FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

SAT Approach for Decomposition Methods

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktorin der Technischen Wissenschaften

by

M.Sc. Neha Lodha
Registration Number 01428755

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Stefan Szeider
Second advisor: Prof. Armin Biere

The dissertation has been reviewed by:

Daniel Le Berre   Marijn J. H. Heule

Vienna, 31st October, 2018   Neha Lodha

Erklärung zur Verfassung der Arbeit

M.Sc. Neha Lodha
Favoritenstrasse 9, 1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 31. Oktober 2018

Neha Lodha

Acknowledgements

First and foremost, I would like to thank my advisor Prof. Stefan Szeider for guiding me through my journey as a Ph.D. student with his patience, constant motivation, and immense knowledge. His guidance helped me during the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor.

Next, I would like to thank my co-advisor Prof. Armin Biere, my reviewers, Prof. Daniel Le Berre and Dr. Marijn Heule, and my Ph.D. committee, Prof. Reinhard Pichler and Prof. Florian Zuleger, for their insightful comments, encouragement, and patience. I am very thankful to Sebastian Ordyniak, who is my unofficial co-advisor, one of the closest collaborators, and somehow around him, all the bugs were automatically resolved. I am sincerely thankful to Prof. Matti Järvisalo and Prof. Torsten Schaub, who provided me with an opportunity to visit them and let me be a part of their research group.

I gained crucial knowledge by collaborating with my co-authors, Simone Bova, Johannes Fichte, Robert Ganian, Ronald de Haan, and Markus Hecher. I am very fortunate to be a part of the doctoral program "LogiCS" at TU Wien funded by FWF, where I got to know world-class researchers from all around the world. I enjoyed the numerous fruitful discussions I have had with my peers; especially Eduard Eiben, Anna Lukina, Katalin Fazekas, Matthias Schlaipfer, Zeynep Gözen Saribatur, Fabian Klute, Tomáš Peitl, and Friedrich Slivovsky. Not to forget the invaluable administrative support I received from Anna Prianichnikova and our secretaries, Juliane Auerböck, Beatrix Bhul, Doris Dicklberger, Eva Nedoma, Karin Prater, and Ulrike Prohaska.

I would like to express my gratitude towards Prof. Saket Saurabh, who recommended me to apply for this position and Prof. R. Ramanujan for encouraging me to venture out in the world. Last but not least, I would like to thank my family, my flatmates, and my friends for supporting me and providing me with a social life.

I have had the great opportunity to be around a very active and collaborative environment during my Ph.D., I could not have wished for anything better.

Kurzfassung

Wir suchen immer nach dem besten Weg ein Berechnungsproblem zu lösen. Jedoch kann der naheliegendste Weg, um ein gegebenes Problem zu lösen, manchmal nicht sehr praktisch sein. InformatikerInnen haben versucht praktische Wege zu finden, um schwere Probleme zu lösen. Eine der Haupttechnologien, die entwickelt wurden, um schwere Probleme zu lösen, sind strukturelle Zerlegungen. Diese ermöglichen eine gegebene Probleminstanz in kleine Teile zu zerlegen. Es gibt verschiedene Techniken die Zerlegungen zu benutzen, um ein Problem zu lösen, aber diese Techniken sind stark auf die Qualität der Zerlegung angewiesen. In dieser Arbeit entwickeln wir SAT-basierte Techniken, um „gute“ Zerlegungen zu finden.

Satisfiablity (SAT) – auf Deutsch das Erfüllbarkeitsproblem der Aussagenlogik – ist eines der zentralsten Probleme in der Informatik. SAT-solver, das sind Werkzeuge, die dieses Problem lösen können, haben sich in den letzten Jahren drastisch verbessert. Aktuelle SAT-solver können einige mehrere MB große SAT-Instanzen innerhalb von Millisekunden lösen. Ein SAT-encoding ist die Transformation eines Berechnungsproblem nach SAT. Wir nutzen die Geschwindigkeit und Effizienz von SAT-basierten Techniken, um „gute“ Zerlegungen zu finden.

In dieser Arbeit entwickeln wir SAT-encodings, um gute Zerlegungen zu finden. Eine der größten Hürden für SAT-encodings ist die Größe der SAT-Formel. Die meisten SAT-encodings sind zumindest kubisch in der Größe der Probleminstanz. Dies führt zu einem harten Limit für die Größe der zu lösenden Probleminstanz. Wir überwinden diese Restriktion indem wir eine neue SAT-basierte lokale Verbesserungstechnik entwickeln, in der wir versuchen, Teile einer gegebenen Zerlegungen mithilfe von SAT-encodings zu verbessern.

Wir schlagen eine neue Charakterisierung für Branch-Zerlegungen vor und benutzen diese, um ein SAT-encoding für Branchwidth zu finden. Wir entwickeln einen lokalen Verbesserungsansatz für die Ermittlung besserer Branch-Zerlegungen, welche wir später auch für Baumzerlegungen benutzen. Wir entwickeln zwei verschiedene Charakterisierungen für spezielle Baumzerlegungen und Pfadzerlegungen und vergleichen diese empirisch. Zudem entwickeln wir auch ein SMT-encoding für die Ermittlung von fraktionalen Hyperbaum-Zerlegungen.

Abstract

We are always trying to find the best way to solve any computational problem. But, sometimes the most natural way to solve a given problem may not be very practical. Computer scientists have been trying to find practical ways for solving hard problems. One of the major technology, that was developed to solve hard problems, is decompositions. Decompositions allow one to decompose a given problem instance in small parts. There are various techniques that use decompositions to solve a problem, but these techniques rely heavily on the quality of the decompositions. In this thesis, we develop SAT-based techniques to find "good" decompositions.

Satisfiability (SAT) is one of the most central problem in computer science and SAT-solvers, tools that can solve this problem, have improved drastically over the last years. Current SAT-solvers can solve some SAT instances, of several MB in size, in a matter of milliseconds. A SAT-encoding is the transformation of a computational problem to SAT. We make use of the speed and efficiency of the SAT based techniques for finding good decompositions.

In this thesis we develop SAT-encodings for finding good decompositions. One of the major hurdle for SAT-encodings is the size of the SAT formula. Most of the SAT-encodings are at least cubic is the size of problem instance, which puts a hard limit on the size of the problem instance that can be solved. We overcome this restriction by developing a new SAT-based local improvement technique, where given a decomposition we try to improve parts of it using SAT-encodings.

We propose a new characterization for branch decompositions and used it for a SAT-encoding for finding branchwidth. We develop local improvement approach for finding better branch decompositions, which we later used for tree decompositions as well. We develop two different characterizations for special tree decomposition and path decomposition, and compare them empirically. We also develop an SMT-encoding for finding fractional hypertree decompositions.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**
1.1 Decompositions 1
1.2 SAT-Solvers 3
1.3 Contribution 4
1.4 Organization 5
1.5 Software 6

2 Preliminaries **9**
2.1 Satisfiability 9
2.2 Partitions 11
2.3 Graphs 11
2.4 Tree Decompositions 14
2.5 Clique-width Decomposition 18
2.6 Decompositions Addressed in this Thesis 21
2.7 Experimental Setup and Benchmarks 22

3 Overview of Results **25**
3.1 Methodologies 25
3.2 Results 27

4 Branchwidth and Carving-Width **31**
4.1 Introduction 31
4.2 Preliminaries 33
4.3 Tree Encoding for Branchwidth 35
4.4 Partition-based Reformulation of Branchwidth 38
4.5 Carving-Width 43
4.6 Local Improvement for Branch Decompositions 46
4.7 Local Improvement for Carving Decompositions 52

4.8 Experimental Results 52
4.9 Chapter Summary 58

5 Treewidth 59
5.1 Introduction 59
5.2 Local Improvement of Tree Decompositions 61
5.3 Experimental Results 64
5.4 Chapter Summary 66

6 Special Treewidth and Pathwidth 69
6.1 Motivation 69
6.2 Preliminaries 72
6.3 Preprocessing 75
6.4 Partition-Based Approach for Special Treewidth 86
6.5 Ordering-Based Approach for Special Treewidth 91
6.6 SAT-Encodings for Pathwidth 96
6.7 Experiments 99
6.8 Chapter Summary 109

7 Fractional Hypertree Width 111
7.1 Introduction 111
7.2 Preliminaries 113
7.3 Ordering-Based Characterization of Fractional Hypertree Width . . . 114
7.4 SMT-encoding for Fractional Hypertree Decomposition 117
7.5 Preprocessing 119
7.6 Symmetry Breaking and Lower Bounds with Cliques 122
7.7 Experimental Work 123
7.8 Chapter Summary 128

8 Conclusion and Future Work 131

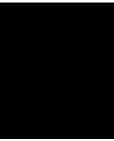
List of Figures 133

List of Tables 135

List of Algorithms 137

Bibliography 139

CHAPTER 1



Introduction

Computer science has become an essential part of almost every technology; querying a database, verifying software and hardware, scheduling processes or events are a few to name. As it turns out, many of these problems are not easy to solve. In case of some of these problems, we expect that one can only verify a given solution efficiently. Fortunately, there exists a silver lining for this predicament. Namely, in some special cases, it is not necessary to solve the entire problem instance. Instead, we can find a solution by first splitting the problem instance in parts, solving these and later combining the partial solutions. The splitting of a problem instance in parts is called "decomposition" and there exist various techniques to achieve this. Each decomposition method has some kind of "width measure" associated with it, which represents the quality of a decomposition. Unfortunately, finding a good decomposition is itself a hard problem and some problem instances do not admit decompositions of small width. In this work, we develop methods on how to efficiently decompose a problem instance, if it can be decomposed.

1.1 Decompositions

To solve any computational problem one relies on a good algorithm. The efficiency of an algorithm is usually measured in terms of its running time, which is the time it takes to solve the given problem and terminate correctly. It is a common practice to estimate the running time based on the size of the input. Numerous problems that arise in computer science are very hard or even impossible to solve. Most of these hard problems have algorithms whose running time is usually exponential in the input size, which makes it hard or even impossible to solve these problems in practice.

Problem decompositions allow one to split a given problem instance in parts and they play a central role in the study of hard problems. Using decompositions allows one to use algorithmic techniques such as *dynamic programming* [KGOD11]. Dynamic programming is a technique where parts of a problem instance are solved recursively

to construct a solution for the entire instance. Decompositions provide an easy and efficient way of obtaining the sub-division required for dynamic programming. Some of the most well known decomposition techniques are tree decomposition [RS83], branch decomposition [RS91], path decomposition [RS83], and hypertree decomposition [GM06].

Computationally hard problems, like the Traveling Salesman Problem or the Boolean Satisfiability Problem, can be solved by a dynamic programming algorithm, if the problem instance has a "good" decomposition. The dynamic programming approach does not just provide efficient algorithms for solving computationally hard problems but also allows us to count their number of solutions, like counting the number of solutions for a Boolean Satisfiability Problem [BDP03]. Observe that counting the number of solutions for a computationally hard problem is even harder than just deciding whether at least one solution exists. These algorithms have a running time that is polynomial in the input but exponential in the width of a decomposition. As one can imagine, for a given problem instance there is a large number of possible decompositions usually exponential in the input size. Each of the decomposition technique has a width measure associated with it. The width of a decomposition technique is the smallest width over all possible decompositions that can be constructed with it and this width is also called the optimal width. Treewidth is the associated measure for tree decompositions, branchwidth for branch decompositions, and so on. In most cases, we additionally consider a decomposition to be good if it has the lowest width found by heuristic methods, which need not be optimal.

In general when using decompositions for solving a problem instance, one decomposes the underlying structure of the instance. This underlying structure is usually represented as a graph or a hypergraph. A graph consists of vertices and edges, where a (un)ordered pair of vertices form an edge. Similarly, a hypergraph consists of vertices and hyperedges, where a hyperedge is a subset of the vertex set. For example, a map can be considered as a graph where each city is a vertex and the roads connecting two cities is an edge.

Given the large number of possible decompositions, it is a hard problem to find a good or optimal decomposition. It is usually NP-hard to compute an optimal decomposition. One would be tempted to think that it is not possible to find optimal decompositions easily, but using combinatorial algorithms one can find optimal decompositions efficiently for some small graphs. There exist various techniques for finding optimal and non optimal decompositions. *Heuristics* are very fast techniques that can compute upper bounds on various widths for very large graphs, but in most cases they do not provide any measure on how close or far the resulting width is from the optimal width. *Approximations* are techniques which can be used for approximating a given width up to a certain factor of optimal width, this factor can be additive or multiplicative.

Most of the techniques for finding decompositions rely heavily on preprocessing and symmetry breaking techniques. Symmetry breaking is a technique where additional constraints are added which admit only one out of several symmetric solutions, such as fixing the order between two vertices which are adjacent to exactly the same set of vertices. Whereas, preprocessing techniques allow one to reduce the problem instance

to a smaller instance such that the optimal width is either preserved or changed by a constant factor.

It would be an obvious question to ask, why it is important to find the optimal width, when finding upper bounds can be much faster. The reason to aim at small width originate from the applications of the decompositions, e.g., the *dynamic programming.* As we already know decompositions are used for solving computationally hard problems efficiently and these algorithms often have exponential dependency on the width associated with the decomposition. Improving the width can allow some problems to be solved by using dynamic programming, as it was noted by Dechter et al. [KGOD11] in the context of treewidth, where the following was noted about inference on probabilistic networks of bounded treewidth:

> *[. . .] since inference is exponential in the tree-width, a small reduction in tree-width (say by even by* 1 *or* 2*) can amount to one or two orders of magnitude reduction in inference time.*

1.2 SAT-Solvers

The Boolean or Propositional Satisfiability Problem (SAT) is one of central problems studied in computer science, standing at the crossroad between logic, graph theory, computer science, computer engineering and operational research [FM09]. Some of the combinatorial problems that are computationally hard can be modeled in conjunctive normal form (CNF). CNF formulas are a special case of SAT formulas. SAT can be defined in numerous ways and in a nutshell, it is to find if a CNF formula is "satisfiable". Moreover, SAT-solvers, tools which can solve SAT instances fast and efficiently, have vastly improved over the last years. Current SAT-solvers can solve some large SAT instances (several hundreds of MBs in size) in a blink of an eye. The effectiveness of SAT-solvers and the versatility of the CNF formulas makes it an effective tool for solving many computationally hard problems.

Since many computationally hard problems can be modeled using SAT, the novelty of finding good encoding lies in the techniques used. To develop a good encoding, it is necessary to pay attention to the structure of the encoding and not just its size [Pre09]. Finding an encoding for a problem which can be solved efficiently by a SAT-solver is a challenging task, it requires one to understand in depths the working of SAT-solvers and the given problem.

The applications of SAT are not limited to solving computationally hard problems, in fact, SAT modulo theories (SMT) allows one to model problems in first order logic, using SAT as an oracle. The SMT solvers use SAT as a black box to solve various SMT instances. Using SMT solvers one can extend the reach of SAT and solve problems which involve real numbers.

Interestingly, one can use SAT-solvers for finding sub-optimal widths, i.e., lower bounds and upper bounds on the width. In fact, it is much faster to check the upper bounds than to check for optimal width [LOS16a]. This interesting observations allows us to use SAT-solvers for finding sub-optimal decompositions.

In this thesis, we focus on developing SAT based techniques for finding problem decompositions. We develop SAT-encodings for finding branch and carving decompositions, special tree decompositions, path decompositions, and an SMT encoding for finding fractional hypertree decompositions. We also use SAT techniques for improving upper bounds for heuristically obtained branchwidth and treewidth.

1.3 Contribution

Our main focus is on developing SAT-based techniques for the following decomposition techniques. In this thesis we demonstrate that SAT and its extensions can be used in a novel way to compute various widths. Moreover, these approaches have great potential to be used practically. In this section we give an overview on the decomposition methods that we focused on and the research methodology that we used to find good decompositions.

1.3.1 New SAT encodings

In order to encode a problem into SAT, we looked at various characterizations of the same decomposition method. Our prominent focus was on:

- ordering-based characterization, and
- partition-based characterization.

In an ordering based characterization one tries to find a specific ordering among either the vertices or the edges of the input graph. Similarly, for a partition based characterization, we arrange the vertices or the edges of the input graph in a specific sequence of partitions. Both of these characterizations induce decompositions for the input graph. It is often a challenging task to prove that these characterizations are equivalent, in the sense that they amount to the same decomposition width.

One of the first SAT-encodings for graph parameters was based on an ordering based characterization of tree decompositions [SV09]. We define three new SAT-encodings, one each for branch decompositions, special tree decompositions and path decompositions, and one SMT-encoding for fractional hyper tree decompositions.

The ordering based characterization was introduced by Heule and Szeider [HS15] for finding exact clique-width. We use a similar approach to a SAT-encoding for finding branch decompositions, special tree decompositions and path decompositions. For special treewidth and pathwidth, we empirically compare the performance of ordering-based and partition-based encodings.

1.3.2 SAT-based Local Improvement

One of the biggest obstacles for the use of SAT-based techniques is their poor scalability. In this thesis we make a successful attempt at overcoming this obstacle by using local improvement. We apply SAT-based local improvement to improve heuristically obtained decompositions. The idea behind is to use SAT-encodings for improving the problematic parts of the heuristic decomposition locally and then updating the same part in the original decomposition.

Our aim here is to use SAT for improving upper bounds rather than finding the optimal width. The motivation behind this idea is the fact that the general solving time is much lower than the solving time for checking optimality. This provides a potential use case for SAT-encodings for finding upper bounds and lower bounds. We utilize the same in local improvement.

1.3.3 Highlights

In this section, we would like to point out some of the highlights of this research work.

- One of the main contributions of this thesis is the SAT-based local improvement approach for improving decompositions which can be used and adapted with other technologies, such as, integer linear programming or combinatorial algorithms.
- We developed the first methods to find optimal special tree decompositions (using SAT- encodings) and optimal fractional hypertree decompositions (using SMT-encodings).
- We focused on comparing the different SAT-encoding techniques in terms of size of the encoding and the underlying techniques.
- We made all our tools and benchmark instances open source and publicly available.

1.4 Organization

Rest of this thesis is organized as follow. We continue with Chapter 2 where we provide the basic definitions related to graph theory, satisfiability theory and the input format. In this chapter we also provide a survey of related work. Chapter 3 presents an overview of our results. In Chapter 4 we lay down our techniques for finding optimal branch decompositions and carving decompositions. We also introduce the local improvement technique for improving branch decompositions in the same chapter, which we later use in Chapter 5 to improve tree decompositions. Chapter 6 includes our study of two fundamental characterization techniques namely ordering-based characterizations and partition-based characterizations. In Chapter 7, we expand our techniques for finding optimal fractional hypertree decomposition by using an SMT approach. We end with

a brief chapter including conclusion and future work. The Chapters 4, 5, 6, and 7 are based on following papers, respectively:

- **Chapter 4:** Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. A SAT Approach to Branchwidth. In Nadia Creignou and Daniel Le Berre, editors, *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing, SAT 2016*, volume 9710 of *Lecture Notes in Computer Science*, pages 179–195. Springer Verlag, 2016 (full version conditionally accepted at the journal Transactions on Computational Logic (TOCL)).
- **Chapter 5:** Johannes Klaus Fichte, Neha Lodha, and Stefan Szeider. SAT-Based Local Improvement for Finding Tree Decompositions of Small Width. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 401–411. Springer, 2017
- **Chapter 6:** Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. SAT-encodings for special treewidth and pathwidth. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 429–445. Springer, 2017 (full version accepted at the Journal of Artificial Intelligence Research (JAIR)).
- **Chapter 7:** Johannes Klaus Fichte, Markus Hecher, Neha Lodha, and Stefan Szeider. An SMT approach to fractional hypertree width. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2018

1.5 Software

During the course of this thesis we not only published numerous publications but also developed software to test the theoretical results in practice. In the following we provide a brief description of software produced for the same purpose:

- **BranchLIS** [LOS16b] is a software that can compute optimal branch decompositions using SAT-solvers, can improve heuristically obtained branch decompositions using our SAT-based local improvement technique, and verify a given branch decomposition. This software supports all standard SAT-solvers.
- **trellis** [FLS17c] is a software that used our SAT-based local improvement technique to improve heuristically obtained tree decompositions. This tool supports all PACE competition [DHJ+17, DKTW18] solver to find tree decompositions.

- **FraSMT** [FHLS18a] is a Z3 SMT solver based software that can compute and verify fractional hypertree decompositions.

CHAPTER 2

Preliminaries

In this chapter we will introduce the basic terminology used across this thesis. We start with the basics of propositional satisfiability which forms the base of our research. Next we define some common concepts in graph theory. At the end of this chapter we introduce the basic input and output format (DIMACS format).

2.1 Satisfiability

A *propositional or boolean formula* consists of literals and the operators conjunction ($\wedge$), disjunction ($\vee$) or negation ($\neg$). The literals are boolean variables or negated boolean variables which can take values true (1) or false (0). We say that a formula is satisfiable if there exists an assignments of the variables of the formula to true or false, such that the formula evaluates to true.

A clause is a disjunction of literals. A propositional formula is said to be in conjunctive normal form (CNF) if it is a conjunction of clauses. We mostly focus on CNF formulas. The following is an example of a CNF formula:

Example 1. $F = (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z)$.

It is easy to see that formula F is satisfiable. One of the satisfying assignments of its variables is $x = true$, $y = true$, and $z = true$. A formula F can have multiple satisfying assignments. A satisfying assignment of the variables of a formula F is also called as a *model* of the formula F.

Definition 1. *The* propositional *or* boolean satisfiability (SAT) *problem asks whether a given CNF formula F is satisfiable.*

2.1.1 Counting

One of the major parts of our encodings is to restrict the number of variables set to true for a given set of variables. Cardinality counters are propositional CNF formulas that are used for this purpose. More precisely, for a given a set of variables $V = \{1, \dots, n\}$ and an integer ω, a cardinality counter is a formula, whose variables include V, and where all its satisfying assignments assign at most ω of the variables in V to true. In this work, we mainly focused on the sequential counter described by Sinz [Sin05] which has previously been used in SAT-encoding for tree decomposition [SV09]. This encoding is considered to be quite robust, thus ensures consistent performance for our experiments.

To illustrate the idea behind the sequential counter, consider the case given in Table 2.1. We have a set $V = \{1, \dots, 6\}$ of boolean variables and we want to set at most 4 of them to true. The sequential counter sorts the assignment for these variables such that the first bits are 1s and for the last variable checks if there are at most 4 variables set to 1.

v	*value*	j			
		1	2	3	4
1	0	0	0	0	0
2	1	1	0	0	0
3	1	1	1	0	0
4	0	1	1	0	0
5	1	1	1	1	0
6	0	1	1	1	0

Table 2.1: An illustration of the behavior of the sequential counter, which counts at most 4, counting the number of variables from the set $V = \{1, \dots, 6\}$ set to true. The last four columns in the table provide the intermediate values of the sequential counter.

To implement this idea let us consider a set $V = \{1, \dots, n\}$ of variables and we want to set at most k of them to true. We introduce variable $\text{ctr}(i, j)$ which is true if from the set $\{1, \dots, i-1\}$ there are exactly j variables set to true. We start by adding the following clauses to ensure that for each variable i if it is set to true then the variable $\text{ctr}(i, 1)$ is also set to true.

$$\neg i \vee \text{ctr}(i, 1) \qquad \text{for all } 1 \leq i \leq n.$$

Next we add the clauses to ensure the propagation of the value of $\text{ctr}(i, j)$ to $\text{ctr}(i+1, j)$

$$\neg\text{ctr}(i, j) \vee \text{ctr}(i+1, j) \qquad \text{for all } 1 \leq i \leq n-1 \text{ and } 1 \leq j \leq k.$$

Next we add the clauses to ensure that if the variables i and $\text{ctr}(i-1, j-1)$ are set to true, i.e., the variable i is set to true and there are $j-1$ variables already set to true before $i-1$ then $\text{ctr}(i, j)$ is set to true, i.e., there are j many variables before i are set to true.

$$\neg i \vee \neg\text{ctr}(i-1, j-1) \vee \text{ctr}(i, j) \qquad \text{for all } 2 \leq i \leq n \text{ and } 2 \leq j \leq k.$$

Lastly we add the following clauses to ensure that there are not more than k variables set to true before i, i.e., it is not the case that both i and $\text{ctr}(i-1, k)$ variables are set to true.

$$\neg i \vee \neg\text{ctr}(i-1, k) \qquad \text{for all } 2 \leq i \leq n.$$

This completes the construction of the sequential counter. We added $\mathcal{O}(nk)$ variables and $\mathcal{O}(nk)$ clauses. In all our SAT-encodings we use sequential cardinality counters to restrict the width of the decompositions.

2.2 Partitions

As (weak) partitions play an important role in our reformulation of width parameters, we state some basic terminologies here.

Definition 2. *A **weak partition** of a set S is a set P of nonempty subsets of S such that any two sets in P are disjoint. Additionally if the union of all the sets in P equals S then the set P is said to be a **partition** of the set S.*

The elements of P are called *equivalence classes*. We denote by $\text{U}(P)$ the union of all sets in P. Let P, P' be weak partitions of S, then P' is a *refinement* of P if $\text{U}(P) \subseteq \text{U}(P')$ and any two elements $x, y \in S$ that are in the same equivalence class of P' are not in distinct equivalence classes of P (this entails the case $P = P'$). Moreover, we say that P' is a *k-ary refinement* of P if additionally it holds that for every $p \in P$ there are $p_1, \ldots, p_k$ in P' such that $p \subseteq \bigcup_{1 \leq i \leq k} p_i$.

Example 2. *An example of partition and weak partition for a set S:*

set: $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$
partition: $P = \Big\{\{1, 2, 3\}, \{4, 7\}, \{5, 6, 8\}\Big\}$
weak partition: $P_{weak} = \Big\{\{1, 2, 3\}, \{4\}, \{8\}\Big\}$
2*-ary refinement of S:* $P_{2-ary} = \Big\{\{1, 2, 3, 4, 5\}, \{6, 7, 8\}\Big\}$

2.3 Graphs

We mostly consider finite and undirected (hyper)graphs. For basic terminology on graphs we refer to a standard text book [Die00]. For a (hyper) graph G we denote by $V(G)$ the vertex set of G and by $E(G)$ the edge set of G. If the (hyper)graph G is clear from context we use V and E for $V(G)$ and $E(G)$, respectively. An edge is a set of two vertices where a hyperedge is any subset of vertices. In general we assume that the vertices are

numbered from 1 to n and edges numbered from 1 to m where $n = |V|$ and $m = |E|$. We denote an edge e between two vertices u and v by $\{u, v\}$ or uv. For a vertex v, the *degree* of v is the number of edges e such that $v \in e$. We denote by $\Delta(G)$ the maximum degree over all vertices of G, i.e., the maximum number of edges containing a particular vertex v of G. For a *directed graph* $D = (V, E)$, a directed edge $e \in E$ is an ordered pair (u, v) such that $u, v \in V$. We usually refer to a directed edge as an *arc*. The *in degree* of a vertex v in D is the number of incoming arcs incident on v, similarly the *out degree* is the number of outgoing arcs incident on v.

We say the graph G is a simple undirected graph if the graph G does not contain any self loops, multi-edges or directed edges. A *path* P of length k is a sequence of edges $e_1, \ldots, e_k$ such that e_i and $e_i + 1$ are incident on the same vertex v_i, where $1 \leq i \leq k - 1$. Moreover, if e_1 is incident to v_0 and e_k is incident to v_k we say that P connects v_0 and v_k. The *radius* of G, denoted by $\text{rad}(G)$, is the smallest integer r such that G has a vertex from which all other vertices are reachable via a path of length at most $\text{rad}(G)$. The *center* of G is the set of vertices v such that all other vertices of G can be reached from v via a path of length at most $\text{rad}(G)$.

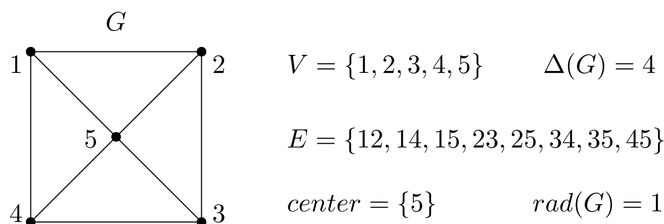


Figure 2.1: A simple undirected graph G with 5 vertices and 8 edges. The max degree of G is 5. The center of G is 5 and its radius is 1

A *cycle* is a path which starts and finishes at the same vertex. In the example above the sequence $12, 23, 35$ forms a path and the sequence $12, 25, 15$ forms a cycle. A graph is said to be *cyclic* if a cycle is present in the graph. Similarly, a graph is *acyclic* if it contains no cycles.

We will often consider various forms of *trees*, i.e., connected acyclic graphs, as they form the backbone of all of the decompositions that we study in this work. Let T be an undirected tree. We will always assume that T is rooted (in some arbitrary vertex r) and hence the parent and child relationships between its vertices are well defined. A *leaf* of a tree T is a vertex of degree one, i.e., it has exactly one neighbor. We say that T is *ternary* if every non-leaf vertex of T has degree exactly three. We will write $\text{p}_T(t)$ (or just $\text{p}(t)$ if T is clear from the context) to denote the *parent* of $t \in V(T)$ in T. We also write T_t to denote the *subtree* of T rooted in t, i.e., the component of $T \setminus \{t\text{p}_T(t)\}$ containing t. For a tree T, we denote by $\text{h}(T)$, the *height* of T, i.e., the length of a longest path between the root and any leaf of T plus one. It is well-known that every tree has at most two center vertices, moreover, if it has two center vertices then they form the endpoints of an edge in the tree. The Figure 2.2 shows two trees T and T' where one of them is a

ternary tree. The leaves in the tree T' are 5, 6, 7, 8, 9, and 10. The center of T is $\{1, 2\}$, where as the center of T' is $\{1\}$.

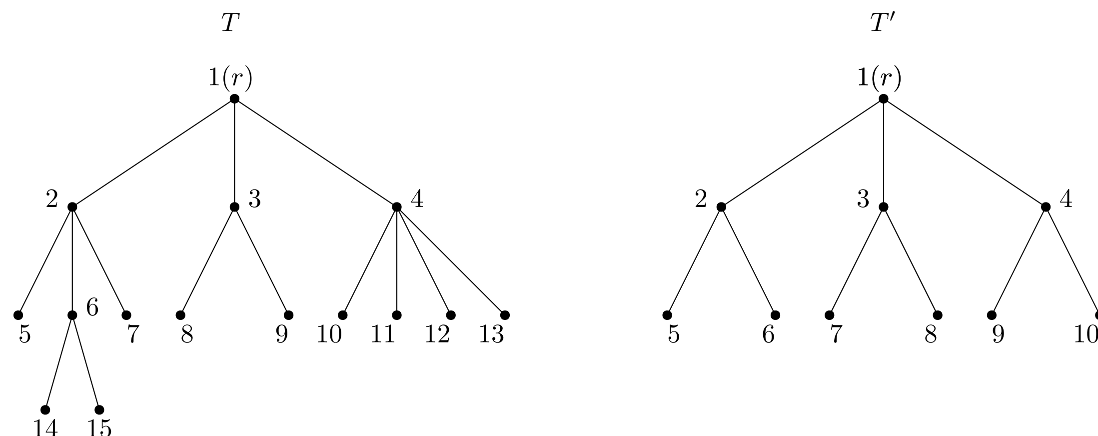


Figure 2.2: Two trees T and T' rooted at vertex 1. The tree T' is a ternary tree of height 3. The tree T is of height 4.

A *complete graph* or *clique* (K_n) is a graph with n vertices with an edge between each pair of vertices. A *bipartite graph* is a graph where we can partition its vertices into two sets such that all the edges in the graph are between vertices belonging to different sets. A *complete bipartite graph* ($K_{m,n}$) is a bipartite graph, where the two sets of the partition have m and n vertices, respectively, with all possible edges present in the graph. A *grid graph* is a graph which can be drawn as a grid. Figure 2.3 shows a K_5, $K_{3,2}$, and a $4 \times 5 - grid$.

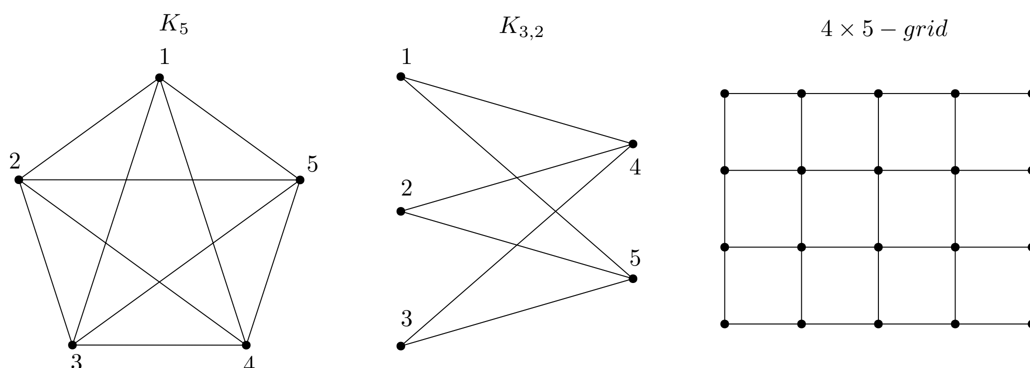


Figure 2.3: A complete graph K_5, a complete bipartite graph $K_{3,2}$ and a $4 \times 5 - grid$.

2.4 Tree Decompositions

Tree decomposition is one of the most well known decomposition methods used to decompose a graph. The treewidth of a graph measures how tree like a given graph is. In this section we will give the basic definition of tree decomposition and treewidth.

Definition 3. *A* tree decomposition $\mathcal{T}$ *of a graph* $G = (V, E)$ *is a pair* (T, χ)*, where* T *is a tree and* χ *is a function that assigns each tree node* t *a set* $\chi(t) \subseteq V$ *of vertices such that the following conditions hold:*

(T1) *For every vertex* $u \in V$*, there is a tree node* t *such that* $u \in \chi(t)$*.*

(T2) *For every edge* $\{u, v\} \in E$ *there is a tree node* t *such that* $\{u, v\} \subseteq \chi(t)$*.*

(T3) *For every vertex* $v \in V$*, the set of tree nodes* t *with* $v \in \chi(t)$ *forms a subtree of* T*.*

The sets $\chi(t)$ *for any* $t \in V(T)$ *are called* bags *of the decomposition* $\mathcal{T}$ *and* $\chi(t)$ *is the bag associated with the tree node* t*. The* width *of a tree decomposition* (T, χ) *is the size of a largest bag minus* 1*. A tree decomposition of minimum width is called* optimal*. The* treewidth *of a graph* G *is the width of an optimal tree decomposition of* G

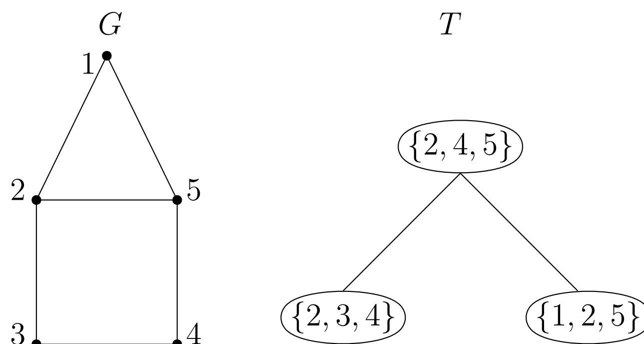


Figure 2.4: A graph G (left) and an optimal tree decomposition $\mathcal{T} = (T, \chi)$ of G (right).

The following is a well-known fact about tree decompositions.

Fact 1 ([Klo94])**.** *Let* (T, χ) *be a tree decomposition of a graph* G *and* K *a clique in* G*. Then there exists at least one node* $t \in V(T)$ *such that* $V(K) \subseteq \chi(t)$*.*

2.4.1 Elimination Orderings

One of the first SAT-encodings for graph decompositions was developed for finding tree decompositions. This encoding was developed by Samer and Veith [SV09] and was based on an alternative characterization of tree decompositions known as elimination orderings.

In this section we will explain the characterization of treewidth based on the so called elimination orderings.

Given an input graph $G = (V, E)$, an elimination ordering is a linear ordering of the vertices of the input graph G. Using this linear ordering one can construct the fill-in graphs of the input graph by deleting one vertex from the graph at a time and making all it neighbors adjacent to each other (see Figure 2.5). The width of a linear ordering is the size of the largest clique in any of the fill-in graphs minus 1. The treewidth of the input graph G equals the minimum width over all possible linear orderings. It is well known that given any tree decomposition one can construct a linear ordering and vice versa. In fact even for an optimal tree decomposition one can construct linear ordering such that the optimality is conserved (see [Bod05] and [Dec06]).

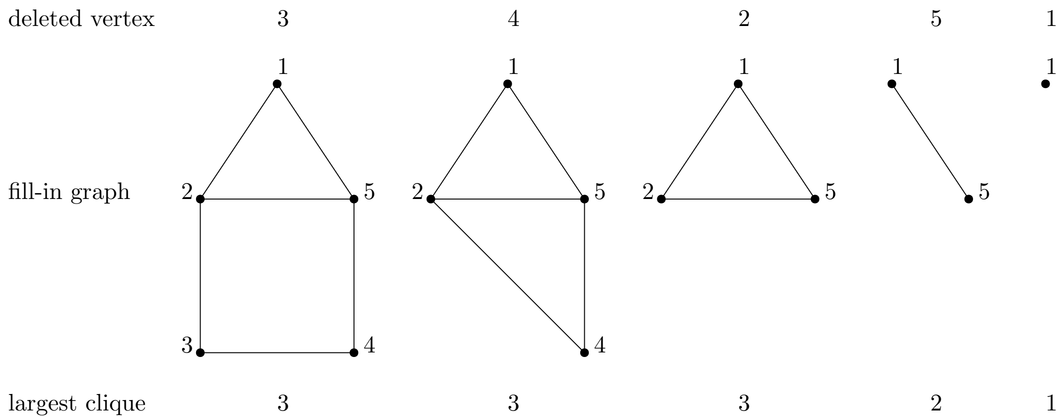


Figure 2.5: The construction of fill-in graphs for the graph G in the Figure 2.4 based on the elimination ordering $L = (3, 4, 2, 5, 1)$. The largest cliques associated with the fill-in graph

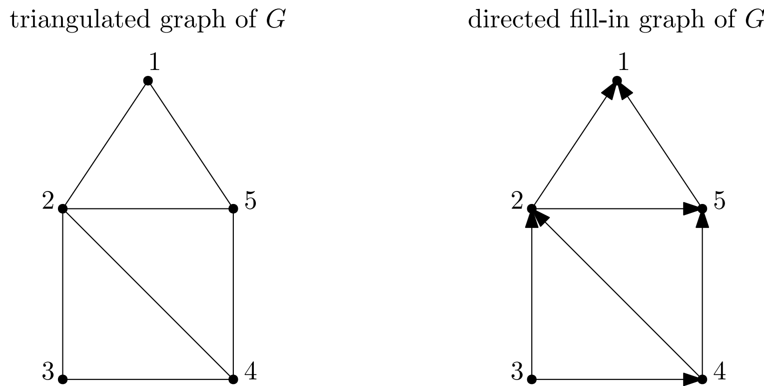


Figure 2.6: The undirected and the directed fill-in graph for the graph G in the Figure 2.4 associated with the elimination ordering $L = (3, 4, 2, 5, 1)$.

For the graph G in Figure 2.4 one of the linear orderings is $L = (3, 4, 2, 5, 1)$. The Figure 2.5 shows the construction of the fill-in graphs associated with this elimination ordering. To complete the construction, Figure 2.6 shows the fill-in graph of the same graph G associated with the elimination ordering L. The fill-in graph can also be constructed as a directed graph where each arc between two vertices is directed from the vertex that is eliminated earlier to the vertex that is eliminated later. Figure 2.6 also shows this directed fill-in graph. Note that the directed fill-in graph is acyclic, i.e., there are no induced directed cycles. The treewidth of such a directed fill-in graph is the maximum number of outgoing arcs from any vertex in the graph.

2.4.2 SAT-Encoding

The standard SAT-encoding for finding optimal tree decompositions is based on elimination orderings. To start off, we first formalize some more concepts regarding the elimination orderings. For a given graph $G = (V, E)$ and a linear ordering L of the vertices V, we say that a vertex u is a predecessor of v (denoted by $u \leq_L v$) if $uv \in E$ and u is eliminated before v. We also assume that the vertices and the edges of the graph G are numbered from 1 to n and 1 to m respectively. Given an integer tw, to check if the graph G has treewidth tw we need to construct a formula $F(G, \mathrm{tw})$, which is satisfiable if and only if G has treewidth tw. In order to do so we first need to construct a formula $F(G)$ which is satisfiable if the graph G has an elimination ordering. The formula $F(G)$ has order variables $\mathrm{o}(u, v)$ which are set to true if u is a predecessor of v in the ordering L and $u < v$. For simplicity the following macro is used:

$$\mathrm{o}^*(u, v) = \begin{cases} \mathrm{o}(u, v) & \text{if } u < v; \\ \neg\mathrm{o}(v, u) & \text{otherwise.} \end{cases}$$

Now we are ready to construct the formula $F(G)$. To ensure that the linear ordering L is transitive the following clauses are added:

$$\neg\mathrm{o}^*(u, v) \vee \neg\mathrm{o}^*(v, w) \vee \mathrm{o}^*(u, w) \qquad \text{for all } 0 < u < v < w < n.$$

The above construction enforces that there is some linear ordering among the vertices of the graph G. Next, to construct the directed fill-in graph, we describe the arcs of this graph. Let $G_T = (V, E(G_T))$ be the directed fill-in graph of G associated with the linear ordering generated by the order variables. Let the variable $\mathrm{a}(u, v)$ denote the arcs in the fill-in graphs for all $u, v \in V$, such that there is an arc from the vertex u to the vertex v. The variable $\mathrm{a}(u, v)$ is set to true if $uv \in E(G_T)$.

Now we are ready to describe the clauses that will define the arcs of the directed fill-in graph G_T. To ensure that the edges present in the original graph are also present in the graph G_T the following clauses are added:

$$\begin{aligned} &\neg\mathrm{o}(u, v) \vee \neg\mathrm{a}(u, v) \\ &\mathrm{o}(u, v) \vee \neg\mathrm{a}(u, v) \qquad \text{for all } 1 \leq u < v \leq n \text{ and } uv \in E. \end{aligned}$$

To ensure that the fill-in edges induced by order variables are present in the directed fill-in graph G_T, the following clauses are added:

$$\neg\mathrm{a}(u,v) \vee \neg\mathrm{a}(u,w) \vee \neg\mathrm{o}(v,w) \vee \mathrm{a}(v,w)$$
$$\neg\mathrm{a}(u,v) \vee \neg\mathrm{a}(u,w) \vee \mathrm{o}(v,w) \vee \mathrm{a}(w,v)$$
$$\text{for all } 1 \le u,v,w \le n,\ u \ne v,\ u \ne w, \text{ and } v < w.$$

To ensure that no self loops are generated in the above construction the following clauses are added:

$$\neg\mathrm{a}(u,u) \qquad \text{for all } 1 \le u \le n.$$

The following redundant clauses are added:

$$\neg\mathrm{a}(u,v) \vee \neg\mathrm{a}(u,w) \vee \mathrm{a}(v,w) \vee \mathrm{a}(w,v)$$
$$\text{for all } 1 \le u,v,w \le n,\ u \ne v,\ u \ne w, \text{ and } v < w.$$

This completes the construction of $F(G)$ which is satisfiable if and only if the graph G has a linear ordering and a directed fill-in graph G_T, which is acyclic. Next we want to extend $F(G)$ to $F(G,\mathrm{tw})$. As we already know from the previous section that the width of a linear ordering of vertices of G is equal to the maximum number of outgoing arcs incident on any vertex of G. To bound the number of outgoing arcs of a vertex u we want to restrict the number of arc variables $\mathrm{a}(u,v)$ for all $v \in V$, set to true, to tw. we use the sequential cardinality counters described in Section 2.1.1 for this purpose. After adding the clauses for counting we finish the construction of $F(G,\mathrm{tw})$. The formula $F(G,\mathrm{tw})$ contains $\mathcal{O}(n^2\mathrm{tw})$ variables and $\mathcal{O}(n^2(n+\mathrm{tw}))$ clauses. From this construction, we obtain the following theorem:

Theorem 2.1. *[SV09] Given a graph $G=(V,E)$ and an integer* tw*, the formula $F(G,\mathrm{tw})$ is satisfiable if the graph G has an elimination ordering which has width* tw*, which is also the treewidth of the graph G.*

The proof this theorem follows from the construction of the formula $F(G,\mathrm{tw})$ and the previous section. This was the first encoding for finding tree decompositions and has proven to be one of the best exact methods to find tree decompositions. The SAT-encoding for finding tree decompositions was further improved by Berg and Järvisalo [BJ14], who proposed encodings for incremental SAT-solvers and MAXSAT solvers. Recently a SAT-based decomposer, Jdrasil [BBE17], was competing in the PACE 2016 [Dell16b] competition. This decomposer also included various clever preprocessing methods which included various symmetry breaking techniques. We will describe these techniques in Chapters 6 and 7.

2.5 Clique-width Decomposition

Another more general decomposition technique that we will discuss in this chapter is *clique-width decompositions* and the associated width parameter, *clique-width*. A SAT-encoding for clique-width was developed by Heule and Szeider [HS15]. Unlike the SAT-encoding for finding tree decompositions, the encoding for finding clique-width decompositions is based on partitions. We will start by defining clique-width and then formulate the encoding.

Given a graph $G = (V, E)$, the clique-width of G is the minimum integer, k, such that the graph G can be constructed using k many labels and the following algebraic operations:

Creation of a new vertex v with label i (denoted by v^i)

Disjoint Union of two labeled graphs (denoted by $\oplus$)

Relabeling of all vertices labeled i with label j (denoted by $\rho_{i \to j}$)

Edge Insertion, adding all edges between all vertices labeled i and all vertices labeled j (denoted by $\eta_{i,j}$)

Using these operations we can describe the construction of a graph with an algebraic expression. The algebraic expression is also known as k-expression and the graphs generated using the operations above are known as k-graphs.

Example 3. *The graph G in Figure 2.4 can be constructed with the following 3-expression:*

$$(\eta_{2,3}\Big((\eta_{1,3}(\eta_{1,2}\Big(1^1 \oplus 2^2 \oplus 5^3\Big))) \oplus (3^3 \oplus 4^2)\Big))$$

2.5.1 Partition-Based Reformulation

For the SAT-encoding, Heule and Szeider [HS15] developed a new characterization of clique-width based on the partitions of the vertices of the input graph. Consider an input graph $G = (V, E)$. Let V be the universe. A template T consists of two partitions $cmp(T)$ and $grp(T)$ of V. The equivalance classes in $cmp(T)$ are called components and the equivalence classes in $grp(T)$ are called groups. A derivation of length t is a sequence $D = (T_0, \ldots, T_t)$ of templates such that the following conditions hold:

D1 $|cmp(T_0)| = |V|$ and $|cmp(T_t)| = |V|$.

D2 $grp(T_i)$ is a refinement of $cmp(T_i)$, for all $0 \leq i \leq t$.

D3 $cmp(T_{i-1})$ is a refinement of $cmp(T_i)$, for all $1 \leq i \leq t$.

D4 $grp(T_{i-1})$ is a refinement of $grp(T_i)$, for all $1 \leq i \leq t$.

Observe that the T_0 starts with singleton components and groups and in the template T_t all vertices are in a single component. The *width* of a component $c \in cmp(T)$ is the number of groups $g \in grp(T)$ such that $g \subseteq c$. The width of a template is the maximum width over its components, and the width of a derivation is the maximum width over its templates. A k-derivation is a derivation of width at most k. A derivation $D = (T_0, \dots, T_t)$ is a derivation of the input graph $G = (V, E)$ if V is the universe of the derivation and the following conditions hold

Edge Property: For any two vertices $u, v \in V$ and $uv \in E$, if u, v are in the same group in some T_i for $0 < i \leq t$ then u, v are in the same component in T_{i-1}.

Neighborhood Property: For any three vertices $u, v, w \in V$ such that $uv \in E$ and $uw \notin E$, if v, w are in the same group in T_i for $0 < i \leq t$ then u, v are in the same component in T_{i-1}.

