TU WIEN Informatics

Scalability for SAT-based combinatorial problem solving

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. André Schidler, BSc
Registration Number 01225113

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr. Stefan Szeider

The dissertation has been reviewed by:

Daniel Le Berre	João Marques-Silva

Vienna, 5th April, 2023

André Schidler

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. André Schidler, BSc

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, 5. April 2023

André Schidler

Acknowledgements

I want to thank my supervisor Stefan Szeider, who took on the task of introducing me to the scientific world in all its peculiar and daunting aspects. Despite a global pandemic, he succeeded at this task, which would have been challenging even at normal times. Although we could not meet for long periods, he managed to guide and support me, whenever needed.

I also want to thank Johannes Fichte, who showed me the first glimpses of academia and convinced me to start a PhD. Furthermore, I want to thank everybody in the LogiCS for creating a doctoral school with a great community for PhD students.

I am grateful to Doris Brazda and Beatrix Buhl for their great help in navigating the bureaucracy often involved in academic endeavors.

I want to thank my colleagues who endorsed my caffeine addiction over great conversations: Ana, Friedrich, Kees, Maya, Rafael, and Robert.

The work presented in this thesis was funded by the FWF (P32441 and W1255) and WWTF (ICT19-065).

Kurzfassung

SAT-Solver sind Programme welche die Erfüllbarkeit aussagenlogischer Formeln bestimmen können. Moderne SAT-Solver können die Erfüllbarkeit von riesigen Formeln, mit millionen von Bedingungen und tausenden von Variablen, bestimmen. Dies ist besonders interessant, da eine große Anzahl kombinatorischer Probleme als aussagenlogische Formel repräsentiert werden kann und diese Probleme dann mittels dieser hochperformanten SAT-Solver gelöst werden können.

Trotz der erstaunlichen Leistungsfähigkeit moderner SAT-Solver reicht die Skalierberkeit oft nicht, um große und komplexe Instanzen zu lösen. Während es viel zu lange dauert, um die Erfüllbarkeit von komplexen Formeln zu lösen, ist dies für zu große Formeln praktisch unmöglich.

In dieser Arbeit präsentieren wir verschiedene Lösungsansätze, um die erwünschte Skalierbarkeit zu erreichen: (i) eine effizientere Repräsentation des kombinatorischen Problems, (ii) „lazy“Ansätze, bei denen nur ein Teil der Bedingungen verwendet wird und (iii) die Einbettung von SAT-Solver-basierten Methoden in heuristische Methoden, bei denen der SAT-Solver für lokale Verbesserungen der Lösung verwendet wird (SLIM).

Abstract

SAT solvers determine whether a given propositional formula is satisfiable. Today's highly engineered SAT solvers can determine the satisfiability of huge formulas with millions of constraints and thousands of variables. Further, stating other combinatorial problems in terms of propositional satisfiability allows the use of this great SAT-solving performance for a large range of combinatorial problems.

Despite the great performance of SAT solvers, scalability becomes an issue whenever the instances become too large or too complex. For complex instances, the SAT solver may not be able to determine satisfiability in a reasonable amount of time and for large instances, the corresponding formula becomes too large, and solving becomes practically impossible.

In this thesis, we propose different methods for overcoming these scalability issues. In particular, we discuss examples of efficient encoding design, scalability using a lazy approach, and embedding SAT-based methods into a heuristic approach using SAT-based local improvement (SLIM). These methods are discussed in the context of applications: we propose encodings for computing hypertree width and twin-width; a lazy approach to the directed feedback vertex problem; and SLIM approaches to graph coloring and decision tree induction.

Contents

CHAPTER 1

Introduction and Summary of Results

Propositional logic relates statements—or *propositions*—with each other, using propositional variables that can be either true or false. The result is a *formula* in propositional logic. Determining whether a given formula is satisfiable—whether there is some assignment to the formula's variables that makes the formula true—plays an important role in complexity theory and many practical applications. The *propositional satisfiability problem (SAT)* is not only an NP-complete problem but the first problem shown to be NP-complete (Cook, 1971; Levin, 1973). Despite its theoretical hardness, powerful *SAT solvers* exist that can determine the satisfiability of even large propositional formulas.

The NP-completeness gives SAT a central role in computational complexity and allows many interesting problems to be expressed—or *encoded*—in terms of SAT. Consider how we can use a SAT solver to solve the following scheduling problem.

Example 1.1. *We want to schedule a meeting with the following people and availabilities:*

- *Person A has time on Monday and Friday.*
- *Person B can meet any day except Thursday and Friday.*
- *Person C has time on Monday and Tuesday.*
- *Person D cannot meet on Tuesday, Wednesday, or Thursday.*

We can represent these constraints in terms of propositional logic using one propositional—or Boolean—variable per possible day. Whenever a variable is set to true, all people are free on the corresponding day. The resulting formula looks as follows:

(Mon or Fri) and (not Thu and not Fri) and (Mon or Tue) and (not Tue and not Wed and not Thu).

We can reformulate the formula using logical symbols:

$$(Mon \vee Fri) \wedge (\neg Thu \wedge \neg Fri) \wedge (Mon \vee Tue) \wedge (\neg Tue \wedge \neg Wed \wedge \neg Thu),$$

and give the formula to a SAT solver, which tells us that Monday is the only possible day for the meeting.

The possibility of encoding various interesting problems into SAT and the availability of powerful SAT solvers makes SAT highly relevant for practical applications. One of the most important applications for SAT solvers is *bounded model checking* (Biere et al., 2003). Bounded model checking formally verifies the correctness of specifications, particularly for hardware, such as microchips (Biere et al., 1999; Biere, 2021) and software (Kroening, 2021). *Equivalence checking* (Kuehlmann et al., 2002)—checking whether two Boolean functions, given for instance as circuits, are equivalent—is another important task that is often tackled with SAT solvers. The power and applicability of modern SAT solvers got widespread attention when the SAT-powered solution to the Pythagorean triples problem resulted in the largest mathematical proof at the time (Heule et al., 2016) of over 200 Terabytes. In Chapter 4, we will see a class of propositional formulas with millions of variables whose satisfiability modern SAT solver can determine in minutes.

While finding the best way to encode a problem into SAT can be challenging, creating the encoding is usually easy. The whole program creating the encoded Pythagorean triples instance has only 26 lines of C++ code. Another advantage of using SAT encodings is their high flexibility, as we can easily add further constraints. Consider the example above. Adding further people or even meeting room availabilities only requires adding constraints to the formula. Further, instead of having a variable per day, we can subdivide each day into 24 hours and increase the precision of our scheduling. Starting from an idea for finding a day when four people are available, we can, with very few modifications, schedule meetings in a large company with hundreds of people.

The actual solving is delegated to the SAT solver, and the solving algorithm performs the hard work of solving the encoded problem. Dedicated algorithms for determining satisfiability have been known since at least the 1960s. The DPLL algorithm (Davis et al., 1962) uses several rules that speed up the search compared to a simple exploration of all possible assignments. With several improvements, the DPLL algorithm was the state-of-the-art for several decades. In the late 1990s, the extension from DPLL to *conflict-driven clause learning (CDCL)* (Silva and Sakallah, 1999; Moskewicz et al., 2001) achieved significantly higher performance and is the basis for the success of modern SAT solvers. CDCL improves upon DPLL in two key aspects: the algorithm learns additional constraints to avoid repeating unsuccessful decisions in later iterations, and these additional constraints allow CDCL a more efficient exploration of the possible assignments. Today, most SAT solvers, and all we use in this thesis, are based on CDCL.

SAT solvers have become sophisticated and highly engineered pieces of software. Although the core algorithm is still CDCL, two decades of work refined every part of the algorithm:

implementation details, heuristics, data structures, pre-, and in-processing. The progress achieved by algorithm engineering for SAT is "*nothing short of spectacular*" (Vardi, 2014). The annual SAT competition[1] shows that despite this stunning performance, significant improvements are achieved yearly. These improvements are highly relevant for SAT-based approaches. Since we can simply replace the older solver with the newer one and use the same encoding, we benefit from the improvements without much extra effort.

With conventional SAT solvers, maximization problems can only be solved by repeatedly asking the solver, "is there a solution larger or equal than k?" and increasing k every time. As soon as the solver answers "no," we know that the last "yes" answer corresponds to the maximum solution. *MaxSAT* extends SAT with the capability to encode optimization problems directly. Hence, we can ask a MaxSAT solver directly, "what is the maximum solution to our encoded instance?" Another extension of SAT, *SAT modulo theory (SMT)* allows for encoding optimization problems and extends propositional logic by fragments from the more powerful first-order logic. While there are many capabilities that SMT introduces, particularly interesting in the context of this thesis is the support of numerical variables. MaxSAT solvers usually solve optimization problems considerably faster than repeatedly calling SAT solvers, and SMT allows encoding problems that are hard to express in propositional logic. Both SMT and MaxSAT solvers are built on top of SAT solvers, and we will see the benefits of using these SAT extensions in Chapter 3.

Despite their capabilities, SAT solvers have limits that hinder their applicability. A frequent issue is that the problem instance we want to encode is already too large, and the encoded instance is usually even larger. E.g., as we will see in this thesis, the encoded instance's size for graph problems is often cubic in the number of the input graph's vertices. Hence, encoding such a problem for a graph with 500 vertices requires over 125 million constraints. If we want to encode larger graphs, we quickly exceed the memory of the machine running the SAT solver. Even if we do not exceed the machine's memory, too many constraints can severely slow down the solver to the point where it takes exceedingly long to determine the satisfiability of the formula.

Worse, even small formulas can be too hard to solve in the desired amount of time if they are too *complex*. It is well known that specific problems, like the pigeonhole principle (Haken, 1985), are notoriously hard for SAT solvers, even though the encoded instances are small. In general, whether the SAT solver succeeds depends on how many assignments the solver has to explore until it either finds a satisfying assignment or can determine that no satisfying assignment exists. As a result, the solver's success is often depending on the considered instances and the chosen encoding.

These issues necessitate approaches scaling SAT-based methods to large and complex formulas. Depending on the issue, different scalability approaches can help. Some problems require sophisticated encodings, such as the encodings we discuss in Chapters 3 and 4. Here, refining the encoding can help scale to larger instances. These refinements include finding a more succinct way to express the problem in propositional logic, tuning

[1] https://satcompetition.github.io/

the encoding to be more amenable to the way CDCL works, and adding constraints restricting the number of assignments the solver has to consider.

Even with refinements, there are limits to the encoding's succinctness. Particularly when the instances we want to solve are large, even the most succinct SAT encoding might not suffice. This issue can be addressed using *lazy encodings*. Here, we call the SAT solver repeatedly using only a small part of the formula. Whenever the SAT solver determines that this part is unsatisfiable, we know that the entire formula is unsatisfiable. Otherwise, the solver returns a satisfying assignment, and we repeatedly extend the partial formula until the solver returns a satisfying assignment that satisfies the entire formula. The part of the formula that suffices for finding an overall solution is often very small compared to the entire formula. Therefore, while the entire formula would by far exceed the capabilities of the SAT solver or even the machine, a lazy approach can still manage to find a solution. In Chapter 5, we achieve scalability for an encoding that is exponentially larger than the encoded problem using a lazy encoding.

Whenever we exceed the limits of the previous methods, we can still find a good solution to optimization instances by combining SAT-encodings with heuristic approaches. The algorithmic framework *SAT-based local improvement (SLIM)* starts from a sub-optimal solution and repeatedly improves smaller—local—parts of the solution using a SAT solver. Focusing on local parts ensures that the encoding's size and complexity are manageable for the SAT solver, which can find the optimal solution for the local part, thereby improving the overall solution. This approach is very flexible, and we show in Chapters 6 and 7 how diverse its applications can be.

In this thesis, we focus on these three approaches to scalability: (i) refining the SAT encoding itself, (ii) using lazy encodings, and (iii) embedding the SAT encoding into a heuristic method within the SLIM framework.

1.1 Summary of Results

1.1.1 Encoding Refinement

Encoding a problem into SAT is often not straightforward. While some problems, such as graph coloring or vertex cover, have a straightforward encoding, other problems require a specific characterization that lends itself to be encoded into SAT. In this thesis, we explore two problems that require a non-trivial characterization. Both problems, determining *hypertree width* and *twin-width* of a (hyper)graph, stem from the general area of width parameters for (hyper)graphs. For hypertree width, we show how developing different characterizations can be beneficial to scalability. In the case of twin-width, we show how developing different encodings based on the same characterization can yield very different results.

Hypertree Width

Hypertree-width (Gottlob et al., 2002) is a generalization of treewidth to hypergraphs. Intuitively, hypertree-width measures how close the hypergraph's primal graph is to a tree. This hypergraph parameter can be used for specialized—or parameterized—algorithms that are faster on hypergraphs of low width than common algorithms. Hypertree width is of interest for problems that can be represented as a hypergraph, such as constraint satisfaction problems and database queries, which become tractable for hypergraph classes of bounded hypertree width. Currently, a Google scholar search for "hypertree width" yields about 1500 results, further underlining its importance. Nonetheless, few methods have been developed to compute it.

In Chapter 3, we propose two SAT encodings for hypertree width that beat the state-of-the-art approaches. The two encodings are based on two different characterizations of hypertree width with complementary performance: depending on properties of the hypergraph, one encoding may perform better than the other, with no clear winner. Furthermore, we explore how different solving paradigms (SAT, MaxSAT, SMT) perform and again see a complementary behavior: the MaxSAT solver performs better for large hypergraphs, while the SMT solver performs better for hypergraphs with large hypertree width.

In the context of scalability, it is particularly interesting to look at the development of the encodings. One of the encodings won PACE 2019 (Schidler and Szeider, 2020; Dzulfikar et al., 2019), beating the state-of-the-art dynamic programming solver. Nonetheless, using a new characterization, our second encoding (Schidler and Szeider, 2021b) achieved significantly better performance. The versions presented in this thesis are refinements of these two encodings, and these refinements achieve similar performance, further highlighting the importance of encoding refinement.

Twin-Width

Twin-width (Bonnet et al., 2022) is a graph parameter that, similar to hypertree width, allows for efficient solving of NP-hard problems on graphs of bounded twin-width. In particular, twin-width allows for fixed-parameter tractable first-order model checking, parameterized by the length of the first-order formula. Many NP-hard problems, like the independent set problem, can be expressed in first-order logic and thus become tractable for graph classes of bounded twin-width. Already the problem of determining if a graph has twin-width ≤ 4 is known to be NP-hard (Bergé et al., 2022).

Twin-width is based on *contractions*: in a contraction, we replace two vertices of a graph with a new vertex adjacent to exactly the neighbors of the two contracted vertices. *Twins* are two vertices where contracting them is identical to simply removing one of them. Twin-width is based on pairwise contracting vertices until the entire graph has been contracted to a single vertex by a sequence of contractions (a *contraction sequence*). Twin-width measures, in a certain sense, how close a graph is to being reducible to a single vertex by contracting only twins.

We propose two SAT encodings for computing the exact twin-width of a graph in Chapter 4. Our encoding is, to date, the only algorithm able to compute the twin-width of graphs. Both encodings are based on our novel characterization of twin-width, and we show how differently encoding the same characterization can lead to more succinct and better scalable encodings. The encodings presented in this thesis are refined versions of our previously proposed encodings (Schidler and Szeider, 2022).

1.1.2 Lazy Encodings

Engineering a good encoding is not necessarily enough for good scaling. Fortunately, we often do not need to encode the entire instance. Often a small set of constraints is sufficient for finding a solution or determining that no solution exists. This fact is used in lazy approaches. Here, one starts with a small subset of all constraints. Whenever a solution is found, it is checked against the full set of constraints. Violated constraints are added to the current subset, and the SAT solver is run again until, eventually, a solution is found that satisfies all constraints. In practice, for many instances, the necessary subset is comparatively small. We explore lazy encodings in the context of the *directed feedback vertex set problem*.

Directed Feedback Vertex Set

The directed feedback vertex set (DFVS) problem is one of Karp's original 21 NP-complete problems (Karp, 1972). Given a directed graph, a directed feedback vertex set is a subset of the set of vertices, such that the graph without the vertices of the subset is acyclic. We solve the problem of finding a minimum DFVS.

A straightforward SAT encoding for the DFVS problem checks whether a vertex can reach itself via any of its successors. This encoding has a size cubic in the number of vertices in the input graph. Another approach is listing all cycles and computing a hitting set: a subset of vertices that contains at least one vertex from each cycle. Since there can be exponentially many cycles in the number of the graph's vertices, this encoding can become even larger than the previous one. Given the large encoding sizes, it is impossible to encode DFVS instances where the graph has several thousands of vertices.

In Chapter 5, we discuss our lazy approach to DFVS that won PACE 2022 and scales to large graphs (Kiesel and Schidler, 2023).[2] We initially encode only some short cycles and give it to a MaxSAT solver. Then, whenever the solution fails to break some cycles, we add more cycles until we obtain a valid solution. Furthermore, we show how we can integrate this logic directly in the MaxSAT solver to further speed up our approach.

1.1.3 SAT-based Local Improvement (SLIM)

So far, we have been able to compute exact solutions to the problems. While the previously discussed methods provide scalability, at some point, instances become too

[2] https://pacechallenge.org/2022/

large to be solved exactly, and we search for good heuristic solutions. Nonetheless, we want these heuristic solutions to be as close to the optimal solution as possible. As it turns out, embedding SAT-based methods into heuristics, the basic idea behind SLIM, is an excellent way of finding good heuristic solutions. SLIM starts from an initial heuristic solution and improves it through a series of local improvements. Key is a method that extracts small *local instances* from the heuristic solution. These local instances are small enough to be tackled by a SAT solver and constructed so that their solution can be reintegrated into the original global solution, such that the local solution improves the heuristic solution.

We explore two SLIM approaches in the course of this thesis, one for graph coloring and one for decision tree learning. The two approaches show how different the details of SLIM approaches can be within the general SLIM framework.

Graph Coloring

Graph coloring is another one of Karp's 21 NP-complete problems (Karp, 1972): given a graph, we want to color each vertex such that adjacent vertices have different colors using as few colors as possible. While encoding the problem into SAT is straightforward and the resulting encoding is small compared to the ones discussed above, large or dense graphs usually result in complex instances.

We propose GC-SLIM (Schidler, 2022), our SLIM approach to graph coloring, in Chapter 6. GC-SLIM can find good colorings for large and dense graphs by removing one color after the other from an initial coloring. The key idea is to recolor subgraphs instead of the entire graph. For each subgraph, GC-SLIM tries to remove a specific color until it is not used anywhere in the graph. By using a SAT encoding for *list-coloring*, we ensure that the new coloring for the subgraph is consistent with the remainder of the graph outside the subgraph. The list-coloring restriction ensures that no vertex in the subgraph gets assigned a color of a neighbor outside the subgraph.

In our experiments, GC-SLIM is able to improve the coloring of state-of-the-art heuristics for graphs with several hundred thousand vertices and more than 1.5 billion edges. GC-SLIM was among the top solvers in the 2022 CG:SHOP challenge (Fekete et al., 2022).

Decision Tree Induction

Decision trees are inherently interpretable machine learning models: it is comparatively easy for a human to understand a given decision tree model, and the path taken for a decision explains the reasons for the decision. The search for interpretable machine learning models has increased the interest in decision trees. In this context, small and shallow decision trees are preferred: the more nodes a decision tree has, the harder it becomes to comprehend. Further, the depth of the decision tree limits the length of the explanation for a single decision. We use the term complexity to refer to the decision tree's size or depth.

Unfortunately, finding decision trees of minimum complexity is an NP-hard problem. SAT encodings are among the methods that have been proposed for finding low-complexity decision trees. Starting with Narodytska et al.'s (2018) encoding, several encodings were proposed in the last years, with the latest encoding (Shati et al., 2021) scaling to medium-sized classification instances. While the performance of the encodings is impressive, many classification instances from standard machine learning repositories are still too large to be tackled directly by them.

