. when the input graph is structurally "simple". Conceivably trees are one of the simplest type of graphs, i.e. undirected connected graphs without cycles. Trees are structurally easy to understand and many NP-hard problems can be solved efficiently on these. The qualitative reasoning for this is: If we consider a subtree T of the input, the solution to the problem restricted to T only interacts with the rest of the graph through a single vertex v. Therefore, by considering the different ways in which v might affect the solution for T, we can essentially decouple solving the problem in T from the solution to the problem in the rest of the tree.

Tree-width ($tw(G)$) is a measure of how tree-like a graph is. It has been inspirational for the introduction of many concepts and parameters measuring similarity of structures to trees or how a structure is distinguished from a tree. Examples include path-width [1], branch-width [2, 3], clique-width [4], and rank-width [5].

These parameters can be used to analyze the structure of input objects in algorithmic problems, and aids in designing efficient algorithms to solve the regarded problem. The runtime of such algorithms mainly depend on the size of the studied width parameter. For solving problems many of these parameterized algorithms first finds a decomposition tree of the given graph of small width. This decomposition is then used in a dynamic programming algorithm to solve the original problem. There exists a large number of both algorithmic and structural results for existing width parameters on graphs, see [6] for an overview.

Computing a decomposition that exposes the minimum value of any of these width parameters is an NP-hard problem in itself. Thus any practical use of these methods must rely on heuristics or approximation algorithms that first computes decompositions of sufficiently small width that the ensuing dynamic programming algorithm can run efficiently. It follows that from a practical point of view there are two main issues that needs to be addressed:

- How does one efficiently compute a decomposition with small width?
- How does one implement the ensuing dynamic programming algorithm so that the combined running time compares favorably to other approaches?

In this thesis we have studied these issues for one such width parameter, namely boolean-width. This is a recently introduced width parameter [7] and relates to the maximum number of distinct neighborhoods across cuts of a recursive decomposition of the input graph. In theory, boolean-width should for many problems be competitive compared to other width parameters. But so far there has not been any studies that document its practical use. We present in this thesis a first study that does this. This includes the development, implementation, and testing of several new algorithms for computing

boolean decompositions. We have also tested the output of these in a dynamic programming algorithm for solving the Maximum Independent Set problem (ISP). As a byproduct of this work we have developed a new efficient algorithm for counting the number of Maximal Independent Sets (MISs) in a graph and a new efficient branch and bound algorithm for the IS problem. In the following parts of this chapter we give a more detailed introduction to width parameters in general and their practical use. We also motivate the use of boolean-width and give a summary of the main contributions of this thesis.

1.1 Width parameters of graphs

Tree-width is the most well studied graph width parameter. Given a decomposition of tree-width k, there are case-specific algorithms solving many NP-hard problems [8]. Similar results hold for branch-width $bw(G)$, since $bw(G) \leq tw(G) + 1 \leq 1.5bw(G)$ [2]. A drawback of tree-width and branch-width arises with dense graphs, where their value can be high. The complete graph K_n has tree-width $n-1$ and $2^{tw(K_n)}$ is thus exponential in n. The introduction of clique-width $cw(G)$ was a large improvement for dealing with dense graphs as it can be low even for these [4]. Moreover, given a decomposition of clique-width k, many NP-hard problems can still be solved reasonably fast. The parameter rank-width $rw(G)$ introduced in [5], is potentially much smaller than clique-width, tree-width, and branch-width: for any graph G it is known that $\log cw(G) \leq rw(G) \leq cw(G)$, $rw(G) \leq tw(G) + 1$, and $rw(G) \leq bw(G)$ [9]. Still, for these parameters to have impact on solving real problems, it is required that one implements and tests the associated algorithms.

1.1.1 Computing width parameters

For tree-width there is an $O(n2^{O(k^3)})$ algorithm for finding a decomposition of tree-width k, if it exists [10]. This algorithm is not practical [11], but much work has been done on finding decompositions of low tree-width in practical settings, see the overviews [12, 13]. The web site TreewidthLIB [14] has been established to provide a benchmark and to join the efforts of people working in experimental settings to solve graph problems using tree-width and branch-width [15, 16]. This includes problems from computational biology [17–19], constraint satisfaction [20, 21], and probabilistic networks [22]. Hicks [23] studied the practical computation of branch-width for specific classes of graphs.

There is a recent study on computing rank-width [24] of real-world graphs. This was inspired by an algorithm for computing boolean-width [25]. The authors compared rank-width on graphs of practical relevance to established bounds of boolean-width. Although the rank-width upper bounds of most graphs is lower than the known upper bounds for tree-width, it turns out that the boolean-width upper bounds from [25] are significantly lower than the rank-width upper bounds in almost all cases.

In [26] a new method was presented for computing the clique-width of graphs. This is based on an encoding to propositional satisfiability (SAT) and then evaluating this using a SAT solver. Results were given for a set of random graphs and named graphs, but it should be noted that this approach only works for a limited number of vertices.

1.1.2 Practical applications of width parameters

Though the concept of width parameters has mainly been considered from a theoretical point of view, in recent years a number of computational studies has shown that results from this field can also be applied successfully in practical settings. Initial attempts were taken by Cook and Seymour [27], who used branch decompositions to obtain close-to-optimal solutions for the Traveling Salesman problem. Path decompositions were used by Verweij [28] to solve lifting problems of cycle inequalities for the independent set problem. Tree decompositions were used to obtain lower bounds and optimal solutions for a special type of frequency assignment problems [29, 30]. Moreover, tree decompositions have also been used to solve problems in the area of expert systems. There exists an efficient algorithm for the inference calculation in probabilistic or Bayesian networks, which builds upon a tree decomposition of the moralized graph of the networks [22]. All these applications show that dynamic programming algorithms based on a path-, tree-, or branch decomposition can be an alternative to other existing techniques for solving hard combinatorial problems on graphs.

1.1.3 Boolean-width

In the following we describe and motivate the graph parameter boolean-width, that was recently introduced by Bui-Xuan, Telle, and Vatshelle [7]. Consider the problem of computing a Maximum Independent Set in a graph G. If we divide the vertices into two sets A and $\overline{A}$ then one way to do this is to compute every possible independent sets of A and $\overline{A}$ separately. Then the optimal solution can be found by taking the two solutions $S_1 \subseteq A$ and $S_2 \subseteq \overline{A}$ such that $|S_1| + |S_2|$ is maximized and there is no edge between a vertex in S_1 and one in S_2. We can speed up this computation and reduce

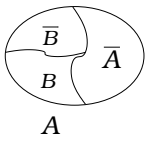

FIGURE 1.1: A partition of the vertices in G.

the storage requirements by noting that if solutions $S \subset A$ and $T \subset A$ both have the same neighborhood in $\overline{A}$, then it is sufficient to only store the larger of these and use this when comparing with the solutions of $\overline{A}$. Thus we can group all the independent sets in A with the same neighborhood in $\overline{A}$ into one equivalence class and only store the best solution for each class. The same applies to the solutions in $\overline{A}$. With some care this idea can be used to design a recursive algorithm for the ISP. Partition A into two parts B and $\overline{B}$ as shown in Figure 1.1 and assume that we have (recursively) computed the largest IS in B for every possible neighborhood in $V - B$ and similarly for $\overline{B}$. Then we can combine these solutions to obtain the best solution for every possible neighborhood in $\overline{A}$ of an independent set in A. To do this we note that for a particular neighborhood in $\overline{A}$ the corresponding independent set of A must consist of an independent set $T_1 \subseteq B$ and one $T_2 \subseteq \overline{B}$ such that there is no edge between any vertex in T_1 and T_2. Thus we iterate over every stored IS $T_1 \subseteq B$ and $T_2 \subseteq \overline{B}$, and if there is no edge between them we test if $T_1 \cup T_2$ gives a new better solution for their combined neighborhood in $\overline{A}$.

For this idea to be efficient it is clear that A should be chosen in such a way that the number of distinct neighborhoods in $\overline{A}$ of vertices in A is small. Since we have to perform the same type of computation on every recursive cut we would like these to be such that the overall work is minimized. But as the number of cuts is linear in the size of $V(G)$ we get a good estimate of the optimal decomposition by requiring that the maximum number of neighborhoods of any cut should be as small as possible. The logarithm of this number is known as the boolean dimension of the cut and the maximum of these over all cuts in a decomposition tree where this maximum is minimized is known as the boolean-width of G.

In this way, boolean-width is defined by a decomposition tree that minimizes the number of different unions of neighborhoods across the resulting cuts in the decomposition tree of the graph. It has been used to solve problems where vertex sets having the same neighborhoods across the cuts can be treated as equivalent. This includes problems related to Independent Set, Dominating Set, Perfect Code, H-Homomorphism, H-Covering, H-Role Assignment etc. [31]. Similarly to tree-width, dynamic programming algorithms to solve these problems using boolean-width employ a table at each node of the decomposition tree, to store solutions to partial problems. Algorithms using boolean-width have been shown to have better runtime compared to dynamic programming algorithms parameterized by other width parameters, due to a lower parameter value for many graphs [7].

1.1.4 Comparing boolean-width to other width parameters

There are several ways to compare different width parameters. This includes looking at the values of the parameters on various graph classes. One can also consider the runtime of algorithms for finding the corresponding optimal decomposition and the classes of problems that can be solved by dynamic programming using a particular decomposition along with the runtime of these parameterized algorithms.

The value of boolean-width, $boolw(G)$, is smaller than clique-width and potentially much smaller than rank-width: we have that $\log cw(G) \leq boolw(G) \leq cw(G)$ and $\log rw(G) \leq boolw(G) \leq 0.25rw^2(G) + O(rw(G))$, with both lower bounds tight to a multiplicative factor [7]. In [7] it is also shown that well-known classes of graphs, like random graphs and interval graphs, have clique-width and rank-width exponential in their boolean-width. It is known that boolean-width is never higher than tree-width or clique-width and can be as low as logarithmic in clique-width. For example, any interval graph or permutation graph has boolean-width $O(\log n)$ [32], while there exist such graphs of clique-width $\Omega(\sqrt{n})$ and tree-width $\Omega(n)$. Also, a random graph with constant edge probability will almost surely have boolean-width $\Theta(\log^2 n)$ [31] but linear clique-width and tree-width.

In contrast to tree-width, the dynamic programming for boolean-width involves a non-negligible preprocessing phase computing indices of the tables, the so-called "representatives". Regardless, the total runtime is in many cases close to that for tree-width, e.g. given a decomposition of tree-width k Maximum Weight Independent Set is solved in time $O(n2^k)$ and Minimum Weight Dominating Set in time $O(n3^k k^2)$ [8]. For a decomposition of boolean-width k Maximum Weight Independent Set is solved in time $O(n^2 k 2^{2k})$ and Minimum Weight Dominating Set in time $O(n^2 + nk2^{3k})$ [7]. These boolean-width-based algorithms are straightforward and described in [7]. Comparing

dynamic programming algorithms based on tree-width versus boolean-width to solve Maximum Independent Set, we see that for Maximum Independent Set the exponential factor in the runtime is $2^{tw(G)}$ versus $2^{2boolw(G)}$. Thus given decompositions of tree-width k and boolean-width k', the boolean-width algorithm becomes preferable when $k > 2k'$. For Minimum Dominating Set the exponential factor in the runtime is 3^{tw} versus 2^{3boolw} and the cutoff is a bit lower, i.e. when $k \geq 1.9k'$.

1.2 The need for good decompositions

A large number of the FPT algorithms uses dynamic programming along a decomposition tree [8, 33]. The efficiency of solving the problems based on its decomposition trees depends greatly on the width of the decompositions. Whereas the dynamic programming phase is application dependent, the calculation of a tree decomposition of small width for a graph can be done independently of the application. The same decomposition can then be used to run many dynamic programming algorithms based on the current width parameter. Thus if we spend more time on the generation of a good decomposition this can improve the running time of any ensuing FPT algorithm.

There are three main measures by which we can compare the quality of the generated decomposition trees. The first is the value of the computed width parameter. Next, the time needed to compute this value is also of importance. Finally, we need to consider the time spent to solve a given problem using dynamic programming along the decomposition tree. These issues will be discussed further when we compare different heuristics in chapters 7 and 8.

1.3 Our contribution

Boolean width is a comparatively newly introduced graph parameter. Previous work on boolean-width has mainly been theoretically oriented, focusing on analytical studies and on developing, but not testing algorithms. In this thesis we investigate and develop techniques to make boolean-width usable in some practical situations. We devise and test several alternative heuristics for generating different decompositions. Computational upper bounds from each of these heuristics are compared against each other as well as with bounds available for other width parameters. Our first heuristic is based on a greedy initialization approach, followed by refinement of the initial decomposition via local search. In this heuristic the boolean dimension computation is done by listing the neighborhoods across a bipartite graph. This is the first work of this kind and shows promising results when comparing boolean-width against other known width values on a number of test graphs.

Computing the boolean dimension is the most time consuming part of evaluating a boolean decomposition. Following our initial work it was shown that one can also compute the boolean dimension of a cut in a decomposition tree by counting the number of maximal independent sets across the same cut. To exploit this we developed a new algorithm for counting the number of maximal independent sets in a general graph and compare this with other existing algorithms. We then test our algorithm on bipartite graphs and compare with the strategy of listing neighborhoods to evaluate cuts in

boolean decomposition trees. Though both approaches are exponential, experimental results show that counting maximal independent sets is favorable in a practical setting and extends the range of graphs for which the boolean dimension can be computed.

In order to increase the number of application areas it is important to be able to compute the boolean decomposition and corresponding boolean-width of a given graph reasonably fast. Preprocessing to reduce the size of the considered problem is almost always a very useful technique while solving large NP-hard problems. We present simple preprocessing rules for boolean-width and show their effectiveness through experiments.

Next we consider how boolean-width of a decomposition tree generated from a tree decomposition compares from a practical point of view. It is known that a boolean decomposition of a graph can always be generated from a tree decomposition of the same graph [31]. We have implemented this method and compared the obtained boolean-width, with the tree-width on a set of benchmark graphs for which tree-width upper bounds are already known. This study shows that boolean-width is in a number of cases substantially smaller than tree-width. From theory all that is known is that boolean-width is never higher than tree-width. We have also computed the optimal boolean-width for a number of small graphs and compared with the corresponding optimal rank-width from [34] and optimal clique-width from [26]. This again compares favorably, with the boolean-width being the smallest in most cases.

We then consider how boolean-width can be used to solve the Maximum Independent Set problem. We do this in two stages. First, we design and test heuristics for generating caterpillar decompositions corresponding to linear orderings of the vertices in a graph. These are compared based on their runtime and width of the generated decomposition. Next we test these decompositions using a dynamic programming algorithm for computing a maximum independent set in a graph. To compare how good these results are we have also designed and implemented a branch-and-bound algorithm for the ISP. This algorithm is based on several of the same ideas as we used for counting maximal independent sets. This comparison shows that there are instances where the dynamic programming algorithm is the fastest.

To conclude, the results from this thesis is a first study to document the practical use of boolean-width.

1.4 Overview of the thesis

In this section we outline the structure of this thesis, which consists of two parts. The first part is composed of 11 chapters. The second part is composed of two published papers that form the core of chapters 3 and 4 from the first part:

Paper I: Eivind Magnus Hvidevold, Sadia Sharmin, Jan Arne Telle, and Martin Vatshelle, Finding Good Decompositions for Dynamic Programming on Dense Graphs, *Proceedings of the 6th International Symposium on Parameterized and Exact Computation, IPEC'11*, LNCS 7112, pages 219-231, 2011.

Paper II: Fredrik Manne and Sadia Sharmin, Efficient Counting of Maximal Independent Sets in Sparse Graphs, *Proceedings of the 12th Symposium on Experimental*

Algorithms, SEA'13, LNCS 7933, pages 103-114, 2013.

Let us give an outline of the 11 chapters of the first part.

- **Chapter 1. Introduction** is the current chapter which covers the motivation for the work carried out in this thesis.
- **Chapter 2. Preliminaries** presents necessary definitions, key examples, common notation, and some known results that are useful for the following chapters.
- **Chapter 3. A First Attempt at Generating Boolean Decompositions** describes the first heuristic algorithm for finding relatively low-width boolean decompositions for real-world graphs. Using greedy initialization and local search we generate various decomposition trees of the graph. These are compared by evaluating the boolean dimension of the cuts induced by edges of the tree, done by listing all unions of neighborhoods of the cut.
- **Chapter 4. Counting Maximal Independent Sets** attacks a bottleneck in the heuristic algorithm from the previous chapter, giving a new and fast algorithm for exactly counting the number of maximal independent sets in a bipartite graph, which corresponds to the number of unions of neighborhoods of the related cut. The new counting algorithm is compared with existing algorithms.
- **Chapter 5. Speeding up the generation of Boolean Decompositions** starts by discussing preprocessing rules for the computation of boolean-width and shows results of three simple reductions on real-world graphs. The insights of Chapters 4 and 5 are then used to improve on the results of Chapter 3.
- **Chapter 6. Generating a Boolean Decomposition from a Tree Decomposition** presents a completely different heuristic algorithm for finding relatively low-width boolean decompositions for real-world graphs. Firstly, using a minimum degree heuristic a tree decomposition of the graph is computed, with relatively low tree-width. Secondly, an algorithm is given that transforms this into a boolean decomposition of no larger width.
- **Chapter 7. Exact and Random Boolean Decompositions** starts by showing the results of brute-force computation of exact boolean-width on some small well-known graphs, comparing their value to that of other width parameters, like tree-width, clique-width, and rank-width. Experiments with randomly generated boolean decompositions are also given.
- **Chapter 8. Generating Caterpillar Decompositions** gives heuristics for finding boolean decompositions with a linear structure rather than a tree structure. Several approaches are compared experimentally.
- **Chapter 9. Maximum Independent Set using Caterpillar Decompositions** gives an exact algorithm computing the Maximum Independent Set in a graph, based on dynamic programming along a caterpillar decomposition as described in the previous chapter. Related experimental results and drawbacks of this approach are discussed.

- **Chapter 10. Maximum Independent Set using Branch-and-Bound** gives a different exact algorithm computing the Maximum Independent Set in a graph, and compares its performance with that of the previous chapter.
- **Chapter 11. Conclusion** gives a summary and points to possible directions for future work.

Chapter 2

Preliminaries

We describe a general framework for decomposing graphs and use this to define unions of neighborhoods (boolean dimension) and boolean-width. To build this framework Chapter 2 begins with some basic graph theoretic terminology. It continues with some fundamental tools from preliminary work regrading boolean-width, with key examples to facilitate understanding. We also list some known facts about boolean-width that we will make use of in the subsequent chapters. Finally, we describe the experimental setup used in throughout this thesis.

2.1 Graph and problems on graph

We start by giving basic graph theoretic definitions. Standard textbooks relevant to graph theory can also be referenced, such as the introductory book by R. J. Wilson [35] or for an in-depth perception the more advanced book by R. Diestel [36].

A *graph* $G = (V(G), E(G))$ consists of the set of vertices $V(G)$ and edges $E(G)$. Each edge consists of an unordered pair of vertices. Vertices u and v are *adjacent* if the edge $\{u, v\} \in E(G)$. Further the vertices u and v are *incident* to the edge $\{u, v\}$, and are referred to as the *endpoints* of the edge $\{u, v\}$. For $v \in V(G)$, we define $E_G(v)$ to be the set of edges incident on v. If $\{v, v\} \notin E(G)$ for all $v \in V(G)$, then G is *loopless*. Two or more edges that are incident to the same two vertices are called *multiple edges*. A graph without loops and multiple edges is *simple*. Unless specifically stated otherwise the graphs considered in this thesis are simple and undirected. The graph obtained by removing a vertex subset $X \subseteq V(G)$ and all its incident edges from G is denoted by $G \setminus X$. The *open neighborhood* or just *neighborhood* of a vertex v in the graph G is the set $N_G(v) = \{u : \{u, v\} \in E(G)\}$ and the *closed neighborhood* of a vertex v is the set $N_G[v] = N_G(v) \cup \{v\}$. The neighborhood of vertex set $S \subseteq V(G)$ is $N_G(S) = (\cup_{v \in S} N_G(v)) \setminus S$. The *degree* of a vertex v, $deg(v)$ is $|N_G(v)|$. When the graph is clear from the context we will omit the subscript. A vertex $v \in V(G)$ of degree 0 is called an *isolated vertex* or *islet*. A *pendant vertex* is a vertex $v \in V(G)$ of degree 1. Two vertices $u, v \in V(G)$ are *twins* if $N_G(u) \setminus v = N_G(v) \setminus u$. If $N_G(u) = N_G(v)$ then u and v are *false twins* and if $N_G[u] = N_G[v]$ they are *true twins*. For a vertex set $A \subseteq V(G)$, the *complement* of A is denoted by $\overline{A} = V(G) \setminus A$. We use the convention that sets are denoted by capital letters. A set is given as an unordered list $\{\cdot\cdot\}$. Sequences are given as ordered lists inside parentheses $(\cdots)$.

A vertex or edge set with a specific property is *minimal (maximal)* if no proper subset (superset) of the set has the property. A graph H is a *subgraph* of a graph G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$ denoted by $H \subseteq G$. For $S \subseteq V(G)$ the subgraph of G induced by S is denoted by $G[S]$ and is the maximal subgraph $H \subseteq G$ having $V(H) = S$.

A *walk* in a graph G is a sequence $v_1, ..., v_k$ of vertices such that $\{v_i, v_{i+1}\} \in E(G)$. A walk that never contains the same vertex twice is called a *path*. A walk where the first and last vertex is the same but other vertices are unique is called a *cycle*. The length of a walk is $k-1$. A graph $G = (V(G), E(G))$ is *bipartite* if the vertices can be divided into two disjoint sets X and Y such that every edge connects a vertex in X to one in Y. We refer to a bipartite graph with the notation $BG(X, Y, E)$. For $A \subseteq V(G)$, a *cut* in a graph G is a bipartition of $V(G)$ in A and its vertex complement $\overline{A}$ and defined as $cut(A, \overline{A}) = cut(A, V(G) \setminus A)$. A $cut(A, \overline{A})$ defines a bipartite graph $BG(A, \overline{A}, E)$ where the edges $E(BG)$ only contains edges in $E(G)$ connecting a vertex from A to a vertex in $\overline{A}$. These are the edges across the cut. A graph G is *connected* if for every pair of vertices $u, v \in V(G)$ there exists a path from u to v. A graph that is not connected is disconnected. A set $S \subseteq V(G)$ is a *separator* if removing S and all edges incident on S from a connected graph G makes G disconnected. A separator of size 1 is called a *cut vertex*.

A *tree* is a connected graph with no cycles. In a tree T, to avoid confusion with a graph, we call the elements in $V(T)$ nodes. A node with degree at most 1 is called a *leaf* and a node of degree at least 2 is called an *internal node*. 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. For a rooted tree T and $u \in V(T)$ the neighbor of u on the path towards the root is called the *parent* of u and a vertex v is a *child* of u if u is the parent of v. A *full binary tree* is a rooted tree where every node is either a leaf or has two children. We do not distinguish between the left and the right child in a binary tree, but for convenience we might refer them as the $left$ and the $right$ child of a node. A *subcubic tree* is a tree where every node has degree at most 3.

A set of vertices $S \subseteq V(G)$ of pair-wise adjacent vertices is called a *clique* and a set $S \subseteq V(G)$ of pairwise non-adjacent vertices is called an *independent set*. Accordingly, a *maximal independent* set is an independent set that is not a subset of any other independent set. A *maximum independent set* is a largest independent set for a given graph G and its size is denoted $\alpha(G)$. The problem of finding such a set is called the Maximum Independent Set problem (ISP) and is an NP-hard optimization problem [37].

2.1.1 Runtime analysis

Let G be a graph with $n = |V(G)|$ vertices. We use big O, O^*, and θ notation to measure the runtime of algorithms. This implies the following for functions f and g:

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

2.2 Decompositions and boolean-width

This section gives definitions and notations regarding boolean-width and decompositions. We do not provide definitions for branch-width, clique-width, and rank-width, as these are not required for understanding the scientific results of this thesis. Definitions and notations regarding tree-width and tree decompositions are presented in Chapter 6 where we use tree decompositions to generate boolean decompositions.

Definition 2.1 (Full and partial decomposition trees)**.** A partial decomposition tree of a graph $G = (V(G), E(G))$ is a pair (T, δ), where T is a full binary tree and δ is a mapping from the nodes of T to non-empty subsets of $V(G)$, satisfying the following: if x is the root of T then $\delta(x) = V(G)$. Moreover if nodes y and z of T are children of a node x then $(\delta(y), \delta(z))$ is a partition of $\delta(x)$. Let $V(T)$ be the nodes of T, then every node $x \in V(T)$ defines a cut $(\delta(x), V(G) \setminus \delta(x))$ of G. If a subtree of T rooted at x has $|\delta(x)|$ leaves then it is called a full decomposition subtree. If T has $|V(G)|$ leaves then (T, δ) is called a full decomposition tree. Note that in a partial decomposition tree (T, δ) of a graph G, if L_a is the set of leaves of T then $V_a = \{\delta(x) : x \in L_a\}$ is a partition of $V(G)$, i.e. $(V_a, V(G) \setminus V_a)$.

In a full decomposition tree there will for each vertex v of G be a unique leaf x of T with $\delta(x) = \{v\}$. Similarly, for each vertex of $\delta(x)$ there is one leaf in a full decomposition subtree rooted at x.

A full decomposition tree is a subcubic tree where each internal node other than root has degree three. If we remove an edge from T, it will result in two subtrees. Most of the FPT algorithms parameterized by boolean-width uses divide-and-conquer dynamic programming on the decomposition tree (T, δ) following the edges of T in a bottom-up fashion. In the conquer step one combines solutions from two cuts given by the edges from a parent node to its two children. Eventually the root stores the solution for the whole graph. The solutions stored at each node depends on the problem being solved.

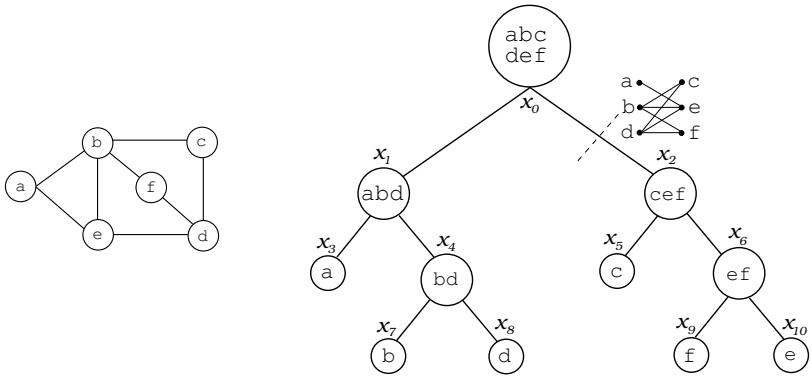

FIGURE 2.1: A graph G and a possible boolean decomposition of G.

Figure 2.1 illustrates a full decomposition tree (T, δ) for the graph given in Figure 2.1. The decomposition tree is rooted at node x_0. The leaves of (T, δ) are mapped to the vertices of G, i.e. $\delta(x_3) = a$. Each edge in the decomposition tree corresponds to a cut. The bipartite graph $BG(\{a, b, d\}, \{c, e, f\}, E)$ induced by the cut corresponding to the edge $\{x_0, x_2\} \in E(T)$ is also shown in Figure 2.1.

Definition 2.2 (Unions of neighborhoods and boolean dimension)**.** The set of unions of neighborhoods of subsets of a set $A \subseteq V(G)$ across the cut $(A, \overline{A})$ is defined as

$$UN(A) = \{N(X) \cap \overline{A} : X \subseteq A\}.$$

where X runs over all possible subsets of A and $UN(A)$ therefore lists all different neighborhoods in $\overline{A}$. For the graph shown in Figure 2.1 $UN(\{a, b, d\}) = \{\emptyset, \{e\}, \{c, e, f\}\}$. For a graph G and for $A \subseteq V(G)$, the boolean dimension of the *cut*$(A, \overline{A})$ is based on a function *bool-dim* : $2^{V(G)} \rightarrow \Re$ and is defined as the logarithm base 2 of the size of the set of unions of neighborhood in $\overline{A}$.

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

The boolean dimension of a cut induced by two adjacent nodes x and y of a decomposition tree (T, δ) i.e. (the edge $\{x, y\} \in E(T)$) is denoted by *bool-dim*$(\{x, y\}, T, \delta)$.

Now, consider the problem of computing the Maximum Independent Set in the graph G of Figure 2.1 using the boolean decomposition tree given in Figure 2.1. Processing the decomposition tree in a bottom up fashion, node x_1 in (T, δ) stores the solution for the subgraph $G[\{a, b, d\}]$ in a table indexed by $UN(\{a, b, d\})$ and node x_2 for the subgraph $G[\{c, e, f\}]$ in a table indexed by $UN(\{c, e, f\})$. Combining solutions from these two tables yields the solution for the root at node x_0, which again is the solution for G.

Definition 2.3 (Boolean-width)**.** The *boolean-width* of a decomposition tree (T, δ) is

$$\textit{boolw}(T, \delta) = \max_{x \in V(T)} \{\log_2 |UN(\delta(x))|\} = \max_{x \in V(T)} \textit{bool-dim}(\delta(x)).$$

The *boolean-width* of a graph G is the minimum boolean-width over all full decomposition trees

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

Boolean-width of a decomposition tree is the maximum boolean dimension over all cuts in that decomposition tree and boolean-width of a graph is the width of the decomposition tree having minimum width among all such trees. Taking the logarithm base 2 in the definition of boolean-width helps in the comparison of boolean-width to other existing graph width parameters as boolean-width of a graph on n vertices then becomes a value between 0 and n.

In many situations it is convenient to define a simpler type of boolean decomposition tree. For this purpose we define a variant of the decomposition tree corresponding to a linear arrangement of $V(G)$.

Definition 2.4 (Caterpillar decomposition)**.** A *caterpillar decomposition* is a binary decomposition tree (T, δ) where every internal node of T has a child that is a leaf. We can construct a caterpillar decomposition (T, δ) from any ordering π of $V(G)$ by letting T be any binary tree where every internal node has a child that is a leaf such that T has $|V(G)|$ leaves and for all $1 \leq i \leq |V(G)|$ let δ map the i-th leaf encountered by a breadth first search starting from the root of T to $\pi(i)$. A caterpillar decomposition is also referred to as a linear decomposition.