Path Property: For any four vertices $u, v, w, x \in V$ such that $uv, vw, vx \in E$ and $wx \notin E$ if u, x are in the same group in T_i and v, w are also in the same group in T_i for $0 < i \leq t$ then u, v are in the same component in T_{i-1}.

In their work Heule and Szeider [HS15] prove the following proposition.

Proposition 2.1. *[HS15, Proposition 3.9] Given a graph G with n vertices and an integer $k < n$, the graph G has clique-width at most k if and only if G has a k-derivation of length of at most $n - k + 1$*

A k-derivation can define more than one graph, where a k-expression defines a unique graph.

Example 4. *The following is a 3-derivation for the graph G from the Figure 2.4*

$$
\begin{aligned}
&cmp(T_0) = \{\{1\},\{2\},\{3\},\{4\},\{5\}\} && grp(T_0) = \{\{1\},\{2\},\{3\},\{4\},\{5\}\} \\
&cmp(T_1) = \{\{1,4,5\},\{2\},\{3\}\} && grp(T_1) = \{\{1\},\{2\},\{3\},\{4\},\{5\}\} \\
&cmp(T_2) = \{\{1,2,4,5\},\{3\}\} && grp(T_2) = \{\{1,5\},\{2\},\{4\},\{3\}\} \\
&cmp(T_3) = \{\{1,2,3,4,5\}\} && grp(T_3) = \{\{1,5\},\{2,4\},\{3\}\}
\end{aligned}
$$

2.5.2 SAT-encoding for Clique-Width

The SAT-encoding for clique-width is based on k-derivations. Let $G = (V, E)$ be a graph with $|V| = n$ and k an integer. Also, let $t = n - k + 1$ and assume that the vertices V are numbered from 1 to n. We start with constructing a formula $F(G, t)$ which expresses that the graph G has a derivation of length t. The formula $F(G, t)$ contains the *component variables* $c(u, v, i)$ for all $1 \leq u, v \leq n$, $u \neq v$ and $0 \leq i \leq t$. A variable $c(u, v, i)$ is set to

true if the vertices u and v are in the same component in the ith template. Similarly, we define *group* variables $g(u, v, i)$ for all $1 \leq u, v \leq n$, $u < v$ and $0 \leq i \leq t$, which is set to true if the vertices u and v are in the same group in the ith template. To ensure the conditions **D1–D4** the following clauses are added:

$$\begin{aligned}
&\neg c(u, v, 0)\\
&c(u, v, t)\\
&c(u, v, i) \vee \neg g(u, v, i)\\
&\neg c(u, v, i-1) \vee c(u, v, i)\\
&\neg g(u, v, i-1) \vee g(u, v, i) && \text{for all } 1 \leq u < v \leq n,\ 1 \leq i \leq t.
\end{aligned}$$

To ensure that the relations of being in the same component and same group are transitive the following clauses are added:

$$\begin{aligned}
&\neg c(u, v, i) \vee \neg c(v, w, i) \vee c(u, w, i)\\
&\neg c(u, v, i) \vee \neg c(u, w, i) \vee c(v, w, i)\\
&\neg c(u, w, i) \vee \neg c(v, w, i) \vee c(u, v, i)\\
&\neg g(u, v, i) \vee \neg g(v, w, i) \vee g(u, w, i)\\
&\neg g(u, v, i) \vee \neg g(u, w, i) \vee g(v, w, i)\\
&\neg g(u, w, i) \vee \neg g(v, w, i) \vee g(u, v, i) && \text{for all } 1 \leq u < v < w \leq n \text{ and } 0 \leq i \leq t.
\end{aligned}$$

To ensure the Edge Property the following clauses are added:

$$c(u, v, i-1) \vee \neg g(u, v, i) \qquad \text{for all } 1 \leq u < v,\ uv \in E \text{ and } 1 \leq i \leq t.$$

To ensure the Neighborhood Property the following clauses are added:

$$\begin{aligned}
&c(\min(u, v), \max(u, v), i-1) \vee \neg g(\min(v, w), \max(v, w), i)\\
&\qquad\qquad \text{for all } 1 \leq u, v, w \leq n,\ uv \in E,\ uw \notin E \text{ and } 1 \leq i \leq t.
\end{aligned}$$

To ensure the Path Property the following clauses are added:

$$\begin{aligned}
&c(u, v, i-1) \vee g(\min(u, x), \max(u, x), i) \vee \neg g(\min(v, w), \max(v, w), i)\\
&\qquad \text{for all } 1 \leq u, v, w, x \leq n,\ u < v,\ uv, uw, vx \in E,\ wx \notin E \text{ and } 1 \leq i \leq t.
\end{aligned}$$

This finishes the construction of the formula $F(G, t)$, which is true if and only if the graph G has a derivation of length t. To find the width of derivation we need to construct a formula $F(G, t, k)$, which is satisfiable if the graph G has a k-derivation. To obtain the formula $F(G, t, k)$ from the formula $F(G, t)$ we need to add clauses which restrict the number of groups in each component. In their work Heule and Szeider [HS15] lay down two different ideas on how to count the number of groups in each component. As our focus is the partition-based characterization of clique-width, we omit the details of how Heule and Szeider [HS15] count the number of groups (although this aspect is significant for the performance of their encoding).

2.6 Decompositions Addressed in this Thesis

In this section we give an overview of the (hyper) graph decompositions and their associated width parameters that we address in this thesis. We consider a wide variety of width parameters ranging from very restrictive width parameter, like, pathwidth to one of the most general width parameter, like, fractional hypertree width.

2.6.1 Branch Decomposition and Carving Decomposition

Branch decompositions and carving decompositions are used for decomposing a graph or hypergraph. A *branch decomposition* of a (hyper)graph G is a specific type of tree T where the leaves are labeled with the (hyper)edges of G. Each edge e of T partitions the edges of G in two sets and e is labeled with the vertices shared by them, called cut vertices. The width of a branch decomposition is the largest number of cut vertices associated with an edge e of T and *branchwidth* is the smallest width over all branch decompositions. *Carving decompositions* are very similar to branch decomposition with the difference of each leaf of T being labeled with a vertex of G and the width is the number of cut edges. *Carving-width* is the smallest width over all carving decompositions.

Branchwidth was introduced by Robertson and Seymour [RS91] in their Graph Minor Project and carving-width was introduced by Seymour and Thomas [ST94]. Branch decomposition are also used for decomposing combinatorial objects such as matroids, integer-valued symmetric submodular functions, etc. It is computationally hard to compute optimal branchwidth and carving-width. In Chapter 4, we develop SAT-encodings for finding both branchwidth and carving-width and a SAT-based local improvement approach for branchwidth, which improves upper bounds (obtained heuristically or approximately).

2.6.2 Tree Decomposition, Special Tree Decomposition and Path Decomposition

Tree decompositions measures how “tree-like” a graph is. A *tree decomposition* of a graph G is a tree T whose nodes are labeled with sets of vertices from G, called *bags*, such that: (1) for each edge of G there is a bag containing both ends of the edge, and (2) for each vertex of G, the nodes of T labeled with bags containing this vertex form a non-empty connected subtree. The width of the tree decomposition is the size of a largest bag minus one, and the treewidth of a graph is the smallest width over all its tree decompositions. *Special treewidth* is defined similar to treewidth, with the additional property that T is a rooted tree, and for each vertex of G there is some root-to-leaf path in T which contains all the nodes labeled with bags containing this vertex. *Pathwidth* is also defined similar to treewidth, where T itself is a path. It follows from these definitions that special

treewidth is in-between treewidth and pathwidth, i.e., for every graph G we have

$$\text{treewidth}(G) \leq \text{special treewidth}(G) \leq \text{pathwidth}(G).$$

Treewidth and pathwidth were introduced by Robertson and Seymour [RS83] also in their Graph Minor project and special treewidth was introduced by Courcelle [Cou10]. Similar to branch decompositions and carving decompositions, finding any of these decompositions optimally is also computationally hard [ACP87]. Moreover, the hardness of finding optimal special tree decompositions is our contribution (Theorem 6.1). Each of these decomposition techniques have found numerous applications in various fields of computer science. In this thesis we develop SAT-encodings for special treewidth and pathwidth (Chapter 6). We also develop local improvement techniques for treewidth (Chapter 5).

2.6.3 Fractional Hypertree Decomposition

Hypertree decompositions are tree decomposition for graphs and hypergraphs, but their width measure is not the bag size but rather the number of hyperedges required to cover the bag. This allows better measure for hypergraphs and for problem instances, whose underlying structure is a hypergraph. We get the fractional hypertree decompositions from hypertree decompositions when the edge covers are fractional. Fractional hypertree width is the associated width measure. It was introduced by Grohe and Marx [GM06, GM14] and is the most general known structural restriction that ensures polytime solvability of the Constraint Satisfaction Problem (CSP). Similar to the other decompositions we have described so far, it is not only computationally hard to find optimal fractional hypertree decompositions, but also for a given real number k and a hypergraph H checking if the fractional hypertree width of H is $\leq k$. In this thesis, we develop an SMT-encoding for computing the fractional hypertree width (Chapter 7).

2.7 Experimental Setup and Benchmarks

Most of our implementations are done in C++11 and Python 2.7. We performed our experiments on a 4-core Intel Xeon CPU E5649, 2.35GHz, 72 GB RAM machine with Ubuntu 14.04 with each process having access to at most 8 GB RAM. To schedule and manage our experiments we used sun grid engine [1]. All of our implementations are publically available via GitHub [LOS16b, LOS17b, FLS17c, FHLS18a].

Our benchmark set of graphs and hypergraphs include the following instances:

Famous Named Graphs [Wei16]: These benchmark instances are the well known named graphs that Heule and Szeider [HS15] used in their work on clique-width.

[1] more information about the hardware and the scheduler can be found at `https://www.ac.tuwien.ac.at/students/grid-engine/`

Even thought the size of these instances is relatively small, i.e. less than 150 vertices, most of these instances are not trivial. This collection contains 42 graphs.

TreewidthLIB [Bod16]: These benchmark instances are the collections of graphs, which are suitable not only for the comparison of algorithms computing treewidth, tree decompositions, but also for algorithms that solve problems related to treewidth, like branchwidth or minimum fill-in. These benchmark instances include all sizes of graphs from small and trivial graphs to large graphs, i.e. from 4 vertices to around 10000 vertices. The total number of instances in this collection is 2646.

Compiler Graphs [Hic02]: These benchmark instances were collected by Hicks [Hic02] to evaluate his work about various tools to compute branchwidth [Hic02, Hic05]. This benchmark collection contains graphs of medium size, i.e., around 150 vertices.

Hyperbench [FGLP17]: These benchmark instances are a collection of hypergraphs that were generated from various applications such as constraint satisfaction problem and databases. The hypergraph instances in this benchmark collection contain small to large sized graphs. This benchmark collection contains a total of 2191 instances.

PACE Competition Graphs [DHJ+17, DKTW18]: These benchmark instances were used in PACE competitions to evaluate the performance of various tools for finding treewidth. The graphs in this collection range from small to large graphs, with a total of 283 graphs.

UAI Competition Networks [Dec13a]: These benchmark instances were generated from moral graphs of the Bayesian networks used for UAI competitions. There are 467 large graphs in this collection.

GTFS-Transit Feed Graphs [Fic16]: These benchmark instances were extracted from the publically available transit graphs from GTFS-transit feed. The 590 graphs in this collection range from small to large graphs.

Random Graphs: We generated some random graphs to evaluate and compare the performance of some of our techniques. We generated 20 random gnp graphs with 20, 30, 40, 50, and 60 vertices and edge probabilities between 0.1 to 0.9 with a step of 0.1. In total we considered 900 instances.

Square Grids: Square grids is one of the sparse graph classes which has high treewidth. We considered Square grids with up to 100 vertices.

Complete Graphs: Complete graphs are the most dense graphs with high treewidth. We considered complete graphs with n vertices (K_n) and complete bipartite graphs with $2n$ vertices $K_{n,n}$. We considered graphs with up to 130 vertices for both classes of complete graphs.

We primarily used the DIMACS format [DIM] as default format for graphs and CNF formulas. For hypergraphs we used the format used in hyperbench library [FGLP17]. We would like to point out that not all of these benchmark instances are interesting for all the parameters we study in this work. Thus for each width parameter that we study we provide a more detailed overview of the benchmarks we use in the corresponding chapters. Similarly, we will provide detailed information on the hardware and software configuration in the corresponding chapters.

In this chapter we described the basic preliminaries used overall in this work. We defined the basic SAT terminologies along with the sequential cardinality counter, basic graph theory along with the SAT-encodings for treewidth and clique width, the DIMACS format, which we use for our input and output. As we already mentioned, we will provide detailed problem specific preliminaries as well as the experimental setup in the corresponding chapters.

CHAPTER 3

Overview of Results

The focus of this work is to study and develop efficient SAT-based approaches for finding optimal or close to optimal structural decompositions of hypergraphs and graphs. We have successfully designed new approaches for finding various decompositions. We focused not only on the exact approaches to find optimal decompositions but also on approaches to improve upper bounds obtained via other methods, like, heuristics or approximations. In this chapter we provide an overview of our SAT-based approaches and their overall performance and significance. We divide the chapter in two parts (i) the methodologies used in our work, i.e., the exact methods, the local improvement technique used for improving the upper bounds, and preprocessing techniques, which enable us to deal with large instance, and (ii) the results for each individual width parameter that we studied, i.e., branchwidth and carving width, treewidth, special treewidth and pathwidth, and fractional hypertree width, respectively.

3.1 Methodologies

3.1.1 Exact Results

One of the crucial initial observations that we made during the development of SAT-encodings was the importance of the characterization of the width parameter on which the encoding is based. Encodings based on different characterizations of the same width parameters can have significantly varied performance. Thus, in the course of finding efficient SAT-encodings we also explored and developed various characterization for width parameters. Two types of characterizations that we focus on are:

- ordering-based characterizations and
- partition-based characterizations.

As we already know from the Chapter 1, an ordering-based characterization was developed for treewidth [SV09] and has so far proven to be the best known characterization for SAT-encodings for finding tree decompositions. Whereas, the novel partition-based characterization was developed for clique-width [HS15]. These two approaches have been our main focus while developing the various SAT-encodings. We also developed SAT-encodings based on more standard characterizations of the decompositions.

The next natural question we focused on was to compare the SAT-encodings based on both types of characterizations. For special treewidth and pathwidth we developed encodings based on both ordering-based and partition-based characterization. We conducted extensive experiments to compare the two characterizations and determine their strengths.

Since the SAT-solvers are restricted to natural numbers we wanted to investigate if we can develop SAT-based methods to compute fractional hypertree width, which ranges over real numbers. We successfully employed an SMT-encoding[1] for finding the fractional hypertree width of hypergraphs. We formalized and based our encoding on an ordering-based characterization.

Overall our work showed that SAT-based approaches can be successfully applied to find various width parameters. As our encodings are polynomial in the size of the input instance and the parameter, they scale better than the combinatorial algorithms, which usually have exponential dependency on the width.

3.1.2 Local Improvement

The SAT-based and combinatorial algorithms, no matter how good they are, cannot solve instances of large size due to the large number of variables and clauses in the encoding. Generally, a SAT-encoding for finding any decomposition for a given graph is at least cubic in the number of vertices of the graph, leading to an encoding of size say larger than 1GB for a graph with 100 vertices. To overcome this limitation one needs to rely on inexact approaches like, heuristics or approximation algorithms. It is a natural question to ask: can we apply SAT-based approaches to larger instances and possibly improve upon known heuristics or approximations? The local improvement approach tries to close the gap between the exact approach and the inexact approach. The idea is to use an exact method to improve the decompositions obtained using inexact methods. To be precise, we first construct a decomposition using the inexact method and then we identify the parts of the decomposition which have high width (local decomposition) and can possibly be improved. We improve these decompositions using SAT-encodings mentioned in the previous section (local method) as long as we can, or reach a time out. We developed local improvement approaches for branchwidth and treewidth. Moreover, we tested these approaches on our benchmark instances. The detailed local improvement approaches for branch decomposition and tree decomposition are given in Chapters 4 and 5, respectively.

[1] Which is basically a SAT-encoding augmented with real number constraints handled by an arithmetic theory solver.

3.1.3 Preprocessing

One of the biggest obstacles during the development of practical techniques and algorithms is the size of the input instance. Preprocessing is one of the techniques that allows us, in many cases, to reduce the input instance to an equivalent smaller instance such that some properties are preserved. In our case, we want to focus on preprocessing techniques which, given an input (hyper)graph G outputs a (hyper)graph G' which is not larger than G (preferably smaller than G), i.e., has fewer vertices and/or edges and has the same width as the input graph G. Similarly one can also use symmetry breaking techniques to aid SAT-solvers by reducing the search space. We consider symmetry breaking methods as a part of preprocessing since symmetries are usually detected in the preprocessing phase.

For treewidth there have been numerous studies on suitable preprocessing techniques. In our work we study the previous existing preprocessing techniques and attempt to extend these techniques so that they can be applied for other width parameters. When studying more restricted parameters, it turns out that not all preprocessing techniques developed for a more general width parameter are applicable. Therefore, we characterize the preprocessing techniques into:

(i) techniques that preserve optimal widths and can be successfully applied for finding optimal width,

(ii) techniques that do not preserve optimal widths but can be used for approximation, and

(iii) techniques that neither preserve optimality nor can be used for approximation.

Since our main focus is on finding optimal widths we use in our implementations only the techniques that preserved optimality. Our experimental evaluations shows that preprocessing has a significant impact, when we want scale to large instances and/or solve more instances.

3.2 Results

3.2.1 Branchwidth and Carvingwidth

We started our work by focusing on developing SAT-based methods for branch decompositions (see Section 2.6.1). After realizing the poor performance of the SAT-encoding based on the standard definition of branch decomposition, we developed a new partition-based characterization for branch decomposition and a corresponding SAT-encoding. This new encoding enabled us to find optimal branch decompositions of input graphs with up to 80 vertices, whereas the initial encoding could only solve graph instances which had less than 40 vertices. Eventhough, for graphs with up to 80 vertices and branchwidth up to 8, our new characterization could not find the exact branchwidth faster than the combinatorial

algorithm, which was developed by Hicks [Hic05]. However, it could scale better, i.e., it could provide us exact solutions for large graphs with high branchwidth. Similar to the SAT-encoding for finding branch decomposition, we developed SAT-encodings for finding carving decompositions (see the definition in Section 2.6.1). This encoding performed similar to the branchwidth encoding

Additionally, we observed that the bottleneck for finding optimal decompositions for a graph G with branchwidth bw is the time taken by the UNSAT call for checking if branchwidht is $\mathrm{bw} - 1$. This call requires significantly more time compared to any SAT call for checking if the width is at most k for $k \geq \mathrm{bw}$. This led us to believe that the SAT-encodings could be useful also for finding upper bounds. But as the SAT-encoding for large graphs is too large for any SAT-solver it is not really feasible to use this technique for finding upper bounds on branchwidth for large graphs. To get around this problem we developed the local improvement technique.

The above illustrated technique performed exceptionally well and showed potential to be used in practice. When measuring the performance of our local improvement technique we looked at various qualitative and quantitative parameters. For the qualitative parameters we looked at the size of the instances and the improvement in the width. The local improvement technique could improve the width of graph instances with more than 10000 edges and could improve the width of some instances by more than 10. The improved instances also included instances which are small enough to be considered for dynamic programming and for some of such instances we could improve the width such that the dynamic programming would be feasible which was not the case for heuristic decomposition.

Overall, this approach was successful as it showed us that SAT-solvers can be used to compute branchwidth. These technique can be used not only for finding optimal width but also for finding close to optimal upper bounds, which then in turn allowed us to use SAT-solver for local improvement.

3.2.2 Treewidth

After the success of local improvement techniques for improving the heuristic branch decompositions, we wanted to study whether the same could be observed for tree decompositions (see Section 2.6.2). In the past few years, the computation of treewidth has been of great interest. Consequently, there are many tools that can compute optimal or suboptimal tree decompositions. In our work on developing local improvement techniques for tree decomposition we used these state of the art tools. One of these tools uses SAT-solvers for finding the treewidth.

In order to successfully apply the local improvement techniques, we developed the correct technique to extract and integrate the local tree decomposition such that the resulting decomposition is a tree decomposition for the input graph. The performance of the local improvement techniques for improving the tree decompositions was similar to the performance of the local improvement technique for improving branch decompositions.

We could handle large graphs with around 5000 vertices and improve the widths of tree decompositions significantly. Similarly, we could also improve widths of some graph instances such that they can be used for dynamic programming.

One of the positive aspect of this project is the versatility of the tool, trellis, that we developed for the local improvement of tree decompositions. One can plug in any tool, as the local solver, that supports the PACE competition [DHJ+17, DKTW18] format.

In our experiments we also observed that the SAT-based solvers performed better than the other exact methods. They could improve more instances and the overall improvement was higher as well. Overall, from our work on developing local improvement techniques for improving tree decompositions we could deduce that this approach is very effective in improving the treewidth upper bounds for large graphs.

3.2.3 Special Treewidth and Pathwidth

After studying SAT-encoding techniques for branch decompositions and tree decompositions we wanted to compare the two characterizations, i.e., the ordering-based characterization and the partition based characterization. We compared these two characterizations in our work on developing SAT-encodings for special treewidth and pathwidth (see Section 2.6.2). We started with developing two new characterizations for special tree decomposition based on the two techniques and a partition-based characterization for path decomposition. For pathwidth, there already existed an ordering-based characterization known as vertex separation number. After implementing and comparing the two techniques we came to the conclusion that even though the ordering-based technique works well on some small sparse graphs, the partition-based technique outperforms it in terms of scalability and robustness. The partition based techniques could consistently solve most of the input graph instances which were in a certain size range.

We also studied various preprocessing techniques which could be applicable for special treewidth and pathwidth. Contrary to one's expectation, most of the known preprocessing techniques that exist for tree decompositions could not be admitted for either of these two parameters. Therefore, we investigated further to determine the exact difference that arose due to the application of some of the preprocessings. We successfully proved that most of these techniques are viable for heuristics and approximations.

3.2.4 Fractional Hypertree Width

After the success of SAT-encodings for the above decompositions we focused on fractional hypertree decompositions. As we already know from Section 2.6.3, fractional hypertree width (fhtw) is the most general structural restriction that allows polytime solvability for CSP. It has so far been mostly of theoretical interest due to lack of techniques that could compute fhtw. The SAT-encodings have proven to be very effective for finding most of the width parameters. Thus it would be natural to question if we can use SAT technologies to find fhtw.

One of the limitations SAT-solver have is that they are restricted to integers. The obvious choice to overcome this issue was to attempt an SMT-encoding, which allows one to use linear arithmetic. In our work for finding exact fhtw, we developed an SMT-encoding and various preprocessing techniques.

In this work we successfully developed the first approach for finding optimal fhtw using the SMT techniques. This will allow further development of practical techniques based on fractional hypertree decompositions and it would not longer be restricted to theoretical studies. Also from our work, we again showed the emphasis preprocessing techniques have over solving time.

In this chapter, we provided an overview of the results presented in this thesis the main focus being on the SAT-based exact method and the local improvement approach to find decompositions, and preprocessing techniques. We also discuss the main results for each width measure that we study.

CHAPTER 4

Branchwidth and Carving-Width

In this chapter we will construe our first SAT-encodings for branchwidth and carving-width and the the first the local improvement approach for branchwidth. This chapter is based on our paper accepted at SAT 2016 [LOS16a] and the full version is conditionally accepted at the journal Transactions on Computational Logic (TOCL). We start with a brief introduction and proceed with describing the SAT-encoding for the branch decomposition which will also form the basis of our local improvement techniques described later in this chapter. After that we present the encoding for carving width and at the end of this chapter we lay out our experimental work.

4.1 Introduction

Background: Branch decomposition is a prominent method for structurally decomposing a graph or hypergraph. This decomposition method was originally introduced by Robertson and Seymour [RS91] in their Graph Minors Project and has become a key notion in discrete mathematics and combinatorial optimization. Branch decompositions can be used to decompose other combinatorial objects such as matroids, integer-valued symmetric submodular functions, and propositional CNF formulas (after dropping of negations, clauses can be considered as hyperedges). The width of a branch decomposition provides a measure of how well it decomposes the given object; the smallest width over its branch decompositions denotes the *branchwidth* of an object. As we already know, many hard computational problems can be solved efficiently by means of dynamic programming along a branch decomposition of small width. Prominent examples include the traveling salesman problem [CS03], the #P-complete problem of propositional model counting [BDP03], and the generation of resolution refutations for unsatisfiable CNF formulas [AR02]. Branch decompositions also form the basis of several width-parameters employed in Knowledge Compilation and Reasoning [Dar09], where they are known as

dtrees. In fact, all decision problems on graphs that can be expressed in monadic second order logic can be solved in linear time on graphs that admit a branch decomposition of bounded width [Gro08].

A bottleneck for all these algorithmic applications is the space requirement of dynamic programming, which is typically single or double exponential in the width of the given branch decomposition. Hence it is crucial to compute first a branch decomposition whose width is as small as possible. This is very similar to the situation in the context of treewidth, noted about inference on probabilistic networks of bounded treewidth [KGOD11] (Chapter 1). Hence small improvements in the width can change a dynamic programming approach from unfeasible to feasible. The boundary between unfeasible and feasible width values strongly depends on the considered problem and the currently available hardware. For instance, Cook and Seymour [CS03] mention a threshold of 20 for the Traveling Salesman Problem. Today one might consider a higher threshold.

Computing an optimal branch decomposition is NP-hard [ST94].

Contribution: In this chapter we propose a practical SAT-based approach for finding a branch decompositions of small width. At the core of our approach is an efficient SAT-encoding which takes a hypergraph H and an integer w as input and produces a propositional CNF formula which is satisfiable if and only if H admits a branch decomposition of width w. By multiple calls of the solver with various values of w we can determine the smallest w for which the formula is satisfiable (i.e., the branchwidth of H), and we can transform the satisfying assignment into an optimal branch decomposition. Our encoding is based on a novel *partition-based* characterization of branch decompositions in terms of certain sequences of partitions of the set of edges. This characterization together with clauses that express cardinality counters allow for an efficient SAT-encoding that scales up to instances with about hundred edges. The computationally most expensive part in this procedure is to determine the optimality of w by checking that the formula corresponding to a width of $w - 1$ is unsatisfiable. If we do not insist on optimality and aim at good upper bounds, we can scale the approach to larger hypergraphs with over two hundred edges.

The number of clauses in the formula is polynomial in the size of the hypergraph and the given width w, but the order of the polynomial can be quintic, hence there is a firm barrier to the scalability of the approach to larger hypergraphs. In order to break through this barrier, we developed a new *SAT-based local improvement* approach where the encoding is not applied to the entire hypergraph but to certain smaller hypergraphs that represent local parts of a current candidate branch decomposition. The overall procedure thus starts with a branch decomposition obtained by a heuristic method and then tries to improve it locally by multiple SAT-calls until a fixed-point (or timeout) is reached. This method scales now to instances with several thousands of vertices and edges and branchwidth upper bounds well over hundred. We believe that a similar approach using a SAT-based local improvement could also be developed for other (hyper)graph width measures.

Encouraged by the good performance of our partitioned-based encoding for branchwidth, we explored whether a similar encoding can be used for other decompositional parameters. We succeeded to develop a similar encoding for *carving-width*, which is a decompositional parameter closely related to branchwidth [ST94] with applications to graph drawing [BV12, Bie14]. We will mainly focus on branchwidth and provide a brief description on how similar techniques can be applied to carving-width.

Implementations of our encodings as well as the local improvement algorithm are publicly available under `https://www.ac.tuwien.ac.at/research/branchlis/`.

Related Work: For finding branch decompositions of smallest width, Robertson and Seymour [RS91] suggested an exponential-time algorithm which was later implemented by Hicks [Hic05]. This algorithm runs in time $O(n^{2w-2}m)$ for checking whether a hypergraph with n vertices and m edges has a branch decomposition of width w. Further exponential-time algorithms have been proposed (see, for instance [FMT09, HO08]) but there seem to be no implementations. Ulusal [Ulu08] proposed three different encoding to integer programming (CPLEX). Out of the three, the size of best performing encoding is exponential in the number of edges, which renders it inapplicable for graphs with more than 20 edges. One could also find suboptimal branch decompositions based on the related notion of tree decompositions; however, finding an optimal tree decomposition is again NP-hard, and by transforming it into a branch decomposition one introduces an approximation error factor of up to 50% [RS91] which makes this approach prohibitive in practice. For practical purposes one therefore mainly resorts to heuristic methods that compute suboptimal branch decompositions [CS03, Hic02, OPB11].

4.2 Preliminaries

We refer for basic definitions to Chapter 2. In this section we will provide some more basic properties specifically related to branch decompositions.

Let $H = (V(H), E(H))$ be a hypergraph. Every subset E of $E(H)$ defines a *cut* of H, i.e., the pair $(E, E(H) \setminus E)$. We denote by $\delta_H(E)$ (or just $\delta(E)$ if H is clear form the context) the set of *cut vertices* of E in H, i.e., $\delta(E)$ contains all vertices incident to both an edge in E and an edge in $E(H) \setminus E$. Note that $\delta(E) = \delta(E(H) \setminus E)$.

A *branch decomposition* $\mathcal{B}(H)$ of H is a pair (T, γ), where T is a ternary tree and $\gamma : L(T) \to E(H)$ is a bijection between the edges of H and the leaves of T (denoted by $L(T)$). For simplicity, we write $\gamma(L)$ to denote the set $\{\, \gamma(l) \mid l \in L \,\}$ for a set of leaves L of T and we also write $\delta(T')$ instead of $\delta(\gamma(L(T')))$ for a subtree T of T'. For an edge e of T, we denote by $\delta_{\mathcal{B}}(e)$ (or simply $\delta(e)$ if $\mathcal{B}$ is clear from the context), the set of *cut vertices* of e, i.e., the set $\delta(T')$, where T' is any of the two components of $T \setminus \{e\}$. Observe that $\delta_{\mathcal{B}}(e)$ consists of the set of all vertices v such that there are two leaves l_1 and l_2 of T in distinct components of $T \setminus \{e\}$ such that $v \in \gamma(l_1) \cap \gamma(l_2)$. The *width* of an edge e of T is the number of cut vertices of e, i.e., $|\delta_{\mathcal{B}}(e)|$ and the *width* of $\mathcal{B}$ is the maximum width of any edge of T. The *branchwidth* of H is the minimum width over all branch

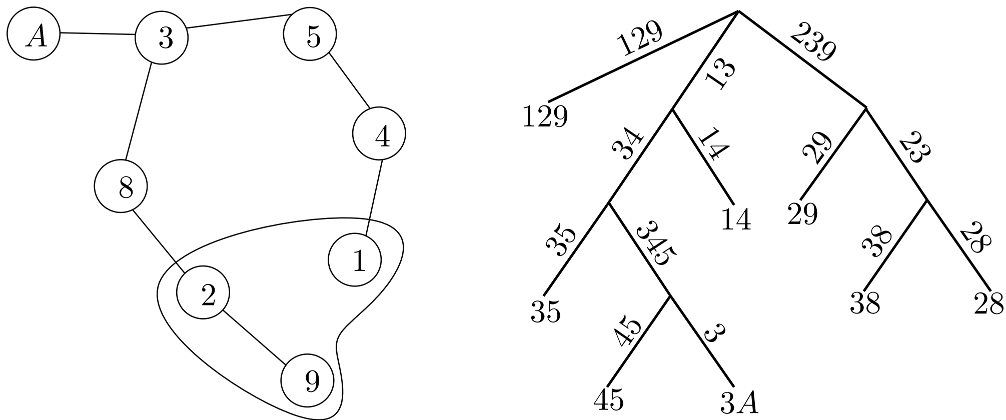


Figure 4.1: A hypergraph H (left) and an optimal branch decomposition (T, γ) of H (right). The labels of the leaves of T are the edges assigned to them by γ and the labels of the edges of T are the cut vertices of that edge.

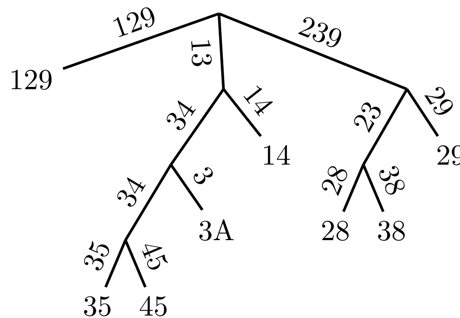


Figure 4.2: An alternative branch decomposition (T', γ') of the hypergraph H from Fugue 4.1. The labels of the leaves of the branch decompositions T and T' are the same and are exactly the edges for the hypergraph H. The two decompositions T and T' only differ in the edges.

decompositions of H (or 0 if $|E(G)| = 0$ and H has no branch decomposition). We also define the *depth* of $\mathcal{B}$ as the radius of T. Fig. 4.1 illustrates a branch decomposition of a small hypergraph. In the figure and in the remainder of the paper we will often denote a set $\{1, 2, 3, A\}$ of vertices as $123A$. We will use the following property of branch decompositions.

Proposition 4.1. *Let $\mathcal{B} := (T, \gamma)$ and $\mathcal{B}' := (T', \gamma')$ be two branch decompositions of the same hypergraph H. Then there is bijection $\alpha : V(T) \to V(T')$ between the vertices of T and T' such that $l \in L(T)$ if and only if $\alpha(l) \in L(T')$ and moreover $\gamma(l) = \gamma'(\alpha(l))$ for every $l \in L(T)$. In other words w.l.o.g. one can assume that $\mathcal{B}$ and $\mathcal{B}'$ differ only in terms of the edges of T and T'.*

Proof. Observe that $\mathcal{B}$ and $\mathcal{B}'$ are branch decompositions of the same hypergraph H, therefor the leaves of both T and T' are the edges of H, i.e. it holds that $|L(T)| = |L(T')| = |E(H)|$. Moreover, because all inner vertices of T and T' are ternary, it holds that $|V(T)| = |V(T')| = 2|E(H)| - 2$. Hence, the bijection α can be obtained by setting $\alpha(l)$ to be the leaf l' of T' with $\gamma(l) = \gamma'(l')$ for every leaf l of T and choosing an arbitrary bijection between the remaining (inner) vertices of T and T'. ☐

Figure 4.2 shows an alternative decomposition for the hypergraph H from the Figure 4.1.

4.3 Tree Encoding for Branchwidth

We start with the straightforward encoding that we constructed based on the original definition of branch decompositions introduced by Robertson and Seymour [RS91]. Let H be a hypergraph. In the following we will give an encoding that guesses a branch decomposition (T, γ) of H using an encoding, which we call tree encoding. It follows from Proposition 4.1 that any two branch decompositions differ only in terms of the edges of the underlying trees. Hence, for the following we will assume that the nodes of T, the set of nodes of T that are leaf nodes as well as the bijection γ are fixed. It will be convenient to assume that the branch decomposition is rooted. The root can be chosen arbitrarily among the inner nodes of T as it has no affect on the branch decomposition or the width of the decomposition. Hence, the edges of T (and also a branch decomposition of H) is completely determined after assigning exactly 3 children to the root and exactly two children to every inner node of T. These ideas also form the main ideas behind our encoding.

In the following we assume that the vertices of H are numbered from 1 to n and the edges of H are numbered from 1 to m. As every branch decomposition of H has exactly $2m - 2$ nodes (of which exactly m are leaf nodes), in the following, we will assume that the nodes of the branch decomposition are numbered from 1 to $2m - 2$ in such a way that parents always appear after their children in the ordering. In particular, the first m nodes represent all the leaf nodes of the branch decomposition.

In our encoding we want to construct a propositional CNF formula $F(H, \text{bw})$ such that the hypergraph H has branchwidth bw. We start the construction of $F(G, \text{bw})$ by first constructing the formula $F(H)$ which encodes the branch decomposition i.e., $F(H)$ is satisfiable if H has a branch decomposition T, γ. The formula $F(H)$ uses the following variables. For every $i, j \in V(T)$ with $i > j$, two variables $\text{left}(i, j)$ and $\text{right}(i, j)$ that are true if and only if j is the left or right child of i in T. We also introduce one variable $\text{mid}(j)$ for every j with $1 \leq j < 2m - 2$, which holds if and only if j is the middle child of the root node. Finally we introduce one variable $\text{leaf}(i, e)$ for every $i \in V(T)$ and $e \in E(G)$, which will hold if and only if the leaf assigned to e is contained in the subtree below i of T.

To ensure that every inner node of T has exactly one left child and one right child, we add the following clauses:

$$\bigvee_{1 \le j < i} \text{left}(i,j) \wedge \bigwedge_{1 \le j < j' < i} (\neg\text{left}(i,j) \vee \neg\text{left}(i,j')).$$
$$\bigvee_{1 \le j < i} \text{right}(i,j) \wedge \bigwedge_{1 \le j < j' < i} (\neg\text{right}(i,j) \vee \neg\text{right}(i,j')).$$
$$\text{for all } m+1 \le i \le 2m-2.$$

We add similar clauses for the middle child of the root node:

$$\bigvee_{1 \le j < 2m-2} \text{mid}(j) \wedge \bigwedge_{1 \le j < j' < 2m-2} (\neg\text{mid}(j) \vee \neg\text{mid}(j')).$$

To enforce that the children of every node are distinct, we add the following clauses:

$$\neg\text{left}(i,j) \vee \neg\text{right}(i,j) \qquad \text{for all } j < i,\ 1 \le j < 2m-2, \text{ and } m < i.$$

We add similar clauses for the root node to say that all three of its children are distinct:

$$(\neg\text{left}(2m-2,j) \vee \neg\text{mid}(j)) \wedge (\neg\text{mid}(j) \vee \neg\text{right}(2m-2,j)) \wedge$$
$$(\neg\text{left}(2m-2,j) \vee \neg\text{right}(2m-2,j))$$
$$\text{for all } 1 \le j \le 2m-1.$$

To ensure that every inner node of T apart from the root node has at most one parent node, we add the following clauses:

$$(\neg\text{left}(i,j) \vee \neg\text{left}(i',j)) \wedge (\neg\text{left}(i,j) \vee \neg\text{right}(i',j)) \wedge$$
$$(\neg\text{right}(i,j) \vee \neg\text{left}(i',j)) \wedge (\neg\text{right}(i,j) \vee \neg\text{right}(i',j)) \wedge$$
$$(\neg\text{mid}(j) \vee \neg\text{left}(i,j) \wedge (\neg\text{mid}(j) \vee \neg\text{right}(i,j))$$
$$\text{for all } 1 \le j \le i,\ m < i < i' \le 2m-2.$$

In the following we will list the clauses ensuring that the variables leaf(i,e) are properly assigned. The assignment of these variables, will follow from the fixed bijection γ (which will assign the edge with number i to the i-th leaf of T) and the assignment of the variables left(i,j) and right(i,j). We start with fixing the bijection γ to be $\gamma(l) = l$ for every leaf node l of T. In order to add this in the formula we add the following unit clauses :

$$\text{leaf}(i,i) \qquad \text{for all } 1 \le i \le m.$$
$$\text{leaf}(i,j) \qquad \text{for all } 1 \le i,j \le m \text{ and } i \ne j.$$

Next we ensure that if an edge belongs to some node of T, then it also belongs to its parent node. To do so we introduce the following clauses for every i, j, and e with $1 \le e \le m$, $1 \le j < i$, and $m < i \le 2m-2$:

$$(\neg\text{left}(i,j) \vee \neg\text{leaf}(j,e) \vee \text{leaf}(i,e)) \wedge$$
$$(\neg\text{right}(i,j) \vee \neg\text{leaf}(j,e) \vee \text{leaf}(i,e))$$
$$\text{for all } 1 \leq e \leq m,\ 1 \leq j \leq i \text{ and } m+1 \leq i \leq 2m-2.$$

As well as the following clauses for every j, and e with $1 \leq e \leq m$, $1 \leq j < 2m-2$:

$$(\neg\text{mid}(j) \vee \neg\text{leaf}(j,e) \vee \text{leaf}(2m-2,e)) \qquad \text{for all } 1 \leq e \leq m \text{ and } 1 \leq j \leq 2m-1.$$

Conversely to say that if $e \in E(H)$ does not belong to any of i's children then e does not belong to i, we add the following clauses:

$$\neg\text{left}(i,j) \vee \neg\text{right}(i,k) \vee \text{leaf}(j,e) \vee \text{leaf}(k,e) \vee \neg\text{leaf}(i,e)$$
$$\text{for all } 1 \leq e \leq m,\ 1 \leq j,k < i < 2m-2, \text{ and } i > m.$$

This concludes the part of the encoding that constructs the formula $F(H)$, which defines the branch decomposition for the hypergraph H. In the following we will give the clauses and variables necessary to identify the cut vertices of each edge of T. We therefore introduce the variables $\text{c}(i,u)$ for every i and u with $1 \leq i \leq 2m-2$ and $u \in V(H)$, which are true if and only if u is a cut vertex of the edge of T incident to i and its parent.

To ensure that whenever u is a cut vertex of the edge incident to i and its parent in T, then $\text{c}(i,u)$ is true. We add the following clauses:

$$(\neg\text{leaf}(i,e) \vee \text{leaf}(i,f) \vee \text{c}(i,u)) \wedge$$
$$(\text{leaf}(i,e) \vee \neg\text{leaf}(i,f) \vee \text{c}(i,u)).$$
$$\text{for all } 1 \leq u \leq n,\ 1 \leq e < f \leq m,\ u \in i,\ u \in f \text{ and } 1 \leq i \leq 2m-2.$$

Note that, here, we do not need the reverse direction of the above, i.e., the above is sufficient if we only want to restrict the number of cut vertices for each edge of T. Hence, the following clauses (that give the reverse direction) are redundant (but could be included to improve the encoding).

$$(\bigvee_{e \in E_u} \neg\text{leaf}(i,e) \vee \neg\text{c}(i,u)) \wedge$$
$$(\bigvee_{e \in E_u} \text{leaf}(i,e) \vee \neg\text{c}(i,u))$$
$$\text{for all } 1 \leq u \leq n,\ 1 \leq i \leq 2m-2, \text{ and } E_u = \{e | u \in e \text{ and } 1 \leq e \leq m\}.$$

We use the sequential cardinality counters (Chapter 2) to restrict the number of $\text{c}(i,u)$ variables set to true to k.

4.4 Partition-based Reformulation of Branchwidth

In this section we introduce our novel characterization and the corresponding encoding for branchwidth. The encoding is based on a partition-based reformulation of branchwidth in terms of derivations, which will also lead to an efficient encoding for the related notion of carving-width.

One might be tempted to think that the original characterization of branch decompositions as ternary trees (Section 4.3) leads to a very natural and efficient SAT-encoding for the existence of a branch decomposition of a certain width. However, to our surprise the performance of the encoding based on this characterization of branch decomposition was very poor. We therefore opted to develop a different encoding based on a new partition-based characterization of branch decomposition which we will introduce next. Compared to this, the original encoding was clearly inferior, resulting in an encoding size that was always at least twice as large and overall solving times that were longer by a factor of 3-10, even after several rounds of fine-tuning and experimenting with natural variants.

Lets start with defining the new partition-based characterization for branch decomposition. Let H be a hypergraph. A *derivation* $\mathcal{P}$ of H of *length* l is a sequence $(P_1, \dots, P_l)$ of partitions of $E(G)$ such that:

(D1) $P_1 = \{\, \{e\} \mid e \in E(H)\,\}$ and $P_l = \{E(H)\}$ and

(D2) for every $i \in \{1, \dots, l-2\}$, P_i is a 2-ary refinement of P_{i+1} and

(D3) P_{l-1} is a 3-ary refinement of P_l.

The *width* of $\mathcal{P}$ is the maximum size of $\delta_H(E)$ over all sets $E \in \bigcup_{1 \leq i < l} P_i$. We will refer to P_i as the i-th *level* of the derivation $\mathcal{P}$ and we will refer to elements in $\bigcup_{1 \leq i \leq l} P_i$ as *sets* of the derivation. We will show that any branch decomposition can be transformed into a derivation of the same width and also the other way around. The following example illustrates the close connection between branch decompositions and derivations.

Example 5. *Consider the branch decomposition $\mathcal{B}$ given in Fig. 4.1. Then $\mathcal{B}$ can, e.g., be translated into the derivation $\mathcal{P} = (P_1, \dots, P_5)$ defined by:*

$$P_1 = \Big\{\{129\}, \{35\}, \{45\}, \{3A\}, \{14\}, \{28\}, \{38\}, \{29\}\Big\}$$

$$P_2 = \Big\{\{129\}, \{35\}, \{45, 3A\}, \{14\}, \{28\}, \{38\}, \{29\}\Big\}$$

$$P_3 = \Big\{\{129\}, \{35, 45, 3A\}, \{14\}, \{28, 38\}, \{29\}\Big\}$$

$$P_4 = \Big\{\{129\}, \{35, 45, 3A, 14\}, \{28, 38, 29\}\Big\}$$

$$P_5 = \Big\{\{129, 35, 45, 3A, 14, 28, 38, 29\}\Big\}$$

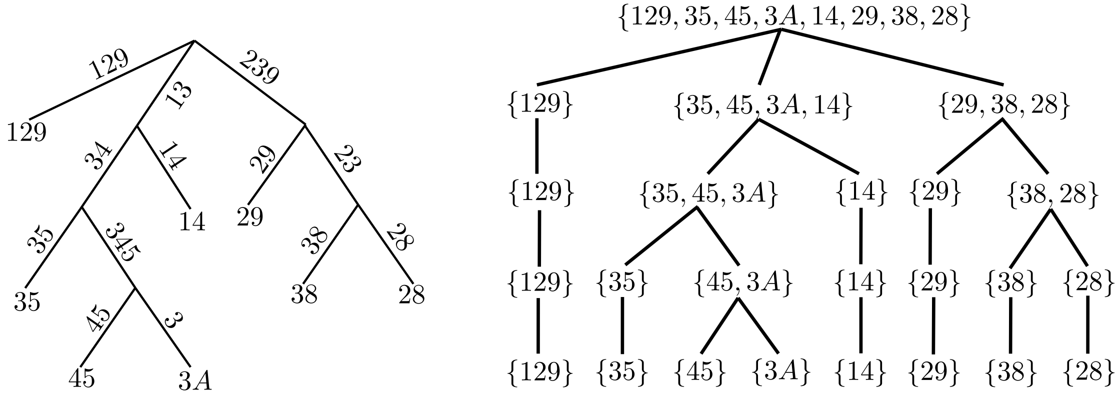


Figure 4.3: A branch decomposition (T, γ) (left), of width 3 and depth 4, and a corresponding derivation $\mathcal{P}$ (right), of width 3 and depth 4.

The width of $\mathcal{B}$ is equal to the width of $\mathcal{P}$.

Figure 4.3 shows the relation between branch decomposition and the derivations. The following theorem shows that derivations provide an alternative characterization of branch decompositions.

Theorem 4.1. *Let H be a hypergraph and w and d two integers. H has a branch decomposition of width at most w and depth at most d if and only if H has a derivation of width at most w and length at most d.*

Proof. Towards showing the forward direction of the theorem, let $\mathcal{B} := (T, \gamma)$ be a branch decomposition of width at most w and depth at most d. Moreover, let r be an arbitrary node of degree three at the center (Section 2.3) of T and assume in the following that T is rooted in r. Observe that because r is in the center of T, it holds that $\mathrm{h}(T_r) = d$. Let t be a node of T. We define $E(t)$ to be the set of all edges of H represented by the leaves of the subtree T_t, i.e., $E(t) := \gamma(L(T_t))$. We claim that $\mathcal{P} := (P_1, \dots, P_{\mathrm{h}(T_r)})$, where $P_i := \{\, E(t) \mid t \in V(T) \text{ and } \mathrm{h}(T_t) = i \,\}$, is a derivation of H of width at most w and length at most d. As in every tree, the set of all subtrees of T of a fixed height partitions the leaves of T, we obtain that P_i is a partition of $E(H)$ for every i with $1 \leq i \leq \mathrm{h}(T_r)$. Because $P_1 = \{\, E(l) \mid l \in L(T) \,\} = \{\, \gamma(l) \mid l \in L(T) \,\}$, $P_{\mathrm{h}(T_r)} = \{E(r)\} = \{E(H)\}$, and $\mathcal{B}$ is a branch decomposition, we obtain that $\mathcal{P}$ satisfies *(D1)*. Since every node of T apart from r has at most two children and r has exactly three children, we obtain that P_i is a 2-ary refinement of P_{i+1} for every i with $1 \leq i < \mathrm{h}(T_r) - 1$ and $P_{\mathrm{h}(T_r)-1}$ is a 3-ary refinement of $P_{\mathrm{h}(T_r)}$, which shows that $\mathcal{P}$ satisfies *(D2)* and *(D3)*. Hence, $\mathcal{P}$ is a derivation of H and because $\mathrm{h}(T_r) = d$, as observed in the beginning of the proof, we obtain that the length of $\mathcal{P}$ is at most d.