Our SLIM approach DT-SLIM (Schidler and Szeider, 2021a) in Chapter 7 scales these encodings to very large decision trees and classification instances. DT-SLIM iteratively improves the complexity of a given decision tree by selecting different subtrees and reducing their complexity using a SAT encoding. Whenever the SAT encoding can find a subtree of lower complexity, it improves the complexity of the whole decision tree.

1.2 Publications

This thesis is based on the following publications.

Chapter 3: Hypertree Decompositions

André Schidler and Stefan Szeider. Computing optimal hypertree decompositions. In Guy E. Blelloch and Irene Finocchi, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*, pages 1–11. SIAM, 2020. doi: 10.1137/1.9781611976007.1. URL https://doi.org/10.1137/1.9781611976007.1

André Schidler and Stefan Szeider. Computing optimal hypertree decompositions with SAT. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, 2021b. doi: 10.24963/ijcai.2021/196. URL https://doi.org/10.24963/ijcai.2021/196

Chapter 4: Twin-Width

André Schidler and Stefan Szeider. A SAT approach to twin-width. In Cynthia A. Phillips and Bettina Speckmann, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2022, Alexandria, VA, USA, January 9-10, 2022*, pages 67–77. SIAM, 2022. doi: 10.1137/1.9781611977042.6. URL https://doi.org/10.1137/1.9781611977042.6

Chapter 5: Directed Feedback Vertex Set

Rafael Kiesel and André Schidler. A dynamic maxsat-based approach to directed feedback vertex sets. In Gonzalo Navarro and Julian Shun, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2023, Florence, Italy,*

January 22-23, 2023, pages 39–52. SIAM, 2023. doi: 10.1137/1.9781611977561.ch4. URL https://doi.org/10.1137/1.9781611977561.ch4.

Chapter 6: Graph Coloring

André Schidler. SAT-based local search for plane subgraph partitions (CG challenge). In Xavier Goaoc and Michael Kerber, editors, *38th International Symposium on Computational Geometry, SoCG 2022, June 7-10, 2022, Berlin, Germany*, volume 224 of *LIPIcs*, pages 74:1–74:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.SoCG.2022.74. URL https://doi.org/10.4230/LIPIcs.SoCG.2022.74

Chapter 7: Decision Trees

André Schidler and Stefan Szeider. SAT-based decision tree learning for large data sets. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Virtual Event, February 2-9, 2021*, pages 3904–3912. AAAI Press, 2021a. URL https://ojs.aaai.org/index.php/AAAI/article/view/16509

CHAPTER 2

Preliminaries

We introduce some general notation and terminology that is relevant to all chapters. Each chapter will introduce further background knowledge relevant for the chapter.

2.1 Graphs

If not stated differently, we consider a graph to be connected, simple and undirected. An undirected graph G consists of a set of vertices $V(G)$ and a set of edges $E(G)$. We will often assume without loss of generality that $V(G) = \{1, \ldots, |V(G)|\}$. We denote the edge between vertices $v, w \in V(G)$ by $\{v, w\}$. For $X \subseteq V(G)$ we denote by $N_G(X) = \{\, u : \{u, v\} \in E(G) \text{ with } v \in X \text{ and } u \notin X \,\}$ the *neighborhood* of X. We write $N_G(v)$ instead of $N_G(\{v\})$ and drop the subscript if G is clear from the context. $G - X$ denotes the graph G' with $V(G') = V(G) \setminus X$ and $E(G') = \{\, \{u, v\} \in E(G) : u, v \in V(G') \,\}$.

2.2 Propositional Satisfiability

A propositional logic formula F is defined over a set V of Boolean variables. Each variable can take the values *true* and *false*. We give the definition of propositional formulas in *conjunctive normal form (CNF)*, as it is the input format we use and every propositional logic formula can be transformed into an equisatisfiable CNF formula in polynomial time (Tseitin, 1983; Plaisted and Greenbaum, 1986).

The smallest unit in a CNF formula is a single variable $v \in V$, it can occur unnegated or negated $\neg v$. Unnegated and negated variables are called *literals*. We say a literal ℓ is *negative* if it is of the form $\ell = \neg v$ and *positive* if $\ell = v$. *Disjunctions* combine one or more literals. A disjunction between two literals v_1 and $\neg v_2$ is denoted as $v_1 \vee \neg v_2$ and a disjunction of a set L of literals as $\bigvee_{\ell \in L} \ell$. A CNF formula is a *conjunction* of clauses. The conjunction of two clauses C_1 and C_2 is denoted by $C_1 \wedge C_2$, and the conjunction of a set C of clauses as $\bigwedge_{v \in L} v$. Example 2.1 shows a CNF.

Example 2.1. $(v_1 \vee v_2 \vee \neg v_3) \wedge (\neg v_2 \vee v_3) \wedge \neg v_1$.

It is often convenient to denote a CNF as a set of clauses, and a clause as a set of literals. The set representation for Example 2.1 would be $\{\{v_1, v_2, \neg v_3\}, \{\neg v_2, v_3\}, \{\neg v_1\}\}$.

A propositional formula is evaluated based on an *assignment* $\sigma : V \to \{\text{true}, \text{false}\}$ of variables to values. In case σ is a partial function, we say it is a *partial assignment*. A positive literal is *satisfied* if the corresponding variable is assigned true, otherwise, it is *unsatisfied*. Similarly, the negative literal is satisfied if the corresponding variable is assigned the value false. Further, a clause is satisfied if at least one of its literals is satisfied, and an empty disjunction can not be satisfied. Finally, a CNF is satisfied if all of its clauses are satisfied. An empty CNF is always satisfied.

The *propositional satisfiability problem (SAT)* asks: given a propositional formula F, does there exist an assignment σ that satisfies F? The satisfying assignments of F are called *models*.

In the remainder of this work, we will use the term *formula* to refer to propositional formulas. For brevity, we will occasionally use logical equivalency $C_1 \leftrightarrow C_2$, where C_1 and C_2 are clauses, and any assignment satisfies either both or none of them.

We denote a SAT encoding for some instance $\mathcal{I}$ as $F(\mathcal{I})$.

Example 2.2. *We illustrate encodings into SAT with the problem of finding an* independent set*: given a graph G, we want to find a subset $V' \subseteq V(G)$ such that for all $u, v \in V'$ with $u \neq v$ it holds that $\{u, v\} \notin E$. This can be encoded into SAT by using V as the variables and then $F(G) = \{\{\neg u, \neg v\} : \{u, v\} \in E\}$. Any model for $F(G)$ can then be converted into an independent set, by selecting exactly the vertices where the corresponding variable is assigned the value true.*

While the encoding in the example is correct, a trivial independent set is the empty set. Hence, we usually want to find an independent set of maximum cardinality, which requires encoding an optimization problem in terms of a decision problem instance.

2.3 Optimization

Since SAT is a decision problem, we need a different approach to solve optimization problems. We encode an instance $\mathcal{I}$ of a maximization problem (minimization is analogous) as a formula $F(\mathcal{I}, d)$, where $F(\mathcal{I}, d)$ is satisfiable if and only if there is a solution to $\mathcal{I}$ of value $\geq d$. Whenever $F(\mathcal{I}, d)$ is satisfiable, we increment d. As soon as $F(\mathcal{I}, d)$ is unsatisfiable, we know the optimal value is $d - 1$.

Example 2.3. *In our running example of finding a (maximum) independent set, we want to create a formula $F(G, d)$ such that $F(G, d)$ is satisfiable if and only if there exists an independent set V' such that $|V'| \geq d$.*

In Example 2.3, we need to constrain the cardinality of V'. For optimization problems, we usually have one or more of these *cardinality constraints*. In the remainder of this section, we show how we can encode cardinality constraints and how extensions of SAT allow for other ways of encoding optimization problems.

2.3.1 Cardinality Constraints

Cardinality constraints can be encoded directly as a set of clauses. Several methods have been proposed (Batcher, 1968; Bailleux and Boufkhad, 2003; Sinz, 2005; Asín et al., 2009; Ogawa et al., 2013; Morgado et al., 2014b). The methods differ in various aspects, including the number of clauses they require. In this thesis, we mainly use the *totalizer* constraints (Bailleux and Boufkhad, 2003), as they performed overall best and they can be incrementally adapted (Martins et al., 2014). Due to their simplicity, we sometimes use variations of the *sequential counters* (Sinz, 2005). In the remainder of this work, we assume that we can express cardinality constraints in propositional logic.

The advantage of using these cardinality constraints is that $F(\mathcal{I}, d)$ remains a formula in propositional logic, and we can use SAT solvers to find an optimal solution. The disadvantages are that cardinality constraints can introduce a very large number of additional clauses and little information is used from one value of d to another that can speed up the search.

The disadvantages give rise to extensions of SAT that allow for explicitly stating both cardinality constraints and optimization problems, which we discuss next.

2.3.2 Maximum Satisfiability (MaxSAT)

MaxSAT allows expressing optimization problems by stating $F(\mathcal{I})$ and the optimization goal separately. We give the definition of *partial MaxSAT* as it is the formalism we use in this thesis. MaxSAT instances are solved using special MaxSAT solvers that, in turn, use SAT solvers in a strategic way that speeds up the search for an optimal solution.

A MaxSAT instance consists of two sets of clauses, the *hard* clauses, given by $F(\mathcal{I})$ and the *soft* clauses, given by $F_s(\mathcal{I})$. A solution for the instance is a model for $F(\mathcal{I})$ that maximizes the number of satisfied clauses in $F_s(\mathcal{I})$. This allows us to directly encode optimization problems in MaxSAT, as the following example shows.

Example 2.4. *We can express the maximum independent set problem as a partial MaxSAT instance as follows. As hard clauses we use $F(G)$ as defined a above and $F_s(G) = \{\{v\} : v \in V\}$. Any independent set obtained from the solution will then contain the maximum number of vertices.*

2.3.3 SAT Modulo Theory (SMT)

SMT extends propositional logic by fragments of first-order logic (Barrett et al., 2009; Monniaux, 2016) and provides many more capabilities than those relevant for this thesis.

Of particular interest for our purposes are variables over integers, arithmetic constraints, and optimization.

SMT introduces variables over integers, which allows expressing Boolean variables as integer variables with values 1 and 0 instead of true and false. Together with arithmetic constraints, we can express the cardinality constraint $|X| \leq d$ as the following example shows.

Example 2.5. *We can express the cardinality constraint $|V'| \geq d$ from our running example by using for each vertex $v \in V(G)$ an integer variable v that has value 1 if $v \in V'$ and value 0 otherwise. We can then express each propositional clause $\neg u \vee \neg v$ as a linear constraint $u + v \leq 1$ and the cardinality constraint as $\sum_{v \in V} v \geq d$. Finally, d is stated as an explicit maximization goal for the SMT solver.*

In the example, we only have integer variables but in more complex encodings integer and Boolean variables are combined, as we see in Chapter 3.

Integer linear programming and *pseudo Boolean* solvers offer the same capability of expressing cardinality constraints in this fashion. We do not consider integer linear programming solvers as we focus on SAT-based methods, and we do not consider pseudo Boolean solvers explicitly, as they did not perform well enough to be comparable to SAT/MaxSAT/SMT solvers in the discussed work.

2.4 SAT-Based Local Improvement (SLIM)

SAT-Based Local Improvement (SLIM) is an *anytime* meta-heuristic that embeds SAT encodings into heuristic algorithms. SLIM is used for instances that cannot be solved directly by a SAT solver due to their size or complexity. Starting from a heuristically computed *global solution*, SLIM finds a better solution by repeatedly improving small parts of the global solution.

Key to this approach is the notion of a *local instance*: a method that converts a small—local—part of the global solution into an instance that can be encoded into SAT. This notion of a local instance must guarantee *replacement consistency*: the solution for the local instance must be able to replace the local part of the global solution such that the global solution remains a feasible solution, and the replacement improves the global solution.

Local instances are extracted by a *selection strategy* that picks the most promising local parts of the global solution that are within a given *budget*. The budget ensures that the SAT solver can solve the instance within a specified *local timeout*. This local timeout presents a good balance between finding enough improvements, how much the SAT solver sees of the global solution, and a reasonable chance for the SAT solver to find an improvement if one exists.

Concrete SLIM approaches are discussed in Chapters 6 and 7.

2.5 Related Work

Here, we discuss the related work for the overarching topic; we will discuss related work specific to the different chapters separately. One method for scaling SAT-based approaches that we do not explore in this thesis is *cube and conquer* (Heule et al., 2011). Cube and conquer splits the instance into many sub-problems and uses several solvers in parallel to solve these sub-problems. The approach is suitable whenever there is a hard or large instance we want to solve, such as for hard math problems (Heule et al., 2016; Heule, 2018; Subercaseaux and Heule, 2022). In this thesis, we focus on methods that, once developed, can be easily applied to different instances of the same problem without further effort.

SAT Encodings

SAT encodings are a popular means for solving hard combinatorial problems, as the work in this thesis and numerous papers show. Given this overwhelming body of work, we focus on work on width parameters for graphs, which are most relevant for the novel SAT encodings presented in this thesis. Encodings have been proposed for treewidth (Samer and Veith, 2009; Berg and Järvisalo, 2014; Bannach et al., 2017); clique-width (Heule and Szeider, 2015), branchwidth (Lodha et al., 2019), special treewidth and pathwidth (Lodha et al., 2017); treedepth and treecut-width (Ganian et al., 2019). For hypergraphs, encodings for generalized hypertree width (Berg et al., 2017) and fractional hypertree width (Fichte et al., 2018, 2020) are known.

The novel encodings introduced in this thesis take methods from this prior work but require novel ideas specific to the problem. The encoding for hypertree width in Chapter 3 is based on the encodings by Berg et al. (2017) and Fichte et al. (2018) for generalized hypertree width, but extra effort was necessary to encode the particular properties of hypertree width. The twin-width encoding in Chapter 4 follows the approach for treewidth by Samer and Veith (2009), which resulted in an encoding that is quartic in the size of the input graph. We discuss how the size can be reduced to cubic, and thereby, better scalability can be reached with twin-width specific adaptions. Surprisingly, the ideas behind the clique-width encoding by Heule and Szeider (2015) can be used for decision tree induction, as we will discuss in Chapter 7.

Lazy Encodings

The idea of lazy encodings is used in various other methods for solving combinatorial problems. The idea of branch & cut used in integer linear programming is a lazy approach that has many concrete problem-specific instantiations, such as logic-based benders decomposition (Hooker, 2000) for logic-based constraint programming. In the context of SAT, the idea is often referred to as *counter-example guided abstraction refinement (CEGAR)* originally introduced by Clarke et al. (2003) for bounded model checking. CEGAR has been successfully used for various other problems, such as Hamiltonian cycles (Soh et al., 2014), graph coloring (Glorian et al., 2019), propositional

circumscription (Janota et al., 2010), SMT solving (Brummayer and Biere, 2009), QBF Solving (Janota et al., 2016), and encoding other PSPACE-hard problems (Lagniez et al., 2017; Seipp and Helmert, 2018).

The extra clauses can be generated at various points in the solving process. This is often done after the solver returns a model for the partial encoding that is not a model for the full encoding. This approach has the advantage that it is solver-agnostic but can be slow and may introduce more clauses than are necessary for solving the instance. The idea of generating the extra clauses during the solving, before the solver returns the assignment, is facilitated using *propagators*, which allow for adding extra logic at different points of the solver's solving process. This has been successfully used for SMT solving (Brummayer and Biere, 2009) and dynamic symmetry breaking (Kirchweger and Szeider, 2021). We show how a CEGAR approach for combinatorial problem solving can be implemented using propagators in Chapter 5.

SLIM

SLIM has shown to be effective in various applications. Although conceptually similar to Large Neighborhood Search (Pisinger and Ropke, 2010), SLIM defines the neighbourhood specifically for the use with SAT and related solvers. SLIM was originally used for graph decomposition problems (Lodha et al., 2019, 2017; Fichte et al., 2017; Ramaswamy and Szeider, 2020). Further applications for Boolean circuits (Kulikov et al., 2022; Reichl et al., 2023) and Bayesian network structure learning (Ramaswamy and Szeider, 2021b,a, 2022) show the versatility of the approach as a general framework for applying SAT-based methods within heuristic algorithms. While successful as a general framework, each SLIM instantiation requires problem-specific ideas, as we will explore in Chapters 6 and 7.

CHAPTER 3

Hypertree Decompositions

3.1 Introduction

Hypertree width is a popular hypergraph invariant which was introduced by Gottlob et al. (2002). Hypertree width is of fundamental nature, as underpinned by the existence of combinatorial, game-theoretic, as well as logical characterizations (Gottlob et al., 2003). In their original paper, Gottlob et al. (2002) showed that critical NP-hard problems arising in databases and constraint satisfaction are polynomial-time tractable for instances whose associated hypergraph has bounded hypertree width. The hypergraph invariant has found many further applications, including Projected Solution Counting, Solution Enumeration (with polynomial delay), Constraint Optimization, and Combinatorial Auctions (see, for example, the survey article (Gottlob et al., 2014)).

Key to these tractability results is the fact that for any fixed constant bound W, one can decide in polynomial time[1] whether a given hypergraph has hypertree width up to W, and in the positive case compute a witnessing *hypertree decomposition* of width W. Thus, in terms of parameterized complexity (Downey and Fellows, 2013), the problem of recognizing hypergraphs of hypertree width W is in XP, when parameterized by W. However, the problem is known to be W[2]-hard (Gottlob et al., 2005) and hence unlikely to be fixed-parameter tractable. The original XP-algorithm by Gottlob et al. (2002) was later improved by Gottlob and Samer (2009), and their implementation of the algorithm (det-k-decomp) represented for several years the state-of-the-art for practically computing optimal hypertree decompositions.

We propose a new practical approach for computing the exact hypertree width of hypergraphs. We follow a logical approach which was initiated by Samer and Veith (Samer and Veith, 2009) for tree decompositions and was later successfully used for other

[1] The recognition problem is complete for the complexity class LOGCFL, which is contained in AC^1 and hence highly parallelizable.

(hyper)graph width measures, including clique-width (Heule and Szeider, 2015), treecut width and treedepth (Ganian et al., 2019), fractional hypertree width (Fichte et al., 2018), and twin-width (Chapter 4). For hypertree width, we had to introduce several new concepts and ideas to make this approach work.

We propose two encodings, based on two different characterizations of hypertree width. Both characterizations are *ordering-based*, where we arrange the vertices of the given hypergraph in a linear ordering subject to certain constraints. Successful SAT-encodings for treewidth and fractional hypertree width (Samer and Veith, 2009; Fichte et al., 2018) already used ordering-based characterizations of the corresponding width measures (for treewidth, this characterization uses the well-known characterization of graphs of bounded tree-width in terms of partial k-trees (Bodlaender, 2005)). However, for hypertree width, an ordering-based characterization is not straightforward. What makes it challenging to express hypertree width in terms of a linear ordering is the *Special Condition* in the definition of hypertree decompositions (see Section 3.2), which is formulated in terms of the *descendancy relation* in the decomposition tree. However, we succeeded in formulating two characterizations in such a way that we could base compact and efficient SAT encodings on it.

Our first encoding, based on the *augmented hypertree ordering* characterization of hypertree width, uses in addition to the linear ordering additional relational information, which allows us to express the Special Condition in a way that closely relates to the original definition of hypertree width.

Our second encoding, based on the *pure hypertree ordering* characterization of hypertree width, avoids the additional relational information and is, therefore, more compact. This characterization is conceptually more elegant than the augmented one but requires more formal arguments to establish its equivalence with the original definition of hypertree width.

For bounding the width of an augmented or pure hypertree ordering, we need to compute small hyperedge covers of vertex sets. For this purpose, we use in our encodings not only propositional cardinality constraints (as Samer and Veith did, see 3.4.1) but also soft clauses in conjunction with a *MaxSAT solver* and certain *arithmetic constraints*. During the solving process, these arithmetic constraints are mapped to propositional logic in an incremental fashion. This incremental encoding is handled by an *SMT* solver (Barrett et al., 2009; Monniaux, 2016), where a First-Order Logic solver (handling the arithmetic constraints) interacts with the SAT solver.