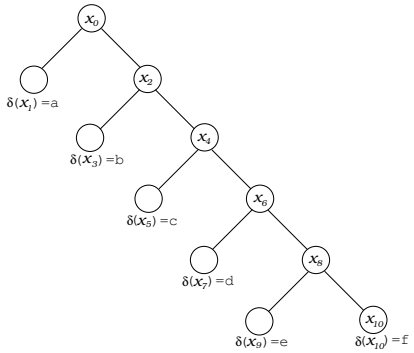

FIGURE 2.2: A caterpillar decomposition of G from Figure 2.1.

Figure 2.2 shows a caterpillar decomposition (T, δ) of the graph G in Figure 2.1, with x_0 being the root of T. The ordering of $V(G)$ used to create (T, δ) is a, b, c, d, e, f. The node $x_6 \in V(T)$ defines via δ the vertex subset $\delta(x_6) = \{d, e, f\}$ of $V(G)$.

Definition 2.5 (Linear boolean-width)**.** The linear boolean-width of G, denoted by $lbw(G)$, is the minimum boolean-width over all caterpillar decomposition trees (T, δ) of G where T has $|V(G)|$ inner nodes, each with an attached leaf, corresponding to a linear arrangement of $V(G)$.

2.3 Properties of boolean-width

While there are several ways to define the boolean-width of a graph, there are some inherent properties that have been proven previously [7, 32, 38]. In following we list some of these, that we will be using in this thesis.

- The number of unions of neighborhoods across the $cut(A, \overline{A})$ is symmetric, i.e. $|UN(A)| = |UN(\overline{A})|$.
- If any cut or the corresponding bipartite graph $BG(A, \overline{A}, E)$ is the union of a complete bipartite graph and some isolated vertices then $|UN(A)| = 2$ and $BG(A, \overline{A}, E)$ has boolean dimension 1.
- If the edges of $BG(A, \overline{A}, E)$ defines a perfect matching of G then $|UN(A)| = 2^{|V(G)|/2}$ and $BG(A, \overline{A}, E)$ has boolean dimension $|V(G)|/2$.
- Since boolean dimension is defined as $\log_2 |UN(A)|$ it follows that, $0 \leq boolw(G) \leq |V(G)|$ for any graph G. As a consequence, the boolean-width of a graph is not always an integer.
- There is a bijection between the union of neighborhoods across a $cut(A, \overline{A})$ and the number of maximal independent sets (MISs) in the bipartite graph $BG(A, \overline{A}, E)$. This implies $|UN(A)|$ = number of maximal independent sets in $(BG(A, \overline{A}, E))$.

- The boolean-width of a graph G will at most be reduced by 1 if a vertex is removed from G. For a graph G and a vertex $v \in V(G)$ this implies that: $boolw(G \setminus v) \leq boolw(G) \leq boolw(G \setminus v)+1$.

2.4 Experimental setup

As this thesis is concerned with the practical use of boolean-width for solving hard problems, it consists to a large extent of implementations of algorithms and heuristics as well as experiments using these. To avoid having to describe the machine configurations in each chapter, we give these here together with the programming languages and data sets used in our experiments.

2.4.1 Machine configuration

We have used two machines for experimentation. All presented results in chapters 3 and 4 have been carried out on a 64-bit Fedora 14 Linux machine with 2.33 GHz Intel Core 2 Duo CPU E6550 and 2GB of main memory. For the rest of the chapters all tests have been performed on a Linux workstation running 64-bit Ubuntu 12.04, with Intel Core i5 CPU 660, 3.33 GHz processors, and 8GB of main memory. The programs written in C are compiled with `gcc` (version 4.5.1) with the `-O3` flag. The implementations done in Java are compiled with `javac` version 1.6.0_30. The programming language used will be specified in each chapter.

2.4.2 Data Set

To evaluate the performance of our proposed algorithms we test out our approaches on graphs originating from various sources. In the following we list all the different sources of our input graphs.

- *TreewidthLIB* [14]: This contains an online repository of around 700 graphs. These are real-world graphs coming from areas such as computational biology, frequency assignment, register allocation problem, evaluation of probabilistic inference systems. We are interested in these graphs because for most of them tree-width upper bounds are known. The effect of preprocessing rules are also known for a subset of these graphs.
- *DIMACS Challenge* [39]: The DIMACS Implementation Challenges provides a collection of graphs where solving specific hard problems can provide guides to realistic algorithm performance. For these graphs different hard problems, such as Graph Coloring, Maximum Independent Set, Maximum Clique have been optimally or approximately solved and the results have been used as benchmarks for many experimental algorithms. We have used graphs from the 2nd DIMACS challenge to evaluate our heuristics as well as to test performance of the dynamic programming algorithm solving the Maximum Independent Set problem.
- *BHOSLIB* [40]: Our experiments also contain graphs from BHOSLIB. For these graphs results for the Maximum Independent Set problem and Maximum Clique

have been reported. These benchmarks are transformed from forced satisfiable SAT benchmarks of Model RB [41], with the set of vertices and the set of edges respectively corresponding to the set of variables and the set of binary clauses in SAT instances. Graphs in BHOSLIB are generated from a simple random graph model as follows:
-Generate n disjoint cliques, each of which has $n\alpha$ vertices (where $\alpha > 0$ is a constant).
-Randomly select two different cliques and then generate without repetitions $pn2\alpha$ random edges between these two cliques (where $0 < p < 1$ is a constant).
-Run Step 2 $rn \ln n - 1$ times (where $r > 0$ is a constant).

- *Random graphs* [42]: We also consider random graphs generated by the *Erdös-Rényi* model. For a constant $0 < p < 1$ the *Erdös-Rényi* model generates a graph G_p of $|V(G_p)| = n$ vertices where for every pair of vertices an edge is added independently with probability p.

- *Named graphs*: Research in graph theory has involved a large number of special graphs. These special graphs have been used as counter examples for conjectures or for showing the tightness of combinatorial results. We consider several well known graphs from the literature and compute exact values of different width parameters for these graphs. Several graphs have names, sometimes inspired by the graph's topology, and sometimes after their discoverer. The definition for each considered graph can be found in MathWorld [43].

Chapter 3

A First Attempt at Generating Boolean Decompositions

FPT algorithms parameterized by the width of the input graph G, are often solved by doing dynamic programming over some decomposition tree of G. Boolean-width also uses this technique. The efficiency of solving problems based on some decomposition tree depends greatly on the width of the decomposition. Thus it is of high interest to generate decompositions of small width. In this chapter we present and evaluate a heuristic to do this

An exponential algorithm that in $O^*(2.52^n)$ time computes a binary decomposition tree of a given graph having optimal boolean-width is given in [38]. It is also known that we can compute a decomposition of boolean-width $2^{2boolw(G)}$ using the algorithm for decompositions of optimal rank-width in FPT time parameterized by $boolw(G)$ [38]. Moreover, there exists polynomial time algorithms for computing decomposition trees with boolean-width polynomial or logarithmic in n for some graph classes [32]. But for general graphs no polynomial time approximation algorithm is known so far. Thus, there is a need for practical algorithms that find decompositions of given graphs of small boolean-width. Heuristics without theoretical quality guarantees are easier to design, provide some guidance, and can serve as a promising practical starting point. Therefore we have developed, implemented, and tested several heuristics to generate boolean decompositions.

This Chapter is based on Paper I and presents a local search heuristic to compute a boolean decomposition tree, where the initial solution is generated using a greedy approach. Local search heuristics have also been used to generate tree decompositions in a practical setting [44–48]. In our approach the key issue is to minimize the width of the boolean decomposition tree. The algorithm is described in Section 3.1. Since tree-width has been extensively studied computationally, we compare the experimental boolean-width upper bounds obtained using this heuristic with existing known tree-width upper bounds on a set of benchmark graphs.

3.1 The algorithm

Given a graph G, our local search heuristic computes a full decomposition tree (T, δ) of G. The initial solution is generated in a greedy fashion. Afterwards search for new solutions in the space of candidate solutions is based on a fine balance between greedy choices and random choices. Each heuristic pass iterates over all decomposition nodes of the current partial decomposition tree, including the children created by this heuristic pass. A newly created tree node always starts out as a leaf node, which maps to a set of vertices of G via δ. In this chapter, for a graph G and a decomposition (T, δ) and node x of T, we denote by P_x the set $\delta(x)$. Thus $\delta(x) = P_x \subseteq V(G)$. Algorithm 1 keeps track of the best full decomposition subtrees found for each $P \subseteq V(G)$ encountered so far and call it $Best(P)$. The heuristic, given in Algorithm 1: LocalSearch(G), runs for a predefined length of time and then returns the best full decomposition found.

Algorithm 1 : LocalSearch(G)

Input: A graph G
Output: A full decomposition tree (T, δ) of G
Step 1: /* Greedily generate initial full decomposition tree */
  Initialize T with $V(T) = \{root\}$, $\delta(root) = V(G)$
  while $\exists$ leaf x of T with $|P_x| > 1$
    $(A, B) \leftarrow$ Split(P_x)
    Add leaves y and z as children of x with $P_y = A$ and $P_z = B$
  for all $x \in V(T)$ store $Best(P_x)$, the subtree rooted at x
Step 2: /* Local Search for better trees */
  for a fixed amount of time **do**
    TryToImproveSubtree($root$)
    if (T, δ) is a full decomposition tree **then** $Best(V(G)) = (T, \delta)$
  return $Best(V(G))$

3.1.1 Greedy initialization

Step 1 of Algorithm 1 greedily generates a full decomposition tree, to serve as the starting tree for the local search in Step 2. The greedy initialization starts with T containing a single node x (as both root and leaf) with $\delta(x) = V(G)$ and repeatedly calls the Split subroutine until we get a full decomposition tree.

A *split* of a set P is a partition into two subsets A and B, with the constraint that $min\{|A|, |B|\} \geq \frac{1}{3}|P|$. The Split($P$) subroutine returns a split (A, B) of P and is given in Algorithm 2. When the Split subroutine is called for the root, one of the partition, A is initially set with $A = \emptyset$ to allow the full benefit of the greedy choices. Then it adds new vertices to A one by one in a greedy fashion while minimizing $|UN(A)|$ and $|UN(P \setminus A)|$, and returns the best split found along the way complying with the split constraint. When Split subroutine is called for node other than root, A is assigned a random half of the vertices of P. Note that the local search in TryToImproveSubtree will for leaves of the current tree make calls to Split(P) but not for $P = V(G)$, since the root of T will never again become a leaf and instead the RandomSwap subroutine described in the next subsection will be applied to the root.

Algorithm 2 : SPLIT(P)

Input: Set of vertices $P \subseteq V$
Output: A partition (A,B) of P such that $min\{|A|, |B|\} \geq \frac{1}{3}|P|$
if $P = V(G)$ **then** $A_1 \leftarrow \emptyset$
else $A_1 \leftarrow$ random half of the vertices in P
$i = 1$
while $|P \setminus A_i| \geq \frac{1}{3}|P|$ **do**
  find $x \in P \setminus A_i$ s.t. $max\{\mathrm{UN}(A_i \cup \{x\}), \mathrm{UN}((P \setminus A_i) \setminus \{x\})\}$ is minimized
  $A_{i+1} \leftarrow A_i \cup \{x\}$
  $i \leftarrow i + 1$
end while
find i such that $max\{\mathrm{UN}(A_i), \mathrm{UN}(P \setminus A_i)\}$ is minimized and $|A_i| \geq \frac{1}{3}|P|$
return $(A_i, P \setminus A_i)$

The objective function optimized locally in SPLIT(P) is $|UN(A)|$, the number of unions of neighborhoods of A, which directly relates to boolean dimension. The computation of $|UN(A)|$ is done in a separate subroutine called UN(A) given in Algorithm 3. This subroutine starts by restricting from the $cut(A, \overline{A})$ to the subsets of vertices (S_1, S_2) having edges going across the $cut(A, \overline{A})$. The list LN is used to accumulate the set $UN(A)$ in a straightforward way. Correctness is easy to show by induction on $|S_1|$.

Algorithm 3 : UN(A)

Input: Set of vertices $A \subseteq V(G)$
Output: $|UN(A)|$, the number of unions of neighborhoods of the cut $(A, \overline{A})$
if $|UN(A)|$ has already been computed **return** the stored value
$S_1 \leftarrow \{v \in A : \exists u \in \overline{A} \wedge \{u, v\} \in E\}$
$S_2 \leftarrow \{v \in \overline{A} : \exists u \in A \wedge \{u, v\} \in E\}$
$LN \leftarrow \{\emptyset\}$ /* neighborhood set accumulator */
for all $u \in S_1$ **do**
  for all $Y \in LN$ **do**
    $X \leftarrow (N(u) \cap S_2) \cup Y$
    if $X \notin LN$ **then** add X to LN
return The number of elements in LN

3.1.2 Local search

The local search used to improve the current decomposition tree is initiated at the root of the tree (T, δ), in Step 2 of Algorithm 1. In the subroutine TRYTOIMPROVESUBTREE(x), given in Algorithm 4, x is a node of the current partial decomposition tree (T, δ) such that $|P_x| > 1$ and the goal is to improve the subtree of T rooted at x. This subroutine has four main parts.

(1) if x is a leaf with $|P_x| > 1$ then find a candidate split of its subset

(2) if x is a non-leaf then find a candidate swap between its two children subsets

(3) conditionally update (T, δ)

(4) for each child of x either use the stored subtree or recurse

For (1) we use the SPLIT subroutine described earlier. For (2) we use the subroutine RANDOMSWAP(A, B) given in Algorithm 5 that randomly swaps vertices between A and B while complying with the split constraint. At the very onset of the local search, the current (T, δ) is the full decomposition tree found by the greedy initialization. However, the current decomposition tree ceases to be full as soon as the split given by RANDOMSWAP(P_y, P_z) in (2) is a good one and (3) updates (T, δ) so that y and z become leaves. If the new P_y is a subset of vertices for which a full decomposition subtree has never been stored, or the stored one is not good enough, then in (4) a recursive call is made to TRYTOIMPROVESUBTREE(y), with y a leaf of the current tree. If in that recursive call the split found in (1) is not good then in (3) we will return with y a leaf of the current (T, δ) having $|P_y| > 1$, which explains the if-statement at the very end of Algorithm 1. Therefore an improved full decomposition tree can only be found if the improvement sustains for all subtrees in the decomposition tree.

Algorithm 4 : TRYTOIMPROVESUBTREE(x)

Input: A node x of T with $\delta(x) = P_x$ and $|P_x| > 1$
Output: Improve or as good as the subtree rooted at x
(1) if x is a leaf **then** $(A, B) \leftarrow$ SPLIT$(\delta(x))$
(2) else
  Let y and z be the children of the node x
  $(A, B) \leftarrow$ RANDOMSWAP(P_y, P_z)
(3) if $max\{\mathrm{UN}(A), \mathrm{UN}(B)\} < boolw(Best(V(G)))$
  then set y and z as new leaf children of x with $P_y = A$ and $P_z = B$
  else if x is still a leaf **then return** /* in case we came from (1) */
(4) if $max\{\mathrm{UN}(P_y), \mathrm{UN}(P_z)\} < boolw(Best(V(G)))$ **then**
  for $w \in \{y, z\}$
    if subtree for P_w is stored and $boolw(Best(V(G))) > boolw(Best(P_w))$
      then use root of $Best(P_w)$ as w.
    else if $|P_w| > 1$ call TRYTOIMPROVESUBTREE(w)
  if the subtree T_x rooted at x is a full subtree of P_x
    then update $Best(P_x)$ to T_x

Algorithm 5 : RANDOMSWAP(P_y, P_z)

Input: $P_y, P_z \subseteq V(G)$ for sibling nodes y and z of (T, δ)
Output: SPLIT(A, B) of $P_y \cup P_z$
Let x be the parent of y and z
choose randomly i in $0..(|P_y)| - \frac{|P_x|}{3})$ and j in $0..(|P_z| - \frac{|P_x|}{3})$
choose randomly $M_i \subset P_y$ and $M_j \subset P_z$ with $|M_i| = i$ and $|M_j| = j$
$A \leftarrow (P_y \setminus M_i) \cup M_j$
$B \leftarrow (P_z \setminus M_j) \cup M_i$
return (A, B)

Note that the local improvements made in the local search are based on randomly swapping vertices between P_y and P_z for two nodes y and z with the same parent. As

usual in local search, there is a fine balance to trying new splits versus sticking with old splits. The goal is to neither get stuck in local minima nor to swap so many nodes that we re-randomize completely and don't get a hill-climbing effect. Note in (4) that we store for each subset P of vertices encountered so far the best found full decomposition subtree $Best(P)$. The decision of when to try new splits and when to use the old splits is tied to the boolean-width of the best subtrees, and to the upper bound on boolean-width of G given by $Best(V(G))$. If the upper bound on boolean-width of the old subtree is below the upper bound on boolean-width of G given by $Best(V(G))$ we continue with the old split.

3.2 Performance analysis

In this section we report the experimental results for the proposed heuristics. We start with discussing some implementation details.

3.2.1 Discussion and implementation details

We implemented the heuristic in Java. Subsets of vertices are stored as bitvectors of length $|V(G)|$, i.e. the number of vertices in the graph. This is an efficient way to store subsets if most of the subsets we store to be of size at least $\frac{|V(G)|}{2}$. Since our implementation of subroutine $\mathrm{UN}(A)$ uses memory proportional to $|V(G)| * |UN(A)|$ bits. We limited the boolean dimension 31, i.e. $|UN(A)| \leq 2^{31}$. The bottleneck is the memory available on our machines. Since $|UN(A)| \leq 2^{min(|A|,|\overline{A}|)}$ the 'boolean-width $\leq$ 31'might become a bottleneck even if the graph has at least 64 vertices. In that case the implementation is handling a list of neighborhoods of size $64 * 2^{31}$ bits which is 16 GB of memory and that is more memory than our desktop had. In Chapter 4 we propose memory efficient and faster methods to compute $|UN(A)|$.

As described, we are currently storing the best full decompositions of subtrees. Since bitvectors are easy to compare they are stored in a binary search tree for quick look-up. Storing all these solutions eats up memory, and for some big graphs this is one of the limiting factors.

Although not specified in the pseudocode, for small subtrees we just return an arbitrary one, since if $|P_x| \leq boolw(Best(V(G)))$ then any full subtree at x will have boolean-width at most $boolw(Best(V(G)))$. The $\textsc{Split}(P)$ subroutine given in Algorithm 2 could be stopped as soon as a subset A_i with low $|UN(A_i)|$ and $|UN(P \setminus A_i)|$ values has been found. It is not clear that this is always better and currently it is not done. The $\mathrm{UN}(A)$ subroutine given in Algorithm 3 does not recompute known values, but otherwise it may seem naive. It forms the inner loop of the heuristic and it is the bottleneck for running on graphs with many vertices. We tried different approaches such as randomly sampling subsets to approximate $|UN(A)|$ and exploiting a correlation between the degree of a vertex and its contribution to $|UN(A)|$. These tests led to only insignificant improvements therefore we kept the naive algorithm.

We did try imposing stronger conditions in order to arrive at better splits sooner, but only minor improvements were seen, and only in some cases.

3.3 Experimental results

All presented results have been carried out on machine configuration specified in Chapter 2. Experiments have been done on graphs from TreewidthLIB.

TreewidthLIB contains 710 graphs. For 482 graphs a tree-width bound is given in TreewidthLIB, and for 426 graphs we give a boolean-width bound using our heuristic. For the comparison we concentrate on the 300 graphs for which we have a bound on both tree-width and boolean-width. Among the rest 410 graphs, there are 126 having only a boolean-width bound, 182 having only a tree-width bound, and 102 having neither. For majority of these 182 graphs our heuristic simply timed out already at the greedy initialization stage. Note that for these 182 graphs, if we were given the decomposition of low tree-width k, we could easily have produced a decomposition of boolean-width at most $k+1$, using the $O(nk^2)$ algorithm which can be deduced from [31]. This approach will be discussed further in Chapter 6.

We now summarize our findings for the 300 graphs having both a tree-width bound and a boolean-width bound. Firstly, the boolean-width bound is always better than the tree-width bound, with the ratio of the tree-width bound divided by the boolean-width bound ranging from 1.15 to 29, with an average of 3.13. Not surprisingly, the ratio increased with higher edge density. In Figure 3.1 we have plotted this ratio against the edge density of the graphs for the 300 graphs. The trend line shows the growth of the ratio with the edge density.

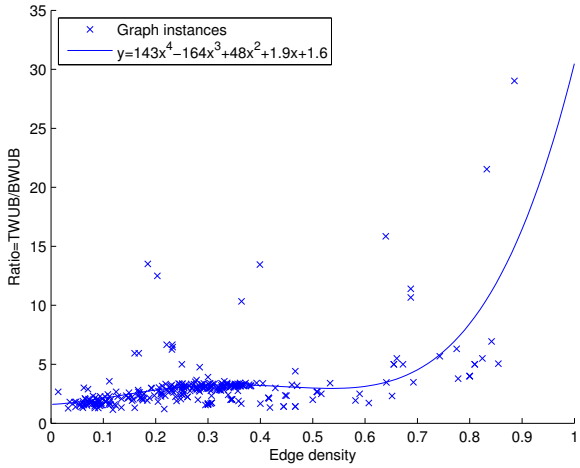

FIGURE 3.1: Ratio (tree-width divided by boolean-width) versus edge density for all the 300 graphs for which heuristically computed upper bounds are known.

Our heuristic algorithm starts with greedily finding a full decomposition tree which gives an *Initial Bound* on boolean-width and then improves this bound iteratively. We set the maximum time for local search including greedy initialization to 1000 seconds. But for some graphs it took more than 1000 seconds to finish the greedy initialization and we

have not tried local serach for those ones, for example, graphs BN_9, 1bkb. In the experiments we kept track of the decrease in the boolean-width over time. In Figure 3.2 and Figure 3.3 the upper bounds on boolean-width, i.e. the values of $boolw(BEST(V(G)))$, are shown as they decrease over time, for the two graphs called eil51.tsp ($V(G)$=51 and $E(G)$=140) and miles1500 ($V(G)$=128 and $E(G)$=5198). For the graph eil51.tsp the *Initial Bound* was 9.1 after less than a second, then at the 'knee'of the curve before the improvement decays we found a *Fast Bound* of 6.2 after 4 seconds, and finally the *Best Bound* of 5.8 was found after 124 seconds. For each graph, we can likewise speak of three bounds: i) the Initial Bound given by the greedy initialization, ii) a Fast Bound found at the knee of the curve, and iii) the Best Bound found possibly after a long runtime.

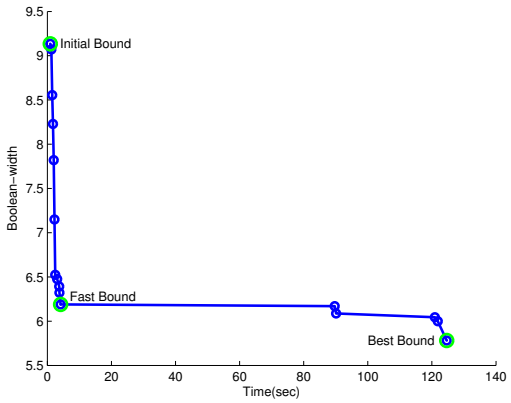

FIGURE 3.2: Improvement of boolean-width upper bound as the local search progresses over time, for the graph eil51.tsp ($V(G)$=51, $E(G)$=140).

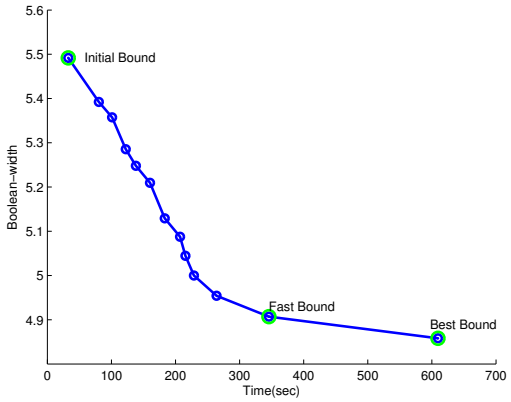

FIGURE 3.3: Improvement of boolean-width upper bound as the local search progresses over time, for the graph miles1500 ($V(G)$=128, $E(G)$=5198).

From Figure 3.2 and Figure 3.3 it can be observed that the rate of improvement of boolean-width upper bound over time using local search varies from graph to graph.

This might be caused by the random choices during swapping or the internal graph structures.

In Table 3.1 we summarize results for 8 selected graphs having a good variety of number of vertices $V(G)$, edge density *density*, time in seconds to find Initial Bound, Fast Bound, and Best Bound on boolean-width, its best known tree-width upper bound,TWUB, and Ratio=TWUB/BWUB(Best Bound). The graphs are sorted by this Ratio. The miles1500 graph is translated from the Stanford GraphBase. The zeroin.i.1 and mulsol.i.5 graphs originate from the 2nd DIMACS implementation challenge [39] and are generated from a register allocation problem based on real code. The queen8_12 also comes from the DIMACS [39] graph coloring problems and is an example of the n-queens puzzle. The graph 1awd is from the field of computational biology with each vertex representing a single side chain and each edge representing the existence of a pairwise interaction between the two side chains. The graph celar06-wpp is a frequency assignment instance. The graph BN_28 originates from Bayesian Network from evaluation of probabilistic inference systems at UAI 2006. The graph eil51.tsp is a Delaunay triangulation of a Traveling salesman problem.

TABLE 3.1: Results for selected graphs

		Edge	Initial Bound		Fast Bound		Best Bound			
Graph	V	*density*	BWUB	Time(s)	BWUB	Time(s)	BWUB	Time(s)	TWUB	Ratio
miles1500	128	0.64	5.5	32.6	4.9	345.7	4.8	609.6	77	15.85
zeroin.i.1	211	0.19	4.0	74.1	3.8	116.2	3.7	168.0	50	13.51
mulsol.i.5	186	0.23	6.4	55.3	5.4	130.0	4.9	365.2	31	6.25
queen8_12	96	0.30	16.7	3055	16.7	3055	16.7	3055	65	3.91
1awd	89	0.27	13.3	67.5	11.1	521.1	10.8	702.9	38	3.52
celar06-wpp	34	0.28	4.5	0.1	3.2	0.8	3.0	4.8	11	3.37
BN_28	24	0.18	3.3	0.02	2.3	0.05	2.0	0.3	5	2.50
eil51.tsp	51	0.11	9.1	0.9	6.2	4.1	5.8	124.6	9	1.55

3.3.1 Small grid graphs

We also ran our heuristic on graphs corresponding to the $n \times n$ grid. However, Algorithm 3 is too memory-intensive and we limit the size to $n \leq 9$. Square grids are sparse graphs having tree-width n and the upper bound on boolean-width is about $0.695n$ [38]. Therefore, the values computed by our heuristic are substantially close to the optimal values.

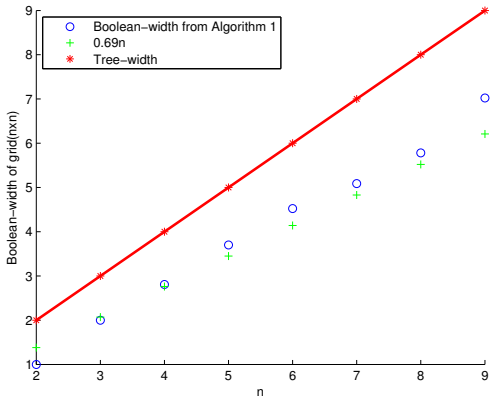

FIGURE 3.4: Upper-bound on boolean-width of square grids.

3.3.2 Rank-width upper bound vs boolean-width upper bound

Finally, we note that inspired by Paper I, on which this chapter is based, a heuristic algorithm for computing rank-width was published [24]. In this work experimental rank-width upper bounds for a set of graphs has been compared to experimental tree-width and boolean-width upper bounds. It was observed that the rank-width upper bound of most of the graphs are lower than the known values for tree-width upper bound, whereas the boolean-width heuristic in Algorithm 1 is able to find decompositions of significantly lower width. They experimented for graphs that have between 25 to 256 vertices. Figure 3.5 (from [24]) plots boolean-width upper bounds of 114 graphs with respect to their rank-width upper bounds. The dotted line in Figure 3.5 marks the equality of both parameters.

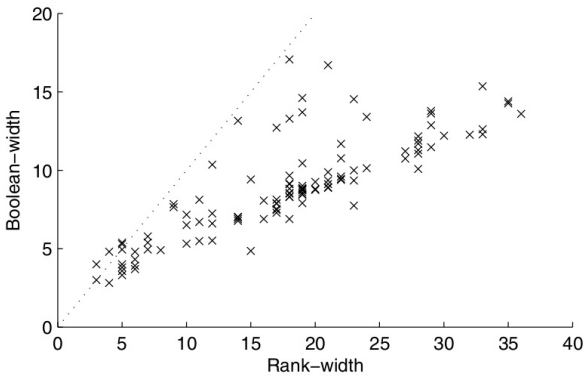

FIGURE 3.5: Comparison of experimental rank-width upper bounds with boolean-width upper bounds.

It was also reported in [24] that for the considered set of graphs, the boolean-width to rank-width ratio was between 1.44 and 0.32 with an average of 0.57.

3.4 Conclusion

In this chapter we have presented the first experimental study on computing decompositions that can achieve a low width parameter even for non-sparse graphs. Experiments show the potential strength of boolean-width versus tree-width, in particular for graphs of higher edge density. This study has made us aware of some of the bottlenecks in the overall process of generating boolean decompositions. The most pressing issue is to have a faster way to compute the boolean dimension of a cut which we address in the next chapter. Another issue is to come up with heuristic for computing a good upper bound on $|UN(A)|$. Although alternative approaches may not asymptotically improve the running-time of the heuristic, these can still have a positive effect on the running time. Moreover, in our experiments the heuristic ran for a predefined amount of time for each graph before stopping. But there are several ways of experimenting with the stopping criteria, for example based on the size of the input graph, or on the fraction of time since an improved tree was last found.

Chapter 4

Counting Maximal Independent Sets

As was stated in Definition 2.2 in Chapter 2, the boolean width of a decomposition tree is defined using the number of unions of neighborhoods across the cuts of the tree. Each cut corresponds to a bipartite graph BG, induced by the vertex partition of the cut. The algorithm presented in Chapter 3 used the $UN(A)$ definition for computing boolean-width. Following this work it was observed [38, 49] that the number of unions of neighborhoods in BG coincides with the number of maximal independent sets (MISs) in BG.

This relationship offers a different and potentially faster way of computing the boolean dimension of a particular cut. However, it is known that counting MISs is #P-hard even for planar bipartite graphs [50] and designing an approximation algorithm for this problem is also intricate [51, 52]. In consequence we started looking at existing exact algorithms for generating or counting all MISs in general graphs and ended up devising our own adaptation for the problem. This is based on certain vertex selection criteria and the use of checking for connectedness. We have performed extensive experiments comparing our algorithm with the previous best algorithms for this problem using both real world as well as synthetic input graphs to facilitate our selection for the best one.