It remains to show that the width of $\mathcal{P}$ is at most w. To see this let $E \in \bigcup_{1 \leq i < d} P_i$. Then $E = E(t)$ for some $t \in V(T) \setminus \{r\}$. Hence, $\delta_H(E)$ is equal to $\delta_{\mathcal{B}}(\{t, \mathrm{p}(t)\})$, which is at most w given the width of $\mathcal{B}$ is at most w.

Towards showing the backward direction of the theorem, let $\mathcal{P} := (P_1, \dots, P_d)$ be a derivation of H of width at most w. We will first show that w.l.o.g. we can assume that $|P_{d-1}| = 3$. Since we can assume that $\mathcal{P}$ is a minimal derivation, i.e., every subsequence of $\mathcal{P}$ is not a derivation, we obtain that $P_{d-1} \neq P_d$ and hence $|P_{d-1}| \geq 2$. Suppose that $|P_{d-1}| = 2$. Because we can assume that H has at least three edges there is a $p \in P_{d-1}$ and a level i with $1 \leq i < d-1$ such that p is the union of two elements p' and p'' in P_i and p occurs in every P_j with $i < j \leq d-1$. Then the derivation obtained from $\mathcal{P}$ after replacing p with p' and p'' in every level j with $i < j \leq d-1$ satisfies $|P_{d-1}| = 3$. Hence, for the remainder of the proof we will assume that $|P_{d-1}| = 3$.

We claim that $\mathcal{B} := (T, \gamma)$ with T and γ as defined below is a branch decomposition of H of width at most w and depth at most d. The tree T contains one node t_p for every $p \in \bigcup_{1 \leq i \leq d} P_i$ and T has an edge between t_p and $t_{p'}$ if and only if there is an i with $1 \leq i < d$ such that $p \in P_i$, $p' \in P_{i+1}$, and $p \subsetneq p'$. Moreover, the bijection γ is defined by setting $\gamma(t_l) = l$ for every $l \in P_1$. It is straightforward to verify that $\mathcal{B}$ is indeed a branch decomposition of H with width at most w and depth at most d. □

One important parameter influencing the size of the encoding for the existence of a derivation is the length of the derivation. The next theorem shows a tight upper bound on the length of any derivation obtained from some branch decomposition. Observe that a simple caterpillar (i.e., a path where each inner vertex has one additional "pending" neighbor) shows that the bound given below is tight. The main observations behind the following theorem are that every branch decomposition has depth at most $\lfloor |E(H)|/2 \rfloor$ and moreover one can further reduce the depth of the branch decomposition by replacing small subtrees at the bottom of the branch decomposition, i.e., subtrees for which no edge has maximum width, with complete binary subtrees of smaller depth.

Theorem 4.2. *Let H be a hypergraph, e the maximum size over all edges of H, and w an integer. Then the branchwidth of H is at most w if and only if H has a derivation of width at most w and length at most $\lfloor |E(H)|/2 \rfloor - \lceil w/e \rceil + \lceil \log \lfloor w/e \rfloor \rceil$.*

Proof. The backward direction of the claim follows immediately from Theorem 4.1.

Towards showing the forward direction we first show that every branch decomposition of width at most w can be transformed into a branch decomposition of the same width and whose depth is at most $\lfloor |E(H)|/2 \rfloor - \lceil w/e \rceil + \lceil \log \lfloor w/e \rfloor \rceil$. The claim then follows from Theorem 4.1.

Let $\mathcal{B} := (T, \gamma)$ be a branch decomposition of H of width at most w. Because T is a ternary tree with exactly $|E(H)|$ leaves, we obtain that its radius is at most $\lfloor |E(H)|/2 \rfloor$. Assume in the following that T is rooted in one of the (at most two) center vertices, say r, of T. The main idea to obtain the exact bound on the radius of T given in the statement of

the theorem is now to replace every subtree of T rooted at some node, say t, that contains at most $\lfloor w/e \rfloor$ edges of H and whose height is maximal with respect to this property with a binary tree (containing the same leaf nodes) of height at most $\lceil \log\lfloor w/e \rfloor \rceil$. Because every edge in the obtained binary tree has width at most $(w/e)e = w$, this replacement does not increase the width of $\mathcal{B}$ and it is straightforward to verify that the depth of the obtained branch decomposition is at most $\lfloor |E(H)|/2 \rfloor - \lceil w/e \rceil + \lceil \log\lfloor w/e \rfloor \rceil$. ☐

4.4.1 Encoding

Let H be a hypergraph with m edges and n vertices, and let w and d be positive integers. We will assume that the vertices of H are represented by the numbers from 1 to n and the edges of H by the numbers from 1 to m. The aim of this section is to construct a formula $F(H, w, d)$ that is satisfiable if and only if H has derivation of width at most w and length at most d. Because of Theorem 4.2 (after setting d to the value specified in the theorem) it holds that $F(H, w, d)$ is satisfiable if and only if H has branchwidth at most w. To achieve this aim we first construct a formula $F(H, d)$ that is satisfiable if and only if H has a derivation of length at most d and then we extend this formula by adding constrains that restrict the width of the derivation to w.

Encoding of a Derivation of a Hypergraph

The formula $F(H, d)$ uses the following variables. A *set variable* $\mathrm{s}(e, f, i)$, for every $e, f \in E(H)$ with $e < f$ and every i with $0 \leq i \leq d$. Informally, $\mathrm{s}(e, f, i)$ is true whenever e and f are contained in the same set at level i of the derivation. A *leader variable* $\mathrm{l}(e, i)$, for every $e \in E(H)$ and every i with $0 \leq i \leq d$. Informally, the leader variables will be used to uniquely identify the sets at each level of a derivation, i.e., $\mathrm{l}(e, i)$ is true whenever e is the smallest edge in a set at level i of the derivation.

We now describe the clauses of the formula. The following clauses ensure *(D1)* and that the derivation is a sequence of refinements.

$$(\neg \mathrm{s}(e, f, 0)) \wedge (\mathrm{s}(e, f, d)) \wedge (\neg \mathrm{s}(e, f, i) \vee \mathrm{s}(e, f, i+1)) \qquad \text{for } e, f \in E(H),\ e < f,\ 1 \leq i < d.$$

The following clauses ensure that the relation of being in the same set is transitive.

$$\begin{aligned}
&(\neg \mathrm{s}(e, f, i) \vee \neg \mathrm{s}(e, g, i) \vee \mathrm{s}(f, g, i)) \\
&\wedge(\neg \mathrm{s}(e, f, i) \vee \neg \mathrm{s}(f, g, i) \vee \mathrm{s}(e, g, i)) \\
&\wedge(\neg \mathrm{s}(e, g, i) \vee \neg \mathrm{s}(f, g, i) \vee \mathrm{s}(e, f, i)) \qquad \text{for } e, f, g \in E(H),\ e < f < g,\ 1 \leq i \leq d.
\end{aligned}$$

The following clauses ensure that $\mathrm{l}(e, i)$ is true if and only if e is the smallest edge contained in some set at level i of a derivation.

$$\underbrace{(\mathrm{l}(e,i) \vee \bigvee_{f \in E(H), f<e} \mathrm{s}(f,e,i))}_{A} \wedge \underbrace{\bigwedge_{f \in E(H), f<e} (\neg \mathrm{l}(e,i) \vee \neg \mathrm{s}(f,e,i))}_{B} \qquad \text{for } e \in E(H),\ 1 \leq i \leq d.$$

Part A ensures that e is a leader or it is in a set with an edge which is smaller than e; part B ensures that if e is not in same set with any smaller edge then it is a leader. The following clauses ensure that at most two sets in the partition at level i can be combined into a set in the partition at level $i+1$, i.e., together with the clauses above it ensures *(D2)*.

$$\neg \mathrm{l}(e,i) \vee \neg \mathrm{l}(f,i) \vee \neg \mathrm{s}(e,f,i+1) \vee \mathrm{l}(e,i+1) \vee \mathrm{l}(f,i+1) \qquad \text{for } e,f \in E(H),\ e<f,\ 1 \leq i < d-1.$$

The following clauses ensure that at most three sets in the partition at level $d-1$ can be combined into a set in the partition at level d, i.e., together with the clauses above it ensures *(D3)*.

$$\neg \mathrm{l}(e,d-1) \vee \neg \mathrm{l}(f,d-1) \vee \neg \mathrm{l}(g,d-1) \vee \neg \mathrm{s}(e,f,d) \vee \neg \mathrm{s}(e,g,d) \vee \mathrm{l}(e,d) \vee \mathrm{l}(f,d) \vee \mathrm{l}(g,d) \qquad \text{for } e,f,g \in E(H),\ e<f<g.$$

All of the above clauses together ensure *(D1)*, *(D2)*, and *(D3)*. We also add the following redundant clauses.

$$\mathrm{l}(e,i) \vee \neg \mathrm{l}(e,i+1) \qquad \text{for } e \in E(H),\ 1 \leq i < d.$$

These clauses use the observation that if an edge is not a leader at level i then it cannot be a leader at level $i+1$. The formula $F(H,d)$ contains at most $\mathcal{O}(m^2 d)$ variables and $\mathcal{O}(m^3 d)$ clauses.

Encoding of a Derivation of Bounded Width

Next we describe how $F(H,d)$ can be extended to restrict the width of the derivation. The main idea is to first identify the set of cut vertices for the sets in the derivation and then restrict their sizes. To this end we first need to introduce new variables (and later clauses), which allow us to identify cut vertices of edge sets in the derivation. In particular, we introduce a *cut variable* $\mathrm{c}(e,u,i)$ for every $e \in E(H)$, $u \in V(H)$ and i with $1 \leq i \leq d$. Informally, $\mathrm{c}(e,u,i)$ is true if u is a cut vertex of the set containing e at level i of the derivation. Once we have assigned the c variables, we use the sequential cardinality counters (Section 2.1.1) to bound the number of true $\mathrm{c}(e,u,i)$ variables, for a leader $e \in E$ and for all $u \in V$. In order to restrict the size of the sets of cut vertices later on we do not need the backward direction of the previous statement. Recall that a

vertex u is a cut vertex for some set p of the derivation if there are two distinct edges incident to u such that one of them is contained in p and the other one is not.

In the following we will present an encoding that has turned out to give the best results in our case. The main idea behind the encoding is to only define the variables $\mathrm{c}(e,u,i)$ for the leading edges e in the current derivation.

The following clauses ensure that whenever two edges incident to a vertex are not in the same set at level i of the derivation, then the vertex is a cut vertex for every leading edge of the sets containing the incident edges.

$$\neg\mathrm{l}(e,i) \vee \mathrm{c}(e,u,i) \vee \mathrm{s}(\min\{e,f\},\max\{e,f\},i) \vee \neg\mathrm{s}(\min\{e,g\},\max\{e,g\},i)$$
$$\text{for } e,f,g \in E(H),\ e \neq f,\ e \neq g,\ u \in V(H),\ u \in f,\ u \in g,\ 1 \leq i \leq d.$$

$$\neg\mathrm{l}(e,i) \vee \mathrm{s}(\min\{e,f\},\max\{e,f\},i) \vee \mathrm{c}(e,u,i)$$
$$\text{for } e,f \in E(H),\ e \neq f,\ u \in V(H),\ u \in e,\ u \in f,\ 1 \leq i \leq d$$

Additionally, we add the following redundant clauses that ensure the "monotonicity" of the cut vertices, i.e., if u is a cut vertex for a set at level i and for the corresponding set at level $i+2$, then it also has to be a cut vertex at level $i+1$.

$$\neg\mathrm{l}(e,i) \vee \neg\mathrm{l}(e,i+1) \vee \neg\mathrm{l}(e,i+2) \vee \neg\mathrm{c}(e,u,i) \vee \neg\mathrm{c}(e,u,i+2) \vee \mathrm{c}(e,u,i+1))$$
$$\text{for } e \in E(H),\ u \in V(H),\ 1 \leq i \leq d-2.$$

The definition of cut vertices adds at most $\mathcal{O}(mnd)$ variables and at most $\mathcal{O}(m^3nd)$ clauses.

After adding the clauses for the sequential counter, we complete the construction of the formula $F(H,w,d)$. In total $F(H,w,d)$ has at most $\mathcal{O}(m^2d + mndw) \subseteq \mathcal{O}(m^3 + m^2n^2)$ variables and at most $\mathcal{O}(m^3nd + mndw) \subseteq \mathcal{O}(m^4n + m^2n^2)$ clauses. By construction, $F(H,w,d)$ is satisfiable if and only H has a derivation of width at most w and length at most d. Because of Theorem 4.1, we obtain the following.

Theorem 4.3. *The formula $F(H,w,d)$ is satisfiable if and only if H has a branch decomposition of width at most w and depth at most d. Moreover, a corresponding branch decomposition can be constructed from a satisfying assignment of $F(H,w,d)$ in time that is linear in the number of variables of $F(H,w,d)$.*

4.5 Carving-Width

In this section we introduce our encoding for carving-width. Carving-width is a decompositional parameter that is closely related to branchwidth and has been introduced by Seymour and Thomas [ST94]. They also showed that given a graph G and an integer k, deciding whether G has carving-width $\leq k$ is NP-complete, but can be decided in

polynomial-time if G is planar. If k is a constant and not part of the input, then it can be decided in linear time whether G has carving-width $\leq k$ [TSB00]. Interestingly, the known polynomial-time algorithm for computing the branchwidth of a planar graph is based on the corresponding algorithm for carving-width and uses the fact that the carving-width of the so-called medial graph of a planar graph is exactly twice the branchwidth of the original graph.

Carving decompositions (or simply *carvings*) are defined very similarly to branch decompositions with two important differences: (1) the leaves of a carving are in correspondence to the vertices instead of the edges of the hypergraph and (2) the "width" of an edge (and in consequence the width of a carving) is measured in terms of the number of edges of the hypergraph with at least one endpoint in both components of the carving decomposition obtained after deleting the edge.

Let $H = (V, E)$ be a hypergraph and $V' \subseteq V$. We denote by $\delta(V')$ the set of edges $e \in E$ that have at least one endpoint in V' and outside of V', i.e., $e \cap V' \neq \emptyset$ and $e \setminus V' \neq \emptyset$. A *carving* $\mathcal{C}(H)$ of a hypergraph $H = (V, E)$ is a pair (T, γ), where T is a ternary tree and $\gamma : L(T) \to V$ is a bijection between the vertices of H and the leaves of T (denoted by $L(T)$). For simplicity, we write $\gamma(L)$ to denote the set $\{\, \gamma(l) \mid l \in L \,\}$ for a set of leaves L of T and we also write $\delta(T')$ instead of $\delta(\gamma(L(T')))$ for a subtree T of T'. For an edge e of T, we denote by $\delta_{\mathcal{C}}(e)$ (or simply $\delta(e)$ if $\mathcal{C}$ is clear from the context), the set of *cut edges* of e, i.e., the set $\delta(T')$, where T' is any of the two components of $T \setminus \{e\}$. The *width* of an edge e of T is the number of cut edges of e, i.e., $|\delta_{\mathcal{C}}(e)|$ and the *width* of $\mathcal{C}$ is the maximum width of any edge of T. The *carving-width* of H is the minimum width over all carvings of H (or 0 if $|V(H)| = 1$ and H has no carving). We also define the *depth* of $\mathcal{C}$ as the radius of T. Fig. 4.4 illustrates a carving of a small hypergraph.

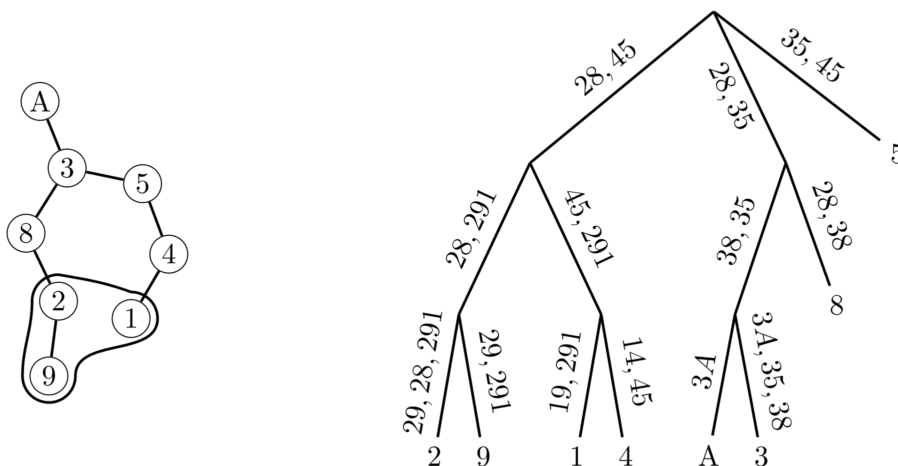


Figure 4.4: A hypergraph H (left) and an optimal carving (T, γ) of H (right). The labels of the leaves of T are the vertices assigned to them by γ and the labels of the edges of T are the cut edges of that edge.

4.5.1 Partition-based Reformulation of Carving-width

In this section, we will describe the new partition-based characterization for the carvings and the SAT-encoding associated with this new characterization.

Let $H = (V, E)$ be a hypergraph. A *carving derivation* $\mathcal{P}$ of H of *length* l is a sequence $(P_1, \dots, P_l)$ of partitions of V such that:

(D1) $P_1 = \{\, \{v\} \mid v \in V \,\}$ and $P_l = \{V\}$ and

(D2) for every $i \in \{1, \dots, l-2\}$, P_i is a 2-ary refinement of P_{i+1} and

(D3) P_{l-1} is a 3-ary refinement of P_l.

The *width* of $\mathcal{P}$ is the maximum size of $\delta_H(V)$ over all sets $V \in \bigcup_{1 \leq i < l} P_i$. We will refer to P_i as the i-th *level* of the carving derivation $\mathcal{P}$ and we will refer to elements in $\bigcup_{1 \leq i \leq l} P_i$ as *sets* of the carving derivation. We will show that any carving can be transformed into a carving derivation of the same width and also the other way around. The following example illustrates the close connection between carvings and carving derivations.

Example 6. *Consider the carving decomposition $\mathcal{C}$ given in Fig. 4.4. Then $\mathcal{C}$ can, e.g., be translated into the derivation $\mathcal{P} = (P_1, \dots, P_5)$ defined by:*

$$P_1 = \Big\{\{2\}, \{9\}, \{1\}, \{4\}, \{A\}, \{3\}, \{8\}, \{5\}\Big\}$$

$$P_2 = \Big\{\{2, 9\}, \{1, 4\}, \{A, 3\}, \{8\}, \{5\}\Big\}$$

$$P_3 = \Big\{\{2, 9, 1, 4\}, \{A, 3, 8\}, \{5\}\Big\}$$

$$P_4 = \Big\{\{2, 9, 1, 4, A, 3, 8, 5\}\Big\}$$

The width of $\mathcal{C}$ is equal to the width of $\mathcal{P}$.

The following theorem shows that derivations provide an alternative characterization of carving decompositions. Since the proof uses the same construction and is also otherwise very similar to the proof of Theorem 4.1.

Theorem 4.4. *Let H be a hypergraph and w and d two integers. H has a carving of width at most w and depth at most d if and only if H has a carving derivation of width at most w and length at most d.*

As in the case of branchwidth it will be beneficial for our encoding to obtain a tight bound on the length of a carving derivation. The next theorem shows a tight upper bound on the length of any carving derivation obtained from some carving decomposition. The main ideas are similar to the ideas used for branch decompositions (Theorem 4.2),

however, there are some subtle differences. Observe that a simple caterpillar (i.e., a path where each inner vertex has one additional "pending" neighbor) shows that the bound given below is tight.

Theorem 4.5. *Let H be a hypergraph with maximum degree Δ, and w an integer. Then the carving-width of H is at most w if and only if H has a carving derivation of width at most w and length at most $\lfloor |V(H)|/2 \rfloor - \lceil w/\Delta \rceil + \lceil \log \lfloor w/\Delta \rfloor \rceil$.*

Proof. The backward direction of the claim follows immediately from Theorem 4.4.

Towards showing the forward direction we first show that every carving of width at most w can be transformed into a carving of the same width and whose depth is at most $\lfloor |V(H)|/2 \rfloor - \lceil w/\Delta \rceil + \lceil \log \lfloor w/\Delta \rfloor \rceil$. The claim then follows from Theorem 4.4.

Let $\mathcal{C} := (T, \gamma)$ be a carving of H of width at most w. Because T is a ternary tree with exactly $|V(H)|$ leaves, we obtain that its radius is at most $\lfloor |V(H)|/2 \rfloor$. Assume in the following that T is rooted in one of the (at most two) center vertices, say r, of T. The main idea to obtain the exact bound on the radius of T given in the statement of the theorem is now to replace every subtree of T rooted at some node, say t, that contains at most $\lfloor w/\Delta \rfloor$ vertices of H and whose height is maximal with respect to this property with a binary tree (containing the same leaf nodes) of height at most $\lceil \log \lfloor w/\Delta \rfloor \rceil$. Because every edge in the obtained binary tree has width at most $(w/\Delta)\Delta = w$, this replacement does not increase the width of $\mathcal{C}$ and it is straightforward to verify that the depth of the obtained carving decomposition is at most $\lfloor |V(H)|/2 \rfloor - \lceil w/\Delta \rceil + \lceil \log \lfloor w/\Delta \rfloor \rceil$. $\square$

4.5.2 Encoding

The encoding for carving derivations is very similar (actually almost identical) to the encoding we presented for branch decompositions in Section 4.4.1. In particular, one can use the exact same encoding for the formulas $F(H, d)$ and $F(H, w, d)$ as for derivations after switching the role that the vertices and edges of the hypergraph play in the encoding. For instance, the set variables $\mathrm{s}(e, f, i)$ that were defined for all edges $e, f \in E(H)$ with $e < f$ in the encoding for branchwidth, will now be defined for all vertices $e, f \in V(H)$ with $e < f$. Similarly, the cut variables $\mathrm{c}(e, u, i)$ that were defined for all edges $e \in E(H)$ and vertices $u \in V(H)$, will now be defined for all vertices $e \in V(H)$ and all edges $u \in E(H)$.

4.6 Local Improvement for Branch Decompositions

The encoding presented in Section 4.4.1 allows us to compute the exact branchwidth of hypergraphs up to a certain size. Due to the intrinsic difficulty of the problem, one can hardly hope to go much further beyond this size barrier with an exact method. In this section we therefore propose a local improvement approach that employs our SAT-encoding to improve small parts of an heuristically obtained branch decomposition. Our local improvement procedure can be seen as a kind of local search procedure that at

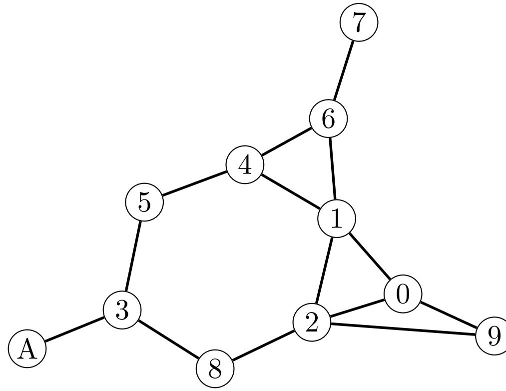


Figure 4.5: The graph H used to illustrate the main idea behind our local improvement procedure.

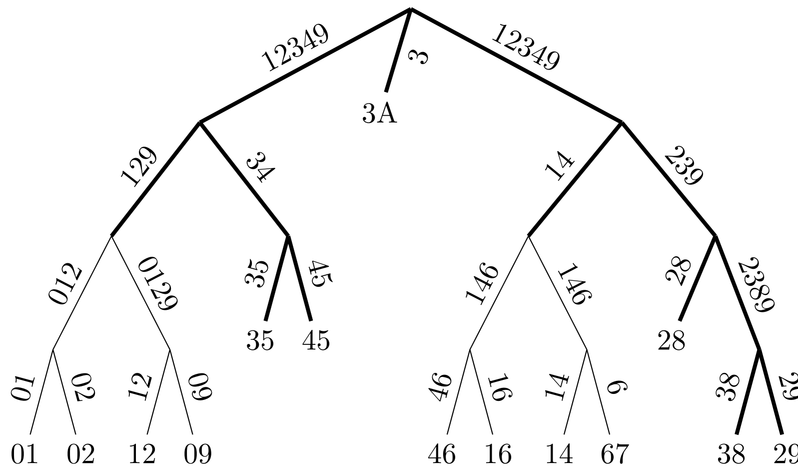


Figure 4.6: A branch decomposition $\mathcal{B}$ of the graph H given in Fig. 4.5 together with an example of a local branch decomposition $\mathcal{B}_L$ (highlighted by thicker edges) chosen by our algorithm.

each step tries to replace a part of the branch decomposition with a better decomposition found by means of the SAT-encoding and repeats this process until a fixed-point (or timeout) is reached.

Let H be a hypergraph and $\mathcal{B} := (T, \gamma)$ a branch decomposition of H. For a connected ternary subtree T_L of T we define the *local branch decomposition* $\mathcal{B}_L := (T_L, \gamma_L)$ of $\mathcal{B}$ by setting $\gamma_L(l) = \delta_{\mathcal{B}}(e)$ for every leaf $l \in L(T_L)$, where e is the (unique) edge incident to l in T_L. We also define the hypergraph $H(T_L)$ as the hypergraph that has one hyperedge $\gamma_L(l)$ for every leaf l of T_L and whose vertices are defined as the union of all these edges. We observe that $\mathcal{B}_L$ is a branch decomposition of $H(T_L)$. The main idea behind our approach, which we will formalize below, is that we can obtain a new branch decomposition of H

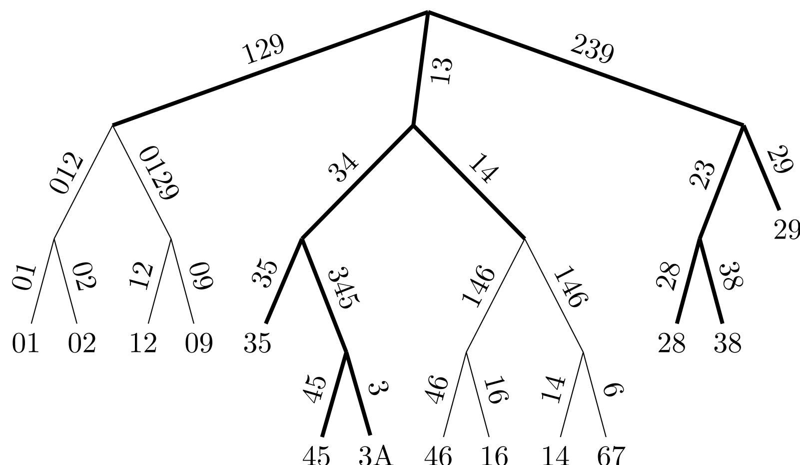


Figure 4.7: The improved branch decomposition $\mathcal{B}'$ obtained from $\mathcal{B}$ after replacing the local branch decomposition $\mathcal{B}_L$ of $H(T_L)$ with an optimal branch decomposition $\mathcal{B}'_L$ of $H(T_L)$ obtained from our SAT-encoding. See Fig. 4.6 for an illustration of $\mathcal{B}$ and $\mathcal{B}_L$.

by replacing the part of $\mathcal{B}$ formed by $\mathcal{B}_L$ with any branch decomposition of $H(T_L)$. In particular, by replacing $\mathcal{B}_L$ with a branch decomposition of $H(T_L)$ of lower width, we will potentially improve the branch decomposition $\mathcal{B}$. This idea is illustrated in Fig. 4.6 and Fig. 4.7.

A general outline of our algorithm is given in Algorithm 4.1. The algorithm uses two global parameters: `globalbudget` gives an upper bound on the size of the local branch decomposition and the function `length(H,w)`, which is only used by the function SATSolve explained below, provides an upper bound on the length of a derivation which will be considered by our SAT-encoding.

Given a hypergraph H, the algorithm first computes a (not necessarily optimal) branch decomposition $\mathcal{B} := (T, \gamma)$ of H using, e.g., the heuristics from [CS03, Hic02]. The algorithm then computes the set M of maximum cut edges of T, i.e., the set of edges e of T with $|\delta(e)| = w$, where w is the width of $\mathcal{B}$. It then computes the set $\mathcal{C}$ of components of $T[M]$, where $T[M]$ is the forest with vertex set $V(T)$ and edge set M, and for every component $C \in \mathcal{C}$ it calls the function `LocalBD` to obtain a local branch decomposition $\mathcal{B}_L := (T_L, \gamma_L)$ of $\mathcal{B}$, which contains (at least) all the edges of C. The function `LocalBD` is given in Algorithm 4.3 and will be described later. Given $\mathcal{B}_L$ the algorithm tries to compute a branch decomposition $\mathcal{B}'_L := (T'_L, \gamma'_L)$ of $H(T_L)$ with smaller width than $\mathcal{B}_L$ using the function `ImproveLD`, which is described later. If successful, the algorithm updates $\mathcal{B}$ by replacing the part of $\mathcal{B}$ represented by T_L with $\mathcal{B}'_L$ according to Theorem 4.6 and proceeds with line 4. If on the other hand $\mathcal{B}_L$ cannot be improved, the algorithm proceeds with the next component C of $T[M]$. This process is repeated until none of the components C of $T[M]$ lead to an improvement.

Algorithm 4.1: Local Improvement

```
    input  : A hypergraph H
    output : A branch decomposition of H
 1  𝓑 ← BDHeuristic(H) // (𝓑 := (T, γ))
 2  improved ← true
 3  while improved do
 4  |   M ← "the set of edges e of 𝓑 whose width (|δ_𝓑(e)|) is maximum"
 5  |   𝓒 ← "the set of components of T[M]"
 6  |   improved ← false
 7  |   for C ∈ 𝓒 do
 8  |   |   𝓑_L ← LocalBD (𝓑, C)
 9  |   |   𝓑'_L ← ImproveLD (𝓑_L)
10  |   |   if 𝓑'_L ≠ NULL then
11  |   |   |   𝓑 ← Replace (𝓑, 𝓑_L, 𝓑'_L)
12  |   |   |   improved ← true
13  |   |   end
14  |   |   else
15  |   |   |   break
16  |   |   end
17  |   end
18  end
19  return B
```

The function LocalBD, which is given in Algorithm 4.3, computes a local branch decomposition $\mathcal{B}_L := (T_L, \gamma_L)$ of $\mathcal{B}$ that contains at least all edges in the component C and which should be small enough to ensure solvability by our SAT-encoding as follows. In the beginning T_L is set to the connected ternary subtree of T obtained from $T[C]$ after adding the (unique) third neighbor of any vertex v of C that has degree exactly two in $T[C]$. It then proceeds by processing the (current) leaves of T_L in a breadth first search manner, i.e., in the beginning all the leaves of T_L are put in a first-in first-out queue Q. If l is the current leaf of T_L, which is not a leaf of T, the algorithm adds the two additional neighbors of l in T to T_L and adds them to Q. It proceeds in this manner until the number of edges in T_L reaches the global budget.

The function ImproveLD tries to compute a branch decomposition of $H(T_L)$ with lower width than $\mathcal{B}_L$ using our SAT-encoding. In particular, if the size of T_L does not exceed the global budget (in which case it would be highly unlikely that a lower width branch decomposition can be found using our SAT-encoding), the function calls the function SATSolve with decreasing widths w until SATSolve does not return a branch decomposition any more. Here, the function SATSolve uses the formula $F(H(T_L), w, d)$ from Theorem 4.3 with d set to length(H,w) to test whether $H(T_L)$ has a branch decomposition of width at most w and depth at most d. If so (and if the SAT-solver

Algorithm 4.2: ImproveLD

```
    input  : A branch decomposition B_L := (T_L, γ_L) of H(T_L)
    output : An "improved" branch decomposition of H(T_L)
 1  if |T_L| >globalbudget then
 2  |  return NULL
 3  end
 4  w ← "the width of B_L"
 5  repeat
 6  |  B_D ← SATSolve(H(T_L), w)
 7  |  if B_D ≠ NULL then
 8  |  |  B'_L ← B_D
 9  |  end
10  |  w ← w − 1
11  until B_D == NULL
12  return B'_L
```

solves the formula within a predefined timeout) SATSolve returns the corresponding branch decomposition; otherwise it returns NULL.

Last but not least the function `Replace` replaces the part of $\mathcal{B}$ represented by $\mathcal{B}_L$ with the new branch decomposition $\mathcal{B}'_L$ according to Theorem 4.6.

Let H be a hypergraph, $\mathcal{B} := (T, \gamma)$ a branch decomposition of H, T_L a connected ternary subtree of T, $\mathcal{B}_L := (T_L, \gamma_L)$ be the local branch decomposition of $\mathcal{B}$ corresponding to T_L, and let $\mathcal{B}'_L := (T'_L, \gamma')$ be any branch decomposition of $H(T_L)$. Note that because $\mathcal{B}_L$ and $\mathcal{B}'_L$ are branch decompositions of the same hypergraph $H(T_L)$, we obtain from Proposition 4.1 that we can assume that $V(T_L) = V(T'_L)$ and $\gamma = \gamma'$. We define the *locally improved* branch decomposition, denoted by $\mathcal{B}(\frac{\mathcal{B}_L}{\mathcal{B}'_L})$, to be the branch decomposition obtained from $\mathcal{B}$ by replacing the part corresponding to $\mathcal{B}_L$ with $\mathcal{B}'_L$, i.e., the tree of $\mathcal{B}'$ is obtained from T by removing all edges of T_L from T and replacing them with the edges of T'_L and the bijection of $\mathcal{B}'$ is equal to γ.

Theorem 4.6. *$\mathcal{B}(\frac{\mathcal{B}_L}{\mathcal{B}'_L})$ is a branch decomposition of H, whose width is the maximum of the width of $\mathcal{B}'_L$ and the maximum width over all edges $e \in E(T) \setminus E(T_L)$ in $\mathcal{B}$.*

Proof. It is easy to verify that $\mathcal{B}(\frac{\mathcal{B}_L}{\mathcal{B}'_L})$ is indeed a branch decomposition of H.

Towards showing that the width of $\mathcal{B}(\frac{\mathcal{B}_L}{\mathcal{B}'_L})$ is equal to the maximum of the width of $\mathcal{B}'_L$ and the maximum width of any edge $e \in E(T) \setminus E(T_L)$ in $\mathcal{B}$, we first give an alternative definition for γ_L.

Let F be the forest obtained from T after deleting all edges of T_L, i.e., F is the forest $T \setminus E(T_L)$. Then every leaf of T_L and also every leaf of T is contained in exactly one

Algorithm 4.3: Local Selection (LocalBD)

input : A branch decomposition $\mathcal{B} := (T, \gamma)$ of H and a component C of T

output : A local branch decomposition of $\mathcal{B}$

```
1  w ← "the width of B"
2  T_L ← C
3  for c ∈ V(C) with deg_C(c) = 2 do
4  |   "add the unique third neighbor and its edge incident to c to T_L"
5  end
6  Q ← "the set of leaves of T_L"
7  while Q ≠ ∅ and |T_L| ≤ globalbudget −2 do
8  |   l ← Q.pop()
9  |   if "l is not a leaf of T" then
10 |   |   c, c' ← "the two neighbors of l in T which are not neighbors of l
   |   |     in T_L"
11 |   |   if δ_B({l, c}) < w and δ_B({l, c'}) < w then
12 |   |   |   "add c and c' together with their edges incident to l to T_L"
13 |   |   |   Q.push(c)
14 |   |   |   Q.push(c')
15 |   |   end
16 |   end
17 end
18 return "the local branch decomposition of B represented by T_L"
```

component of F. Moreover, because T is a tree every component of F contains at most one leaf of T_L. Let $\text{Lcut} : L(T_L) \to L(T)$ be the mapping that assigns to every leaf l of T the set of all leaves of T that are contained in the same component as l in F. Note that Lcut naturally associates every leaf l of T_L to the cut $(\gamma(\text{Lcut}(l)), E(H) \setminus \gamma(\text{Lcut}(l)))$. Also, $\delta_H(\gamma(\text{Lcut}(l))) = \delta_{\mathcal{B}}(e)$ for every $l \in L(T_L)$, where e is the (unique) edge in T_L incident to l. Hence, in the following we will assume that $\gamma_L(l)$ is equal to $\delta_H(\gamma(\text{Lcut}(l)))$.

We are now ready to prove the statement of the theorem concerning the width of $\mathcal{B}(\frac{\mathcal{B}_L}{\mathcal{B}'_L})$. Observe that it is sufficient to show that for every edge e of T' either $\delta_{\mathcal{B}'}(e) = \delta_{bd}(e)$ if $e \in E(T') \setminus E(T'_L)$ or $\delta_{\mathcal{B}'}(e) = \delta_{\mathcal{B}'_L}(e)$ if $e \in E(T'_L)$. Towards showing the former case, let $e \in E(T') \setminus E(T'_L)$. Because T_L and T'_L are connected the components of $T \setminus \{e\}$ are the same as the components of $T' \setminus \{e\}$ for every such edge e. Hence, $\delta_{\mathcal{B}'}(e) = \delta_{\mathcal{B}}(e)$, as required.

Towards showing the later case, let $e \in E(T'_L)$ and let C_1 and C_2 be the two components of $T' \setminus \{e\}$.

We start by showing that $\delta_{\mathcal{B}'}(e) \subseteq \delta_{\mathcal{B}'_L}(e)$. Because $v \in \delta_{\mathcal{B}'}(e)$, we obtain that there are two leaves l_1 and l_2 of T' with $l_1 \in V(C_1)$ and $l_2 \in V(C_2)$ such that $v \in \gamma'(l_1) \cap \gamma'(l_2)$.

Observe that $f(l_1) \in V(C_1)$ and $f(l_2) \in V(C_2)$ and hence $v \in \gamma'_L(f(l_1))$ and $v \in \gamma'_L(f(l_2))$. Consequently, $v \in \delta_{\mathcal{B}'_L}(e)$. This shows that $\delta_{\mathcal{B}'}(e) \subseteq \delta_{\mathcal{B}'_L}(e)$ and it remains to show that $\delta_{\mathcal{B}'_L}(e) \subseteq \delta_{\mathcal{B}'}(e)$. Because $v \in \delta_{\mathcal{B}'_L}(e)$, we obtain that there are two leaves l_1 and l_2 of T'_L with $l_1 \in V(C_1)$ and $l_2 \in V(C_2)$ such that $v \in \gamma'_L(l_1) \cap \gamma'_L(l_2)$. Because $v \in \gamma'_L(l_1)$, we obtain that there is a leaf $l'_1 \in f^{-1}(l_1)$ such that $v \in \gamma'(l'_1)$. Similarly, because $v \in \gamma'_L(l_2)$, we obtain that there is a leaf $l'_2 \in f^{-1}(l_2)$ such that $v \in \gamma'(l'_2)$. Note that because $l'_1 \in f^{-1}(l_1)$ it holds that $l'_1 \in V(C_1)$ and similarly because $l'_2 \in f^{-1}(l_2)$ it holds that $l'_2 \in V(C_2)$. Hence, $v \in \delta_{\mathcal{B}'}(e)$, which completes the proof of the theorem. □

4.7 Local Improvement for Carving Decompositions

The local improvement approach introduced in the previous section for branchwidth can also be employed for carving-width in a very similar manner. Namely if $\mathcal{C}(H) = (T, \gamma)$ is a carving of a hypergraph $H = (V, E)$ and T_L is a connected ternary subtree of T, we can define the *local carving* $\mathcal{C}_L = (T_L, \gamma_L)$ of $\mathcal{C}$ by setting $\gamma_L(l) = \gamma(L(T^l))$ for every leaf $l \in L(T_L)$, where T^l is the unique subtree of T' containing l and T' is the subgraph of T obtained after deleting all edges in T_L from T. Note that (T_L, γ_L) is strictly speaking not a carving of H since its leaves are assigned to subsets of vertices instead of single vertices. Since (T_L, γ_L) only partially decomposes H, i.e., it does not decompose the subsets of vertices assigned to its leaves, we call it a *partial carving*. One can now show, in a very similar manner as for branch decompositions, that any *partial carving* of H, whose leaves correspond to the same subsets of $V(H)$ as the leaves in (T_L, γ_L), can be used to replace (T_L, γ_L) in (T, γ) to obtain a carving of H with potentially smaller width. Moreover, finding a partial carving of smaller width can be achieved by employing almost the same encoding as introduced in Section 4.5. In particular, one merely needs to adapt Property (D1) of carving derivations (see Subsection 4.5.1) to ensure that the initial partition of a derivation is equal to the partition of $V(H)$ given by the leaves of (T_L, γ_L). Hence the local improvement approach for branchwidth can be easily adapted to carvings with one exception: For the local improvement approach to work it is crucial that one can obtain an initial carving very efficiently, e.g., via a heuristic methods as in the case of branch decompositions. Unfortunately, we are not aware of any suitable heuristic method for the computation of carvings and have therefore refrained from implementing the local improvement approach for carvings.

4.8 Experimental Results

We have implemented the single SAT-encoding for branchwidth and carving-width and the SAT-based local improvement method for branchwidth and tested them on various benchmark instances, including famous named graphs from the literature [Wei16], graphs from TreewidthLIB [Bod16] which origin from a broad range of applications, and a series of circular clusters [Cor01] which are hypergraphs denoted C^e_v with v vertices and v edges of size e. For these experiments we used the SAT-solver Glucose 4.0 (with standard

parameter setting) as it performed best in our initial tests compared to other solvers such as GlueMiniSat 2.2.8, Lingeling, and Riss 4.27.

4.8.1 Single SAT-encoding

To determine the branchwidth or carving-width of a graph or hypergraph with our encodings, one could either start from $w = 1$ and increase w until the formula becomes satisfiable, or by setting w to an upper bound on the width obtained by a heuristic method, and decrease it until the formula becomes unsatisfiable. For both approaches the solving time at the threshold (i.e., for the largest w for which the formula is unsatisfiable) is, as one would expect, by far the longest. Table 4.1 shows this behavior on some typical instances. Hence whether we determine the width from below or from above does not matter much. A more elaborate binary search strategy could save some time, but overall the expected gain is little compared to the solving time at the threshold.

Table 4.1: Distribution of solving time in seconds for various values of w for some famous named graphs of branchwidth 6.

w	2	3	4	5	6	7	8	9	10
Graph	unsat	unsat	unsat	unsat	sat	sat	sat	sat	sat
FlowerSnark	1.2	4.4	25.5	889.9	1.6	1.3	1.6	1.3	1.3
Errera	5.7	22.7	79.4	1530.9	12.0	7.3	6.7	5.4	6.1
Folkman	3.4	13.7	98.6	2747.0	6.1	5.3	3.7	3.8	5.2
Poussin	3.3	9.2	68.7	941.2	4.5	3.5	3.9	2.9	3.4

The solving time varies and depends on the structure of the (hyper)graph. We could determine the exact branchwidth and carving-width of many famous graphs known from the literature, see Table 4.2. For many of the graphs the exact branchwidth or carving-width has not been known before.

We also tested the encodings for the circular cluster hypergraphs C^i_{2i-1}. We were able to find the exact branchwidth for those instances up to $i = 26$ and the exact carving-width for instances up to $i = 16$ using a timeout of 2000 seconds.

We verified the correctness of our encoding for branchwidth by comparing the widths computed by our method with the widths computed by Hick's [Hic05] tangles-based algorithm. For carving-width, we are not aware of any other implemented algorithm, but as a sanity check we used the fact that the carving-width of the medial graph of a planar graph is exactly two times the branchwidth of the graph [ST94], and we tested this for a number of planar graphs.

Table 4.2: Exact branchwidth and carving-width of the famous named graphs.

Graph	$\lvert V\rvert$	$\lvert E\rvert$	branchwidth	carving-width
Watsin	50	75	6	6
Kittell	23	63	6	12
Holt	27	54	9	12
Shrikhande	16	48	8	16
Errera	17	45	6	12
Brinkmann	21	42	8	12
Clebsch	16	40	8	16
Folkman	20	40	6	12
Paley13	13	39	7	16
Poussin	15	39	6	11
Robertson	19	38	8	12
McGee	24	36	7	8
Nauru	24	36	6	8
Hoffman	16	32	6	10
Desargues	20	30	6	6
Dodecahedron	20	30	6	6
Flower Snark	20	30	6	6
Goldner-Harary	11	27	4	10
Pappus	18	27	6	6
Sousselier	16	27	5	7
Chvátal	12	24	6	8
Grötzsch	11	20	5	7
Dürer	12	18	4	4
Franklin	12	18	4	4
Frucht	12	18	3	4
Herschel	11	18	4	6
Tietze	12	18	4	5
Petersen	10	15	4	5
Pmin	9	12	3	4
Wagner	8	12	4	4
Moser spindle	7	11	3	4
Prism	6	9	3	4
Butterfly	5	6	2	4

4.8.2 SAT-Based Local Improvement

We tested our local improvement method on graphs with several thousands of vertices and edges and with initial branch decomposition of width up to above 200. In particular, we tested it on all graphs from TreewidthLIB omitting graphs that are minors of other

graphs as well as small graphs with 150 or fewer edges (small graphs can be solved with the single SAT-encoding). These are in total 740 graphs with up to 5934 vertices and 17770 edges. We ran our SAT-based local improvement algorithm on each graph with a timeout of 6 hours, where each SAT-call had a timeout of 1200 seconds and a memory limit of 8GB. We computed the initial branch decomposition by a greedy heuristic developed by Hicks [Hic05] and kindly provided to us by the author.

We conducted our experiments using different values for the budget, i.e., the parameter `globalbudget` used in Algorithm 4.1 bounding the maximum number of edges in the local hypergraph, as well as different values of depth for the derivation of the local hypergraph (the parameter `length(H,w)` used in Algorithm 4.1). Tables 4.4, 4.5, and 4.6 illustrate our experimental results for budgets between 120 and 210 and depths ranging between $m/5$ and m as well as the depth given in Theorem 4.2, where m is the number of edges in the local hypergraph. To compare the performance of our approach for different values of these two parameters, we use the following performance indicators:

- the sum of the improvement over all instances (Table 4.4),
- the maximum improvement for any of the instances (Table 4.5),
- the total number of instances whose branchwidth could be improved by at least one (Table 4.6).

For sum of improvements (Table 4.4) as well as for the total number of improved instances (Table 4.6) the best combination turned out to be a budget of 200 and a depth of $m/3$. For this combination the sum of improvements is 1483 and we improved the width of 476 out of 740 instances. With regards to the maximum improvement of any instance (Table 4.5), this combination performed well with a maximum improvement of 20; however the combination with a budget of 140 and using the optimal depth performed even better, allowing us to improve the width of an instance by 22.

In Table 4.3 we list some instances that we found particularly notable for various aspects, such as significant improvement, large number of vertices and edges, and particular low or high width of the initial branch decomposition.

4.8.3 Discussion

As discussed earlier, we are aware of only two implemented algorithms that determine the exact branchwidth of a graph or hypergraph: Hick's combinatorial algorithm based on tangles [Hic05], and Ulusal's integer programming encodings [Ulu08]. We reimplemented the integer programming encodings and compared our algorithm with this approach and the tangle based algorithm, for which we obtained the source code from the authors. Our algorithm greatly outperformed the integer programming encodings in case of both graphs and hypergraphs. For instance none of the integer programming encodings could solve the circular cluster hypergraphs C^i_{2i-1} for $i > 7$, whereas we could go up to $i = 26$.

Table 4.3: Results for SAT-based local improvement for a selection of example instances from TreewidthLIB.