We implemented the encodings based on augmented and pure hypertree orderings and tested them on an extensive set of benchmark instances (Hyperbench) consisting of real-world hypergraphs from various application domains with a number of vertices and hyperedges ranging up to 2900. We tested two variants, one where the cardinality constraints are encoded in propositional logic (plain SAT and MaxSAT) and one where the cardinality constraints are encoded by arithmetic constraints that are dealt with by the theory component of a SAT Modulo Theory solver (SMT). We compared the two encodings

with each other, and also with the newest version of Gottlob and Samer's combinatorial XP-algorithm *new-det-k-decomp* (Gottlob and Samer, 2009; Fischl et al., 2018a). The results are highly encouraging and show that each of our methods improves upon the state-of-the-art by solving up to 621, or almost 50%, more instances. Furthermore, our new methods work complementary and can solve even more instances in a portfolio approach, combining the different encodings and solver paradigms.

The remainder of the chapter is organized as follows. In Section 3.2, we give basic definitions of hypergraphs, edge covers, and hypertree decompositions. Afterwards in Section 3.3, we introduce our new ordering-based characterization of hypertree width, whose correctness we establish in A.1.1 and A.1.2. In Section 3.4, we explain how the new characterizations can be encoded as a SAT problem. Finally, in Section 3.5, we present our experimental results.

3.2 Preliminaries

A *hypergraph* H consists of a set $V(H)$ of *vertices* and a set $E(H)$ of *hyperedges*, each hyperedge is a subset of $V(H)$. A hypergraph H' is a *partial hypergraph* of hypergraph H if $E(H') \subseteq E(H)$ and $V(H') = \bigcup_{e \in E(H')} e$.

The *primal graph* (or 2-*section*) of a hypergraph H is the graph $G(H)$ with vertex set $V(H)$ and edge set $E(G(H)) = \{\, \{u, v\} : u \neq v$, there is some $e \in E$ such that $\{u, v\} \subseteq e \,\}$. A hypergraph is *connected* if its primal graph is connected, otherwise it is *disconnected*. A *connected component* of a hypergraph H is a maximal connected partial hypergraph of H. A hypergraph H decomposes into its connected components $H_1, \ldots, H_r$, where $V(H_i) \cap V(H_j) = \emptyset$ and $E(H_i) \cap E(H_j) = \emptyset$, $1 \leq i < j \leq r$. We write $\mathrm{CC}(H) = \{H_1, \ldots, H_r\}$ for the set of connected components of H.

Consider a hypergraph H and a set $S \subseteq V$. An *edge cover* of S (with respect to H) is a set $F \subseteq E(H)$ such that for every $v \in S$ there is some $e \in F$ with $v \in e$. The *size* of an edge cover is its cardinality. Given an edge cover F, we use the shorthand $\bigcup F := \bigcup_{e \in F} e$.

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

- **T1** for each $v \in V(H)$ there is some $t \in V(T)$ with $v \in \chi(t)$ ("v is covered by t"),
- **T2** for each $e \in E(H)$ there is some $t \in V(T)$ with $e \subseteq \chi(t)$ ("e is covered by t"),
- **T3** for any three $t, t', t'' \in V(T)$ where t' lies on the path between t and t'' in T, we have $\chi(t) \cap \chi(t'') \subseteq \chi(t')$ ("bags containing the same vertex are connected").

We assume the tree T to be rooted at some arbitrary node $r \in V(T)$, as this will be needed for extending tree decompositions to hypertree decompositions below.

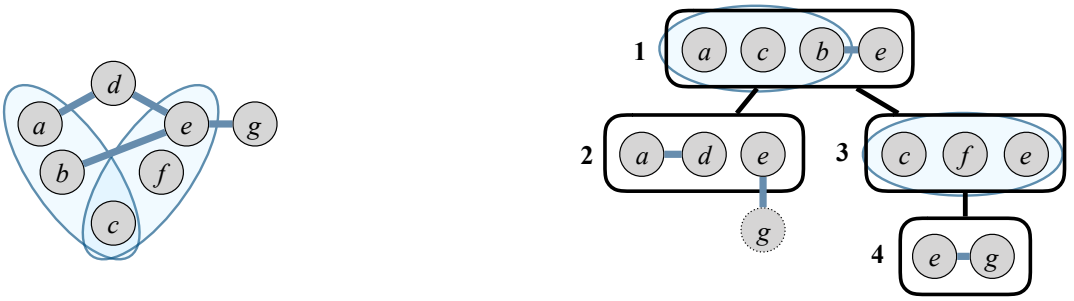


Figure 3.1: An example of a hypergraph H (left) and one of its possible hypertree decompositions $\mathcal{D} = (T_D, \chi_D, \lambda_D)$ (right). The bags $\chi_D(t)$ are represented by rectangles and the hyperedges in the edge covers $\lambda_D(t)$ are indicated by either blue lines or ellipsis. The vertex g is omitted at bag 2, which does not violate T4, as the node has no descendants. A case of a Special Condition violation can easily be constructed if we replace the hyperedge $\{b, e\}$ in the root's edge cover by $\{e, g\}$. T1–T3 would still hold, but as g occurs in a bag of a leaf, T4 gets violated. Furthermore, this conflict could not be resolved by adding g to bag 1, as that would violate T3 since g does not occur in bag 3.

We say that a vertex $v \in V(H)$ is *forgotten* at node $t \in V(T)$ if $v \in \chi_D(t)$, but v is not in the bag of t's parent. Every vertex in the bag of the root node r is forgotten at r. We observe that each vertex v is forgotten at exactly one node t due to T3. Hence, we can write $f(v) = t$.

A *hypertree decomposition* (Gottlob et al., 2002) of H is a triple $\mathcal{D} = (T_D, \chi_D, \lambda_D)$ where (T_D, χ_D) is a tree decomposition of H, and λ_D is a mapping that assigns each $t \in V(T_D)$ an *edge cover* $\lambda_D(t) \subseteq E(H)$ of $\chi_D(t)$. The *width* of $\mathcal{D}$ is the size of a largest edge cover in λ_D. Moreover, the rooted tree T_D satisfies, in addition to T1–T3, also a certain Special Condition (T4). To formulate the Special Condition, we call a vertex v to be *omitted* at a node $t \in V(T_D)$, if $v \notin \chi_D(t)$, but $\lambda_D(t)$ contains a hyperedge e with $v \in e$. The *Special Condition* now states the following:

T4 If a vertex v is omitted at t, then it must not appear in the bag $\chi_D(t')$ of any descendant node t' of t.

In other words, T4 states that if $t, t' \in V(T_D)$ are nodes such that t' is a descendant of t, then for each $e \in \lambda_D(t)$, we have $(e \setminus \chi_D(t)) \cap \chi_D(t') = \emptyset$. The *hypertree width* $htw(H)$ of H is the smallest width over all hypertree decompositions of H. We say a hypertree decomposition $\mathcal{D}$ of a hypergraph H is *optimal* if the width of $\mathcal{D}$ equals $htw(H)$. See Figure 3.1 for an example.

If the decomposition is not required to satisfy the Special Condition, then we call it a *generalized hypertree decomposition*, and we define accordingly the *generalized hypertree width* $ghtw(H)$ of H as the smallest width over all generalized hypertree decompositions

of H. Clearly, $ghtw(H) \leq htw(H)$. It is already NP-hard to decide whether a given hypergraph has generalized hypertree width ≤ 2 (Fischl et al., 2018b), hence dropping the Special Condition increases the parameterized complexity of the recognition problem from XP to para-NP. The even more general parameter *fractional hypertree width* (with the same para-NP-hard recognition problem (Fischl et al., 2018b)) arises when one considers fractional edge covers of the bags instead of edge covers (Grohe and Marx, 2014). Powerful decomposers have been developed for generalized and fractional hypertree width (Fichte et al., 2018; Korhonen et al., 2019).

To avoid trivial cases, we consider only hypergraphs H where each $v \in V(H)$ is contained in at least one $e \in E(H)$. Consequently, every considered hypergraph H has an edge cover, and $htw(H)$ is always defined. If $|V(H)| = 1$ then $htw(H) = ghtw(H) = 1$.

We will also focus on connected hypergraphs since we can proceed component-wise to compute an optimal hypertree decomposition:

Proposition 3.1. *For every hypergraph H we have $htw(H) = \max_{H' \in CC(H)} htw(H')$.*

3.3 Elimination Orderings for Hypertree Width

For this section, we consider a fixed, connected hypergraph H and a linear ordering $v_1 \prec \cdots \prec v_n$ of $V(H)$. We think of the ordering as an *elimination ordering* in the very same sense as has been used in the context of tree decompositions for graphs (Bodlaender, 2005). We eliminate one vertex after the other from the primal graph. Each time a vertex is eliminated, we make all its (remaining) neighbors adjacent (see, for example, (Bodlaender, 2005)), where newly added edges are referred to as *fill-in edges*. For our purposes and the forthcoming encodings, it is convenient to consider this elimination process on a directed version of the primal graph, where edges are oriented with respect to the linear ordering.

To that effect, let the *set of active arcs* A be the smallest subset of $V(H) \times V(H)$ such that

1. if $\{u, v\} \in E(G(H))$ and $u \prec v$ then $(u, v) \in A$, and
2. if $(u, v) \in A$, $(u, w) \in A$ and $v \prec w$ then $(v, w) \in A$.

We can now define the following central concept: For a fixed linear ordering $\prec$ of $V(H)$, a rooted tree $T_\prec$ is a *canonical tree for* $\prec$ if

1. $V(T_\prec) = \{\, \tau_\prec(v) : v \in V(H) \,\}$, i.e., the tree contains one node for each of H's vertices,
2. if r is the $\prec$-largest vertex in $V(H)$, then $\tau_\prec(r)$ is the root of $T_\prec$, and

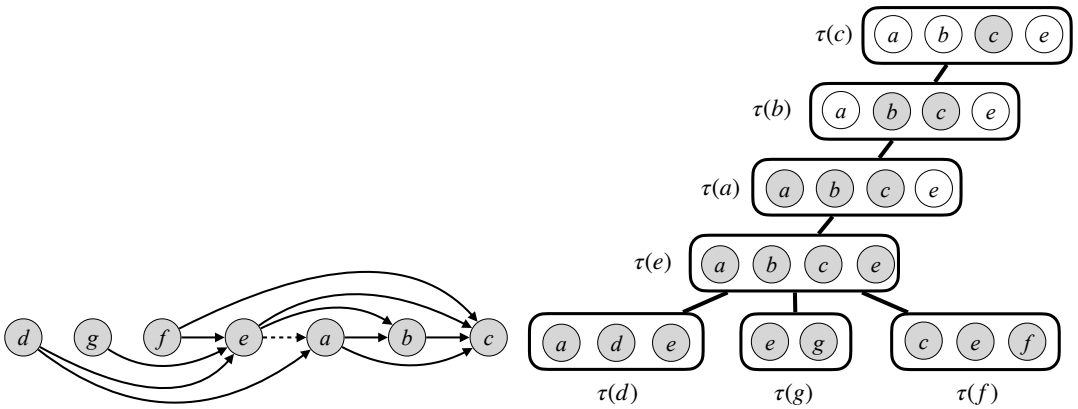


Figure 3.2: An elimination ordering for the hypergraph in Figure 3.1 (top) and the corresponding decomposition (bottom). Solid lines represent hyperedges and the dotted line a fill-in edge. The decomposition without the white vertices shows the canonical translation of the adjacent ordering into a decomposition.

3. for any $u \in V(H) \setminus \{r\}$, if w is the $\prec$-smallest vertex with $(u, w) \in A$, then $\tau_\prec(w)$ is the parent of $\tau_\prec(u)$ in $T_\prec$.

Since all canonical trees are isomorphic, we call $T_\prec$ *the* canonical tree for $\prec$.

In the sequel, we will also use the *transitive closure* A^* of A, i.e., the smallest set containing A with the property that whenever $(u, v) \in A^*$ and $(v, w) \in A^*$, then also $(u, w) \in A^*$. We note that in the last item of the definition of $T_\prec$, we could have used A^* instead of A without changing the definition.

Using these definitions, we can construct a generalized hypertree decomposition $\mathcal{D} = (T_\prec, \chi_D, \lambda_D)$ based on $\prec$, where for each $v \in V(H)$:

- $\chi_D(\tau(v)) := \{v\} \cup \{\, w : (v, w) \in A \,\}$, and
- $\lambda_D(v)$ is a smallest edge cover of $\chi_D(\tau(v))$.

Figure 3.2 (bottom) shows an example of such a decomposition. Here, we can see an inherent problem of this way of characterizing hypertree width in terms of elimination orderings that we need to address ("accumulating hyperedges"): The bag of the root cannot be covered without violating T4, as any vertex adjacent to c occurs in the bag of some descendant. In general, this problem can occur anywhere in the decomposition, not only at the root. In the example, we can resolve the problem by adding the white vertices. We observe that even when we ignore the redundant bags above $\tau_\prec(e)$, we do not get the same decomposition as in Figure 3.1 since this decomposition tree is not the canonical tree of any ordering.

We present two characterizations of hypertree width in terms of elimination orderings that address this issue. The *pure hypertree ordering* uses properties of hypertree decompositions to implicitly constrain the elimination ordering, such that the Special Condition is not violated. The *augmented hypertree ordering* augments the elimination ordering by an equivalence relation to address the issue of accumulating hyperedges. The two characterizations will provide the basis for SAT/MaxSAT/SMT encodings that we will present in Section 3.4.

3.3.1 Pure Hypertree Orderings

Before we can discuss this characterization, we need further concepts and considerations.

We define for each vertex $v \in V(H)$ the set

$$\chi_{\prec}(v) := \{v\} \cup \{\, w : (v, w) \in A \,\}.$$

This definition is closely related to the construction for the generalized hypertree decomposition's bags at the beginning of the section.

For any two vertices $u, w \in V(H)$, such that $u \prec w$, we define their *arc-path*

$$P(u, w) := \{u, w\} \cup \{\, v : (u, v) \in A^*, (v, w) \in A^* \,\},$$

and we say a vertex $w \in V(H)$ is *arc-reachable* from u if $(u, w) \in A^*$.

In Figure 3.2 (top), we can find arc-paths by simply following the arcs from left to right. This illustrates the motivation for arc-paths: whenever one vertex is arc-reachable from another, the corresponding nodes have a descendancy relationship in the canonical tree.

We now introduce the construction that ensures that the Special Condition holds. Assuming a mapping $\lambda_{\prec}$ that assigns an edge cover for $\chi_{\prec}$ to each vertex, we divide the forbidden vertices—vertices that would violate the Special Condition—at any vertex w into two disjoint sets, $B(w)$ and $R(w)$.

$$R(w) := \{\, u : (v, w) \in A^* \text{ and } \lambda_{\prec}(v) \subseteq \lambda_{\prec}(w), \text{for every } v \in P(u, w) \,\}$$
$$B(w) := \{\, u : (v, w) \in A^* \text{ and } \lambda_{\prec}(v) \setminus \lambda_{\prec}(w) \neq \emptyset, \text{for some } v \in P(u, w) \,\}$$

The T4-violations in $R(w)$ can be repaired: whenever every vertex on the arc-path between u and w has as its edge cover a superset of u's edge cover, we can omit u at w. If this property holds, we can add $\chi_{\prec}(u)$ to all the bags along the arc-path. Whenever we can find an ordering and an edge cover such that $B(w)$ is empty for each vertex $w \in V(H)$, we can convert the ordering into a hypertree decomposition.

In Figure 3.2 (bottom), R is represented by the white vertices. For the root, B consists of all vertices except c and the white vertices.

With these definitions in hand, we can now state our first ordering-based characterization of hypertree width.

Definition 3.1 (Pure Hypertree Orderings)**.** *A* pure hypertree ordering *of H is a pair $\mathcal{P} = (\prec, \lambda_\prec)$ where*

- *$\prec$ is a linear ordering of $V(H)$ and*
- *$\lambda_\prec$ is a mapping, assigning to each vertex $v \in V(H)$ an edge cover for $\chi_\prec(v)$,*

such that the following properties hold:

P1 *for all $v \in V(H)$, $\chi_\prec(v) \subseteq \bigcup \lambda_\prec(v)$, and*

P2 *for all $v \in V(H)$, $B(v) \cap \bigcup \lambda_\prec(v) = \emptyset$.*

The width *of the pure hypertree ordering $\mathcal{P}$ is* $\max_{v \in V(H)} |\lambda_\prec(v)|$.

Condition P1 states that $\lambda_\prec$ assigns edge covers to the respective vertices. P2 represents the Special Condition, by defining the distinction between B and R.

Theorem 3.2. *The hypertree width of a connected hypergraph equals the minimum width over all its pure hypertree orderings.*

We prove the theorem in Appendix A.1.1 and introduce our second characterization next.

3.3.2 Augmented Hypertree Orderings

This characterization uses a different approach that allows for a more direct definition of the Special Condition in terms of linear orderings.

Definition 3.2 (Augmented Hypertree Orderings)**.** *An* augmented hypertree ordering *of H is a triple $\mathcal{A} = (\prec, \lambda^{\equiv}_\prec, \equiv)$ where*

- *$\prec$ is a linear ordering of $V(H)$,*
- *$\lambda^{\equiv}_\prec$ is a mapping that assigns each $v \in V(H)$ a set $\lambda^{\equiv}_\prec(v) \subseteq E(H)$, and*
- *$\equiv$ is an equivalence relation on $V(H)$*

such that the following conditions hold:

A1 *For each vertex $v \in V(H)$, $\lambda^{\equiv}_\prec(v)$ is an edge cover of*

$$\chi^{\equiv}_\prec(v) := \{v\} \cup \{\, w : (v, w) \in A \,\} \cup \{\, w \in V(H) : v \equiv w \,\}.$$

A2 *For any two vertices $u, v \in V(H)$ with $(u, v) \in A^*$ and any $e \in \lambda^{\equiv}_\prec(v)$, it holds that $(e \setminus \chi^{\equiv}_\prec(v)_\prec) \cap \chi^{\equiv}_\prec(u) = \emptyset$.*

	Vars	Range	Semantics	Count
Base	$o_{i,j}$	$1 \leq i < j \leq n$	$v_i \prec v_j$	$\binom{n}{2}$
	$o^*_{i,j}$	$1 \leq i, j \leq n, i \neq j$	$v_i \prec v_j$	Shorthand
	$w_{i,k}$	$1 \leq i \leq n, 1 \leq k \leq m$	$e_k \in \lambda_\prec(\tau_\prec(v_i))$	$m \cdot n$
	$a_{i,j}$	$1 \leq i, j \leq n, i \neq j$	$(v_i, v_j) \in A$	$n \cdot (n-1)$
	$b_{i,j}$	$1 \leq i, j \leq n, i \neq j$	$(v_i, v_j) \in A^*$	$n \cdot (n-1)$
Pure	$r_{i,j}$	$1 \leq i, j \leq n, i \neq j$	$v_i \in R(v_j)$	$n \cdot (n-1)$
	$\ell_{i,j}$	$1 \leq i, j \leq n, i \neq j$	T4 permits $v_i \in \bigcup \lambda_\prec(\tau_\prec(v_j))$	$n \cdot (n-1)$
Aug.	$e_{i,j}$	$1 \leq i < j \leq n$	$v_i \equiv v_j$	$\binom{n}{2}$
	$e^*_{i,j}$	$1 \leq i, j \leq n, i \neq j$	$v_i \equiv v_j$	Shorthand

Table 3.1: Variables used in the SAT encoding for a hypergraph with n nodes and m edges.

A3 *For all vertices $u, v, w \in V(H)$, such that $u \equiv w$ and $(u, v), (v, w) \in A^*$, it holds that $u \equiv v$.*

The width *of the augmented hypertree ordering $\mathcal{A}$ is* $\max_{v \in V(H)} |\lambda^{\equiv}_{\prec}(v)|$.

Theorem 3.3. *The hypertree width of a connected hypergraph equals the minimum width over all its augmented hypertree orderings.*

We prove this theorem in Appendix A.1.2.