Our proposed algorithm was initially developed for counting MISs in general graphs and was presented in Paper II. In the following we first describe the algorithm along with accompanying experiments and comparisons with other algorithms for this problem while using general graphs. We then show results from applying the algorithm to bipartite graphs.

4.1 Previous algorithms for counting and enumerating MISs

In the following we first present previously suggested algorithms for counting or enumerating all MISs of a general graph $G = (V(G), E(G))$. For the enumeration problem we present algorithms that have been used in experimental studies and that are fairly straight forward to implement. We also outline the counting algorithm by Gaspers et al. [53]. Since our main interest is to count the number of MISs, we describe all algorithms as applied to this problem.

The Bron-Kerbosch algorithm in its basic form uses recursive backtracking to list all maximal cliques in a given graph [54]. In the following we present the dual of this algorithm, so that instead of cliques the algorithm counts all MISs in $G = (V(G), E(G))$.

Given three vertex sets R, P, and X, Algorithm 6: BKMis(R, P, X) finds all MISs that include all vertices in R, any possible legal subset of the vertices from P, and none of the vertices in X. The recursion is initiated by setting both R and X to $\emptyset$ and $P = V(G)$. Within each recursive call, the algorithm considers in turn every vertex in P for inclusion in R. Thus for each $v \in P$ the algorithm makes a recursive call in which v is moved from P to R and any neighbor of v is removed from P and X. In any subsequent call where both P and X are empty, R is counted as an MIS. This will find all maximal independent set extensions of R that contain v. When the recursive call returns, v is moved from P to X before the algorithm continues with the next vertex in P.

Intuitively, one can think of the algorithm as having already found the MISs that contain any vertex from X. Thus any set that does not dominate every vertex in X cannot be a new MIS.

Algorithm 6 : BKMis(R, P, X)

Input: Three vertex sets R, P, and X.
Output: Number of MISs containing all vertices in R, some from P and none from X.
if $P \cup X = \emptyset$ **then**
  Count R as a MIS
end if
for each vertex $v \in P$ **do**
  BKMis$(R \cup \{v\}, P \setminus N_G[v], X \setminus N_G(v))$
  $P \leftarrow P \setminus \{v\}$
  $X \leftarrow X \cup \{v\}$
end for

The Bron–Kerbosch algorithm is not output-sensitive meaning that it does not run in polynomial time per generated set. The worst-case running time of the Bron–Kerbosch algorithm is $O(3^{\frac{n}{3}})$, matching the Moon and Moser bound [55, 56]. In 1965 Moon and Moser showed that, the maximum number of distinct maximal cliques in an $|V(G)| = n$ vertex graph (with $n > 1$) is $3^{\frac{n}{3}}$, $4 \cdot 3^{\frac{(n-4)}{3}}$, or $2 \cdot 3^{\frac{(n-2)}{3}}$, according to the value of n mod 3 being $0, 1, 2$ respectively. The complement is applicable for counting maximal independent sets.

The lower bound can be realized by forming a graph from the disjoint union of copies of K_3 (the complete graph on 3 vertices i.e. a triangle). Each maximal independent set has exactly one vertex from each of these complete subgraphs from which the formula follows.

Tomita et al. presented an improved variant of the Bron-Kerbosch algorithm by using a pivoting heuristic [57]. Here we present its dual for computing MISs. In Algorithm 6, $|P|$ recursive calls are made, one for each vertex in P. The pivoting strategy seeks to reduce this number. Consider a vertex $u \in P \cup X$. It follows that no vertex in $N_G[u]$ has been added to R so far. But for the current R to be expanded to a MIS at least one vertex of $P \cap N_G[u]$ must be included in R, otherwise R will not be maximal. Thus once the *pivot*

u has been selected, it is sufficient to iterate over the vertices in $P \cap N_G[u]$ for inclusion in R. The idea in Algorithm 7: TomitaMis(R, P, X) is then to choose u such that this number is as small as possible. Computing both the pivot and the vertex sets for the

Algorithm 7 : TomitaMis(R, P, X)

Input: Three vertex sets R,P and X.
Output: Number of MISs containing all vertices in R, some vertices from P and no vertex from X.
if $P \cup X = \emptyset$ **then**
  Count R as a MIS
end if
Choose a pivot $u \in P \cup X$ that minimizes $|P \cap N_G(u)|$
for each vertex $v \in P \cap N_G[u]$ **do**
  TomitaMis$(R \cup \{v\}, P \setminus N_G[v], X \setminus N_G(v))$
  $P \leftarrow P \setminus \{v\}$
  $X \leftarrow X \cup \{v\}$
end for

recursive calls can be done in time $O(|P|(|P| + |X|))$ within each call to the algorithm using an adjacency matrix, giving an overall running time of $O(3^{\frac{n}{3}})$. Experimental comparisons have shown that the maximal clique algorithm by Tomita et al. is faster by orders of magnitude compared to other algorithms [57]. However, both the theoretical analysis and implementation rely on the use of an adjacency matrix representation of the input graph. For this reason, the algorithm has limited applicability for large graphs, whose adjacency matrix may not fit into working memory [58].

Eppstein et al. [59] also proposed a variant of the Bron-Kerbosch algorithm. On the top level this algorithm is similar to the Bron-Kerbosch algorithm, although the vertices are processed according to a degeneracy ordering. Such an ordering can be found by repeatedly selecting and removing a minimum degree vertex. The algorithm then makes $|V|$ calls to the algorithm by Tomita et al., each time with R initially set to the next vertex in the ordering and with P and X updated accordingly. With this setup the algorithm can be implemented to list all maximal cliques of an n-vertex graph in time $O(dn3^{\frac{d}{3}})$, where a graph has degeneracy d if every subgraph has a vertex of degree at most d. In a recent study Eppstein and Strash [58] show that the algorithm is highly competitive with the algorithm by Tomita et al. This is particularly true for large sparse graphs where it in many cases outperform the algorithm by Tomita et al. by orders of magnitude.

Gaspers et al. gave a fast exponential time algorithm of complexity $O(1.3642^n)$ for counting the number of MISs in a graph [53]. This running time is lower than the Moon and Moser bound, something that is possible since the algorithm, unlike the previous mentioned ones, does not enumerate the MISs but only counts their number.

The structure of the algorithm is similar to TomitaMis in that a vertex $u \in P \cup X$ is selected as a pivot according to a degree based criterion before branching on the vertices in $P \cap N[u]$. But unlike the previous algorithms, it will in each call first try if any of seven reduction rules (see [53]) can be applied to achieve a smaller but equivalent instance. If this is possible then the instance is reduced accordingly before calling the recursive function again. We note that all rules but one, will return the value given by the following recursive call. The only exception being a rule which checks if there exist

two vertices u and v such that their current neighborhoods are identical (*false twins*). In this case v will be removed from the graph and the value of the recursive call will be returned plus the number of MISs discovered in this call that contained u. Another difference is that the algorithm tests if there is a vertex in X having no neighbor in P indicating that the current configuration cannot be expanded to a MIS. If this is the case then the algorithm returns immediately. In the paper it is also noted that if the graph at some stage should become disconnected then the algorithm is called (recursively) for each of its connected components, and the product of the returned values then gives the number of MISs. As far as we know there has been no study of how practical the algorithm is. We refer the interested reader to [53] for the details of the algorithm.

4.2 A new algorithm

In the following we present a simple recursive branching algorithm for counting the number of MISs in a graph. Our algorithm is based on locating and exploiting vertex separators of the graph, and is similar in spirit to the algorithm by Lipton and Tarjan for computing a Maximum Independent Set (IS) in a planar graph [60].

The Lipton and Tarjan algorithm initially finds a vertex separator $S \subset V(G)$ such that $|S| = O(\sqrt{n})$ and such that no component of $G \setminus S$ contains more than $\frac{2}{3}|V(G)|$ vertices. This is possible since $G = (V(G), E(G))$ is assumed to be planar. Then for every independent set I_S of S the algorithm recursively finds a maximum independent set for each connected component of $G \setminus (S \cup N_G(I_S))$. The solution giving the combined largest solution is then the maximum independent set of G. The running time of the algorithm is $2^{O(\sqrt{n})}$.

We modify the Lipton and Tarjan algorithm to compute the number of MISs by using ideas from BKMis and TomitaMis. Note however first that it is not possible to use the algorithm of Lipton and Tarjan to count MISs. The reason for this is that if we pick a particular independent set I_S from a separator S in $G = (V(G), E(G))$ and (recursively) calculate the number of MISs in each component of $G[V \setminus (S \cup N_G(I_S))]$, then it is not given that I_S together with every combination of MISs from each of the components will form a MIS in G as some combinations might leave undominated vertices in $S \setminus I_S$.

The new algorithm, Algorithm 8: CCMis, is recursive and uses two vertex sets P and X to count the number of MISs in $G[P \cup X]$ containing any combination of vertices from P while using none of the vertices in X. Thus if $P \cup X = \emptyset$ this will be counted as one MIS. Also, similar to the algorithm by Gaspers et al. if there exist a vertex in X that is not adjacent to any vertex in P then the algorithm will return 0, as this indicates that the current solution cannot be expanded into a complete MIS. The algorithm also tests at each level of recursion if $G[P \cup X]$ is connected. If this is not the case then the recursive procedure will be called once for each connected component and the product of the number of MISs in each component will be returned. Checking for connectedness and listing the components is done using a linear depth first search through $G[P \cup X]$.

In the case that none of the mentioned conditions apply, the algorithm picks one remaining vertex v from P and then performs two recursive calls, first to compute the number of MISs containing v and then to compute the number of MISs excluding v. Finally, the sum of these two numbers is returned. When counting the number of MISs containing v, any vertex in $N_G[v]$ is first removed from P and X as these will be dominated by v.

Similarly, when counting the number of MISs not containing v, the vertex v is moved from P to X as it must then be dominated by some other vertex in P in an MIS. Note that it is only following a recursive call where v is set to be in the current MIS that the structure of $G[P \cup X]$ will change so that there is any possibility of getting a disconnected graph. The recursion is initiated by setting $X = \emptyset$ and $P = V(G)$.

Algorithm 8 : CCMis(P, X)

Input: Two vertex sets P and X.
Output: Number of MISs in $G[P \cup X]$ containing only vertices from P.
if $P \cup X = \emptyset$ **then**
  return 1
end if
if $\exists w \in X$ with no neighbor in P **then**
  return 0
end if
if $G[P \cup X]$ is not connected **then**
  $count \leftarrow 1$
  for each connected component $CC(V_{CC}, E_{CC})$ of $G[P \cup X]$ **do**
    $count \leftarrow count *$ CCMis$(V_{cc} \cap P, V_{cc} \cap X)$
  end for
  return *count*
end if
Select a vertex $v \in P$ to branch on
$count \leftarrow$ CCMis$(P \setminus N_G[v], X \setminus N_G(v))$
$count \leftarrow count +$ CCMis$(P \setminus \{v\}, X \cup \{v\})$
return *count*

As we explain in the following the algorithm CCMis differs substantially from the previous algorithms in which order the vertices are selected from P to branch on. It is clear from the description of CCMis that one can select any vertex $v \in P$ to branch on. Thus one could similar to the previous algorithms use degree based information when selecting the branching vertex v. Picking a maximum degree vertex could be advantageous for the first recursive call as it would give a maximum reduction in the size of $G[P \cup X]$, thus making it more likely that the remaining graph is disconnected. Picking a minimum degree vertex could be advantageous for the second recursive call as there would be fewer remaining vertices in P that could dominate v. However, as our main interest is in computing the number of MISs for sparse graphs we use a different selection criterion that exploits this. Algorithm 8: CCMis has a considerable advantage over the Bron-Kerbosh type enumeration algorithms whenever the remaining graph becomes disconnected. This follows since the CCMis algorithm does not have to generate every MIS but only needs to find the number of MISs in each connected component and then to multiply these numbers together. Although the algorithm by Gaspers et al. also exploit connected components in this way, their algorithm is bound to using a degree based criterion when selecting a pivot. Thus this might limit how often the remaining graph becomes disconnected. Since we have no restrictions in CCMis when selecting the branching vertex $v \in P$ we do so with the sole objective that the remaining graph should become disconnected.

Prior to running the algorithm we compute a *nested dissection* ordering $\pi = \{v_1, v_2, \ldots, v_{|V(G)|}\}$ on the vertices of $G = (V(G), E(G))$ [61]. Such an ordering strives to number vertices that make up a (preferably small) separator $S \subseteq V(G)$ of G

first, with the added constraint that the remaining components of $G \setminus S$ should be of roughly equal size. This is then repeated recursively for each connected component. One can also view a nested dissection ordering as an *elimination tree* [62]. This tree displays the separators in π, with vertices in a separator S making up a path hanging off the preceding separator S' on the component containing S. Within each separator, a vertex $v_j \in S$ will be a child of the highest numbered vertex $v_k \in S$ where $k < j$. If v_j, $j \neq 1$, is the first ordered vertex in S then v_j will be a child of the last ordered vertex of S', where S' is as defined above. It follows that a low elimination tree height is an indication that it was possible to (recursively) partition the graph using small separators.

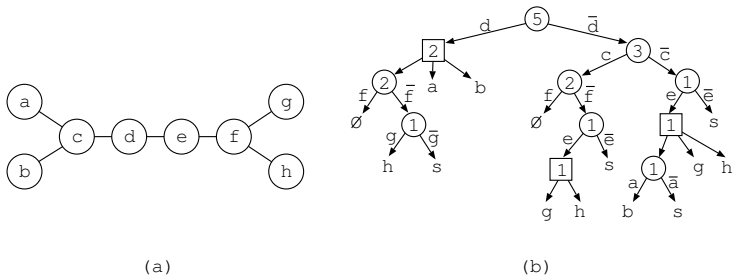

FIGURE 4.1: A possible execution of Algorithm 8.

The strategy we employ is now to choose the first vertex $w \in \pi$ that is also in $P \cup X$. We have two cases for selecting the vertex v to branch on. If $w \in P$ then we set $v = w$ and if $w \in X$ then we select v to be a vertex in $P \cap N_G(w)$. Such a vertex must exist since the algorithm would already have returned if $w \in X$ had no neighbor in P. The effect of following π in this way is that we will only expand solutions where each vertex in S is eventually included in the current MIS or is being dominated. Thus we are ensured that the remaining graph will become disconnected. Note that the strategy of picking a vertex in $N_G(w)$ to branch on whenever $w \in X$ is similar to the pivoting strategy in TomitaMis. Comparing with the algorithm by Lipton and Tarjan the difference is that even though we follow the separator structure, for a particular separator S we allow for vertices in $N_G(S)$ to be assigned values before deciding exactly which vertices from S should be in the MIS.

The tree in Figure 4.1(b) shows the recursion tree of the algorithm when applied to the graph in Figure 4.1(a). Each time the algorithm branches on a particular vertex is denoted by a round node, where the left branch denotes that the branching vertex is in the current MIS and the right branch that it is not. Whenever the remaining graph consist of just one vertex in P we only show the name of the vertex as it must be in any MIS. When the remaining graph is empty we write $\emptyset$, and if a particular branch cannot be extended to a MIS we write s. We use a square node to indicate when the graph has become disconnected and then draw one branch for each connected component. The number inside each node is the number of MISs returned by a particular branch.

With the current description of the algorithm there is still some freedom as to the order in which the branching vertices are selected. As already pointed out, we can reorder the vertices within a separator in π. Also, once a vertex $v \in S$ has been chosen to branch on then in the configuration where v is considered to be out of the current MIS, we are free to decide the order in which we pick vertices from $N_G(v) \cap P$ to dominate v. We will expand further on these issues in Section 4.3.

4.3 Experimental results

In the following we describe experiments performed to evaluate the presented algorithms. Machine configuration is the same as for in Chapter 3. The programs are written in C (compiled with `gcc` (version 4.5.1) with the `-O3` flag) and Java (compiled with `javac` version 1.6.0_30). Each reported running time is the average of five runs.

4.3.1 General graphs

For this experiments we use graphs from TreewidthLIB [14]. We have chosen a set of 22 graphs drawn from areas such as computational biology, frequency assignment, register allocation problem, and evaluation of probabilistic inference systems. The graphs were chosen so that in most cases our implementation of TomitaMis would terminate within 24 hours. This limited the maximum size to about 400 vertices. Moreover we also avoided most graphs having fewer than 10^6 MISs as all algorithms would spend less than a second on these. Table 4.1 gives the statistics for the chosen 22 graphs. Here p gives the edge density, eth gives the elimination tree height, while $MISs$ gives the number of maximal independent sets. In addition to these graphs we have performed experiments using rectangular grids.

Our first set of experiments concerns a comparison between the algorithm by Gaspers et al. and TomitaMis. In addition to the regular algorithm by Gaspers et al. we also implemented variants of it where we only apply the reduction rules at regular intervals, the most extreme case being when the reduction rules are not used at all. Since the algorithm by Gaspers et al. is by far the most complex of the considered algorithms, we have performed these comparisons using Java as this offers better support for more complex data structures such as sets. The results of the comparisons on nine representative graphs can be seen in the Figure 4.2. Here the first seven graphs are the ones marked with * in Table 4.1, while the 8th graph is a path on 40 vertices, and the 9th and 10th graphs are grids of size 7×7 and 8×8, respectively. The numbers are reported relative to the performance of the regular algorithm by Gaspers et al. (G100). G50 denotes the algorithm where the reduction rules are only applied in 50% of the recursive calls and G0 where they are not used at all.

TABLE 4.1: Description for benchmark real world graphs from TreewidthLIB.

Graph No.	Graph name	V	E	p	eth	$MISs$
1*	risk	42	83	0.01	13	66498
2*	pigs-pp	48	137	0.12	17	131402
3*	1sem	57	570	0.35	41	12405
4*	BN_100	58	273	0.16	31	160312
5*	1r69	63	692	0.35	46	22993
6*	1ail	69	631	0.26	44	134201
7	macaque71	71	444	0.18	30	182044
8	jean	80	508	0.16	22	1251960
9*	1aba	85	886	0.25	54	1067404
10	david	87	406	0.11	22	$4.41\text{x}10^{7}$
11	celar02	100	311	0.06	29	$2.87\text{x}10^{10}$
12	celar06	100	350	0.07	22	$2.72\text{x}10^{10}$
13	1lkk	103	1162	0.22	62	$1.44\text{x}10^{7}$
14	1fs1	114	1351	0.21	73	$5.10\text{x}10^{7}$
15	1a62-pp	120	1507	0.21	73	$7.56\text{x}10^{7}$
16	miles250	128	387	0.05	36	$1.75\text{x}10^{13}$
17	anna	138	493	0.05	23	$2.75\text{x}10^{10}$
18	mulsol1.i.5	186	3973	0.23	47	$3.33\text{x}10^{9}$
19	celar05	200	681	0.03	36	$7.86\text{x}10^{20}$
20	zeroin.i.3	206	3540	0.17	43	$1.29\text{x}10^{7}$
21	zeroin.i.2	211	3541	0.16	43	$1.81\text{x}10^{7}$
22	BN_93	422	1705	0.02	38	$4.55\text{x}10^{11}$

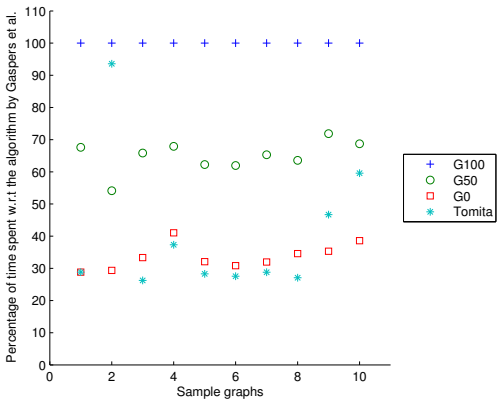

FIGURE 4.2: Relative performance of TOMITAMIS compared to the algorithm by Gaspers et al..

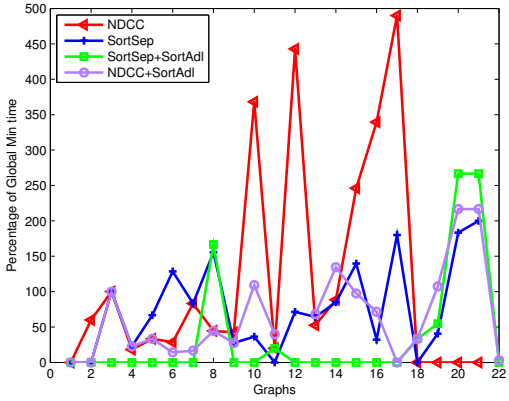

FIGURE 4.3: Relative performance of different CCMIS algorithms.

As can be observed there is no advantage in using the reduction rules, and when they are not used at all the performance is very similar to that of TOMITAMIS. Based on these results we did not pursue the algorithm by Gaspers et al. any further. For the remaining experiments all algorithms have been implemented in C as this gave considerable faster code compared to using Java.

We then compared BKMIS, TOMITAMIS, and the algorithm by Eppstein et al. These experiments showed that, as expected, TOMITAMIS outperformed BKMIS, while there was little difference between TOMITAMIS and the algorithm by Eppstein et al. We note that this last observation does not contradict the results in [58] as these were concerned with enumerating cliques in sparse graphs which is equivalent to enumerating MISs in dense graphs, while we are enumerating MISs in sparse graphs.

Our next set of experiments concerns different variants of CCMIS where we use METIS [63] to precompute a nested dissection ordering. The time spent on this was insignificant compared to the algorithm itself and is not included in the timings. The versions we tried include the basic algorithm (NDCC) where the vertices are processed for branching according to the ordering given by Metis and versions where we reorder the vertices within each separator and also the relative order of the neighbor lists. Similar in spirit with TOMITAMIS we tried a version where one branches on a vertex v in the current separator such that $|P \cap N_G[v]|$ is minimized. This slowed down the algorithm compared to NDCC and we therefore switched to presorting each separator based on their degree in G. We label this algorithm SortSep. Next we considered the order in which the neighbor lists are ordered. This is of importance when trying to dominate a vertex v currently in X. Consider a vertex w with several undominated neighbors in the current separator S. In the configuration where w is in the current MIS all neighbors of w will be dominated, thus reducing the number of undominated vertices in S. In the configuration where w is in X each undominated neighbor of w will have one vertex less that must be tried to dominate it. Based on these observations we implemented a version (SortAdl) where the adjacency list of every vertex v was presorted according to the number of neighbors each vertex has in the same separator as v belonged to. We also tried to compute this ordering on the fly using the number of remaining undominated vertices in the current separator but this only increased the running time. In Figure 4.3 we display

the relative running time for all four combinations of these approaches. For each graph we report the relative performance compared to the best algorithm for that graph. In all of these implementations we only check if the graph is disconnected if the previous call to CCMis moved a vertex into the current MIS.

The average distances from the best algorithm was for SortSep + SortAdl 36%, for NDCC 185%, for SortSep 172%, for NDCC + SortAdl 167%. Thus it is clear that sorting both the separators and the neighbor lists is crucial for performance.

Finally, we tried two versions of CCMis where the selection criterion for which vertex to branch on was strictly based on the degree of the remaining vertices, one where we always selected the vertex of minimum degree and one where we selected the vertex of maximum degree (MaxDegCC). Both of these were considerably slower than any of the other CCMis variations. The absolute running times for MaxDegCC, TomitaMis, NDCC, and SortSEp+SortAdl are given in Table 4.2. We note that the average distance from the best algorithm for each graph was for MaxDegCC 1371% and for TomitaMis 1.6×10^6%.

Table 4.2: CPU time(sec) for benchmark real world graphs from TreewidthLIB

Graph	1	2	3	4	5	6	7	8	9	10	11
TomitaMIS	0.07	0.22	0.02	0.25	0.03	0.14	0.31	0.69	1.09	29.05	20095.5
NDCC	0.01	0.08	0.02	0.26	0.04	0.09	0.11	0.13	1.04	1.03	0.06
MaxDegCC	0.02	0.23	0.02	0.46	0.05	0.14	0.15	0.09	1.59	0.31	4.56
SortSep+SortAdl	0.01	0.05	0.01	0.22	0.03	0.07	0.06	0.24	0.73	0.22	0.06
Graph	12	13	14	15	16	17	18	19	20	21	22
TomitaMIS	10648.1	16.24	61.5	87.85	-	32716.1	2722.0	-	23.81	32.66	135407.1
NDCC	0.38	8.7	16.4	84.38	3.56	1.18	0.03	187.63	0.06	0.06	1303.0
MaxDegCC	8.67	15.3	40.9	82.06	7.17	0.69	0.18	-	0.84	0.86	1658.85
SortSep+SortAdl	0.07	5.7	8.7	24.37	0.81	0.2	0.04	290.57	0.22	0.22	76.12

As can be seen the running time of TomitaMis is by far the highest, for some graphs the algorithm did not finish. Also, following a nested dissection ordering is advantageous in most cases, and as already noted presorting the separators and neighbor lists further emphasizes this effect.

We have also experimented with how often one should check if the graph is disconnected in CCMis. We tried version where we only checked for a certain percentage of the calls, where we only checked once a separator had been dominated, and checking when the remaining graph is at least of some predefined size. From these tests we conclude that when the remaining graph has at least 10 vertices, then checking every time after some vertex has been added be in the current MIS was the best option.

4.3.2 Bipartite graphs

In the following we describe experiments using Algorithm 8 on bipartite graphs. As our main motivation for this is computing boolean dimension we compare our results with those from Algorithm 3 presented in Chapter 3.

Note that these algorithms solve slightly different problems. While the algorithms presented in this chapter only counts the number of MISs, Algorithm 3 generates and stores each MIS. This can require a substantial amount of memory; i.e. for a graph with 100

vertices and boolean dimension ≥ 30 at least 32GB of memory is needed, even if subsets of vertices are stored as bitvectors of length $|V(G)|$. This is beyond the capacity of our computer. But if the intention is only to evaluate the boolean dimension of a particular cut then this is clearly not needed.

Algorithm 3 has been implemented in Java as it uses the underlying graph data structure designed for experimenting with the generation of boolean decompositions. So in order to expedite the comparison we have implemented the **MaxDegCC** version of Algorithm 8 in Java as well. The reason for not implementing the **SortSep+SortAdl** version is both simplicity and that this requires access to METIS which has been done through writing to files, thus slowing down the execution considerably. Moreover as experiments show, the **MaxDegCC** version is considerably faster than Algorithm 3. In the following we report and analyze results for random bipartite graphs as well as bipartite graphs obtained from random bipartition of real world graphs.

4.3.2.1 Random bipartite graphs

For these experiments we have generated random bipartite graphs $BG = (A, B, E)$ with $|A| = |B| = 30$ and varying the average degree. Starting from 1 the average degree increases by 1 until it reaches 30. We have generated five random graphs of same average degree and the reported timing for each graph is the average of five runs, both for Algorithm 3 and Algorithm 8. For random bipartite graphs with $min(|A|, |B|) > 30$, Algorithm 3 takes more than a minute when the average degree is in between 2 to 5, i.e. the edge density ranges from 0.05 to 0.2. This is the reason for limiting the the size of the partitions to 30.

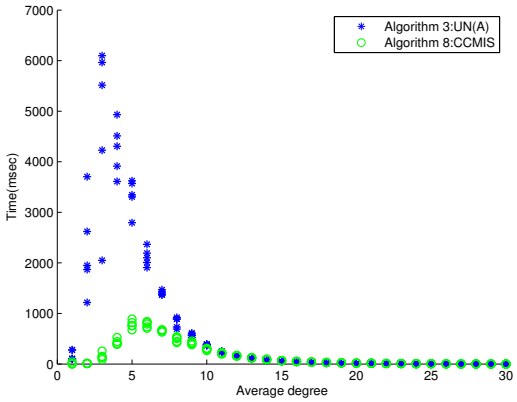

FIGURE 4.4: Comparing time for CCMIS and UN(A) with respect to average degree for bipartite graphs $BG = (30, 30, E)$.

Figure 4.4 illustrates the time required in msec for CCMIS and UN(A). The time is plotted against the average degree of the vertices in the bipartite graphs. .

Figure 4.5 shows the time required in msec for CCMIS and UN(A) for the same set of bipartite graphs. But this time the results are plotted against the number of MISs

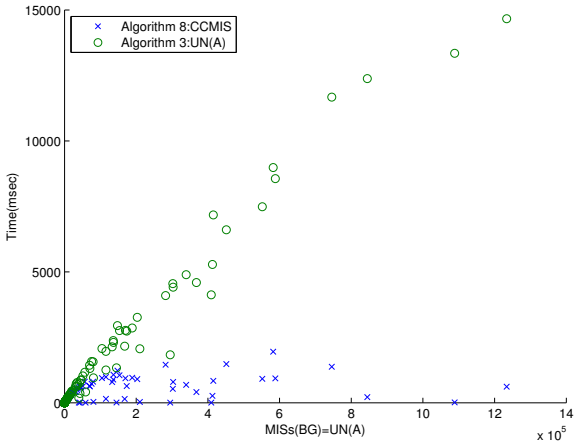

FIGURE 4.5: Comparing time for CCMIS and UN(A) with respect to the number of MISs for bipartite graphs $BG = (30, 30, E)$.

in BG. The time and the number of MISs reported is the average of five runs. From this figure it can be seen that the time to list grows linearly with the number of MISs and takes considerably more time than counting MISs in particular when the boolean dimension is high.

4.3.2.2 Real-world graphs

In this set of experiments we use 18 graphs from Table 4.1 as listed in Table 4.3.