Graph	$\|V\|$	$\|E\|$	Branchwidth		Diff	
			iw	fw		
bn_63-pp	426	1489	73	51	22	significant improvement
bn_51	661	2131	95	75	20	
bn_77	1020	2616	40	23	17	
rl5915.tsp	5915	17728	70	64	6	large number of edges
fl3795.tsp	3795	11326	49	42	7	
fnl4461.tsp	4461	13359	82	72	10	
bn_43-pp	254	725	23	17	6	low initial width
fl1400.tsp-pp	1390	4108	23	14	9	
vm1084.tsp-pp	808	2312	29	19	10	
graph09	458	1667	125	117	8	large initial width
graph13-wpp	427	1778	137	129	8	
pignet2-pp	1024	3774	175	173	2	

Table 4.4: Sum of improvements over all the instances for the various configurations.

budget	$m/1$	$m/2$	optimal	$m/3$	$m/4$	$m/5$
120	809	1182	1138	1200	911	670
130	1055	1249	1216	1343	1221	896
140	1043	1338	1299	1387	1291	961
150	1055	1338	1318	1454	1375	1028
160	1004	1350	1350	1453	1390	1047
170	962	1350	1352	1460	1390	1035
180	913	1322	1293	1454	1342	1033
190	934	1309	1296	1478	1401	1121
200	780	1288	1090	**1483**	1156	907
210	891	1209	1349	1395	1363	1046

The tangles-based algorithm performed better than our SAT-approach for the famous graphs. However, we could not use the tangles-based approach for our local improvement method for the following reasons:

1. The current implementation of the tangles-based approach does not support hypergraphs. Moreover, even though there is a reduction from hypergraphs to graphs conserving the branchwidth, this reduction increases both the number of vertices and the number edges significantly, which makes our approach more efficient than the tangles-based approach.

Table 4.5: Maximum improvement over all the instances for the various configurations.

budget	$m/1$	$m/2$	optimal	$m/3$	$m/4$	$m/5$
120	14	21	21	20	17	16
130	17	19	20	20	17	14
140	16	20	**22**	20	17	16
150	18	19	20	20	18	17
160	10	19	20	20	17	15
170	10	19	20	20	16	15
180	20	19	20	20	18	19
190	10	19	20	20	19	15
200	10	19	20	20	16	15
210	10	19	20	20	16	17

Table 4.6: Number of improved instance for the various configurations.

budget	$m/1$	$m/2$	optimal	$m/3$	$m/4$	$m/5$
120	316	395	390	401	322	260
130	394	421	411	426	412	331
140	406	435	428	442	431	353
150	412	448	439	459	447	370
160	417	458	448	465	458	380
170	405	454	450	465	458	380
180	392	457	446	466	444	368
190	403	462	448	470	463	422
200	339	466	378	**476**	393	332
210	379	444	515	462	456	380

2. In contrast to the SAT-based approach the tangles-based approach cannot compute upperbounds for the branchwidth of a (hyper-)graph. Computing upperbounds is however crucial when used inside the local improvement algorithm, as the local hypergraphs are too large to be solved exactly by any known method. In particular, as our experiments show the local improvement method performs best when the number of hyperedges in the local hypergraphs is around 200.

3. As also pointed out by [Hic05] the space and time complexity of the tangles-based approach grows exponentially with the branchwidth and is therefore not applicable for (hyper-)graphs with high branchwidth, which is normally the case for the local hypergraphs encountered during local improvement.

Another advantage of our SAT-based approach is the reduced space requirements, which in contrast to the tangles-based approach grow only linearly, instead of exponential, with the branchwidth.

Our experiments show that the SAT-based local improvement approach scales well to large graphs with several thousands of vertices and edges and branchwidth upper bounds well over hundred. These are instances that are by far out of reach for any known exact method, in particular, for the tangles-based algorithm which cannot handle large branchwidth. The use of our SAT-encoding which scales well with the branchwidth is therefore essential for these instances.

Our results on TreewidthLIB instances show that in some cases the obtained improvement can make a difference of whether a dynamic programming algorithm that uses the obtained branch decomposition is feasible or not. Our experiments also show that it can be worth to tune the local improvement approach using parameters such as budget and depth.

4.9 Chapter Summary

We have presented a first SAT-encoding for branchwidth based on a novel partition-based formulation of branch decompositions and introduced the new method of SAT-based local improvements for branch decompositions. Our SAT-based local improvement method provides the means for scaling the SAT-approach to significantly larger instances and exhibits a fruitful new application field of SAT-solvers. In many cases the SAT-based local improvement could obtain branch decompositions of a width that makes a dynamic programming feasible, which was not possible with the original branch decomposition obtained by a heuristics.

For both the single SAT-encoding and the SAT-based local improvement we see several possibilities for further improvement. For the encoding one can try other ways for stating cardinality counters and one could apply incremental SAT solving techniques. Further, one could consider alternative encoding techniques based on MaxSAT, which have been shown effective for related problems [BJ14]. Also for the local improvement we see various directions for further research. For instance, when a local branch decomposition cannot be improved, one could use a SAT-solver to obtain an alternative branch decomposition of the same width but where other parameters are optimized, e.g., the number of maximum cuts. This could propagate into adjacent local improvement steps and yield an overall branch decomposition of smaller width. Our experiments show the performance of our local improvement approach depends on the exact combination of various parameters such as budget and depths. It would therefore be interesting for future work to investigate the benefit from tools for automated parameter configuration [FLH15].

Finally we would like to mention that branch decompositions are the basis for several other (hyper)graph width measures such as rankwidth and Boolean-width [ABR+10], as well to width-parameters employed in Knowledge Compilation and Reasoning [Dar09]. Hence we think it might be fruitful to extend our methods to such other width measures related to branchwidth and leave this for future research.

CHAPTER 5

Treewidth

In this chapter we focus on SAT-based local improvement for *tree decompositions*. This chapter is based on our paper published at SAT 2017 [FLS17b]. After a brief introduction on tree decompositions we start with some basic concepts required to prove the correctness of out approach. Next, we describe the exact algorithm used for the same. We provide a brief description of our experiments and conclude this chapter with chapter summary.

5.1 Introduction

Treewidth is arguably the most prominent graph invariant with various important application in discrete algorithms and optimization [BK08, CMZ12], constraint satisfaction [Dec06, Fre85], knowledge representation and reasoning [GPW10], computational biology [SLM+05], and probabilistic networks and inference [Dar03, LS88a, OS13]. Treewidth was introduced by Robertson and Seymour in their Graph Minors Project and according to Google Scholar[1], the term is mentioned in over 18,400 research articles.

Small treewidth of a graph indicates in a certain sense its tree-likeness and sparsity. Many otherwise NP-hard graph problems such as Hamiltonicity and 3-colorability, but also problems "beyond NP" such as the #P-complete problem of determining the number of perfect matchings in a graph are solvable in polynomial time for graphs of bounded treewidth [CMR01]. Treewidth is based on certain decompositions of graphs, called tree decompositions, where sets of vertices of the input graph are arranged in bags at the nodes of a tree such that certain conditions are satisfied. The width of a tree decomposition is the size of a largest bag minus 1. A tree decomposition is optimal for a given graph if the graph has no tree decomposition of smaller width. The treewidth of a graph is the width of an optimal tree decomposition.

[1] Retrieved on September 3, 2018.

Algorithms that exploit the small treewidth of a graph usually proceed by dynamic programming along the tree decomposition where at each node of the tree, information is gathered in tables. The size of these tables is usually exponential or even double exponential in the size of the bag. Thus, it is important to obtain a tree decomposition of small width. However, since finding an optimal tree decomposition is an NP-hard task [ACP87], the following two main approaches have been proposed in the literature:

(a) *Exact methods* that compute optimal tree decompositions. Optimal tree decompositions are found using specialized combinatorial algorithms based on graph separators [ACP87], branch-and-bound algorithms [GD04], but also by means of *SAT encodings* [BJ14, SV09]. These exact methods are limited to rather small graphs with about hundred vertices.

(b) *Heuristic methods* that compute sub-optimal tree decompositions. These algorithms are usually based on so-called elimination orderings which are found by a greedy approach [BK10, HMS15a]. The heuristic methods are quite fast and scale up to large graphs with thousands of vertices, but lead to tree decompositions that can be far from optimal.

In fact, because of the split into these two categories of algorithmic approaches, also the recent PACE challenge [DR16], where finding good tree decompositions was one of the main tasks, featured two respective categories: one asking for the exact treewidth of small graphs, and one asking for sub-optimal tree decompositions of large graphs.

5.1.1 SAT-Based Local Improvement

In this chapter, we propose a new approach to finding tree decompositions, which combines exact methods with heuristics. The basic idea is to (i) start with a tree decomposition obtained with a heuristic method (the *global solver*) and (ii) subsequently select parts of the tree decomposition, trying to improve it with another method (the *local solver*). It turned out that SAT-based exact methods are particularly well-suited for providing the local solver.

Consider a given graph G and a tree decomposition $\mathcal{T}$ of G, obtained by the global solver. We select a small part $\mathcal{S}$ of $\mathcal{T}$, which is a tree decomposition of the subgraph $G_{\mathcal{S}}$ of G, induced by all the vertices that appear in bags at nodes in $\mathcal{S}$. Once the local solver finds a better tree decomposition of $G_{\mathcal{S}}$, we would like to replace $\mathcal{S}$ in $\mathcal{T}$ with the new tree decomposition found by the local solver. This, however, does not work in general, as the new tree decomposition might not fit into the remaining parts of $\mathcal{T}$. Fortunately we can make this approach work by using the following trick. We add to $G_{\mathcal{S}}$ certain cliques, which we call *marker cliques*, and which tell us how to replace the original local tree decomposition $\mathcal{S}$ with the new one. Due to a general property of tree decompositions, there is always a bag that contains all vertices of a clique. Hence, in particular, the new local decomposition will contain for each marker clique a bag that contains it, and this

bag will be an anchor point for connecting the new decomposition to the parts of the old one. Details of this construction are explained in next section.

We refer to Chapter 2 for basic definitions.

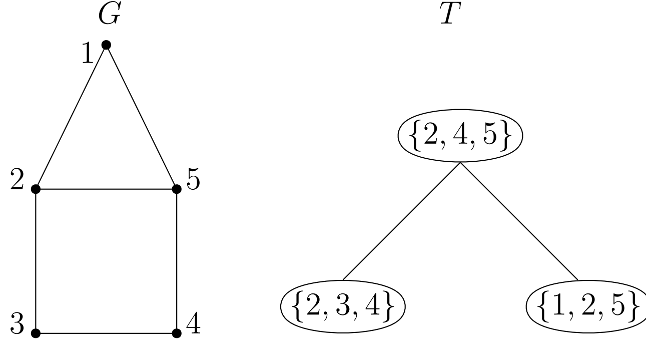


Figure 5.1: A graph G (left) and an optimal tree decomposition $\mathcal{T} = (T, \chi)$ of G (right).

5.2 Local Improvement of Tree Decompositions

5.2.1 Local Tree Decompositions

For the following considerations we fix a graph G and a tree decomposition $\mathcal{T} = (T, \chi)$ of G. We consider a subtree S of T.

We call $\mathcal{S} = (S, \chi_S)$ a *local tree decomposition of* $\mathcal{T}$ (*induced by* S), where χ_S is the restriction of χ to the nodes of S. Let $G_\mathcal{S}$ denote the subgraph of G induced by all the vertices of G that appear in a bag of $\mathcal{S}$. The following observation is an immediate consequence of the definitions.

Observation 1. *$\mathcal{S}$ is a tree decomposition of $G_\mathcal{S}$ of width $\leq \mathsf{w}(\mathcal{T})$.*

Our goal is to replace $\mathcal{S}$ with an improved tree decomposition $\mathcal{S}'$ of $G_\mathcal{S}$, i.e., one of smaller width, and to insert $\mathcal{S}'$ back into $\mathcal{T}$ so that we obtain a new tree decomposition $\mathcal{T}'$ of G of possibly smaller width. In order to make this work, we need to modify $G_\mathcal{S}$ such that any tree decomposition of the modified graph can be added back into $\mathcal{T}$.

Let us first introduce some auxiliary notions. For an edge st of $\mathcal{T}$ we define $\lambda_\mathcal{T}(st) = \chi(s) \cap \chi(t)$ to be the *cut set* associated with st. We call an edge st of T to be a *boundary edge* (w.r.t. $\mathcal{S}$) if $s \in V(S)$ and $t \notin V(S)$.

Now we define the *augmented local graph* $G^*_\mathcal{S}$ by setting $V(G^*_\mathcal{S})$ to be the set of all vertices of G that appear in a bag of $\mathcal{S}$, and $E(G^*_\mathcal{S})$ to be the set of edges uv with $u, v \in V^*$ such that $uv \in E(G)$ or $u, v \in \lambda_\mathcal{T}(e)$ for a boundary edge e of T. In other words, the augmented local graph $G^*_\mathcal{S}$ is obtained from $G_\mathcal{S}$ by forming cliques over cut sets associated with boundary edges. We will use these cliques as "markers" in order to

connect a new tree decomposition of $G^*_\mathcal{S}$ to the parts of the tree decomposition $\mathcal{T}$ that we keep. Therefore we call these cliques *marker cliques*.

Observation 2. *$\mathcal{S}$ is a tree decomposition of $G^*_\mathcal{S}$ of width $\leq \mathsf{w}(\mathcal{T})$.*

Proof. In view of Observation 2, it remains to check that for each edge $uv \in E(G^*_\mathcal{S})\backslash E(G_\mathcal{S})$ there is a node s of S such that $u, v \in \chi(s)$. For such an edge uv there is a boundary edge e of T such that $u, v \in \lambda_\mathcal{T}(e)$. By definition of a boundary edge, exactly one end of e, say s, belongs to $V(S)$. Now $u, v \in \lambda_\mathcal{T}(e) \subseteq \chi(s)$. □

Let $\mathcal{S}^* = (S^*, \chi^*)$ be another tree decomposition of $G^*_\mathcal{S}$ with $\mathsf{w}(\mathcal{S}^*) \leq \mathsf{w}(\mathcal{S})$. W.l.o.g., we assume that S^* and T do not share any vertices (if not, we can simply use a tree that is isomorphic to S^*). We define a new tree decomposition $\mathcal{T}' = (T', \chi')$ of G as follows.

Let $T_1, \dots, T_r$ be the connected components of $T - S$ (each T_i is a tree). Each T_i gives raise to a local tree decomposition $\mathcal{T}_i = (T_i, \chi_i)$, where χ_i is the restriction of χ to the nodes of T_i.

For each T_i, let t_i be the leaf of T_i that was incident with a boundary edge $e_i = t_i s_i$ in T. The boundary edge e_i is responsible for a marker clique $K(e_i)$ on the vertices in $\lambda_\mathcal{T}(e_i)$. By Fact 1, we can choose a node $s'_i \in V(S^*)$ such that $V(K(e_i)) = \lambda_\mathcal{T}(e_i) \subseteq \chi^*(s'_i)$.

We define a new tree decomposition $\mathcal{T}' = (T', \chi')$ where T' is the tree defined by $V(T') = V(S^*) \cup \bigcup_{i=1}^r V(T_i) = V(S^*) \cup V(T) \setminus V(S)$ and $E(T') = E(S^*) \cup \bigcup_{i=1}^r E(T_i) \cup \{t_1 s'_1, \dots, t_r s'_r\}$. It remains to define the bags of the tree decomposition $\mathcal{T}'$. For $t \in V(T_i)$ we define $\chi'(t) = \chi(t)$ and for $s \in V(S^*)$ we define $\chi'(s) = \chi^*(s)$. We denote $\mathcal{T}'$ as $\mathcal{T}(\frac{\mathcal{S}}{\mathcal{S}'})$ and say that $\mathcal{T}'$ *is obtained from $\mathcal{T}$ by replacing $\mathcal{S}$ with $\mathcal{S}'$*.

Observation 3. *$\mathcal{T}(\frac{\mathcal{S}}{\mathcal{S}'})$ is a tree decomposition of G of width*

$$\max(\mathsf{w}(\mathcal{T}_1), \dots, \mathsf{w}(\mathcal{T}_r), \mathsf{w}(\mathcal{S}^*)) \leq \max(\mathsf{w}(\mathcal{T}), \mathsf{w}(\mathcal{S}^*)) \leq \max(\mathsf{w}(\mathcal{T}), \mathsf{w}(\mathcal{S})) \leq \mathsf{w}(\mathcal{T}).$$

Proof. Let $\mathcal{T}(\frac{\mathcal{S}}{\mathcal{S}'}) = \mathcal{T}' = (T'\chi')$. First we observe that T' is indeed a tree, as each tree T_i is connected to the central tree S^* with exactly one edge. Clearly $\mathcal{T}'$ satisfies the first of the two conditions in the definition of a tree decomposition. To see that it also satisfies the second condition, we observe that if a vertex v of G appears in bags at two different local tree decompositions $\mathcal{T}_i$ and $\mathcal{T}_j$ then v must also appear in the sets $\lambda_\mathcal{T}(e_i)$ and $\lambda_\mathcal{T}(e_j)$. Consequently, it appears in the bags of s'_i and s'_j (we use the notation from above). As $\mathcal{S}^*$ satisfies the second condition of a tree decomposition, v is contained in all the bags on the path between s'_i and s'_j in S^*. This shows that $\mathcal{T}'$ is indeed a tree decomposition of G. The claimed bound on its width follows directly from the construction. □

5.2.2 SAT-encodings for Tree Decompositions

A SAT-encoding for tree decompositions was first proposed by Samer and Veith [SV09]. Given a graph G and an integer k, we produce a CNF formula, which is satisfiable if

and only if G has a tree decomposition of width $\leq k$. As already stated in Chapter 2, for the construction of $\Phi(G, k)$, an alternative characterization of tree decompositions in terms of *elimination orderings* is used. Here a linear ordering of the given graph G is guessed, and based on the ordering certain "fill-in edges" are added to the graph, providing a "triangulation" of G. The ordering is represented by Boolean variables, one for every pair of vertices, whose truth value indicates the relative ordering of the two vertices. Transitivity of the ordering is ensured by suitable clauses. Then, for each vertex v of G it is checked whether it has at most k neighbors that appear in the ordering right to v. This is checked via cardinality counters [Sin05]. The exact treewidth is then found by systematically calling a SAT-solver for a heuristically computed upper bound u with $\Phi(G, k)$ for $k = u, u-1, u-2, \ldots$ and until $\Phi(G, k)$ is found unsatisfiable. From a satisfying assignment of $\Phi(G, k)$ one can obtain a tree decomposition of G of width k efficiently by a *decoding procedure*.

5.2.3 The Local Improvement Loop

In this section we will formalize the idea of local improvement for tree decomposition. Let G be an input graph. First we obtain a tree decomposition $\mathcal{T} = (T, \chi)$ of G using a standard heuristic method, which we refer to as the **global solver**.

The local improvement loop operates with the following parameters which are positive integers: the local budget **lb**, the local timeout **lt**, the global timeout **gt**, and the number of no-improvement rounds **ni**.

We select a node t from T with largest bag size, i.e., $|\chi(t)| = \mathsf{w}(\mathcal{T})$. The hope here is that using local improvement we can reduce the size of this bag. To do this we perform a modified breadth-first-search (BFS) starting at t in T. We use an auxiliary set variable L which, at the beginning of the BFS is set to $\chi(t)$. For each node t' visited by the BFS, we add the new elements of $\chi(t')$ to L. If a node t' was visited via an edge e, a neighbor t'' of t' is only visited if $\lambda_{\mathcal{T}}(t't'') < \lambda_{\mathcal{T}}(e)$. The BFS terminates as soon as visiting another node would increase the size of L beyond the local budget **lb**. The nodes visited in this way induce a subtree S of T, and in turn, this yields a local tree decomposition $\mathcal{S} = (S, \chi_S)$ of $\mathcal{T}$, as defined above. The set L contains the vertices of the local graph $G_{\mathcal{S}}$ (or equivalently, of the augmented local graph $G^*_{\mathcal{S}}$) which by construction can be at most **lb** many vertices.

Next we run the **local solver**, i.e., we check satisfiability of the formula obtained by the SAT-encoding, to obtain a tree decomposition $\mathcal{S}^*$ of $G^*_{\mathcal{S}}$ whose width is as small as possible. We start the SAT-encoding with $k = \mathsf{w}(\mathcal{S}) - 1$ and upon success decrease k step by step. Each SAT-call has a timeout of **lt** seconds, and we stop if either we get an unsatisfiable instance or we hit the timeout. With the reached value of k, the treewidth of $G^*_{\mathcal{S}}$ is at most $k+1$. Since the SAT-encoding with value $k+1$ is satisfiable, we can extract with a decoding procedure from the satisfying assignment a tree decomposition $\mathcal{S}^*$ of $G^*_{\mathcal{S}}$. Now we replace $\mathcal{S}$ in $\mathcal{T}$ by $\mathcal{S}^*$, and we repeat the local improvement loop with $\mathcal{T}(\frac{\mathcal{S}}{\mathcal{S}^*})$. We note that a local replacement is done even if there was no local width improvement,

i.e., if $\mathsf{w}(\mathcal{S}^*) = \mathsf{w}(\mathcal{S})$, as there is the possibility that the change triggers improvements in subsequent rounds of the local improvement loop.

We repeat the local improvement loop until either the global timeout **gt** is reached, or if the loop has been iterated **ni** times without any local width improvements.

5.3 Experimental Results

5.3.1 Solvers

As the global solver we used the greedy ordering heuristics-based algorithm from Abseher et al. [AMW17, rev. 075019f] which we refer to as `heur`. It computes upper bounds for treewidth and outputs a certificate decomposition. The solver scored third in the heuristic track of of the PACE 2016 challenge [DR16]. It is very space efficient and reports initial useful tree decompositions extremely fast compared to other solvers. It leaves almost the full time resource for the local improvement. We used the following three local solvers:

1. `sat`: a solver based on an improved version of Samer and Veith's [SV09] SAT-encoding by Bannach et al. [BBE16, rev. 25d6a98]. The solver employs Glucose as a SAT-solver, PBLib for cardinality encodings, and progresses downwards from an upper bound. The solver scored third in the exact track of the PACE 2016 treewidth challenge and was there the best SAT-based solver.

2. `comb`: an implementation of Arnborg et al.'s combinatorial algorithm [ACP87] by Tamaki [Tam16, rev. d5ba92a], This solver won the exact track of the PACE 2016 treewidth challenge. It incrementally checks for the exact treewidth, it progresses upwards from 1.

3. `heur`: the same solver that we also use as global solver.

Our implementation is publicly available on GitHub [FLS17a]. Our experiments mainly focus on two questions: (i) can we improve with local improvement over traditional greedy heuristics and (ii) which solvers are favorable as local solver.

5.3.2 Instances

We considered an initial selection of overall 3168 graphs from various publicly available graph sets. Our sets consisted of the *TreewidthLIB* [vdBB10], networks from the *UAI competition* [Dec13a], publicly available transit graphs from *GTFS-transit feeds* [Fic16], and graphs from the *PACE 2016 treewidth challenge* [DR16]. Since we aimed for larger graphs where exact methods cannot be used, we restricted ourselves to graphs that contain more than 100 vertices, resulting in 1946 graphs in total.

5.3.3 Experimental Setup:

The experiments ran on a Scientific Linux cluster of 24 nodes (2x Xeon E5520 each) and overall 224 physical cores [Kit17]. Due to the large number of instances, we started only from one initial decomposition (with random seed) and did not repeat the runs. In order to have reproducible results we used a benchmark cluster run generator and analysis tool[2]. All solvers have been compiled with gcc version 4.9.1, ran on Python 2.7.5, and Java 1.8.0_122 HotSpot 64-bit server VM, respectively. We executed solvers in single core mode. We limited available memory (RAM) to 8GB, wall clock time of the global solver to 15 seconds, wall clock time of the overall search to 7800 seconds, and wall clock time of the local solver to 1800 seconds. For the SAT-solver we imposed an additional restriction that the individual SAT call runs at most 900 seconds (**st**). Resource limits where enforced by *runsolver* [Rou11].

For our experiments, we systematically tested the parameters **lb** $\in$ {75, 100,125, 150}, **lt** $\in \{90, 900, 1800\}$, **gt** $= 7200$, and **ni** $= 10$. For the parameter **ni** we also tried values 40 and 100 on a selected set of instances, but obtained no improvements. Individual results are publicly available [FLS17a].

Table 5.1: Summary of treewidth improvements.

#improved	improvements (sum)	improvement (max)	solver configuration
647	**2015**	13	`sat-100-1800(900)`
584	1984	16	`sat-125-1800(900)`
630	1805	15	`comb-100-1800`
493	1676	**20**	`sat-150-1800(900)`
609	1460	12	`comb-075-1800`
447	1077	19	`comb-125-1800`
368	822	14	`comb-150-1800`
325	538	9	`heur-150-1800`
258	421	8	`heur-100-1800`

5.3.4 Results

Table 5.1 summarizes the improvements we obtained with our experiments. Configurations in the legend are given in the form `solver-lb-lt(st)`. The best results in each column are highlighted in bold font. Table 5.2 shows some of the best and notable improvements we obtained with local improvements. The value “hash” provides the first four digits of sha-1 hash sum for the instance in DIMACS graph format. Column “htw” has the heuristically obtained treewidth, and “itw” has the treewidth after local improvement. The configuration with which we got these improvements are in the column “local solver.”

[2] The run and analysis tool is available online at `https://github.com/daajoe/benchmark-tool`. The file `benchmark-tool/runscripts/treewidth/localimprovement.xml` contains all solver flags to reproduce our benchmark runs.

The best improvement we obtained is 20, for the instance or_chain_224.fg, from the graph set networks. Among further entries in the table are instance graph13pp with a width over 100, and instance Promedus_38 where we could reduce the width from 23 to 16, which makes this instance feasible for dynamic programming.

Table 5.2: Some of the best and notable improvements

instance (hash)	$\|V\|$	$\|E\|$	graphs	itw	htw	local solver
or_chain_224.fg (a4cb)	1638	3255	networks	75	95	`sat-150-1800-10`
or_chain_54.fg (a6fc)	1404	2757	networks	65	84	`comb-125-1800-10`
or_chain_187.fg (826a)	1668	3197	networks	79	97	`sat-150-1800-10`
1bkr_graph (003a)	107	1340	twlib	44	56	`comb-075-1800-10`
dimacs_fpsol2.i.1-pp (69aa)	191	4418	pace2016	61	72	`sat-150-1800-10`
graph13pp (eb9d)	456	1874	twlib	115	125	`comb-150-1800-10`
Cell120 (b625)	600	1200	pace2016	94	104	`comb-150-1800-10`
bkv-zrt_20120422_0314 (fbca)	907	2209	transit	74	83	`sat-150-1800-10`
Promedus_38 (02d7)	668	1235	networks	16	23	`sat-150-1800-10`

5.3.5 Discussion

For our instance set, we can see that even a heuristic solver as local solver (**lb** = 150) improved the upper bounds. Both in terms of number of improved instances and when considering the cumulative sum of improvements, the SAT-based solver performed best. For both the combinatorial solver and the SAT-based solver, a local budget **lb** = 100 resulted in more solved instances. However, in terms of overall improvement the difference between the two local solvers is small. A local budget **lb** = 125 allowed us to increase the cumulative sum of improvements relatively early.

In consequence, we obtained the best results by using a SAT-based solver as local solver. Using a SAT-based solver, we can hope that an improved SAT-encoding or new techniques in solvers immediately yield better upper bounds for treewidth using local improvement. We also computed the virtually best solver, which improved 200 instances more than the best SAT-based configuration. This indicates that we can very likely improve a much higher number of instances when applying a portfolio based solving approach.

5.4 Chapter Summary

In this chapter, we have presented a new SAT-based approach to finding tree decompositions of small width based on a cross-over between standard heuristic methods and exact methods. Our work offers several directions for further research.

For instance, one could possibly improve the current setup by (a) upgrading the method for selecting the local tree decomposition, which is currently based on a relatively simple

breadth-first-search, and (b) tuning and optimizing the SAT-based local solver specially to handle the type of instances that arise within the local improvement loop.

Another promising direction involves adding additional constraints to the SAT-encoding, which yield local tree decompositions with special properties. For instance, when the local solver cannot improve the width of the current local tree decomposition, it could still replace it with one that increases the likelihood of success for further rounds of local improvements (for instance, by minimizing the number of large bags). Another application would be the computation of "customized tree decompositions" [AMW17] which are designed to speed-up dynamic programming algorithms. Such additional constraints are relatively easy to build into a SAT-based local solver, but seem difficult to build into a local solver based on combinatorial methods.

Finally, due to the modularity of our approach (local solver, budget, time out, invoked SAT-solver), it could benefit from automated algorithm configuration and parameter tuning, and it could provide the elements of a portfolio approach.

CHAPTER 6

Special Treewidth and Pathwidth

This chapter is dedicated to study the two techniques used for encoding width parameters in CNF formulas. It is based on an extended version of our paper published at SAT 2017 [LOS17a] which has been accepted at the Journal of Artificial Intelligence Research (JAIR). We start with the standard definitions of special treewidth and pathwidth. After that we are ready to start characterizing special treewidth using partition-based and ordering-based characterization. Along with each characterization we also provide the formulas for SAT-encoding of the same. Next, we provide a similar study for pathwidth. We finalize this chapter by comparing the encoding empirically and providing some concluding remarks.

6.1 Motivation

Graph decompositions are a central topic in the context of combinatorial algorithms, with applications in many areas of computer science. Decomposition methods give rise to so-called width parameters that indicate how well the graph is decomposable by the considered decomposition method. For instance, tree decomposition, the most famous decomposition method introduced independently by Bertele and Brioschi [BB73], Halin [Hal76], and Robertson and Seymour [RS84], gives rise to the parameter *treewidth*, where the treewidth of a graph is the smallest width over all tree decompositions. Many NP-hard problems become solvable in linear-time for instances whose treewidth is bounded by some constant [AP89, BK08, DF13]. In fact, a famous result by Courcell [Cou90] states that the same applies to any graph property expressible in monadic second-order logic. These theoretical results have inspired a wide-range of practical applications for treewidth and related width parameters in areas such as probabilistic and constraint networks [LS88b, Dec99, Dec13b], propositional model counting [BDP03], satisfiability [AR02, OD14], frequency assignment [KvHK99], logic programming [MMP+12], and problems on graphs [FBN15, CS03].

Crucial for the application of decomposition-based approaches are efficient algorithms for finding an *optimal decomposition*, i.e., one of smallest width; or at least one that is close to being optimal. Since this is often an NP-hard task, the development of efficient exact algorithms as well as heuristics that compute suboptimal solutions is of high importance. For instance, computing an optimal tree decomposition is known to be NP-hard [ACP87] and this has inspired the development of a large number of exact algorithms [SG97, GD04, BB06, Tam17b] as well as heuristic approaches [BK10, HMS15b]. Even though heuristic approaches are often employed in practice, there are several reasons why one is interested in optimal decompositions. If the purpose of the decomposition is to facilitate the solution of a hard problem by means of dynamic programming, then a suboptimal decomposition may impose an exponential increase on time and space requirements for the dynamic programming algorithm, and therefore may render the approach infeasible for the instance under consideration. For instance as Dechter et al. [KGOD11] noted about inference on probabilistic networks of bounded treewidth: "[...] since inference is exponential in the tree-width, a small reduction in tree-width (say even by 1 or 2) can amount to one or two orders of magnitude reduction in inference time." Besides such algorithmic applications, optimal decompositions are also useful for scientific purposes, for instance to evaluate a heuristic method that provides an upper bound on the decomposition width, or to support theoretical investigations by facilitating the construction of gadgets for hardness reductions.

Previous work on exact algorithms for computing optimal decompositions indicates that SAT provides a valuable practical approach for finding optimal decompositions. This approach was pioneered by Samer and Veith [SV09] for treewidth; their methods was further improved by Berg and Järvisalo [BJ14], and Bannach et al. [BBE17] and achieved excellent results in a recent solver challenge [DR16]. Heule and Szeider [HS15] developed the first practically feasible approach for computing the decomposition parameter *clique-width* by means of a SAT-encoding, which allowed for the first time to identify the clique-width of some well-known named graphs.

6.1.1 Pathwidth and Special Treewidth

In this thesis we consider new SAT-encodings for the decomposition parameters *pathwidth* and *special treewidth*. Both parameters are closely related to the well-known parameter treewidth. Specifically, pathwidth and special treewidth can both be defined in terms of a more restricted variant of tree decompositions (see Section 2.6.2). The motivation for special treewidth is that it, like pathwidth, allows for more efficient model-checking algorithms for variants of Monadic Second Order Logic than treewidth, but is often smaller than pathwidth [Cou10, Cou12]. Special treewidth has been the subject of several theoretical investigations [BKK+17, BKK13]. Pathwidth, on the other hand, was introduced by Robertson and Seymour [RS83] in the first of their famous series of papers on graph minors and has since then found applications in a wide variety of areas such as genome research [KS96], VLSI design [OMK+79], compiler design [BGT98], linguistics [KT92], and most prominently graph drawing [Sud04, Hli03, BCDM17, DFK+08, DMW02]. Com-

puting pathwidth and special treewidth is NP-hard and both width parameters cannot be approximated to within a constant factor. For the former this has been known [ACP87] for long, for the latter we observe that it can be deduced from known results (Theorem 6.1).

In the case of special treewidth, our encodings provide the first practical methods for computing special treewidth and its associated decomposition and in the case of pathwidth our encodings significantly improve on known tools [BBN+13]. Our results therefore provide a first step of bridging theoretical with experimental research for special treewidth and pathwidth.

6.1.2 Characterizations of Width Parameters

Previous work on SAT-encodings for treewidth, branchwidth and clique-width indicates that identifying a suitable characterization of the considered decomposition method is key for a practically feasible SAT-encoding. In fact, the standard encoding for treewidth [SV09] is based on the characterization of treewidth in terms of *elimination orderings*, which are linear orderings of the vertices of the decomposed graph, where after adding certain "fill-in" edges, the largest number of neighbors of a vertex ordered higher than the vertex itself, gives the width of the decomposition. For clique-width, on the other hand, no characterization based on elimination ordering is known, and the known SAT-encoding [HS15] uses a *partition-based* characterization, where one considers a sequence of partitions of the vertex set. We use a similar partition-based characterization for the SAT-encoding of branchwidth [LOS16a]. An encoding for pathwidth and similar decompositional parameters based on the interval model of a path decomposition has been introduced by [BBN+13].

Here, we develop four SAT-encodings based on two characterizations of special treewidth and two characterizations for pathwidth and provide a rigorous experimental comparison on a wide range of benchmark instances, i.e., a collection of well-known graphs from the literature [Wei16], the instances from TreewidthLIB [Bod16], a set of standard graphs containing square grids, complete graphs, and complete bipartite graphs, and a set of random graphs. We also study the applicability of preprocessing procedures known for treewidth to pathwidth and special treewidth.

Results for Special Treewidth For special treewidth we develop a new characterization based on elimination orderings (Theorem 6.4), as one could expect that a characterization that is similar to the characterization successfully used for a SAT-encoding of treewidth [SV09] also works well for special treewidth. We also develop a partition-based characterization which is close to the original characterization by [Cou10]. Our experiments show that the partition-based encoding clearly outperforms the ordering-based encoding. For instance, the former could process square grids and complete graphs being almost twice as large as the square grids and complete graphs within the reach of the latter. The partition-based encoding also beats the ordering-based encoding on many of the well-known named graphs that we consider by an order of magnitude and is competitive in running-times to the currently leading encoding for treewidth.

Results for Pathwidth For pathwidth, there exists a well-known characterization in terms of linear orderings [Kin92] which gives rise to a natural SAT encoding, similar in spirit to [SV09] encoding for treewidth. However, we also considered a partition-based encoding, similar in spirit to [HS15] encoding for clique-width. Our experiments indicate that both encodings have their merits; whereas the ordering-based encoding performs better overall on the collection of well-known graphs, the partition-based encoding has a slight edge on almost all other benchmark sets. In general, the ordering-based encoding seems to have an advantage on sparse graphs, while the partition-based encoding performs better on dense graphs. This encourages the development of a portfolio-based approach for SAT-encodings for pathwidth.

Preprocessing Preprocessing is one of the most important approaches to speed up algorithms on real-world instances. A good example for this is the recent development of algorithms for computing treewidth along the PACE competition [DHJ+17, DKTW18], where improvements by several orders of magnitude have recently been achieved and attributed to largely improved preprocessing procedures. In particular, the huge improvement (by two orders of magnitude) obtained by the winner of PACE 2017 [LS17] has been mostly attributed to preprocessing; in fact the winning algorithm of 2017 is basically an extension of the winning algorithm of 2016 [Tam17a] with preprocessing. We therefore conducted a systematic study of the applicability of known preprocessing procedure for treewidth to pathwidth and special treewidth. As a result we identified several preprocessing procedures that can be employed for pathwidth and special treewidth that we implemented as part of our algorithms for computing pathwidth and treewidth. For preprocessing procedures that provable do not preserve pathwidth or special treewidth, we also provide exact bounds on how far the pathwidth or special treewidth differs before and after preprocessing. Interestingly, we could show that many of the preprocessing procedures that do not preserve pathwidth or special treewidth exactly, only introduce a small error, i.e., the pathwidth or special treewidth can increase after preprocessing by at most +1 or a factor of 2. This makes those procedures promising candidates to be used as part of approximation algorithms or heuristics.

6.2 Preliminaries

For the basic definitions we will refer to the Chapter 2. In this section we will provide definitions and preliminaries specific to the special treewidth and pathwidth.

To define special treewidth, it is convenient to first introduce treewidth and pathwidth and then show how to adapt the definition to obtain special treewidth. For the convenience of our exposition we start with defining treewidth one more time.

A *tree decomposition* $\mathcal{T}$ of a graph $G = (V, E)$ is a pair (T, χ), where T is a tree and χ is a function that assigns each tree node t a set $\chi(t) \subseteq V$ of vertices such that the following conditions hold:

(T1) For every vertex $u \in V$, there is a tree node t such that $u \in \chi(t)$.

(T2) For every edge $\{u, v\} \in E$, there is a tree node t such that $u, v \in \chi(t)$.

(T3) For every vertex $v \in V$, the set of tree nodes t with $v \in \chi(t)$ forms a subtree of T.

The sets $\chi(t)$ for any $t \in V(T)$ are called *bags* of the decomposition $\mathcal{T}$ and $\chi(t)$ is the bag associated with the tree node t. The *width* of a tree decomposition (T, χ) is the size of a largest bag minus 1. A tree decomposition of minimum width is called *optimal* tree decomposition. The *treewidth* of a graph G is the width of an optimal tree decomposition of G. A *path decomposition* is a tree decomposition $\mathcal{T} = (T, \chi)$, where T is required to be a path and the *pathwidth* of a graph is the minimum width of any of its path decompositions.

A *special tree decomposition* $\mathcal{T} = (T, \chi)$ of a graph $G = (V, E)$ is a tree decomposition that is rooted at some node $r \in V(T)$ and additionally satisfies the following property [Cou10, BKK13]:

(ST) For every vertex $v \in V$, the set of tree nodes t with $v \in \chi(t)$ forms a subpath of a path in T from r to a leaf.

Note that *(ST)* subsumes *(T3)*, which implies that a special tree decomposition merely needs to satisfy *(T1)*, *(T2)*, and *(ST)*. The *width* of a special tree decomposition as well as the special treewidth of a graph G, denoted by $\mathrm{sptw}(G)$, are defined analogously to the width of a tree decomposition and the treewidth, respectively. Figure 6.1 illustrates an (optimal) special tree decomposition and path decomposition of a graph.

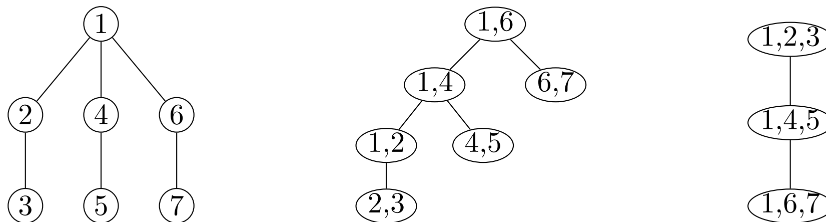


Figure 6.1: A graph G (left), an optimal (special) tree decomposition $\mathcal{T} = (T, \chi)$ of G (middle), and an optimal path decomposition (right).

As a prerequisite for the development of SAT-encodings for the problem, and since to the best of our knowledge this has never been explicitly stated previously, we first show that computing the special treewidth of a graph is NP-hard and, like pathwidth, special treewidth cannot be approximated within a constant factor.

Theorem 6.1. *Given a graph G and an integer ω, then determining whether G has special treewidth at most ω is NP-complete. Moreover, it is NP-hard to approximate special treewidth within a constant factor.*

Proof. The problem is clearly in NP, as there is always an optimal (special) tree decomposition, where the number of nodes is at most the number of vertices in the graph. Towards showing the NP-hardness and inapproximability of special treewidth, we employ the following simple reduction from pathwidth to special treewidth given by Bodlaender et al. [BKK+17, Lemma 2.5]: Let G be an undirected graph. Then the pathwidth of G plus one is equal to the special treewidth of the graph G' obtained from G by adding a vertex a and making a adjacent to all vertices of G. Because computing pathwidth is NP-hard and pathwidth cannot be approximated within a constant factor [BGHK95], the simple reduction shows that the same applies to special treewidth. □

We remark that if ω is constant and not part of the input, then one can check in linear time whether a given graph has special treewidth at most ω (the running time depends exponentially on ω) [BKK13]; similar results are well known to hold for treewidth and pathwidth.

We say that a tree decomposition (T, χ) is *small* if for every two distinct nodes t and t' in $V(T)$ and it does not hold that $\chi(t) \subseteq \chi(t')$. It is well-known [BK11, Lemma 2], that any tree decomposition can be turned into a small tree decomposition of the same width in polynomial-time; and the same applies to small path decompositions and small special tree decompositions.

We will need the following lemma, which is a slightly modified version of Lemma 2.5 [BKK+17], showing that pathwidth and special treewidth are essentially the same for any graph containing an *apex vertex*, i.e., a vertex adjacent to all other vertices of G.

Lemma 6.1 ([BKK+17, Lemma 2.5])**.** *Let G be a graph containing at least one apex vertex. Then any small special tree decomposition is also a path decomposition.*

Proof. Let G be a graph with apex vertex $a \in V(G)$ and let (T, χ) be a special tree decomposition of G. It suffices to show that T is a path. We start by showing that every bag of (T, χ) contains a. Suppose this is not the case and let $t \in V(T)$ be a bag with $a \notin \chi(t)$. For a vertex $v \in V(G)$, let $\chi^{-1}(v)$ be the subgraph of T induced by all bags containing v. Note that because of Property *(ST)*, $\chi^{-1}(v)$ is actually a subpath of a path of T from its root to a leaf. Then for every $u, v \in \chi(t)$, we have that $t \in \chi^{-1}(u) \cap \chi^{-1}(v)$ and hence $\chi^{-1}(u) \cap \chi^{-1}(v) \neq \emptyset$. Moreover, because $\{a, v\} \in E(G)$ for every $v \in \chi(t)$, we obtain that $\chi^{-1}(a) \cap \chi^{-1}(v) \neq \emptyset$. It follows that the sets $\{\, \chi^{-1}(v) \mid v \in \chi(t) \cup \{a\} \,\}$ are a set of pairwise intersecting subtrees of T, which implies that $\bigcap_{v \in \chi(t) \cup \{a\}} \chi^{-1}(v) \neq \emptyset$. Hence there is a $t_I \in V(T)$ with $\chi(t) \cup \{a\} \subseteq \chi(t_I)$ contradicting our assumption that (T, χ) is small. Hence a is contained in every bag of T and because of Property *(ST)*, T must be a path. □

Since a small path decomposition of a graph is always also a small special tree decomposition, we obtain the following corollary.

Corollary 6.1. *Let G be a graph containing at least one apex vertex. Then (T, χ) is a small special tree decomposition if and only if it is a small path decomposition of G.*

It is well known that the bi-connected components of a graph can be arranged in terms of a so-called *block-cut tree*. A *block-cut tree* B of a graph G is a tree that has one node for each block of G as well as one node for each articulation point of G and that has an edge between a block b and an articulation point a if $a \in b$. Here an *articulation point* a of G is any vertex of G that is shared by more than one block of G. For convenience we will always assume that the block-cut tree B of a graph G is rooted in an arbitrary block of G. Moreover, for a node n of B, we denote by B_n the subtree of B rooted at n and we denote by $C(n)$ the children of n in B. Finally, for any subtree B' of B we denote by $B(B')$ the set of all block nodes in B' and by $G[B']$ the subgraph of G induced by $\bigcup_{b \in B(B')} b'$. Figure 6.2 shows a biconnected graph and its corresponding block-cut tree.

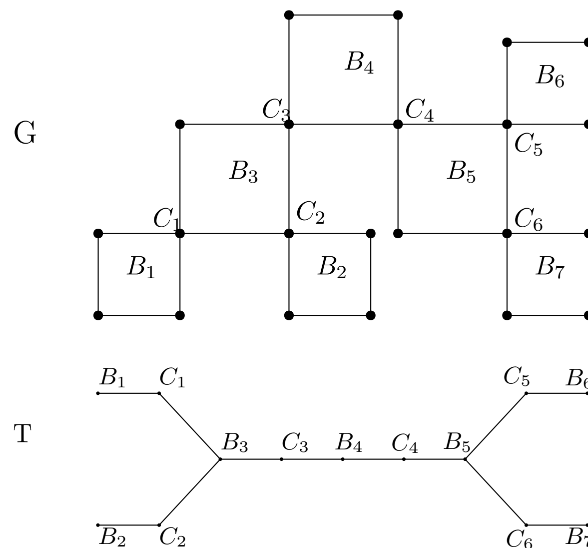


Figure 6.2: A biconnected graph G and its block-cut tree T.

6.3 Preprocessing

The aim of this section is to study common preprocessing procedures as well as symmetry breaking rules that have been used successfully for the computation of treewidth w.r.t. their applicability to pathwidth and special treewidth. Since all of the considered procedures involve at least some preprocessing, we will not distinguish between preprocessing and symmetry breaking but instead list all of them under the common denominator preprocessing procedure. In particular, we study the following preprocessing procedures:

- The *clique* preprocessing procedure uses the observation that every graph G has an optimal tree decomposition, whose root bag contains C, where C is any maximal

clique in G (Figure 6.3). To exploit this idea one first computes a maximal clique of G and one then only considers tree decompositions containing C in its root bag. Depending on the size of the maximal clique this can significantly reduce the search-space [BBE17].

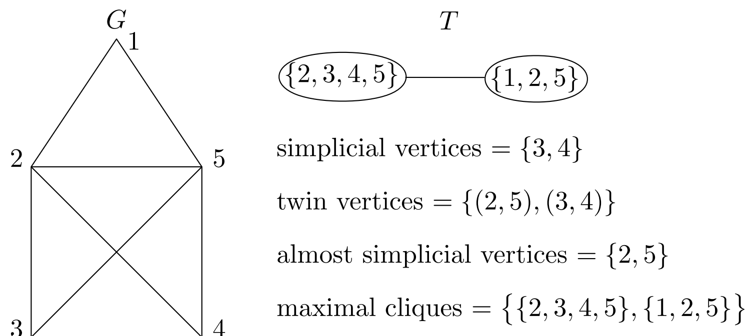


Figure 6.3: A graph G with maximal clique $C = \{2, 3, 4, 5\}$ and its optimal tree decomposition T, containing C in its root bag. The vertices 3 and 4 are twin vertices.

- The *twin* preprocessing procedure uses the observation that whenever two vertices u and v of a graph G are *twins*, i.e., either $N_G(u) = N_G(v)$ or $N_G[u] = N_G[v]$, then the role of u and v in any tree decomposition can be switched. Namely, if (T, χ) is a tree decomposition of G, then (T, χ') is also a tree decomposition of G (that has the same width as (T, χ)), where for every $t \in V(T)$ we set χ' as following:

$$\chi'(t) = \begin{cases} (\chi(t) \setminus \{u\}) \cup \{v\}), & \text{if } \chi(t) \cap \{u, v\} = \{u\} \\ (\chi(t) \setminus \{v\}) \cup \{u\}), & \text{if } \chi(t) \cap \{u, v\} = \{v\} \\ \chi(t), & \text{otherwise.} \end{cases}$$

  This idea can then be exploited by forcing a strict ordering on any set of twins and using it to break the symmetry between the twins. In Figure 6.3, the vertices 2 and 3 are twin vertices.