The equivalence relation directly addresses the problem of accumulating hyperedges. Condition A1 redefines bags as $\chi^{\equiv}_{\prec}$, based on $\equiv$, while Condition A3 ensures that T3 still holds with the definition of $\chi^{\equiv}_{\prec}$. With this new definition in hand, the Special Condition can be expressed as Condition A2. Given that A^* expresses the descendancy relation in the canonical tree, Condition A2 restates T4, replacing the use of the descendancy relation with A^*. This definition allows having already eliminated vertices in a bag and we can thereby deal with accumulating hyperedges. In Figure 3.1, the accumulating hyperedges would be solved by setting $a \equiv b \equiv c$.

Next, we use our characterizations to propose the corresponding encodings.

3.4 Encodings

In this section, we utilize the new characterizations of hypertree width for two new SAT/MaxSAT/SMT encodings. In the basic setup, we follow closely the SAT encoding for treewidth as proposed by Samer and Veith (2009), variants of which served as the basis for several other encodings, for treewidth and other width measures (Bannach et al., 2017; Berg et al., 2017; Fichte et al., 2018; Lodha et al., 2017; Schidler and Szeider, 2022).

Table 3.1 gives an overview of the variables used. We note in passing that the challenge of finding a characterization of hypertree width that can be put on top of the Samer-Veith encoding of treewidth was our original motivation to pursue this work.

Again, we assume a given connected hypergraph $H = (V, E)$ with n vertices $v_1, \ldots, v_n \in V(H)$, m edges $e_1, \ldots, e_m \in E(G(H))$, and a bound W on the hypertree width. The task is to produce a propositional formula $F(H, W)$ in Conjunctive Normal Form, which is satisfiable if and only if $htw(H) \leq W$. The construction will allow us to efficiently transform a satisfying assignment for $F(H, W)$ into a hypertree ordering and into a hypertree decomposition of width $\leq W$.

3.4.1 Base Encoding

First, we review the basic setup following Samer and Veith's (Samer and Veith, 2009) treewidth encoding. The variables $o_{i,j}$ define $\prec$, where $o_{i,j}$ is true if and only if $v_i \prec v_j$. We use the shorthand $o^*_{i,j}$:

$$o^*_{i,j} := \begin{cases} o_{i,j} & \text{if } i < j; \\ \neg o_{j,i} & \text{otherwise.} \end{cases}$$

The following set of clauses establishes transitivity:

$$\bigwedge_{\substack{1 \leq i,j,k \leq n \\ i \neq j \neq k \neq i}} \neg o^*_{i,j} \vee \neg o^*_{j,k} \vee o^*_{i,k}.$$

The variables $a_{i,j}$ define A, where $a_{i,j}$ is true if and only if $(v_i, v_j) \in A$. Initially, A consists of the edges from $E(G(H))$ in forward direction, expressed by the following three sets of clauses:

$$\bigwedge_{\substack{1 \leq i,j \leq n, \\ i \neq j}} \neg o^*_{i,j} \vee \neg a_{j,i}, \qquad \bigwedge_{\{v_i, v_j\} \in E(G(H))} \neg o^*_{i,j} \vee a_{i,j}, \qquad \bigwedge_{\{v_i, v_j\} \in E(G(H))} \neg o^*_{j,i} \vee a_{j,i}. \tag{3.1}$$

The following clauses add fill-in edges and complete the definition of A:

$$\bigwedge_{\substack{1 \leq i,j,k \leq n \\ i \neq j \neq k \neq i}} \neg o^*_{j,k} \vee \neg a_{i,j} \vee \neg a_{i,k} \vee a_{j,k}.$$

Variables $b_{i,j}$ express A^*, where $b_{i,j}$ is true if and only if $(v_i, v_j) \in A^*$. The following clauses initialize A^* with A and enforce transitivity:

$$\bigwedge_{\substack{1 \leq i,j \leq n, \\ i \neq j}} \neg a_{i,j} \vee b_{i,j}, \qquad \bigwedge_{\substack{1 \leq i,j,k \leq n \\ i \neq j \neq k \neq i}} \neg a_{j,k} \vee \neg b_{i,j} \vee b_{i,k}.$$

We conclude the base encoding, by expressing P1/A1 and thereby $\lambda_\prec / \lambda^{\equiv}_\prec$. This follows the encoding by Berg et al. (2017) for generalized hypertree decompositions. The *weight*

variable $w_{i,k}$ is true if and only if $e_k \in \lambda_\prec(v_i)$. For the SMT encoding, instead of Boolean variables, we use integer valued variables. These integer variables can take the values 1 and 0, corresponding to true and false. The following clauses express P1/A1, or $\bigcup \lambda_\prec(v) \supseteq \chi_\prec(v)$:

$$\bigwedge_{1 \leq i \leq n} \bigvee_{\substack{e_k \in E, \\ v_i \in e_k}} w_{i,k}, \qquad \bigwedge_{\substack{1 \leq i,j \leq n, \\ i \neq j}} (\neg a_{i,j} \vee \bigvee_{\substack{e_k \in E, \\ v_j \in e_k}} w_{i,k}).$$

For the SMT encoding, we replace the disjunctions over $w_{i,k}$ by a constraint stating that the sum over the weight variables must be at least 1.

Cardinality Constraints

It remains to add cardinality constraints that encode that each bag has an edge cover of size $\leq W$. We use the SAT encodings with the cardinality constraints discussed in Section 3.4.1.

We also use a MaxSAT encoding. We use the totalizer cardinality constraints as in the SAT encoding. These constraints define the variables $c_{i,k}$, $1 \leq i \leq n$, $1 \leq k \leq W$, where $c_{i,k}$ is true if $|\lambda_\prec(v_i)| \geq k$. Therefore, the encoding so far constitutes the set of hard clauses. Additionally, we add variables m_k for $1 \leq k \leq W$, where m_k is true if the hypertree width is less than k (as by Berg et al. (2017) in the context of generalized hypertree width). We ensure the semantics of m by adding for $1 \leq i \leq n$, $1 \leq k \leq W$, the clauses $\neg m_k \vee \neg c_{i,k}$. Finally, we let the solver minimize the width by adding for $1 \leq k \leq W$ the soft clauses m_k. The MaxSAT solver now finds a solution setting as many variables m_k to true as possible and thereby finds the model corresponding to a hypertree decomposition of minimum width.

An SMT solver allows for more abstract handling of cardinalities by the algebraic theory solver within the SMT approach (as used by Fichte et al. (2018) in the context of fractional hypertree width). In our SMT encoding, we can encode the necessary constraints for $i \leq n$ directly as $\sum_{e_k \in E} w_{i,k} \leq W$. Additionally, SMT solvers can, similar to MaxSAT solvers, automatically search for the minimum W such that the encoding is satisfiable.

This concludes the base encoding. We now discuss the encodings of the pure and augmented characterizations of hypertree orderings, respectively.

3.4.2 Pure Encoding

This encoding expresses property P2 in two parts: (i) define R, and (ii) use this definition to express the Special Condition. We discuss this encoding and a possible improvement in this section.

The repairable T4-violations R are represented by the variables $r_{i,j}$, where $r_{i,j}$ is true if and only if for every vertex v_k on the arc-path between v_i and v_j it holds that $\lambda_\prec(v_i) \subseteq \lambda_\prec(v_k)$. We first encode that every hyperedge in the edge cover of v_i must also

occur in the edge cover of v_j, thereby ensuring $\lambda_\prec(v_i) \subseteq \lambda_\prec(v_j)$, expressed by the clauses

$$\bigwedge_{\substack{1\leq i,j\leq n,\\ 1\leq k\leq m,\\ i\neq j}} \neg r_{i,j} \vee \neg w_{j,k} \vee w_{i,k}. \tag{3.2}$$

This property must hold along the entire arc-path. We express this by stating that the property only holds for v_k if it also holds for all predecessors $v_j \in P(v_i, v_k)$ along the path:

$$\bigwedge_{\substack{1\leq i,j,k\leq n\\ i\neq j\neq k\neq i}} \neg b_{i,j} \vee \neg b_{j,k} \vee \neg r_{i,k} \vee r_{i,j}.$$

The Special Condition can be concisely stated with these two sets of variables. Whenever a vertex v_j is arc-reachable from vertex v_i, the edge cover of v_j must not use any hyperedge containing v_i. The only exception is the case when the edge cover of every vertex on the arc-path is a superset of v_i's edge cover. The following clauses express the Special Condition:

$$\bigwedge_{\substack{1\leq i,j\leq n, e_k\in E(H)\\ i\neq j, v_i\in e_k}} \neg b_{i,j} \vee r_{j,i} \vee \neg w_{j,k}.$$

Improving the encoding of the Special Condition

The number of variables can be further reduced by encoding P2 in a less direct way. This improvement uses two ideas. The first idea is not to encode the forbidden but the permitted vertices, i.e., $V(H) \setminus B(v)$. This makes it possible to use only one set of variables that combines $b_{i,j}$ and $r_{i,j}$. The other idea is to restrict the vertices we consider for R.

We use variables $\ell_{i,j}$, where $\ell_{i,j}$ is true if and only if v_i is permitted in the edge cover of v_j. The first set of clauses restricts the use of hyperedges in the edge covers as before:

$$\bigwedge_{\substack{1\leq i,j\leq n, e_k\in E(H)\\ i\neq j, v_i\in e_k}} \ell_{i,j} \vee \neg w_{j,k},$$

and the second set of clauses ensures that the allowed property holds along arc-paths:

$$\bigwedge_{\substack{1\leq i,j,k\leq n,\\ i\neq j\neq k\neq i}} \neg a_{j,k} \vee \ell_{i,j} \vee \neg \ell_{i,k}.$$

It remains to encode the conditions that define when a vertex is forbidden. Due to the following result, we can restrict the scope of these arc-paths.

Proposition 3.4. *Let $u \in V(H)$ and v be the $\prec$-largest vertex, such that $(u, v) \in A$. For every vertex w such that $v \prec w$, we can remove any hyperedge containing u from $\lambda_\prec(w)$.*

Proof. Consider vertices u, v, and w as in the proposition. If P2 holds before the removal of the hyperedge, it does so afterwards, as the intersection can only become smaller. It remains to show that P1 holds.

From the definition of A, we know that for each vertex v' that is adjacent to u, it holds that either $(u, v') \in A$ or $(v', u) \in A$. w is, therefore, not adjacent to u. Furthermore, there is no vertex w' such that $w \prec w'$ and $\{u, w'\} \in E(G(H))$, as v is the $\prec$-largest vertex with an arc from u. Therefore, for all w' such that $v \prec w'$, it holds that w' is not adjacent to u. This implies that no vertex in $\chi_\prec(w)$ is adjacent to u, and we can safely remove any hyperedge containing v from $\lambda_\prec(w)$, as $\chi_\prec(w)$ will still be covered. □

Using this result, we can discard all vertices that are not direct successors of v_i. We add the clauses

$$\bigwedge_{\substack{1\leq i,j\leq n, e_k\in E(H)\\ i\neq j, v_i\in e_k}} \neg a_{i,j} \vee \neg a_{j,k} \vee \neg o^*_{i,k} \vee a_{i,k} \vee \neg \ell_{i,k}.$$

It remains to check the subset property. As before, we add the clauses

$$\bigwedge_{\substack{1\leq i,j\leq n,\\ 1\leq k\leq m,\\ i\neq j}} \neg a_{i,j} \vee \neg \ell_{i,j} \vee \neg w_{i,k} \vee w_{j,k}. \tag{3.3}$$

This concludes the improved pure encoding. The improvement halves the number of variables and slightly reduces the number of clauses. Next, we discuss our second encoding.

3.4.3 Augmented Encoding

This encoding follows Definition 3.2 for augmented hypertree orderings. This encoding has three parts: (i) definition of $\equiv$, (ii) extending A to $A^{\equiv}$, and (iii) verifying the Special Condition.

The variables $e_{i,j}$ encode the equivalence relation $\equiv$, where $e_{i,j}$ is true if and only if $v_i \equiv v_j$. We define the shorthand $e^*_{i,j}$ such that

$$e^*_{i,j} := \begin{cases} e_{i,j} & \text{if } i < j; \\ e_{j,i} & \text{otherwise.} \end{cases}$$

The following clauses ensure the transitivity of $\equiv$ and property A3: if two vertices are equivalent, they are equivalent to all the vertices on the arc-path between them:

$$\bigwedge_{\substack{1\leq i,j,k\leq n\\ i\neq j\neq k\neq i}} \neg e^*_{i,j} \vee \neg e^*_{j,k} \vee e^*_{i,k}, \qquad \bigwedge_{\substack{1\leq i,j,k\leq n\\ i\neq j\neq k\neq i}} \neg e^*_{i,k} \vee \neg b_{i,j} \vee \neg b_{j,k} \vee e^*_{i,j}.$$

We adapt A to $A^{\equiv}$ next. This requires a change of the clauses in Equation 3.1, allowing backward-arcs within equivalence classes:

$$\bigwedge_{\substack{1\leq i,j\leq n,\\ i\neq j}} e^*_{i,j} \vee \neg o^*_{i,j} \vee \neg a_{j,i}, \qquad \bigwedge_{\substack{\{v_i,v_j\}\in\\ E(G(H))}} \neg o^*_{i,j} \vee a_{i,j}, \qquad \bigwedge_{\substack{\{v_i,v_j\}\in\\ E(G(H))}} \neg o^*_{j,i} \vee a_{j,i}.$$

The backward-arcs within equivalence classes are enforced using the following clauses:

$$\bigwedge_{\substack{1\leq i,j\leq n,\\ i\neq j}} \neg e^*_{i,j} \vee a_{i,j}$$

Verifying the Special Condition is now possible with the following set of clauses:

$$\bigwedge_{\substack{1\leq i,j\leq n, e_k\in E(H)\\ i\neq j, v_i\in e_k}} \neg b_{i,j} \vee \neg w_{j,k} \vee e^*_{i,j}.$$

This completes the augmented encoding.

3.4.4 Encoding Comparison

The correctness of the encodings follows by construction and Theorems 3.2 and 3.3. Obviously, the SAT/MaxSAT/SMT formula $F(H, W)$ can be constructed in polynomial time, given H and W. Hence, we arrive at the following result.

Theorem 3.5. *Given a hypergraph H with n vertices and m hyperedges and an integer W, we can construct in time polynomial in $n + m + W$ a SAT/MaxSAT/SMT formula $F(H, W)$ which is satisfiable if and only if $\mathrm{htw}(H) \leq W$. In case of the MaxSAT encoding, this holds for the hard clauses. Additionally, the model for the optimal solution to the MaxSAT encoding corresponds to a hypertree decomposition of width $\mathrm{htw}(H)$ (see Section 3.4.1).*

Asymptotically, the number of clauses is in $\Theta(n^3 + mn^2)$ for the pure encoding and in $\Theta(n^3+mn)$ for the augmented encoding, while the number of variables is in $\Theta(n^2)$ for both. These asymptotics are similar to other encodings for structural decompositions (Fichte et al., 2018; Samer and Veith, 2009; Schidler and Szeider, 2020). However, the actual constants have a significant influence on performance and scalability of the encoding.

Table 3.2 shows how many variables and clauses are used by the different encodings. On top of the shown numbers, variables and clauses are added to encode the cardinality constraints, which depend on the method chosen. The different hypertree ordering encodings are approximately double the size of the base encoding. The augmented encoding requires the most clauses but is not as dependent on the number of primal graph edges as the pure encoding. How many clauses the pure encoding requires depends

Encoding	#Variables	#Clauses
Base	$mn + \frac{3n^2-3n}{2}$	$2n^3 - 5n^2 + 4n + 2m$
Pure	$2n^2 - 2n$	$2n^3 - 5n^2 + 3n + mn^2 + mn$
Pure Improved	$n^2 - n$	$2n^3 - 6n^2 + 4n + mn^2 + mn$
Augmented	$\frac{3n^2-3n}{2}$	$3n^3 - 7n^2 + 4n + 2mn$

Table 3.2: Number of variables and clauses used in the encodings. The encoding of A^* is not included in the base encoding but in the encodings using A^* since the Pure Improved encoding does not use A^* directly.

strongly on m. Since m is bound by n^2, the encoding can become asymptotically very large for dense graphs but is small for sparse graphs.

Introducing auxiliary variables can circumvent this problem. Instead of $w_{i,e}$, Equations 3.2 and 3.3 use variables $c_{i,j}$, where $c_{i,j}$ is true if and only if $v_j \in \bigcup \lambda_\prec(v_i)$. This requires $\frac{n!}{(n-3)!}$ many clauses instead of $mn \cdot (n-1)$. This method also adds n^2 variables $c_{i,j}$, and $2mn + n^2$ clauses for its definition. The result is an encoding that has better asymptotic bounds for dense graphs and requires slightly more variables and clauses than the augmented encoding.

Next, we discuss our empirical evaluation of these encodings.

3.5 Experiments

Results[2] and code[3] of the experiments are available online.

Experimental Setup

We implemented the encodings presented in Section 3.4 using PySAT 1.6.0[4]. We compared the encoding-based approaches with the state-of-the-art decomposer *new-det-k-decomp*[5] by Fischl et al. (2018a), which implements an XP-algorithm based on dynamic programming over separators.[6]

We used the following solvers. *Optimathsat* 1.7.2[7] (Sebastiani and Trentin, 2020) was used for the SMT encodings, as it performed better than *z3* 4.8.9. We tried the SAT solvers provided by *PySAT* (Ignatiev et al., 2018), as well as the solver *KisSAT* 2.0.1[8] (Biere

[2] https://doi.org/10.5281/zenodo.7271869
[3] https://github.com/ASchidler/htdsmt
[4] https://pysathq.github.io
[5] https://github.com/TUfischl/newdetkdecomp/ (Commit c06232d)
[6] Recently the algorithm has been parallelized in the solver *BalancedGo* (Gottlob et al., 2022). We compare against new-det-k-decomp as our solution is not parallelized.
[7] https://optimathsat.disi.unitn.it
[8] https://fmv.jku.at/kissat/

et al., 2020) for SAT instances. In our experiments, we used *MapleChrono* as it performed best. MaxSAT instances were solved by *MaxHS* 4.0.0[9] (Berg et al., 2020), the winner of the MaxSAT Evaluation 2020[10]. For the SAT encoding, we start at an initial heuristically computed width. This width is then decremented until the formula is unsatisfiable. We used servers with two Intel Xeon E5540 CPUs, each running at 2.53 GHz per core. The servers used Ubuntu 18.04. Each run was limited to six hours and 32 GB RAM. In the results, each non-solved instance is counted as six hours towards the total runtime.

Benchmark Instances

We tested against the full set of Hyperbench[11] instances. Hyperbench consists of instances gathered from various database queries and constraint satisfaction problem instances (Benedikt et al., 2017; Bonifati et al., 2017, 2019; Fischl et al., 2019; Pottinger and Halevy, 2001), for which hypertree width is of practical relevance. The instances of PACE 2019 form a subset of the Hyperbench instances. We removed all disconnected instances, as by Proposition 3.1 we could solve them component-wise, hence their inclusion would those results that are based on the instance size. We ran each configuration against the remaining 3008 instances. 2396 of these instances were solved by any solver configuration, and 1261 were solved by all solver configurations. We present the results for the 2396 solved instances.

3.5.1 Experimental Results

The aim of the experiments was to see how encodings based on different characterizations of hypertree width perform in comparison, and not to determine the fastest exact method for computing the hypertree width.

In Table 3.3, we see a summary of how many instances the different configurations and solvers were able to solve. This summary, grouped by the hypertree width of the instances, is shown in Table 3.4. Further, Figures 3.3 and 3.4 show how runtime and memory usage develop, as the instances become harder to solve.

Encodings

The pure encoding performed slightly better than the augmented encoding in terms of solved instances by a single solver type. The difference is very small, also in terms of uniquely solved instances: using a MaxSAT solver, the pure encoding solved 55 instances that the augmented encoding could not solve. Interestingly, runtime and memory usage behaved almost identically as well.