TABLE 4.3: CPU time(sec) for bipartite graphs generated from the graphs listed in Table 4.1

Graph	V	E	$\|A\|$	$\|B\|$	$\|E(BG)\|$	p_{BG}	MISs	*bool-dim*(A)	T_{CCMIS}	$T_{\text{UN}(A)}$
pigs-pp	48	137	24	24	67	0.12	26532	14.70	0.54	2.10
1sem	57	570	29	28	295	0.36	4556	12.15	1.26	0.46
BN_100	58	273	29	29	144	0.17	117457	16.84	2.39	2.59
1ail	69	631	35	34	316	0.27	33344	15.03	2.71	1.47
jean	80	254	40	40	129	0.08	18388	14.17	0.37	1.66
1aba	85	886	43	42	424	0.23	171572	17.39	5.58	27.30
david	87	406	44	43	206	0.11	283050	18.11	2.09	12.11
celar02	100	311	50	50	156	0.06	$3.16\text{x}10^7$	24.92	0.58	-
celar06	100	350	50	50	172	0.07	$6.33\text{x}10^7$	25.92	0.48	-
1lkk	103	1162	52	51	585	0.22	1509424	20.53	31.88	-
1fs1	114	1351	57	57	687	0.21	7359279	22.81	53.91	-
1a62-pp	120	1507	60	60	775	0.22	$1.91\text{x}10^7$	24.19	223.48	-
miles250	128	387	64	64	201	0.05	$4.53\text{x}10^9$	32.08	1.05	-
anna	138	493	69	69	252	0.05	$2.01\text{x}10^7$	24.27	0.51	-
mulsol.i.5	186	3973	93	93	1970	0.23	965115	19.88	0.56	310.25
celar05	200	681	100	100	344	0.03	$3.68\text{x}10^{15}$	51.71	54.79	-
zeroin.i.3	206	3540	103	103	1787	0.17	240274	17.87	2.01	61.74
BN_93	422	1705	211	211	847	0.02	$1.02\text{x}10^{10}$	33.26	4.42	-

To obtain bipartite graphs we split the vertices into two random groups A and B of equal size (differing by one if $|V(G)|$ is odd). The edges in the bipartite graphs are then the edges crossing the $cut(A, B)$. We are interested in these bipartite graphs as they are similar to the ones we consider while computing the boolean-width of a decomposition tree. We ran Algorithm 8 for all these graphs but Algorithm 3 only for those having $|UN(A)| < 2^{20}$. When $|UN(A)| > 2^{20}$ it took more than 10 minutes to finish the computation and oftentimes this ran out of memory. The boolean dimension, *bool-dim*(A), the time required for CCMis (T_{CCMIS}), and the time required for UN(A) ($T_{\text{UN}(A)}$) are reported in Table 4.3.

From these above mentioned comparisons it is obvious that Algorithm 3: UN(A) is substantially and often in orders of magnitude slower than Algorithm 8: CCMis (for example, mulsol.i.5). This is particularly true when the boolean dimension is high and the graph is sparse.

4.4 Conclusion

The duality of the number of maximal independent sets in a bipartite graph and boolean dimension shows an interesting way to look at these problems. Both algorithms considered in the previous section are exponential, but counting MISs decomposes into small subproblems when the remaining graph becomes disconnected. It is evident from the results that the multiplicative factor we get while combining MISs from different connected components allows us to compute significantly higher boolean dimensions.

We have experimented on dense graphs as well, but the sparse graphs are the ones where listing takes substantially more time than branching. Using CCMis can give us far-reaching improvement is case of memory requirement as well. In addition, as this approach can rapidly evaluate large cuts it could also help us to speed up the initialization process described in Chapter 3. Furthermore, the local search in Algorithm 1 could be able to explore the search space to a greater extent within the same predefined amount of time.

Chapter 5

Speeding up the generation of Boolean Decompositions

In this chapter we investigate several ways of speeding up the generation of boolean decompositions. This can be done by applying simple preprocessing rules to reduce the instances as well as computing the boolean dimension faster. In Chapter 4 we have already discussed a faster and more memory efficient way to compute boolean dimension. In this chapter we first develop and test three very simple preprocessing rules for reducing instances prior to computing boolean-width. A number of experiments in Section 5.2 show the contribution of both the preprocessing and applying Algorithm 8 for accelerating Algorithm 1 from Chapter 3.

Reduction rules can be very useful when solving NP-hard problems, particularly for large instances that can possibly be reduced to a manageable size. For graph problems reduction rules typically consists of removing or contracting edges or vertices in such a way that a solution to the reduced problem can be expanded to a solution of the original one. The rules that are used for doing this are often simple and based on local properties. Thus they can be applied very efficiently and save considerable time in the ensuing computations.

Preprocessing rules have been used in the computation of tree-width and weighted tree-width and have been studied in practical settings [64–66]. The experimental results reported in these papers show that there are simplification routines that give significant size reductions for many practical instances, making it more feasible to compute the tree-width of those graphs exactly or approximately. Theoretical analysis of the potential of preprocessing for tree-width, studying whether there are efficient preprocessing procedures whose effectiveness can be proven, and what the resulting size bounds look like have been investigated in [67], with the help of kernelization [68], and analyzed based on parameterized complexity theory [69, 70].

5.1 Reduction rules

Given a graph G, a vertex v is *safe* (with respect to boolean-width) if removing v and all incident edges does not change the boolean-width of G. Similarly, a reduction rule is *safe* if applying the reduction rule does not change the boolean-width of the given

graph. In the following we present three simple safe reduction rules. We will show the correctness of the rules in Section 5.1.1. In the following $u, v \in V(G)$. The three rules are as follows.

1. Islet rule : If $deg(v) = 0$ then remove v.

2. Pendant rule : If $deg(v) = 1$ and $|E(G)| > 1$ then remove v.

3. Twin rule : If $|E(G)| > 1$ while $N[u] = N[v]$ or $N(v) = N(u)$ (i.e. v and u are twins) then remove v.

5.1.1 Proof of correctness

In the following we assume that there will always be at least one edge left after the preprocessing. This means that $boolw(G) \geq 1$. Assume now that (T, δ) is a boolean decomposition of $V(G) \setminus v$. We will show for each of the three rules how this can be expanded to a boolean decomposition of $V(G)$ without changing the boolean width.

To do so we will pick two vertices $x, y \in V(T)$ such that x is a leaf in T and $\{x, y\} \in E(T)$, i.e. y is the parent of x. Let (T', δ') be the tree obtained from (T, δ) by deleting $\{x, y\}$ and adding two new nodes w and z and three new edges $\{y, w\}, \{w, x\}$, and $\{w, z\}$ as shown in Figure 5.1. We further set $\delta'(z) = v$ while keeping $\delta'(a) = \delta(a)$ for each $a \in V(T)$. It follows that (T', δ') is a boolean decomposition of G. We now show for each rule how x should be chosen so that the boolean dimension of the cuts in (T', δ') does not increase from the boolean dimension of the cuts in (T, δ).

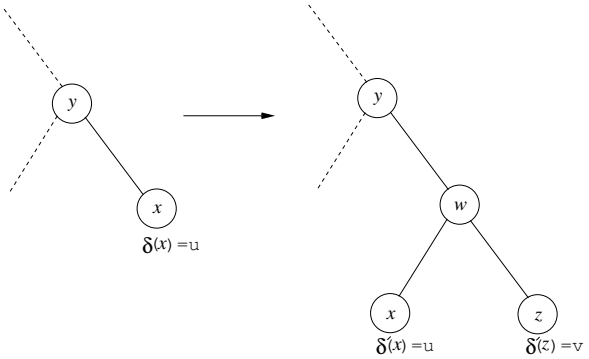

FIGURE 5.1: Reversing the reduction rules by adding the reduced vertex v.

Lemma 5.1. *If v is an islet and x is any leaf in T, then $boolw(T, \delta) = boolw(T', \delta')$.*

Proof. Note first that the boolean dimension of a cut having only a vertex in one partition of the cut is 0 if the vertex is an islet and 1 otherwise. Thus $bool\text{-}dim(\{w, z\}, T', \delta') = 0$. Since v is not incident on any edge crossing any cut in (T', δ') it follows that v will not influence the boolean dimension of any cut. In particular, for the cuts defined by $\{w, x\}$ and $\{y, w\}$ any crossing edge must be incident on u and we have $bool\text{-}dim(\{w, x\}, T', \delta')$ $= bool\text{-}dim(\{y, w\}, T', \delta') = bool\text{-}dim(\{y, x\}, T, \delta)$. In the same way it is straight forward to see that the boolean dimension of all other cuts will remain unchanged from (T, δ) to (T', δ') and the result follows. □

Lemma 5.2. *If v is a pendant, $\delta(x) = u$, and $\{v, u\} \in E(G)$, then $boolw(T, \delta) = boolw(T', \delta')$.*

Proof. The boolean dimension of $\{w, z\} \in E(T)$ is 1 as $\{u, v\} \in E(G)$ is the only edge crossing this cut. Similarly, the boolean dimension of $\{w, x\} \in E(T)$ is 1 as u is a singleton with at least one adjacent edge. For all other cuts v is not adjacent to any crossing edge and does not contribute to the boolean dimension. It follows that $bool\text{-}dim(\{y, x\}, T, \delta) = bool\text{-}dim(\{y, w\}, T', \delta') \leq 1$ and that the boolean dimension of all other cuts remain unchanged from (T, δ) to (T', δ'). Since $boolw(T, \delta) \geq 1$ and $\{y, w\}, \{w, x\}$, and $\{w, z\}$ all have boolean dimension at most 1 in (T', δ') the result follows. □

Lemma 5.3. *If $\delta(x) = u$ and v and u are twins, then $boolw(T, \delta) = boolw(T', \delta')$.*

Proof. Since u and v are twins it follows that $bool\text{-}dim(\{w, x\}, T', \delta') = bool\text{-}dim(\{w, z\}, T', \delta')$ which again is equal to $bool\text{-}dim(\{y, x\}, T, \delta)$. Also, v does not contribute to any new neighborhood across any cut in (T', δ') that did not already exist in (T, δ). Note that none of these observations are dependent on if $\{u, v\} \in E(G)$ or not. It follows that $boolw(T, \delta) = boolw(T', \delta')$. □

Since lemmas 5.1 through 5.3 are true for any decomposition tree T, they are in particular true for one giving the optimal boolean width decomposition. We thus have that $boolw(G \setminus v) = boolw(G)$ as long as v is removed according to one of rules 1 through 3.

5.1.2 Tree-width reductions that do not work

Tree-width preprocessing has been studied for quite some time and there are several effective reduction rules. Removing a simplicial vertex is one such rule. A vertex v is simplicial in a graph G if the neighbors of v form a clique in G. We note that both an islet and a pendant are simplicial. In addition, tree-width has the almost simplicial rule. In the following we explain why the simplicial vertex rule is not safe for boolean-width preprocessing.

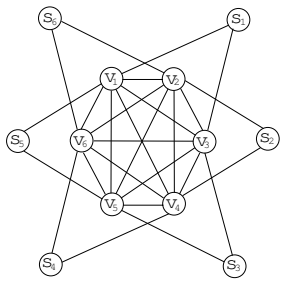

FIGURE 5.2: Simplicial vertices $s_1, s_2, \ldots, s_6$.

In Figure 5.2 the vertices $v_1, v_2, \ldots, v_6$ form a clique and vertices $s_1, s_2, \ldots, s_6$ are simplicial as the neighborhood of each of these vertices form a clique. The boolean-width of any boolean decomposition of the induced subgraph containing $v_1, v_2, \ldots, v_6$ is 1, as all vertex subsets have exactly two different neighborhoods (one being $\emptyset$) across all the

cuts in the decomposition tree. Now while introducing the simplicial vertices, whenever we have a simplicial vertex along with one other vertex on one side of a cut, this vertex subset will have more than two different neighborhoods across the cut, thus increasing the boolean-width of the decomposition. It follows that a simplicial rule would not be safe. On the other hand twin rule cannot be used for tree-width preprocessing.

It is well known that, the tree-width of a minor of G is never larger than the tree-width of G itself [71]. A graph H is a minor of a graph G if H can be obtained from G by zero or more vertex deletions, edge deletions, and edge contractions (edge contraction is the operation that replaces two adjacent vertices v and w by a single vertex that is connected to all neighbors of v and w). On the contrary, boolean-width of a minor of G can be larger than the boolean-width of G. This is most obvious from the fact that the boolean-width of a complete graph is 1, and every graph is a minor of some complete graph.

5.1.3 Implementation

To preprocess a graph, rules 1 through 3 are applied iteratively until no rule is applicable. When a boolean decomposition of the reduced graph has been generated, we can undo the reduction operations in reverse order to get a boolean decomposition of the original graph. Each time a rule is applied, a vertex and its remaining incident edges are removed. To be able to execute rules 1 and 2 it is sufficient to maintain a counter for each vertex giving the number of remaining neighbors. When this reaches 0 or 1 the appropriate rule can be applied. It is clear that this can be done in linear time. For discovering twins we maintain a bit vector representation of the adjacency lists. Then we can efficiently determine if two vertices u and v are twins by testing if $N(v) \setminus u = N(u) \setminus v$. Note that this test is true both if v and u are true or false twins. Thus to discover all twins we loop over all pairs of vertices. We note that this could have been done faster by only considering distance 2-neighbors, but the preprocessing still only takes a small fraction of the time compared to the boolean-width computation.

5.2 Experimental results

In this section we report on results from computational experiments when applying the three very simple preprocessing rules on a number of graphs. We also report the effect of using these preprocessing rules prior to running Algorithm 1. Moreover, we compare the running times of the greedy initialization part of Algorithm 1 using both Algorithm 3 and Algorithm 8 for computing the boolean dimension. Finally, we show how preprocessing and Algorithm 8 together help improving the running time of Algorithm 1 and allow us to run greedy initialization for larger graphs. Note that in Chapter 3 most of the graphs having more than 270 vertices timed out during the greedy initialization process.

5.2.1 Preprocessing

We have implemented the reduction rules in Java and use the machine configuration described in Chapter 2. For performance evaluation we use both random graphs and graphs from TreewidthLIB [14] and DIMACS [39]. The graphs listed in Table 5.1 are

from probabilistic networks developed for real-life instances. The graphs alarm, boblo, diabetes, link, munin*, oesoca*, pignet2, and several others are taken from medical applications. There are networks for agricultural purposes, such as barley and mildew. The water graph models a water purification process and oow* graphs are developed for maritime applications. For the graphs in Table 5.1 experiments using tree-width preprocessing rules has been reported in [66].

In Table 5.1 and 5.2 we report the number of vertices removed by rules 1, 2, and 3 for each graph. For the twin rule the number of edges removed is also listed. Column 8 and 9 of Table 5.1 and 5.2 show the total number of vertices and edges removed by all three rules. Column 10 reports the percentage of vertices removed. In addition the two rightmost columns of Table 5.1 list the total number of vertices and edges removed by the preprocessing for tree-width (TW_{pp}) reported in [66].

Table 5.1 shows that the number of vertices and edges removed by tree-width preprocessing rules is substantially larger than what is removed by the boolean-width preprocessing rules on this particular set of graphs. Among the boolean-width preprocessing rules the pendant rule has removed the most vertices except for the two graph pigs and pignet2 where the twin rule reduces the most. On average around 20% of the vertices are removed by the preprocessing for boolean-width, with a minimum of 0% and a maximum of 48%. Whereas for tree-width on average over 50% of the vertices are removed for these 23 graphs.

TABLE 5.1: Effect of reduction rules when applied to graph instances from probabilistic networks

			Removed by individual rules				Removed by (1+2+3)			Removed by TW_{pp}	
Graph	V	E	Islet(1)	Pendant(2)	Twin(3)	E(Twin)	V	E	$\%V$	V	E
wilson-hugin	21	27	0	12	1	3	13	15	61.9	21	27
oow_bas	27	54	0	1	1	2	2	3	7.41	27	54
water	32	123	0	2	1	3	3	5	9.38	11	29
oow_trad	33	72	0	0	0	0	0	0	0	10	18
mildew	35	80	0	1	0	0	1	1	2.86	35	80
alarm	37	65	0	10	3	7	13	17	35.14	37	65
vsd-hugin	38	62	0	15	1	2	16	17	42.11	38	62
oesoca	39	67	0	15	2	5	17	20	43.59	39	67
oow_solo	40	87	0	2	1	2	3	4	7.5	13	24
oesoca42	42	72	0	17	1	3	18	20	42.86	42	72
barley	48	126	0	1	2	5	3	6	6.25	23	50
ship-ship	50	114	0	0	0	0	0	0	0	26	49
oesoca+-hugin	67	208	0	19	0	0	19	19	28.36	53	133
munin1	189	366	0	29	3	7	32	36	16.93	123	178
boblo	221	328	0	104	1	4	105	108	47.51	221	328
diabetes	413	819	0	2	4	11	6	13	1.45	297	543
pigs	441	806	0	7	24	48	31	55	7.3	394	672
link	724	1738	10	73	1	2	84	75	11.6	416	580
munin2	1003	1662	0	184	0	0	184	184	18.34	838	1211
munin4	1041	1843	0	179	1	2	180	181	17.29	826	1201
munin3	1044	1745	0	192	0	0	192	192	18.39	962	1472
munin_kgo	1066	1730	0	184	0	0	184	184	17.26	1066	1730
pignet2	3032	7264	0	2	157	314	159	316	5.24	2030	3534

Table 5.2 also shows the effect of the preprocessing rules when applied to graphs from TreewidthLIB but generated from other sources than probabilistic networks. The graphs are sorted by the number of vertices. The zeroin.i.*, inithx.i.1, and mulsol.i.* graphs originate from the 2nd DIMACS implementation challenge [39] and are generated from a register allocation problem based on real code. The other graphs are collected from the Stanford GraphBase [72]. These representative graphs are selected from a large set

of preprocessed graphs and have a good variety both in the number of vertices and edge densities.

TABLE 5.2: Effect of reduction rules when applied to graph instances generated from other sources than probabilistic networks

			Removed by individual rules				Removed by (1+2+3)		
Graph	V	E	Islet(1)	Pendant(2)	Twin(3)	E(Twin)	V	E	$\%V$
zeroin.i.1-pp	54	1267	0	1	51	1265	52	1266	96.3
zeroin.i.2-pp	57	1097	0	0	18	710	18	710	31.58
jean	80	254	3	18	3	13	24	31	30
anna	138	493	0	26	2	4	28	30	20.29
mulsol.i.5	186	3973	10	0	30	937	40	937	21.51
mulsol.i.1	197	3925	59	0	20	746	79	746	40.1
zeroin.i.3	206	3540	49	0	34	1322	83	1322	40.29
zeroin.i.1	211	4100	85	0	12	973	97	973	45.97
zeroin.i.2	211	3541	54	0	31	1256	85	1256	40.28
fpsol2.i.1	496	11654	227	0	23	1358	250	1358	50.4
homer	561	1628	5	215	5	2	225	217	40.11
inithx.i.1	864	18707	345	0	63	2632	408	2632	47.22
BN_26	3025	14075	0	1010	0	0	1010	1010	33.39

In Table 5.2 the majority of the vertices are removed by the twin rule. We also note that the number of edges removed by the twin rule can be significantly higher than the number of vertices removed by this rule. For the set of graphs listed in Table 5.2 on average around 37% of the vertices are removed, with a minimum of 20% and a maximum of 96%. Though the graphs in Table 5.1 are not reduced as much by the boolean-width preprocessing as by the tree-width preprocessing rules, Table 5.2 shows that there are graphs that can be considerably simplified using the boolean-width preprocessing rules. It was also observed that most of the graphs from protein structure and Delaunay triangulation do not reduce at all using these preprocessing rules.

5.2.2 Speeding up Algorithm 1

In the following we show the effect of applying the simple preprocessing rules prior to running Algorithm 1. The graphs presented in Table 5.3 are from TreewidthLIB and already used in Chapter 3 for the experiments on Algorithm 1. We report the number of vertices, V_{pp} and edges, E_{pp} for each preprocessed graph. Columns 6 and 7 present the boolean-width upper bound of the greedy initial decompositions, (Initial Bound) and time required in seconds for the original graphs respectively when using Algorithm 1. Similarly, columns 8 and 9 present the initial bound and time required for the preprocessed graphs respectively when using Algorithm 1. Each reported running time in tables 5.3 and 5.4 is the average of five runs. It can be observed from Table 5.3 that though the initial boolean-width upper bounds for the original and the preprocessed graphs compare favorably with each other, the running times for the preprocessed graphs are substantially lower for these graphs. The average reduction in running time is 287% for the preprocessed graphs.

TABLE 5.3: Comparison of greedy initialization for original and preprocessed graphs

	Original		Preprocessed		Original		Preprocessed	
Graph	V	E	V_{pp}	E_{pp}	Initial Bound	Time(s)	Initial Bound	Time(s)
BN_28	24	49	13	34	3.32	0.02	2.58	0.02
celar06-wpp	34	156	28	96	4.46	0.11	4.17	0.1
oesoca42	42	72	24	52	3.7	0.12	3.32	0.1
jean	80	508	56	223	5.21	1	5.04	0.9
anna	138	493	110	463	8.35	15.8	8.28	5.1
mulsol.i.5	186	3973	146	3036	6.38	55.3	4.58	16.6
zeroin.i.3	206	3540	123	2218	5.39	65.7	5.32	32.8
zeroin.i.1	211	4100	114	3127	4	74.1	4	16.9

In Table 5.4 we compare the time required for greedy initialization in Algorithm 1 using UN(A) and CCMIS. To do this we report the boolean-width upper bound of the greedy initial decompositions (Initial Bound), time required in seconds using UN(A), T_1, and CCMIS, T_2. Table 5.4 shows that for this set of graphs the time required using UN(A) is substantially larger than the time required using CCMIS. For these selected graphs the average reduction in running time is 289%.

TABLE 5.4: Comparison of the time required for greedy initialization in Algorithm 1 using UN(A) and CCMIS

Graph	V	E	Initial Bound	T_1	T_2
david	87	406	7.06	2.3	1.9
graph05-pp	91	394	14.22	312.6	37.5
1aac	104	1316	12.64	150	105.6
BN_9	105	1259	18.43	4358	391.1
1a62	122	1516	13.65	740	107.9
celar10-pp	133	646	10.74	24.2	9.6
anna	138	493	8.35	15.7	6.4
pr152	152	428	10.64	28.3	14.2
mulsol.i.5	186	3973	6.38	55.3	30.7
munin4-wpp	271	724	10.61	93.9	91.1

Results from the final set of experiments are reported in Table 5.5. We have used a set of graphs having more than 271 vertices to test how preprocessing and CCMIS affects Algorithm 1 when running on larger graphs. For the graphs in Table 5.5 we ran greedy initialization on the preprocessed graphs having V_{pp} vertices and E_{pp} edges and CCMIS was used for the computation of boolean dimension. The values in the column labeled Initial Bound gives the resulting boolean-width found after T_2 seconds. The reported times are the average of two runs.

TABLE 5.5: Greedy initialization using CCMIS on large graphs

Graph	V	E	V_{pp}	E_{pp}	Initial Bound	T_2
a280	280	788	280	788	12.64	147.7
Link-pp	308	1158	305	1152	21.86	188.6
Diabetes-wpp	332	662	329	651	7.13	192.3
Link-wpp	339	1194	336	1175	21.18	243.4
celar10	340	1130	326	1106	9.91	193.2
celar08	458	1655	433	1622	8.77	621.2
fpsol2.i.1	496	11654	246	10296	6.78	391.1

Though preprocessing and faster computation of boolean dimension allow us to run the greedy initialization on larger instances, greedy selection still takes a significant amount of time as the size of the input grows. As greedy selection requires $O(n^2)$ computations of the boolean dimension which can be exponential in n, this is still most likely the bottleneck in the whole decomposition generation process.

5.3 Conclusion

The size and structure of the graphs is a limiting factor when experimenting with algorithms and heuristics for computing boolean decompositions. As the experimental results show, preprocessing can have a significant impact on reducing the size of the input graphs. Since there is a direct connection between the running time of a decomposition algorithm and the size of the input, preprocessing can both help in reducing the running time and also allow us to investigate a larger search space within a fixed amount of time. The suggested rules are all very simple. It would be of interest to investigate if there exist more complex rules that can be used to reduce the graphs even further. Moreover, for larger graphs we should look for heuristics that generate decompositions in polynomial time.

Chapter 6

Generating a Boolean Decomposition from a Tree Decomposition

Tree-width has been inspirational for the investigation of many other well known width parameters. A number of NP-hard graph problems can be solved in linear time for graphs of bounded tree-width [8, 33, 73]. Several studies have shown that this is not only of theoretical interest but can also be successfully applied to optimally solve many optimization problems [74–76]. Independent of the problem being solved one must first compute a tree decomposition of a given graph. Computing a tree decomposition of the minimum width is known to be NP-hard [77].

A significant amount of work has been done to determine the tree-width of graphs [10, 78, 79]. In practice, algorithms with a large constant hidden in the running time, or with the tree-width in the exponent have limited use [10]. For this reason a number of heuristics have been developed with easily computable upper and lower bounds for tree-width [11–14].

It is known that, for a given graph G, $boolw(G) \leq tw(G) + 1$ [31]. If we are given a tree decomposition of tree-width k, we can generate a boolean decomposition of width at most $k+1$, using the $O(nk)$ algorithm that can be inferred from [31]. In this chapter we investigate the practical applicability of this approach. A promising outcome of this will not only benefit the dynamic programming algorithms using boolean decompositions, but also establishes an interesting application of tree decompositions. To do this we have used one of the simplest heuristics described in [12] to first produce a tree decomposition. which we then use to compute a boolean decomposition of the graph.

6.1 Generating a tree decomposition

We start this section by giving the definition of a tree decomposition and tree-width. We also discuss some properties of a tree decomposition which facilitates the understanding of the subsequent transformation algorithm.

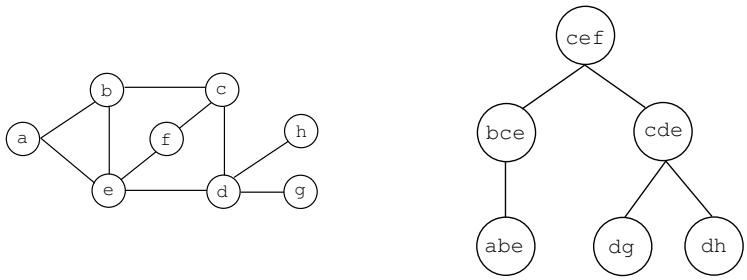

FIGURE 6.1: A graph G and a corresponding tree decomposition of $tw(G) = 2$.

Definition 6.1 (Tree-width and tree decompositions)**.** A *tree decomposition* of a graph $G = (V(G), E(G))$ is a pair (X, T) where $T = (I, F)$ is a tree and $X = \{X_i | i \in I$ is a collection of subsets of $V(G)\}$ called *bags*, such that

- for all $v \in V(G)$, there exists an $i \in I$ with $v \in X_i$, i.e. $\bigcup_{i \in I} X_i = V(G)$
- for each edge $\{u, v\} \in E(G)$, there is an $i \in I$ with $u, v \in X_i$, and
- for each $v \in V(G)$, the set of nodes $I_v = \{i \in I | v \in X_i\}$ forms a connected subtree of T.

The *width* of a tree decomposition $(\{X_i | i \in I\}, T = (I, F))$ equals $max_{i \in I}\{|X_i| - 1\}$. The *tree-width* of a graph G is the minimum width over all tree decompositions of G.

The Definition 6.1 implies all vertices and edges of G are in some bag in the decomposition tree (X, T). Suppose S be a node (bag) in (X, T), then $X - S$ has $X_1, X_2, ..., X_d$ components and subgraphs induced by $G[X_1], G[X_2], ..., G[X_d]$ has no vertices in common and there are no edges between them. An example of a tree decomposition is shown in Figure 6.1.

In the following we review a heuristic for computing the tree-width of a graph $G = (V(G), E(G))$. This is based on first computing a permutation π of the vertices of $V(G)$ called an *elimination ordering*. The fill-in graph H of G with respect to π is constructed as follows: for $i = 1$ to $|V(G)|$, we iteratively add an edge between each pair of higher numbered neighbors of the i-th vertex in the order. In [12] a recursive procedure is given that builds a tree decomposition from a permutation or elimination ordering where the width of the decomposition is given by the maximum number of higher numbered neighbors of any vertex in the fill-in graph H. We thus see that any algorithm that computes an elimination ordering also can be seen as a heuristic for computing an upper bound on tree-width.

6.2 Computing a boolean decomposition from a tree decomposition

As shown in [31] it is possible to transform any tree decomposition into a boolean decomposition. The intuition behind this result is as follows:

Consider a tree decomposition (X, T) and two adjacent bags X_1 and X_2. Let (X_1, T_1) and (X_2, T_2) be the two components of (X, T) after deletion of the edge $\{X_1, X_2\} \in (X, T)$. Then deleting $V(G[X_1]) \cap V(G[X_2]) = S$ from $V(G)$ disconnects G into the two subgraphs $G[(V(G[X_1]) - S) = A]$ and $G[(V(G[X_2]) - S) = B]$ as shown in Figure 6.2. More precisely these two subgraphs do not share any vertices and there is no edge with one end in each of them. Therefore for $\forall Y \subseteq S$ the boolean dimension of $cut(A \cup Y, B \cup (S \setminus Y)) \leq \log_2(2^{|S|}) \leq |S|$, as S is a separator of G.

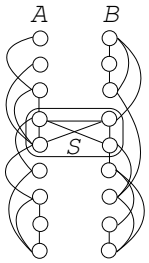

FIGURE 6.2: For $\forall Y \subseteq S$ boolean dimension of $cut(A \cup Y, B \cup (S \setminus Y)) \leq |S|$.

Since this is true for any tree decomposition, including the optimal one, where we have $|S| \leq tw(G) + 1$, we have that $boolw(G) \leq tw(G) + 1$ as long as we can refine and transform a tree decomposition into a complete boolean decomposition without increasing the boolean dimension of any cut beyond this value. In the following we show how this is possible. There are some structural differences between a tree decomposition and a boolean decomposition. The bags of a tree decomposition can be overlapping, and there might not be one leaf bag for every vertex of G. To alleviate this we apply a series of transformations to get a boolean decomposition from a given tree decomposition. The procedure is outlined in Algorithm 9, and described in more detail in sections 6.2.1 through 6.2.4.

Algorithm 9 : TDTOBOOLD(X, T)

Input : A tree decomposition (X, T) of graph $G = (V(G), E(G))$
Output : A boolean decomposition (T, δ) of $G = (V(G), E(G))$
Root the tree decomposition
(1) Remove multiple copies of any vertex $v \in V(G)$ from the tree decomposition such that v remains only in the bag which is closest to the root
(2) Copy every non-leaf bag to a new leaf node as a child of the original bag
(3) Copy every vertex from each bag into its parent node if the vertex is not already present in the parent
(4) Make the decomposition tree both binary and full
return the boolean decomposition obtained by (1)-(4)

6.2.1 Making nodes of the tree disjoint

We root the tree decompositions at the last bag returned by the tree decomposition process. In a tree decomposition every edge appears in some bag and bags containing

the same vertex form a connected subtree. But in a boolean decomposition tree a vertex of a parent node only appears in one of its children, i.e. children of the same parent have disjoint vertices. So in order to make the vertex partition disjoint we retain only the topmost instance of every vertex.

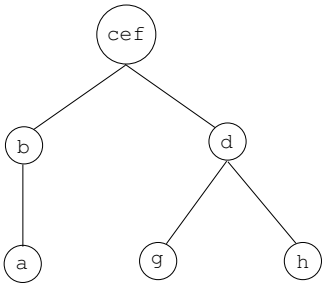

FIGURE 6.3: Starting from Figure 6.1 after making the bags disjoint.