- The *degree one* preprocessing procedure uses the observation that if v is a vertex of degree one in a graph G, then $\text{tw}(G) = \max\{1, \text{tw}(G \setminus \{v\})$. Since any vertex of degree zero can be removed without changing the treewidth of the graph, this allows one to iteratively remove all vertices of degree at most one in G. Note that if we apply this procedure to an undirected tree, then the procedure results in an empty graph.

- Let G be a graph, with treewidth at least 2, and $v \in V(G)$ be a vertex of degree exactly two in G. The *contraction* of v in G is the operation defined by removing v from G and adding an edge between the two vertices adjacent to v in G. The *degree two* preprocessing procedure uses the observation that $\text{tw}(G) = \text{tw}(G')$, where G' is

obtained from G after *contracting* any vertex of degree exactly two, which allows one to iteratively contract vertices of degree two and thereby reduce the size of the graph.

- The *bi-connected* preprocessing procedure uses that observation that for every graph G, it holds that $\text{tw}(G) = \max_{B \in \mathcal{B}(G)} \text{tw}(G[B])$, where $\mathcal{B}(G)$ is the set of bi-connected components of G.

- Let G be a graph and $v \in V(G)$. We say that v is *simplicial* if $G[N_G(v)]$ is a clique. The *simplicial* preprocessing procedure uses the observation that $\text{tw}(G) = \max\{|N_G(v)|, \text{tw}(G \setminus \{v\}\}$ for any graph G and simplicial vertex $v \in V(G)$. Hence iteratively removing simplicial vertices reduces the size of the graph. In Figure 6.3, vertices 2 and 3 are simplicial vertices.

- The *almost simplicial* preprocessing procedure is slightly stronger version of the simplicial preprocessing procedure. Namely, given a graph G and $v \in V(G)$, we say that v is *almost simplicial* if v has a neighbor u such that $G[N_G(v) \setminus \{u\}]$ is a clique. The *almost simplicial* preprocessing procedure uses the observation that $\text{tw}(G) = \max\{|N_G(v)|, \text{tw}(G \setminus \{v\}\}$ for any graph G and almost simplicial vertex $v \in V(G)$. Hence iteratively removing almost simplicial vertices reduces the size of the graph. In Figure 6.3, vertices 1 and 4 are almost simplicial vertices.

In the following, we explore the applicability of the above procedures for pathwidth and special treewidth. In particular, we study which of the above procedures preserves pathwidth or special treewidth and can hence be used as a preprocessing step for an exact algorithm computing pathwidth or special treewidth. In the case that a certain procedure does not reserve pathwidth or special treewidth, we also provide upper bounds and lower bounds for the change in pathwidth respectively special treewidth incurred by the procedure. For instance, even though the bi-connected preprocessing procedure does not preserve special treewidth, we show that the special treewidth of the preprocessed instance and the original instance can differ by at most one. This means that the procedure might still be worth considering, when one is only interested in an approximate solution. Our results are summarized in Table 6.1 and the proofs of the results can be found in the appendix.

As can be seen from Table 6.1, only the twin preprocessing procedure preserves both width parameters exactly. Moreover, apart from the twin preprocessing procedure only the degree one rule is applicable for the exact computation of special treewidth. An experimental comparison of the effect of these preprocessing procedures can be found in Section 6.7.

The remainder of this section is devoted to proofs of the properties summarized in Table 6.1. We start by showing that the clique preprocessing procedure is not applicable to pathwidth and special treewidth, however, it can still be used as part of an 2-approximation algorithm.

preprocessing	sptw		pw	
clique	✗ (Cor. 6.1)	≤ 2× (Lem. 6.3)	✗ (Lem. 6.2)	≤ 2× (Cor. 6.3)
twin vertices	✓		✓	
degree one	✓ (Lem. 6.5)		✗ (Lem. 6.4)	
degree two	✗ (Lem. 6.6)	≤ +1 (Lem. 6.7)	✗(Lem. 6.6)	≤ +1 (Cor. 6.4)
bi-connected components	✗ (Lem. 6.8)	≤ +1 (Lem. 6.9)	✗ (Lem. 6.10	
simplicial vertex	✗ (Lem. 6.11)		✗(Cor. 6.5)	
almost simplicial vertex	✗ (Lem. 6.11)		✗(Cor. 6.5)	

Table 6.1: Applicability of the considered preprocessing procedures to pathwidth and special treewidth. A ✓indicates the preprocessing procedure is applicable, a ✗indicates that it is not. If the procedure is not applicable, but still allows for the computation of an approximate solution, we indicate the approximation error in parenthesis.

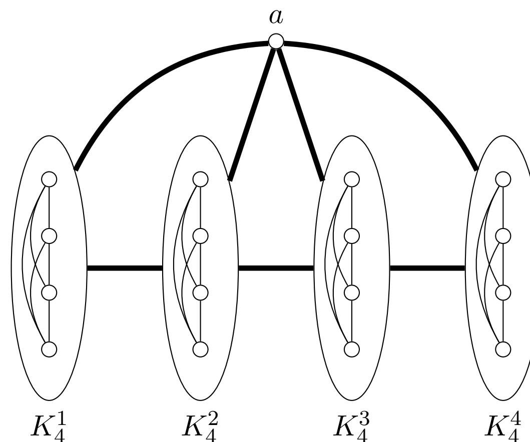


Figure 6.4: The graph P_4 used in the proof of Lemma 6.2. Bold edges indicate that all edges between the connected sets of vertices are present.

Let n be a natural number and K_n be the complete graph on n vertices. Let P'_n be the graph obtained from four copies of K_n arranged to a path of length four, i.e., P'_n is obtained from the disjoint union of four copies of K_n after adding all edges between the i-th and the $i+1$-th copy of K_n for every i with $1 \leq i < 4$. Moreover, let P_n be the graph obtained form P'_n after adding a new apex vertex a; see also Figure 6.4 for an illustration of the graph P_n. Let C be the union of the second and third copy of K_n in P'_n and $\{a\}$. Then C is a maximal clique of P_n. We claim that the width of any path decomposition

that starts or ends with a bag containing all of C has width at least $\frac{3}{2}\text{pw}(P_n)$.

Lemma 6.2. *The width of any path decomposition of P_n containing C in its first bag is at least $\frac{3}{2}\text{pw}(P_n)$.*

Proof. In the following we will denote by K_n^i for every $i \in \{1, 2, 3, 4\}$ the i-th copy of K_n in P_n. First note that P_n has pathwidth $2n$, since $P_n[C \cup \{a\}]$ forms a clique of size $2n+1$ and (P, χ) with $P = (p_1, p_2, p_3, p_4)$ and $\chi(p_i) = V(K_n^i) \cup V(K_n^{i+1}) \cup \{a\}$, for every i with $1 \leq i < 4$, is a path decomposition for P_n of width $2n$. We will show next that any path decomposition $\mathcal{P} = (P, \chi)$, for P_n that contains C in its first bag, has width at least $3n$, which concludes the proof of the lemma. Let $p \in V(P)$ be the left-most bag of $\mathcal{P}$ that does not contain all vertices in C, and let v be a vertex of C that is not contained in $\chi(p)$. Since P_n is symmetric, we can assume that v is contained in K_n^2. Because v is adjacent to a and all vertices in K_n^1, it follows that every vertex of K_n^1 must be contained in some bag that is to the left of p in P. Moreover, because $G[K_n^1 \cup \{a\}]$ is a clique, we obtain that there is at least one bag to the left of p in P, say p', that contains all vertices in $K_n^1 \cup \{a\}$. Consequently, $\chi(p')$ contains at least $3n+1$ vertices, i.e, all vertices in $V(C) \cup V(K_n^1) \cup \{a\}$. It follows that $\mathcal{P}$ has width at least $3n$. ☐

Since P_n contains the apex vertex a, we obtain from Corollary 6.1.

Corollary 6.2. *The width of any special tree decomposition containing C in its root bag is at least $\frac{3}{2}\text{sptw}(P_n)$.*

Lemma 6.2 and Corollary 6.2 rule out the use of the clique preprocessing procedure for use within any exact algorithm for computing pathwidth and special treewidth. However, as shown in Lemma 6.3 and Corollary 6.3 below, the procedure can still be employed (as a sub procedure) to obtain a 2-approximation for pathwidth and special treewidth.

Lemma 6.3. *Let G be a graph and C be a maximal clique of G. Then there is a special tree decomposition of G that contains C in its root bag of width at most $2\text{sptw}(G)$.*

Proof. Let (T, χ) be an optimal special tree decomposition of G. Then because C is a clique, there is a bag $t \in V(T)$ such that $V(C) \subseteq \chi(t)$. Let P be the set of all nodes of T that are on the path from the root of T to t in T and let $\chi' : V(T) \to 2^{V(G)}$ be the function defined by setting $\chi'(t') = \chi(t')$ for every $t \in V(T) \setminus P$ and $\chi'(t') = \chi(t') \cup V(C)$ for every $t \in P$. Then it is straightforward to verify that (T, χ') is a special tree decomposition of G of width at most $2\text{sptw}(G)$. Since (T, χ') contains C in its root bag this concludes the proof of the lemma. ☐

By observing that the same proof as in Lemma 6.3 can be employed to show the corresponding result for path decompositions, we obtain.

Corollary 6.3. *Let G be a graph and C be a maximal clique of G. Then there is a path decomposition of G that contains C in its first bag of width at most $2\text{pw}(G)$.*

This completes our results for the clique preprocessing processing procedure and we now continue with the twin preprocessing procedure. Recall that the twin preprocessing procedure is based on the observation that every two twins can switch their role in a tree decomposition. It is easy to see that this still applies to path decompositions and special tree decompositions and we show how this can be exploited by our SAT-encodings in Sections 6.6.1 and 6.6.2 for pathwidth and in Sections 6.4.2 and 6.5.2 for special treewidth.

We now turn our attention towards the degree one preprocessing procedure. We first show that the procedure is not applicable for pathwidth. In fact we show that there can be an arbitrary difference between the pathwidth of the original graph and the pathwidth of the graph after application of the degree one procedure. This rules out the use of the degree one procedure for pathwidth even if one is only interested in computing approximate solutions.

Lemma 6.4. *For every $n \in \mathbb{N}$, there is a graph G_n with $\text{pw}(G_n) = n$ and $\text{pw}(G'_n) = 0$, where G'_n is obtained from G_n after exhaustively removing vertices of degree at most one.*

Proof. It is well known that for every $n \in \mathbb{N}$ there is a tree G_n with $\text{pw}(G_n) = n$ [Die95]. Since exhaustively removing vertices of degree at most one from a tree results in the empty graph, we also obtain that $\text{pw}(G'_n) = 0$. □

In contrast to pathwidth the following lemma shows that the degree one procedure can be used for the computation of special treewidth.

Lemma 6.5. *Let G be a graph and l be a vertex of degree one in G, then $\text{sptw}(G) = \max\{1, \text{sptw}(G \setminus \{l\})\}$.*

Proof. It is straightforward to show that $\text{sptw}(G) \geq \max\{1, \text{sptw}(G \setminus \{l\})\}$. Moreover if $\text{sptw}(G \setminus \{l\}) = 0$, then G is a tree and $\text{sptw}(G) = 1$, as required. Hence let (T, χ) be a special tree decomposition of $G \setminus \{l\}$ of width at least one and let p be the unique vertex adjacent to l in G. Finally, let $t_p \in V(T)$ be the unique node of T with $p \in \chi(t)$ that is furthest away from the root of T. Then (T', χ'), where T' is obtained from T after adding the node t_l and the edge $\{t_l, t_p\}$, and χ' is defined by setting $\chi'(t_l) = \{p, l\}$, and $\chi'(t) = \chi(t)$ for every $t \in V(T)$, is a special tree decomposition of G, whose width is equal to the width of (T, χ), as required. □

We now consider the degree two preprocessing procedure. We start by giving a simple example showing that contracting vertices can decrease the pathwidth and special treewidth of a graph.

Lemma 6.6. *There is a graph H with $\text{pw}(H) = \text{pw}(H') + 1$, where H' is the graph obtained from H after exhaustively executing contractions. A similar statement holds for special treewidth, i.e., there is a graph G with $\text{pw}(G) = \text{pw}(G') + 1$, where G' is the graph obtained from G after exhaustively executing contractions.*

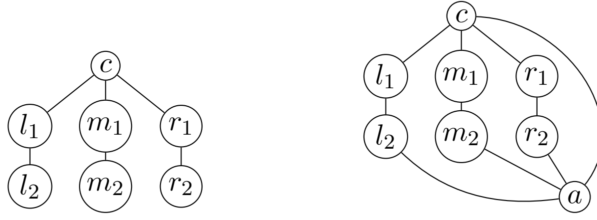


Figure 6.5: The graph H (left) and the graph G (right) from the proof of Lemma 6.6.

Proof. Let H be the graph illustrated in Figure 6.5, i.e., H has vertices c, l_1, l_2, m_1, m_2, r_1, and r_2 and contains the following edges:

1. an edge between c and all vertices in $\{l_1, m_1, r_1\}$,
2. an edge between l_1 and l_2,
3. an edge between m_1 and m_2, and
4. an edge between r_1 and r_2.

Then the graph H' is simply the star with three leaves and $\text{pw}(H') = 1$. It remains to show that $\text{pw}(H) = 2$. Clearly $\text{pw}(H) \leq 2$. Towards showing that $\text{pw}(H) > 1$ suppose for a contradiction that this is not the case and there is a path decomposition $\mathcal{P} = (P, \chi)$ of H of width one. Then $\mathcal{P}$ has to contain:

1. a bag l with $\chi(l) = \{c, l_1\}$,
2. a bag m with $\chi(m) = \{c, m_1\}$, and
3. a bag r with $\chi(r) = \{c, r_1\}$.

Due to the symmetry of H we can assume w.l.o.g. that l is to the left of m, which in turn is to the left of r in P. Moreover, $\mathcal{P}$ has to contain a bag m' with $\chi(m') = \{m_1, m_2\}$ and since c is contained in every bag between l and r in P, the bag m' has to occur either to the left of l or to the right of r in P. However, this contradicts our assumption that $\mathcal{P}$ is a path decomposition since in either case the set of bags containing m_1 would not be connected in $\mathcal{P}$.

Towards showing the result for special treewidth, let G be the graph illustrated in Figure 6.5, i.e., G is obtained from H after adding the vertex a and the following edges:

1. an edge between a and c,
2. an edge between a and l_2,

3. an edge between a and m_2, and
4. an edge between a and r_2.

Then the graph G' is the star with three leaves with apex vertex a and $\mathrm{sptw}(G') = 2$. Since $\mathrm{sptw}(G) \leq 3$, it remains to show that $\mathrm{sptw}(G) > 2$. Assume for a contradiction that this is not the case and there is a special tree decomposition $\mathcal{T} = (T, \chi)$ for G of width two. We first show the following claim.

Claim 6.1. *$\mathcal{T}$ contains a bags l, m, and r such that:*

1. $\chi(l) = \{a, c, l_1\}$ *or* $\chi(l) = \{a, c, l_2\}$,
2. $\chi(m) = \{a, c, m_1\}$ *or* $\chi(l) = \{a, c, m_2\}$,
3. $\chi(r) = \{a, c, r_1\}$ *or* $\chi(l) = \{a, c, r_2\}$.

Proof. Due to the symmetry of G it is sufficient to show the existence of the bag l in $\mathcal{T}$. Towards showing this let t_{ac}, t_{al_2}, t_{cl_1}, and $t_{l_1l_2}$ be bags of $\mathcal{T}$ containing $\{a, c\}$, $\{a, l_2\}$, $\{c, l_1\}$, and $\{l_1, l_2\}$, respectively. Then t_{ac} and t_{al_2} have to lie on a common path from the root to a leaf of T and moreover $t_{ac} \neq t_{al_2}$ since otherwise $\chi(t_{ac}) = \{a, c, l_2\}$. Due to the symmetry of G we can assume that t_{ac} lies closer to the root than t_{al_2}. It follows that t_{cl_1} cannot lie between t_{ac} and t_{al_2} since otherwise $\chi(t_{cl_1}) = \{a, c, l_1\}$. Also t_{cl_1} cannot lie after t_{al_2} since otherwise $\chi(t_{al_2}) = \{a, c, l_2\}$.

Hence, either t_{cl_1} lies before t_{ac} or there is a node t between t_{ac} and t_{al_2} such that t_{cl_1} lies below t in $\mathcal{T}$. In the former case all four nodes t_{ac}, t_{al_2}, t_{cl_1}, and $t_{l_1l_2}$ have to lie on one path from the root to a leave of T. But then either $\chi(t_{ac}) = \{a, c, l_2\}$ if $t_{l_1l_2}$ lies before t_{ac} on the path or $\chi(t_{ac} = \{a, c, l_1\}$ if $t_{l_1l_2}$ lies after t_{ac} on the path. In the later case $t_{l_1l_2}$ has to lie on the path from the root to t in $\mathcal{T}$ and hence $\chi(t) = \{a, c, l_1\}$. □

Let l, m, and r be the nodes of T given in the above claim. Because all of them contain a, we obtain that they have to lie on a path from the root to a leave of T. Due to the symmetry of G, we can assume w.l.o.g. that m lies between l and r on that path. Let t be a bag of $\mathcal{T}$ containing m_1 and m_2. Then t cannot lie on a path from the root of T to l, because otherwise $\chi(l)$ would have to contain either m_1 or m_2 and would, hence, have width at least 3.

Similarly, t cannot lies anywhere on the path from l to r, since every bag on that path already contains at least a and c. Also t cannot lie below r, because otherwise $\chi(r)$ contains either m_1 or m_2 and would have width at least 3. Consequently, there is a node t' on the path between l and r such that t is below t' in $\mathcal{T}$. Note that t' cannot lie above m in T, since otherwise the set of bags containing $\chi(m) \cap \chi(t)$ would not lie on a path from the root to a leave of T. Hence t' lies between m and r in T. We now distinguish two cases:

1. $\chi(m) = \{a, c, m_1\}$ or
2. $\chi(m) = \{a, c, m_2\}$.

In the former case, let t'' be a bag of T containing $\{a, m_2\}$. Since t'' contains a and m_2 it has to lie on a path from the root of T to t' and consequently $\chi(t')$ contains a, c, m_1, and m_2. In the later case, let t'' be a bag of T containing $\{c, m_1\}$. Since t'' contains a and m_1 it has to lie on a path from the root of T to t' and consequently $\chi(t')$ contains a, c, m_1, and m_2. □

Interestingly, the next two lemmas show that the pathwidth/special treewidth of the graph obtained after contracting vertices differs from the corresponding measure on the original graph by at most one.

Lemma 6.7. *Let G be a graph and let G' be a graph obtained from G via a sequence of contractions, then* $\text{sptw}(G') \leq \text{sptw}(G) \leq \text{sptw}(G') + 1$.

Proof. The fact that special treewidth is closed under contracting vertices, i.e., $\text{sptw}(G') \leq \text{sptw}(G)$ has already been shown by Courcelle [Cou12, Proposition 20]; we only include the proof here for the convenience of the reader. Towards showing that $\text{sptw}(G') \leq \text{sptw}(G)$, let (T, χ) be a special tree decomposition of G and let v be a vertex of degree two in G. We will show how to convert (T, χ) into a special tree decomposition (T^v, χ^v) with the same width as (T, χ) of the graph G^v obtained from G after contracting v.

Since, G' is obtained from G by a sequences of such contractions this shows that $\text{sptw}(G') \leq \text{sptw}(G)$. Let u and w be the two neighbors of v in G and let t_v be the bag of T containing v that is furthest away from the root of T. W.l.o.g. we can assume that t_v contains u or w. This is because if t_v does not contain u or w, we can remove v from t_v, without violating any of the properties of a special tree decomposition. Hence suppose that w.l.o.g. t_v contains w. Then it is straightforward to verify that (T^v, χ^v) can be obtained from (T, χ) by replacing every occurrence of v in a bag of T with the vertex w, i.e., $T^v = T$ and $\chi^v(t) = \chi(t)$ for every $t \in V(T)$ with $v \notin \chi(t)$ and $\chi^v(t) = \chi(t) \setminus \{v\} \cup \{w\}$ otherwise. □

It remains to show that $\text{sptw}(G) \leq \text{sptw}(G') + 1$. Note that every edge $\{u, v\} \in E(G')$ is either an edge of G or corresponds to a path between u and v in G. Then we obtain a special tree decomposition for G of width at most $\text{sptw}(G') + 1$ by exhaustively applying the following claim to an optimal special tree decomposition for G', i.e., we apply the following claim for every edge $\{u, v\} \in E(G')$ that corresponds to a path between u and v in G.

Claim 6.2. *Let H be a graph having two vertices u and v that are connected by a path P whose inner vertices all have degree exactly two in H and let H' be the graph obtained from H after contracting all inner vertices of P. Then H has a special tree decomposition $\mathcal{T}$ of width at most* $\text{sptw}(H') + 1$ *that additionally satisfies:*

(*) *for every edge $e' \in E(H') \setminus \{u,v\}$ the bag of $\mathcal{T}$ containing both endpoints of e' that is furthest away from the root has width at most* $\mathrm{sptw}(H')$.

Proof. We will show how to obtain $\mathcal{T} = (T, \chi)$ from an optimal special tree decomposition $\mathcal{T}' = (T', \chi')$ for H'. Let $t \in V(T')$ be the unique bag of $\mathcal{T}'$ containing u and v that is closest to the root and assume that $P = (v_1, \dots, v_l)$ with $v_1 = u$, $v_l = v$, and v_i has degree exactly two in H for every i with $1 < i < l$. We distinguish two cases (1) t is the root of T', or (2) t has a parent p in T'.

In the former case, T is obtained from T' by adding the path $(t_1, \dots, t_{l-2}, t)$ and making t_1 the new root of T', where $t_1, \dots, t_{l-2}$ are new nodes. Moreover, χ is obtained by setting $\chi(t') = \chi'(t')$ for every $t' \in V(T')$ and $\chi(t_i) = \{v_1, v_{i+1}, v_{i+2}\}$, for every i with $1 \leq i \leq l-2$. It is straightforward to verify that (T, χ) is indeed a special tree decomposition for H of width at most $\mathrm{sptw}(H') + 1$ that additionally satisfies (*).

In the later case, $\{u, v\} \subseteq \chi(p)$ and w.l.o.g. we can assume that $u \notin \chi(p)$. We now obtain T from T' by replacing the edge between p and t in T' by the path $(p, t_1, \dots, t_{l-1}, t)$, where $t_1, \dots, t_{l-1}$ are new nodes. Moreover, χ is obtained by setting $\chi(t') = \chi'(t')$ for every $t' \in V(T')$ and $\chi(t_i) = (\chi(t) \setminus \{u\}) \cup \{v_i, v_{i+1}\}$, for every i with $1 \leq i \leq l-1$. It is straightforward to verify that (T, χ) is indeed a special tree decomposition for H of width at most $\mathrm{sptw}(H') + 1$ that additionally satisfies (*). ☐

Using the same proof as in the proof of Lemma 6.7, we obtain the corresponding result for pathwidth.

Corollary 6.4. *Let G be a graph and let G' be a graph obtained from G via a sequence of contractions, then* $\mathrm{pw}(G') \leq \mathrm{pw}(G) \leq \mathrm{pw}(G') + 1$

In the following we consider the bi-connected components preprocessing procedure. The graph G_1 [BKK+17] shows that the special treewidth of a graph can be one more than the maximum special treewidth of any of its bi-connected components.

Lemma 6.8 ([BKK+17])**.** *There is a graph G with* $\mathrm{sptw}(G) = (\max_{i=1}^{n} \mathrm{sptw}(B_i)) + 1$, *where $B_1, \dots, B_l$ are the bi-connected components of G.*

We will now show that even though the bi-connected preprocessing procedure does not preserve special treewidth exactly, it can lead to a difference of at most plus one. Note that the result was claimed without a proof by Courcelle [Cou12, Proposition 25].

Lemma 6.9. *Let G be a graph and $B_1, \dots, B_l$ its bi-connected components. Then* $\max_{i=1}^{n} \mathrm{sptw}(B_i) \leq \mathrm{sptw}(G) \leq (\max_{i=1}^{n} \mathrm{sptw}(B_i)) + 1$.

Proof. Since each B_i is a subgraph of G, it follows that $\max_{i=1}^{n} \mathrm{sptw}(B_i) \leq \mathrm{sptw}(G)$. Towards showing that $\mathrm{sptw}(G) \leq (\max_{i=1}^{n} \mathrm{sptw}(B_i)) + 1$, let B be the BC-tree of G. We will show how to construct a special tree decomposition of G of width at most

$(\max_{i=1}^{n} \text{sptw}(B_i)) + 1$ via a bottom-up dynamic programming algorithm on B. The lemma now follows by applying the following claim to every block node of B in a bottom-up manner. □

Claim 6.3. *Let b be a block node of B such that for every block node c that is a child of a child of b in B there is a special tree decomposition of $G[B_c]$ of width at most $(\max_{c' \in B(B_c)} \text{sptw}(c')) + 1$ such that all bags containing at least one vertex in $c \setminus C(c)$ have width at most $\text{sptw}(c)$. Then there is a special tree decomposition of $G[B_b]$ of width at most $(\max_{b' \in B(B_b)} \text{sptw}(b')) + 1$ such that all bags containing vertices in $b \setminus C(b)$ have width at most $\text{sptw}(b)$.*

Proof. Let (T', χ') be an optimal special tree decomposition of b and let a be a child of b in B, who itself has children $c_1, \dots, c_r$ in B. We will start by constructing a special tree decomposition (T^a, χ^a) of $G[B_a]$ of width at most $(\max_{b' \in B(B_a)} \text{sptw}(b')) + 1$ that contains a in its root bag.

For every i with $1 \leq i \leq r$, let (T_i, χ_i') be a special tree decomposition of $G[B_{c_i}]$ with root r_i satisfying the conditions given in the statement of the claim and let t_i^1 and t_i^2 be the node in T_i closest respectively furthest away from the root r_i that contains a. Let (T_i, χ_i) be the special tree decomposition obtained from (T_i, χ') after adding a to every bag on the path from t_i^1 to the root r_i of T_i. Then (T^a, χ^a) is obtained from the disjoint union of $(T_1, \chi_1), \dots, (T_r, \chi_r)$ after adding the edges $\{t_1^2, r_2\}, \dots, \{t_{r-1}^2, r_r\}$ and identifying r_1 as the root of T^a.

Finally, the desired special tree decomposition (T, χ) for $G[B_b]$ is obtained from the disjoint union of (T', χ') and (T^a, χ^a) for every child a of b in B after adding the edges $\{t^a, r^a\}$, where t^a is the bag in (T', χ') containing a that is furthest away from the root of T' and r^a is the root of (T^a, χ^a). It is straightforward to verify that (T, χ) is a special tree decomposition of $G[B_b]$ of width at most $(\max_{b' \in B(B_b)} \text{sptw}(b')) + 1$ such that all bags containing vertices in $b \setminus C(b)$ have width at most $\text{sptw}(b)$. □

The following lemma shows that, in contrast to special treewidth, the bi-connected component preprocessing procedure cannot used be used for pathwidth.

Lemma 6.10. *For every $n \in \mathbb{N}$, there is a graph T_n such that $\text{pw}(T_n) = n$, but the pathwidth of any bi-connected component of T_n is zero.*

Proof. It is well known that for every $n \in \mathbb{N}$ there is a tree T_n with $\text{pw}(G_n) = n$ [Die95]. The lemma now follows because every bi-connected component of a tree has pathwidth zero. □

We now turn our attention to the simplicial and almost simplicial preprocessing procedure. Namely, we will show that neither can be employed for pathwidth or special treewidth.

Lemma 6.11. *For every $n \in \mathbb{N}$, there is a graph G_n with $\text{sptw}(G_n) = n$ such that there is an ordering $v_1, \ldots, v_n$ of the vertices of G such that for every i with $1 \leq i \leq n$ the at most two neighbors of v_i in $\{ v_{i+1}, \ldots, v_n\}$ form a clique in G.*

Proof. It is well known that for every $n \in \mathbb{N}$ there is a binary tree T_n with $\text{pw}(T_n) = n$ [Die95]. Let G_n be obtained from T_{n-1} after adding a new vertex a and making it adjacent to every vertex in T_n. Then $\text{pw}(G_n) = n$ and moreover because of Corollary 6.1 also $\text{sptw}(G_n) = n$. Note that every leaf of T_{n-1} is simplicial and has degree two in G_n. The desired ordering of $V(G_n)$ can hence be obtained by iteratively removing leaves from T_{n-1}. □

Since the graph G_n given in the proof of Lemma 6.11 above contains the apex vertex a, we obtain from Corollary 6.1.

Corollary 6.5. *For every $n \in \mathbb{N}$, there is a graph G_n with $\text{pw}(G_n) = n$ such that there is an ordering $v_1, \ldots, v_n$ of the vertices of G such that for every i with $1 \leq i \leq n$ the at most two neighbors of v_i in $\{ v_{i+1}, \ldots, v_n\}$ form a clique in G.*

6.4 Partition-Based Approach for Special Treewidth

In this section we introduce a novel characterization of special treewidth, in terms of special derivations. The characterization is inspired by the partition-based approaches employed for branchwidth and clique-width [HS15, LOS16a].

6.4.1 Characterization: Special Derivations

Let $G = (V, E)$ be a graph. A *special derivation* $\mathcal{P}$ of G of *length* l is a sequence $(P_1, \ldots, P_l)$ of weak partitions of V such that:

(SD1) $\text{U}(P_1) = V$,

(SD2) for every $i \in \{1, \ldots, l-1\}$, P_i is a refinement of P_{i+1}, and

(SD3) for every edge $\{u, v\} \in E$ it holds that there is a P_i and a set $p \in P_i$ such that $\{u, v\} \subseteq p$.

The *width* of $\mathcal{P}$ is the maximum size of any set in $P_1 \cup \ldots \cup P_l$ minus 1. We will refer to P_i as the i-th *level* of $\mathcal{P}$ and we will refer to elements in $\bigcup_{1 \leq i \leq l} P_i$ as *sets* of $\mathcal{P}$. We will show that any special tree decomposition can be transformed into a special derivation of the same width and vice verse. The following example illustrates the close connection between special tree decompositions and special derivations.

Example 7. *Consider the special tree decomposition $\mathcal{T}$ given in Figure 6.1. Then $\mathcal{T}$ can, e.g., be translated into the special derivation $\mathcal{P} = (P_1, \ldots, P_4)$ defined by setting*

$P_1 = \{\{1\}, \{2, 3\}, \{4, 5\}, \{6, 7\}\},$

$P_2 = \{\{1, 2\}, \{4, 5\}, \{6, 7\}\},$

$P_3 = \{\{1, 4\}, \{6, 7\}\},$

$P_4 = \{\{1, 6\}\}.$

The width of $\mathcal{T}$ is equal to the width of $\mathcal{P}$.

The following theorem shows that special derivations provide an alternative characterization of special tree decompositions. The main observation behind the proof of the equivalence between the two characterizations is that after padding the special tree decomposition such that every leaf has the same distance from the root, it holds that the weak partition on a certain level of a special derivation is given by the set of bags that are at the same distance from a leaf in a special tree decomposition and vice versa.

Theorem 6.2. *A graph G has a special tree decomposition of width at most ω and height at most h if and only if G has a special derivation of width at most ω and length at most h.*

Proof. Let $\mathcal{T} = (T, \chi)$ be a special tree decomposition of G with root r of width at most ω. For a node $t \in V(T)$, we denote by T_t the subtree of T rooted at t and we denote by $h(t)$ the height of T_t, i.e., the longest path from t to any leaf of T_t in T_t with $h(t) = 1$ if T_t consists only of t.

We first observe that w.l.o.g. we can assume that all paths from the root r of T to a leaf of T have the same length $h(T_r)$. Suppose not and let $l \in V(T)$ be a leaf at distance i with $i < h(t) - 1$ from the root. We simply attach to l a path with $h(t) - (i + 1)$ novel bags all containing $\chi(l)$. By iterating this process for every such leaf, we obtain a new special tree decomposition of the same width that satisfies the property.

We denote by L_i the set $\bigcup_{t \in V(T) \wedge h(t)=i} \chi(t)$ and by $L_{>i}$ the set $(\bigcup_{i \leq j \leq h(t)} L_j) \setminus L_i$. We claim that $\mathcal{P} = (P_1, \ldots, P_{h(T)})$ with $P_i = \{\, \chi(t) \mid t \in V(T) \wedge h(t) = i \,\} \cup \{\, \{v\} \mid v \in L_{>i} \,\}$, for every i with $1 \leq i \leq h(T)$ is a special derivation of G with width at most ω and length at most $h(t)$. By construction the width of $\mathcal{P}$ is at most ω and its length is at most $h(T)$. Thus it remains to show that $\mathcal{P}$ is a special derivation. Note that for every i with $1 \leq i \leq h(t)$, it holds that the sets $\chi(t)$ for every node $t \in V(T)$ with $h(t) = i$ are pairwise disjoint, since otherwise Property *(ST)* of $\mathcal{T}$ would be violated. It follows that P_i is a weak partition for every i with $1 \leq i \leq h(T)$. Moreover, by construction $\mathcal{P}$ satisfies *(SD1)* and because of *(T2)* also *(SD3)*.

Towards showing *(SD2)* assume for a contradiction that this is not the case, i.e., there is an i with $1 \leq i < l$ such that P_i is not a refinement of P_{i+1}. It follows that there are two

vertices u and v that occur in the same set of P_i but in distinct sets of P_{i+1}. Hence there is a bag $t \in V(T)$ with $h(t) = i$ containing both u and v and there are two bags t_1 and t_2 in T with $h(t_1) = h(t_2) = i + 1$ such that $u \in \chi(t_1)$ and $v \in \chi(t_2)$. However, this is not possible since otherwise either t would have to have two parents in T or one of u or v would violate Property *(ST)*.

Let $\mathcal{P} = (P_1, \ldots, P_l)$ be a special derivation of G of width at most ω. W.l.o.g. we can assume that $P_l \neq \emptyset$ since otherwise we could delete P_l from $\mathcal{P}$ while maintaining all properties of a special derivation. Let T be the tree with one vertex t_p for every $p \in \bigcup_{1 \leq i \leq l} P_i$ and that has the following edges:

(E1) For every $p \in P_i$ and every $p' \in P_{i+1}$ with $P \cap P' \neq \emptyset$ an edge between t_P and $t_{P'}$,

(E2) For every i with $1 \leq i \leq l$ let $p_1, \ldots, p_a$ be all sets in P_i for which $p_j \cap \mathrm{U}(P_{i+1}) = \emptyset$, where $1 \leq j \leq a$, then T contains the edges $\{p_1, p_2\}, \{p_2, p_3\}, \ldots, \{p_{a-1}, p_a\}$. Moreover, if there is a set $p \in P_i$ with $p \cap \mathrm{U}(P_{i+1}) \neq \emptyset$, then T additionally contains the edge $\{p, p_1\}$.

Let r be an arbitrary node t_p such that $p \in P_l$. We claim that $\mathcal{T} = (T, \chi)$ with $\chi(t_p) = p$ and root r is a special tree decomposition of G with width at most ω and height at most l. By construction the width of $\mathcal{T}$ is at most ω and its height is at most l. Thus it remains to show that $\mathcal{T}$ satisfies the properties *(T1)*, *(T2)*, and *(ST)* of a special tree decomposition. Because $\mathrm{U}(P_1) = V$, we obtain that $\mathcal{T}$ satisfies *(T1)*. Moreover, because of *(SD3)* the same applies to *(T2)*.

Finally, consider a vertex $v \in V$. Because every P_i is a weak partition, it holds that v is contained in at most one set of P_i for every i with $1 \leq i < l$. Furthermore, because $\mathrm{U}(P_{i+1}) \subseteq \mathrm{U}(P_i)$ it holds that once v does not occur in P_i it will also not occur in P_j for any $j > i$. It now follows from the edges added in *(E1)* that the set of bags in T containing v forms a subpath of T from r to the leaf $t_{\{v\}}$. □

The following theorem allows us to restrict the search to derivations of length at most $|V(G)| - \omega$. The proof is mainly based on the previous theorem together with the observation that a restricted form of tree decompositions, so called small tree decompositions, can be shown to have bounded at most $|V(G)| - \omega$.

Theorem 6.3. *Let G be a graph and ω an integer. Then the special treewidth of G is at most ω if and only if G has a special derivation of width at most ω and length at most $|V(G)| - \omega$.*

Proof. The backward direction of the claim follows immediately from Theorem 6.2.

Towards showing the forward direction we first show that every special tree decomposition of width at most ω can be transformed into a special tree decomposition of the same width and whose height is at most $|V(G)| - \omega$. The claim then follows from Theorem 6.2.

It is well-known that any tree decomposition can be transformed in polynomial-time into a so called small tree decomposition of the same width [Klo94]. A *small tree decomposition* is a tree decomposition (T, χ) with the following additional property:

(STD) For every two nodes $t, t' \in V(T)$ it holds that $\chi(t) \nsubseteq \chi(t')$.

We will first show that the same holds for any special tree decomposition and moreover that there is a small special tree decomposition of the same width whose leaves are full, i.e., all leaf bags contain $\omega + 1$ vertices of G. Towards showing this let (T, χ) be a special tree decomposition of the graph G with width ω. We first show that we can assume that every leaf is full. Suppose not and let $l \in V(T)$ be a leaf of T with $|\chi(l)| < \omega + 1$ and parent $p \in V(T)$. If $|\chi(p) \cup \chi(l)| \leq \omega + 1$, then we replace p and l in T with a new node t that is adjacent to all neighbors of p in T (apart from l) and set $\chi(t) = \chi(p) \cup \chi(l)$. It is straightforward to verify that the result is still a special tree decomposition of G of width ω that contains one less leaf l with $|\chi(p) \cup \chi(l)| \leq \omega + 1$. Hence by repeating this procedure exhaustively, we obtain a special tree decomposition of G with width ω such that $|\chi(p) \cup \chi(l)| > \omega + 1$ for any leaf l with parent t.

It remains to show how to deal with those leaves, i.e., let $l \in V(T)$ be a leaf with parent $p \in V(T)$ such that $|\chi(p) \cup \chi(l)| > \omega + 1$ and $|\chi(l)| < \omega + 1$. Then we make l full by adding $\omega + 1 - |\chi(l)|$ vertices from $\chi(p) \setminus \chi(l)$ to $\chi(l)$. This shows that we can assume that all leaves of (T, χ) are full. We show next that we can also ensure that (T, χ) satisfies *(STD)*.

Suppose not, then there are two distinct nodes $t, t' \in V(T)$ such that $\chi(t) \subseteq \chi(t')$. Because of Property *(ST)* it follows that $\chi(t)$ is a subset of $\chi(t'')$ for every node t'' on the unique path from t to t' in T. Hence w.l.o.g. we can assume that t and t' are adjacent in T. It is now straightforward to verify that the tree decomposition (T', χ') that is obtained from (T, χ) by replacing t and t' in T with a new node t'' making t'' adjacent to all neighbors of t and t' in T and setting $\chi'(t'') = \chi(t')$ is a special tree decomposition of G with same width as (T, χ) that has one less pair of nodes violating *(STD)*.

By repeating this procedure until there are no more pairs of nodes violating *(STD)*, we obtain a small special tree decomposition of G with width ω. Because this operation also does not introduce novel non-full leaves, we can in the following assume that (T, χ) is a small special tree decomposition whose leaves are all full and we are now ready to show that every path from the root r of T to a leaf l of T has length at most $|V(G)| - \omega$. Because of Property *(STD)* together with Property *(ST)*, we obtain that for every node p on P, the bag $\chi(p)$ has to contain a vertex that is not contained in any bag below p in T. Moreover, because $\chi(l)$ contains $\omega + 1$ vertices, we obtain that the length of the path from r to l in T is at most $|V(G)| - (\omega + 1) + 1 = |V(G)| - \omega$, as required. □

6.4.2 SAT-Encoding of a Special Derivation

Here we will provide our encoding for special derivations. Namely, we will construct a CNF formula $F(G, \omega, l)$ that is satisfiable if and only if G has a special derivation of

width at most ω and length at most l. Because of Theorem 6.3 (after setting l to the value specified in the theorem) it holds that $F(G, \omega, n - \omega)$ is satisfiable if and only if G has special treewidth at most ω. To achieve this aim we first construct a formula $F(G, l)$ that is satisfiable if and only if G has a special derivation of length at most l

The formula $F(G, l)$ uses a *set variable* $\text{set}(u, v, i)$, for every $u, v \in V(G)$ and i with $u \leq v$ and $1 \leq i \leq l$. Informally, $\text{set}(u, v, i)$ is true whenever either $u \neq v$ and u and v are contained in the same set at level i of the special derivation or $u = v$ and u is contained in some set at level i. We now describe the clauses of the formula. The following clauses ensure transitive relation between two vertices $u, v \in V(G)$ defined by $\text{set}(u, v, i)$ for every i with $1 \leq i \leq l$.

$$\begin{aligned}
&(\neg\text{set}(u, v, i) \vee \neg\text{set}(u, w, i) \vee \text{set}(v, w, i)) \\
&\wedge(\neg\text{set}(u, v, i) \vee \neg\text{set}(v, w, i) \vee \text{set}(u, w, i)) \\
&\wedge(\neg\text{set}(u, w, i) \vee \neg\text{set}(v, w, i) \vee \text{set}(u, v, i)) \\
&\wedge(\neg\text{set}(u, v, i) \vee \neg\text{set}(u, u, i)) \qquad \text{for } u, v, w \in V(G),\ u < v < w,\ 1 \leq i \leq l.
\end{aligned}$$

To ensure Property *(SD1)*, we add the clause $\text{set}(u, u, 1)$ for every $u \in V(G)$.

The following clauses ensure *(SD2)*, i.e., P_i is a refinement of P_{i+1} for every $1 \leq i < l$.

$$\begin{aligned}
&(\neg\text{set}(u, u, i+1) \vee \neg\text{set}(v, v, i+1) \vee \text{set}(u, v, i+1) \vee \neg\text{set}(u, v, i)) \\
&\wedge(\text{set}(u, u, i) \vee \neg\text{set}(u, u, i+1)) \qquad \text{for } u, v \in V(G),\ u < v,\ 1 \leq i < l
\end{aligned}$$

Towards presenting the clauses employed to ensure *(SD3)*, we will use the following property that is easily seen to be equivalent to *(SD3)*.

(SD3') For every edge $\{u, v\} \in E$, it holds that:

- if there is an i with $1 \leq i < l$ such that $u, v \in \text{U}(P_i)$ and $v \notin \text{U}(P_{i+1})$, then $u, v \in p$ for some $p \in P_i$ and
- if $u, v \in \text{U}(P_l)$, then $u, v \in p$ for some $p \in P_l$.

Note that *(SD3)* and *(SD3')* are equivalent because whenever there is a set $p \in P_i$ for some i with $1 \leq i \leq l$ containing two vertices u and v, then such a set also exists in every P_j for $j \geq i$ as long as $u, v \in \text{U}(P_j)$. The following clauses now ensure *(SD3')* and thereby *(SD3)*.

$$\begin{aligned}
&((\neg\text{set}(u, u, i) \vee \neg\text{set}(v, v, i) \vee \text{set}(u, u, i+1)) \vee \text{set}(u, v, i)) \\
&\wedge((\neg\text{set}(u, u, i) \vee \neg\text{set}(v, v, i) \vee \text{set}(v, v, i+1)) \vee \text{set}(u, v, i)) \\
&\wedge((\neg\text{set}(u, u, l) \vee \neg\text{set}(v, v, l)) \vee \text{set}(u, v, l)) \\
&\qquad \text{for } e \in E(G),\ u, v \in e,\ u < v,\ 1 \leq i < l
\end{aligned}$$

We are now ready to extend $F(G, l)$ to the formula $F(G, \omega, l)$. We achieve this by restricting the sizes of all sets in P_i for every $1 \leq i \leq l$ to be at most $\omega + 1$, or in other words for every $v \in V(G)$ and i with $1 \leq i \leq l$, we need to restrict the number of variables $\text{set}(v, u, i)$ set to true to be at most $\omega + 1$. We achieve this by using the sequential counter approach described in Subsection 2.1.1. The obtained formula $F(G, l, \omega)$ contains $\mathcal{O}(n^3\omega)$ variables and $\mathcal{O}(n^4 + mn^3)$ clauses.

Exploiting Twins To exploit the twin preprocessing rule for our encoding, we add the following clauses, which ensure that the order in which twin vertices are forgotten cannot be arbitrary but corresponds to $<$.

$$\text{set}(u, u, i) \vee \neg\text{set}(v, v, i)$$
$$\text{for } u, v \in V(G),\ N_G(u) = N_G(v) \text{ or } N_G[u] = N_G[v],\ u < v,\ 1 \leq i \leq l.$$

6.5 Ordering-Based Approach for Special Treewidth

In this section we introduce a second characterization of special treewidth, namely special elimination orderings, inspired by elimination orderings characterizing treewidth [Klo94].

6.5.1 Characterization: Special Elimination Orderings

We start by introducing an ordering-based characterization of treewidth and then show how to adapt the notion in the context of special treewidth. Towards this aim we start with a slightly non-standard definition of elimination orderings for treewidth, from which it is particularly easy to obtain our adaptation for special treewidth.

Let G be a graph with n vertices and let $\leq_S$ be a total order $(v_1, \dots, v_n)$ of the vertices of G. For two vertices u and v with $u \leq_S v$ we denote by $N_G^{\leq_S}(u, v)$ the set of all neighbors of u in G that are larger than v w.r.t. $\leq_S$. We extend this notation to sets $U \subseteq V(G)$, where $u \leq_S v$ for every $u \in U$, by setting $N_G^{\leq_S}(U, v)$ to be the set $\bigcup_{u \in U} N_G^{\leq_S}(u, v)$.

We next define the sequence $G_0^{\leq_S}, \dots, G_{n-1}^{\leq_S}$ of supergraphs of G inductively as follows: We set $G_0^{\leq_S} = G$ and for every i with $1 \leq i < n$ we let $G_i^{\leq_S}$ be the graph obtained from $G_{i-1}^{\leq_S}$ after adding all edges in the set $E_i^{\leq_S}$, which is defined as follows. Let $\mathcal{C}_i^{\leq_S}$ be the set of all components of the graph $G_{i-1}[v_1, \dots, v_{i-1}, v_i]$. Then $E_i^{\leq_S}$ is the set $\{\, \{u, v\} \mid u, v \in N_{G_{i-1}}^{\leq_S}(C, v_i) \wedge C \in \mathcal{C}_i^{\leq_S} \,\}$.

We call $G_{\leq_S} = G_{n-1}^{\leq_S}$ the *fill-in graph* of G w.r.t. $\leq_S$ and $G_i^{\leq_S}$ the *i-th fill-in graph* of G w.r.t. $\leq_S$. Then any total ordering $\leq_S$ gives rise to an *elimination ordering* of G and the *width* of an elimination ordering $\leq_S$ is the maximum of $\max\{\, |N_{G_{\leq_S}}^{\leq_S}(C, v_i)| \mid C \in \mathcal{C}_i^{\leq_S} \,\}$ over all i with $1 \leq i < n$. Furthermore, the *elimination width* of a graph G is the minimum width of any elimination ordering of G. It is known that the elimination width of a graph is equal to the treewidth of a graph [Klo94].

We are now ready to show how to adapt elimination orderings for special treewidth. Informally, the crucial observation here is that because of Property *(ST)* a special tree decomposition, in contrast to a normal tree decomposition, cannot have separate branches for components that have at least one common neighbor. This property directly translates to elimination orderings in the sense that whenever two components C and C' in $\mathcal{C}_i^{\leq_S}$ share a neighbor that comes later in the ordering, they need to be handled together both for obtaining the fill-in edges as well as for determining the width of the ordering.