The augmented encoding was the better encoding if we would use more than one solver type in a portfolio approach. While using an SMT solver for the pure encoding had

[9] https://github.com/fbacchus/MaxHS
[10] https://maxsat-evaluations.github.io
[11] https://hyperbench.dbai.tuwien.ac.at/

Configuration/Solver	# Solved	Time [s]	Unique
Augmented (MAX)	2099 (88%)	8195.7	13
Augmented (SAT)	1866 (78%)	10295.7	4
Augmented (SMT)	2055 (86%)	8593.1	113
Pure (MAX)	2103 (88%)	8159.5	12
Pure (SAT)	1820 (76%)	10710.3	0
Pure (SMT)	1837 (77%)	10557.2	6
New-DetK	1478 (62%)	13793.4	14

Table 3.3: Comparison of different configurations and solvers. *Unique* shows how many instances could be solved by this solver type alone. Time gives the average time required to solve the instances.

Configuration/Solver	1	2	3	4	5	6	>6
	461	223	306	381	439	447	139
Augmented (MAX)	461	217	305	381	428	284	23
Augmented (SAT)	461	217	305	380	425	64	14
Augmented (SMT)	457	215	280	316	347	319	121
Pure (MAX)	461	217	305	381	426	299	14
Pure (SAT)	461	217	305	381	403	52	1
Pure (SMT)	457	211	264	313	334	257	1
New-DetK	461	223	306	381	50	40	17

Table 3.4: Number of solved instances grouped by hypertree width. The second row shows the total number of instances.

little benefit over the MaxSAT solver, it produced interesting results for the augmented encoding. Here, the SMT solver solved 112 additional instances, thereby surpassing the pure encoding in a portfolio approach together with the MaxSAT solver.

The difference in performance might come from how well the two encodings handle high-width instances. Table 3.4 suggests that the augmented encoding handles higher widths better. While the difference is small for MaxSAT, the gap becomes significant for the other solver types. The reason why the pure encoding did slightly better with the MaxSAT solver might stem from the handling of larger graphs, as the asymptotics suggest. Indeed, Figure 3.5 shows that the pure encoding performed slightly better on instances with a large number of vertices.

In general, both encodings were able to solve large instances. As Figure 3.5 shows, there is no order of magnitude, where all instances remained unsolved.

Solver Types

Overall, the MaxSAT solver performed best for both encodings and almost strictly better than the SAT solver. Surprisingly this is also the case in terms of memory consumption. Given that the encoding size is the same for SAT and MaxSAT encodings, we expected a similar memory requirement, but the results show that the MaxSAT solver has significantly lower memory demands. As a consequence, the MaxSAT solver can use memory limits more effectively: the SAT solver quickly switched from moderate memory requirements to exceeding the memory limit, while the MaxSAT solver's memory consumption increased gradually. The SMT solver's performance strongly depended on the encoding. For the pure encoding, it performed almost strictly worse than the MaxSAT solver. For the augmented encoding, it performed slightly worse than the MaxSAT solver. Interestingly, the number of uniquely solved instances is very high. Table 3.4 shows that the SMT solver excelled at instances with high hypertree width. This may be due to different handling of cardinality constraints in the MaxSAT and SMT solver. Furthermore, Figure 3.6 suggests that the SMT solver performs significantly better on large hypergraphs.

New-det-k-decomp

New-detk-decomp solved considerably fewer instances than any of our encodings with any of the different solver types.[12] Table 3.4 suggests that the encodings are better at handling instances with higher widths: new-det-k-decomp solves very few instances with a hypertree width of 5 or higher. This behaviour is not surprising, as the nature of the algorithm requires $\Omega(|V(H)|^{htw})$ time and space. Figures 3.3 and 3.4 show that runtime and memory consumption behaved very similarly to the SAT solver: both requirements are very low for instances of small width until they almost instantly exceed the limits. New-det-k-decomp exhibited this behaviour approximately 400 instances before the SAT solvers do.

3.6 Conclusion

Summary

We presented the first ordering-based characterizations of hypertree width that are purely based on linear elimination orderings. This characterization provides new combinatorial insights into hypertree decompositions. We utilized the characterizations for new SAT/MaxSAT/SMT encodings and tested them on an extensive set of benchmark instances. Indeed, the new encoding clearly outperforms the state-of-the-art combinatorial algorithm for hypertree decompositions. We expect that the new ordering-based

[12] Since a bug in new-det-k-decomp caused errors and wrong results for some instances, we counted each run that finished within the time and memory limit as successful. In total, for 21 instances and 3 uniquely solved instances it is not clear, whether the instances have indeed been solved by new-det-k-decomp.

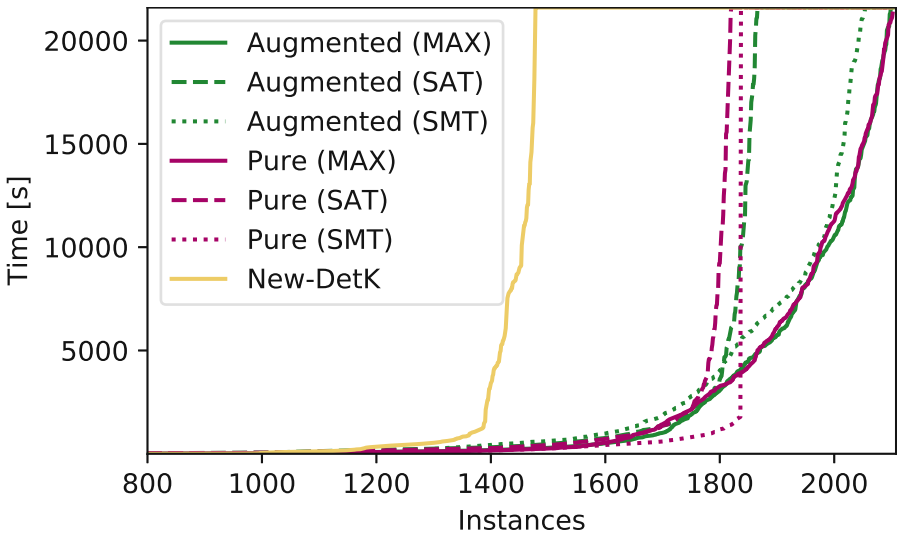


Figure 3.3: A cactus plot comparing the runtime of different solvers. The instances are sorted by runtime for each solver.

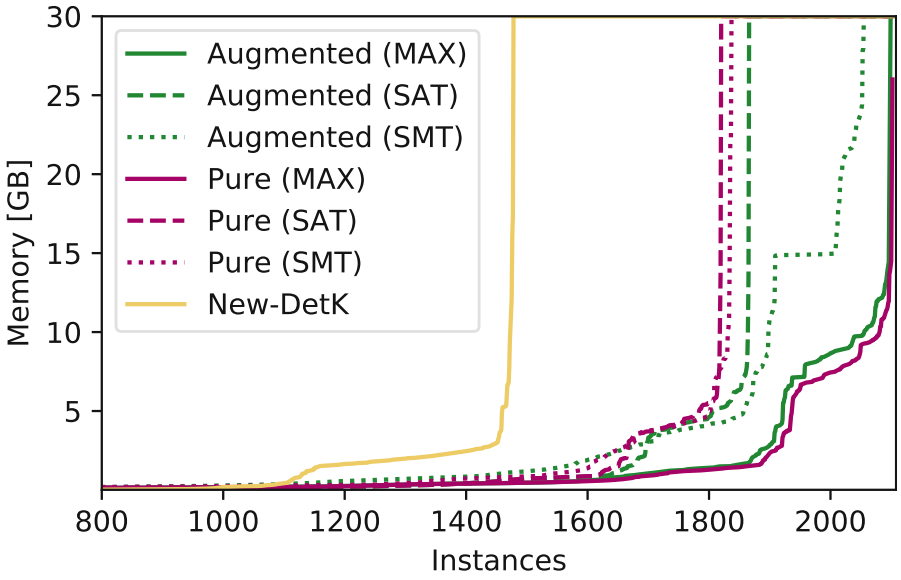


Figure 3.4: A cactus plot comparing the memory usage of different solvers. The instances are sorted by memory usage for each solver.

characterizations of hypertree width will also be of interest outside SAT encodings, for instance, for Branch & Bound algorithms.

Our experimental results show how better scaling can be achieved by looking at multiple characterizations and solver types. We also showed that it is beneficial to look beyond the single-best approach: simply using the combination that solves the most instances

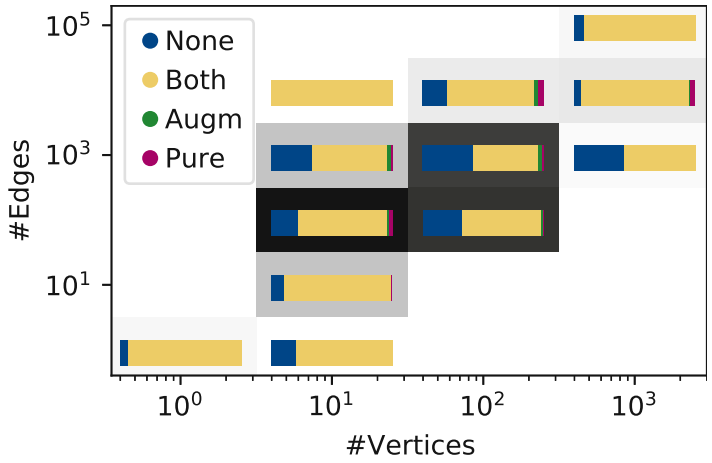


Figure 3.5: Instance sizes and whether the instance could be solved by the Pure or Augmented MaxSAT encoding. The number of edges refers to the primal graph. The bar charts represent the shaded area around them. The degree of shading indicates how many instances of that size were in the instance set, with black being the most.

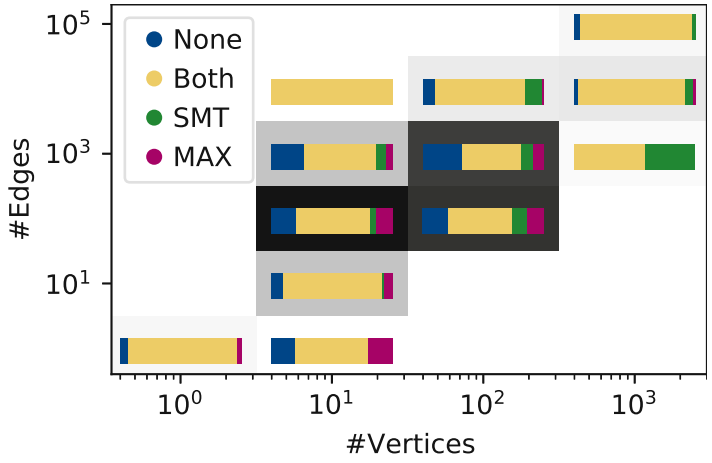


Figure 3.6: Instance sizes and whether Instance sizes and whether the Augmented MaxSAT or SMT encoding could solve the instance. The number of edges refers to the primal graph. The bar charts represent the shaded area around them. The degree of shading indicates how many instances of that size were in the instance set, with black being the most.

would miss out on the complementary nature of the different configurations. Particularly important for dealing with instances of high hypertree width is the SMT solver's capability to manage high upper bounds for the cardinality constraints.

Future Work

We hope that in the future, one can build upon our ordering-based characterizations and encoding, and develop further improvements, including preprocessing and inprocessing techniques, so that optimal hypertree decompositions can be efficiently found for even larger instances.

CHAPTER 4

Twin-Width

4.1 Introduction

Twin-width is a new graph invariant recently introduced by Bonnet et al. (2021a,b, 2022), inspired by previous work by Guillemot and Marx (Guillemot and Marx, 2014). Graph classes of bounded twin-width admit the fixed-parameter tractability of First-Order (FO) model checking, parameterized by the length of the FO formula, provided a witness for bounded twin-width is given. Many NP-hard problems, such as "does the input graph contain an independent set of size at least r?" or "does the input graph contain a subgraph that is isomorphic to a fixed pattern graph H?" can be naturally expressed as FO model checking. Graph classes of bounded twin-width subsume and generalize several dense graph classes for which FO model checking is fixed-parameter tractable, including map graphs, bounded rank-width graphs, bounded clique-width graphs, cographs, and unit interval graphs. Thus, twin-width boundedness plays a similar role for dense graph classes as *nowhere density* plays for sparse graph classes (Grohe et al., 2017).

Bonnet et al.'s (2022) FO model checking algorithm for graphs of bounded twin-width requires a certificate that the input graph's twin-width is bounded by a constant d. There are no practical algorithms known to compute the twin-width of a graph exactly or approximately, and it is known that checking whether a graph's twin-width ≤ 4 is NP-hard (Bergé et al., 2022).

4.1.1 Contributions

In this chapter, we take a SAT-based approach to the exact computation of twin-width. We propose methods for computing lower and upper bounds for d that allow us to reduce the interval of possible values of d for running the SAT solver. Both encodings are based on a new characterization of twin-width in terms of elimination orderings, which are somewhat related to SAT encodings used for other width measures (Ganian et al., 2019;

Samer and Veith, 2009) and our encoding from Chapter 3. However, for twin-width, the situation is more involved because it is not sufficient to globally bound certain static values (like out-degrees in an elimination ordering for treewidth (Samer and Veith, 2009)).

We demonstrate the potentials and limits of our encodings by utilizing them in the following three computational experiments.

1. *Twin-width of small Random Graphs.* We determine experimentally how the twin-width of a random graph depends on its density. As one expects, the twin-width is small for dense and sparse graphs. Graphs of edge-probability 0.5 have the highest twin-width.

2. *Twin-width of Famous Named Graphs.* Over many decades of research in combinatorics, researchers have collected several special graphs, which have been used as counterexamples for conjectures or for showing the tightness of combinatorial results. We considered several such special graphs from the literature and computed their exact twin-width. We believe that these results will be of interest to people working in combinatorics. This way, we have identified a certain class of strongly regular graphs (Paley graphs) that provide high lower bounds for twin-width.

3. *Twin-Width Numbers.* In general, it is not known how many vertices are required to form a graph of a certain twin-width. In fact, there is limited knowledge on lower-bound techniques for twin-width. We use our SAT encoding together with a graph generator to identify the smallest graphs of twin-width $1, 2, 3, 4$, and provide tight bounds for twin-width 5 and 6. This way, we can determine the first few twin-width numbers, where the d-th twin-width number is the smallest number of vertices of a graph with twin-width d. A similar computation has been conducted for clique-width (Heule and Szeider, 2015). Interestingly, up to isomorphism, there are unique smallest graphs of twin-width 1, 2, and 4, respectively, and there are five such graphs for twin-width 3.

4.2 Twin-width

A *trigraph* is an undirected graph G with vertex set $V(G)$ whose edge set $E(G)$ is partitioned into a set $B(G)$ of black edges and a set $R(G)$ of red edges. We consider an ordinary graph as a trigraph with all its edges being black. The set $N_G(v)$ of neighbors of a vertex v in a trigraph G consists of all the vertices adjacent to v by a black or red edge. We call $u \in N_G(v)$ a *black neighbor* of v if $\{u, v\} \in B(G)$, and we call it a *red neighbor* if $uv \in R(G)$. The *red degree* of a vertex $v \in V(G)$ of a trigraph G is the number of its red neighbors. A d-trigraph is a trigraph where each vertex has red degree at most d.

4.2.1 Twin-Width via Sequences of d-Contractions

We give the original definition of twin-width (Bonnet et al., 2021a,b, 2022).

A trigraph G' is obtained from a trigraph G by *contraction*: two (not necessarily adjacent) vertices u and v are merged into a single vertex w, and the edges of G are updated as follows: Every vertex in the *symmetric difference* $N_G(u) \triangle N_G(v)$ is made a red neighbor of w. If a vertex $x \in N_G(u) \cap N_G(v)$ is a black neighbor of both u and v, then w is made a black neighbor of x; otherwise, w is made a red neighbor of x. The other edges (not incident with u or v) remain unchanged.

A *sequence of d-contractions* or *d-sequence* for a graph G is a sequence of d-trigraphs $G_0, G_1, \ldots, G_{n-1}$ where $G_0 = G$, G_{n-1} is the graph on a single vertex, and G_i for $i \geq 1$ is obtained from G_{i-1} by contraction. We observe that $|V(G_i)| = n - i$ for $0 \leq i < n = |V(G)|$. The *twin-width* of a trigraph G, denoted $tww(G)$, is the smallest integer d such that G admits a d-sequence.

It is indeed sometimes necessary to contract non-adjacent vertices. For instance, Figure 4.1 shows a sequence of 2-contractions for the Wagner graph. Without contracting non-adjacent vertices, a vertex of red degree > 2 would be created by the first contraction since each vertex has degree 3 and shares no neighbor with any of its neighbors.

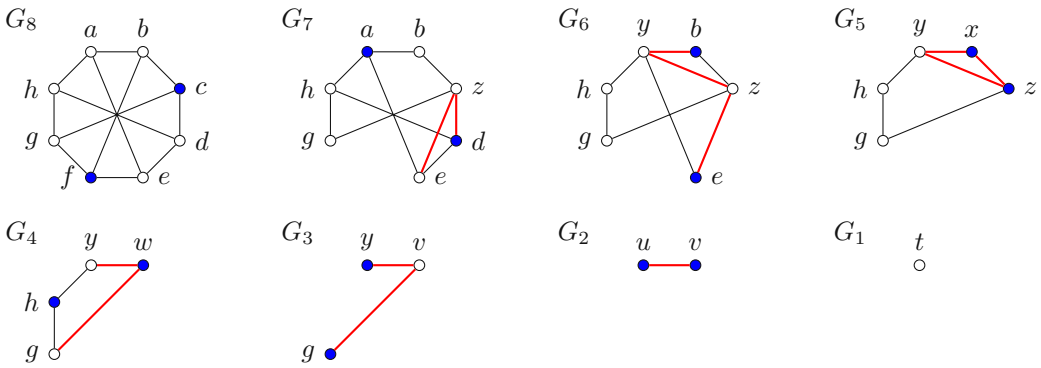


Figure 4.1: A sequence of 2-contractions for the Wagner graph. Vertices that will be contracted next are marked blue.

We state here some basic properties of twin-width observed in the original paper (Bonnet et al., 2022).

Fact 4.1. *If G' is and induced subgraph of a graph G, then $tww(G') \leq tww(G)$.*

For a graph G, we denote by $\overline{G}$ its *complement graph*, which is defined by $V(\overline{G}) = V(G)$ and $E(\overline{G}) = \{\, uv : u, v \in V(G), \{u, v\} \notin E(G), u \neq v \,\}$.

Fact 4.2. *For every graph G, we have $tww(G) = tww(\overline{G})$.*

4.2.2 Twin-Width via d-Elimination Sequences

Next, we give an alternative definition of twin-width which is better suited for formulating our SAT encodings.

Let G be a graph, T a tree with $V(T) = V(G)$, rooted at some vertex r_T, and $\prec$ a linear ordering of $V(T)$, where $u \prec v$ for two vertices $u, v \in V(T)$ such that v is the parent of u in T. We call T a *contraction tree*, $\prec$ an *elimination ordering*, and the pair $(T, \prec)$ a *twin-width decomposition* of G. Thus, when we write $V(G) = \{v_1, \dots, v_n\}$ such that $v_1 \prec \cdots \prec v_n$ and $v_n = r_T$, then T and G define a sequence of graphs $H_0, \dots, H_{n-1}$ with $V(H_i) = \{v_{i+1}, \dots, v_n\}$. We denote by p_i the parent of v_i in T. By definition, $v_i \prec p_i$.