Applying step (1) on the tree decomposition in Figure 6.1, we get the tree in Figure 6.3 where every vertex now appears only in one bag. It can be noticed that this stage involves only vertex deletions so that it is not possible to introduce new neighborhoods across any edge in the decomposition tree.

6.2.2 Placing every vertex in a leaf node

In a boolean decomposition every vertex should be in some leaf node as vertices of the graph are mapped to the leaves of the decomposition tree. But in a tree decomposition this is not a necessary condition. The next step is then to copy every non-leaf bag to a new leaf as a child of the original bag.

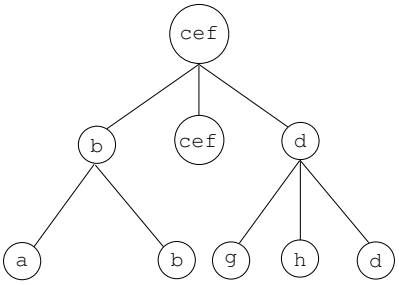

FIGURE 6.4: Starting from Figure 6.1 after making all bags leaves.

In Figure 6.4, every internal bag of the tree in Figure 6.3 has been copied to a leaf. Note that although new edges can be introduced to the decomposition tree following this operation, every new leaf containing a set of vertices A will have $|UN(A)| \leq 2^{|A|}$, where $|A|$ is bounded by the size of the maximum bag in the original tree decomposition.

6.2.3 Making a parent node contain every vertex in its children

For a binary boolean decomposition tree every vertex contained in a child node should be present in the parent node. To ensure this every vertex from each bag is copied into its parent node if the vertex is not already present in the parent. Note that this is done recursively so that the root node contains all vertices in $V(G)$.

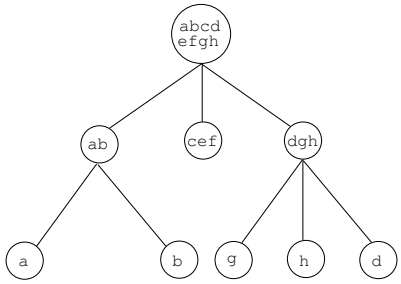

FIGURE 6.5: Starting from Figure 6.1 after copying every vertex to its parent.

This operation might cause new vertices to be introduced in the bags and for our example we now get the tree in Figure 6.5. But as discussed in Section 6.2, the size of $UN(A)$ of a vertex set A in a bag will not exceed $2^{|S|}$, where $|S|$ is the size of the maximum separator in the tree.

6.2.4 Making the decomposition binary

Following the steps (1)-(3) of Algorithm 9 the root of the tree contains $V(G)$ and every vertex is now exactly in one leaf. Step (4) makes the tree binary and ensures that every leaf of the decomposition tree corresponds to a unique vertex of $V(G)$. To do this we consider the following cases.

- An internal node having two children does not need to be altered and considers one child as its left child and the other one as its right child.
- A node having more than two children sets the first child as its left child and adds a new node as its right child. This node gets all the remaining children of the original node as its children and is assigned the union of their vertices. If the new node has more than two children, the process is repeated recursively.
- Similarly a leaf node assigned more than one vertex just splits the vertices randomly into two parts and adds these as its left and right child. The leaf nodes are decomposed recursively until we get a full binary decomposition or every vertex is mapped to exactly one leaf of the decomposition tree.

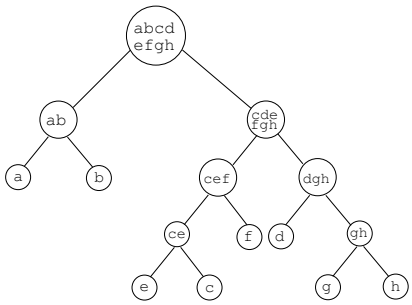

FIGURE 6.6: Starting from Figure 6.1 a binary boolean decomposition tree of G with $boolw(G) = \log(3)$=1.58.

Figure 6.6 shows the final full decomposition tree for the graph in Figure 6.1 obtained from Algorithm 9.

6.2.5 Time complexity

Step (1) in Algorithm 9 is accomplished using a depth first search in time $O(nk)$. The time complexity of step (2) is proportional to the number of nodes in the tree which is $O(n)$. In step (3) copying vertices to the parent node checks for containment, therefore the running time of this step is $O(nk)$. The time complexity of transforming the tree to a binary one is $O(n)$. Therefore Algorithm 9: TDTOBOOLD(X, T) has a total running time of $O(nk)$.

6.3 Experimental results

Computing a tree decomposition of low tree-width often depends heavily on the first chosen vertex and the overall criteria used for selecting vertices for the elimination ordering. It is therefore important to have a good criteria even at the expense of increasing the running time. There are a number of different criteria that can be used for this. In our work we have used the minimum degree heuristic for selecting the next vertex for π. In each step this selects the remaining vertex of smallest degree and adds to π. This heuristic was designed by Markowitz [80] in the context of sparse matrix computations and has been compared against other greedy approaches for generating elimination orderings in [12]. Once π has been computed we then use the algorithm from [12] to compute the tree decomposition.

In the following we describe experiments performed to test the procedure from Section 6.2. Different heuristics for computing tree-width upper bounds such as Greedy Degree, Greedy Fill-in, Minimum Triangulation have been tested and reported on graphs from TreewidthLIB [14]. For performance evaluation we ran our experiments on around 450 graphs from TreewidthLIB with known tree-width upper bounds. Moreover we tested on random graphs as well as other real world graphs for which tree-width upper bounds has not been listed in [14]. The graphs presented in Table 6.1 are selected from TreewidthLIB [14] in such a way that, they have good variation in the number of vertices, edge densities,

and tree-width. Machine configuration is the same as described in Chapter 2 and the algorithm is implemented in Java.

Table 6.1 lists the tree-width upper bounds obtained from the *Minimum Degree Fill-In* heuristic, *TW-MDH* and the boolean-width upper bound values computed from the boolean decomposition generated by Algorithm 9, *BW-A9*. We report the tree-width upper bounds listed in TreewidthLIB [14] for the corresponding graph instances. They are denoted by *TW-LIB*. The running time of Algorithm 9, T_{A9} in seconds is also reported. Both the time for generating the tree decomposition and for the transformation to get boolean decomposition is included in T_{A9}. The time in parentheses is the time to compute the initial tree decomposition.

TABLE 6.1: Comparison of boolean-width upper bound obtained from Algorithm 9 and tree-width upper bound listed in TreewidthLIB

Graph	V	E	*TW-MDH*	*BW-A9*	*TW-LIB*	T_{A9}
water	32	123	11	7.85	9	0.01 (0)
queen7_7	49	476	36	11.36	35	0.01 (0)
jean	80	254	9	4.81	9	0.01 (0)
david	87	406	13	9.84	13	0.01 (0)
celar06	100	350	11	4.58	11	0.03 (0.01)
graph01	100	358	26	22.05	24	0.02 (0)
queen10_10	100	1470	83	20.77	72	0.05 (0.01)
1lkk	103	1162	48	21.75	34	0.45 (0.05)
1gef	119	1492	52	23.19	43	0.36 (0.16)
anna	138	493	12	9.77	12	0.07 (0.01)
kroA150	150	432	16	16.61	12	0.04 (0.01)
mulsol.i.5	186	3973	30	5.17	31	0.15 (0.02)
celar07	200	817	16	8.82	16	0.11 (0.01)
zeroin.i.1	211	4100	42	6.73	50	0.22 (0.02)
munin4-wpp	271	724	8	7.6	8	0.07 (0)
pigs	441	806	10	10	9	0.14 (0)
link	724	1738	15	15.4	13	1.75 (0.02)
munin2	1003	1662	7	7.32	7	0.69 (0.01)
munin3	1044	1745	7	8	7	0.87 (0.01)

Note that the graphs in Table 6.1 are preprocessed according to the rules described in Chapter 5 before we generate the tree decompositions. From Chapter 5, we know that it is always possible to get a boolean decomposition of the original graph from the boolean decomposition of the reduced graph without increasing the width of the decomposition. Because of this preprocessing for some graphs we get *TW-MDH* lower than *TW-LIB*. Each reported value from our experiments is the minimum over five runs. This is because some randomness might emerge in the tree decomposition while breaking ties among vertices with the same degree. The generated boolean decompositions might also vary due to the use of a random bipartition of the vertices in the leaves. However in most cases *TW-MDH* and *BW-A9* only vary within a very small range.

From the results in Table 6.1 it can be observed that the upper bound obtained from TDTOBOOLD(X, T) complements the theoretical result as *BW-A9*$\leq$ *TW-MDH*+1. Oftentimes the boolean-width upper bound can be substantially lower than the tree-width of the corresponding tree decomposition. This is true for graphs such as queen10_10,

zeroin.i.1, and celar06. Though we have used one of the simplest heuristics to generate the tree decompositions, *TW-MDH* compares favorably with *TW-LIB*. The average distance of *TW-MDH* and *TW-LIB* is around 10% for the presented graphs. The ratio of *TW-LIB* to *BW-A9* is ranging from 0.72 to 7.8, with an average of 2.08. For most of the graphs in Table 6.1 except for 1gef, the tree decomposition generation phase takes only a small fraction of the total time.

In Chapter 3 we presented Algorithm 1 that generates full boolean decompositions using refinement via local search after an initial decomposition. As the greedy initialization is computationally expensive Algorithm 1 has only been tested on graphs having less than or equal to 271 vertices. In Table 6.2 we compare the boolean-width upper bound values obtained from Algorithm 1 and Algorithm 9. For this we have selected a set of representative graphs having less than or equal to 271 vertices from the set of graphs listed in TreewidthLIB. In Table 6.2 *BW-A1* and *BW-A9* represents boolean-width upper bounds obtained from Algorithm 1 and Algorithm 9 respectively. The running times in seconds for Algorithm 1 is given as T_{A1}. This includes the time for both greedy initialization and the local search.

Table 6.2: Comparison of boolean-width upper bounds obtained from Algorithm 1 and Algorithm 9

Graph	V	E	*BW-A9*	*BW-A1*	T_{A1}	T_{A9}
water	32	123	7.85	4.39	175.3	0.26 (0.04)
queen7_7	49	476	11.36	10.36	22.9	0.9 (0.1)
jean	80	254	4.81	3.91	61	0.39 (0.05)
david	87	406	9.84	5.32	284	0.55 (0.05)
celar06	100	350	4.58	3.81	132	0.47 (0.04)
1lkk	103	1162	21.75	11.89	968	9.8 (0.3)
1gef	119	1492	23.19	13.6	1917	58.1 (0.2)
anna	138	493	9.77	6.67	178	0.97 (0.07)
mulsol.i.5	186	3973	5.17	4.95	365	1.56 (0.16)
zeroin.i.1	211	4100	6.73	3.7	168	1.28 (0.18)
munin4-wpp	271	724	7.6	9.98	747	1.19 (0.09)

From the results listed in Table 6.2 it can be observed that in almost all cases the boolean-width upper bound values from Algorithm 1 are better than those obtained from Algorithm 9. The average distance from the best boolean-width upper bound is 45% for Algorithm 9 and 2% for Algorithm 1 for the graphs in Table 6.2. But in terms of computational time Algorithm 9 is always better and almost always orders of magnitude faster than Algorithm 1.

Analyzing the results reported in tables 6.1 and 6.2 we observed that Algorithm 9 can be applied to comparatively larger graphs than Algortihm 1. Though Algorithm 1 gives better boolean-width upper bounds, this comes at the cost of considerably higher running times. We note that this *TW-MDH*+1 can always serve as a boolean-width upper bound for a decomposition obtained using Algorithm 9, if the computation of the actual boolean-width is computationally too expensive.

6.4 Conclusion

In this chapter we have shown how to generate boolean decompositions from tree decompositions in practice. Though this approach is basically based on combinatorial insights, the experimental outcomes show that this approach can serve as a good alternative for generating boolean decompositions within a reasonable time. As stated we have chosen one of the simplest heuristics to generate our tree decompositions. It is possible to replace this by other more intricate algorithms to get a starting tree decomposition of smaller width. Moreover, local search over the resulted boolean decomposition can also play an important role to obtain a lower boolean-width upper bound.

Chapter 7

Exact and Random Boolean Decompositions

In order to apply FPT algorithms to solve hard problems on graphs, where the parameter is the width measure of the input, it is desirable that the value of the parameter be as small as possible. Moreover, in order to choose between algorithms parameterized by different width measures, the exact values of these parameters can play an important role. For example, for a graph of tree-width k and boolean-width k' there are dynamic programming algorithms to solve Minimum Dominating Set with running times $O^*(3^k)$ and $O^*(8^{k'})$, respectively. Discounting constant and polynomial factors it therefore seems that boolean-width is preferable if $k \geq 1.9k'$.

In chapters 3 and 6 we studied two different methods for generating boolean decompositions. However, these experiments did not tell us how the optimal boolean-width of the considered graphs compares to the optimal values of other width parameters. In this chapter we therefore first compare the optimal boolean-width to available optimal values for tree-width, clique-width, and rank-width for a set of small graphs. We also present experiments with random decompositions consisting of random bipartition of vertices. For random graphs there exists theoretical bounds on the values of different width parameters. It is known that random graphs with constant edge probability has linear tree-width, clique-width, and rank-width [81–83]. Experimental results from these approaches will expedite a way of verifying the existing theoretical bounds for these parameters. This can also help to show slackness and point to where such bounds can be strengthened.

7.1 Exact boolean decomposition

It is known that we can compute a decomposition of boolean-width $2^{2boolw(G)}$ using the algorithm for decompositions of optimal rank-width [5] in FPT time parameterized by $boolw(G)$ [38]. Given G, a binary decomposition tree having optimal boolean-width can also be computed in $O(2.52^n)$ time, where $n = |V(G)|$ [38, 49].

We have implemented an algorithm for computing the exact boolean-width from a corresponding decomposition of a graph. Our algorithm is based on brute-force search and

explores all possible decomposition trees to find the one with the minimum boolean-width. For this reason it can only solve graphs with less than 32 vertices within 30 minutes. For a particular boolean decomposition the corresponding width is the maximum boolean dimension over all cuts. In our algorithm we always keep track of the best decomposition found so far. While trying new decompositions we discard any decomposition as soon as it is known that the particular decomposition is not good enough to beat the so far best decomposition.

7.1.1 Experimental results

Algorithms for computing decompositions of minimum width has previously been developed for tree-width [14], clique-width [26], and rank-width [34]. Even though generating tree decompositions is reasonably fast, obtaining exact values are computationally more expensive for the other width parameters. Krause et al. has computed exact rank-width, rw [34]. This can only handle up to 32 vertices and works up to 28 vertices within a reasonable amount of time. The exact clique-width values, cw, are collected from [26], which used a SAT solver approach for computation. This approach is also computationally expensive and in [26] the exact clique-width has been reported for a set of named graphs with $|V(G)| \leq 24$ and random graph with $|V(G)| = 5, 10, 20$. Tree-width values, tw, are collected from TreewidthLIB.

In Table 7.1 we compare values for exact tree-width and rank-width with the exact boolean-width for a set of real world graphs from TreewidthLIB [14]. The computation of exact rank-width, rw, has been done using the rank-width software package provided by Krause et al. [34]. As exact clique-width is not available for these graphs we do not compare clique-width in Table 7.1.

TABLE 7.1: Exact rw, $boolw$, and tw for a set of small real world graphs

Graph	V	E	rw	$boolw$	tw
Diabetes-pp-002	8	17	2	2.32	4
Mainuk-pp	9	28	2	1.58	6
pcb3038-pp-001	11	22	3	3.00	5
fl3795-pp-004	11	23	3	3.00	4
1fjl-pp-003	11	47	2	1.58	8
Pathfinder-pp	12	43	3	2.58	6
oesoca+-hugin-pp	14	75	3	2.00	11
games120-pp-001	14	63	3	2.58	9
weeduk	15	49	2	1.58	7
Mildew-wpp	15	31	3	2.58	4
fungiuk	15	36	2	1.58	4
celar06-pp	16	101	1	1.00	11
Barley-pp-001	16	50	4	3.32	7
munin2-pp-005	16	36	3	3.00	5
celar02-pp	19	115	3	2.00	10
anna-pp	22	148	6	3.46	12
Water-pp	22	96	5	4.17	9
myciel4	23	71	6	4.95	10
Grid5x5	25	40	4	3.58	5
Queen5_5	25	320	5	5.29	18

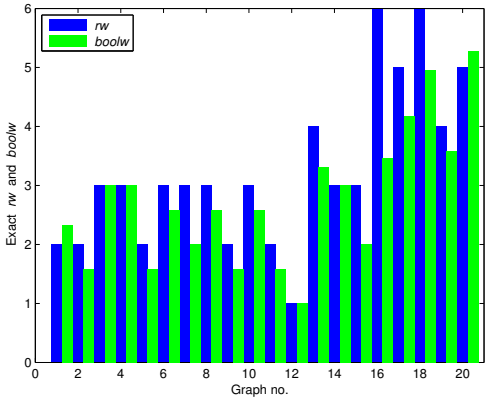

FIGURE 7.1: Exact *rw* and *boolw* for a set of small real world graphs

Table 7.1 shows that for most of the graphs boolean-width is the smallest among the three widths. Only for 2 out of the 20 graphs is boolean-width larger than rank-width and then only by a small factor (Diabetes-pp-002 and Queen5_5). These factors stays within the theoretical bound $\log rw(G) \leq boolw(G) \leq 0.25rw^2(G) + rw(G)$ [7] as stated earlier in Chapter 1 (Section 1.1.4). For 4 graphs out of the 20 the optimal values are equal for rank-width and boolean-width. Figure 7.1 shows this graphically. The average rank-width to boolean-width ratio is 1.2 with a minimum of 0.86 and maximum of 1.73. Similarly, the tree-width to boolean-width ratio ranges from 1.33 to 11 with an average of 3.3.

We next consider implications on FPT algorithms parameterized by the practical values obtained from the experiment reported in Table 7.1. For a graph of tree-width k, boolean-width k', and rank-width k'' FPT algorithms (using dynamic programming) solving the Minimum Dominating Set problem have running times $O^*(3^k)$ [8], $O^*(8^{k'})$ [7], and $O^*(2^{0.75k''^2+O(k'')})$ [84] respectively. Focusing only on the exponential factor in the running times, for 11 out of 20 graphs the boolean-width algorithm is preferable over the rank-width algorithm. For tree-width the corresponding result is 17 out of 20.

Similarly, for Maximum Independent Set the running times are $O^*(2^k)$ [8], $O^*(4^{k'})$ [7], and $O^*(2^{0.25k''^2+O(k'')})$ [84], respectively. Analyzing the exponential factor, for only 1 out of 20 graphs, the rank-width algorithm is preferable over the boolean-width algorithm, and for 5 out of the 20 graphs the tree-width algorithm is preferable over the boolean-width algorithm. Note that we have only considered the effect of the width parameters on the exponent, although the running times also include the number of vertices and edges as well as polynomial functions depending on the width measures.

In Table 7.2 we compare exact boolean-width with exact rank-width and clique-width for a set of named graphs. The values of width parameters on these graphs may be of interest in combinatorics and graph theory. The exact clique-width for these graphs has been collected from [26]. Exact rank-widths are obtained using [34]. As TreewidthLIB does not list bounds for these graphs we do not compare tree-width in Table 7.2.

TABLE 7.2: Exact *cw*, *rw*, and *boolw* for a set of named graphs

Graph	V	E	cw	rw	$boolw$
Petersen	10	15	5	3	3
Chavatal	12	24	5	4	3.17
Franklin	12	18	4	2	2
Frucht	12	18	5	3	3
Poussin	15	39	7	4	3.7
Clebesch	16	40	8	4	4
Hoffman	16	32	6	3	3.58
Shrikhande	16	48	9	4	4.52
Sousselier	16	27	6	4	3.7
Errera	17	45	8	4	3.7
Pappus	18	27	8	4	4
Robertson	19	38	9	5	5.64
Desargues	20	30	8	4	4.64
Flower snark	20	30	7	5	4.81
Folkman	20	40	5	3	3.32
Brinkmann	21	42	10	6	5.81
Kittell	23	63	8	5	4

From Table 7.2 it can be observed that for 12 out of the 17 graphs boolean-width is the smallest among the three widths. For these 17 graphs the average rank-width to boolean-width ratio is 1.01 with a minimum ratio of 0.83 and maximum ratio of 1.26. Similarly, the clique-width to boolean-width ratio ranges from 1.45 to 2.16 with an average of 1.8. For 5 of the graphs boolean-width is marginally larger than rank-width. However, this value stays within the theoretical proven bound $\log rw(G) \leq boolw(G) \leq 0.25rw^2(G) + rw(G)$ [7].

7.2 Random boolean decompositions

There exists theoretical evidence that random bipartitions are useful for boolean decompositions, at least for random graphs [31]. The analysis shows that any decomposition of a random graph is expected to be a decomposition of relatively low boolean-width. Furthermore, for random graphs the expected boolean-width is significantly lower than tree-width, clique-width, and rank-width. In the following we consider random graphs generated by the *Erdös-Rényi* model. For a constant $0 < p < 1$ the *Erdös-Rényi* model generates a graph G_p with $|V(G_p)| = n$ vertices where for every pair of vertices an edge is added independently with probability p. It is known from [31] that $boolw(G_p) \in O(\frac{\ln^2 n}{p})$. If p is a constant, then almost surely: $tw(G_p), cw(G_p), rw(G_p) \in \theta(n)$, whereas $boolw(G_p) \in \theta(\log^2 n)$ [31, 81–83]. To demonstrate this in practice, we have experimented with random boolean decompositions.

Random boolean decompositions are easy to generate. We start with a root node containing all the vertices and then recursively use a random bipartition of the remaining vertices at every node of the boolean decomposition tree until each vertex has a one-to-one bijection to a leaf of the decomposition tree.

7.2.1 Experiments with random graphs

In Figure 7.2 we present the boolean-width of random decompositions of random graphs with constant edge probability p=0.5. For each graph we generate five random decompositions and report the average boolean-width. The boolean width is plotted against the number of vertices, for $n = 5$ to 100. As the θ notation hides any constant factor we have drawn the trend line $y = m \log^2 x$ with $m = \frac{1}{3}$. It can be seen from the figure that this trend line approximates the observed results fairly well. The time taken for each of these experiments increases with n and for $n = 100$ it takes about 7 seconds in total to generate and evaluate one random decomposition.

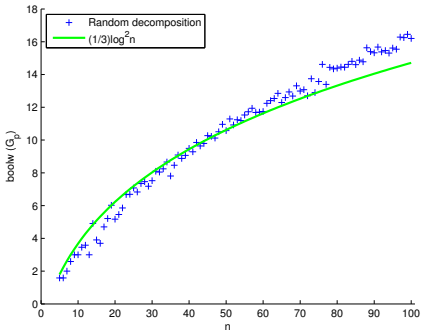

FIGURE 7.2: Boolean-width of random decompositions on random graphs with edge probability 0.5.

The second set of experiments consists of three sets of *Erdös-Rényi* graphs having $|V(G_p)| = 30$, 40, and 50 respectively. For each set of graphs we vary the edge probability p from 0 to 1 with an increase of 0.01. For each random graph generated within these configurations the plotted boolean-width is an average of the boolean-width of five random decompositions.

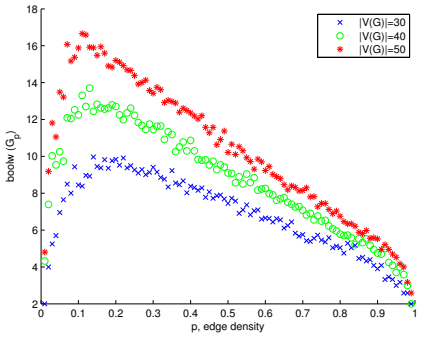

FIGURE 7.3: Boolean-width of random decompositions on random graphs.

Figure 7.3 illustrates how the boolean-width of the random graphs changes as p varies. The time taken for these experiments increases with n and for $n = 50$ and $p = 0.1$ it takes about 1 second in total to generate and evaluate one random decomposition.

Theoretical result $boolw(G_p) \in O(\frac{\ln^2 n}{p})$ implies that for fixed n, $boolw(G_P)$ decreases as p increases. On the other hand Figure 7.3 indicates that this relation holds only after a certain value of p. Moreover, this threshold value seems to depend on n. We note that there exists similar theoretical results for other width parameters, where separate bounds hold for very sparse random graphs. For example, let c be a constant with $p = c/n$ then the following holds asymptotically almost surely [83]:

(1) If $c > 1$, then $rw(G_p)$, $cw(G_p)$, and $tw(G_p)$ are at least $c'n$ for some constant c' depending only on c.
(2) If $c < 1$, then $rw(G_p)$ and $tw(G_p)$ are at most 2 and $cw(G_p)$ is at most 5.

Together with the results in Figure 7.3 this could be an indication that it might also be possible to split the edge density scale at the very sparse end depending on the number of vertices to obtain tighter bounds for sparse random graphs. It can also be noted that the maximum width obtained from the random decompositions for any n and p ranges from 0 to $n/3$.

Next, in Figure 7.4 we have compared the optimal boolean-width with the boolean-width obtained from random decompositions for a set of random graphs. For this set of experiments we have used $n = 20$ while varying p from 0 to 1 with an increase of 0.05. The results show that the exact boolean-width of random graphs can be substantially lower than those from random decompositions. However, we note that this is only based on small graphs on 20 vertices. The relative difference in the computed numbers between exact and random decompositions ranges from 21% to 206% with an average of 65%.

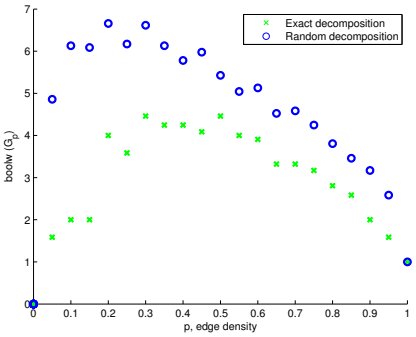

FIGURE 7.4: Boolean-width of random graph with $n = 20$ vertices using exact and random decompositions.

7.2.2 Experiments with real world graphs

In the next set of experiments we test how random decompositions perform on real world graphs. To do this we compare the experimental boolean-width upper bounds obtained using random decompositions with the boolean-width upper bounds from Algorithm 1

and Algorithm 9. In Table 7.3 we have used the same set of graphs as was used in Table 6.2. The boolean-width upper bounds for Algorithm 1 and Algorithm 9 is denoted by *BW-A1* and *BW-A9*, respectively. For each graph the average boolean-width upper bound from five random decompositions is denoted by *BW-RD* in Table 7.3.

TABLE 7.3: Comparison of boolean-width upper bounds obtained from random decompositions, Algorithm 1, and Algorithm 9

Graph	V	E	*BW-A1*	*BW-A9*	*BW-RD*
water	32	123	4.39	7.85	8.22
queen7_7	49	476	10.36	11.36	11.93
jean	80	254	3.91	4.81	13.23
david	87	406	5.32	9.84	16.57
celar06	100	350	3.81	4.58	23.72
1lkk	103	1162	11.89	21.75	20.94
1gef	119	1492	13.6	23.19	23.31
anna	138	493	6.67	9.77	24.97
mulsol.i.5	186	3973	4.95	5.17	19.63
zeroin.i.1	211	4100	3.7	6.73	10.4
munin4-wpp	271	724	9.98	7.6	59.97

From the above experiments it can be observed that random decompositions are always outperformed by the two other approaches. In some cases the values from random decompositions are substantially larger than those from Algorithm 1 and Algorithm 9. For example on munin4-wpp *BW-RD* is almost 8 times larger than *BW-A1*. The average distance from the best boolean-width upper bound for each graph is 242% for random decompositions, 45% for Algorithm 9, and 2% for Algorithm 1. We can thus conclude that random bipartitions are not as useful for real world graphs as for random graphs.

7.3 Conclusion

Since graph width measures are important structural graph parameters used for capturing the algorithmic tractability of computationally hard problems, it is interesting to see how the exact values of these parameters behave for a set of graphs of practical relevance. Studying the probabilistic behavior of boolean-width of random graphs is also an interesting combinatorial problem. Except for the theoretically established bounds there has been no previous work reflecting the strength of random boolean decompositions. The experiments in this chapter show that the exact boolean-width is in most cases smaller than the other studied exact width measures. Likewise, exact width values on certain graph classes can give us better combinatorial insight.

The performance of the random decompositions on random graphs seems to indicate that tight bounds are possible for sparse random graphs. Also, these decompositions can serve as a starting point prior to computing improvements via local search. In any case, random decompositions can be used by the dynamic programming algorithms solving hard problems on random graphs.

Chapter 8

Generating Caterpillar Decompositions

In chapters 3 and 6 we presented algorithms for generating full binary boolean decompositions. Even though these in many cases generated decompositions of low boolean-width, this also came at the expense of high computational time. For a specific problem instance one has to look at the combined time to first compute the decomposition and then to solve the problem. As generating a good decomposition is a hard problem in itself, sometimes it is convenient to work with simpler decompositions. For this reason we now investigate efficient heuristics for computing caterpillar decompositions. As these are linear orderings (as opposed to tree orderings) we expect that they will be easier and faster to generate. Even though these orderings might give a higher boolean-width upper bound (which we will refer to as linear boolean-width) than those from more general algorithms, it is still possible that the combined solution time is lower. These decompositions also serve as an upper bound of what can be achieved by a full decomposition. Moreover, given a caterpillar decomposition tree of linear boolean width k one can solve the ISP in time $O^*(2^k)$, Dominating Set, Independent Dominating Set, and Total Dominating Set in time $O^*(2^{2k})$ [49]. The combination of linear decompositions and dynamic programming gives moderately-exponential exact algorithms on general graphs also for solving weighted and counting versions of all these problems [49].

In the following we propose various approaches for generating linear decompositions and evaluate their performance experimentally. We will in particular be looking at approaches where the vertices are chosen based on some greedy condition.

8.1 Algorithms for generating caterpillar decompositions

As stated, all our heuristics use a greedy approach and select vertices in the order they will appear in the final caterpillar decomposition. Thus, except for some trivial cases, each heuristic uses a different criterion for selecting the next vertex. This vertex is then appended to the end of the current ordering before the procedure is repeated until all vertices have been selected and the ordering is returned. Algorithm 10 outlines the general structure of our heuristics.