To formalize this idea, we say that two components C and C' in $\mathcal{C}_i^{\leq_S}$ *clash* if $N^{\leq_S}_{G_{i-1}}(C, v_i) \cap N^{\leq_S}_{G_{i-1}}(C', v_i) \neq \emptyset$. Moreover, let H be the graph with vertex-set $\mathcal{C}_i^{\leq_S}$ having an edge between two vertices C and C' if and only if their associated components clash and let $\mathcal{P}_i^{\leq_S}$ be the partition of $\mathcal{C}_i^{\leq_S}$ that corresponds to the connected components of H. Then *special elimination orderings* are obtained from elimination orderings by using $\mathcal{P}_i^{\leq_S}$ instead of $\mathcal{C}_i^{\leq_S}$ to determine both the fill-in edges as well as the width of the ordering. Formally, for special elimination orderings the set $E_i^{\leq_S}$ becomes $\{\, \{u, v\} \mid u, v \in N^{\leq_S}_{G_{i-1}}(P, v_i) \wedge P \in \mathcal{P}_i^{\leq_S} \,\}$ and the width of $\leq_S$ becomes the maximum of $\max\{\, |N^{\leq_S}_{G_{\leq_S}}(P, v_i)| \mid P \in \mathcal{P}_i^{\leq_S} \,\}$ over all i with $1 \leq i < n$.

We show next that special elimination orderings properly characterize special treewidth. The main ideas behind the proof of the theorem are similar to the proof showing the equivalence between eliminations orderings and treewidth [Klo94], however, the proof is significantly more involved due to the properties of special treewidth.

Theorem 6.4. *A graph G has a special tree decomposition of width at most ω if and only if G has a special elimination ordering of width at most ω.*

Proof. Towards showing the forward direction let $\mathcal{T} = (T, \chi)$ be a special tree decomposition of G with root r of width at most ω. For a vertex $v \in V(G)$ let $F_T(v)$ be the unique node $t \in V(T)$ such that $v \in \chi(t)$ but not $v \in \chi(p)$, where p is the unique parent of t in T. Let $\leq'_S$ be the partial ordering of $V(G)$ defined by setting $u \leq'_S v$ if and only if $u = v$ or $F_T(v)$ is on the unique path from $F_T(u)$ to r in T.

We claim that any linear extension $\leq_S$ of $\leq'_S$ is a special elimination ordering for G of width at most ω. We start by showing via induction that $\mathcal{T}$ is also a special tree decomposition of $G_i^{\leq_S}$ for every i with $0 \leq i \leq n-1$. The claim clearly holds for $G_0^{\leq_S} = G$. So suppose the claim holds for $G_i^{\leq_S}$. It suffices to show that for every edge $e = \{n_1, n_2\}$ in $E_i^{\leq_S}$ there is a node $t \in V(T)$ with $e \subseteq \chi(t)$. Because $e \in E_i^{\leq_S}$, it holds that $v_i <_S n_1, n_2$ and there are $a, b \in P \in \mathcal{P}_i^{\leq_S}$ with $a, b \leq_S v_i$ and $n_1, n_2 \in N_{G_{i-1}}(a) \cup N_{G_{i-1}}(b)$. W.l.o.g. let us assume that $\{n_1, a\} \in E(G_{i-1}^{\leq_S})$ and $\{n_2, b\} \in E(G_{i-1}^{\leq_S})$. Because χ is a tree decomposition of $G_{i-1}^{\leq_S}$, we obtain that there are nodes t_a and t_b with $\{n_1, a\} \subseteq \chi(t_a)$ and $\{n_2, b\} \in \chi(t_b)$. Hence together with the fact that $a, b \leq_S n_1, n_2$ it follows that $a \leq'_S n_1$ and $b \leq'_S n_2$. We claim that $a \leq'_S n_2$ and symmetrically $b \leq'_S n_1$ and hence either $n_1 \leq'_S n_2$ or $n_2 \leq'_S n_1$, which in the first case implies that $n_1, n_2 \in \chi(F_T(n_1))$ and in the second case implies that $n_1, n_2 \in \chi(F_T(n_1))$.

Towards this aim we first show that if C is a component in $\mathcal{C}_i^{\leq_S}$ and $n \in N^{\leq_S}_{G_{i-1}}(C, v_i)$, then for every $v \in V(C)$ it holds that $v \leq'_S n$. Assume this is not the case and let $v \in V(C)$ witness this, i.e., it does not hold that $v \leq'_S n$ and hence because $v \leq_S n$ it does also not hold that $n \leq'_S v$. Let $a \in V(C)$ be a neighbor of n (which must exist since $n \in N^{\leq_S}_{G_{i-1}}(C, v_i)$). Then because χ is a tree decomposition of $G^{\leq_S}_{i-1}$ there is a node $t \in V(T)$ such that $a, n \in \chi(t)$.

Hence because $a \leq_S n$, we obtain that $a \leq'_S n$. Let C' be the subset of C such that $c \leq'_S n$ for every $c \in C'$. Then $\mathcal{T}$ cannot contain a bag that contains both a vertex from C' and a vertex from $C \setminus C'$. Because $a \in C'$ it holds that C' is non-empty and moreover because $v \notin C'$ also $C \setminus C'$ is non-empty. But this contradicts our assumption that $\mathcal{T}$ is a tree decomposition of G_{i-1} because C is a component and hence there has to exists at least one edge between a vertex in C' and a vertex in $C \setminus C'$ (but this edge cannot be covered by $\mathcal{T}$).

Note that the above also implies that if C and C' are two components in $\mathcal{C}_i^{\leq_S}$ that clash say in some vertex $n \in N^{\leq_S}_{G_{i-1}}(C, v_i) \cap N^{\leq_S}_{G_{i-1}}(C', v_i)$, then $v \leq'_S n$ for every $v \in V(C) \cup V(C')$. We will show next that the same holds for any vertex $n' \in N^{\leq_S}_{G_{i-1}}(C, v_i) \cup N^{\leq_S}_{G_{i-1}}(C', v_i)$, i.e., for any such vertex n' we have $v \leq'_S n'$ for every $v \in V(C) \cup V(C')$. W.l.o.g. let us assume that $n' \in\in N^{\leq_S}_{G_{i-1}}(C, v_i) \setminus N^{\leq_S}_{G_{i-1}}(C', v_i)$ (the other case is analogous). If $n \leq'_S n'$ then the claim holds hence we can assume that $n' <'_S n$. Then $v \leq'_S n'$ for every $v \in V(C)$. Moreover, because n has a neighbor in C, we obtain that $n \in \chi(F_T(n'))$. Because $v \leq_S n'$ for every vertex $v \in V(C')$ it cannot hold that $n' \leq'_S v$. Moreover, if there is a vertex $v \in V(C')$ with $v \leq'_S n'$, then this has to hold for all vertices in $V(C')$ (same argument as above because it is a component).

Consequently we can assume that neither $v \leq'_S n'$ not $n' \leq'_S v$ for every $v \in V(C')$. This implies that for every $v \in V(C')$, the node $F_T(v)$ must be in some branch below $F_T(n)$ other than the branch rooted by $F_T(n')$. However because n is a neighbor of some vertex in C', this branch has to contain at least one bag that also contains n contradicting our assumption that $\mathcal{T}$ is a special tree decomposition for G_{i-1}. Note that this argument can now be easily extended to sets of components contained in $\mathcal{P}_i^{\leq_S}$, i.e., we obtain that for every $P \in \mathcal{P}_i^{\leq_S}$ and every vertex $n \in N^{\leq_S}_{G_{i-1}}(P, v_i)$ it holds that $p \leq'_S n$ for every $p \in P$.

This concludes the proof that $\mathcal{T}$ is a special tree decomposition of $G_i^{\leq_S}$ and by induction of $G_{\leq_S}$ and it remains to show that $\leq_S$ has width at most ω. To see that this is indeed the case note that the width of $\leq_S$ is the maximum size of some sets of vertices that form a clique in $G_{\leq_S}$ and because $\mathcal{T}$ is a tree decomposition it has to contain a bag containing any clique in $G_{\leq_S}$. Hence the width of $\leq_S$ is at most $\omega - 1$.

Towards showing the backward direction let $\leq_S = (v_1, \dots, v_n)$ be a special elimination ordering of G of width at most ω. For every i with $\omega \leq i \leq n$, we will iteratively construct special tree decompositions $\mathcal{T}_i = (T_i, \chi_i)$ of $G_{\leq_S}[v_{n-i}, \dots, v_n]$ satisfying:

(*) For every $P \in \mathcal{P}^{\leq_S}_{n-i-1}$, T_i has a unique leaf l with $N^{\leq_S}_{G_{\leq_S}}(P, v_{n-i-1}) \subseteq \chi_i(l)$.

We start by defining T_ω as follows. T_ω has a root node r and for every $P \in \mathcal{P}^{\leq_S}_{n-\omega-1}$ one leaf node l_P. Moreover, we set $\chi_\omega(r) = \{v_{n-\omega}, \ldots, v_n\}$ and for every $P \in \mathcal{P}^{\leq_S}_{n-\omega-1}$ we set $\chi(l_P) = N^{\leq_S}_{G_{\leq_S}}(P, v_{n-\omega-1})$. Note that χ_ω is a clearly a tree decomposition for $G_{\leq_S}[v_{n-\omega}, \ldots, v_n]$ of width at most ω satisfying (*). Moreover, because the sets $N^{\leq_S}_{G_{\leq_S}}(P, v_{n-\omega-1})$ and $N^{\leq_S}_{G_{\leq_S}}(P', v_{n-\omega-1})$ are disjoint for every $P, P' \in \mathcal{P}^{\leq_S}_{n-\omega-1}$ it is also a special tree decomposition.

Hence assume that $\mathcal{T}_i$ has already been constructed, let $P \in \mathcal{P}^{\leq_S}_{n-i-1}$ be the part containing v_{n-i-1}, and let l_P be the leaf of $\mathcal{T}_i$ containing $N^{\leq_S}_{G_{\leq_S}}(P, v_{n-i-1})$, which exists due to (*). Then we obtain T_{i+1} from T_i be adding a new node n making it adjacent to l_P and adding one leaf $l_{P'}$ (adjacent to n) for every $P' \in \mathcal{P}^{\leq_S}_{n-i-2}$ with $P' \subseteq P$. Moreover, we set $\chi_{i+1}(n) = P \cup \{v_{n-i-1}\}$ and $\chi_{i+1}(l_{P'}) = N^{\leq_S}_{G_{\leq_S}}(P', v_{n-i-2})$. Again it is straightforward to verify that $\mathcal{T}_{i+1} = (T_{i+1}, \chi_{i+1})$ is a tree decomposition of $G_{\leq_S}[v_{n-i-1}, \ldots, v_n]$ of width at most ω satisfying (*). Finally, because the sets $N^{\leq_S}_{G_{\leq_S}}(P_1, v_{n-i-2})$ and $N^{\leq_S}_{G_{\leq_S}}(P_2, v_{n-i-2})$ are disjoint for every $P_1, P_2 \in \mathcal{P}^{\leq_S}_{n-i-2}$ it is also a special tree decomposition.

Since $\mathcal{T}_{n-1}$ is a special tree decomposition of $G_{\leq_S}$ (and hence also of G) of width at most ω, this concludes the backward direction of the theorem. □

6.5.2 SAT-Encoding for Special Elimination Orderings

Here we provide our encoding for special elimination orderings as introduced in the previous subsection. In particular, we will construct a CNF formula $F(G, \omega)$ that is satisfiable if and only if G has a special elimination ordering of width at most ω. Because of Theorem 6.4 it then holds that $F(G, \omega)$ is satisfiable if and only if G has special treewidth at most ω. Towards this aim we first construct the formula $F(G)$ that is satisfiable if and only if G has a special elimination ordering and building upon $F(G)$ we will then use cardinality counters to obtain $F(G, \omega)$. For the definition of the formula we use the same notation as introduced in Section 6.5.1, i.e, we refer to the required elimination ordering by $\leq_S$, and use $\mathcal{C}^{\leq_S}_v$ and $\mathcal{P}^{\leq_S}_v$ to refer to the components and parts of the graph $G^{\leq_S}_{v-1}[1, \ldots, v]$ (recall that we assume that the vertices of G are numbered from 1 to n).

The formula $F(G)$ uses the following variables. An *order variable* $\mathrm{o}(u, v)$ for all $u, v \in V(G)$ with $u < v$. The variable $\mathrm{o}(u, v)$ will be true if and only if $u < v$ and $u \leq_S v$. The idea behind the variable $\mathrm{o}(u, v)$ is that it can used to model the total ordering $\leq_S$ witnessing the elimination width of G by requiring that $u \leq_S v$ for arbitrary $u, v \in V(G)$ if and only if $u = v$ or $u < v$ and $u \leq_S v$ or $u > v$ and $\neg\mathrm{o}(v, u)$. In order to be able to refer to $\leq_S$ in the clauses of $F(G)$, we define the "macro" $\mathrm{o}^*(u, v)$ by setting $\mathrm{o}^*(u, v) = \mathit{true}$ if $u = v$, $\mathrm{o}^*(u, v) = \mathrm{o}(u, v)$ if $u < v$ and $\mathrm{o}^*(u, v) = \neg\mathrm{o}(v, u)$ if $u > v$. Additionally, $F(G)$ contains an *arc variable* $\mathrm{a}(u, v)$ for all $u, v \in V(G)$. The variable $\mathrm{a}(u, v)$ is true if $u \leq_S v$ and $\{u, v\} \in E(G_{\leq_S})$ and moreover it is not true if $v <_S u$. Finally, $F(G)$ has a *part variable* $\mathrm{p}(u, v)$ for all $u, v \in V(G)$. The variable $\mathrm{p}(u, v)$ is true if and only if the vertices u and v belong to the same part in $\mathcal{P}^{\leq_S}_v$. Observe that whenever a vertex u belongs to

the same part as a vertex v in $\mathcal{P}_v^{\leq_S}$, then u will also be in the same part as v in $\mathcal{P}_w^{\leq_S}$ for any w with $v \leq_S w$.

We will now provide the clauses for the formula $F(G)$. The following clauses ensure that $\mathrm{o}^*(u,v)$ is a total ordering of $V(G)$ by ensuring that the relation between u and v defined by $\mathrm{o}^*(u,v)$ is transitive:

$$(\neg\mathrm{o}^*(u,v) \vee \neg\mathrm{o}^*(v,w) \vee \mathrm{o}^*(u,w)) \quad \text{for } u,v,w \in V(G) \text{ where } u,\ v, \text{ and } w \text{ are pairwise distinct.}$$

We also introduce the clause $\mathrm{a}(u,v) \vee \mathrm{a}(v,u)$ for every $\{u,v\} \in E(G)$, which ensure that at least one of $\mathrm{a}(u,v)$ or $\mathrm{a}(v,u)$ is true for every edge $\{u,v\} \in E(G)$. Towards ensuring that the ordering $\leq_S$ represented by $\mathrm{o}^*(u,v)$ is compatible with the direction of the edges given by $\mathrm{a}(u,v)$, we introduce the clause $\neg\mathrm{a}(u,v) \vee \mathrm{o}^*(u,v)$ for every $u,v \in V(G)$. Moreover, to ensure that the relation given by $p(u,v)$ is reflexive, i.e., every vertex belongs to its own part, we introduce the clause $p(v,v)$ for every $v \in V(G)$.

The following clauses ensure that if $\mathrm{p}(u,v)$ is true, then also $\mathrm{p}(w,v)$ is true for every w that is in the same component as u in $\mathcal{C}_v^{\leq_S}$. This is achieved by enforcing that whenever a vertex w with $w \leq_S v$ is connected via an edge in $G_{\leq_S}$ to some vertex u with $\mathrm{p}(u,v)$ being true, then also $\mathrm{p}(w,v)$ is true.

$$\begin{aligned}&(\neg\mathrm{a}(u,w) \vee \neg\mathrm{p}(u,v) \vee \neg\mathrm{o}^*(w,v) \vee \mathrm{p}(w,v))\\ &\wedge(\neg\mathrm{a}(w,v) \vee \neg\mathrm{p}(u,v) \vee \neg\mathrm{o}^*(w,v) \vee \mathrm{p}(w,v))\\ &\qquad\qquad \text{for } u,w,v \in V(G) \text{ and } u \neq w \text{ and } w \neq v.\end{aligned}$$

The following clauses complete the definition of $\mathrm{p}(u,v)$ by enforcing that whenever there is a vertex u with $u \leq_S v$ that shares a neighbor x with some vertex w with $\mathrm{p}(w,v)$ being true, then also $\mathrm{p}(u,v)$ is true, as u must also be in this part.

$$\neg\mathrm{a}(u,x) \vee \neg\mathrm{a}(w,x) \vee \neg\mathrm{p}(w,v) \vee \neg\mathrm{o}^*(u,v) \vee \mathrm{p}(u,v) \quad \text{for } u,w,x,v \in V(G) \text{ and } u \neq w \neq x.$$

The following clauses ensure that at least one of $\mathrm{a}(u,v)$ or $\mathrm{a}(v,u)$ is true for every "fill-in edge", i.e., for every edge in $E(G_{\leq_S}) \setminus E(G)$.

$$\begin{aligned}&\neg\mathrm{p}(u_1,v) \vee \neg\mathrm{p}(u_2,v) \vee \neg\mathrm{a}(u_1,w_1) \vee \neg\mathrm{a}(u_2,w_2) \vee \neg\mathrm{o}^*(v,w_1) \vee \neg\mathrm{o}^*(v,w_2)\\ &\vee\mathrm{a}(w_1,w_2) \vee \mathrm{a}(w_2,w_1) \qquad \text{for } u_1,u_2,w_1,w_2,v \in V(G) \text{ with } w_1 \neq w_2.\end{aligned}$$

This completes the construction of $F(G)$. Informally, the crucial parts to verify the correctness of the formula are that for any ordering of the vertices of G, which is defined by the setting of the ordering variables $\mathrm{o}(u,v)$, the formula ensures that whenever $\{u,v\} \in G_{\leq_S}$ then either $\mathrm{a}(u,v)$ or $\mathrm{a}(v,u)$ is true. This way the formula ensures that all

edges of $G_{\leq S}$ are considered for the definition of the part variables $\mathrm{p}(u, v)$, which in turn ensures the correctness of the formula.

We are now ready to construct the formula $F(G, \omega)$. To achieve this it only remains to restrict the sizes of the sets $N^{\leq S}_{G_{\leq S}}(P, v)$ to be at most ω for every $v \in V(G)$ and $P \in \mathcal{P}^{\leq S}_v$. Indeed we need to restrict the number of vertices w satisfying the formula $\mathrm{a}(u, w) \wedge \mathrm{p}(u, v) \wedge \mathrm{o}^*(v, w)$ for every $u, v \in V(G)$. We achieve this again by using the sequential cardinality counters described in Subsection 2.1.1. This concludes the description of the formula $F(G, \omega)$, which contains $\mathcal{O}(n^2\omega)$ variables and $\mathcal{O}(n^5)$ clauses.

only if

Exploiting Twins To exploit the twin preprocessing rule for our encoding, we add the following clauses, which ensure twin vertices are ordered according to $<$.

$$\mathrm{o}(u, v) \qquad \text{for } u, v \in V(G),\ N_G(u) = N_G(v) \text{ or } N_G[u] = N_G[v],\ u < v,\ 1 \leq i \leq l.$$

6.6 SAT-Encodings for Pathwidth

In this section we introduce our characterizations and encodings for pathwidth. Namely, we first introduce an encoding for pathwidth based on the well-known vertex separation number and then provide a second encoding based on path decompositions, which can be seen as a special case of the derivation-based encoding for special treewidth.

6.6.1 Partition-Based Encoding for Pathwidth

In this section we provide the partition-based encoding for pathwidth. Note that since a path decomposition has no branches, and therefore the partition on every level consists merely of a single set, the partition-based characterization of pathwidth becomes much simpler than its counterpart for special treewidth. In particular, the encoding is very closely based on the characterization of pathwidth in terms of a path decomposition, which can be equivalently stated as follows. A path decomposition can be seen as a sequence $(P_1, \dots, P_\ell)$ of bags satisfying the following conditions:

(P1) For every $v \in V(G)$ there is a bag P_i with $v \in P_i$.

(P2) For every i with $1 \leq i < \ell$, if $v \in P_i$ and $v \notin P_{i+1}$, then $v \notin P_j$ for every $j > i$. We say that the vertex v has been forgotten at level $i + 1$.

(P3) For every $u, v \in V(G)$ with $\{u, v\} \in E(G)$ and every i with $1 \leq i < \ell$, it holds that if u and v have not yet been forgotten at level i but u is forgotten at level $i + 1$, then u and v are contained in P_i.

In the following we describe the CNF formula $F(G, \omega, \ell)$, which for a graph G and two integers ω and ℓ is satisfied if and only if G has a path decomposition of width at

most ω with at most ℓ bags. Note that since path decompositions are a special case of special tree decompositions, we can bound the maximum number of bags in an optimal path decomposition by $n - \omega$ in accordance with Theorem 6.3. Therefore, the formula $F(G, \omega, n - \omega)$ is satisfied if and only if G has a path decomposition of width at most ω.

We start with describing the variables that form the formula $F(G, \omega, \ell)$. The formula $F(G, \omega, \ell)$ contains *bag variable* $\mathrm{s}(v, i)$ for every $v \in V(G)$ and every i with $1 \leq i \leq \ell$, which is true if P_i contains the vertex v. Next, we have the *forgotten variable* $\mathrm{f}(v, i)$ for every $v \in V(G)$ and every i with $1 \leq i \leq \ell$, which is true if the vertex v has been forgotten at some step $j \leq i$. has the vertex v. vertex v has

Now we are ready to describe the clauses of the formula $F(G, \omega, \ell)$. First to ensure the Property (*P1*), we add the following clauses

$$\mathrm{f}(v, \ell) \qquad \text{for all } v \in V(G).$$

To ensure that no vertex is marked forgotten at (or before) the first bag of the path decomposition, we add the following clauses

$$\neg\mathrm{f}(v, 1) \qquad \text{for all } v \in V(G).$$

To ensures that if a vertex does occur in the bag at level i but not in the bag at level $i + 1$, then it is marked as forgotten, we add the following clauses

$$\neg\mathrm{s}(v, i) \vee \mathrm{s}(v, i + 1) \vee \mathrm{f}(v, i + 1) \qquad \text{for all } v \in V(G) \text{ and } 1 \leq i \leq \ell - 1.$$

To ensures that if a vertex has already been forgotten at level i, then it does not occur in the i-th bag of the path decomposition, we add the following clauses

$$\neg\mathrm{f}(v, i) \vee \neg\mathrm{s}(v, i) \qquad \text{for all } v \in V(G) \text{ and } 1 \leq i \leq \ell - 1.$$

To ensures that if a vertex is forgotten at level i then it remains forgotten at any level $j > i$, we add the following clauses

$$\neg\mathrm{f}(v, i) \vee \mathrm{f}(v, i + 1) \qquad \text{for all } v \in V(G) \text{ and } 1 \leq i \leq \ell - 1.$$

Note that these clauses together with the previous clauses ensure Property (*P2*). To ensure Property (*P3*), we add the following clauses

$$\mathrm{f}(u, i) \vee \mathrm{f}(v, i) \vee \neg\mathrm{f}(u, i + 1) \vee \mathrm{s}(u, i)$$
$$\mathrm{f}(u, i) \vee \mathrm{f}(v, i) \vee \neg\mathrm{f}(u, i + 1) \vee \mathrm{s}(v, i)$$
$$\text{for all } u, v \in V(G),\ \{u, v\} \in E(G) \text{ and } 1 \leq i \leq \ell - 1.$$

Finally, it remains to restrict the maximum size of the set $\mathrm{s}(u, i)$ for any level i to be at most $\omega + 1$, i.e., for every level i with $1 \leq i \leq \ell$, we need to restrict the number of variables $\mathrm{s}(u, i)$ set to true to be at most $\omega + 1$. We achieve this using the sequential cardinality counters described in Subsection 2.1.1. This completes the construction of the formula $F(G, \omega, \ell)$, which including the counter variables and clauses contains $\mathcal{O}(n^2\omega)$ variables and $\mathcal{O}(n^3)$ clauses.

Exploiting Twins To exploit the twin preprocessing rule for our encoding, we add the following clauses, which ensure that the order in which twin vertices are forgotten cannot be arbitrary but corresponds to $<$.

$$\neg \mathrm{f}(u, i) \vee \mathrm{f}(v, i)$$
$$\text{for } u, v \in V(G),\ N_G(u) = N_G(v) \text{ or } N_G[u] = N_G[v],\ u < v,\ 1 \leq i \leq \ell.$$

6.6.2 Ordering-Based Encoding for Pathwidth

Our second encoding for pathwidth is based on the characterization of pathwidth in terms of the vertex separation number, which is defined as follows. Given a graph G, an ordering $\leq_V$ of the vertices of G, and a vertex $v \in V(G)$, we denote by $S_{\leq_V}(v)$ the set of all vertices in G that are smaller or equal to v w.r.t. $\leq_V$. Moreover, for a subset S of the vertices of G, we denote by $\delta(S)$, the set of *guards* of S in G, i.e., the set of all vertices in S that have a neighbor in $V(G) \setminus S$. Then a graph G has *vertex separation number* at most ω if and only if there is an ordering $\leq_V$ of its vertices such that $|\delta(S_{\leq_V}(v))| \leq \omega$ for every $v \in V(G)$. It is well-known that G has vertex separation number at most ω if and only if G has pathwidth at most ω [Kin92].

We will now show how to construct the formula $F(G, \omega)$ which is satisfiable if and only if G has vertex separation number (and hence pathwidth) at most ω. Apart from the variables needed for counting (which we will introduce later), the formula $F(G, \omega)$, has an *order variable* $\mathrm{o}(u, v)$ for every $u, v \in V(G)$ with $u < v$. The variable $\mathrm{o}(u, v)$ will be true if and only if $u < v$ and $u \leq_V v$. The idea behind the variable $\mathrm{o}(u, v)$ is that it can used to model the total ordering $\leq_V$ witnessing the vertex separation number of G by requiring that $u \leq_V v$ for arbitrary $u, v \in V(G)$ if and only if $u = v$ or $u < v$ and $u \leq_V v$ or $u > v$ and $\neg \mathrm{o}(v, u)$. In order to be able to refer to $\leq_V$ in the clauses, we define the "macro" $\mathrm{o}^*(u, v)$ by setting $\mathrm{o}^*(u, v) = \mathit{true}$ if $u = v$, $\mathrm{o}^*(u, v) = \mathrm{o}(u, v)$ if $u < v$ and $\mathrm{o}^*(u, v) = \neg \mathrm{o}(v, u)$ if $u > v$. Moreover, $F(G, \omega)$ has a *guard variable* $\mathrm{c}(v, u)$ for every $u, v \in V(G)$, which is true if $u \leq_V v$ and vertex u has a neighbor vertex w such that $v \leq_V w$, i.e., vertex u contributes to the separation number for vertex v.

We will next provide the clauses for $F(G, \omega)$. Towards ensuring that $\mathrm{o}^*(u, v)$ is a total ordering of $V(G)$, it is sufficient to ensure that the relation described by $\mathrm{o}^*(u, v)$ is transitive, which is achieved by the following clauses:

$$\neg \mathrm{o}^*(u, v) \vee \neg \mathrm{o}^*(v, w) \vee \mathrm{o}^*(u, w)$$
$$\text{for } u, v, w \in V(G) \text{ where } u,\ v, \text{ and } w \text{ are pairwise distinct.}$$

The next clauses provide the semantics for the variables $\mathrm{c}(v,u)$. Namely, $\mathrm{c}(v,u)$ is set to true if $u \leq_V v$ and there is an edge $\{u,w\} \in E(G)$ with $v \leq_V w$.

$$\neg\mathrm{o}^*(u,v) \vee \neg\mathrm{o}^*(v,w) \vee \mathrm{c}(v,u) \qquad \text{for } v \in V(G),\ \{u,w\} \in E(G) \text{ and } v \neq w.$$

It remains to restrict the number of guards of each vertex set $S_{\leq_V}(v)$ given by the ordering $\mathrm{o}^*(u,v)$. Using the variables $\mathrm{c}(v,u)$ this is equivalent to restricting the number of variables $\mathrm{c}(v,u)$ that are true to be at most ω for every $v \in V(G)$. Towards this aim, we again employ the sequential cardinality counters described in Subsection 2.1.1. This completes the construction of the formula $F(G,\omega)$, which including the variables and clauses used for counting has $\mathcal{O}(n^2\omega)$ variables and $\mathcal{O}(n^3)$ clauses.

Exploiting Twins To exploit the twin preprocessing rule for our encoding, we add the following clauses, which ensure twin vertices are ordered according to $<$.

$$\mathrm{o}(u,v) \qquad \text{for } u,v \in V(G),\ N_G(u) = N_G(v) \text{ or } N_G[u] = N_G[v],\ u < v,\ 1 \leq i \leq l.$$

6.7 Experiments

We implemented all four encodings in C++ and they are publically available [LOS17b]. As benchmark instances we used the benchmark set of well-known named graphs from the literature [Wei16], the instances from TreewidthLIB [Bod16], a set of standard graphs containing square grids, complete graphs, and complete bipartite graphs, and a set of random graphs. We will describe the exact set of instances along with the results. We compared the performance of the encodings on well-known named graphs using the SAT-solvers Minisat 2.2 (`m`), Glucose 4.0 (`g`), and MapleSAT (`a`). For the rest of the benchmarks we focused on the most robust of these solvers, i.e., Glucose 4.0.

We use the twin preprocessing for both special treewidth and pathwidth, whereas, we use the degree one preprocessing only for special treewidth. In the following we will refer to the two encodings introduced in Subsections 6.4.2 and 6.6.1 as *partition-based encodings* (P) and to the encodings introduced in Subsections 6.5.2 and 6.6.2 as *ordering-based encodings* (O). We add a T to denote *twin* preprocessing and a D to denote *degree one* preprocessing.Thus the configuration pwPT represents the partition-based encoding with twin preprocessing and the encoding sptwOTD represents the ordering-based encoding for special treewidth with twin and degree one preprocessing.

We also compared our pathwidth encodings with the framework *GDSAT* [BBN+13]. GDSAT is a tool based on generic SAT model, developed to capture a variety of different grid-based graph layout problems. GDSAT can solve 6 different NP-complete problems, including finding optimal pathwidth. GDSAT uses an API for Minisat thus it was not possible to compare its performance using other solvers. However, this tool was not designed to achieve maximal performance for a specific problem rather it is a light framework designed for various classes of graphs which are not too large.

All our experimental results as well as the code for the compilation of our encodings can be found at `https://github.com/nehal73/SATencoding`.

6.7.1 Results

In this section we describe the results of our experiments on the considered benchmarks. We start with describing the results for the named graphs. The goal here is to obtain the exact widths of these well-known named graphs and to compare the performance of different SAT-solvers allowing us to focus on one solver (Glucose) for the remaining experiments. Next, we describe the performance of our encodings on TreewidthLIB, here we focus on comparing the efficiency of our various encodings among themselves. Finally, we test the scalability of our encodings on our set of standard graphs.

Well-Known Named Graphs

Table 6.2 shows our results for the benchmark set of well-known named small to mid-sized graphs from the literature that was previously used in the comparison of encodings for other width measures such as clique-width [HS15] and branchwidth [LOS16a]. For each graph in the benchmark set we run our four encodings as well as, for comparison, the encoding for treewidth based on elimination orderings [SV09], using the three above mentioned SAT-solvers with the aim of computing the exact width of the graph. Namely, starting from width zero ($\omega = 0$) we increased ω by one as long as either the instance became satisfiable (in which case the current ω equals the width of the graph) or the SAT-call reached the timeout of 1000 seconds (in which case the current ω minus 1 is a lower bound for the width of the graph). If we reached a timeout, we further increased ω until the instance could be solved again within the timeout and returned satisfiable, thereby obtaining an upper bound for the width of the graph.

As a typical example, we exhibit in Table 6.3 the running-times for the graph Dodecahedron, which has special treewidth 6. The running-time is given for each ω between 1 and 10. As it can be seen from the table the running-time for $\omega = 5$ is approximately 500 times higher than the running-time for $\omega = 6$. A similar phenomenon was observed for all the instances, which indicates that the main bottleneck for obtaining the optimal width of a graph is the last UNSAT call right before the first SAT call, i.e., the call for the optimal width minus 1. This behavior indicates that our encoding can provide good lower bounds and upper bounds even if identifying the optimal width of a graph is beyond its capabilities.

In three cases (marked with an asterisk in Table 6.2) we obtained the exact width using a longer timeout of 10000 seconds using the partition-based encoding for special treewidth. For each width parameter the obtained width of the graph (or an interval for the width giving the best possible lower bound and upper bound obtained by any encoding) is provided in the ω column of the table. Moreover, for special treewidth and pathwidth, the table contains the two columns (P) and (O), which show the best result obtained by any SAT-solver for the partition-based and ordering-based encodings, respectively.

Table 6.2: Experimental results for the benchmark set of well-known named graphs. A detailed description of the table can be found in Section 6.7.1.

Instance	$\|V\|$	$\|E\|$	Special Treewidth			Pathwidth			Treewidth [SV09]	
			ω	O	P	ω	O	P	ω	O
Petersen	10	15	5	5.03[m]	**0.48**[m]	5	**0.28**[a]	0.35[m]	4	0.15
Goldner-Harary	11	27	4	2.53[m]	**0.27**[m]	4	0.17[g]	0.17[m]	3	0.11
Grötzsch	11	20	5	4.27[m]	**0.43**[m]	5	**0.18**[a]	0.36[m]	5	0.28
Herschel	11	18	4	3.32[m]	**0.34**[m]	4	**0.17**[a]	0.20[m]	3	0.14
Chvátal	12	24	6	11.09[m]	**0.92**[m]	6	**0.44**[g]	0.69[a]	6	0.61
Dürer	12	18	4	7.28[a]	**0.63**[m]	4	**0.13**[m]	0.33[m]	4	0.25
Franklin	12	18	5	12.30[a]	**1.40**[m]	5	**0.30**[m]	0.60[m]	4	0.30
Frucht	12	18	4	7.56[g]	**0.71**[m]	4	**0.21**[g]	0.31[a]	3	0.12
Tietze	12	18	5	11.34[m]	**1.26**[m]	5	**0.27**[m]	0.53[m]	4	0.21
Paley13	13	39	8	22.13[m]	**1.16**[m]	8	**1.02**[a]	1.23[m]	8	2.60
Poussin	15	39	6	61.07[a]	**1.65**[m]	6	**0.39**[m]	0.65[m]	6	0.37
Clebsch	16	40	9	234.28[m]	**13.20**[m]	9	25.76[a]	**17.17**[a]	8	6.30
4x4-grid	16	24	4	97.98[m]	**1.13**[m]	4	**0.22**[a]	0.39[m]	4	0.28
Hoffman	16	32	7	204.73[a]	**20.22**[m]	7	**6.30**[g]	8.21[m]	6	2.39
Shrikhande	16	48	9	234.76[m]	**10.42**[m]	9	11.78[a]	**8.04**[m]	9	131.11
Sousselier	16	27	5	127.87[m]	**3.33**[m]	5	**0.24**[m]	0.62[m]	5	0.31
Errera	17	45	6	153.83[a]	**2.78**[m]	6	**0.40**[m]	0.76[m]	6	0.49
Paley17	17	68	12	504.54[a]	**15.76**[m]	12	106.99[a]	**27.52**[a]	11	35.23
Pappus	18	27	7	912.69[a]	**438.24**[g]	7	**16.47**[g]	54.62[g]	6	160.90
Robertson	19	38	8	1082.73[a]	**130.26**[m]	8	**11.84**[g]	36.02[g]	8	307.21
Desargues	20	30	6	1349.67[m]	**237.57**[g]	6	**0.84**[m]	10.16[m]	6	324.21
Dodecahedron	20	30	6	1564.23[a]	**337.20**[g]	6	**4**[g]	38.52[g]	4-6	4-6
FlowerSnark	20	30	6	1352.67[m]	**201.40**[g]	6	**1.04**[m]	10.99[m]	6	400.06
Folkman	20	40	7	1434.93[a]	**130.20**[m]	7	**2.84**[g]	23.15[m]	6	10.87
Brinkmann	21	42	8	2548.46[m]	**354.62**[m]	8	**14.85**[g]	63.71[g]	8	593.45
Kittell	23	63	7	160.33[g]	**24.70**[m]	7	**1.05**[m]	8.28[m]	7	4.38
McGee	24	36	8*	5-8	5-8	8	**62.47**[a]	524.21[g]	5-7	5-7
Nauru	24	36	8*	5-8	5-8	8	**181.73**[a]	6-8	6	457.92
Holt	27	54	10*	7-10	6-10	10	**386.16**[a]	8-10	7-9	7-9
Watsin	50	75	3-8	M.O.	3-8	7	**76.77**[m]	5-7	4-7	4-7
B10Cage	70	106	2-20	M.O.	2-20	8-16	8-16	6-16	4-17	4-17
Ellingham	78	117	3-9	M.O.	3-9	6	**22.88**[m]	5-7	4-6	4-6

Namely, if the exact width of the graph could be determined, then the column shows the overall running-time in seconds (the sum of all SAT-calls) for the best SAT-solver, whose initial is given as a superscript. Otherwise the table shows the best possible interval that could be obtained within the timeout or "M.O." if every SAT-call resulted in a memory

Table 6.3: Running-time in seconds for each call of ω between 1 and 10 for one of the well known graphs, Dodecahedron, which has special treewidth 6.

	U	U	U	U	U	S	S	S	S	S
ω	1	2	3	4	5	6	7	8	9	10
solving time (s)	0.34	0.57	4.02	19.80	**311.72**	0.74	0.30	0.22	0.22	0.20

out. None of the well-known named graphs have *twin* or *degree one* vertices thus we do not need to compare these techniques here.

It is apparent from Table 6.2, that our encodings could compute the optimal widths for the vast majority of well-known graphs. For the few remaining cases, we were able to compute relatively tight lower bounds and upper bounds.

Finally, we would like to mention a few general observations concerning the performance of the three SAT-solvers. Generally the differences in the performance of the three SAT-solvers were quite minor over all encodings. With respect to the special treewidth encodings, it can be inferred from Table 6.2 that MiniSAT has the best performance for more instances than Glucose or MapleSAT. However, we observed that Glucose was the most robust among the three solvers, since there are instances that could only be solved by Glucose and all instances that could be solved by any of the solvers could also be solved by Glucose. With respect to the pathwidth encodings, the differences between the solvers is less pronounced, each having advantages on about the same number of instances.

TreewidthLIB (TWLIB)

We compared the efficiency of our encodings in combination with the applicable preprocessing rules on the combined instances from the well-known graphs and TreewidthLIB, we will refer to this benchmark set as TWLIB. The benchmark collection TreewidthLIB contains graph instances generated from coloring problem, frequency assignment problem and probabilistic networks. It also contains a big collection of minors of these graphs. Table 6.4 provides the number of solved instances for each of our four encodings and combination of applicable preprocessing procedures. For comparison the table also shows the number of instances solved by GDSAT. Since graphs with more than 60 vertices are hardly within reach of our encodings, we restricted the experiments to instances from TWLIB containing at most 60 vertices; leading to a total of 611 instances. For all these experiments we used a timeout of 2000 seconds for each individual SAT-call and an overall timeout of 6 hours. To provide a more detailed picture of the performance of the various encodings, we also provide the two cactus plots resulting from our experiments on TWLIB in Figures 6.7 and 6.6. The former illustrates the performance of our encodings for pathwidth (as well as GDSAT) and the latter illustrates the performance of our

encodings for special treewidth. As usual, the cactus plots show the running-times of all runs completed within the timeout ordered by running-time (ascending). Hence, the rightmost point of each curve gives the total number of instances solved for the corresponding encoding (configuration).

Using the best encoding, we were able to determine the optimal special treewidth for 442 instances (72%) and the pathwidth for 500 instances (82%); out of 611 instances in total. The partition-based encoding performed significantly better than the ordering-based encoding for both pathwidth and special treewidth; solving an additional 100 instances in the case of special treewidth and an additional 64 instances in the case of pathwidth. The preprocessing procedures only slightly improved the performance of the ordering-based encodings, but had almost no effect on the performance of the partition-based encodings. Finally, GDSAT was only able to solve 234 instances (38%).

Table 6.4: Number of solved instances from TWLIB for all applicable combinations of encodings and preprocessing rules for *special treewidth*, restricted to instances containing at most 60 vertices. A detailed description of the results in this table can be found in Section 6.7.1.

	O				P			
	-	T	D	T+D	-	T	D	T+D
solved instances	321	336	324	340	434	434	436	433

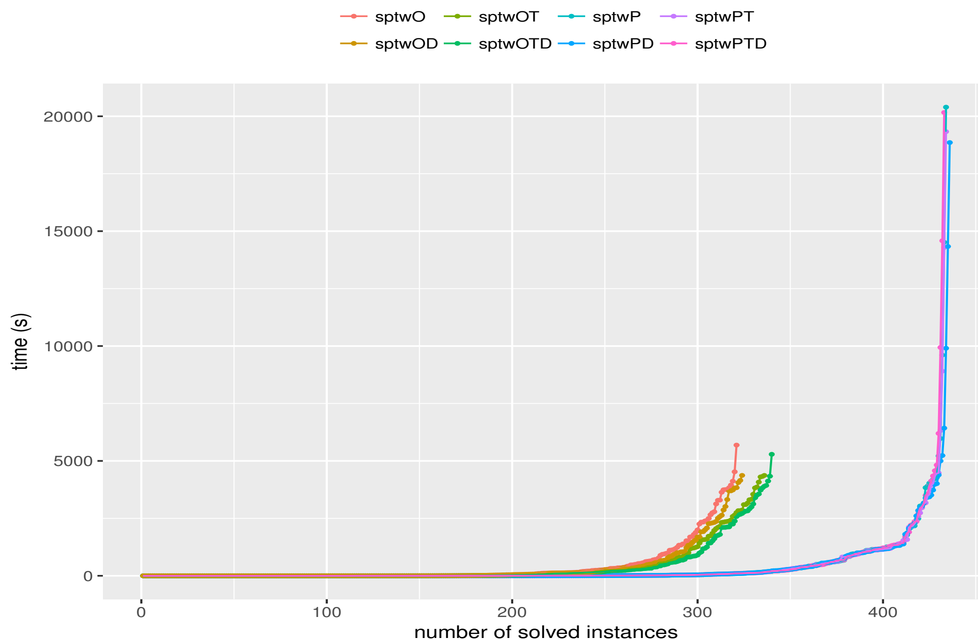
Table 6.5: Number of solved instances from TWLIB for all applicable combinations of encodings and preprocessing rules and GDSAT for *pathwidth*, restricted to instances containing at most 60 vertices. A detailed description of the results in this table can be found in Section 6.7.1.

	O		P		GDSAT
	-	T	-	T	
solved instances	408	422	477	486	234

Standard Graphs

We compared the scalability of our four encodings and GDSAT on a set of standard graphs, i.e., square grids, complete graphs, and complete bipartite graphs (having the same number of vertices on each side). The idea behind using square grids, complete graphs, and complete bipartite graphs is that they represent two types of graphs with high treewidth. Moreover, on all these graphs the pathwidth, special treewidth, and treewidth are equal an well-known, i.e., an $n \times n$ grid has width n, a complete graph on n vertices has width $n-1$, and a complete bipartite graph with n vertices on each side has width n. As before, we used a timeout of 2000 seconds for each individual SAT-call and an overall timeout of 6 hours. For all our encodings, the Table 6.6 shows the largest size

Figure 6.6: Cactus plot representing the number of solved instances with at most 60 vertices from TWLIB by each configuration for special treewidth encoding



(given in terms of the number of vertices) of square grids, complete graphs, and complete bipartite graphs, whose width could be determined exactly within the timeout. Note that we did not consider the preprocessing procedures since they are either not applicable, in the case if grids, or result in a trivial instance, in the case of complete (bipartite) graphs.

As we can see from the Table 6.6 the partition-based encodings scale significantly better than the ordering-based encodings. For instance, the partition-based encoding for special treewidth could find the optimal width for the 6×6 grid, the complete graph with 76 vertices and the complete bipartite graph with 27 vertices on each side, whereas the ordering-based encoding could only determine the optimal widths for the 4×4 grid, the complete graph with 34 vertices and the complete bipartite graph with 16 vertices on each side. A similar trend, which is even more pronounced on the two dense standard graphs, can be observed for pathwidth. In comparison, GDSAT is significantly behind our two encodings for pathwidth.

Random Graphs

Finally, we compared our encodings on a set of random graphs. That is, we generated random graphs on n vertices by starting with the edge-less graph and adding an edge between any pair of vertices with probability p. Specifically, for every $n \in \{20, 30, 40, 50, 60\}$ and every $p \in \{0.1, 0.2, \dots, 0.9\}$, we generated 20 random graphs on n vertices with edge-probability p; resulting in 900 graphs in total. Table 6.7 gives the total number of

Figure 6.7: Cactus plot representing the number of solved instances with at most 60 vertices from TWLIB for each configuration of the pathwidth encodings.

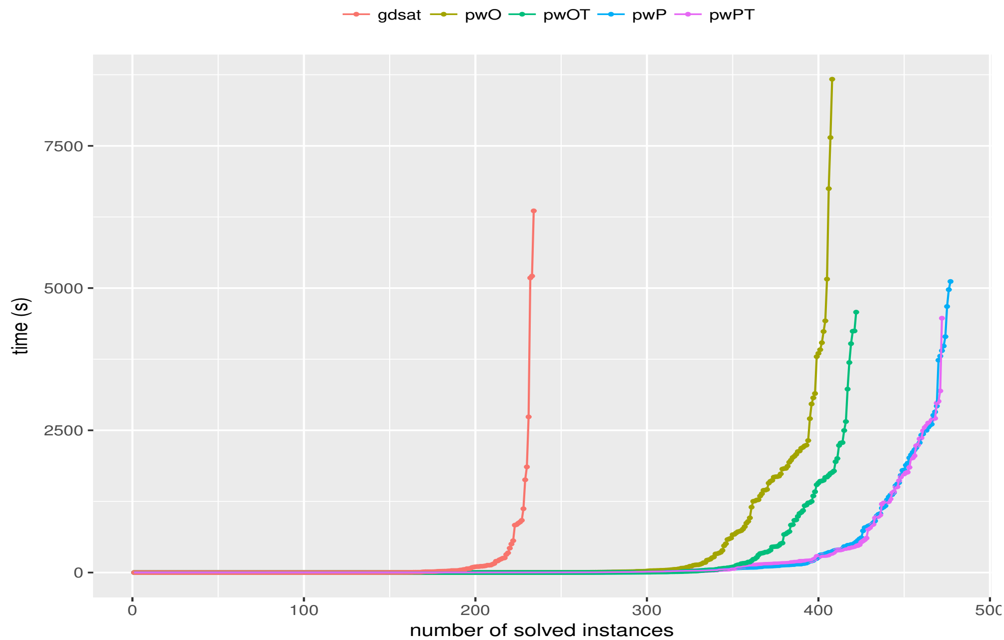


Table 6.6: Experimental results for square-grids, complete graphs, and complete bipartite graphs. For each standard graph class, the table shows the maximum number of vertices for which the width of the graph could still be determined exactly by each of the four encodings as well as GDSAT within the given timeout. See also Section 6.7.1 for an explanation of the results.

Graphs	sptw		pw		
	O	P	O	P	GDSAT
square grids	16	36	64	81	16
complete graphs	34	76	26	123	18
complete bipartite graphs	32	54	40	120	18

instances solved for each of our encodings and GDSAT within the given timeout; we used the same timeouts as for our experiments given in Section 6.7.1. A detailed overview of the results is given in the Tables 6.8–6.11. Each of the four tables provides the percentage of random graphs solved within the timeout by a particular encoding for all combinations of n (number of vertices) and p (edge probability). For instance, Table 6.11 provides these results for the partition-based encoding for pathwidth and it can for example be seen that the encoding solved 10 percent of the random graphs having 40 vertices and an edge probability of 0.2; observe that 10 percent corresponds to 2 out of 20 instances.

Table 6.7: Total number of random graphs solved by our encodings and GDSAT within the timeout. See Section 6.7.1 for more information about the table.

width	sptw		pw		
	O	P	O	P	GDSAT
instances solved	211	482	258	441	33

Note that we do not consider preprocessing for random graphs because we found that the large majority of random graphs did not allow for any of the preprocessing rules to be applied.

Our best encodings could compute the optimal special treewidth of 482 instances (54%) and the optimal pathwidth of 441 instances (49%); out of a total of 900 instances. GDSAT was only able to solve 33 instances. As it was already the case for the TWLIB benchmark set, the partition-based encodings significantly outperform both ordering-based encodings; the difference between the two types of encodings is even more pronounced for random graphs. It becomes apparent from Tables 6.8–6.11, that the better overall performance of the partition-based encoding can partly be attributed to its much better performance on dense graphs.