We define the edge set $E(H_i)$ recursively as follows. For $i = 0$, we set $E(H_0) = \emptyset$, and for $1 \leq i < n$, we set

$$
\begin{aligned}
E(H_i) = \{\, \{u, v\} \in E(H_{i-1}) : u, v \in V(H_i) \,\} & \qquad (4.1a)\\
\cup\ \{\, \{u, p_i\} : \{v_i, u\} \in E(H_{i-1}), u \neq p_i \,\} & \qquad (4.1b)\\
\cup\ \{\, \{u, p_i\} : \{v_i, u\} \in E(G), \{p_i, u\} \notin E(G), u \in V(H_i) \,\} & \qquad (4.1c)\\
\cup\ \{\, \{u, p_i\} : \{v_i, u\} \notin E(G), \{p_i, u\} \in E(G), u \in V(H_i) \,\}. & \qquad (4.1d)
\end{aligned}
$$

We call the sequence $H_0, \dots, H_{n-1}$ the *elimination sequence* for G defined by the twin-width decomposition $(T, \prec)$; if for an integer d, all the H_i have a maximum degree $\leq d$, we call $H_0, \dots, H_{n-1}$ a d-elimination sequence. The *width* of the twin-width decomposition $(T, \prec)$ of G is the smallest integer d such that $(T, \prec)$ defines a d-elimination sequence.

Figure 4.2 shows an example of a 2-elimination sequence, and in Figure 4.3 the same elimination sequence is superimposed on the graph.

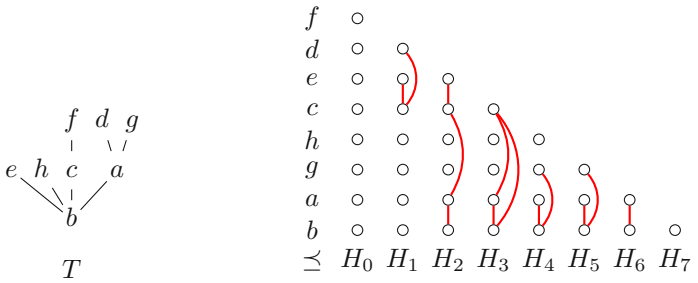


Figure 4.2: A 2-elimination sequence for the Wagner graph, defined by the linear ordering $\prec$ and the contraction tree T. This is the 2-elimination sequence that we get by applying the construction from the proof of Theorem 4.3 to the sequence of 2-contractions shown in Figure 4.1.

Theorem 4.3. *Let G be a graph and $<$ an arbitrary linear ordering of $V(G)$. G has twin-width $\leq d$ if and only if there exists a twin-width decomposition $(T, \prec)$ of width $\leq d$ such that*

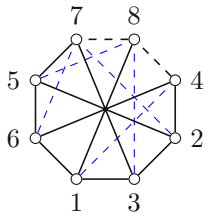


Figure 4.3: The Wagner graph with linear ordering $\prec$ from Figure 4.2 indicated by index numbers. The contraction tree T is superimposed on the graph, where blue dashed edges indicate tree edges that are not shared with the graph, and black dashed edges indicate tree edges that are shared with the graph.

1. *if x is the parent of y in T, then $x < y$;*
2. *the root of T is the $<$-maximal element of $V(G)$.*

Proof. Let G be a graph and assume that $tww(G) \leq d$. By definition, there exists a d-sequence $G_0, G_1, \ldots, G_{n-1}$, and each G_i, $i > 0$, is obtained from G_{i-1} by contracting two vertices u_i and v_i, i.e., merging them into w_i, a new vertex. We slightly change contraction steps. Instead of introducing a new vertex w_i, we reuse one of the two vertices u_i, v_i as w_i. We use the ordering $<$ to decide which of the two vertices to reuse:

$$w_i = \begin{cases} u_i & \text{if } u_i > v_i; \\ v_i & \text{otherwise.} \end{cases} \tag{4.2}$$

This way, we obtain a sequence $G'_0, G'_1, \ldots, G'_{n-1}$, with $V(G'_i) \subseteq V(G)$, where each G'_i is isomorphic to G_i. Since $V(G) = V(G'_0) \supsetneq \cdots \supsetneq V(G'_{n-1})$, this gives us a linear ordering $\prec$ of $V(G)$ in a natural way. We obtain a contraction tree T by taking $V(T) = V(G)$ and $E(T) = \{\, \{u_i, v_i\} : 1 \leq i \leq n-1 \,\}$. Because of (4.2), the contraction tree satisfies the two conditions claimed in the statement of the theorem. A d-elimination sequence $H_0, \ldots, H_{n-1}$ is provided by taking H_i as the subgraph of G'_i formed by its red edges. Thus $(T, \prec)$ is a twin-width decomposition of G of width $\leq d$.

Conversely, assume $(T, \prec)$ is a twin-width decomposition of G of width $\leq d$. Let $H_0, \ldots, H_{n-1}$ be the corresponding d-elimination sequence. We turn the d-elimination sequence into a d-sequence by contracting pairs of vertices as indicated by T. Hence $tww(G) \leq d$. □

4.3 Preprocessing

In this section, we show how to decompose a given graph G in polynomial time into a collection $prime(G)$ of induced subgraphs of G, such that $tww(G) = \max_{H \in prime(G)} tww(H)$. This decomposition can serve as a preprocessing step for twin-width computation.

We require some definitions. A *module* of a graph G is a nonempty set $M \subseteq V(G)$ such that for any $x, y \in M$ and $z \in V(G) \setminus M$ we have $\{x, z\} \in E(G)$ if and only if $\{y, z\} \in E(G)$. A module M is *trivial* if $M = V(G)$ or $|M| = 1$. M is a maximal module if it is not strictly contained in any nontrivial module. A graph is *prime* if all its maximal modules are trivial. For every graph G, there exists a unique partition $P_{\max}$ of $V(G)$ into maximal modules $M_1, \ldots, M_s$, and this partition can be found in linear time (Cournier and Habib, 1994; McConnell and Spinrad, 1994). This partition gives rise to the quotient graph $G/P_{\max}$ whose vertices are the maximal modules of P, and where two modules M_i, M_j, $i \neq j$, are joint by an edge if and only if all the pairs $x_i \in M_i, x_j \in M_j$ are joined by an edge in G. If we select for each module M_i a representative vertex $x_i \in M_i$, then the set $\{x_1, \ldots, x_s\}$ of representatives induces a subgraph of G that is isomorphic to $G/P_{\max}$. If G and its complement graph $\overline{G}$ are connected, then $G/P_{\max}$ is a prime graph (Gallai, 1967; Habib and Paul, 2010). We recursively define the set $\mathit{prime}(G)$ as follows:

1. If G is disconnected, then $\mathit{prime}(G)$ is the union of the sets $\mathit{prime}(C)$ for all connected components C of G.
2. If $\overline{G}$ is disconnected, then $\mathit{prime}(G)$ is the union of the sets $\mathit{prime}(\overline{C})$ for all connected components C of $\overline{G}$.
3. If both G and $\overline{G}$ are connected, then $\mathit{prime}(G)$ is the union of $\{G/P_{\max}\}$ and the sets $\mathit{prime}(G[M])$ for all nontrivial $M \in P_{\max}$.

The three cases above give rise to the *modular decomposition* of the graph G, represented as a rooted tree (Habib and Paul, 2010). The root of the tree is associated with G, the children of each vertex are associated with the connected components (cases 1 and 2), or the maximal modules (case 3) of the graph associated with their parent. The leaves of the tree are in a 1-to-1 correspondence with the vertices of G.

Theorem 4.4. *For every graph G we have $\mathit{tww}(G) = \max_{P \in \mathit{prime}(G)} \mathit{tww}(P)$.*

Proof. Let $d = \max_{P \in \mathit{prime}(G)} \mathit{tww}(P)$. As observed above, $G/P_{\max}$ is isomorphic to an induced subgraph of G; by induction, this holds for all the graphs in $\mathit{prime}(G)$. Because of Fact 4.1, $\mathit{tww}(G) \geq d$ follows.

For showing $\mathit{tww}(P) \leq d$, we proceed by induction on $|V(G)| = n$. The statement is certainly true if $n = 1$ since then $\mathit{prime}(G) = \{G\}$. Now assume $n > 1$. We distinguish several cases.

Consider the case where G is disconnected into components $C_1, \ldots, C_r$. For each $1 \leq i \leq r$ we have $\mathit{prime}(C_i) \subseteq \mathit{prime}(G)$, and so, by induction, we have $\mathit{tww}(C_i) \leq \max_{P \in \mathit{prime}(C_i)} \mathit{tww}(P) \leq d$. Thus, for each C_i there is a d-sequence ending in a single-vertex graph. Using the contractions of these d-sequences, we obtain a d-sequence for G, which ends in an edgeless graph that consists of r isolated vertices.

We can extend this d-sequence by contracting the isolated vertices pairwise in any order, eventually obtaining a single-vertex graph without generating any red edges. Thus $tww(G) \leq d$. The case where $\overline{G}$ is disconnected follows from the previous argument and Fact 4.2.

Finally, assume that G and $\overline{G}$ are connected. Thus $G/P_{\max}$ is prime and is isomorphic to an induced subgraph $G' \in prime(G)$ of G. For each $M \in P_{\max}$, $prime(G[M]) \subseteq prime(G)$. By induction hypothesis, $tww(G') \leq d$ and $tww(G[M]) \leq d$. We thus obtain a d-sequence for G by putting together d-sequences for $G[M]$, $M \in P_{\max}$, and a d-sequence for G', which contract first each $G[M]$ on a single vertex of G', and then contract G' on a single vertex. Hence, $tww(G) \leq d$. □

Theorem 4.4 provides the basis for a preprocessing phase for twin-width computation. If the given graph G is not prime, we compute $prime(G)$ and determine the twin-width of all the graphs in $prime(G)$. Since for a non-prime graph G, the graphs in $prime(G)$ are smaller than G, it is more efficient to run a costly twin-width algorithm on the graphs in $prime(G)$ than on G itself. Hence, this preprocessing can be highly beneficial for non-prime graphs.

4.4 Encodings

In this section, we present two SAT encodings for twin-width. Assume we are given a graph G with vertices $v_1 \dots v_n$ and an integer d. We will define a propositional formula $F(G, d)$ in Conjunctive Normal Form (CNF) that is satisfiable if and only if $tww(G) \leq d$. For the construction of $F(G, d)$, we use the characterization of twin-width in terms of a twin-width decomposition $(T, \prec)$, as established in Theorem 4.3. We use the indices $1 \leq i, j, k, m \leq n$ and subsequently omit the upper and lower bounds for readability. Furthermore, we use the mapping $\varphi_{\prec}(v_i)$ to denote the position of v_i in $\prec$.

We give two different encodings for $F(G, d)$ that differ in how we encode $\prec$. In the *relative encoding* F_{rel}, we encode for all mutually distinct pairs v_i, v_j if $v_i \prec v_j$, while in the *absolute encoding* F_{abs} we encode $\varphi_{\prec}(v_i)$ for all v_i. The absolute encoding in this thesis differs from the one we originally proposed (Schidler and Szeider, 2022), and performs much better than the original version. We first introduce our two encodings and discuss the differences in Section 4.6.

4.4.1 Relative Encoding

In our first encoding, we use a relative ordering of the vertices, as used in the treewidth encoding by Samer and Veith (2009): instead of encoding $\varphi_{\prec}(v_i)$ directly, we encode for vertices $v_i, v_j \in V(G)$, whether $\varphi_{\prec}(v_i) < \varphi_{\prec}(v_j)$ or not. Table 4.1 shows the variables utilized in the encoding. For the ordering, we use $\binom{n}{2}$ variables $o_{i,j}$ with $i < j$, where $o_{i,j}$ is true if and only if $v_i \prec v_j$. We subsequently use the shorthand $o^*_{i,j}$ where $o^*_{i,j}$ is $o_{i,j}$ if $i < j$ and $\neg o_{j,i}$ if $i > j$. We encode the semantics by enforcing transitivity: for mutually

Name	Range	Meaning
$a_{i,j}$	$1 \leq i < j \leq n$	$\{v_i, v_j\} \in E_k$ for some k
$o_{i,j}$	$1 \leq i < j \leq n$	$v_i \prec v_j$
$p_{i,j}$	$1 \leq i < j \leq n$	$p_i = v_j$
$r_{i,j,k}$	$1 \leq i, j \leq n$ and $j < k \leq n$	$\{v_j, v_k\} \in E(H_{\varphi_\prec(v_i)})$ after eliminating v_i

Table 4.1: The variables used in the relative encoding.

distinct $1 \leq i, j, k \leq n$ we add the clauses

$$\neg o^*_{i,j} \vee \neg o^*_{j,k} \vee o^*_{i,k}.$$

Next, we encode the contraction tree T. In view of Theorem 4.3, we can assume that when p_i is the parent of p_j in T, then $i < j$ (Condition 1), and v_n is the root of T (Condition 2). Hence, we can use $\binom{n}{2}$ variables $p_{i,j}$ with $i < j$, where $p_{i,j}$ is true if and only if $p_i = v_j$. We encode that every vertex, except the root, has exactly one parent. For that, we utilize at-least-one constraints by adding for each $1 \leq i < n$ the clause $\bigvee_{i<j} p_{i,j}$ and at-most-one constraints by adding for mutually distinct $1 \leq i, j, k \leq n$ the clauses $\neg p_{i,j} \vee \neg p_{i,k}$. Additionally, we ensure that $v_i \prec v_j$ holds between a vertex v_i and its parent v_j, by adding for $1 \leq i < j \leq n$ the clauses

$$\neg p_{i,j} \vee o^*_{i,j}.$$

So far we have encoded $\prec$ and T. Next, we encode the elimination sequence $H_0, \ldots, H_n$ with two additional sets of variables. We take $n\binom{n}{2}$ variables $r_{i,j,k}$ with $j < k$, where $r_{i,j,k}$ is true if and only if after eliminating v_i it holds that $\{v_j, v_k\} \in E(H_{\varphi_\prec(v_i)})$. We also use $\binom{n}{2}$ auxiliary variables $a_{i,j}$ with $i < j$, where $a_{i,j}$ is true if and only if there exists a k such that $\{v_i, v_j\} \in E(H_k)$. We use shorthands a^* and r^* which are defined analogously to o^*.

We encode the semantics of a by adding, for all mutually distinct $1 \leq i, j, k \leq n$ with $i < j$, the clauses

$$\neg o^*_{i,j} \vee \neg o^*_{i,k} \vee \neg r^*_{i,j,k} \vee a^*_{j,k}.$$

Furthermore, we encode the semantics of r by encoding Subsets (4.1a)–(4.1d) of $E(H_i)$ according to the definition given in Section 4.2. Subsets (4.1c) and (4.1d) are encoded by adding for $1 \leq i < j \leq n$ and $v_k \in (N_G(v_i) \triangle N_G(v_j)) \setminus \{v_i, v_j\}$ the clause

$$\neg p_{i,j} \vee \neg o^*_{i,k} \vee r^*_{i,j,k}.$$

Further, Subset (4.1b) is encoded by adding, for mutually distinct $1 \leq i, j, k \leq n$, with $i < j$, the clauses

$$\neg p_{i,j} \vee \neg o^*_{i,k} \vee \neg a^*_{i,k} \vee r^*_{i,j,k}.$$

Name	Range	Meaning
$o'_{t,i}$	$1 \leq t < n-d-1$	$\varphi_{\prec}(v_i) = t$
$p'_{t,j}$	$1 \leq t < n-d-1, 1 \leq i \leq n$	$p_{\varphi_{\prec}^{-1}(t)} = v_j$
$r'_{t,i,j}$	$1 \leq t < n-d-1, 1 \leq i < j \leq n$	$\{v_i, v_j\} \in E(H_t)$
$e_{t,i}$	$1 \leq t < n-d-1, 1 \leq i \leq n$	$\{v_{\varphi_{\prec}^{-1}(t)}, v_j\} \in E(H_{t-1})$

Table 4.2: The variables used in the absolute encoding.

Finally, we encode Subset (4.1a) by adding for mutually distinct $1 \leq i, j, k, m \leq n$, $k < m$ the clauses

$$\neg o^*_{i,j} \vee \neg o^*_{j,k} \vee \neg o^*_{j,m} \vee \neg r^*_{i,k,m} \vee r^*_{j,k,m}.$$

The $O(n^4)$ clauses required to encode the Subset (4.1a) dominate the size of the encoding. Unfortunately, this is unavoidable: without knowing $\varphi_{\prec}(.)$, we have $O(n^2)$ possible orderings of v_i, v_j, and for each such ordering, we have $O(n^2)$ possible edges $v_k v_m$.

We enforce the upper bound d using cardinality constraints as discussed in Section 2.3.1. For each vertex v_i, we limit the set $\{ r^*_{i,j,k} : 1 \leq j, k \leq n, i \neq j \neq k \neq i \}$ to at most d true values.

4.4.2 Absolute Encoding

The main idea behind our second encoding is to directly encode the absolute position of each vertex in $\prec$. The resulting encoding's size is in $O(n^3)$, significantly improving upon the bound of the relative encoding. We use the index t with $1 \leq t < n-d-1$ to denote the current position in $\prec$. The limit of $n-d-1$ comes from the fact that a graph with $d+1$ vertices can have at most tww d, and we can eliminate the last $d+1$ vertices in any order. We first propose the encoding and then discuss the differences between the encodings.

We use $n \cdot (n-d-2)$ variables $o'_{t,i}$, where $o'_{t,i}$ is true if and only if $\varphi_{\prec}(v_i) = t$. For each t, we require one at-most-one and one at-least-one constraint over the variables $\{ o'_{t,i} : 1 \leq i \leq n \}$. Further, for each i, we require an at-most-one constraint over the variables $\{ o'_{t,i} : 1 \leq t < n-d-1 \}$.

The trick for a more succinct encoding is the representation of T. Instead of using $p_{i,j}$ to encode T, we use $n \cdot (n-d-2)$ variables $p'_{t,j}$ with the semantics that $p'_{t,j}$ is true if and only if v_j is the parent of vertex $\varphi_{\prec}^{-1}(t)$. For each t we limit the set $\{ p'_{t,i} : 1 \leq i \leq n \}$ with one at-most-one and one at-least-one constraint. Further, for each i, we limit the set $\{ p'_{t,i} : 1 \leq t < n-d-1 \}$ with an at-most-one constraint. These constraints ensure that each eliminated vertex has exactly one parent in T. Additionally, we ensure that if $p'_{t,j}$ is true, it holds that $\varphi_{\prec}(v_j) > t$, i.e., that the parent is never an eliminated vertex, by adding for all $1 \leq i \leq n$, $1 \leq t' \leq t < n-d-1$ the clauses

$$\neg o'_{t',i} \vee \neg p'_{t,i}.$$

We conclude the encoding of T by enforcing that the parent is lexicographically larger than the child by adding for all t and j the clauses

$$\neg p'_{t,j} \vee \bigvee_{i<j} o'_{t,i}.$$

Next, we encode the elimination sequence $H_0, \ldots, H_{n-d-2}$. We start by introducing $(n-d-2) \cdot \binom{n}{2}$ variables $r'_{t,i,j}$ with $i < j$, where $r'_{t,i,j}$ is true if and only if $\{v_i, v_j\} \in E(H_t)$. We use the shorthand r'^* similarly to r^* in the relative encoding.

In the previous section, encoding Subset (4.1a) was the main contributor to the encoding's size. This subset can be encoded more succinctly in the absolute encoding, as we can add for all $1 < t < n-d-1$ and $1 \leq i < j \leq n$ the clauses

$$o'_{t,i} \vee o'_{t,j} \vee \neg r'^*_{t-1,i,j} \vee r'^*_{t,i,j},$$

which only requires a number of clauses in $O(n^3)$.