Once the ordering $\pi = (v_1, v_2, ..., v_{|V(G)|})$ has been computed it is straightforward to compute the caterpillar decomposition (T, δ). We do this by generating a tree T with $|V(G)|$ leaves and where every internal node has a right child that is a leaf. Then δ maps the ith leaf in the caterpillar numbered from left to right to the ith vertex in the ordering. The linear boolean-width of the decomposition is then the maximum number of maximal independent sets among all graphs of $BG = (\{v_1..v_i\}, \{v_{i+1}...v_{|V(G)|}\}, E_{BG})$ for $i = 1$ to $|V(G)| - 1$.

Algorithm 10 : GenerateVertexOrdering(G)

Input : Graph $G = (V(G), E(G))$
Output : Vertex Ordering $Sequence = (v_1, v_2, ..., v_{|V(G)|})$
$Right \leftarrow V(G)$.
$v \leftarrow$Vertex returned by the strategy for selecting the first vertex
$Sequence \leftarrow \{v\}$
$Left \leftarrow \{v\}$
$Right \leftarrow Right \setminus \{v\}$
while $Right \neq \emptyset$ **do**
  $chosen \leftarrow \emptyset$
  for $\forall v \in Right$ **do**
    if v belongs to any of the trivial cases **then**
      $chosen \leftarrow v$
      break
    end if
  end for
  if $chosen == \emptyset$ **then**
    $chosen \leftarrow$ Best v in $Right$ as described in algorithms 11, 12, and 13
  end if
  $Sequence \leftarrow Sequence \cdot \{chosen\}$
  $Left \leftarrow Left \cup \{chosen\}$
  $Right \leftarrow Right \setminus \{chosen\}$
end while
return $Sequence$

In the following we outline three different heuristics for obtaining a linear decomposition. But first we look at some issues that are common to all three heuristics.

8.1.1 Selecting the first vertex

Our heuristics progress by selecting the next vertex based on its relationship to the already chosen ones. This leaves open the question of how to choose the first vertex. One possibility would be to start from a random vertex, but as the following example shows all vertices are not equally suitable to be used as the starting vertex. Consider the graph in Figure 8.1.

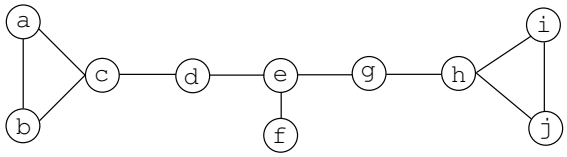

FIGURE 8.1: A graph G.

Let $\pi_1 = (a, b, c, d, e, f, g, h, i, j)$ and $\pi_2 = (f, g, e, d, h, a, b, c, i, j)$ be two different caterpillar decompositions of the graph, where δ and δ' map the leaves of the decomposition trees to the vertices of G respectively. Note that π_1 starts with the vertex a while π_2 starts with the minimum degree vertex f. The linear boolean-width of π_1 and π_2 is 1 and 2 respectively. In fact, when we have vertex f in one side of a cut along with two other vertices, then the number of unions of neighborhoods across the cut will be more than 2, hence increasing the linear boolean-width of the caterpillar decomposition.

It is not hard to see that in π_2 once f and g have been selected the remaining graph is not connected. Whereas in π_1 we are able to keep both the selected graph and the remaining graph connected. Thus we suggest to choose a vertex to be the first vertex in such a way that its neighborhood does not get scattered over different connected components. Such a vertex can be found by performing a breadth first search (BFS) starting from any vertex and then picking a vertex v from the last BFS level. In fact, we found that repeating this process once more from v gave an even better starting point and this is what we do in all our heuristics.

8.1.2 Trivial cases

Once the initial vertex v has been selected we place this as the first vertex in *Sequence* and also in a set labeled $Left$, whereas all other vertices are placed in a set labeled *Right*. The heuristics now progress by selecting vertices from *Right* and adding these to $Left$ while also appending them at the end of the linear ordering.

There are some cases common to all heuristics where we override the selection criteria, thus we cover these first.

1. If the neighbors of a vertex $v \in Right$ are all contained in $Left$, then select v as the next vertex.

2. If there are two vertices $v \in Right$ and $u \in Left$ such that v and u have exactly the same set of neighbors in $Right \setminus v$ then select v. That is, select v if $N(v) \cap Right = (N(u) \setminus v) \cap Right$. In Figure 8.2 x is such a vertex.

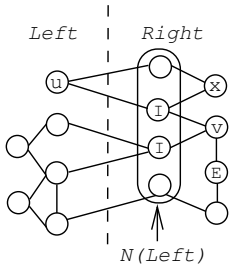

FIGURE 8.2: One stage of the selection process

Note that for both of these cases no new neighborhoods will be created in $Right$. Thus using either of them will not increase the boolean dimension of the ordering being computed. We now present the three different selection criteria that we have based our heuristics on.

8.1.3 Least Uncommon Neighbors

In the Least Uncommon Neighbors approach we pick the next vertex from $Right$ as the one having the fewest neighbors in $Right$ that are not adjacent to a vertex in $Left$. That is, we pick a vertex v that minimizes the number of vertices $u \in (N(v) \cap Right)$ where $N(u) \cap Left = \emptyset$.

The intuition behind this strategy is that each step will add as few new vertices as possible in $Right$ that are adjacent to vertices in $Left$. The selection criteria is outlined in Algorithm LEASTUNCOMMONNEIGHBOR.

Algorithm 11 : LEASTUNCOMMONNEIGHBOR($Left, Right$)

Input : A partitioning of $V(G)$ into $Left$ and $Right$
Output : A vertex from $Right$ having the least number of uncommon neighbors with $N(Left)$
$MinSymmDiff \leftarrow |Right|$
for $\forall v \in Right$ **do**
  $SymmDiff_v \leftarrow |(N(v) \cap Right) \setminus N(Left)|$
  if $SymmDiff_v < MinSymmDiff$ **then**
    $chosen \leftarrow v$
  end if
end for
return *chosen*

8.1.4 Relative Neighborhood

While the Least Uncommon Neighbors heuristic selects a vertex based on trying to minimize the absolute number of vertices in $Right$ that becomes adjacent to a vertex in $Left$, our next strategy, Relative Neighborhood, selects a vertex such that its ratio of neighbors not in $N(Left) \cap Right$ relative to those which are already in $N(Left) \cap Right$ is kept low.

More formally, for every vertex $v \in N(Left)$ we partition each vertex $w \in N(v) \cap Right$ into one of two sets: $Internal(v)$ if $w \in N(Left)$ and $External(v)$ if $w \notin N(Left)$. In Figure 8.2 $Internal(v)$ and $External(v)$ are labeled as I and E respectively. Our heuristic then selects the next vertex $v \in Right$ such that $|External(v)|/|Internal(v)|$ is minimized. This selection criteria is outlined in Algorithm 12.

Algorithm 12 : RelativeNeighborhood($Left, Right$)

Input : A partitioning of $V(G)$ into $Left$ and $Right$
Output : A vertex from $Right$ having the least $Ratio$
$Minratio \leftarrow |Right|$
for $\forall v \in N(Left)$ **do**
  $Ratio_v \leftarrow \frac{External(v)}{Internal(v)}$
  if $Ratio_v < Minratio$ **then**
    $chosen \leftarrow v$
  end if
end for
return *chosen*

With this criteria we want v to be adjacent to a large number of vertices that were already in $N(Left)$. Then a vertex $u \in Left$ where $N(u) \cap N(Left) \subseteq N(v) \cap N(Left)$ will not create any new neighborhood together with v.

8.1.5 Greedy

Our final heuristic, Greedy, is outlined in Algorithm 13. This strategy tries all vertices in $Right$ and picks the one that gives the smallest boolean dimension when moved to $Left$. The boolean dimension of the potential cuts are evaluated using the routine CCMis($Left \cup v$) for the bipartite graph $BG = (Left \cup \{v\}, Right \setminus \{v\}, E(BG))$ described in Chapter 4. We note that although this strategy picks the locally best vertex this might not be the globally best choice. Also, computing the boolean dimension multiple times increases the running time.

Algorithm 13 : LeastCutValue($Left, Right$)

Input : A partitioning of $V(G)$ into $Left$ and $Right$
Output : A vertex of $Right$ which gives the least boolean dimension compared when moved to $Left$
$MinBooldim \leftarrow 2^{min(|Left|+1,|Right|-1)}$
for $\forall v \in Right$ **do**
  $Booldim_v \leftarrow$ CCMis($BG = (Left \cup \{v\}, Right \setminus \{v\}, E(BG))$)
  if $Booldim_v < MinBooldim$ **then**
    $chosen \leftarrow v$
    $MinBooldim \leftarrow Booldim_v$
  end if
end for
return *chosen*

8.1.6 Time complexity and implementation

To compute a linear ordering of a given graph G Algorithm 10 first tries to pick a vertex according to one of the trivial rules. Rule 1 checks every vertex in *Right* hence it needs time proportional to the size of the *Right* i.e. $O(n)$ if $|V(G)| = n$. Rule 2 determines if two vertices $u \in Left$ and $v \in Right$ are twins by testing every pair of vertices thus yielding a time complexity of $O(n^2)$. If no trivial rule succeeds, then the selection criteria of one of the algorithms 11, 12, and 13 is used. This is done $O(n)$ times. Algorithm 11 selects a vertex in running time proportional to the size of *Right* i.e. in time $O(n)$. In addition to selecting a vertex in time $O(n)$, Algorithm 12 has an extra operation of updating external and internal neighbors of all vertices in $N(Left)$. The time complexity for this is $O(n)$. Therefore Algorithm 12 has a running time of $O(n^2)$. The greedy heuristic in Algorithm 13 requires $O(n)$ computations of boolean dimension to select the next vertex. Therefore Algorithm 10 using Algorithm 13 will require $O(n^2)$ computations of boolean dimension, where computing the boolean dimension is exponential in n. We have implemented vertex subsets as bitsets in Java so that set operations such as union, intersections, and checking for containment of an element in the set take constant time.

8.2 Experimental results

In the following we describe experiments performed to evaluate the heuristics presented for computing a caterpillar decomposition. The programs are written in Java and compiled with `javac` version 1.6.0_30. These heuristics are mainly run on the benchmarks graphs from TreewidthLIB [14] and the 2nd DIMACS challenge [39] that we have already used to assess previous boolean decomposition algorithms described in chapters 3 and 6. This includes graphs originating from various computational problems and real life applications as described in Chapter 2.

There are several measures that can be used to compare the quality of the different caterpillar decompositions. Some examples include:

1. The time taken by each heuristic to compute the vertex ordering and also the time taken to compute the linear boolean width.
2. The value of the computed linear boolean-width of the corresponding caterpillar decomposition.
3. The time taken by the subsequent dynamic programming algorithm that uses the computed decomposition.
4. The total time required to compute the vertex ordering and the subsequent dynamic programming.

In the following experiments we will only consider items 1 and 2. In Chapter 9 we will look at an application where we actually use the computed decompositions to solve other problems. We have partitioned our experiments into two parts, the first is for graphs having less than 300 vertices, while the second part is for graphs having more than 300 vertices. We do not include experiments for the Greedy heuristic for the larger graphs.

This is because this heuristic requires $O(n^2)$ computations of boolean dimension for each graph which took at least several minutes for graphs with more than 300 vertices.

The results for the small graphs are presented in tables 8.1 and 8.2. We ran experiments on all graphs from TreewidthLIB having vertices less than 300. However the graphs listed in tables 8.1 and 8.2 are selected in such a way that we have boolean-width upper bounds from Algorithm 1 and Algorithm 9 as well as tree-width upper bounds for all of these graphs. This will not only help us measure the relative performance of our proposed heuristics, but also to compare boolean-width with tree-width.

The results from LeastUncommonNeighbor is labeled as O_1, the results from RelativeNeighborhood as O_2, and LeastCutValue as O_3. In Table 8.1 for O_1 and O_2 we report the time required to compute the ordering, T_1, the time to compute the linear boolean-width upper bound of the ordering, T_2, and the sum of these, $Total$. For O_3 we report the total time, $Total$, as this includes both generating and computing linear boolean-width upper bounds, lbw.

Table 8.2 lists statistics for the linear boolean-width upper bounds obtained from all three ordering heuristics. Moreover, we report boolean-width upper bounds obtained from Algorithm 1, *BW-A1*, and Algorithm 9, *BW-A9*. Tree-width upper bounds listed in TreewidthLIB are also shown for the same set of graphs in Table 8.2 as *TW-LIB*. All times are in seconds measured as the average of five runs. Algorithm 8, CCMis, described in Chapter 4 has been used for computing the boolean-dimension.

Table 8.1: Running time of the heuristics on small graphs

			O_1			O_2			O_3
Graph	V	E	T_1	T_2	$Total$	T_1	T_2	$Total$	$Total$
alarm	37	65	0.04	0.03	0.07	0.02	0.01	0.02	0.25
barley	48	126	0.03	0.05	0.08	0.03	0.04	0.07	0.43
pigs-pp	48	137	0.04	2.3	2.35	0.04	0.14	0.18	0.45
BN_100	58	273	0.06	10.32	10.38	0.03	7.93	7.97	7.1
eil76	76	215	0.06	0.52	0.58	0.07	1.07	1.14	1.65
david	87	406	0.06	1.41	1.47	0.03	0.1	0.14	2.22
1jhg	101	841	0.05	0.76	0.81	0.09	1.53	1.62	6.94
1aac	104	1316	0.1	22.99	23.09	0.09	113.95	114.04	118.41
celar04-pp	114	524	0.1	12.85	12.94	0.11	3.91	4.02	4.77
1a62	122	1516	0.08	7.2	7.28	0.11	12.81	12.93	108.08
1bkb-pp	127	1473	0.08	2.26	2.34	0.11	14.72	14.83	46.13
miles250	128	387	0.08	0.14	0.22	0.08	0.16	0.24	4.07
miles1500	128	5198	0.08	0.11	0.2	0.15	0.11	0.26	39.07
1dd3	128	1356	0.07	2.66	2.73	0.11	5.63	5.73	30.24
celar10-pp	133	646	0.08	1.41	1.49	0.11	1.86	1.97	9.03
anna	138	493	0.14	16.24	16.38	0.05	1.02	1.07	6.41
pr152	152	428	0.08	9.68	9.76	0.13	1.43	1.56	16.33
munin2-pp	167	455	0.08	1.2	1.28	0.19	0.1	0.28	13.46
mulsol.i.5	186	3973	0.08	1.18	1.26	0.09	0.33	0.42	34.36
zeroin.i.2	211	3541	0.11	0.35	0.46	0.06	0.06	0.11	15.92
boblo	221	328	0.08	2.18	2.26	0.09	0.01	0.1	3.14
fpsol2.i-pp	233	10783	0.13	3.29	3.42	0.17	2.02	2.19	222.26
munin4-wpp	271	724	0.15	0.8	0.95	0.23	0.88	1.11	105.52

TABLE 8.2: Linear boolean-width upper bounds given by the heuristics on small graphs

Graph	lbw_1	lbw_2	lbw_3	*BW-A1*	*BW-A9*	*TW-LIB*
alarm	7.71	4.95	3	2.58	4	4
barley	7.9	6.54	6	4	6.95	7
pigs-pp	8.33	6.7	7.07	5.7	8.55	9
BN_100	9.72	8.05	12.07	10.06	15.92	21
eil76	8.95	6.52	9.22	7.17	13.9	14
david	9.15	7.21	7.03	5.32	9.84	13
1jhg	9.52	8.28	8.49	8.87	14.43	25
1aac	11.87	13.07	12.4	12.29	20.76	41
celar04-pp	9.34	8.08	9.6	7.29	11.79	16
1a62	11.1	11.24	11.85	13.62	24.78	37
1bkb-pp	10.17	9.98	9.98	11.55	19.72	30
miles250	8.97	7.66	4.7	4.95	14.33	9
miles1500	10.28	9.35	5.58	4.86	7.87	77
1dd3	10.32	9.05	9.82	11.68	24.14	31
celar10-pp	9.46	6.92	10.95	9.08	9.24	16
anna	10.39	7.19	8.2	6.67	9.77	12
pr152	9.55	7.77	10.36	6.7	13.81	13
munin2-pp	9.66	7.95	10.54	5.49	6.12	7
mulsol.i.5	7.53	8.25	4.81	4.95	5.17	31
zeroin.i.2	9.94	8.44	4.64	5.39	5.67	32
boblo	9.19	7.61	3.81	3.54	3.32	3
fpsol2.i-pp	8.1	6.73	5.78	4.91	11.68	66
munin4-wpp	10.52	9.61	9.27	9.98	7.6	8

As can be seen from the results in Table 8.1, for both O_1 and O_2, in almost all cases it takes considerably more time to compute the linear boolean-width than to compute the ordering. Comparing the total time of O_1 and O_2 it follows that it varies which algorithm is the fastest, but on average O_2 is the faster one. For the graph boblo O_1 is more than a factor of 20 slower than O_2. O_3 is as expected the slowest algorithm being up to two orders of magnitude slower than the best of O_1 and O_2.

Comparing the boolean-width upper bounds in Table 8.2 the average distance from the best upper bound for each graph is 60.68% for O_1, 34.81% for O_2, 24.55% for O_3, 8.98% for Algorithm 1, and 74.13% for Algorithm 9. The average ratio of *TW-LIB* to the minimum boolean-width upper bounds for these graphs is 3.66, with a minimum ratio of 0.9 and maximum of 15.8. Though O_3 is the slowest among the three heuristics, in terms of linear boolean-width upper bounds it performs better than O_1 and O_2. Specifically for the 1∗ graphs related to protein structures lbw_{O_3} is almost always lower than *BW-A1*, which was obtained using local search over a greedy initialization.

Tables 8.3 and 8.4 give similar statistics as tables 8.1 and 8.2 but for a set of graphs each containing more than 300 vertices. Boolean-width upper bounds from Algorithm 1, (*BW-A1*) and Algorithm 9, (*BW-A9*) are not presented for these graphs as Algorithm 1 is computationally very expensive and Algorithm 9 is already outperformed by other heuristics on the smaller graphs.

TABLE 8.3: Running time of the heuristics on large graphs

			O_1			O_2		
Graph	V	E	T_1	T_2	$Total$	T_1	T_2	$Total$
link-pp	308	1158	0.12	1.72	1.84	0.28	77.3	77.58
diabetes-wpp	332	662	0.09	1.77	1.86	0.34	1.3	1.64
link-wpp	339	1194	0.12	1.99	2.11	0.37	28.39	28.76
celar10	340	1130	0.12	1.94	2.05	0.32	1.2	1.52
celar11	340	975	0.09	1.88	1.97	0.29	1.16	1.45
rd400	400	1183	0.14	6.68	6.81	0.6	5.05	5.65
diabetes	413	819	0.14	3.32	3.46	0.54	2.19	2.73
fpsol2.i.3	425	8688	0.13	3.3	3.44	0.26	1.85	2.1
pigs	441	806	0.15	2.98	3.13	0.48	2.29	2.78
celar08	458	1655	0.17	3.88	4.05	0.58	2.96	3.54
d493	493	1467	0.15	10.7	10.85	0.98	67.12	68.11
homer	561	1628	0.17	200.96	201.13	0.35	176.21	176.56
rat575	575	1699	0.17	11.33	11.49	1.8	21.54	23.34
u724	724	2117	0.32	22	22.31	3.1	67.52	70.62
inithx.i.1	864	18707	0.21	11.62	11.83	0.54	8.7	9.24
munin2	1003	1662	0.37	21.53	21.9	3.6	20.66	24.27
vm1084	1084	2869	0.49	64.72	65.21	13.58	198.71	212.29
BN_24	1819	4541	0.32	151.81	152.13	32.71	151.68	184.38
BN_25	1819	4541	0.34	151.87	152.21	32.42	151.29	183.71
BN_23	2425	6055	0.83	453.13	453.95	106.53	450.16	556.69
BN_26	3025	14075	0.81	386.3	387.11	84.85	380.35	465.21

TABLE 8.4: Linear boolean-width upper bounds given by the heuristics on large graphs

Graph	lbw_1	lbw_2	*TW-LIB*
link-pp	34.81	28.68	13
diabetes-wpp	8.58	18.58	4
link-wpp	35	29.03	13
celar10	20.81	15	16
celar11	19.54	14.7	16
rd400	34.73	21.32	
diabetes	29.32	19.32	4
fpsol2.i.3	15.87	8.92	31
pigs	24.04	18	9
celar08	24.95	15	16
d493	20.29	48.1	
homer	36.22	28.49	31
rat575	16.48	37.23	
u724	18.72	50.09	26
inithx.i.1	11.98	7.22	56
munin2	31.25	12.13	7
vm1084	15.21	48.95	23
BN_24	4.91	2.32	4
BN_25	4.64	2.32	4
BN_23	8.48	3.17	4
BN_26	6.98	2.32	4

As can be seen from the results in Table 8.3, both for O_1 and O_2, in almost all cases it takes considerably more time to compute the linear boolean-width than to compute the ordering. Comparing the total time of O_1 and O_2 indicates that on average O_1 is the faster one. For the graph link-pp O_2 is more than factor of 40 slower than O_1. For large graphs the average distance from the best total running time is 12.5% for O_1 and 311.8% for O_2.

Table 8.4 reports the linear boolean-width upper bounds corresponding to O_1 and O_2. The average distances from the best upper bound is 58.8% for O_1 and 36.6% for O_2. The average ratio of *TW-LIB* to the minimum of lbw_1 and lbw_2 for these graphs is 1.53. It follows that for large graphs RelativeNeighborhood generates orderings with lower width but at the cost of running time.

In [26] a new method is presented for computing the exact clique-width of graphs. This is based on encoding the problems as a satisfiability instance (SAT) and then solving these using a SAT solver. This approach is used on a set of random graphs as well as some named graphs. We have already compared optimal clique-width and boolean-width for graphs in Table 7.2. In Table 8.5 we compare the computed exact clique-width *cw* from [26] with the linear boolean-width upper bound given by Algorithm 13 LeastCutValue .

Table 8.5: Comparing linear boolean-width upper bounds of graphs with exact clique-width

Graph	V	E	cw	lbw_3	T_3
Petersen	10	15	5	5.32	0.01
Chavatal	12	24	5	4	0.1
Franklin	12	18	4	3.32	0.019
Frucht	12	18	5	3	0.019
Poussin	15	39	7	3	0.002
Clebesch	16	40	8	4	0.1
Hoffman	16	32	6	4.91	0.003
Shrikhande	16	48	9	5.17	0.07
Errera	17	45	8	4.58	0.046
Pappus	18	27	8	4.17	0.1
Robertson	19	38	9	4.09	0.12
Desargues	20	30	8	5.81	0.57
Flower snark	20	30	7	5.39	0.18
Folkman	20	40	5	4	0.003
Brinkmann	21	42	10	6.77	1.6
Kittell	23	63	8	4.75	0.1

It can be observed that for all of these graphs except one (Petersen), the linear boolean-width upper bound is smaller than the exact clique-width. The average ratio of *cw* to lbw_3 for these graphs is 1.5, with a minimum ratio of 0.93 and maximum of 2.3. We also note that the time needed to compute the linear boolean-width, even using the greedy heuristic, is negligible, whereas the SAT approach for exact clique-width is computationally expensive. As reported in [26] for a random graph with 20 vertices and 95 edges it took about 358 seconds only to generate the representative encoding for the SAT solver on a 4-core Intel Xeon CPU.

8.3 Other orderings

In addition to the three heuristics already mentioned, we have also tried other orderings that did not prove successful. For completeness we list these here. The following two criteria for picking the next vertex in Algorithm 10 were tested:

- Choose a minimum degree vertex.
- Choose a maximum degree vertex.

We have also unsuccessfully tried orderings based on other algorithms. These include:

- Nested dissection [61]: This is an ordering $\pi = \{v_1, v_2, \ldots, v_{|V(G)|}\}$ on the vertices of $G = (V(G), E(G))$, that strives to number vertices that make up a (preferably small) separator $S \subseteq V(G)$ of G first, with the added constraint that the remaining components of $G \setminus S$ should be of roughly equal size. This is then repeated recursively for each connected component. One can also view a nested dissection ordering as an elimination tree [62]. We have used the graph partitioning tool METIS [63] to generate this ordering. Similar orderings were also used in Algorithm 8, CCMis.
- Minimum degree fill-in ordering: This ordering can be obtained by repeatedly selecting the minimum degree vertex from the remaining subgraph and making its neighborhood in the subgraph into a clique. We used this type of ordering in Chapter 6 to generate tree decompositions.
- Degeneracy ordering: We have also tried using the degeneracy ordering of a graph G as defined in Chapter 4, Section 4.1. Note that the degeneracy of a graph is a measure of how sparse it is and has also been used by Eppstein et al. for counting maximal cliques [59].

8.4 Conclusion

In this chapter we have designed and tested various heuristics to generate linear decompositions and studied their performance relative to previous suggested algorithms as well as to tree-width and clique-width. Caterpillar decompositions are simple to generate and give a quick upper bound on the boolean-width of a graph. As the experiments showed these orderings gave competitive upper bounds for boolean-width relatively fast compared to the approaches described in chapters 3 and 6.

We have already seen in Chapter 5 that preprocessing rules help in reducing the input graphs. Therefore designing preprocessing rules for caterpillar decompositions is an issue to investigate in order to handle larger graphs. Our vertex selection heuristics are rather general and not customized for specific structure of the given graph. Orderings with a focus on the graph structure might also be favorable for linear decompositions of relatively low width.

Chapter 9

Maximum Independent Set using Caterpillar Decompositions

The main motivation for computing width parameters of graphs is to be able to solve NP-hard problems on instances where the parameter is small. In the following chapter we will solve the Maximum Independent Set problem (ISP) using the caterpillar decompositions obtained from the algorithms developed in the previous chapter. To do this we must also have an algorithm that solves the ISP based on the decomposition.

In [7] Bui-Xuan et al. presented a number of dynamic programming algorithms parameterized by boolean-width. These algorithms traverse the rooted decomposition tree in a depth first manner. For each node the algorithm builds up solutions on the subgraph induced by vertices corresponding to the leaves of the processed subtree, with the final solution being computed when all the vertices are processed.

For caterpillar decompositions one end of the linear ordering is set as the root such that every internal node has a child that is a leaf. Thus in each step we are merging a larger solution with that of a leaf. This simplifies the corresponding dynamic programing algorithm. For instance, for the ISP the running time for a general boolean decomposition tree of boolean width k is $O(n^2 k 2^{2k})$, whereas for a caterpillar decomposition tree of boolean width k' one can solve the ISP in time $O(n2^{k'})$. Thus if the linear boolean-width is no larger than twice the boolean-width then this gives an indication that this approach could be competitive.

The ISP can also be solved using other width parameters. For a value k of the corresponding width parameter, this includes tree-width in $O^*(2^k)$ time, branch-width in $O^*(2.28^k)$ time, clique-width in $O^*(2^k)$ time, and rank-width in $O^*(1.42^{k^2})$ time. Similar results also exists for other problems such as the Dominating Set problem.

In this chapter we give an FPT algorithm for the ISP parameterized by linear boolean-width. Though linear boolean-width algorithms are implicit in the general algorithms in [7], our proposed algorithm differs substantially on the information stored across the cut. We evaluate the performance of our algorithm on a set of benchmark graphs.

9.1 Using a Linear Decomposition to solve the ISP

In the following we describe our dynamic programming algorithm for solving the ISP based on a linear decomposition. This algorithm is a variation of the algorithm for solving the ISP outlined in [7], using a general boolean decomposition. The current algorithm is given as Algorithm 14. It traverses the vertices from left to right as given by a linear decomposition (T, δ). All unprocessed vertices are kept in the set $Right$. As they are processed, they are moved to the set $Left$. Initially $Right = V(G)$ and $Left = \emptyset$.

For each processed vertex, the algorithm maintains a set LN containing different unions of neighborhoods of different independent sets in $Left$. One possibility could then be to store the size of all the independent sets seen so far among the processed vertices. But instead of doing this, if two independent sets have the same set of neighbors among the unprocessed vertices, only the size of a largest one is kept. In this way the number of different independent sets kept at any given time will never be higher than the number of different unions of neighborhoods across the $cut(Left, Right)$. It follows that we will always have $|LN| \leq 2^{lbw(T,\delta)}$.

When processing a new vertex v we check how this vertex interacts with the stored neighborhoods in LN. Let J be an element in LN. Then J represents an independent set I in $Left$ that is not explicitly stored. Instead we store the size of I in $J.size$ and the set $N(I) \cap Right$ in $J.elements$. We define an operator $LN.update(J)$ that will add J to LN if there is not already an entry with the same neighborhood as $J.elements$. If such an entry already exists then the $size$ value of this will be set to $J.size$ if this is smaller than its current value. To be able to easily iterate through the elements of LN we place any updated element of LN in a new set New_LN. When each element of LN has been processed we set LN equal to New_LN.

We now consider two cases depending on if the current vertex v is in $J.elements$ or not. Note that in either case v will be moved from $Right$ to $Left$.

Case (1): If $v \in J.elements$ then the independent set I that gave rise to J is still a valid independent set in $Left$, but with $N(I) = J.elements \setminus \{v\}$. We therefore update $J.elements$ accordingly before reinserting J. Note again that the outcome of this update operation is dependent on the already existing neighborhoods and their corresponding independent sets.

Now if $v \notin J.elements$ then we must consider two cases depending on if v is added to I or not.
Case 2(a): If v is not added to I then I is still an independent set in $Left$ and $N(I) \cap Right$ remains unchanged. We therefore store J unchanged in New_LN.
Case 2(b): If v is added to I then the size of I increases by one while $N(I)$ also contains $N(v) \cap Right$. We therefore update J accordingly before inserting it in New_LN.

When the algorithm terminates LN will contain an empty neighborhood where the corresponding $J.size$ will be the size of the maximum independent set in the given graph.

As an example of using Algorithm 14 consider the graph G in Figure 9.1.

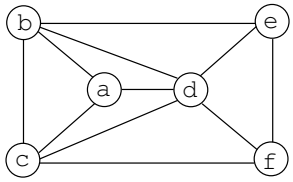

FIGURE 9.1: A graph G

Let the linear ordering for $V(G)$ be $\pi = \{a, b, c, d, e, f\}$ and assume that the algorithm has reached a stage when $Left = \{a, b, c\}$, $Right = \{d, e, f\}$ and d is the next vertex to be considered. For the $cut(\{a, b, c\}, \{d, e, f\})$ the set LN will contain $\{\{\emptyset, 0\}, \{\{d\}, 1\}, \{\{d, e\}, 1\}, \{\{d, f\}, 1\}\}$. These four sets of $Right$ correspond to the neighborhoods and sizes of the independent sets $\{\emptyset\}$, $\{a\}$, $\{b\}$, and $\{c\}$ in $Left$ respectively. After the next vertex d has been processed then the set LN will contain $\{\{\emptyset, 1\}, \{\{e\}, 1\}, \{\{f\}, 1\}, \{\{e, f\}, 1\}\}$. These four sets of $Right$ correspond to the neighborhoods and sizes of the independent sets $\{a\}$, $\{b\}$, $\{c\}$, and $\{d\}$ in $Left$ respectively.