Table 6.8: Percentage of random graphs solved within the timeout using the ordering-based encoding for *special treewidth* for all combinations of n (number of vertices; represented by the rows) and p (edge probability; represented by the columns). Refer to Section 6.7.1 for more information about the table.

n	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
20	100	100	100	100	100	100	100	100	100
30	100	55	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0	0
50	0	0	0	0	0	0	0	0	0
60	0	0	0	0	0	0	0	0	0

6.7.2 Discussion

In the case of special treewidth, our experiments indicate that the partition-based encoding is far superior to the ordering-based encoding for all of the considered benchmark sets. For instance, Table 6.2 shows that the partition-based encoding beats the ordering-based encoding by one and sometimes even two orders of magnitude on the majority of well-known graphs from the literature. Similarly, Table 6.6 shows that the partition-based encoding is able to handle square grids, complete graphs, and complete bipartite graphs

Table 6.9: Percentage of random graphs solved within the timeout using the partition-based encoding for *special treewidth* for all combinations of n (number of vertices; represented by the rows) and p (edge probability; represented by the columns). Refer to Section 6.7.1 for more information about the table.

n	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
20	100	100	100	100	100	100	100	100	100
30	100	90	85	90	100	100	100	100	100
40	50	0	0	0	0	5	100	100	100
50	0	0	0	0	0	0	0	95	100
60	0	0	0	0	0	0	0	0	95

Table 6.10: Percentage of random graphs solved within the timeout using the ordering-based encoding for *pathwidth* for all combinations of n (number of vertices; represented by the rows) and p (edge probability; represented by the columns). Refer to Section 6.7.1 for more information about the table.

n	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
20	100	100	100	100	100	100	100	100	100
30	100	95	75	15	0	0	5	0	0
40	100	0	0	0	0	0	0	0	0
50	0	0	0	0	0	0	0	0	0
60	0	0	0	0	0	0	0	0	0

Table 6.11: Percentage of random graphs solved within the timeout using the partition-based encoding for *pathwidth* for all combinations of n (number of vertices; represented by the rows) and p (edge probability; represented by the columns). Refer to Section 6.7.1 for more information about the table.

n	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
20	100	100	100	100	100	100	100	100	100
30	100	100	100	95	95	100	100	100	100
40	100	10	0	0	0	0	5	100	100
50	5	0	0	0	0	0	0	0	95
60	0	0	0	0	0	0	0	0	0

that are almost twice as large as the instances solved by the ordering-based encoding. Similar conclusions can be drawn from our experiments for TWLIB; see Figure 6.6 and Table 6.4. The difference between the two encodings is even more pronounced on random graphs, where as shown in Table 6.7, the partition-based encoding solves more than twice

Table 6.12: The number of variables and clauses for our four encodings in terms of the number n of vertices, the number m of edges m, and the width ω

	sptw		pw	
	vars	cls	vars	cls
P	$\mathcal{O}(n^3\omega)$	$\mathcal{O}(n^4 + mn^3)$	$\mathcal{O}(n^2\omega)$	$\mathcal{O}(n^3)$
O	$\mathcal{O}(n^2\omega)$	$\mathcal{O}(n^5)$	$\mathcal{O}(n^2\omega)$	$\mathcal{O}(n^3)$

as many instances as its ordering-based counterpart. It is also interesting to note that the partition-based encoding has an extraordinary advantage on dense random graphs as can be seen when comparing Tables 6.8 and 6.9.

In the case of pathwidth the differences between the two encodings are far less pronounced. In general it can be observed that the ordering-based encoding has a slight advantage on sparse graphs, whereas the partition-based encodings performs far better on dense graphs. In particular, on the mostly sparse well-known graphs from the literature, the ordering-based encoding has a significant advantage over the partition-based encoding; even though the difference between the two encoding is much smaller than for special treewidth. Interestingly, the ranking is reversed on all of the remaining benchmark sets. For instance, the partition-based encoding solves significantly larger standard graphs 6.6; here it is worth noting that the difference becomes very pronounced on the two dense standard graphs (complete (bipartite) graphs), where the partition-based encoding is able to solve instances that are 3 to 5 times larger. The partition-based encoding also has a significant advantage on the TWLIB benchmark with 433 compared to 340 instances solved; see Table 6.5 and also Figure 6.7. Finally, the partition-based encoding solves almost twice as many random graphs (441 compared with 258); see Table 6.7. Comparing Tables 6.10 and 6.11, one can again see that the partition-based encoding performs much better on dense graphs.

In general the preprocessing mainly improved the performance of the ordering-based encodings. For instance, using all preprocessing procedures allowed for the solution of 19 additional instances from TWLIB for the ordering-based encoding of special treewidth. A similar improvement for the number of solved instances can be seen for the ordering-based encoding of pathwidth (422 compared to 408 instances from TWLIB solved). In contrast, preprocessing provides almost no improvement for the two partition-based encodings, where the overhead incurred by preprocessing can even lead to minor losses in overall performance. The overall rather poor performance of our preprocessing procedures is rather surprising to us. Especially, on the TWLIB benchmark set we would have expected better performance gains due to preprocessing. Developing more effective preprocessing procedures is therefore an important topic for future work.

6.8 Chapter Summary

We introduced four novel encodings for width parameters, two for pathwidth and two for special treewidth. For the former, the encodings beat the thus far only available tool for the computation of pathwidth quite significantly. For the latter, the encodings provide the first practical tool for the computation of special treewidth and are based on two novel characterizations that provide new insight into this relatively new parameter. We see our encodings as a first step towards bridging the gap between theory and practice for these important parameters.

Our empirical results emphasize that the performance of SAT-encodings can strongly depend on the underlying characterization. Interestingly, for special treewidth, a partition-based encoding far outperforms an ordering-based encoding. This finding is significant since the latter encoding is closely related to the currently leading encoding for the prominent width parameter treewidth. Unfortunately, all our attempts to develop a similar partition-based encoding for treewidth have thus far remained unfruitful. Moreover, for pathwidth, we obtained two SAT-encodings which both perform well, each of them having an advantage on different classes of instances; which suggests a portfolio-based approach.

Extending the scalability of our algorithms to even larger graphs can be seen as the main challenge for future work. Since our approach supports the computation of lower bounds and upper bounds out-of-the-box, a first step in this direction could be a rigorous experimental analysis of this feature on large graphs. In fact, the behavior of our encodings exemplified in Table 6.3 suggests that one can expect to obtain good lower bounds and upper bounds for instances that are significantly larger. Combined with novel and more generally applicable preprocessing procedures this could potentially already greatly improve the scalability of our approach. Finally, SAT-based local improvement approaches as they have recently been developed for branchwidth and treewidth [LOS16a, FLS17b], provide an interesting venue for future work.

CHAPTER 7

Fractional Hypertree Width

The focus of this chapter is to compute optimal fractional hypertree decompositions. This chapter is based on a paper published at CP 2018 [FHLS18b]. As usual, we start with introducing the same and continue with more technical details about our approach. We provide a detailed experimental analysis before summarizing this chapter.

7.1 Introduction

A prominent research question is the identification of structural restrictions that make the constraint satisfaction problem (CSP) tractable [CC16]. Structural restrictions are concerned only in the way how constraints and variables interact, in contrast to language restrictions that are only concerned with the relations that appear in the constraints. Hybrid restrictions are concerned with both aspects.

In his seminal work, Freuder [Fre82] showed that the CSP is tractable under structural restrictions imposed in terms of bounded treewidth of the constraint graph. The following decades brought a phalanx of results that identified more and more general structural restrictions that still guarantee tractability of the CSP, some prominent notions are spread-cut width [CJG08] and hypertree width [GLS02]. This line of research found its culmination point in the work of Grohe and Marx [GM06, GM14], who introduced the notion of *fractional hypertree width*, which generalized al hitherto known structural restrictions, and which was shown by Marx [Mar13] to be the most general structural restriction that ensures polynomial-time solvability of CSP, subject to a complexity theoretic assumption. Recently Khamis et al.. [KNR15] showed that the Functional Aggregate Query (FAQ) also can be solved in polynomial-time under bounded fraction hypertree width, which lead to having tractability for various problems in databases, logic, matrix operations, probabilistic graphical models under bounded fractional hypertree width of the underlying structure. As their algorithm depends exponentially on the width it is even more crucial to obtain optimal widths for feasible running time.

So far, fractional hypertree width was mostly of theoretical interest, because of the lack of practical algorithms for actually computing the associated decompositions. In fact, computing a decomposition that witnesses the fractional hypertree width of a CSP instance is known to be NP-hard [Mar10]. The known polynomial time approximation algorithm [Mar10] has a cubic error factor that is prohibitive for practical applications, since CSP algorithms that exploit (fractional) hypertree decompositions are exponential (both in time and space) in the width of the decomposition [CJG08, GLS02, GM06, GM14].

Contributions: Here, we propose, implement and test the first practical approach to compute the fractional hypertree width. Our approach is based on an efficient SMT-encoding of the problem, and utilizes preprocessing and symmetry breaking methods. We establish an *ordering-based* characterization of fractional hypertree-width which is similar to the well-known elimination order characterization of treewidth (see, e.g., [Bod98, Dec06], which traces back to the work of Rose [Ros74]. As we already know from the previous chapters, ordering-based characterizations of treewidth have been shown to be well-suited for SAT-encodings of treewidth and related width measures [BBE17, BJ14, LOS17a, SV09], hence it was promising to establish such a characterization also for fractional hypertree width. This indeed turned out to be both feasible as well as effective. In fact, to encode the linear ordering as well as the hyperedges induced by the ordering, we could utilize the very same boolean variables and constraints that have been used for treewidth encodings. However, for treewidth one needs to bound the cardinalities of certain sets of vertices, which in the existing encodings was accomplished by SAT-based cardinality counters or Max-SAT formulations. For fractional hypertree width, however, we need to find certain *real-valued weights* of hyperedges and enforce lower and upper bounds on the sums of weights of certain sets of hyperedges. We found that these constraints can be handled well by the SAT modulo Theory (SMT) framework, in particularly by SMT with linear Arithmetic as implemented in the state-of-the-art SMT solver Z3 [dMB08]. On top of the SMT-encoding we also developed various *preprocessing* and *symmetry breaking* methods.

At this juncture we would like to point out that for CSP instances of bounded fractional hypertree width, one can not only decide satisfiability, but also count the number of satisfying assignments in polynomial time, as observed by Duran and Mengel [DM15]. Hence also from a complexity theoretic point of view it seems adequate to use an SMT solver which operates in the class NPto facilitate the solution of a #P-complete problem.

We implemented our methods creating the prototype tool FraSMT and performed extensive experiments on benchmark instances which contain real-world instances from various application domains. To the best of our knowledge, there have not been any practical algorithms for fractional hypertree width reported in the literature. Thus we took as a reference point the algorithm det-k-decomp for the related (but less general) parameter hypertree width as proposed by Gottlob and Samer [GS09a], which in turn was shown to outperform the algorithm opt-k-decomp proposed earlier by Gottlob et

al. [GLS99].

Our results show that on the considered benchmark instances the new SMT approach clearly outperforms the known algorithm det-k-decomp, even without preprocessing or symmetry breaking. Adding these techniques gives again a significant performance boost.

In summary, our findings are significant as they show

(i) that fractional hypertree width can indeed be computed for a wide range of benchmark instances, and

(ii) that SMT techniques can be successfully applied for structural decomposition, outperforming a known tailor-suited combinatorial decomposition algorithm.

7.2 Preliminaries

A *hypergraph* is a pair $H = (V(H), E(H))$, consisting of a set $V(H)$ of *vertices* and a set $E(H)$ of *hyperedges*, each hyperedge being a subset of $V(H)$.

For a hypergraph $H = (V, E)$ and a vertex $v \in V$, we write $E_H(v)$ for the set of all hyperedges $e \in E$ with $v \in e$. $N_H(v) = (\cup E_H(v)) \setminus \{v\}$ denotes the *neighborhood* of v in H. If $u \in N_H(v)$ we say that u and v are *adjacent* in H. $|N_v(e)|$ is the degree of v in H. To avoid trivial cases, we consider only hypergraphs without vertices of degree 0.

The *hypergraph* $H - v$ *excluding* v is defined by $H = (V \setminus \{v\}, \{e \setminus \{v\} \mid e \in E\})$.

The *primal graph* (or 2-*section*) of a hypergraph $H = (V, E)$ is the graph $P = (V, E_P)$ with $E_P = \{\, \{u, v\} \mid u \neq v$, there is some $e \in E$ such that $\{u, v\} \subseteq e\,\}$. A *hyperclique* is a set $S \subseteq V$ such that S forms a complete graph in the primal graph P, i.e., for any two vertices $u, v \in S$, there exists and hyperedge e in P such that u, v appear together in e.

Consider a hypergraph $H = (V, E)$ and a set $S \subseteq V$. An *edge cover* of S is a set $F \subseteq E$ such that for every $v \in S$ there is some $e \in F$ with $v \in e$. A *fractional edge cover* of S (with respect to H) is a mapping $\gamma : E \to [0, 1]$ such that for every $v \in S$ we have $\sum_{e \in E,\, v \in e} \gamma(e) \geq 1$. The *weight* of γ is defined as $\sum_{e \in E} \gamma(e)$. The *fractional edge cover number* of S with respect to hypergraph H, denoted $\mathsf{fn}_H(S)$, is the minimum weight over all its fractional edge covers.

A *tree decomposition* of a hypergraph $H = (V, E)$ is a pair $\mathcal{T} = (T, \chi)$ where $T = (V(T), E(T))$ is a tree and χ is a mapping that assigns each $t \in V(T)$ a set $\chi(t) \subseteq V$ (called the *bag* at t) such that the following properties hold:

- for each $v \in V$ there is some $t \in V(T)$ with $v \in \chi(t)$ ("v is covered by t"),
- each $e \in E$ there is some $t \in V(T)$ with $e \subseteq \chi(t)$ ("e is covered by t"),
- for any three $t, t', t'' \in V(T)$ where t' lies on a path between t and t'', we have $\chi(t') \subseteq \chi(t) \cap \chi(t'')$ ("bags containing the same vertex are connected").

The width of a tree decomposition $\mathcal{T}$ of H is the size of a largest bag of $\mathcal{T}$ minus 1. The treewidth $\text{tw}(H)$ of H is the smallest width over all its tree decompositions. It is easy to see that $\text{tw}(H) = \text{tw}(P(H))$.

A *generalized hypertree decomposition* of H is a triple $\mathcal{H} = (T, \chi, \lambda)$ where (T, χ) is a tree decomposition of H and λ is a mapping that assigns each $t \in V(T)$ an *edge cover* $\lambda(t)$ of $\chi(t)$. The *width* of $\mathcal{H}$ is the size of a largest edge cover $\lambda(t)$ over all $t \in V(T)$. A hypertree decomposition is a generalized hypertree decomposition that satisfies a certain additional property [GLS02]. The generalized hypertree width $\mathsf{ghtw}(H)$ of H is the smallest width over all generalized hypertree decompositions of H. The *hypertree width* $\mathsf{htw}(H)$ is the smallest width over all hypertree decompositions of H.

A *fractional hypertree decomposition* of H is a triple $\mathcal{F} = (T, \chi, \gamma)$ where (T, χ) is a tree decomposition of H and γ is a mapping that assigns each $t \in V(T)$ a fractional edge cover $\lambda(t)$ of $\chi(t)$ with respect to H. The *width* of $\mathcal{H}$ is the largest weight of the fractional edge covers $\lambda(t)$ over all $t \in V(T)$. The fractional hypertree width $\mathsf{fhtw}(H)$ of H is the smallest width over all fractional hypertree decompositions of H.

Since an edge cover can be seen as the special case of a fractional edge cover, with weights restricted to $\{0, 1\}$, it follows that for every hypergraph, the following holds for every hypergraph H:

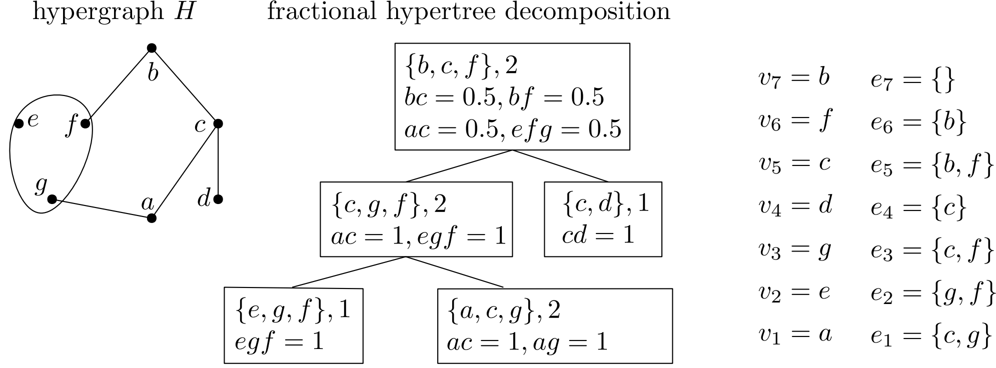
$$\mathsf{fhtw}(H) \leq \mathsf{ghtw}(H) \leq \mathsf{htw}(H) \leq \text{tw}(P(H)).$$

7.3 Ordering-Based Characterization of Fractional Hypertree Width

As we already know from Chapter 1, the first SAT-encoding for treewidth, suggested by Samer and Veith [SV09], has been used for modern SAT-based tree decomposition tools [BBE17, BJ14]. We therefore developed an ordering-based characterization for fractional hypertree width, and based our SMT-encoding on it. Similar characterization was also used by Khamis et al. [KNR15] in their recent work. The remainder of this section is devoted to the definition of this characterization and a proof of correctness of the equivalence between the two characterizations.

Let $H = (V, E)$ be a hypergraph with $n = |V|$ and $L = (v_1, \dots, v_n)$ a linear ordering of the vertices of H. We define the *hypergraph induced by* L as $H_L^n = (V, E^n)$ where E^n is obtained from E by adding hyperedges successively as follows. We let $E^0 = E$, and for $1 \leq i \leq n$ we let $E^i = E^{i-1} \cup \{e_i\}$ where $e_i = \{\, v \in \{v_{i+1}, \dots, v_n\} \mid \text{there is some } e \in E^{i-1} \text{ containing } v \text{ and } v_i \,\}$.

Let $\mathsf{Succ}_L \subseteq \{\, (v_i, v_j) \in V \times V \mid i < j \,\}$ denote the binary relation obtained by the following two steps:



(i) adding all pairs (v_i, v_j) with $i < j$ such that v_i, v_j belong to the same hyperedge in E;

(ii) successively adding all (v_i, v_j) with $i < j$ such that there exists some v_k with $(v_k, v_i), (v_k, v_j) \in \mathsf{Succ}_L$.

We also write $\mathsf{Succ}_L(i) = \{v_i\} \cup \{ v_j \mid (v_i, v_j) \in \mathsf{Succ}_L \}$.Thus we get that $\mathsf{Succ}_L(i) = e_i \cup \{v_i\}$. The following is a direct consequence of the above definitions.

Proposition 7.1. *Let $H = (V, E)$ be a hypergraph, $L = (v_1, \ldots, v_n)$ a linear ordering of V, and $1 \leq i < j \leq n$. Then v_i and v_j are adjacent in H_L^n if and only if $(v_i, v_j) \in \mathsf{Succ}_L$.*

The *fractional hypertree width of H with respect to the linear ordering L*, denoted $\mathsf{fhtw}_L(H)$, is the largest fractional edge cover number with respect to H over all the sets $\mathsf{Succ}_L(i)$, i.e.,

$$\mathsf{fhtw}_L(H) = \max_{i=1}^{n} \mathsf{fn}_H(\mathsf{Succ}_L(i)).$$

We would like to emphasize that in this definition the fractional covers are defined with respect to the original hypergraph H, and not with respect to the induced hypergraph H^n.

Theorem 7.1. *The fractional hypertree width of a hypergraph H equals the smallest fractional width over all its linear orderings, i.e., $\mathsf{fhtw}(H) = \min_L \mathsf{fhtw}_L(H)$.*

We establish the theorem by means of two lemmas below. Before doing so, we introduce some additional terminology.

Let $H = (V, E)$ be a hypergraph and E' a subset of its edges. An *E'-fractional* hypertree decomposition of H is a fractional hypertree decomposition $\mathcal{F} = (T, \chi, \gamma)$ of H where each factional cover $\gamma(t)$ assigns edges $e \in E \setminus E'$ the value 0.

Similarly, the *E'-fractional* hypertree width of H with respect to a linear ordering L, denoted $\mathsf{fhtw}_L(E', H)$, is computed by using only fractional covers that assign edges $e \in E \setminus E'$ the value 0, i.e.,

$$\mathsf{fhtw}_L(E', H) = \max_{i=1}^{n} \mathsf{fn}_{(V,E')}(\mathsf{Succ}_L(i)).$$

Lemma 7.1. *Let $H = (V, E)$ be a hypergraph, $L = (v_1, \ldots, v_n)$ a linear ordering of V, and $E' \subseteq E$. Then H has an E'-fractional hypertree decomposition of width $\leq$ $\mathsf{fhtw}_L(E', H)$.*

Proof. We proceed by induction on n. If $n = 1$ the statement is vacuously true. Now assume $n > 0$ and that the statement holds for all smaller n. Let $w = \mathsf{fhtw}_L(E', H)$. Let $e_1, \ldots, e_n$ and $S_1, \ldots, S_n$ as in the definition of a fractional hypertree width of H with respect to the linear ordering L.

We obtain from H the hypergraph H_2 by restricting H to $H - v_1$, and by adding to it the hyperedge e_1. Furthermore, obtain from E' the set E'_2 by removing v_1 from every edge in E'.

Now $L_2 = (v_2, \ldots, v_n)$ is a linear ordering of H_2, and we observe that its width cannot be larger than the width of L, since otherwise, the sequence of sets $\mathsf{Succ}_{L_2}(i)$ for $1 \leq i \leq n-1$ is exactly the same as the sequence of sets $\mathsf{Succ}_L(i)$ for $2 \leq i \leq n$. Hence $\mathsf{fhtw}_L(E'_2, H_2) \leq \mathsf{fhtw}_L(E', H) \leq w$.

By induction hypothesis, it follows that H_2 has an E'_2-fractional hypertree decomposition $\mathcal{F}_2 = (T_2, \chi_2, \gamma_2)$ of fractional width $\leq w$. By definition of a tree decomposition, there must be a node $t_2 \in V(T_2)$ such that $e_1 \subseteq \chi_2(t_2)$. We define a fractional hypertree decomposition $\mathcal{H} = (T, \chi, \gamma)$ of H as follows.

(i) We obtain T by adding a new node t_1 to T_2 and making it adjacent with t_2.

(ii) We set $\chi(t_1) = \{v_1\} \cup e_1 = S_1$ and $\chi(t) = \chi_2(t)$ for all other tree nodes t.

(iii) We choose for $\gamma(t_1)$ an E'-fractional edge cover of S_1 of smallest weight, which must be $\leq w$ since L was assumed to have weight w, and we set $\gamma(t) = \gamma_2(t)$ for all other tree nodes t.

We observe that (T, χ) satisfies all conditions of a tree decomposition, and conclude that $\mathcal{H}$ is indeed an E'-fractional hypertree decomposition of H of width $\leq w$. □

Lemma 7.2. *Let $H = (V, E)$ be a hypergraph, $E' \subseteq E$ and $\mathcal{H} = (T, \chi, \gamma)$ an E'-fractional hypertree decomposition of H of width w. Then there is a linear ordering $L = (v_1, \ldots, v_n)$ of V such that $\mathsf{fhtw}_L(E', H) \leq w$.*

Proof. As above we proceed by induction on n and observe again that the statement is vacuously true for $n = 1$. Now assume $n > 0$ and that the statement holds all smaller n.

W.l.o.g., we may assume that for each leaf t of T there must be some $v \in \chi(t)$ that does not belong to $\chi(t')$ for any other node $t' \in V(T) \setminus \{t\}$. Namely, if such a $v \in \chi(t)$ does not exist, then the properties of a tree decomposition imply that $\chi(t) \subseteq \chi(t'')$ for the unique neighbor t'' of t in T, and so all vertices and hyperedges covered at node t are also covered at node t'', and t can be omitted.

Based on the above assumption, we conclude that there must be some $v_1 \in V$ which belongs to $\chi(t)$ for a leaf t of T, but v_1 does not belong to $\chi(t')$ for any other node $t' \in V(T) \setminus \{t\}$.

Let $e_1 = \{\, v \in \{v_2, \dots, v_n\} \mid \text{there is some } e \in E \text{ containing } v \text{ and } v_1 \,\}$ and $S_1 = \{v_1\} \cup e_1$ (as in the definition of fractional hypertree width of H with respect to the linear ordering). Since $S_1 \subseteq \chi(t)$, $\gamma(t)$ gives a E'-fractional cover of S (with respect to H) of weight $\leq w$, hence $\mathsf{fn}_{(V,E')}(S_1) \leq w$.

We obtain the hypergraph $H_2 = (V_2, E_2)$ where $V_2 = V \setminus \{v_1\}$ and $E_2 = \{\, e \setminus \{v_1\} \mid e \in E \,\} \cup \{e_1\}$. We also set $E'_2 = \{\, e \setminus \{v_1\} \mid e \in E' \,\}$. It is easy to see that from $\mathcal{H}$ we can obtain an E'_2-fractional hypertree decomposition $\mathcal{H}_2 = (T, \chi_2, \gamma_2)$ of H_2 of width $\leq w$ as follows.

(i) We define $\chi_2(t) = \chi(t) \setminus \{v_1\}$, and $\chi_2(t') = \chi(t')$ for all other tree nodes t.

(ii) For every a hyperedge $e_2 \in E'_2$ we let $\gamma_2(t)[e_2] = \max\{\, \gamma(t)[e_1 \cup \{v_1\}] \mid e_1 \cup \{v_1\} \in E' \,\} \cup \{\, \gamma(t)[e_1] \mid e_1 \in E' \,\}$.

The induction hypothesis applies and hence we can conclude that there exists a linear ordering $L_2 = (v_2, \dots, v_n)$ of $V(H_2)$ such that $\mathsf{fhtw}_{L_2}(E'_2, H_2) \leq w$. We now extend L_2 by adding v_1 at the first position and obtain the ordering $L = (v_1, \dots, v_n)$. We have already observed above that $\mathsf{fn}_{(V,E')}(S_1) \leq w$, hence $\mathsf{fhtw}_L(E', H) \leq w$. □

Theorem 7.1 follows by Lemmas 7.1 and 7.2.

7.4 SMT-encoding for Fractional Hypertree Decomposition

In this section we provide the encoding for elimination ordering for hypergraphs. This encoding is an adaptation of elimination order based encoding for treewidth [SV09]. Given a hypergraph $H = (V, E)$, with $V = \{v_1, \dots, v_n\}$, we produce a formula $F(H, w)$ which is true if and only if the hypergraph H has an elimination ordering L of V such that $\mathsf{fhtw}_L(H) \leq w$.

In fact, Proposition 7.1 shows that the relation Succ_L can be computed in exactly the same way as Samer and Veith compute the graph induced by the ordering. We therefore use the same notation and introduce Boolean *ordering variables* $\mathrm{o}_{i,j}$ for $1 \leq i < j \leq n$ and Boolean *arc variables* $\mathrm{a}_{i,j}$ for $1 \leq i, j \leq n$.

An ordering variable $\mathrm{o}_{i,j}$ is true if and only if $i < j$ and v_i precedes v_j in L. Similar to the treewidth encoding, we use the following macro:

$$\mathrm{o}^*(i,j) = \begin{cases} \mathrm{o}(i,j) & \text{if } i < j \\ \neg \mathrm{o}(j,i) & \text{otherwise.} \end{cases}$$

Consequently, to enforce that L is indeed a linear ordering, we must ensure transitivity, which can be accomplished with the following clauses:

$$[\neg \mathrm{o}^*(i,j) \vee \neg \mathrm{o}^*(j,k) \vee \mathrm{o}^*(i,k)] \qquad \text{for } 1 \leq i,j,k \leq n \text{ and } i,j,k \text{ are distinct.}$$

The arc variables are used to represent the relation Succ_L for the ordering L represented by the ordering variables, where $\mathrm{a}(i,j)$ is true if and only if $(v_i, v_j) \in \mathsf{Succ}_L$, i.e., if $v_j \in \mathsf{Succ}_L(i)$.

A straightforward encoding of the definitions of Succ_L gives rise to the following clauses:

$$\begin{array}{ll}
[\neg \mathrm{o}(i,j) \vee \mathrm{a}(i,j)] \text{ and } [\mathrm{o}(i,j) \vee \mathrm{a}(j,i)] & \text{for } \{v_i, v_j\} \in E(P(H)) \text{ and } i < j. \\
[\neg \mathrm{a}(i,j) \vee \neg \mathrm{a}(i,l) \vee \neg \mathrm{o}(j,l) \vee \mathrm{a}(j,l)] & \text{for } 1 \leq i,j,l \leq n,\ i \neq j,\ i \neq l, \text{ and } j < l. \\
[\neg \mathrm{a}(i,j) \vee \neg \mathrm{a}(i,l) \vee \mathrm{o}(j,l) \vee \mathrm{a}(l,j)] & \text{for } 1 \leq i,j,l \leq n,\ i \neq j,\ i \neq l, \text{ and } j < l. \\
[\neg \mathrm{a}(i,j) \vee \neg \mathrm{a}(i,l) \vee \mathrm{a}(j,l) \vee \mathrm{a}(l,j)] & \text{for } 1 \leq i,j,k \leq n,\ i \neq j,\ i \neq k \text{ and } j < k. \\
[\neg \mathrm{a}(i,i)] & \text{for } 1 \leq i \leq n.
\end{array}$$

The SAT-encoding for treewidth uses cardinality counters, which are encoded by means of sequential counters. But for our encoding we use real valued weight variables representing the fractional covers, instead of the cardinality counters. In fact, this makes the overall SMT-encoding for fractional hypertree width even simpler and more compact than the SAT-encoding for treewidth.

More precisely, we introduce a *weight variable* $w(i,e)$ for each $1 \leq i \leq n$ and $e \in E$, representing the weight of e in a fractional edge cover $\gamma_L(i)$ of the set $\mathsf{Succ}_L(i)$, where L is the ordering represented by the ordering variables.

To ensure that $\gamma_L(i)$ is indeed a fractional edge cover of $\mathsf{Succ}_L(i)$, we add the following two constraints; the first checks that all the vertices in $\mathsf{Succ}_L(i) \setminus \{v_i\}$ are covered by $\gamma_L(i)$, the second checks that v_i is covered by $\gamma_L(i)$.

$$\begin{array}{ll}
[\neg \mathrm{a}(i,j) \vee \sum_{e \in E_H(v_j)} w(i,e) \geq 1] & \text{for all } 1 \leq i \neq j \leq n, \\
[\sum_{e \in E_H(v_i)} w(i,e) \geq 1] & \text{for all } 1 \leq i \leq n.
\end{array}$$

Finally to restrict the fractional hypertree width of H with respect to L, i.e., that $\mathsf{fn}_L(H) \leq w$, $\mathsf{fn}_H(\mathsf{Succ}_L(i)) \leq w$ for all $1 \leq i \leq n$, we add the constraint

$$[\textstyle\sum_{e \in E} w(i,e) \leq w] \qquad \text{for } 1 \leq i \leq n.$$

This completes the construction of the formula $F(H,w)$. The formula $F(H,w)$ has $\mathcal{O}(n(n+m))$ variables where $\mathcal{O}(n^2)$ are Boolean variables and $\mathcal{O}(nm)$ are real variables, and $\mathcal{O}(n^3)$ clauses, where only $\mathcal{O}(n^2)$ are used for restricting the width.

In view of the construction of the formula by Theorem 7.1 and Proposition 7.1 we infer the following result.

Theorem 7.2. *A hypergraph H has fractional hypertree width $\leq w$ if and only if $F(H, w)$ is satisfiable.*

In view of the remark from the end of Section 7.2, we conclude that by replacing the real variables with integer variables yields an encoding for generalized hypertree width.

7.5 Preprocessing

In this section, we formulate several preprocessing methods. Some of them originate in the context of treewidth [BBE17] and are adapted for fractional hypertree width accordingly. In other cases the preprocessing techniques decrease the encoding size significantly, which not only speeds up the solving process, but also extends the scope of our method to larger instances. As fractional hypertree width is not closed under minors, i.e., the fractional hypertree width of a minor of a hypergraph can be larger than the fractional hypertree width of the hypergraph under consideration, the correctness of the preprocessing techniques requires correctness.

We exhaustively apply the following preprocessing rules in their order of occurrence.

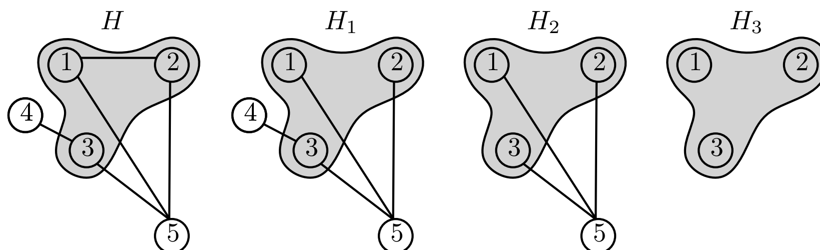


Figure 7.1: A hypergraphs H (left) and hypergraph H_3 obtained after applying all preprocessing rules. H_1 is obtained from H after applying contained hyperedge preprocessing. H_2 is obtained from H_1 after applying deletion of vertices of degree 1 preprocessing. Finally, H_3 is obtained from H_2 after applying simplicial vertex preprocessing.

7.5.1 Contained Hyperedges

A basic preprocessing technique is that a hyperedge which is a subset of another hyperedge can be removed safely, i.e., without changing the fractional hypertree width. The following preposition formalizes this preprocessing technique.

Proposition 7.2. *Let $H = (V, E)$ be a hypergraph, $e, f \in E$ be hyperedges such that $e \subsetneq f$, then*

$$\mathsf{fhtw}(H) = \mathsf{fhtw}((V, E \setminus \{e\})).$$

Proof. Consider a fractional hypertree decomposition $\mathcal{F} = (T, \chi, \lambda)$ of H that assigns $\lambda(e) > 0$. We construct an alternative fractional hypertree decomposition $\mathcal{F}' =$

(T, χ, λ') where the fractional covers are set as following:

$$\lambda'(e') = \begin{cases} \lambda(e') & \text{for } e' \in E \setminus \{e, f\} \\ \lambda(f) + \lambda(e) & \text{for } e' = f \end{cases}$$

Observe that λ' is a valid fractional edge cover for H, i.e., $\mathcal{F}'$ is a valid fractional hypertree decomposition of H. Thus, we obtain that $\mathsf{fhtw}(H) = \mathsf{fhtw}(V, E \setminus \{e\})$. □

In Figure 7.1 the edge $\{1, 2\}$ is a subset of edge $\{1, 2, 3\}$. According to the contained edge preprocessing rule we can safely remove edge $\{1, 2\}$ from H.

7.5.2 Biconnected Components

A hypergraph H is *connected* if for any two vertices $u, v \in V$ there exists vertices $v_1, \ldots, v_k \in V$ such that $u = v_1$, $v = v_k$ and v_i and v_{i+1} are adjacent in H for $1 \leq i \leq k-1$. H is *biconnected* if $H - v$ is connected for every $v \in V$. A *biconnected component* of H is a maximal biconnected hypergraph $H' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. Observe that two biconnected components of H can have at most one vertex in common.

During preprocessing, we can split any hypergraph into biconnected components and compute the fractional hypertree width of each component separately. The following preposition formalizes the biconnected component preprocessing.

Proposition 7.3. *Let H be a hypergraph and $H_1, \ldots, H_\ell$ its biconnected components. Then*

$$\mathsf{fhtw}(H) = \max_{i=1}^{k} \mathsf{fhtw}(H_i).$$

Proof. Observe that $\mathsf{fhtw}(H) \geq \mathsf{fhtw}(H_i)$, hence, $\mathsf{fhtw}(H) \geq \max_{i=1}^{k} \mathsf{fhtw}(H_i)$.

To see the reverse direction, let $\mathcal{H}_i = (T_i, \chi_i, \lambda_i)$ be a fractional hypertree decomposition of H_i of smallest width, $1 \leq i \leq k$. W.lo.g, we may assume that all the trees T_i are mutually disjoint. Consider the graph $\mathcal{G}$ which has all the H_i's as vertices, where H_i and H_j are connected by an edge if and only if $V_i \cap V_j \neq \emptyset$. As observed above, two biconnected components may share at most one vertex, therefore, if $V_i \cap V_j \neq \emptyset$ then $|V_i \cap V_j| = 1$. $\mathcal{G}$ cannot contain a cycle, since otherwise, the union of the biconnected components forming the cycle would be biconnected, contradicting the maximality property of a biconnected component. Hence we can obtain a tree $T = (V(T), E(T))$ where $V(T) = \bigcup_{i=1}^{k} V(T_i)$ and $V(E) \subseteq \bigcup_{i=1}^{k} E(T_i)$, by the following:

(i) adding an edge between any two trees T_i and T_j whenever $|V_i \cap V_j| = 1$, by choosing a node $t_i \in V(T_i)$ and a node $t_j \in V(T_j)$ with $\chi_i(t_i) \cap \chi_j(t_j) \neq \emptyset$ and adding the edge $\{t_i, t_j\}$;

(ii) adding further edges arbitrarily until T is connected.

We now obtain a fractional hypertree decomposition $\mathcal{H} = (T, \bigcup_{i=1}^{k} \chi_i, \bigcup_{i=1}^{k} \lambda_i)$, whose width is bounded by $\max_{i=1}^{k} \mathsf{fhtw}(H_i)$. Observe that $\mathcal{H}$ is also a fractional hypertree decomposition of H therefore $\mathsf{fhtw}(H) \leq \max_{i=1}^{k} \mathsf{fhtw}(H_i)$. Thus, we obtain that $\mathsf{fhtw}(H) = \max_{i=1}^{k} \mathsf{fhtw}(H_i)$. □

7.5.3 Deletion of Vertices of Degree 1

During preprocessing, we can safely remove a vertex of degree 1, i.e., a vertex occurring in only one hyperedge. The following preposition formalizes the deletion of vertices of degree 1 preprocessing.

Proposition 7.4. *Let $H = (V, E)$ be a hypergraph and $v \in V$ be a vertex of degree one, i.e., $|E_H(v)| = 1$. Then, if $\mathsf{fhtw}(H - v) \geq 1$, one can safely remove v such that $\mathsf{fhtw}(H) = \mathsf{fhtw}(H - v)$.*

Proof. We know that $\mathsf{fhtw}(H) \geq \mathsf{fhtw}(H - v)$. For showing $\mathsf{fhtw}(H) \leq \mathsf{fhtw}(H - v)$, we take a fractional hypertree decomposition $\mathcal{F} = (T, \chi, \lambda)$ of hypergraph $H - v$ and modify $\mathcal{F}$ to obtain a fractional hypertree decomposition $\mathcal{F}' = (T', \chi', \lambda')$ of H. In particular, we know that there exists at least one node t in T with $\chi(t) = e \setminus \{v\}$, where $e \in E$ such that $v \in e$. Then, we construct $\mathcal{F}'$ by taking $\mathcal{F}$, adding a fresh node t' as a child node of t to T', and assigning $\chi(t') = e$ and $\lambda'(t') = 1$. Since $\mathsf{fhtw}(H) \geq 1$, we get that $\mathsf{fhtw}(H) \leq \mathsf{fhtw}(H - v)$, this concludes the proof. □

In Figure 7.1 vertex 4 is a degree 1 vertex. According to this preprocessing rule deleting vertex 4 from H is safe.

7.5.4 Simplicial Vertices

Let $H = (V, E)$ be a hypergraph. A vertex $v \in V$ is a *simplicial vertex* of H if the neighborhood of v in H forms a clique in the primal graph of H.

During the preprocessing, we can remove a simplicial vertex v as long we maintain $\mathsf{fn}_H(N_H[v] \cup \{v\})$ as a lower bound for the fractional hypertree width. The following preposition formalizes the simplicial vertex preprocessing.

Proposition 7.5. *Let $H = (V, E)$ be a hypergraph and v a simplicial vertex of H. Then,*

$$\mathsf{fhtw}(H) = \max(\mathsf{fhtw}(H - v), \mathsf{fn}_H(N_H[v] \cup \{v\})).$$

Proof. In order to prove this proposition, we proceed similarly to the proof of Proposition 7.4, where one modifies a fractional hypertree decomposition $\mathcal{F}$ for $H - v$ in order to obtain one for H. However, in this case, the fresh decomposition node t' contains $N_H[v] \cup \{v\}$ in its bag. □

In Figure 7.1 vertex 5 is a simplicial vertex. Therefore according to simplicial preprocessing rule, deleting the vertex 5 from H is safe.

7.6 Symmetry Breaking and Lower Bounds with Cliques

In this section we present the utilization of cliques in the primal graph for the following two purposes:

(i) as symmetry breaking technique, for fixing a part of the linear ordering in order to reduce the search space;

(ii) as lower bound technique, to find good lower bounds on fractional hypertree width by finding hypercliques with certain properties.

To start with, we can choose any clique (i.e., a complete subgraph) in the primal graph and put the vertices of the clique at the end of the ordering. This can be seen as a symmetry breaking method that decreases the search space. In particular, it helps to speed up the optimality check (i.e., the $F(H, w)$ call when $w = \mathsf{fhtw}(H) - 1$), as here the full search space needs to be explored. Such techniques have previously been used for a SAT-encoding of treewidth [BBE17].

To formalize, consider a hypergraph $H = (V, E)$ and $S \subseteq V$ a *hyperclique*. The next proposition ensures that we can indeed force a hyperclique to be the last in the ordering without effecting the fractional hypertree width.

Proposition 7.6. *Let $H = (V, E)$ and be a hypergraph and $S = \{v_1, \ldots, v_\ell\}$ a hyperclique in H. Then, there is an ordering $L = (\ldots, v_1, \ldots, v_\ell)$ in which the vertices of S appear at the end, such that $\mathsf{fhtw}_L(H) = \mathsf{fhtw}(H)$.*

Proof. Let $\mathcal{F} = (T, \chi, \lambda)$ be a fractional hypertree decomposition of H of width $\mathsf{fhtw}(H)$. Observe that by the properties of a tree decomposition, there exists a node t in T with $S \subseteq \chi(t)$ (see, e.g., [BM93]), i.e., all the vertices of S are present altogether in some bag of T. We consider T to be rooted in t and construct a linear ordering L according to the proof of Lemma 7.2. Since we always pick vertices belonging to the bags corresponding the leaves of T, we are left with t as the last tree node, and hence the vertices from $\chi(t)$ will be picked last. As a result, we obtain an ordering L, where vertices V' appear at the end and $\mathsf{fhtw}_L(H) = \mathsf{fhtw}(H)$. □

Hypercliques can also be used to obtain a lower bound on the fractional hypertree width. In case of treewidth, a graph containing a clique of size k has treewidth at least $k - 1$ (see, e.g., [BM93]). However, in the context of hypergraphs and fractional hypertree width, we need to take into account the fractional edge cover number of the clique. Consider for instance a hypergraph $H = (V, \{V\})$. It is easy to see that $\mathsf{fhtw}(H) = 1$, although V forms a hyperclique. However, we still can show the following:

Proposition 7.7. *Let $H = (V, E)$ be a hypergraph and S a hyperclique of H. Then,* $\mathsf{fhtw}(H) \geq \mathsf{fn}_H(S)$.

Proof. Assume any fractional hypertree decomposition $\mathcal{F} = (T, \chi, \lambda)$ of H. Since S is a hyperclique, there exists a node t in $\mathcal{F}$ whose bag contains S, i.e., $\chi(t) \supseteq S$ (see, e.g., [BM93]). Then, by definition of fractional hypertree decompositions, every vertex of S is covered in t. As a result, the weight $\lambda(t)$ is at least $\mathsf{fn}_H(S)$, and $\mathsf{fhtw}(H) \geq \mathsf{fn}_H(S)$. $\square$

In sight of the above proposition, we want to find large hypercliques with potentially large fractional edge cover number. To this end, we propose the following notion.

A hyperclique S of a hypergraph $H = (V, E)$ is *k-hyperclique* if no hyperedge of H intersects with S in more than k vertices. Intuitively, small values of k prevent large hyperedges, whereas bigger values provides us with flexibility, resulting in potentially larger cliques. In Figure 7.1 the vertices $\{1, 2, 3, 5\}$ form a 3-hyperclique.

As already discussed at the beginning of this section, we rely on the cliques for symmetry breaking and obtaining lower bounds. As a large hyperedge in the input graph is a large clique in the primal graph with fractional hypertree width 1. Therefore, a large clique in primal graph might not provide us with good lower bounds. To overcome the issue of finding a good clique with high hypertree width and large number of vertices, we compute a maximum cardinality k-hyperclique, for some fixed k.

In the following, we discuss how to search for a k-hyperclique (S) of size at least ℓ for given hypergraph $H = (V, E)$ by means of a SAT-encoding. Here, we assume k to be a small constant. For each vertex v we introduce a boolean variable x_v, which is true if v belongs to the k-hyperclique S. We add the following constraints to find such a k-hyperclique S:

$[\neg x_{v_1} \vee \neg x_{v_2}]$	for $v_1, v_2 \in V$ and $v_2 \notin N[v_1]$.
$[\neg x_{v_1} \vee \cdots \vee \neg x_{v_k}]$	for $v_1, \ldots, v_k \in V$, $e \in E$ and $v_1, \ldots, v_k \in e$.
$[\sum_{v \in V} x_v \geq \ell]$	cardinality counters to enforce $\lvert S \rvert \geq \ell$.

7.7 Experimental Work

We performed a series of experiments on various publicly available benchmark sets, in order to obtain the fractional hypertree width of these instances, to evaluate whether our SMT-based approach fits well to obtain exact values on the width, and to investigate how well our approach scales. The source code of our SMT-based decomposer, benchmarks, and detailed results are publicly available via an anonymous dropbox link[1].

7.7.1 Implementation.

We implemented our encoding into our prototypical decomposer FraSMT. We used Python 2.7.14 [vR95] based on an Anaconda[2] distribution, which includes dependency han-

[1] See: `https://www.dropbox.com/sh/lcxscc4wxhjr46v/AAD4DO18ozwXp0nTp2D3gGCNa?dl=0`:{instances,results,src}

[2] `https://conda.io/docs/user-guide/install/download.html`

dling for binaries packages. We used the graph library networkX 2.1 [HSS08], the answer-set programming solver clingo version 5.2.2 (gringo 5.2.2 and clasp 3.3.3) [GKKS17], and the SMT solver Z3 4.6.2 [dMB08]. Our implementation consists of two separate tools: a validator and the actual decomposer.

Validator: The first part is a reusable validator that allows to validate computed fractional hypertree decompositions and related decompositions such as tree decompositions and hypertree decompositions. The validator takes as input an extended format of the format used for the treewidth track of the Parameterized Algorithms and Computational Experiments Challenge (PACE) [DKTW17]. Since the graph library networkX does not support hypergraphs, we implemented hypergraph classes and classes that allow for a primal graph view on such a hypergraph. Both classes implement a networkX-like hypergraph API. Practically, we represent the maximum width by a real value. Since the SMT solver Z3 does not allow for reals of arbitrary precision and we may have a precision loss due to the representation of the real numbers [Com08], we check for width $w + \epsilon$ for some small $\epsilon \geq 0$. By default we set ϵ to 0.001.