The critical part of the absolute encoding is Subset (4.1b), as a straightforward encoding would be quartic. We avoid this using the different representation of T and introducing $(n-d-2) \cdot n$ variables $e_{t,i}$, where $e_{t,i}$ is true if and only if $\{\varphi_\prec^{-1}(t), v_i\} \in H_{t-1}$. In other words, $e_{t,i}$ is true if and only if for H_t the edge to v_i is in Subset (4.1a). These semantics are enforced by adding for all $1 < t < n-d-1$ and distinct $1 \leq i, j \leq n$ the clauses

$$\neg o'_{t,i} \vee \neg r'^*_{t-1,i,j} \vee e_{t,j}.$$

We can then encode Subset (4.1b) succinctly by adding for all $1 < t < n-d-1$ and distinct $1 \leq i, j \leq n$ the clauses

$$\neg p'_{t,i} \vee \neg e_{t,j} \vee r'^*_{t,i,j}.$$

The remaining clauses to encode the elimination sequence are similar to the relative encoding. Subsets (4.1c) and (4.1d) are encoded by adding for each $1 \leq t < n-d-1$, $1 \leq i < j \leq n$, and $v_k \in (N_G(v_i) \triangle N_G(v_j)) \setminus \{v_i, v_j\}$ the clause

$$\neg o'_{t,i} \vee \neg p'_{t,j} \vee r'^*_{t,j,k} \vee \bigvee_{t' \leq t} o'_{t',k}.$$

Finally, we use cardinality constraints for r' as in the relative encoding to enforce that the width is at most d. The resulting encoding's number of clauses is then in $O(n^3)$.

Towards Better Reasoning

The absolute encoding is much more succinct than the relative encoding in terms of the number of clauses. This is shown by the asymptotics, but also practically, as we discuss in Section 4.6.

The reduced number of clauses comes at the price of a more indirect encoding. Splitting the variables $p_{i,j}$ into $o'_{t,i}$ and $p'_{t,j}$ means that it is hard for the solver to learn about the structure of T. Further, the variables $o'_{t,i}$ encode specific positions in the ordering, further limiting the information gain from conflicts. In practice, we observed that showing unsatisfiability often takes very long using the absolute encoding as stated above.

We can improve the encoding with two changes: we reintroduce the $p_{i,j}$ variables on top of the absolute encoding and use the at-most-one constraints to get something similar to reasoning about relative positions. While adding $p_{i,j}$ increases the number of both variables and clauses, it improves the capabilities to reason about the structure of T.

We encode the semantics of $p_{i,j}$ using the absolute encoding's variables. We make the values of $p'_{t,i}$ and $p_{i,j}$ consistent with each other by adding for all $1 \leq t < n - d - 1$ and $1 \leq i < j \leq n$ the clauses

$$\neg o'_{t,i} \vee \neg p'_{t,j} \vee p_{i,j}.$$

The semantics of $p_{i,j}$ is then enforced by constraining for each i the cardinality of $\{ p_{i,j} : i < j \}$ with an at-most-one constraint, ensuring that each vertex has at most one parent.

We slightly change the cardinality constraints for $o'_{t,i}$. We do not introduce additional constraints but specify that the at-most-one constraints for a given i on the cardinality of $\{ o'_{t,i} : 1 \leq t < n-d-1 \}$ is encoded using sequence counters (Sinz, 2005). This introduces a new variable $c_{t,i}$ for each $o'_{t,i}$ with the semantics that $c_{t,i}$ is true if $\varphi_{\prec}(v_i) \leq t$. We strengthen the "if" to "if and only if" by adding for all $1 \leq t < n - d - 1$ and $1 \leq i \leq n$ the clauses

$$\neg c_{t,i} \vee \bigvee_{t' \leq t} o'_{t'i}.$$

The new variables allow us to reason in a more relative way about the position of vertices. Instead of stating that a vertex is eliminated at position t, we can now state a vertex is eliminated before at position t.

We use the variables $c_{t,i}$ to encode the property that $p'_{t,i}$ implies $\varphi_{\prec}(v_i) > t$ more succinctly by using for all $1 \leq t < n - d - 1$ and $1 \leq i \leq n$ the clauses

$$\neg c_{t,i} \vee \neg p'_{t,i}.$$

The biggest change to the observed behaviour of the encoding comes from encoding Subsets (4.1c) and (4.1d) using the newly introduced variables. We use for each $1 \leq t \leq n - d - 1$, $1 \leq i < j \leq n$, and $v_k \in (N_G(v_i) \triangle N_G(v_j)) \setminus \{v_i, v_j\}$ the clause

$$\neg c_{t,i} \vee c_{t,j} \vee c_{t,k} \vee \neg p_{i,j} \vee r'^{*}_{t,j,k}.$$

Iterative Absolute Encoding

The absolute encoding allows for a straightforward iterative use. We modify the encoding to a formula $F_{\text{abs}}(G, d, s)$, with $s \leq n - d - 1$, which is satisfiable if and only if there

exists an elimination sequence $H_0, \ldots, H_{s-1}$ for G such that all H_i with $1 \leq i < s$ have red degree $\leq$ d. Hence, we limit the length of the elimination sequence by limiting t to s instead of to $n - d - 1$. For small values of s, this method severely reduces the number of clauses.

The incremental encoding is originally $F_{\text{abs}}(G, d, 1)$; when the formula is satisfiable, we add the clauses for $F_{\text{abs}}(G, d, 2)$ and so forth, until $s = n - d - 1$. Whenever the formula is unsatisfiable, we increment d. This incremental approach is useful to lower the time required for unsatisfiable calls, i.e., when $d < \mathit{tww}(G)$. Here, one can often establish that $\mathit{tww}(G) \geq d$ with only a few contractions and, therefore, with a much smaller encoding. E.g., for a 6×6 grid graph it takes 32 contractions to establish $\mathit{tww}(G) \leq 3$, but it takes only $F_{\text{abs}}(G, 2, 9)$ to verify that $\mathit{tww}(G) > 2$.

Since the constructions of $F_{\text{rel}}(G, d)$ and $F_{\text{abs}}(G, d)$ closely follow the definitions given in Section 4.2, we have the following result.

Theorem 4.5. *Given a graph G with n vertices and an integer d, we can construct in time polynomial in $n \cdot d$ a propositional formula $F(G, d)$, which is satisfiable if and only if $\mathit{tww}(G) \leq d$.*

This concludes the introduction of our encodings. We discuss the empirical differences between our encodings in Section 4.6.

4.5 Lower and Upper Bounds

This section describes a simple approach for deriving lower and upper bounds for the twin-width of graphs. We use these bounds for limiting the range for d when running the SAT solver on $F(G, d)$.

We first discuss the lower bound. Let r be a positive integer and G a graph with at least r vertices. We define the *lower bound lb_r of order r* for $\mathit{tww}(G)$ as the maximum degree of the first $r + 1$ graphs $H_0, \ldots, H_{r-1}$ of any elimination sequence for G. In particular, for $r = 1$ we have

$$\mathit{lb}_1(G) = \min_{u,v \in V(G), u \neq v} |N_G(u) \triangle N_G(v)|.$$

Clearly, $\mathit{lb}_1(G) \leq \mathit{lb}_2(G) \leq \cdots \leq \mathit{lb}_n(G) = \mathit{tww}(G)$. If r is a constant, then $\mathit{lb}_r(G)$ can be computed in polynomial time.

For obtaining an upper bound on the twin-width of a given graph G, we propose a simple greedy algorithm. The algorithm computes an elimination ordering $\prec$ and a contraction tree T step-by-step, greedily choosing the next vertex v_i in the ordering. Assume we have already computed the first i vertices of the elimination ordering $v_1, \ldots, v_{i-1}$ and the corresponding sequence of graphs $H_0, \ldots, H_{i-1}$ with $V(H_{i-1}) = \{v_i, \ldots, v_n\}$. We choose the next vertex $v_i \in V(H_{i-1})$ and the corresponding parent $p_i \in \{v_{i+1}, \ldots, v_n\}$, $p_i > v_i$ in the lexicographic ordering of the vertices, such that the degree of p_i in H_i is minimized;

in case of a tie, we take the lexicographically minimal pair (v_i, p_i). We add the edge $\{v_i, p_i\}$ to the contraction tree. The width of the resulting twin-width decomposition $(T, \prec)$ gives the upper bound ub_{greedy} on the twin-width of G. Our implementation of the greedy heuristic uses caching to avoid computing the degree of potential pairs (v_i, p_i) over and over again.

4.6 Experiments

We computed the twin-width of several graphs using our encodings[1]. We implemented and ran the encodings using Python 3.8.0 and PySAT 1.6.0 (Ignatiev et al., 2018)[2].

We ran the large-scale experiments using random instances on servers with two Intel Xeon E5-2640 v4 CPUs—10-cores running at 2.40 GHz—with 160 GB memory and using Cadical[3] as a SAT solver. We used Cadical as it outperformed the other SAT solvers implemented in PySAT by far.

The experiments on named graphs were run on a computer with an Intel Core i5-9600KF CPU with six cores running at 3.70 GHz, 32 GB RAM and using Kissat, an improved rewrite of Cadical (Biere et al., 2020)[4].

4.6.1 Named Graphs

We computed the twin-width of several named graphs, which are well-known from the literature (Weisstein, 2021). The names of the graphs either reflect their topology or their discoverer. For most of the considered graphs, the twin-width was not known. Table 4.3 provides an overview of our results, including lower and upper bounds as described in Section 4.5. Preprocessing has no effect on the named graphs, which all turned out to be prime (as one would expect, as these graphs often provide a smallest example or counterexample for a combinatorial property).

Interestingly, the lower bound lb_1 often coincides with the exact twin-width. One possible explanation is the high level of symmetry in many of the graphs. A particularly interesting class of symmetric graphs are the *strongly regular graphs*: these graphs are usually parameterized by the tuple (n, k, λ, μ), where n is the number of vertices, k is the degree of each vertex, and every pair of vertices has either λ common neighbors if they are adjacent, or share μ neighbors otherwise. For a strongly regular graph G with parameters (n, k, λ, μ), we can immediately determine the lower bound of order 1

$$lb_1(G) = \min\{2(k-\mu), 2(k-\lambda-1)\}.$$

[1] Source code can be found at https://github.com/ASchidler/twin_width. The results can be found at https://doi.org/10.5281/zenodo.7271869.

[2] https://pysathq.github.io

[3] http://fmv.jku.at/cadical/

[4] http://fmv.jku.at/kissat/

Graph	$\|V\|$	$\|E\|$	lb_1	tww	ub_{greedy}
Balaban 10-Cage*	70	105	4	≤ 5	6
Brinkmann	21	42	6	6	6
Chvátal	12	24	2	3	5
Clebsch	16	40	6	6	8
Desargues	20	30	4	4	5
Dodecahedron	20	30	4	4	4
Dürer	12	18	2	3	4
Ellingham*	78	117	4	4	5
Errera	17	45	4	5	6
FlowerSnark	20	30	4	4	4
Folkman	20	40	2	3	3
Franklin	12	18	2	2	4
Frucht	12	18	2	3	3
Goldner	11	27	2	2	4
Grid 6×8*	48	82	2	3	4
Grötzsch	11	20	2	3	5
Herschel	11	18	2	2	4
Hoffman	16	32	2	4	5
Holt	27	54	6	6	7
Kittell	23	63	4	5	6
McGee	24	36	4	4	5
Moser	7	11	2	2	2
Nauru	24	36	4	4	5
Paley-73*	73	1314	36	36	64
Pappus	18	27	4	4	5
Peterson	10	15	4	4	4
Poussin	15	39	3	4	5
Robertson	19	38	6	6	6
Rook 6×6*	36	180	10	10	12
Shrikhande	16	48	6	6	8
Sousselier	16	27	4	4	5
Tietze	12	18	2	4	4
Wagner	8	12	2	2	2
Watsin*	50	75	4	4	5

Table 4.3: Results for famous named graphs. For all graph not marked with *, the twin-width could be computed in at most five seconds. lb_1 gives the lower bound of order 1, ub_{greedy} gives the width of an elimination ordering computed by the greedy algorithm of Section 4.5.

Examples of strongly regular graphs in Table 4.3 are *Clebsch* $(16, 5, 0, 2)$, *Peter-*

son $(10, 3, 0, 1)$, *Rook* $n \times n$ $(n^2, 2n-2, n-2, 2)$, and *Shrikhande* $(16, 6, 2, 2)$. A family of strongly regular graphs, the *Paley graphs*, stick out due to their high twin-width in relation to their size. For every prime power n such that $n \equiv 1 \pmod 4$, the Paley graph on n vertices (Paley-n) is defined and is strongly regular with parameters $k=(n-1)/2$, $\lambda=(n-5)/4$, $\mu=(n-1)/4$. Further, Paley graphs are *self-complementary*, i.e., Paley-n and $\overline{\text{Paley-}n}$ are isomorphic (Godsil and Royle, 2001). With our relative SAT encoding, we could verify that for Paley graphs with up to 73 vertices, the lower bound of order 1 gives the exact twin-width. We originally conjectured (Schidler and Szeider, 2022) that $tww(\text{Paley-}n) = (n-1)/2$ holds in general. This has later been independently proven by Ahn et al. (2021). It is still an open question, whether there are graphs of the same size that have higher twin-width, or in the contrary, Paley graphs define an upper bound on the twin-width, given a specific number of vertices.

The twin-width of two-dimensional grid graphs is also interesting. They are known to have unbounded treewidth and clique-width, but it is easy to see that their twin-width is at most 4 (Bergé et al., 2022). Interestingly, with our encodings, we found that smaller grid graphs, of size up to 8×6, do have twin-width 3. We see it as an interesting challenge to determine the exact twin-width of all square grids. The 3-elimination sequence that we found with our encodings do not suggest any obvious general pattern that could be generalized to all grid graphs, hence we still expect that at a certain size the width switches from 3 to 4.

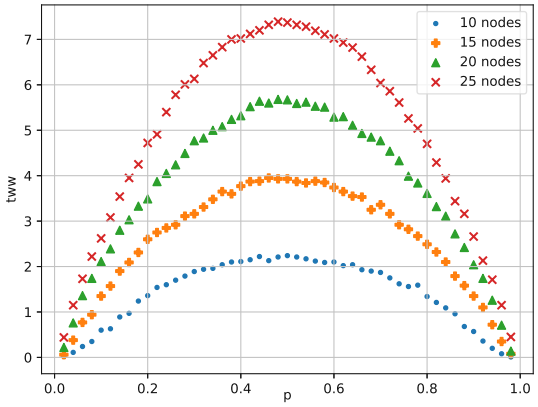


Figure 4.4: Twin-width for randomly generated graphs: each edge exists with probability p. Each point is the average over 100 graphs.

4.6.2 Random Graphs

We tested the twin-width on randomly generated graphs. For this purpose, we created Erdős-Rény graphs $G(n, p)$, where $|V(G)| = n \in \{10, 15, 20\}$ and each edge exists with probability p, where p takes values between 0 to 1 in 0.02 increments.

The results in Figure 4.4 show that the twin-width increases quickly with increasing graph size. Furthermore, the vertical distance between the peaks is similar. The symmetric

shape is expected due to Fact 4.2.

Many of the graphs can be simplified using the preprocessing discussed in Section 4.3.

4.6.3 The Twin-Width Numbers

For every $d > 0$, let tww_d be the smallest integer such that there exists a graph with tww_d many vertices of twin-width d. We call tww_d the *d-th twin-width* number. In contrast to other width measures like treewidth, where similar numbers are easy to compute (the d-th treewidth number is $d + 1$), no uniform method is known for computing the twin-width numbers. The situation is similar for clique-width, where no uniform method is known either; Heule and Szeider (2015) computed the first few clique-width numbers.

The computation of twin-width numbers provides a challenge for any exact method, as the search space grows quickly with each increment of d. However, with our encodings, run on prime graphs generated by Nauty[5] (McKay and Piperno, 2014), we were able to identify the first few twin-width numbers and give tight bounds for further ones.

Proposition 4.6. *The sequence of twin-width numbers starts with* $4, 5, 8, 9$*; the fifth twin-width number is* 11 *or* 12*; the sixth twin-width number is at most* 13*.*

For computing the twin-width numbers, we only need to consider graphs G with $|E(G)| \leq \binom{n}{2}/2$, as by Fact 4.2, $|E(G)| > \binom{n}{2}/2$ implies $|E(\overline{G})| \leq \binom{n}{2}/2$. Further, according to Theorem 4.4, we only need to consider prime graphs. In particular, since every prime graph G and its complement graph $\overline{G}$ are connected, we only need to consider connected graphs. The results are shown in Table 4.4.

The preprocessing described in Section 4.3 can be used for all graphs that are not prime. We can see in Table 4.4 that there are many connected graphs that are not prime, and thereby eligible for preprocessing.

Interestingly, for the first, second, and fourth twin-width number tww_d, there is a unique graph, up to isomorphism, with tww_d many vertices and twin-width d. For the third twin-width number, there are five such graphs: $G_{8,3,i}, i = 1, \ldots, 5$. $G_{8,3,3}$ is self-complementary; the other four form two complementary pairs. In Figure 4.5, we display these graphs together with an optimal d-sequence, showing only one graph from each complementary pair.

The unique graph certifying $tww_1 = 4$ is the path on 4 vertices (P_4). The unique graph certifying $tww_1 = 5$ is the cycle on five vertices (C_5). The unique graph certifying $tww_4 = 9$ is the graph Paley-9 (see Section 4.6.1). In fact, $C_5 =$ Paley-5, so also tww_2 is certified by a Paley graph. Further, if we remove any vertex from Paley-9, we obtain $G_{8,3,3}$. Similarly, we obtain P_4 by removing a vertex from Paley-5. Therefore, Paley graphs are related to all of the first four twin-width numbers. We could establish with our method that among all graphs with 10 vertices, there is no graph of twin-width

[5] http://cs.anu.edu.au/people/bdm/

5, hence $tww_5 \geq 11$. We could not check all graphs with 11 vertices, as there are too many. Paley-13 shows that $tww_6 \leq 13$. By deleting any single vertex from Paley-13, its twin-width drops to 5. This implies that $tww_5 \leq 12$, and so $11 \leq tww_5 \leq 12$ as stated in Proposition 4.6.

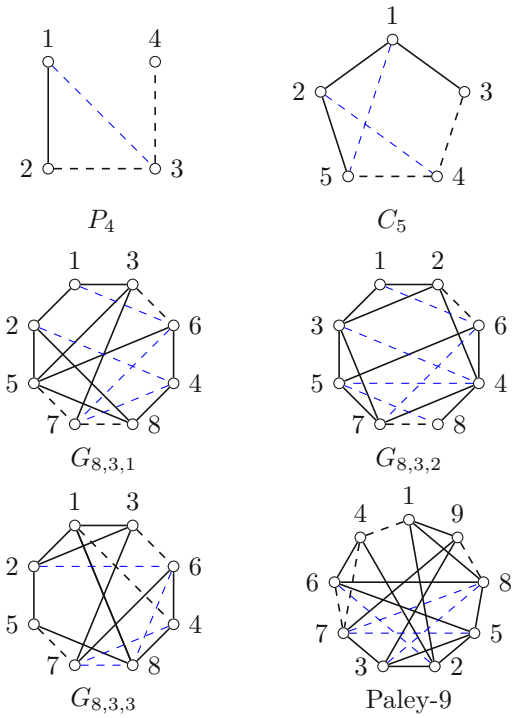


Figure 4.5: Smallest graphs for given twin-width d. The integer vertex labels give a d-sequence, and the dashed edges give a contraction tree, as in Figure 4.3.

			twin-width			
$\|V\|$	connected	prime	1	2	3	4
4	3	1	1	0	0	0
5	11	4	3	1	0	0
6	73	26	16	10	0	0
7	618	260	90	170	0	0
8	8573	4670	655	4010	5	0
9	224875	145870	4488	137565	3816	1
10	11716571	8110356	30318	6144756	1935226	56

Table 4.4: The number of graphs, prime graphs, and prime graphs of a specific twin-width, with a specific number of vertices.

4.6.4 Encoding Comparison

In these experiments, we compare our two encodings. Table 4.5 shows a comparison based the named graph where determining the twin-width is hard. The results show the runtimes for running both $F(G, tww)$ (SAT) and $F(G, tww - 1)$ (UNSAT).