Algorithm 14 : MAXIS(G, π)

Input: π, a linear arrangement of $V(G)$ of $G = (V(G), E(G))$ and G
Output: Size of the maximum independent set in $G = (V(G), E(G))$
$Left \leftarrow \emptyset$
$Right \leftarrow V(G)$
$LN \leftarrow \emptyset$
$LN.update(\{\emptyset, 0\})$
$MaxIS \leftarrow 0$
while $Right \neq \emptyset$ **do**
  $v \leftarrow NextVertex(\pi)$
  $New\_LN \leftarrow \emptyset$
  for $J \in LN$ **do**
    if (1): $v \in J.elements$ **then**
      $J.elements \leftarrow J.elements \setminus \{v\}$
      $New\_LN.update(J)$
    else
      // **2(a)**: v is not included in the current IS
      $New\_LN.update(J)$
      // **2(b)**: v is included in the current IS
      $J.size \leftarrow J.size + 1$
      $J.elements \leftarrow J.elements \cup (N_G(v) \cap Right)$
      $MaxIS \leftarrow$**Max**$(MaxIS, J.size)$
      $New\_LN.update(J)$
    end if
  end for
  $Left \leftarrow Left \cup \{v\}$
  $Right \leftarrow Right \setminus \{v\}$
  $LN \leftarrow New\_LN$
end while
return $MaxIS$

9.1.1 Proof of correctness

Assume that vertices $\pi(1)$ to $\pi(i)$ have been processed and that LN has been filled correctly. We show that after executing one more iteration of the while loop ($v = \pi(i+1)$) in Algorithm 14 the set LN will be filled correctly with the neighborhoods and sizes corresponding to the independent sets (I) in $Left$.

To show this we consider how v interacts with all the neighborhoods stored in LN and updates LN correctly for the next iteration. First we consider the neighborhoods in LN containing v (Case 1). The independent sets having these neighborhoods will not be affected by moving v from $Right$ to $Left$ and thus cannot result in larger independent sets than already computed. Next we consider neighborhoods not containing v. If v is not included in corresponding I (Case 2(a)) the new entry will be the same as the previous entry in LN. Similarly, including v in I will increase the size of I by one while expanding the neighborhood with $N_G(v) \cap Right$ as well (Case 2(b)). To finish the correctness proof, we need to show that if $max_{J \in LN}(J.size) = k$ then there exists an independent set of size k in G. Note that only the operation in Case 2(b) can increase the size of the largest independent set seen so far. Since LN has been filled correctly for up to $\pi(i)$ we have a new independent set in G with $I \cup \{v\}$ of size one larger than the size of I.

9.1.2 Time complexity and implementation

The while loop in Algorithm 14 is executed $O(n)$ times and loops over $|LN|$ entries. In each execution of this loop we must check if v is in $J.elements$ or not. As vertex subsets are implemented as bitsets, this is done in constant time. Whether an entry already exists in LN is determined by searching the hashmap which stores LN. The amortized time for searching in a hashmap is $O(1)$ and adding new element is also a constant time operation. The total runtime for Algorithm 14 is therefore $O(n \cdot |LN| \cdot O(1))$, hence $O(n2^{lbw(T,\delta)})$ since $|LN| \leq 2^{lbw(T,\delta)}$.

Even though the basic principle of Algorithm 14 is the same as the one in [7], there are some differences that allows us to speed up the computation. The algorithm in [7] solves the ISP and stores the solution for all different unions of neighborhoods across the cuts of the decomposition tree. This should be compared with Algorithm 14 that only stores the neighborhoods where there actually exists a valid independent set across the cut. Still, a linear boolean decomposition is more restricted than a general boolean decomposition and will therefore most likely give a higher linear boolean-width.

One observation that can be made from Algorithm 14 is that it is not strictly necessary that the linear order be precomputed before starting the algorithm. This opens up the possibility of selecting the next vertex in each step in such a way that it can exploit properties of the problem being solved. Besides using a precomputed ordering for the ISP, we have also experimented with an approach where the next vertex v from $Right$ was selected as the one being present in the most number of neighborhoods in $Right$. Since v cannot be combined with any independent set having v in their neighborhood, this will be Case (1) and neighborhoods will be updated only by extracting v. In order to find v according to this strategy we have to loop over the set LN and then for each entry count the number of times each vertex appears. This increases the running time of the while loop in Algorithm 14 by $O(|LN| \cdot |Right|)$.

9.2 Experimental results

In the following we describe experiments to compute the size of a maximum independent set for a given graph using Algorithm 14. The machine configuration is as described in Chapter 2 while the Algorithm has been developed in Java. To get orderings for the algorithm we use the heuristics from Chapter 8. For input graphs we mainly use the same benchmark graphs from TreewidthLIB [14] and the 2nd DIMACS [39] as we have used in previous chapters.

We have partitioned our experiments into two parts, the first is for graphs having less than 300 vertices, while the second part is for larger and denser graphs for the same reason as described in Chapter 8. In tables 9.1 and 9.2 we present results using small graphs from TreewidthLIB. We ran experiments on around 500 graphs and selected a set of representative graphs having a good variety in the different reported measures. These graphs are from different categories. Munin$*$, diabetes, and BN$*$ are from probabilistic networks, celar$*$ graphs are from frequency assignment, 1$*$ graphs are from protein structure, and pr$*$ graphs originate from Delanay triangulations. The zeroin.i.$*$, inithx.i.1, and mulsol.i.$*$ graphs are from the 2nd DIMACS implementation challenge and are generated from a register allocation problem based on real code.

The results from Algorithm 14 using vertex orderings from LeastUncommonNeighbor is labeled O_1. Similarly the results from using RelativeNeighborhood is labeled O_2, and LeastCutValue is labeled as O_3. Results for the runtime vertex selection approach is denoted by $O_{Runtime}$. In order to compute the runtime order we keep track of some additional statistics. For each vertex v in *Right*, i.e. the unprocessed ones, we maintain the number of different neighborhoods that contain v. According to these scores we pick the next vertex as one that is contained in the maximum number of different neighborhoods in *Right*. For each of the algorithm we report the size of the maximum independent set IS, the time needed to generate the ordering ,T_1 and time spent on the dynamic programming, T_{DP}. We also include the summation of the time spent by the heuristics and the dynamic programming, denoted by $Total$. Since DP indicates the total time for the runtime selection approach we only report this time for $O_{Runtime}$. In this way it is possible to compare the total time spent on computing the size of the maximum independent set for each graph for each heuristic. The linear boolean-width (lbw) upper bound for each heuristic is also listed in tables 9.1 and 9.2. For $O_{Runtime}$ we keep track of the order in which vertices are chosen to compute the associated linear boolean-width upper bound.

TABLE 9.1: Comparing running times for Algorithm 14 using different orderings

				O_1				O_2			
Graph	V	E	IS	T_1	T_{DP}	$Total$	lbw	T_1	T_{DP}	$Total$	lbw
BN_0-pp	67	243	21	0.001	0.38	0.381	16.53	0.003	0.078	0.081	15.47
BN_1-pp	74	340	21	0.004	0.29	0.294	17.98	0.008	0.054	0.062	17.92
huck	74	301	27	0.003	0.001	0.004	8.73	0.004	0.001	0.005	4.46
jean	80	254	38	0.003	0.003	0.006	8.64	0.003	0.001	0.004	5.81
david	87	406	36	0.001	0.008	0.009	10.46	0.004	0.001	0.005	9.33
celar02	100	311	34	0.002	0.007	0.009	9.32	0.008	0.001	0.008	7
miles1000	128	3216	8	0.014	0.002	0.016	8.74	0.02	0.001	0.021	9.07
anna	138	493	80	0.005	0.28	0.285	14.9	0.017	0.001	0.018	10.18
pr152	152	428	51	0.003	0.02	0.023	13.24	0.021	0.006	0.027	10.48
1g3p	185	2221	28	0.018	0.18	0.198	15.65	0.053	0.003	0.056	16.11
mulsol.i.2	188	3885	90	0.014	0.003	0.017	7.54	0.029	0.001	0.03	8.25
1cuk	189	2404	27	0.014	0.05	0.064	15.43	0.052	0.025	0.077	16.97
celar08-pp	189	1016	43	0.009	0.02	0.029	13.72	0.039	0.003	0.042	10.86
mulsol.i.1	197	3925	100	0.011	0.002	0.013	7.26	0.032	0.002	0.034	7.01
celar03	200	721	64	0.012	1.46	1.472	17.49	0.054	0.001	0.055	9.44
celar05	200	681	66	0.009	0.59	0.599	16.92	0.048	0.002	0.05	9.11
zeroin.i.1	211	4100	120	0.015	0.001	0.016	7.15	0.028	0.001	0.029	4.58
tsp225	225	622	73	0.01	0.03	0.04	12.25	0.083	0.11	0.193	16.99
fpsol2.i.1-pp	233	10783	63	0.056	0.005	0.061	8.14	0.049	0.003	0.052	7.1
munin4-wpp	271	724	101	0.013	0.006	0.019	9.64	0.122	0.003	0.125	9.78
a280	280	788	90	0.088	0.03	0.118	12.11	0.237	0.25	0.487	18.81
pr299-pp	286	828	92	0.011	0.19	0.201	14.53	0.174	0.026	0.2	15.84

TABLE 9.2: Comparing running times for Algorithm 14 using different orderings

	O_3				$O_{Runtime}$	
Graph	T_1	T_{DP}	Total	lbw	T_{DP}	lbw
BN_0-pp	10.166	0.034	10.2	13.58	0.091	17.51
BN_1-pp	47.675	0.062	47.737	15.01	0.16	18.5
huck	0.92	0.001	0.92	3.91	0.002	5.29
jean	1.211	0.001	1.211	4.7	0.001	6.75
david	2.632	0.002	2.634	7.12	0.001	8.41
celar02	2.922	0.001	2.923	6.39	0.001	6.58
miles1000	41.037	0.002	41.039	7.03	0.011	9.82
anna	13.57	0.001	13.571	10.02	0.008	12.28
pr152	17.833	0.006	17.839	10.36	0.03	15.58
1g3p	398.385	0.004	398.389	13.91	2.342	25.61
mulsol.i.2	71.776	0.001	71.777	4.81	0.004	8.82
1cuk	552.521	0.005	552.526	14.9	1.002	20.15
celar08-pp	32.675	0.007	32.682	10.95	0.063	13.61
mulsol.i.1	56.602	0.001	56.603	4.7	0.004	7.16
celar03	35.74	0.003	35.743	10.07	0.007	9.91
celar05	39.349	0.003	39.352	9.09	0.005	9.86
zeroin.i.1	54.73	0.001	54.731	3.32	0.008	7.14
tsp225	72.797	0.027	72.824	12.45	1.361	20.24
fpsol2.i.1-pp	343.975	0.003	343.978	5.78	0.009	6.32
munin4-wpp	100.178	0.004	100.182	9.27	0.018	11.67
a280	146.531	0.02	146.551	11.64	0.532	18.02
pr299-pp	174.346	0.012	174.358	9.95	0.497	17.85

For O_1 and O_2 the ordering generation time is comparatively smaller but the value of *lbw* is higher compared to O_3. It can be noted that for O_3 the total running times are higher than other approaches due to the greedy vertex selection. For this reason, despite spending less time on the DP compared to the other approaches, we did not use O_3 for larger graphs. From the results in tables 9.1 and 9.2 it can be calculated that for the total running time the average distance from the best algorithm for each graph is 1852% for O_1, 312% for O_2, 594598% for O_3, and 439% for $O_{Runtime}$. We can therefore conclude that O_2 and the runtime selection performs better than the other heuristics in terms of combined running time for the selected set of graphs. If we exclude the time for T_1 and only compare the dynamic programming time, then the average distances from the best algorithm is 9812% for O_1 , 100% for O_2, 30% for O_3, and 5307% for $O_{Runtime}$.

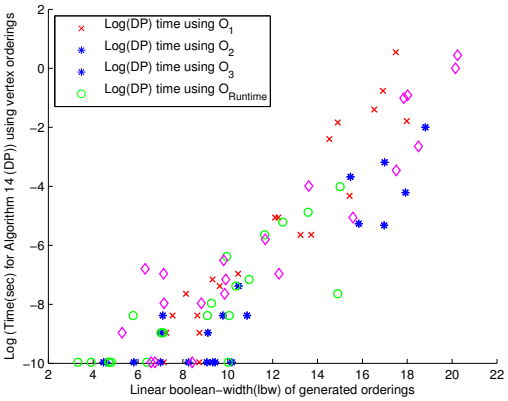

FIGURE 9.2: Logarithm (base 2) of the time required for dynamic programming for different orderings

In Figure 9.2 the logarithm base 2 of the time required for the dynamic programming is plotted against the linear boolean-width of the corresponding ordering for the graphs listed in Table 9.1. As can be observed from the above figure the logarithm of the DP time increases linearly with the *lbw*. This complements the running time from Section 9.1.2.

For the set of small graphs, O_2 and runtime ordering performed relatively better than O_1 and O_3 in terms of total running time. Therefore we compare O_2 and $O_{Runtime}$ for a set of larger and denser graphs. For this comparison we have used graphs from the 2nd DIMACS challenge [39]. We have selected instances for which the exact or approximate size of the maximum clique has been reported. From this set we tested on graphs having between 300 and 1035 vertices and report results for those which did not run out of memory due to the size of LN. Similar measures are reported as in tables 9.1 and 9.2. But as the *lbw* computation takes at least several hours for some of these graphs we have not calculated this value. Instead we list the maximum size of the set LN.

TABLE 9.3: Comparing running times for Algorithm 14 using different orderings for the 2nd DIMACS graphs

				O_2				$O_{Runtime}$	
Graph	V	E	IS	T_1	T_{DP}	$Total$	$\|LN\|$	T_{DP}	$\|LN\|$
p_hat300-3	300	33390	9	0.3	0.96	1.25	21436	7.1	20301
MANN_a27	378	70551	3	0.26	0.03	0.3	30	0.3	145
brock400_3	400	59681	7	0.7	1.9	2.6	32111	33.6	34729
San400_0.9_1	400	71820	5	0.58	0.94	1.52	747	20	757
San400_0.7_1	400	55860	11	0.7	0.05	0.75	16179	1	20127
johnson32-2-4	496	107880	31	1.2	0.07	1.3	840	7	3229
C500.9	500	112332	5	1.5	0.34	1.85	3202	7	3521
c-fat500-1	500	4459	40	0.73	0.002	0.73	6	0.03	6
c-fat500-10	500	46627	4	0.5	0.012	0.52	6	0.04	6
c-fat500-5	500	23191	8	0.6	0.006	0.6	6	0.04	6
p_hat700-3	700	183010	10	3.9	82.8	86.7	756052	1133	674885
keller5	776	225990	31	5.04	20.3	25.34	109753	1310	351350
C1000.9	1000	450079	6	12	4.2	16.2	20863	200.5	22591
hamming10-2	1024	518656	2	12.5	0.21	12.7	513	7.3	514
MANN_a45	1035	533115	3	5.85	0.17	6.02	48	5.1	376

Looking at the total run times in Table 9.3 the average distance from the best algorithm for each graph is 340% for O_2 and 734% for $O_{Runtime}$. If we exclude the time for T_1 and only compare the time spent on dynamic programming, then the average distance from the best algorithm is 19% for O_2. Similarly, excluding the time for vertex selection the average distance from the best algorithm is 29% for $O_{Runtime}$. It can also be noted that $|LN|$ is almost always smaller for O_2 than for $O_{Runtime}$. From the above experiments in can be remarked that even though LEASTCUTVALUE on small graphs takes much longer time than the other orderings, it pays of while doing dynamic programming. If the total running time is the main concern then O_2 performs better than other strategies for both the smaller and larger graphs. O_2 takes moderate time for generating the ordering and Algorithm 14 using O_2 also outperforms other approaches in terms of DP time. Moreover, it is applicable for larger graphs. It follows that based on the graph sizes and the available computation time one can alternate among different vertex orderings to get the best possible result.

9.3 Conclusion

As the experiments show, using dynamic programming along a caterpillar decomposition can be an alternative for solving NP-hard problems such as ISP. Which ordering strategy to use depends on the type of graph and the problem being solved, but it seems to be a good idea to tailor the strategy to the particular problem. The main drawback that we have experienced with this approach is that we sometimes run out of memory. This is because we have to store a large number of neighborhoods across each cut. For some graphs we ran out of memory when the number of different neighborhoods was larger than 2^{30}. In this case even if the graph has 100 vertices, the implementation is in the worst case handling a list of neighborhoods of size $100 * 2^{30}$ bits. This is 16 GB of memory and is more than we had available. Moreover, during the computation we look

up in LN if a particular neighborhood already exists or not. Though the amortized time for searching elements in a hashmap is $O(1)$, it might require more time when LN gets larger. It would also be of interest to find more memory efficient methods to store the neighborhoods.

We note that the approach in Algorithm 14 can easily be extended to general boolean decompositions to exploit issues such as connectedness of the underlying graph, preprocessing, and other techniques to speed up the computation. This has not been done in our current implementation but we will expand further on this issue in the next chapter. We have also implemented an algorithm from [7] for the ISP using a general decomposition tree. As this was done using a different underlying graph data structure we do not report the results. Although *lbw* of a caterpillar decomposition is the maximum boolean dimension over all cuts, Figure 9.2 shows the exponential trend in the running time of the given algorithm. Therefore it is of key importance that we look for orderings with as low linear boolean-width as possible. Moreover, if we spend more time to generate an ordering of low width, it can be reused for other dynamic programming algorithms parameterized by linear boolean-width.

Chapter 10

Maximum Independent Set using Branch-and-Bound

As we saw in Chapter 9 dynamic programming on linear decompositions can be used to efficiently solve the Maximum Independent Set problem (ISP) on a number of graphs. In order to test how good this approach is we now present and compare it to a branch-and-bound algorithm for the same problem. This algorithm is based on many of the same ideas that were used for developing CCMis in Chapter 4.

10.0.1 Background

ISP is an NP-hard optimization problem and is also complementary to the Maximum Clique problem that asks for the size of the largest clique in a graph. The relationship is that if a vertex set S is a solution to ISP on G then S is also a solution to the Maximum Clique problem on $\overline{G}$, where $V(G) = V(\overline{G})$ and where two vertices in $\overline{G}$ are adjacent if and only if they are not adjacent in G. Thus both solutions and algorithmic techniques that applies to one of these problems also applies to the other. If S is a solution to ISP then $V \setminus S$ is also a minimum vertex cover in G, that is a minimum subset of vertices such that each edge of G is incident on at least one vertex in the set.

The fastest exact algorithm for ISP on general graphs was given by Robson [85] and has a running time $O(1.1888^n)$. Fomin et al. [86] subsequently applied the Measure and Conquer technique to the analysis of a very simple backtracking algorithm obtaining a time bound of $O(1.2209^n)$. This is competitive with the so far best time bounds achieved by more complicated algorithms and analysis. A significant amount of work has also been done for sparse graphs [87]. These algorithms are all search-tree based using a branch and reduce paradigm along with a set of reduction and branching rules.

For some classes of graphs, including claw-free graphs and perfect graphs, ISP can be solved in polynomial time [88]. In general, ISP cannot be approximated to a constant factor in polynomial time (unless P=NP). However, there are efficient approximation algorithms for restricted classes of graphs, such as planar graphs and graph families closed under taking minors [89, 90].

In addition to the improvement in the theoretical running times for solving ISP/Clique, there has also been a substantial amount of experimental work to solve these problems

using search based heuristics. Most of this work has focused on the Clique problem. This includes a series of papers by Tomita et al. [57, 91–93]. A recent computational study and comparison of several algorithms for the Clique problem can be found in [94]. This paper also contains a summary of previous work on the Clique problem. Even more recent work is presented in [95, 96].

In the following we present an algorithm for solving ISP using the classical branch-and-bound approach. We employ a number of well known techniques to limit the search space. The main difference compared to previous implementations for ISP (or the Clique problem) is that our algorithm exploits connectedness. This is similar to Algorithm 8 from Chapter 4 that selects vertices for branching so that the remaining graph would more likely become disconnected. A variation of this idea was also proposed by Lipton and Tarjan [60] in the setting of planar graphs, but as far as we know has never tested experimentally.

We have conducted experiments using various benchmark graph data sets, including graphs from the 2nd DIMACS challenge [39] and the Florida Sparse Matrix library [97]. We also present comparisons with some recent Maximum Clique algorithms [95, 96] as well as the dynamic programming approach from Chapter 9.

10.1 Our algorithm

Consider a graph G on which we wish to solve ISP. Then one can branch on any vertex v obtaining two possible sets of solutions, those that include v in the current IS and those that do not include v. This observation lies at the bottom of all branching algorithms for ISP. The main difference between algorithms lies in the order in which vertices are selected and which pruning rules are used to discard infeasible solutions.

Thus a branch-and-bound algorithm will incrementally build up an IS of G by selecting or discarding vertices one by one from a candidate set P. Initially this set includes all vertices in $V(G)$. If a vertex is not to be in the current IS then it is moved from P to a set X, indicating that this vertex will not be considered again in the current branch. Once P is empty the current IS cannot be expanded further and the algorithm backtracks and explores the remaining branches.

Our algorithm follows the basic branching strategy creating two branches for every selected vertex v, one where v is in the current IS and one where it is not (i.e. v is placed in X). We note that other strategies are possible, for example the algorithm by Tomita [57] will select a vertex v and then create one branch for every vertex in $N[v]$, the intuition being that for a maximal IS at least one vertex in $N[v]$ must be included.

Before going into the details of the algorithm we first point out some simple reduction rules that we employ to simplify a given instance.

10.1.1 Preprocessing/reduction rules

There exists a number of well known preprocessing rules for reducing an instance of ISP/Clique/Vertex cover [98]. In our algorithm we employ the following ones:

1. Singleton: Include any vertex v of degree 0 in the IS and remove v from G.
2. Pendant: Include any vertex of degree 1 in the IS and remove all vertices in $N[v]$ from G.
3. Simplicial: If $N[v]$ induces a clique in G then include v in the IS and remove $N[v]$ from G.
4. Neighborhood subset: If $\exists v, w \in P : N_G[v] \subseteq N_G[w]$ then remove w from G.
5. Path: If $\exists u, v \in P$ both of degree 2 such that $\{u, v\} \in E(G)$ then remove u and v from G and increase the IS count by 1. In addition, add the edge $\{x, y\}$, where $x \in ((N(u) \setminus v)$ and $y \in (N(v) \setminus u)$.

Any vertex found to satisfy Rule 3 is said to be *simplicial*. Note that Rule 3 covers rules 1 and 2 as special cases. We still choose to include these as separate rules as they are simpler and easier to check for during computation. The intuition for Rule 3 is that at least one vertex in $N[v]$ has to be included in a maximal IS and then v is the one that restricts the current solution the least.

The motivation for Rule 4 is that any IS that includes w can be transformed by replacing w with v in the IS without decreasing the size of the solution. To see this consider a solution that includes w, then clearly no vertex in either $N[v] \setminus w$ or $N(w) \setminus N(v)$ can be in the solution. But replacing w with v in the IS maintains the size of the current solution and possibly allows for vertices from $N(w) \setminus N(v)$ to be included in the IS.

Figure 10.1 outlines the actions taken in Rule 5. The idea behind this rule is that there will always exist an optimal solution that includes either u or v. To see this consider a solution that neither includes u nor v. To be maximal this solution must include both x and y. But then we can replace x with u (or y with v) and get a less restrictive solution of the same size.

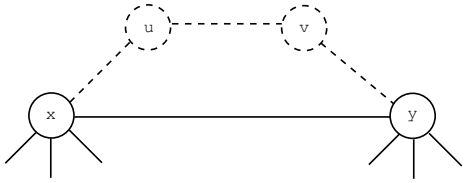

FIGURE 10.1: Degree 2 vertex reduction

We apply all of these rules initially before starting the main recursive algorithm. The rules are applied iteratively until no further rule is applicable. During the execution of the algorithm we also check if Rule 3 can be applied but only for vertices of degree at most two, thus including rules 1 and 2.

10.1.2 The recursive algorithm

In the following we outline the structure of the main recursive procedure. We note that this strongly resembles Algorithm 8 from Chapter 4. The current version of the

algorithm only computes the size of the largest IS, but it would be straight forward to return the actual vertices as well.

The algorithm considers the vertices in some linear order. Initially this order contains all vertices in G but as the algorithm progresses vertices are removed from G. We keep track of the remaining vertices that can be included in an IS by maintaining a set of active vertices P. In addition we maintain a set X of vertices that explicitly cannot be included in the current IS.

For each considered vertex v the algorithm makes two recursive calls, one where v is included in the IS and where all vertices in $N[v]$ are removed from P and X, and one where v is excluded from the solution and thus moved from P to X. When the algorithm backtracks from these two calls, v will again be placed in P before returning. Throughout the computation the algorithm maintains the size of the largest IS seen so far.

To avoid having to test every possible configuration it is important to be able to cut of branches of the computation if they cannot lead to a solution better than the current best one. We do this in several ways. At the start of the recursive procedure we first test if there is a vertex in X that is not adjacent to any vertex in P. If this is the case it follows that the current solution cannot be expanded to a maximal IS and therefore neither to a maximum IS.

We compute upper and lower bounds on the size of the largest IS in P. The lower bound is computed by running a greedy IS algorithm on the vertices in P. This selects vertices in P as long as they are not adjacent to any previously selected vertex. Thus the number of vertices already included in the current IS plus the size of the greedily selected set gives a possible solution. If this is larger than the largest IS seen so far, we update this accordingly.

For the upper bound we use a *clique cover*. This is a partition of the vertices in G into disjoint subsets such that the vertices in each set induces a clique in G. Then it follows that an IS can at most include one vertex from each such set. Thus the number of sets for any clique cover gives an upper bound on the maximum IS in G. To avoid having to compute a clique cover for every recursive call, we instead do this before the first time the recursive routine is called but after the initial reduction rules have been applied. This is done using a greedy algorithm. Then during the recursive computation we keep track of the number of cliques that still have vertices left in P. It follows that the current solution cannot be expanded by more than this number. Thus if the size of the current solution plus the number of remaining cliques is not larger than the best solution seen so far, then the current solution cannot be expanded to a new optimal solution and we return from the current call. Also, if the number of remaining cliques is equal to the lower bound given by the greedy IS algorithm, then the greedy solution must be optimal and there is no need to pursue the current branch further. The idea of using a clique cover for the ISP problem is equivalent to the use of a vertex coloring for the Clique problem [94].

The final technique we use is to test at each level of recursion if $G[P \cup X]$ is connected. We do this using a linear depth first search through $G[P \cup X]$. If this graph is not connected then the recursive procedure is called once for each connected component to compute the size of its maximum IS. As each component is treated individually it should be possible to get tighter upper and lower bounds in this way. Also, we expect the sum

of the processing times for each component to be less than if the graph is treated as a whole. Still, testing for connectedness comes at a cost. Thus it is not immediately clear if the overall contribution of this approach will be positive.

Note that throughout the computation we maintain the current degree of each vertex in order to easily check if we can apply the simplicial reduction rule on any vertex of degree at most two.

The algorithm is given as Algorithm 15: MaxIS() and is initially called with MaxIS$(V, \emptyset, 0, 0)$. The two first parameters P and X have already been described. The third parameter $SolutionSoFar$ gives the size of the current IS computed prior to this call while the fourth parameter $RemainingUpperBound$ gives an upper bound on the largest IS that can be found in other connected components apart from the current one that is being processed.

Initially, a series of tests are made to check if this is a trivial case where we immediately can determine the outcome. Whenever it is found that it is not possible to compute a solution larger than the current best lower bound the algorithm will return a value of $-\infty$. We are assuming that there is a global variable $GlobalBest$ where we will keep the current best lower bound.

Next, if $G[P \cup X]$ is disconnected the algorithm starts processing the different connected components. To do this we first compute upper and lower bounds for each component using the number of cliques each component intersects with and a greedy IS algorithm, respectively. We also accumulate all the upper bounds in a variable $NewUpperBound$. When processing a specific component this variable will contain an upper bound on all other unprocessed connected components. Note that this also includes components not visible on the current level of recursion. Next, the size of a maximum IS of each component is computed recursively. While doing this we accumulate the size of this IS in the variable $CurrentSolution$. Note that the upper bound on the unprocessed components is decremented accordingly for each component. Also, we can avoid the recursion if the lower and upper bounds of a component are equal. If it should become evident that the current solution cannot lead to a new better global solution (by a return value of $-\infty$) the current call immediately exits.

If the current graph is connected then a vertex v is selected to branch on. Two branches are created, one where v is in the current solution and one where it is not. For each of these, new sets for P and X are computed before the routine TestConfig is called. This is given as Algorithm 16. TestConfig first removes any simplicial vertices of degree at most 2 (using Algorithm 17) before computing upper and lower bounds for the current graph. The lower bound is then used to test if we have a new better global solution (where the best solution is stored in $GlobalBest$). If the solution given by the lower bound is optimal or if there are not sufficient vertices left that a new better global solution can be reached the algorithm returns. If this is not the case then a new recursive call to MaxIS is made to continue solving this component.

Algorithm 15 : MaxIS($P, X, SolutionSoFar, RemainingUpperBound$)

Input: Two vertex sets P and X, the size of the IS found so far, and an upper bound on the optimal IS of other connected components than the current one.
Output: Size of the maximum IS in $G[P \cup X]$ containing only vertices from P, $-\infty$ if this cannot be expanded to a new optimal solution.

```
if P ∪ X = ∅ then
  return 0
end if
if ∃w ∈ X with no neighbor in P then
  return −∞
end if
if |P| = 1 then
  return 1
end if

if G[P ∪ X] is disconnected then
  NewUpperBound = RemainingUpperBound
  for each connected component CC_i(V_i, E_i) of G[P ∪ X] do
    UpperBound_i ←RemainingCliques(V_i)
    LowerBound_i ←GreedyIS(V_i)
    NewUpperBound ← NewUpperBound + UpperBound_i
  end for

  CurrentSolution ← 0
  for each connected component CC_i(V_i, E_i) of G[P ∪ X] do
    NewUpperBound ← NewUpperBound − UpperBound_i
    if UpperBound_i = LowerBound_i then
      CurrentSolution ← CurrentSolution + UpperBound_i
    else
      CurrentSolution ← CurrentSolution+
      MaxIS(V_i ∩ P, V_i ∩ X, SolutionSoFar + CurrentSolution, NewUpperBound)
    end if
    if CurrentSolution = −∞ then
      return −∞
    end if
  end for

  return CurrentSolution
end if

Select a vertex v ∈ P to branch on
InP = P \ N_G[v]
InX = X \ N_G(v)
CurrentSolution ← 1
InTotal = TestConfig(InP, InX, SolutionSoFar, CurrentSolution,
RemainingUpperBound)

OutP = P \ {v}
OutX = X ∪ {v}
CurrentSolution ← 0
OutTotal = TestConfig(OutP, OutX, SolutionSoFar, CurrentSolution,
RemainingUpperBound)

return max{InTotal, OutTotal}
```

Algorithm 16 : TestConfig($P, X, SolutionSoFar, CurrentSolution, RemainingUpperBound$)

Input: Vertex sets P and X, various bounds as explained in the text.
Output: The size of a maximum IS in $G[P]$, $-\infty$ if this cannot be expanded to a new optimal solution.