Decomposer and its Configurations: The second and main part is our decomposer FraSMT, which implements the preprocessing techniques, the SMT-encoding as discussed in the previous sections, starting the SMT solver, as well as reconstructing a decomposition from the solver assignments and outputting a decomposition (if possible). Our decomposer *always* reduces contained hyperedges and splits a hypergraph into biconnected components and computes the width of each component separately. We optionally run finding and deleting degree 1 vertices as well as simplicial vertices. Later configurations including `P` use this preprocessing while configurations containing `p` disable this preprocessing technique. Further, our decomposer computes as a preprocessing step large cliques for the following two purposes

(i) symmetry breaking for fixing some part of the linear ordering

(ii) for obtaining better lower bounds

For computing large cliques we employ an answer-set programming (ASP) solver, which allows for a trivial encoding of a largest clique[3]. Despite the easy encoding, the ASP solver enables us to use (implicit) incremental solving and a technique called unsatisfiable core shrinking [AD16], which allows us to obtain a large (potentially not largest) clique at any time during the optimization (as long as at least one clique has been computed). We then use a large clique to apply symmetry breaking in our encoding as described in Proposition 7.6 and we use cliques to obtain additional lower bounds for the encoding. Moreover, we take the maximum width over the previously computed components and feed this value into the next computation. In that way we might obtain unsatisfiability and cannot output a decomposition, however, we cut the search space for the SMT solver

[3]see e.g., `https://en.wikipedia.org/wiki/Answer_set_programming#Large_clique`

as the solver does not necessarily need to find an exact solution in order to avoid an easy-hard-easy behavior. In the following configurations we use symmetry breaking as well as employ lower bounds while configurations containing `s` disable this technique. Finally, we implemented the encoding via a direct Python interface to the solver provided using additional feature of Z3.

Other Solvers. In order to obtain results for hypertree width of our instances, we used a backtracking-based implementation det-k-decomp by Gottlob and Samer [Got18, GS09b]. Since this implementation can only check for hypertree width of size at most k of an instance, we added a simple progression step on top, which for every iteration reduces the result of det-k-decomp by 1 to check optimality.

7.7.2 Benchmark Instances

We considered a selection of 2191 instances, which contain hypergraphs that originate in CQs and CSPs instances from various sources. The hypergraphs contain up to 2993 vertices and 2958 hyperedges. The first set `DaimlerChrysler` consists of 15 instances, the second set `Grid2D` consists of 12 instances, and the third set `ISCAS'89` consists of 24 instances on circuits [GS09b]. Moreover, the benchmarks contain 35 instances in the set `MaxSAT` [BLJS17] and two sets (`csp_application` and `csp_random`) of instances from the well known XCSP benchmarks [ABLP16] with less than 100 constraints such that all constraints are extensional. The set `csp_application` contains 1090 instances and the set `csp_random` contains 863 instances. Further, the set `csp_other` contains 82 instances, which have been collected for works on hypertree decompositions[4]. The set `CQ` consists of 156 instances from various conjunctive queries [AGCM15, BKM+17, GMPS14, GPH05, LGM+15, Tra14]. All instances have been collected by Wolfgang Fischl [Fis18]. We gratefully acknowledge him for providing us with this large collection of benchmark instances.

7.7.3 Benchmark Setting

Hardware: Our results were gathered on Ubuntu 16.04 LTS Linux machines kernel 4.13.0-3 on GCC 5.4.1, both post-Spectre and post-Meltdown kernels[5]. We ran on a cluster of 16 nodes. Each node is equipped with two Intel Xeon E5-2640v4 CPUs consisting of 10 physical cores each at 2.4 GHz clock speed and 160 GB RAM. Hyper threading was disabled.

Setup and Limits: In order to draw conclusions about the efficiency of FraSMT, we mainly inspected the wall clock time. We set a timeout of 7200 seconds and limited available RAM to 8 GB per instance. Resource limits where enforced by *runsolver* [Rou11].

[4] `https://www.dbai.tuwien.ac.at/proj/hypertree/benchmarks.zip`
[5] See: `spectreattack.com`

Due to hardware resource limitations we conducted only 1 run per instance and configuration. However, we benchmarked a few instances with multiple runs and observed no significant difference.

7.7.4 Results

We used a tool to gather data and control the benchmark generation, evaluation, and cluster setting [KSR+17]. We publicly provide all experimental data[6], including raw data such as all command line flags used, system sampling (RAM/sysload), standard output and standard error during the run.

config	N	$t[s]$ median	avg	std
FraSMT (C6P)	**1451**	1189	3124	3299
FraSMT (C4P)	1435	1187	3192	3326
FraSMT (C4p)	1283	1760	3461	3432
FraSMT (c0p)	1107	7200	4019	3398
det-k-decomp	838	7200	4672	3357

Table 7.1: Overview on the number N of instances for which the respective decomposer configuration outputted the exact (fractional) hypertree width of the instance. Configuration: c/C represents disabled or enabled symmetry breaking and lower bound techniques, respectively. p/P represents disabled or enabled preprocessing techniques. 0,4,6 represents a k-hyperclique for $k = 0$, 4, or 6. t median (avg, std) represents the median (average, standard deviation) of the runtime in seconds of the decomposer over all instances of our benchmark instances, including the timeouts.

fhtw	1	$(1,2]$	$(2,3]$	$(3,4]$	$(4,5]$	$(5,6]$	$(6,7]$	$(7,8]$	$(8,9]$
N	145	123	198	255	308	273	65	81	1

Table 7.2: Distribution gives the intervals of the fractional hypertree width w of the instances for which we successfully obtained an exact result using the best solver configuration. (The interval $(i, j]$ includes all widths w such that $i < w \leq j$.)

Solved Instances/Runtime: Table 7.1 provides basic statistics on the benchmarks. The table contains the tested configurations of our decomposer and the number of solved instances for which we obtained the (fractional) hypertree width and presents total (average, minimum) runtime of the decomposer. We include timeouts of 7200 seconds into the average and median. Figure 7.2 illustrates runtime results for the tested decomposer

[6] See: `tinyurl.com/fhtdsmt`:results

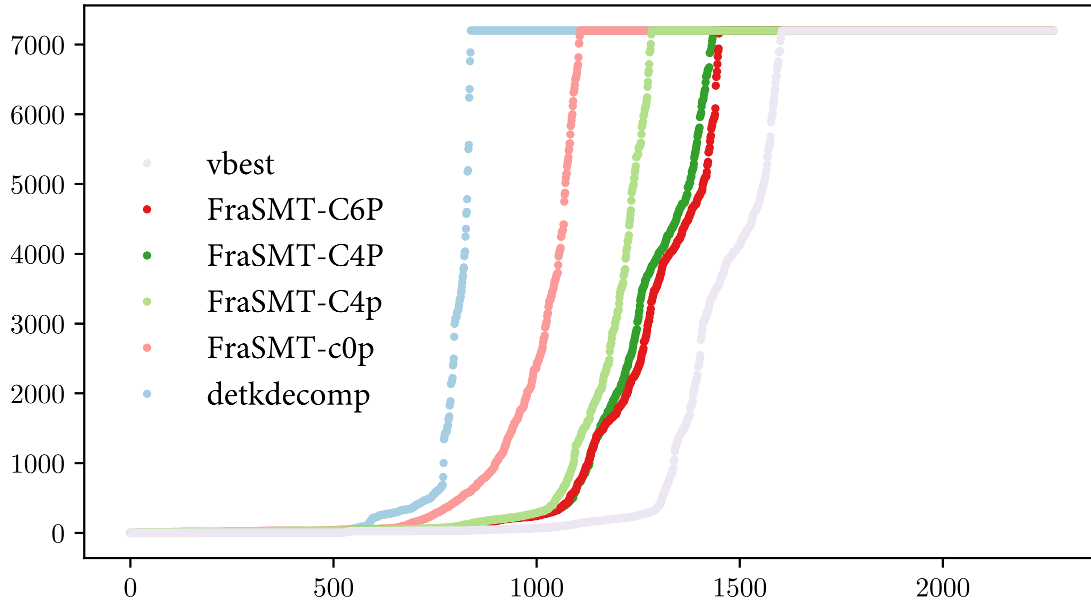


Figure 7.2: Runtime in seconds on the considered benchmark instance. `k-grid`). vbest refers to the virtual best solver. The x-axis labels consecutive integers that identify instances. The instances are ordered by running time, individually for each solver.

configurations as cactus plot. We solved instances that have up to 1453 vertices, up to 891 hyperedges, and up to hyperedges of size 16. The best configuration, namely FraSMT (C6P), was capable of decomposing 1451 out of the total number of 2191 instances. Using k-hypercliques of $k = 4$ instead of $k = 6$ for symmetry breaking solved 1435 instances. Without preprocessing (p), FraSMT was able to solve 1283 instances. Without preprocessing and symmetry breaking, FraSMT could obtain 1107 fractional hypertree decompositions of exact width. Solver det-k-decomp was able to solve 838 instances, although both underlying methods are exact and det-k-decomp computes the less general parameter hypertree width in the same time. We further observe that by analyzing the virtual best solver (vbest), there are some instances a single best configuration cannot solve but can be solved by different configurations, which, however, might be due to the relatively small difference also result from the cluster setup and usage.

(Fractional) Hypertree Width: We computed the fractional hypertree width for our benchmarks using FraSMT and the hypertree width using det-k-decomp. The sets contain a few identical instances that occur in multiple sets. Even though we provide here only an overview on all instances, we decided to keep the duplicate instances to analyze the benchmark sets as provided from the original source for easier comparability. We provide detailed statistics online[6]. Using det-k-decomp we obtained the hypertree width for 838 instances. Table 7.2 provides the distribution of the number of instances and their respective fractional hypertree width.

Considering all sets, 33% of the instances have fractional hypertree width below 4, 60% of the instances have fractional hypertree width below 6. 66% of the instances have fractional hypertree width Overall we were able to obtain the exact width for 66% of the instances.

Fractional Hypertree Width vs. Hypertree Width: When considering the obtained fractional hypertree width and hypertree width for these instances in our benchmark set that have been solved by both methods, the best FraSMT configuration and det-k-decomp, we observed a difference between fhtw and *htw* on 221 instances. The maximum difference was 2, and among these 221 instances the median difference was 0.6. However, since det-k-decomp could decompose significantly fewer instances and by construction works better on instances on small htw, we expect the difference between fhtw and htw to be significantly higher on the remaining instances.

7.8 Chapter Summary

Our SMT-based encoding for fractional hypertree width, preprocessing, and its implementation enable the computation of the most general structural restriction for CSP that still guarantees tractability. In this way, fractional hypertree width is not just a theoretical measure for CSP. Our results show that a majority of our considered benchmark instances have low fractional hypertree width (below 10). However, we are unable to compute the exact width for about 33% of the instances. Consequently, we think that upper bound computations either using heuristics for hypertree width or modifying our encoding to obtain only upper bounds can be of interest for future investigations.

Interestingly, we obtained the exact fractional hypertree width for more instances using our decomposer FraSMT than the exact hypertree width using det-k-decomp, although our decomposer determined the more general parameter. An important factor is the extensive preprocessing and symmetry breaking, which is not present in det-k-decomp, as preprocessing and symmetry breaking resulted in 16% more solved instances for our decomposition technique. However, even without preprocessing or symmetry breaking our method solved more instances than det-k-decomp.

Since our results are limited to relatively small hypergraphs (up to about 1400 vertices, 900 hyperedges, and hyperedges of small size), heuristics or combinations of heuristics and exact methods might be interesting for practical purposes. Our techniques can be very helpful to evaluate the accuracy of heuristics. Efficient and precise heuristics would enable us to obtain a broad picture about available instances in CSP which might lead to a usage of fractional hypertree decompositions for solving actual CSP instances, in particular, for problems such as model counting in CSP.

The focus here, was the exact computation of fhtw. We would like to point out that with our approach one can also compute just upper bounds on the fhtw by not verifying and solving to optimality. We have reasons to believe that this will scale to significantly larger

instances, since a similar behavior has been observed in related work [FLS17b, LOS16a]. We are interested to address this potential of our method systematically in future work.

We think that our decomposer might as well be useful for theoretical purposes to understand better the structural measure fractional hypertree width.

CHAPTER 8

Conclusion and Future Work

We have presented rigorous conclusion and future work emerging from each chapter in the dedicated chapters. In this chapter, we summarize some of the major topics that emerge from our research.

To conclude, in this thesis we presented various SAT-based techniques for finding optimal and close to optimal decompositions. We showed the importance of developing customized characterizations in order to construct efficient encoding via our experimental analysis. One of the most important contributions of this thesis is that of using our local improvement techniques to overcome the restriction that SAT-encodings can only solve small problem instances. In fact, from our experiments we show that SAT-based techniques can be very effective for finding upper and lower bounds. We re-emphasized the importance of preprocessing in order to solve any given problem instance.

Our major focus was to develop effective SAT-encodings for decomposition parameters, therefore we did not rigorously investigate the effect of various SAT technologies that usually aid the SAT solving, such as the cardinality counters or alternative encodings for transitive clauses. As a future work it would be interesting to observe how these techniques affect the performance of individual SAT-encodings. Another evident direction of future work is to compare the black box SAT solving with incremental solving and MAXSAT solving. It would be of general interested to identify which encodings techniques can be beneficial for one of these approaches. This takes us to our next open topic for future work, which is portfolio approaches. The flexibility of SAT approaches allows one to identify and use various different techniques. Using some heuristics on the input instance, one can identify which encoding/encoding parameters, SAT-solver, SAT technique, etc., are most suitable for the current input instance.

Another promising future research emerging from this thesis is the automated parameter tuning for all of our approaches. The local improvement techniques have a large scope of improvement if they have well tuned parameters. Similarly, one can maximize the

impacts of various cardinality counters, symmetry breaking and preprocessing using parameter tuning. Local improvement techniques for special treewidth, pathwidth, and fractional hypertree width could also prove to be crucial for development of practical approached based on these techniques. Specially, techniques that can find sub-optimal decompositions such that the local improvement can be done efficiently can vastly improve the performance of our local improvement approach. Developing SAT-encodings that can construct local decompositions that aid in local improvement, could play a crucial role. Along with the SAT-encodings, the local decomposition plays a crucial role in the local improvement. It would be worth investigating how to extract these in order to maximize the impact of local improvement.

Naturally, developing SAT-based techniques for other width parameters, such as boolean-width [ABR$^+$10], CV-width [Dar03], etc., can provide more insight into these parameters. Finally, we would like to emphasize that in order to develop SAT-based techniques, one has to look at the given problem in an alternative way, which requires a good understanding of both (the problem and SAT technologies), and on the other hand, these techniques can allow one to have better insight in the problem itself.

List of Figures

2.1 A simple undirected graph G with 5 vertices and 8 edges. The max degree of G is 5. The center of G is 5 and its radius is 1 12
2.2 Two trees T and T' rooted at vertex 1. The tree T' is a ternary tree of height 3. The tree T is of height 4. 13
2.3 A complete graph K_5, a complete bipartite graph $K_{3,2}$ and a $4 \times 5 - grid$. 13
2.4 A graph G (left) and an optimal tree decomposition $\mathcal{T} = (T, \chi)$ of G (right). 14
2.5 The construction of fill-in graphs for the graph G in the Figure 2.4 based on the elimination ordering $L = (3, 4, 2, 5, 1)$. The largest cliques associated with the fill-in graph 15
2.6 The undirected and the directed fill-in graph for the graph G in the Figure 2.4 associated with the elimination ordering $L = (3, 4, 2, 5, 1)$. 15

4.1 A hypergraph H (left) and an optimal branch decomposition (T, γ) of H (right). The labels of the leaves of T are the edges assigned to them by γ and the labels of the edges of T are the cut vertices of that edge. 34
4.2 An alternative branch decomposition (T', γ') of the hypergraph H from Fugue 4.1. The labels of the leaves of the branch decompositions T and T' are the same and are exactly the edges for the hypergraph H. The two decompositions T and T' only differ in the edges. 34
4.3 A branch decomposition (T, γ) (left), of width 3 and depth 4, and a corresponding derivation $\mathcal{P}$ (right), of width 3 and depth 4. 39
4.4 A hypergraph H (left) and an optimal carving (T, γ) of H (right). The labels of the leaves of T are the vertices assigned to them by γ and the labels of the edges of T are the cut edges of that edge. 44
4.5 The graph H used to illustrate the main idea behind our local improvement procedure. 47
4.6 A branch decomposition $\mathcal{B}$ of the graph H given in Fig. 4.5 together with an example of a local branch decomposition $\mathcal{B}_L$ (highlighted by thicker edges) chosen by our algorithm. 47
4.7 The improved branch decomposition $\mathcal{B}'$ obtained from $\mathcal{B}$ after replacing the local branch decomposition $\mathcal{B}_L$ of $H(T_L)$ with an optimal branch decomposition $\mathcal{B}'_L$ of $H(T_L)$ obtained from our SAT-encoding. See Fig. 4.6 for an illustration of $\mathcal{B}$ and $\mathcal{B}_L$. 48

5.1 A graph G (left) and an optimal tree decomposition $\mathcal{T} = (T, \chi)$ of G (right). 61

6.1 A graph G (left), an optimal (special) tree decomposition $\mathcal{T} = (T, \chi)$ of G (middle), and an optimal path decomposition (right). 73
6.2 A biconnected graph G and its block-cut tree T. 75
6.3 A graph G with maximal clique $C = \{2, 3, 4, 5\}$ and its optimal tree decomposition T, containing C in its root bag. The vertices 3 and 4 are twin vertices. 76
6.4 The graph P_4 used in the proof of Lemma 6.2. Bold edges indicate that all edges between the connected sets of vertices are present. 78
6.5 The graph H (left) and the graph G (right) from the proof of Lemma 6.6. . 81
6.6 Cactus plot representing the number of solved instances with at most 60 vertices from TWLIB by each configuration for special treewidth encoding 104
6.7 Cactus plot representing the number of solved instances with at most 60 vertices from TWLIB for each configuration of the pathwidth encodings. . 105

7.1 A hypergraphs H (left) and hypergraph H_3 obtained after applying all preprocessing rules. H_1 is obtained from H after applying contained hyperedge preprocessing. H_2 is obtained from H_1 after applying deletion of vertices of degree 1 preprocessing. Finally, H_3 is obtained from H_2 after applying simplicial vertex preprocessing. 119
7.2 Runtime in seconds on the considered benchmark instance. `k-grid`). vbest refers to the virtual best solver. The x-axis labels consecutive integers that identify instances. The instances are ordered by running time, individually for each solver. 127

List of Tables

2.1 An illustration of the behavior of the sequential counter, which counts at most 4, counting the number of variables from the set $V = \{1, \dots, 6\}$ set to true. The last four columns in the table provide the intermediate values of the sequential counter. 10

4.1 Distribution of solving time in seconds for various values of w for some famous named graphs of branchwidth 6. 53
4.2 Exact branchwidth and carving-width of the famous named graphs. . . . 54
4.3 Results for SAT-based local improvement for a selection of example instances from TreewidthLIB. 56
4.4 Sum of improvements over all the instances for the various configurations. 56
4.5 Maximum improvement over all the instances for the various configurations. 57
4.6 Number of improved instance for the various configurations. 57

5.1 Summary of treewidth improvements. 65
5.2 Some of the best and notable improvements 66

6.1 Applicability of the considered preprocessing procedures to pathwidth and special treewidth. A ✓indicates the preprocessing procedure is applicable, a ✗indicates that it is not. If the procedure is not applicable, but still allows for the computation of an approximate solution, we indicate the approximation error in parenthesis. 78
6.2 Experimental results for the benchmark set of well-known named graphs. A detailed description of the table can be found in Section 6.7.1. 101
6.3 Running-time in seconds for each call of ω between 1 and 10 for one of the well known graphs, Dodecahedron, which has special treewidth 6. 102
6.4 Number of solved instances from TWLIB for all applicable combinations of encodings and preprocessing rules for *special treewidth*, restricted to instances containing at most 60 vertices. A detailed description of the results in this table can be found in Section 6.7.1. 103
6.5 Number of solved instances from TWLIB for all applicable combinations of encodings and preprocessing rules and GDSAT for *pathwidth*, restricted to instances containing at most 60 vertices. A detailed description of the results in this table can be found in Section 6.7.1. 103

6.6 Experimental results for square-grids, complete graphs, and complete bipartite graphs. For each standard graph class, the table shows the maximum number of vertices for which the width of the graph could still be determined exactly by each of the four encodings as well as GDSAT within the given timeout. See also Section 6.7.1 for an explanation of the results. 105
6.7 Total number of random graphs solved by our encodings and GDSAT within the timeout. See Section 6.7.1 for more information about the table. . . . 106
6.8 Percentage of random graphs solved within the timeout using the ordering-based encoding for *special treewidth* for all combinations of n (number of vertices; represented by the rows) and p (edge probability; represented by the columns). Refer to Section 6.7.1 for more information about the table. . . 106
6.9 Percentage of random graphs solved within the timeout using the partition-based encoding for *special treewidth* for all combinations of n (number of vertices; represented by the rows) and p (edge probability; represented by the columns). Refer to Section 6.7.1 for more information about the table. . . 107
6.10 Percentage of random graphs solved within the timeout using the ordering-based encoding for *pathwidth* for all combinations of n (number of vertices; represented by the rows) and p (edge probability; represented by the columns). Refer to Section 6.7.1 for more information about the table. 107
6.11 Percentage of random graphs solved within the timeout using the partition-based encoding for *pathwidth* for all combinations of n (number of vertices; represented by the rows) and p (edge probability; represented by the columns). Refer to Section 6.7.1 for more information about the table. 107
6.12 The number of variables and clauses for our four encodings in terms of the number n of vertices, the number m of edges m, and the width ω 108

7.1 Overview on the number N of instances for which the respective decomposer configuration outputted the exact (fractional) hypertree width of the instance. Configuration: c/C represents disabled or enabled symmetry breaking and lower bound techniques, respectively. p/P represents disabled or enabled preprocessing techniques. 0,4,6 represents a k-hyperclique for $k = 0$, 4, or 6. t median (avg, std) represents the median (average, standard deviation) of the runtime in seconds of the decomposer over all instances of our benchmark instances, including the timeouts. 126
7.2 Distribution gives the intervals of the fractional hypertree width w of the instances for which we successfully obtained an exact result using the best solver configuration. (The interval $(i, j]$ includes all widths w such that $i < w \leq j$.) 126

List of Algorithms

4.1 Local Improvement 49
4.2 ImproveLD 50
4.3 Local Selection (LocalBD) 51

Bibliography

[ABLP16] G. Audemard, F. Boussemart, C. Lecoutre, and C. Piette. XCSP3: an XML-based format designed to represent combinatorial constrained problems. `http://xcsp.org`, 2016.

[ABR+10] Isolde Adler, Binh-Minh Bui-Xuan, Yuri Rabinovich, Gabriel Renault, Jan Arne Telle, and Martin Vatshelle. On the Boolean-Width of a Graph: Structure and Applications. In Dimitrios M. Thilikos, editor, *Graph Theoretic Concepts in Computer Science - 36th International Workshop, WG 2010, Zarós, Crete, Greece, June 28-30, 2010 Revised Papers*, volume 6410 of *Lecture Notes in Computer Science*, pages 159–170, 2010.

[ACP87] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987.

[AD16] Mario Alviano and Carmine Dodaro. Anytime answer set optimization via unsatisfiable core shrinking. *Theory Pract. Log. Program.*, 16(5-6):533–551, 2016.

[AGCM15] Patricia C. Arocena, Boris Glavic, Radu Ciucanu, and Renée J. Miller. The iBench Integration Metadata Generator. In Chen Li and Volker Markl, editors, *Proceedings of Very Large Data Bases (VLDB) Endowment*, volume 9:3, pages 108–119. VLDB, November 2015.

[AMW17] Michael Abseher, Nysret Musliu, and Stefan Woltran. htd – A Free, Open-Source Framework for (Customized) Tree Decompositions and Beyond. In Domenico Salvagnin and Michele Lombardi, editors, *Proceedings of the 14th International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR'17)*, 2017.

[AP89] Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1):11–24, 1989.

[AR02] Michael Alekhnovich and Alexander A. Razborov. Satisfiability, Branch-Width and Tseitin Tautologies. In *43rd Symposium on Foundations of Computer Science (FOCS 2002), 16-19 November 2002, Vancouver, BC, Canada, Proceedings*, pages 593–603. IEEE Computer Society, 2002.

[BB73] Umberto Bertele and Francesco Brioschi. On non-serial dynamic programming. *Journal of Combinatorial Theory, Series A*, 14(2):137–148, 1973.

[BB06] Emgad H Bachoore and Hans L Bodlaender. A branch and bound algorithm for exact, upper, and lower bounds on treewidth. In *International Conference on Algorithmic Applications in Management*, pages 255–266. Springer, 2006.

[BBE16] Max Bannach, Sebastian Berndt, and Thorsten Ehlers. Jdrasil: A Modular Library for Computing Tree Decompositions. Technical report, Lübeck University, Germany, 2016.

[BBE17] Max Bannach, Sebastian Berndt, and Thorsten Ehlers. Jdrasil: A Modular Library for Computing Tree Decompositions. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, volume 75 of *LIPIcs*, pages 28:1–28:21. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[BBN+13] Therese C. Biedl, Thomas Bläsius, Benjamin Niedermann, Martin Nöllenburg, Roman Prutkin, and Ignaz Rutter. Using ILP/SAT to Determine Pathwidth, Visibility Representations, and other Grid-Based Graph Drawings. In Stephen K. Wismath and Alexander Wolff, editors, *Graph Drawing - 21st International Symposium, GD 2013, Bordeaux, France, September 23-25, 2013, Revised Selected Papers*, volume 8242 of *Lecture Notes in Computer Science*, pages 460–471. Springer, 2013.

[BCDM17] Therese C. Biedl, Markus Chimani, Martin Derka, and Petra Mutzel. Crossing Number for Graphs with Bounded~Pathwidth. In Yoshio Okamoto and Takeshi Tokuyama, editors, *28th International Symposium on Algorithms and Computation, ISAAC 2017, December 9-12, 2017, Phuket, Thailand*, volume 92 of *LIPIcs*, pages 13:1–13:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[BDP03] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and Complexity Results for #SAT and Bayesian Inference. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pages 340–351. IEEE Computer Society, 2003.

[BGHK95] Hans L. Bodlaender, John R. Gilbert, Hjálmtyr Hafsteinsson, and Ton Kloks. Approximating Treewidth, Pathwidth, Frontsize, and Shortest Elimination Tree. *J. Algorithms*, 18(2):238–255, 1995.

[BGT98] Hans L. Bodlaender, Jens Gustedt, and Jan Arne Telle. Linear-Time Register Allocation for a Fixed Number of Registers. In Howard J. Karloff, editor, *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, 25-27 January 1998, San Francisco, California.*, pages 574–583. ACM/SIAM, 1998.

[Bie14] Therese C. Biedl. On area-optimal planar graph drawings. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 198–210. Springer, 2014.

[BJ14] Jeremias Berg and Matti Järvisalo. SAT-Based Approaches to Treewidth Computation: An Evaluation. In *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014*, pages 328–335. IEEE Computer Society, 2014.

[BK08] Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial Optimization on Graphs of Bounded Treewidth. *Comput. J.*, 51(3):255–269, 2008.

[BK10] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations. I. Upper bounds. *Information and Computation*, 208(3):259–275, 2010.

[BK11] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations II. Lower bounds. *Inf. Comput.*, 209(7):1103–1119, 2011.

[BKK13] Hans L. Bodlaender, Stefan Kratsch, and Vincent J. C. Kreuzen. Fixed-Parameter Tractability and Characterizations of Small Special Treewidth. In Andreas Brandstädt, Klaus Jansen, and Rüdiger Reischuk, editors, *Graph-Theoretic Concepts in Computer Science - 39th International Workshop, WG 2013, Lübeck, Germany, June 19-21, 2013, Revised Papers*, volume 8165 of *Lecture Notes in Computer Science*, pages 88–99. Springer, 2013.

[BKK+17] Hans L. Bodlaender, Stefan Kratsch, Vincent J. C. Kreuzen, O-joung Kwon, and Seongmin Ok. Characterizing width two for variants of treewidth. *Discr. Appl. Math.*, 216(part 1):29–46, 2017.

[BKM+17] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. Benchmarking the Chase. In Floris Geerts, editor, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'17)*, pages 37–52, Chicago, Illinois, USA, 2017. Assoc. Comput. Mach., New York.

[BLJS17] J. Berg, N. Lodha, M. Järvisalo, and S. Szeider. MaxSAT benchmarks based on determining generalized hypertree-width. Technical report, MaxSAT Evaluation 2017, 2017.

[BM93] Hans L. Bodlaender and Rolf H. Möhring. The Pathwidth and Treewidth of Cographs. *SIAM J. Discrete Math.*, 6(2):181–188, 1993.

[Bod98] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoret. Comput. Sci.*, 209(1-2):1–45, 1998.

[Bod05] H. L. Bodlaender. Discovering treewidth. In *Proceedings of the 31st Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'05)*, volume 3381 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag, 2005.

[Bod16] Hans Bodlander. TreewidthLIB A benchmark for algorithms for Treewidth and related graph problems, 2016. `http://www.staff.science.uu.nl/~bodla101/treewidthlib/`.

[BV12] Therese C. Biedl and Martin Vatshelle. The point-set embeddability problem for plane graphs. In Tamal K. Dey and Sue Whitesides, editors, *Symposuim on Computational Geometry 2012, SoCG '12, Chapel Hill, NC, USA, June 17-20, 2012*, pages 41–50. ACM, 2012.

[CC16] Clément Carbonnel and Martin C. Cooper. Tractability in constraint satisfaction problems: a survey. *Constraints*, 21(2):115–144, 2016.

[CJG08] David Cohen, Peter Jeavons, and Marc Gyssens. A unified theory of structural tractability for constraint satisfaction problems. *J. of Computer and System Sciences*, 74(5):721–743, 2008.

[CMR01] B. Courcelle, J. A. Makowsky, and U. Rotics. On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. *Discr. Appl. Math.*, 108(1-2):23–52, 2001.

[CMZ12] Markus Chimani, Petra Mutzel, and Bernd Zey. Improved Steiner tree algorithms for bounded treewidth. *J. Discrete Algorithms*, 16:67–78, 2012.

[Com08] Microprocessor Standards Committee. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[Cor01] Gérard Cornuéjols. Combinatorial Optimization: Packing and Covering. Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2001.

[Cou90] Bruno Courcelle. The Monadic Second-Order Logic of Graphs. I. Recognizable Sets of Finite Graphs. *Inf. Comput.*, 85(1):12–75, 1990.

[Cou10] Bruno Courcelle. Special tree-width and the verification of monadic second-order graph properties. In Kamal Lodaya and Meena Mahajan, editors,

IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India, volume 8 of *LIPIcs*, pages 13–29. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[Cou12] Bruno Courcelle. On the model-checking of monadic second-order formulas with edge set quantifications. *Discrete Applied Mathematics*, 160(6):866–887, 2012.

[CS03] William Cook and Paul Seymour. Tour merging via branch-decomposition. *INFORMS J. Comput.*, 15(3):233–248, 2003.

[Dar03] Adnan Darwiche. A differential approach to inference in Bayesian networks. *J. ACM*, 50(3):280–305, 2003.

[Dar09] Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks.* Cambridge University Press, 2009.

[Dec99] Rina Dechter. Bucket Elimination: A Unifying Framework for Reasoning. *Artif. Intell.*, 113(1-2):41–85, 1999.

[Dec06] Rina Dechter. Tractable Structures for Constraint Satisfaction Problems. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume I, chapter 7, pages 209–244. Elsevier, 2006.

[Dec13a] Rina Dechter. Graphical Model Algorithms at UC Irvine. Technical report, UC Irvine, 2013. `http://graphmod.ics.uci.edu/group`. The network instances consist of Bayesian and Markov networks used in UAI competition and protein folding/side-chain prediction problems.

[Dec13b] Rina Dechter. *Reasoning with Probabilistic and Deterministic Graphical Models: Exact Algorithms.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2013.

[DF13] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity.* Texts in Computer Science. Springer, 2013.

[DFK+08] Vida Dujmovic, Michael R. Fellows, Matthew Kitching, Giuseppe Liotta, Catherine McCartin, Naomi Nishimura, Prabhakar Ragde, Frances A. Rosamond, Sue Whitesides, and David R. Wood. On the Parameterized Complexity of Layered Graph Drawing. *Algorithmica*, 52(2):267–292, 2008.

[DHJ+17] Holger Dell, Thore Husfeldt, Bart M. P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. The First Parameterized Algorithms and Computational Experiments Challenge. In Jiong Guo and Danny Hermelin, editors, *11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*, volume 63 of *Leibniz International Proceedings*

in Informatics (LIPIcs), pages 30:1–30:9, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Die95] Reinhard Diestel. Graph Minors 1: A Short Proof of the Path-width Theorem. *Combinatorics, Probability & Computing*, 4:27–30, 1995.

[Die00] Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer Verlag, New York, 2nd edition, 2000.

[DIM] DIMACS. The second DIMACS implementation challenge: 1992-1993. NP hard problems: Maximum clique, graph coloring, and satisfiability. `http://dimacs.rutgers.edu/Challenges`.

[DKTW17] Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. In Daniel Lokshtanov and Naomi Nishimura, editors, *Proceedings of the 12th International Symposium on Parameterized and Exact Computation (IPEC'17)*, LIPIcs, pages 30:1—30:13, 2017.

[DKTW18] Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. In Daniel Lokshtanov and Naomi Nishimura, editors, *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 30:1–30:12, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[DM15] Arnaud Durand and Stefan Mengel. Structural tractability of counting of solutions to conjunctive queries. *Theoret. Comput. Sci.*, 57(4):1202–1249, 2015.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems Tools (TACS'08)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Verlag, 2008.

[DMW02] Vida Dujmovic, Pat Morin, and David R. Wood. Path-Width and Three-Dimensional Straight-Line Grid Drawings of Graphs. In Stephen G. Kobourov and Michael T. Goodrich, editors, *Graph Drawing, 10th International Symposium, GD 2002, Irvine, CA, USA, August 26-28, 2002, Revised Papers*, volume 2528 of *Lecture Notes in Computer Science*, pages 42–53. Springer, 2002.

[DR16] Holger Dell and Frances Rosamond. The Parameterized Algorithms and Computational Experiments Challenge. `https://pacechallenge.wordpress.com/`, 2016.

[FBN15] Stefan Fafianie, Hans L. Bodlaender, and Jesper Nederlof. Speeding Up Dynamic Programming with Representative Sets: An Experimental Evaluation of Algorithms for Steiner Tree on Tree Decompositions. *Algorithmica*, 71(3):636–660, 2015.

[FGLP17] Wolfgang Fischl, Georg Gottlob, Davide M. Longo, and Reinhard Pichler. HyperBench: a benchmark of hypergraphs. `http://hyperbench.dbai.tuwien.ac.at`, 2017.

[FHLS18a] Johannes Klaus Fichte, Markus Hecher, Neha Lodha, and Stefan Szeider. FraSMT, 2018. `https://www.ac.tuwien.ac.at/research/frasmt/`.

[FHLS18b] Johannes Klaus Fichte, Markus Hecher, Neha Lodha, and Stefan Szeider. An SMT approach to fractional hypertree width. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2018.

[Fic16] Johannes K. Fichte. daajoe/gtfs2graphs – A GTFS transit feed to Graph Format Converter. `https://github.com/daajoe/gtfs2graphs`, 2016.

[Fis18] Wolfgang Fischl. hypergraph collection. Personal Communication., 2018.

[FLH15] Stefan Falkner, Marius Thomas Lindauer, and Frank Hutter. SpySMAC: Automated Configuration and Performance Analysis of SAT Solvers. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 215–222. Springer, 2015.

[FLS17a] Johannes K. Fichte, Neha Lodha, and Stefan Szeider. trellis: TREewidth LocaL Improvement Solver. `https://github.com/daajoe/trellis`, 2017.

[FLS17b] Johannes Klaus Fichte, Neha Lodha, and Stefan Szeider. SAT-Based Local Improvement for Finding Tree Decompositions of Small Width. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 401–411. Springer, 2017.

[FLS17c] Johannes Klaus Fichte, Neha Lodha, and Stefan Szeider. trellis, 2017. `https://github.com/daajoe/trellis`.

[FM09] John Franco and John Martin. A history of satisfiabilty. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 13, pages 425–454. IOS Press, 2009.

[FMT09] Fedor V. Fomin, Frédéric Mazoit, and Ioan Todinca. Computing branchwidth via efficient triangulations and blocks. *Discr. Appl. Math.*, 157(12):2726–2736, 2009.

[Fre82] Eugene C. Freuder. A sufficient condition for backtrack-bounded search. *J. ACM*, 29(1):24–32, 1982.

[Fre85] Eugene C. Freuder. A sufficient condition for backtrack-bounded search. *J. ACM*, 32(4):755–761, 1985.

[GD04] Vibhav Gogate and Rina Dechter. A Complete Anytime Algorithm for Treewidth. In *Proceedings of the Proceedings of the Twentieth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-04)*, pages 201–208, Arlington, Virginia, 2004. AUAI Press.

[GKKS17] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *CoRR*, abs/1705.09811, 2017.

[GLS99] Georg Gottlob, Nicola Leone, and Francesco Scarcello. On Tractable Queries and Constraints. In *Database and Expert Systems Applications, 10th International Conference, DEXA '99, Florence, Italy, August 30 - September 3, 1999, Proceedings*, volume 1677 of *Lecture Notes in Computer Science*, pages 1–15, 1999.

[GLS02] G. Gottlob, N. Leone, and F. Scarcello. Hypertree Decompositions and Tractable Queries. *J. of Computer and System Sciences*, 64(3):579–627, 2002.

[GM06] Martin Grohe and Dániel Marx. Constraint Solving via Fractional Edge Covers. In *Proceedings of the of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, pages 289–298. ACM Press, 2006.

[GM14] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Transactions on Algorithms*, 11(1):Art. 4, 20, 2014.

[GMPS14] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and cleaning. In Isabel Cruz, Elena Ferrari, and Yufei Tao, editors, *Proceedings of the IEEE 30th International Conference on Data Engineering (ICDE'14)*, pages 232–243, March 2014.

[Got18] Georg Gottlob. detkdecomp. Personal Communication, 2018.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[GPW10] Georg Gottlob, Reinhard Pichler, and Fang Wei. Bounded treewidth as a key to tractability of knowledge representation and reasoning. *Artificial Intelligence*, 174(1):105–132, 2010.

[Gro08] Martin Grohe. Logic, graphs, and algorithms. In Jörg Flum, Erich Grädel, and Thomas Wilke, editors, *Logic and Automata: History and Perspectives*, volume 2 of *Texts in Logic and Games*, pages 357–422. Amsterdam University Press, 2008.

[GS09a] Georg Gottlob and Marko Samer. A backtracking-based algorithm for hypertree decomposition. *ACM J. Exp. Algorithmics*, 13(Issue publication date previously given as 2008):Article 1.1, 19, 2009.

[GS09b] Georg Gottlob and Marko Samer. A Backtracking-based Algorithm for Hypertree Decomposition. *jea*, 13:1:1.1–1:1.19, February 2009.

[Hal76] Rudolf Halin. S-functions for graphs. *Journal of geometry*, 8(1-2):171–186, 1976.

[Hic02] I.V. Hicks. Branchwidth Heuristics. *Congr. Numer.*, 159:31–50, 2002.

[Hic05] Illya V. Hicks. Graphs, branchwidth, and tangles! Oh my! *Networks*, 45(2):55–60, 2005.

[Hli03] Petr Hlinený. Crossing-number critical graphs have bounded path-width. *J. Comb. Theory, Ser. B*, 88(2):347–367, 2003.

[HMS15a] Thomas Hammerl, Nysret Musliu, and Werner Schafhauser. Metaheuristic Algorithms and Tree Decomposition. In Janusz Kacprzyk and Witold Pedrycz, editors, *Springer Handbook of Computational Intelligence*, pages 1255–1270. Springer Verlag, Berlin, Heidelberg, 2015.

[HMS15b] Thomas Hammerl, Nysret Musliu, and Werner Schafhauser. Metaheuristic Algorithms and Tree Decomposition. In Janusz Kacprzyk and Witold Pedrycz, editors, *Springer Handbook of Computational Intelligence*, pages 1255–1270. Springer, 2015.

[HO08] Petr Hliněný and Sang-il Oum. Finding branch-decompositions and rank-decompositions. *SIAM J. Comput.*, 38(3):1012–1032, 2008.

[HS15] Marijn Heule and Stefan Szeider. A SAT Approach to Clique-Width. *ACM Trans. Comput. Log.*, 16(3):24, 2015.

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In Travis Vaught Gäel Varoquaux and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference (SciPy'08)*, pages 11–15, Pasadena, CA, USA, August 2008.

[KGOD11] Kalev Kask, Andrew Gelfand, Lars Otten, and Rina Dechter. Pushing the Power of Stochastic Greedy Ordering Schemes for Inference in Graphical

Models. In Wolfram Burgard and Dan Roth, editors, *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011.* AAAI Press, 2011.

[Kin92] Nancy G. Kinnersley. The vertex separation number of a graph equals its path-width. *Information Processing Letters*, 42(6):345–350, 1992.

[Kit17] Klemens Kittan. Zuse Cluster. `http://www.cs.uni-potsdam.de/bs/research/labsZuse.html`, 2017.

[Klo94] T. Kloks. *Treewidth: Computations and Approximations.* Springer Verlag, Berlin, 1994.

[KNR15] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: Questions Asked Frequently. *CoRR*, abs/1504.04044, 2015.

[KS96] Haim Kaplan and Ron Shamir. Pathwidth, Bandwidth, and Completion Problems to Proper Interval Graphs with Small Cliques. *SIAM J. Comput.*, 25(3):540–561, 1996.

[KSR$^+$17] Roland Kaminski, Marius Schneider, Tobias Rabener, et al. benchmark-tool. `https://github.com/potassco/benchmark-tool`, 2017.

[KT92] András Kornai and Zsolt Tuza. Narrowness, pathwidth, and their application in natural language processing. *Discrete Applied Mathematics*, 36(1):87–92, 1992.

[KvHK99] Arie M. C. A. Koster, Stan P. M. van Hoesel, and Antoon W. J. Kolen. Solving Frequency Assignment Problems via Tree-Decomposition[1]. *Electronic Notes in Discrete Mathematics*, 3:102–105, 1999.

[LGM$^+$15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How Good Are Query Optimizers, Really? *Proceedings of Very Large Data Bases (VLDB) Endowment*, 9(3):204–215, November 2015.

[LOS16a] Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. A SAT Approach to Branchwidth. In Nadia Creignou and Daniel Le Berre, editors, *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing, SAT 2016*, volume 9710 of *Lecture Notes in Computer Science*, pages 179–195. Springer Verlag, 2016.

[LOS16b] Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. BranchLIS, 2016. `https://www.ac.tuwien.ac.at/research/branchlis/`.

[LOS17a] Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. SAT-encodings for special treewidth and pathwidth. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th*

International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings, volume 10491 of *Lecture Notes in Computer Science*, pages 429–445. Springer, 2017.

[LOS17b] Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. Special Treewidth and Pathwidth, 2017. `https://github.com/nehal73/SATencoding`.

[LS88a] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *J. Roy. Statist. Soc. Ser. B*, 50(2):157–224, 1988.

[LS88b] Steffen L Lauritzen and David J Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society, Series B (Methodological)*, -:157–224, 1988.

[LS17] Lukas Larisch and Felix Salfelder. https://github.com/freetdi/p17, 2017.

[Mar10] Dániel Marx. Approximating fractional hypertree width. *TALG*, 6(2):Art. 29, 17, 2010.

[Mar13] Dániel Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *JACM*, 60(6):Art. 42, 51, 2013.

[MMP+12] Michael Morak, Nysret Musliu, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. Evaluating Tree-Decomposition Based Algorithms for Answer Set Programming. In Youssef Hamadi and Marc Schoenauer, editors, *Learning and Intelligent Optimization - 6th International Conference, LION 6, Paris, France, January 16-20, 2012, Revised Selected Papers*, volume 7219 of *Lecture Notes in Computer Science*, pages 130–144. Springer, 2012.

[OD14] Umut Oztok and Adnan Darwiche. CV-width: A New Complexity Parameter for CNFs. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 675–680. IOS Press, 2014.

[OMK+79] Tatsuo Ohtsuki, Hajimu Mori, Ernest S. Kuh, Toshinobu Kashiwabara, and Toshio Fujisawa. One-dimensional logic gate assignment and interval graphs. In *The IEEE Computer Society's Third International Computer Software and Applications Conference, COMPSAC 1979, 6-8 November, 1979, Chicago, Illinois, USA*, pages 101–106. IEEE, 1979.

[OPB11] Arnold Overwijk, Eelko Penninkx, and Hans L. Bodlaender. A Local Search Algorithm for Branchwidth. In Ivana Cerná, Tibor Gyimóthy, Juraj Hromkovic, Keith G. Jeffery, Rastislav Královic, Marko Vukolic, and Stefan

Wolf, editors, *SOFSEM 2011: Theory and Practice of Computer Science - 37th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 22-28, 2011. Proceedings*, volume 6543 of *Lecture Notes in Computer Science*, pages 444–454. Springer, 2011.

[OS13] Sebastian Ordyniak and Stefan Szeider. Parameterized Complexity Results for Exact Bayesian Network Structure Learning. *J. Artif. Intell. Res.*, 46:263–302, 2013.

[Pre09] Steven David Prestwich. Cnf encodings. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 75–97. IOS Press, 2009.

[Ros74] Donald J. Rose. On simple characterizations of k-trees. *Discrete Math.*, 7:317–322, 1974.

[Rou11] Olivier Roussel. Controlling a Solver Execution with the runsolver Tool. *J on Satisfiability, Boolean Modeling and Computation*, 7:139–144, 2011.

[RS83] Neil Robertson and P. D. Seymour. Graph minors. I. Excluding a forest. *J. Combin. Theory Ser. B*, 35(1):39–61, 1983.

[RS84] Neil Robertson and Paul D Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[RS91] Neil Robertson and P. D. Seymour. Graph minors X. Obstructions to tree-decomposition. *J. Combin. Theory Ser. B*, 52(2):153–190, 1991.

[SG97] Kirill Shoikhet and Dan Geiger. A practical algorithm for finding optimal triangulations. In *AAAI/IAAI*, pages 185–190, 1997.

[Sin05] Carsten Sinz. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In Peter van Beek, editor, *Proceedings of the 11th International Conference Principles and Practice of Constraint Programming, CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer Verlag, 2005.

[SLM+05] Yinglei Song, Chunmei Liu, Russell L. Malmberg, Fangfang Pan, and Liming Cai. Tree Decomposition Based Fast Search of RNA Structures Including Pseudoknots in Genomes. In *Proceedings of the 4th International IEEE Computer Society Computational Systems Bioinformatics Conference, CSB 2005*, pages 223–234. IEEE Computer Society, 2005.

[ST94] P. D. Seymour and R. Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.

[Sud04] Matthew Suderman. Pathwidth And Layered Drawings Of Trees. *Int. J. Comput. Geometry Appl.*, 14(3):203–225, 2004.

[SV09] Marko Samer and Helmut Veith. Encoding Treewidth into SAT. In *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 45–50. Springer Verlag, 2009.

[Tam16] Hisao Tamaki. "TCS-Meiji". "`https://github.com/TCS-Meiji/treewidth-exact`", 2016.

[Tam17a] Hasio Tamaki. https://github.com/TCS-Meiji/treewidth-exact, 2017.

[Tam17b] Hisao Tamaki. Positive-Instance Driven Dynamic Programming for Treewidth. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, volume 87 of *LIPIcs*, pages 68:1–68:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[Tra14] Transaction Processing Performance Council (TPC). TPC-H decision support benchmark. Technical report, TPC, 2014.

[TSB00] Dimitrios M. Thilikos, Maria J. Serna, and Hans L. Bodlaender. Constructive Linear Time Algorithms for Small Cutwidth and Carving-Width. In D. T. Lee and Shang-Hua Teng, editors, *Algorithms and Computation, 11th International Conference, ISAAC 2000, Taipei, Taiwan, December 18-20, 2000, Proceedings*, volume 1969 of *Lecture Notes in Computer Science*, pages 192–203. Springer, 2000.

[Ulu08] Elif Ulusal. *Integer Programming Models for the Branchwidth Problem.* PhD thesis, Texas A&M University, May 2008.

[vdBB10] Jan-Willem van den Broek and Hans Bodlaender. TreewidthLIB – A benchmark for algorithms for Treewidth and related graph problems. Technical report, Faculty of Science, Utrecht University, 2010.

[vR95] G. van Rossum. Python tutorial. Cs-r9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995.

[Wei16] Eric Weisstein. MathWorld online Mathematics resource, 2016. `http://mathworld.wolfram.com`.