$CurrentSolution \leftarrow CurrentSolution +$ Reduce(P, X)
$UpperBound \leftarrow$RemainingCliques(P)
$LowerBound \leftarrow$GreedyIS(P)
$GlobalBest \leftarrow \max(GlobalBest, SolutionSoFar + CurrentSolution + LowerBound)$
if $UpperBound = LowerBound$ **then**
  return $CurrentSolution + UpperBound$
end if
if $GlobalBest \geq SolutionSoFar + CurrentSolution + UpperBound + RemainingUpperBound$ **then**
  return $-\infty$
end if
return $CurrentSolution +$
MaxIS($P, X, SolutionSoFar + CurrentSolution, RemainingUpperBound$)

Algorithm 17 : Reduce(P, X)

Input: Vertex sets P and X.
Output: Number of vertices included in the IS by reduction rules. Updated sets P and X.

$NrIncluded \leftarrow 0$
while $\exists v \in P$ such that $deg(v) \leq 2$ and v is simplicial in $G[P]$ **do**
  $NrIncluded \leftarrow NrIncluded + 1$
  $P \leftarrow P \setminus N[v]$
  $X \leftarrow X \setminus N(v)$
end while
return $NrIncluded$

10.1.3 Vertex ordering

The only design issue that remains to specify is in what order vertices are selected to branch on. This is a crucial decision in any branch-and-bound algorithm as obtaining good solutions early on in the computation can allow for more pruning later on. We have experimented with a number of different orderings, including the following.

- Process vertices according to a nested dissection ordering on the original graph. Then when the vertices of a separator have either been included in the current IS or removed, the remaining graph will be disconnected. To avoid that vertices placed in X maintain the connectedness it is possible to first branch on all neighbors of a vertex from the current separator that has been placed in X. This is similar to what was done in Algorithm CCMis.

- Order the vertices according to the initial clique cover, with vertices in larger cliques coming before vertices in smaller cliques. Then when a vertex is selected to be in the IS the rest of the vertices in its clique gets removed.
- Order vertices based on a nested dissection ordering of the clique graph H. This graph is constructed by replacing each clique from the original clique cover on G by a node in H. Two nodes in H are adjacent if the corresponding cliques contains a vertex each that are adjacent in G. The vertices of each clique would then be ordered consecutively, but the order between the cliques would be determined by the nested dissection ordering on H. The intuition behind this ordering is that one can at most select one vertex from each clique and whenever a vertex from each of the nodes of the first separator in H has been included in the current IS, then the remaining graph of G must be disconnected.

10.1.4 Experimental results

In the following we describe experiments performed on Algorithm 15: MaxIS.

We use the same machine configuration as described in Chapter 2. The algorithm has been implemented in C and each reported timing is given in seconds and is the average of five runs. For these experiments we use graphs from TreewidthLIB [14], the Florida Sparse Matrix collection [97], the DIMACS challenge [39], and from BHOSLIB [40]. TreewidthLIB and the Florida Sparse matrix collection include graphs from various computational problems and real life applications as described in Chapter 2. From the DIMACS challenge we have chosen graphs that were also used in [95, 96] to facilitate comparison. The BHOSLIB graphs are also chosen as they have been used [96].

Among the different vertex orderings we have tried, we found that in almost all cases it was most efficient to order the vertices according to the initial clique cover, with vertices belong to large cliques being processed before vertices of smaller cliques. This consistently outperformed all versions where the graph was ordered according to some nested dissection ordering, either on G itself or on the clique meta graph. For this reason we only present results using the clique ordering and report timings where the only variation is whether we employ the discovery and processing of connected components or not.

As there is very little experimental work on solving ISP we instead compare our results with those from computing the maximum clique in a graph. To do this we first take the complement of the considered graph before applying our algorithms. As we did not have access to the code we make comparisons with what is reported in the papers. For this reason one should not read too much into the relative numbers. All processors involved are Nehalem based Xeon processors from the same generation. Also, in many cases a problem is either solved rapidly or not solved at all. Thus these comparisons mainly show if our algorithm is in the correct ballpark or not.

The time required for preprocessing, identifying connected components, and checking for branches that can be pruned was negligible compared to the total branching time and never more than 10% of the total time. To see the effect of disconnectedness on the total running time we therefore report the total time of Algorithm 15 including, $WithCC$ and excluding, $WithoutCC$ connected component part.

Our first set of experiments is a comparison with the work done in [96]. This presents a parallel branch-and-bound algorithm for the maximum clique problem. The algorithm processes the vertices by non-increasing vertex degree and uses a graph coloring to bound the search (in a similar way as using a clique cover for ISP). It is also optimized to use bit-set encoding. Experiments are performed on a set of DIMACS graphs [39] and graphs from BHOSLIB [40]. For the experiments they used a computer with two 2.4 GHz Intel Xeon E5645 processors, with a total of at most 12 threads (24 with hyper threading).

Algorithm 15 did not finish within 60 minutes on any of the complemented DIMACS graphs. We note that the running times reported in [96] for these graphs varies from seconds to several weeks. The results from running Algorithm 15 on the graphs from BHOSLIB are given in Table 10.1. These graphs were not reduced during the initial preprocessing step. We also show timings from [96], both when running on a single thread and also when using 24 threads. For all of these graphs the dynamic programming approach from Chapter 9 ran out of memory. For each graph we report the size of the maximum independent set in the column labeled IS.

TABLE 10.1: Comparing Algorithm 15 with the algorithm of McCreesh et al.

				MaxClique from [96]		Algorithm 15	
Graph	V	E	IS	Single thread	24 threads	*WithCC*	*WithoutCC*
Frb30-15-1	450	17827	30	657.1	35.5	167	118
Frb30-15-2	450	17874	30	1183.1	65.8	456.8	339
Frb30-15-3	450	17809	30	356.7	19.5	289.1	190
Frb30-15-4	450	17831	30	1963.2	124.4	45	32
Frb30-15-5	450	17794	30	577.1	42.1	520	354
Frb35-17-1	595	27856	35	51481.7	2532	896	448
Frb35-17-2	595	27847	35	91275	5677.3	3718	1665
Frb35-17-3	595	27931	35	33852.1	2349.3	1537	571
Frb35-17-4	595	27842	35	37629.2	2196.1	1232	348
Frb35-17-5	595	28143	35	205356	10363.4	6728	3702

From the results in Table 10.1 it can be observed that the running time of Algorithm 15 is lower if we omit checking for connected components. This indicates that these graphs do not disconnect well. Comparing the results with those from [96] we see that Algorithm 15 without checking for connected components, is always faster than the sequential algorithm from [96] and in many cases it is faster than the parallel algorithm using 24 threads. We believe that this is due to that Algorithm 15 is able to compute good upper and lower bounds and thus to prune more of the computation. We note that obtaining the lower bound on the fly comes at a cost as we compute a greedy IS in each recursive step of the algorithm.

Our next set of experiments consists of a set of graphs from the DIMACS challenge [39] on which Pattabiraman et al. [95] were unsuccessful in computing the maximum clique size. In their studies they used a sequential branch-and-bound code particularly suited for sparse graphs and running on a 2.00 GHz Intel Xeon E7540 processor. We note that there is no overlap between the DIMACS instances from [96] and those in [95]. We believe that this might be due to [96] omitting instances where their code did not use enough time to warrant using a parallel algorithm. The data from the experiments is given in Table 10.2. Note that all reported graph statistics are on the complemented graphs.

TABLE 10.2: Running times for Algorithm 15 on a set of graphs from the 2nd DIMACS challenge

					Algorithm 15	
Graph	V	E	IS	Density	$WithCC$	$WithoutCC$
brock200_1	200	5066	21	0.25	631	1560
san200_0._1	200	5970	30	0.3	1.5	975.9
hamming8-2	256	1024	128	0.03	< 0.01	< 0.01
hamming8-4	256	11776	16	0.36	202	306.3
MANN_a27	378	702	126	0.01	55.1	40
p_hat500-2	500	61804	36	0.5	3000	> 10000
c-fat500-5	500	101559	64	0.81	< 0.01	< 0.01
hamming10-2	1024	89600	512	0.17	< 0.01	< 0.01

From the results in Table 10.2 it can be observed that the running time of Algorithm 15 increases if we omit checking for connected components except for the graph MANN_a27. We also tried the dynamic programming approach for the graphs in Table 10.2. Using vertex orderings from Algorithm 12 Algorithm 14 took 0.12 seconds for the graph c-fat500-5 and 258.4 seconds for the graph hamming8-4. For the rest of the graphs it ran out of memory. Note that for these two graphs Algorithm 15 performs better than Algorithm 14. None of graphs in Table 10.2 reduced during the initial reduction phase.

Results from the final set of experiments are reported in Table 10.3. Here we compare Algorithm 15 and Algorithm 14 from Chapter 9 on graphs from TreewidthLIB. This includes all graphs that were used in the experiments in Chapter 9. For several of these the ISP specific reduction rules were able to reduce the graph completely or to just a few remaining vertices. Thus we omit these from our listing. In addition we have included a representative set of graphs where at least one of the algorithms was able to compute a solution within 100 seconds. We put an * for any algorithm that did not finish within one hour.

For Algorithm 14, we report running times using vertex orderings from Algorithm 12 RELATIVENEIGHBORHOOD (O_2) and using runtime vertex selection ($O_{Runtime}$). For O_2 we report the time needed to generate the ordering (T_1), and the time spent on the dynamic programming (T_{DP}). We also include the summation of the time spent by the heuristics and the dynamic programming ($Total$). Since DP gives the total time for the runtime selection approach we only report this time for $O_{Runtime}$. In this way it is possible to compare the total time spent on computing the size of the maximum IS for each graph for each approach. These graphs reduce to some extent during the preprocessing and we report the number of remaining vertices (V_{pp}) and edges (E_{pp}).

TABLE 10.3: Comparing running times of Algorithm 15 and Algorithm 14

						Algorithm 15		Algorithm 14			
Graph	V	E	V_{pp}	E_{pp}	IS	$WithCC$	$WithoutCC$	T_1	T_{DP}	$Total$	T_{DP}
Queen10_10	100	1470	100	1470	10	0	0	0.02	1.1	1.12	3.7
eil101	101	290	100	285	32	4.24	5.01	0.02	0.008	0.03	0.006
games120	120	638	117	612	22	0.21	3.53	0.02	1.9	1.92	2.76
Queen11_11	121	1980	121	1980	11	0.01	0.01	0.03	8.7	8.73	27.1
1on2	135	1527	108	944	20	0.68	1.18	0.01	0.004	0.02	0.04
kro150	150	432	119	313	48	11.21	7.76	0.02	0.04	0.06	0.003
1g3p	185	2221	120	1069	28	10.63	2.43	0.02	0.01	0.03	0.05
1cuk	189	2404	135	1136	27	32.26	15.33	0.03	0.02	0.05	0.22
graph04	200	734	116	294	52	4.61	*	0.02	0.15	0.17	0.6
tsp225	225	622	73	206	534	*	*	0.08	0.11	0.19	1.36
a280	280	788	242	651	90	*	*	0.1	0.004	0.104	0.6
pr299-pp	286	828	92	286	828	*	*	0.17	0.02	0.19	0.49
rajat04	1041	4317	215	535	578	3.02	0.08	0.22	0.14	0.36	0.18

We observe again that there is no clear cut decision on whether one should use exploit the connected components in Algorithm 15. There are instances where each approach outperforms the other. The same is true when comparing the total time spent on solving each graph. For the 1* graphs from protein structure Algorithm 14 using O_2 performs better than the other approaches. Whereas for the Queen* graphs Algorithm 15 outperforms the dynamic programming algorithm.

10.1.5 Conclusion

We have presented a simple branch-and-bound algorithm for ISP. The novelty of the algorithm is in that it exploits that the graph can become disconnected during the processing. Experiments showed that there are cases where the algorithm is competitive with state of the art branch-and-bound algorithms.

Comparisons with Algorithm 14 showed that the dynamic programming approach can be the best choice. However, for this to be a feasible approach for more instances one must limit the memory use of the algorithm. This could possibly be done by developing specific reduction rules that could exploit that some solutions cannot be expanded to global optimal solutions. Also, one could try to keep track of the connected components similarly to what is done in Algorithm 15.

Chapter 11

Conclusion

In recent years a noticeable amount of computational experiments has been done to explore the upper and lower bounds, as well as exact values of different width parameters of graphs in practical settings [12, 13, 15, 16, 23, 24, 26]. Our focus has been to do the same for boolean-width and demonstrate its usefulness in solving hard problems.

In this thesis we have designed several algorithms for computing boolean decompositions and tested their suitability compared to other well known graph parameters. Moreover, we have tried simple preprocessing rules to reduce the input size. To demonstrate the practical use of these methods we have implemented a dynamic programming algorithm using the generated decompositions to solve the Maximum Independent Set problem. The results of this thesis indicates that boolean-width can be used not only in theory but also in practice and that it is competitive with other approaches. Still there is much potential for improvement. In this chapter we present some open problems and point to possible future directions for advancing the implementations.

11.1 Open problems

To be able to use boolean-width in practical settings, generating boolean decompositions of small width reasonably fast is of high relevance. Our experimental results shows that fast evaluation of boolean dimension can aid in this regard. In addition more intricate reduction rules could help to simplify the problem. Finally, designing meaningful experiments and performance analysis is of great importance. Consequently, the selection of graphs to test the proposed heuristics also plays a vital role. In the following we discuss some observations from this thesis that might lead to further relevant research.

- Maximum matching on the bipartite graph induced by a cut is an upper bound on the boolean dimension of the cut. We can use this upper bound whenever computing the exact boolean dimension is too expensive. Tighter upper bounds can also help in this regard.
- An approximation algorithm for boolean dimension can also be a good practical alternative to the exact boolean dimension.

- We employed local search to improve an initial decomposition in Chapter 3. This was based on random swapping of vertices across the cuts of the decomposition tree. Improving the local search search strategy, the predefined time, and the search space can be potential areas for overall improvement of the generated decompositions. Swapping vertices can be based on their neighborhoods across the cuts as well as their internal neighbors. Moving to a new upgraded cut or staying in the current state can be decided probabilistically as is done in simulated annealing. The allowed time for local search could also be a function of improvement and cost.
- Implementing decomposition algorithms for special graph classes, like planar graphs, can be investigated.
- The preprocessing rules we tried are fairly simple. More complex reduction rules would be of interest.
- Boolean-width based dynamic programming algorithms can be tested and compared against similar approaches parameterized by other width parameters.
- As dynamic programming using linear decompositions runs out of memory when the linear boolean-width is high, storing neighborhoods efficiently could help this to improve in some extent.
- Several algorithms in this thesis could most likely benefit from using parallel methods.

Bibliography

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

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

[3] N. Robertson and P. D. Seymour. Graph minors. XIII. The disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63:65–110, 1995.

[4] B. Couecelle, J. Engelfriet, and G. Rozenberg. Handle-rewriting hyper-graph grammars. *Journal of Graph Theory*, 46(2):218–270, 1993.

[5] P. Hliněný and S. Oum. Finding branch-decomposition and rank-decomposition. In *Proceedings of the 15th Annual European Symposium on Algorithms, ESA 2007*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 4698, pages 163–174. Springer Berlin Heidelberg, 2007.

[6] P. Hliněný, S. Oum, D. Seese, and G. Gottlob. Width parameters beyond tree-width and their applications. *The Computer Journal*, 51(3):326–362, 2008.

[7] B. M. Bui-Xuan, J. A. Telle, and M. Vatshelle. Boolean-width of graphs. *Theoretical Computer Science*, 412(39):5187–5204, 2011.

[8] J. M. M. van Rooij, H. L. Bodlaender, and P. Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In *Proceedings of the 17th Annual European Symposium on Algorithms, ESA 2009*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 5757, pages 566–577, 2009.

[9] S. Oum. Rank-width is less than or equal to branch-width. *Journal of Combinatorial Theory, Series B*, 57(3):239–244, 2008.

[10] H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996.

[11] H. Röhrig. Tree decomposition: A feasibility study. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1998.

[12] H. L. Bodlaender and A. M. C. A. Koster. Treewidth computations I. Upper bounds. *Information and Computation*, 208:259–275, 2010.

[13] H. L. Bodlaender. Treewidth: Characterizations, applications, and computations. In *Proceedings of the 32nd International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2006*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 4271, pages 1 – 14, 2006.

[14] TreewidthLIB. A benchmark for algorithms for treewidth and related graph problems. See http://www.staff.science.uu.nl/ bodla101/treewidthlib/, 2004.

[15] I. V. Hicks, A. M. C. A. Koster, and E. Kolotoğlu. Branch and tree decomposition techniques for discrete optimization. In *TutORials 2005*, INFORMS Tutorials in Operations Research Series, chapter 1, pages 1–29. INFORMS Annual Meeting, 2005.

[16] A. Overwijk, E. Penninkx, and H. L. Bodlaender. A local search algorithm for branchwidth. In *Proceedings of the 37th Conference on Current Trends in Theory and Practive of Computer Science, SOFSEM 2011*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 6543, pages 444–454, 2011.

[17] Y. Song, C. Liu, R. Malmberg, F. Pan, and L. Cai. Tree decomposition based fast search of RNA structures including pseudoknots in genomes. In *Proceedings of the 2005 IEEE Computational Systems Bioinformatics Conference, CSB'05*, pages 223–234, 2005.

[18] J. Zhao, D. Che, and L. Cai. Comparative pathway annotation with protein-DNA interaction and operon information via graph tree decomposition. In *Proceedings of Pacific Symposium on Biocomputing, PSB 2007*, volume 12, pages 496–507, 2007.

[19] J. Zhao, R. L. Malmberg, and L. Cai. Rapid ab initio prediction of RNA pseudoknots via graph tree decomposition. *Journal of Mathematical Biology*, 56 (1–2):145–159, 2008.

[20] H. Chen. Quantified constraint satisfaction and bounded treewidth. In *Proceedings of the 17th European Conference on Artificial Intelligence, ECAI 2004*, pages 161–165, 2004.

[21] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, 2000.

[22] S. J. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *The Journal of the Royal Statistical Society. Series B (Methodological)*, 50:157–224, 1988.

[23] I. V. Hicks. *Branch Decompositions and their Applications.* PhD thesis, Rice University, Houston, Texas, 2000.

[24] M. Beyß. Fast algorithm for rank-width. In *Proceedings of Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science MEMICS'12*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 7721, pages 82–93, 2013.

[25] E. M. Hvidevold, S. Sharmin, J. A. Telle, and M. Vatshelle. Finding good decompositions for dynamic programming on dense graphs. In *Proceedings of the 6th International Symposium on Parameterized and Exact Computation, IPEC'11*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 7112, pages 219–231, 2011.

[26] M. H. Heule and S. Szeider. A sat approach to clique-width. In *Proceedings of 16th International Conference on Theory and Applications of Satisfiability Testing, SAT 2013*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 7962, pages 318–334. Springer Berlin Heidelberg, 2013.

[27] W. Cook and P. D. Seymour. An algorithm for ring-router problem. Technical report, Bellcore, 1994.

[28] B. Verweij. *Selected applications of integer programming: A computational study.* Phd thesis, Department of Information and Computing Sciences, Utrecht University, 2000.

[29] A. M. C. A. Koster, S. P. M. van Hoesel, and A. W. J. Kolen. Solving frequency assignment problems with tree decomposition. Technical report, Maastricht University, 1999.

[30] A. M. C. A. Koster, S. P. M. van Hoesel, and A. W. J. Kolen. Lower bounds for minimum interference frequency assignment problems. *Ricerca Operativa*, 50 (94-95):101–116, 2002.

[31] I. Adler, B. M. Bui-Xuan, Y. Rabinovich, G. Renault, J. A. Telle, and M. Vatshelle. On the boolean-width of a graph: Structure and applications. In *Proceedings of the 36th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2010*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 6410, pages 159–170, 2010.

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

[33] H. L . Bodlaender. Dynamic programming on graphs with bounded treewidth. *Automata, Languages and Programming*, Springer Verlag, Lecture Notes in Computer Science, vol. 317:105–118, 1988.

[34] P. K. Krause. A program that calculates rank-width and rank-decompositions, rw v 0.2. http://pholia.tdi.informatik.uni-frankfurt.de/ philipp/software/rw.shtml.

[35] R. J. Wilson. Introduction to graph theory. Fourth edition. Prentice Hall, 1996.

[36] R. Diestel. Graph theory (graduate texts in mathematics). Fourth edition. Springer-Verlag, 2010.

[37] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, page 85–103., 1972.

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

[39] DIMACS. The second DIMACS implementation challenge: NP-Hard Problems: Maximum Clique, Graph Coloring, and Satisfiability. See http://dimacs.rutgers.edu/Challenges/, 1992–1993.

[40] BHOSLIB. Benchmarks with hidden optimum solutions for graph problems. See http://www.nlsde.buaa.edu.cn/ kexu/benchmarks/graph-benchmarks.htm.

[41] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. A simple model to generate hard satisfiable instances. In *Proceedings of 19th International Joint Conference on Artificial Intelligence, IJCAI 2007*, volume 171, pages 337–342, 2005.

[42] P. Erdös and A. Rényi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5:17–61, 1960.

[43] E. Weisstein. Wolfram mathworld - the web's most extensive mathematics resource. See http://mathworld.wolfram.com.

[44] P. Larranaga, C.M.H Kujipers, M. Poza, and R.H. Murga. Decomposing bayesian networks: triangulation of the moral graph with genetic algorithms. *Statistics and Computing (UK)*, 7(1):19–34, 1991.

[45] F. Clautiaux, A. Moukrim, S. Ngre, and J. Carlier. Heuristic and meta-heurisistic methods for computing graph treewidth. *RAIRO-Operations Research*, 38(1): 13–26, 2004.

[46] U. Kjaerulff. Optimal decomposition of probabilistic networks by simulated annealing. *Statistics and Computing*, 2(1):2–17, 1992.

[47] N. Musliu. Generation of tree decompositions by iterated local search. In *Evolutionary Computation in Combinatorial Optimization*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 4446, pages 130–141. 2007.

[48] T. Hammerl and N. Musliu. Ant colony optimization for tree decompositions. In *Evolutionary Computation in Combinatorial Optimization*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 6022, pages 95–106. 2010.

[49] Y. Rabinovich, J. A. Telle, and M. Vatshelle. Upper bounds on boolean-width with applications to exact algorithms. In *Proceedings of the 6th International Symposium on Parameterized and Exact Computation, IPEC'13*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 6410, pages 159–170, 2013.

[50] S. P. Vadhan. The complexity of counting in sparse, regular, and planar graphs. *SIAM Journal on Computing*, 31(2):398–427, 1997.

[51] J. Radhakrishnan M. M. Halldorsson. Improved approximation of independent sets in bounded-degree graphs via subgraph removal. *Nordic Journal of Computing*, 1:475–492, 1994.

[52] T. Matsui. Approximation algorithms for maximum independent set problems and fractional coloring problems on unit disk graphs. *Discrete and Computational Geometry*, Springer Verlag, Lecture Notes in Computer Science, vol. 1763: 194–200, 2000.

[53] S. Gaspers, D. Kratsch, and M. Liedloff. On independent sets and bicliques in graphs. *Graph-Theoretic Concepts in Computer Science*, Springer Verlag, Lecture Notes in Computer Science, vol. 5344:171–182, 2008.

[54] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph. *Communications of the ACM*, 16:575–577, 1973.

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

[56] R. E. Miller and D. E. Muller. A problem of maximum consistent subsets. *IBM Research Report RC-240, J. T. Watson Research Center, Yorktown Heights, NY*, 1960.

[57] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363:28–42, 2006.

[58] D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real world graphs. In *Proceedings of the 10th Symposium on Experimental Algorithms, SEA 2011*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 6630, pages 364–375, 2011.

[59] D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *Proceedings of the 20th International Symposium on Algorithms and Computation, ISAAC 2010*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 6506, pages 403–414, 2010.

[60] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9(3):615–627, 1980.

[61] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[62] S. C. Eisenstat and J. W. H. Liu. The theory of elimination trees for sparse unsymmetric matrices. *SIAM Journal on Matrix Analysis and Applications*, 26(3): 686–705, 2005.

[63] METIS. Metis - serial graph partitioning and fill-reducing matrix ordering. See http://glaros.dtc.umn.edu/gkhome/views/metis/.

[64] H. L. Bodlaender, A. M. C. A. Koster, and F. V. D. Eijkhof. Preprocessing rules for triangulation of probabilistic networks. *Computer Intelligence*, 21(3):286–305, 2005.

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

[66] H. L. Bodlaender, A. M. C. A. Koster, and F. V. D. Eijkhof. Safe reduction rules for weighted treewidth. *Algorithmica*, 47(2):138–158, 2007.

[67] H. L. Bodlaender, B. M. P. Jansen, and S. Kratsch. Preprocessing for treewidth: A combinatorial analysis through kernelization. *SIAM journal of Computing*, 27 (4):2108–2142, 2013.

[68] H. L. Bodlaender. Kernelization : New upper and lower bound techniques. In *Proceedings of the 6th International Symposium on Parameterized and Exact Computation, IPEC'09*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 5971, pages 17–37, 2009.

[69] R. Downey and M. R. Fellows. Parameterized complexity. *Monographs in Computer Science. Springer, New York*, 1999.

[70] J. Flum and M. Grohe. Parameterized complexity theory. *Springer-Verlag New York*, 2006.

[71] H. L. Bodlaender. A partial k-arboretum of graphs with bounded tree-width. *Theoretical Computer Science*, 209(1–2):1–45, 1998.

[72] D. E. Knuth. *The Stanford Graph Base: A Platform for Combinatorial Computing*. Addison-Wesley, 1993.

[73] S. Arnborg and A. Proskurowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1):11–24, 1989.

[74] A. M. C. A. Koster. *Frequency Assignment - Models and Algorithms*. PhD thesis, University Maastricht, Maastricht, The Netherlands, 1999.

[75] A. M. C. A. Koster, S. P. M. van Hoesel, and A. W. J. Kolen. Solving partial constraint satisfaction problems with tree decomposition. *Networks*, 40(3): 170–180, 2002.

[76] J. Alber, F. Dorn, and R. Niedermeier. Experimental evaluation of a tree decomposition-based algorithm for vertex cover on planar graphs. *Discrete Applied Mathematics*, 145(2):219 – 231, 2005.

[77] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8: 277–284, 1987.

[78] F. V. Fomin, D. Kratsch, and I. Todinca. Exact (exponential) algorithms for treewidth and minimum fill-in. *Automata, Languages and Programming*, Springer Verlag, Lecture Notes in Computer Science, vol. 3142:568–580, 2004.

[79] V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, UAI '04, pages 201–208, 2004.

[80] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Manage Science*, 3:255–269, 1957.

[81] T. Kloks and H. L. Bodlaender. Only few graphs have bounded treewidth. Technical Report UU-CS-92-35, Department of Information and Computing Sciences, Utrecht University, Utrecht, the Netherlands, 1992.

[82] Ö. Johansson. Cique decomposition, nlc-decomposition and modular decomposition- relationships and results for random graphs. *Congressus Numerantium*, 132:39–60, 1998.

[83] C. Lee, J. Lee, and S. Oum. Rank-width of random graphs. *Journal of Graph Theory*, 57(3):239–244, 2008.

[84] B. M. Bui-Xuan, J. A. Telle, and M. Vatshelle. H-join decomposable graphs and algorithms with runtime single exponential in rankwidth. *Discrete Applied Mathematics*, 158(7):809–819, April 2010.

[85] J. M. Robson. Finding a maximum independent set in time $O(2^{n/4})$. Technical report, LaBRI, Université Bordeaux I, 2001.

[86] F. V. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: A simple $O(2^{0.288n})$ independent set algorithm. *SIAM Journal on Computing*, 36(2): 354–393, 2006.

[87] M. Fürer. A faster algorithm for finding maximum independent sets in sparse graphs. In *LATIN 2006: Theoretical Informatics*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 3887, pages 491–501. 2006.

[88] N. Sbihi. Algorithme de recherche dún stable de cardinalité maximum dans un graphe sans étoile. *Discrete Mathematics*, 29:53–76, 1980.

[89] S. B. Baker. Approximation algorithms for np-complete problems on planar graphs. *Journal of the ACM*, 41(1):153–180, 1994.

[90] M. Grohe. Local tree-width, excluded minors, and approximation algorithms. *Combinatorica*, 23:613–632, 2003.

[91] E. Tomita and T. Seki. An efficient branch-and-bound algorithm for finding a maximum clique. *Theoretical Computer Science*, Springer Verlag, Lecture Notes in Computer Science, vol. 2731:278–289, 2003.

[92] E. Tomita and T. Kameda. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global Optimization*, 37:95–111, 2007.

[93] E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, and M. Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *Proceedings of the WALCOM : Algorithms and Computation, 2010*, volume Springer Verlag, Lecture Notes in Computer Science, vol. 5942, pages 191–203, 2010.

[94] P. Prosser. Exact algorithms for maximum clique: A computational study. *Algorithms*, 5(4):545–587, 2012.

[95] B. Pattabiraman, M. M. A. Patwary, A. H. Gebremedhin, W. Liao, and A. N. Choudhary. Fast algorithms for the maximum clique problem on massive sparse graphs. In *Proceedings of the 10th International Workshop on Algorithms and Models for the Web Graph, WAW 2013*, pages 156–169, 2013.

[96] C. McCreesh and P. Prosser. Multi-threading a state-of-the-art maximum clique algorithm. *Algorithms*, 6(4):618–635, 2013.

[97] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, 2011.

[98] F. N. Abu-Khzam, R. L. Collins, M. R. Fellows, M. A. Langston, W. H. Suters, and C. T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experimentation and the 1st Workshop on Analytic Algorithmics and Combinatorics, ALENEX/ANALCO 2004*, pages 62–69, 2004.