Graph	|V|	*tww*	Encoding	SAT Vars	SAT Clauses	SAT Time	UNSAT Time
Balaban 10-Cage[6]	70	≤ 5	Relative	1.2 m	15 m	>43200 s	>43200 s
			Absolute	1.2 m	4.6 m	1600 s	>43200 s
Ellingham	78	4	Relative	1.7 m	22 m	958 s	153 s
			Absolute	1.6 m	6.1 m	128 s	4 s
Grid 8×6	48	3	Relative	371 k	3.4 m	3690 s	3 s
			Absolute	348 k	1.3 m	160 s	7 s
Paley-73	73	36	Relative	2.4 m	28 m	159 s	880 s
			Absolute	1.2 m	10 m	262 s	12 s

Table 4.5: Comparison of the two twin-width encodings on difficult famous graph instances. m denotes million and k thousand.

As expected, the number of clauses for the absolute encoding is much smaller than for the relative encoding, even for small graphs. The story is different for the number of variables. For small d, when $|V| - d - 1$ is close to $|V|$, the absolute encoding uses a similar number of variables compared to the relative encoding. This behaviour changes with a higher upper bound, as for the Paley-73 graph, where the twin-width is almost $\frac{|V|}{2}$. Here, the absolute encoding uses significantly fewer variables.

Comparing the encoding sizes based on different graphs reveals additional details. The Ellingham and Balaban 10-Cage graphs have small twin-width, while the Paley-73 graph has a very high twin-width. All three graphs have roughly the same size. The relative encoding's size and number of variables increase significantly with higher twin-widths, due to the cardinality constraints requiring more variables and clauses. This is confirmed by looking at $F_{\text{rel}}(\text{Paley-73}, 4)$, where the number of variables drops to 1.4 million and the number of clauses to 17 million. The still much higher number of clauses compared to the other two graphs is explained by the Paley-73 graph's high density, which requires more clauses to encode Subsets (4.1c) and (4.1d).

The absolute encoding behaves differently. Since a higher upper bound means that the encoded elimination sequence is shorter, the number of variables does not increase. Table 4.5 still suggests that the number of clauses increases, but a look at $F_{\text{abs}}(\text{Paley-73}, 4)$ shows that the excess number of clauses is only due to the higher density. Indeed,

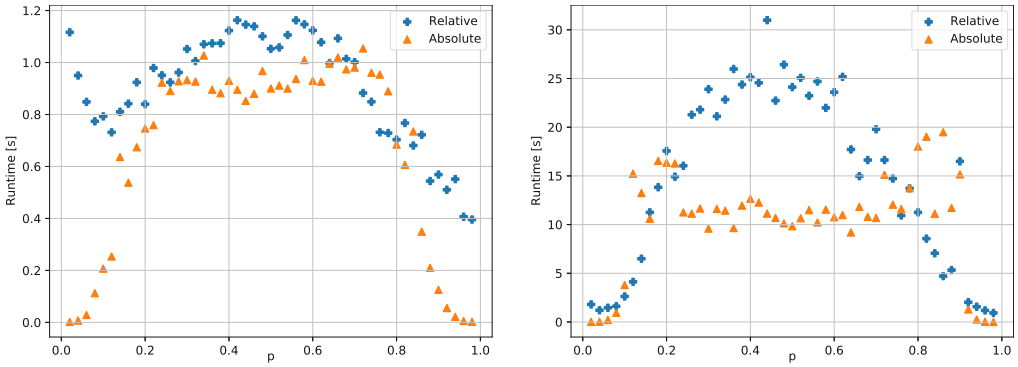
[6]The encodings could not verify whether the twin-width is 5 or 4. For the purposes of the discussion we treat 5 as the twin-width, as it still shows the comparative performance of the two encodings.

$F_{\text{abs}}(\text{Paley-73}, 4)$ even requires slightly more variables and clauses than $F_{\text{abs}}(\text{Paley-73}, 36)$. Hence, at least in terms of size, the absolute encoding handles higher twin-widths better than the relative encoding.

The runtime for the satisfiable solver calls generally reflects the difference in encoding sizes, except for the Paley-73 graph. It is not completely clear why the relative encoding performs slightly better here. One possible explanation is that the graph's twin-width is comparatively easy to find: due to the high twin-width and high density, non-optimal contractions quickly exceed the bound. Indeed, the solver using the relative encoding runs into comparatively few conflicts: in not even twice the time, the solver runs into more than twelve times as many conflicts compared to the absolute encoding.

The runtimes for the unsatisfiable calls using the relative encoding are somewhat puzzling. Here, the absolute encoding behaves as expected, with very short runtimes when $lb_1 = tww(G)$ and a high runtime for the hard Balaban 10-Cage instance. The relative encoding, on the other hand, requires a long time to figure out that there is not a single contraction possible without violating the upper bound.

The named graphs used in this experiment are few and very structured. We scale the comparison to a larger number of graphs without a clear structure by running both experiments on a large number of random graphs. The results are shown in Figure 4.6, where each point represents the average runtime over 100 instances. We omit the ten-vertex graphs, as both encodings solve them in under a second on average. For fifteen-vertex graphs, the absolute encoding starts to perform visibly better, and for twenty-vertex graphs, particularly with higher densities, the absolute encoding performs better on almost all instances.



(a) 15 Vertices.

(b) 20 Vertices.

Figure 4.6: Runtimes for twin-width encodings on random graphs with different numbers of vertices. The x-axis shows the edge density and each point represents the average runtime over 100 graphs.

The strength of the relative encoding is its consistency. This is illustrated in Figure 4.6b, where the points for the relative encoding form a bell shape, while the points for the absolute encoding form no clear shape. Further analysis of the results reveals the reason

for this difference. While the absolute encoding is indeed faster than the relative encoding on almost all instances, the average runtime is dominated by very few instances, where the absolute encoding takes a comparatively long time for the last, unsatisfiable, solver run. The relative encoding's runtime, on the other hand, is similar on all instances.

4.7 Conclusion

We proposed the first practical approach to computing the exact twin-width of graphs, utilizing the power of state-of-the-art SAT-solvers. This allowed us to reveal the twin-width of several important graphs. Our results provide the first step for showing general twin-width bounds for infinite graph classes. For instance, our data suggests $tww(\text{Paley-}n) = (n-1)/2$. Surprisingly, up to $n = 6$, the $n \times n$ grids have twin-width 3. It would be interesting to know if and when twin-width 4 is required.

The two proposed SAT encodings' different performance is impressive: while the relative encoding can determine the twin-width of small graphs very fast, the absolute encoding scales better to larger graphs and graphs for which finding the twin-width is hard. The absolute encoding's better scaling comes not only from its reduced size but also from being tweaked towards better complementing the SAT-solving process. Our results show the benefits of investing extra effort into refining the SAT encoding.

We hope that our results provide new insights and stimulate further theoretical investigations on twin-width. We also hope that our results provide a first step towards a practical use of twin-width. A next step would be the implementation and testing of twin-width-based dynamic programming algorithms like the algorithms for k-Independent Set and k-Dominating Set proposed by Bonnet et al. (2021b), which are single exponential in the twin-width.

CHAPTER 5

Directed Feedback Vertex Set

5.1 Introduction

The *directed feedback vertex set problem (DFVSP)* is one of Karp's original 21 NP-complete problems (Karp, 1972) and has a wide range of applications such as for argumentation frameworks (Dvorák et al., 2012, 2022), deadlock detection, program verification and VLSI chip design (Silberschatz et al., 2018). The problem is to find for a given directed graph a set of vertices—a *directed feedback vertex set (DFVS)*—such that every directed cycle uses at least one of the DFVS' vertices. In this chapter, we consider the optimization version of the problem, where we search for a minimum DFVS of the graph.

While the problem is known to be fixed parameter-tractable (Chen et al., 2008) in the size of the minimum DFVS, these algorithms do not help us in practice (Fleischer et al., 2009), where the size of a minimum DFVS can become prohibitively large. Other possible approaches include explicit branch-and-bound solvers (Lin and Jou, 1999), as well as encodings into a standard format for optimization problems. However, also these do not manage to solve all practically relevant instances (Lin and Jou, 1999).

We attack this problem from two directions. First, we use knowledge from the *Vertex Cover Problem (VCP)* where the key to success is the application of *data reductions*, which results in a smaller graph that allows for easier solving. The reductions' transfer from VCP to DFVSP is possible since VCP is a special case of DFVSP. Moreover, although many DFVSP instances do not match this special case, we give a theoretical basis that allows us to lift many data reductions from VCP to DFVSP. We use the fact that for these data reductions, it is often sufficient that the DFVSP instance behaves *locally* like a VCP instance. We use this to lift several VCP data reductions to DFVS and, additionally, introduce several more general data reductions.

Our second advancement revisits the idea of using a general-purpose optimization solver for DFVSP. We consider Maximum Satisfiability (MaxSAT) solvers as they build on top

of Propositional Satisfiability (SAT) solvers, which are designed for efficiently handling binary decisions, in our case, inclusion or exclusion of a vertex, and have seen tremendous advances in the last decade.

The problem we face when encoding DFVSP in propositional logic is that the resulting formula quickly becomes prohibitively large. A straightforward well-performing encoding has a size cubic in the number of vertices (Janota et al., 2017). Worse, the size of the encoding that is most suitable for our purposes can become exponential in the number of vertices: we list all cycles and state that any solution must contain at least one vertex from every cycle.

Fortunately, it is known that a subset of an instance's constraints is often sufficient for finding a feasible solution for the whole instance. This is formalized in approaches like *counter-example guided abstraction refinement (CEGAR)* (Clarke et al., 2003). CEGAR starts with a subset of the constraints, called an under-abstraction, and whenever the solver returns an infeasible solution, further constraints are added for the next solver run until a feasible solution is obtained. We propose an even faster approach: by implementing *cycle propagation* directly into the MaxSAT solver, it is possible to immediately add necessary constraints whenever they are needed instead of waiting for the solver to finish. The solver then immediately corrects its decision and never returns an infeasible solution.

Our implementation won PACE 2022 (Schulz et al., 2022).

5.1.1 Contributions

Our main contributions are as follows:

1. We found a general condition that allows us to lift data reductions from the vertex cover problem to the DFVSP problem.
2. For many reductions, we went even further and provided non-trivial generalizations of vertex cover reductions that are applicable in a wide range of cases.
3. We show that implementing cycle propagation is highly beneficial for MaxSAT-based DFVSP-solving, especially since it is tightly integrated into the MaxSAT solver.
4. In terms of data reductions, our experimental evaluation reveals that our novel reductions can significantly reduce instance sizes, especially on structured instances.
5. In terms of cycle propagation, the experiments showed a significant speedup, which is even larger when the cycle propagation happens within the MaxSAT solver.

Overall, the theoretical innovations of this work, together with the highly engineered MaxSAT solvers as a basis, provide a new and highly efficient strategy of exactly solving DFVSP instances that significantly outperforms any currently available implementation.

5.1.2 Related Work

We discussed CEGAR already in Section 2.5 and will focus on DFVSP. There has been previous work on data reductions for DFVSP (Koehler, 2005; Lemaic, 2008; Levy and Low, 1988; Lin and Jou, 1999), approximate solvers (Even et al., 1998; Zhou, 2016), and exact solvers (Bao et al., 2018; Fages and Lal, 2006; Funke and Reinelt, 1996). Especially for the latter, there has been a large increase recently since DFVSP was the topic of this year's PACE (Schulz et al., 2022). Here, many approaches were also based on a CEGAR-like method, i.e., repeatedly refining a small set of constraints but using integer linear programming solvers instead of MaxSAT solvers. Nevertheless, to the best of our knowledge, both our data reductions as well as the idea of using cycle propagation *within* a solver for DFVSP are novel.

5.2 Preliminaries

(Di)Graphs

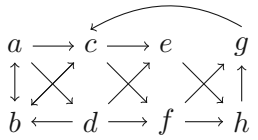
We consider both undirected and directed graphs (digraphs). For a (di)graph G, we denote by $V(G)$ its set of vertices and by $E(G)$ its set of edges (or arcs if G is a digraph). Further, we denote an (undirected) edge between vertices u and v as $\{u, v\}$ and the arc from u to v as (u, v). If for an arc $(u, v) \in E(G)$ also the arc in the other direction is present, i.e., $(v, u) \in E(G)$, then we call it a bi-edge, and an arc (u, u) is called a *loop*. The neighbors of a vertex in a graph G are denoted by $N_G(v)$, and for a digraph we use $N_G^{pre}(v), N_G^{succ}(v), N_G^{bi}(v), N_G(v)$ for the set of predecessors, successors, their intersection, and their union, respectively. If G is clear from the context, we may omit the subscript. In a (di)graph a (di)clique is a set S of vertices where each $v \in S$ satisfies $S \setminus \{v\} \subseteq N(v)$.

In the later sections we need some operations to modify (di)graphs. We define

- $G - S$ as the (di)graph obtained by removing a set S of vertices or edges from a (di)graph G. If S is a singleton set $\{v\}$ or $\{(u, v)\}$ we may omit the brackets and write $G - v$ or $G - (u, v)$.
- $G + S$ is defined analogously but adds vertices or edges.
- $G[S]$ as the induced sub(di)graph of G with respect to a set of vertices S, i.e., $V(G[S]) = S$ and $E(G[S]) = E(G) \cap S \times S$.
- $\Pi(G)$ for a digraph G as the graph with $V(\Pi(G)) = V(G)$ and $E(\Pi(G)) = \{\{u, v\} : (u, v), (v, u) \in E(G)\}$.

The Directed Feedback Vertex Set Problem (DFVSP)

We can now introduce the central problem addressed in this chapter.



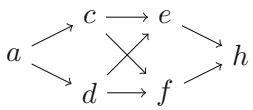
(a) A DFVSP instance.

$$\mathcal{C} = \{\{a, b\}, \{c, f, g\}, \{b, c\}, \{c, e, g, h\}\}$$
$$\mathcal{S} = \{\{\neg a\}, \{\neg b\}, \{\neg c\}, \{\neg d\} \{\neg e\}, \{\neg f\}, \{\neg g\}, \{\neg h\}\}$$

(b) As CNF $\mathcal{C}$ and set of soft clauses $\mathcal{S}$.

$$P = \{\neg a, b, \neg c, \neg d, \neg e, \neg f, g, \neg h\}$$

(c) An optimal solution.



(d) The resulting DAG.

Figure 5.1: Example of a DFVSP instance, a solution, and its encoding as a MaxSAT instance.

Definition 5.1 (Cycle, DFVS)**.** *Given a digraph G a path is a list of vertices $v_1, \ldots, v_n$ such that for $i = 1, \ldots, n-1$, there exists an arc $(v_i, v_{i+1}) \in E(G)$. A cycle is a path $v_1, \ldots, v_n$ such that $v_1 = v_n$. Furthermore, a path (or cycle) is uncovered, if there is no cycle $v'_1, \ldots, v'_m$ such that $\{ v'_i : i = 1, \ldots, m \} \subsetneq \{ v_i : i = 1, \ldots, n \}$ and the* length *of a cycle is the number of distinct vertices in the cycle. $\mathcal{C}(G)$ refers to the set of all uncovered cycles in G.*

A Directed Feedback Vertex Set (DFVS) *of G is a set $D \subseteq V$ such that every cycle of G contains at least one vertex in D. A minimum DFVS is a DFVS of minimum cardinality. We denote by $DFVS(G)$ the minimum cardinality of any DFVS of G.*

An example of a DFVSP-instance with a solution is given in Figures 5.1a and 5.1d. As an example for covered and uncovered cycles, consider the cycle a, c, b, a. The cycle is covered, as vertices of both (uncovered) cycles a, b, a and c, b, c are proper subsets of $\{ a, b, c \}$.

We introduce problems (Vertex Cover, Propositional Satisfiability) and relate them to DFVSP. Throughout this chapter, we make this correspondence clearer by using similar names for corresponding objects, e.g., C_i for a clause in propositional logic as they correspond to cycles.

Whenever each uncovered cycle has length 2 and using $\Pi(G)$, we can state DFVSP as follows:

Definition 5.2 (Vertex Cover)**.** *Let G be an undirected graph. A minimum Vertex Cover (minimum VC) is a set $D \subseteq V(G)$, such that D is of minimum cardinality and for each edge $\{u, v\} \in E(G)$, it holds that $\{u, v\} \cap D \neq \emptyset$. $VC(H)$ denotes the minimum cardinality over all VCs of H.*

Propositional Logic and SAT solvers

We use the definitions from Section 2.2 and connect them to DFVSP. We refer to a CNF formula as $\mathcal{C}$, the set of variables used in $\mathcal{C}$ as V, the clauses of $\mathcal{C}$ as C_i, and denote clauses as a set of literals. Further, we use the equivalency $\neg\neg v = v$.

We represent truth assignments as subsets of $V \cup \{ \neg v : v \in V \}$ that for any $v \in V$ does not contain both v and $\neg v$. Here, a positive literal indicates the value true, a negative literal the value false, and if a variable does not occur in the assignment, it isn't assigned a value (yet).

We encode DFVSP as a MaxSAT instance by using $V(G)$ as the variables and adding for each uncovered cycle $v_1 \dots v_n \in \mathcal{C}(G)$ a corresponding hard clause $\{ v_1, \dots, v_n \}$. Thus, if v_i is true, it is in the DFVS. Accordingly, to achieve minimum cardinality, we add a soft clause $\{\neg v\}$ for each $v \in V(G)$. A MaxSAT solution then maximizes the elements that are not in the DFVS, effectively minimizing the elements that are in the solution. In Figures 5.1b and 5.1c, we see an example of the MaxSAT encoding and the corresponding solution.

5.3 Solver Architecture and Outline

We give a brief overview of our algorithm, which also determines the structure of the chapter. The general algorithm is shown in Algorithm 5.1.

Algorithm 5.1: The complete DFVSP algorithm.

```
1 G ←Reduce(G)
2 C' ← FindShortCycles(G)
3 if C' = C(G) then
4 |   G,C' ← ReduceWithAllCycles(G, C')
5 end
6 D ←DFVS_MaxSAT(G,C')
7 return D
```

The algorithm starts with performing general data reductions, discussed in Section 5.4, and then searches for a small set of short cycles.

Short cycles are cycles with a specified maximum length. Bounded depth-first search can easily find these short cycles, as long as the bound, i.e., cycle length, is small enough. We discuss the associated limits on both the cycle length and the number of cycles in the implementation details in Section 5.6.

Should the set of short cycles represent all uncovered cycles, we can apply additional data reductions, which are also discussed in Section 5.4. We increase our chances of finding all uncovered cycles by ignoring the aforementioned limits and incrementally extending

the maximum length of a cycle, as long as the number of new cycles we find decreases monotonically.

A MaxSAT solver is then used to solve the reduced instance. The set $\mathcal{C}' \subseteq \mathcal{C}(G)$ is given to the solver as the initial set of constraints. In case $\mathcal{C}'$ contains all uncovered cycles, the MaxSAT solver simply computes a minimum DFVS. When $\mathcal{C}'$ is not complete, the cycle propagation discussed in Section 5.5 ensures that the solver still returns a valid DFVS.

We evaluate how effective our solver is in Section 5.6.

5.4 Data Reductions[1]

Data reductions are the first step in our approach. They aim to shrink the input digraph in such a way that a minimum DFVS for the reduced digraph can easily be extended to a minimum DFVS for the original digraph. To this end, we apply a wide range of reduction rules.

5.4.1 DFVSP Reductions

For DFVSP, there is already a wide range of rules by Levy and Low (1988); Lemaic (2008); Lin and Jou (1999), which we use in an unmodified manner. In this section, we focus on our novel reductions and list already established reductions in Appendix A.2.1. We only give the following reduction by Lin and Jou (1999), as its inapplicability is a precondition for a later reduction:

Reduction 5.1 (PIE)**.** *If there is an arc $(u, v) \in E(G)$ such that $(v, u) \notin E(G)$ and every path from v to u in G uses a bi-edge, replace G by $G - (u, v)$.*

Apart from this, we generalized DOME (Lin and Jou, 1999) from arcs dominated by length 2 paths to arcs dominated by arbitrary length paths:

Reduction 5.2 (DOME++)**.** *If there is an arc $(v, u) \in E(G)$ such that $(u, v) \notin E(G)$ and (i) every path that starts at v and ends at u uses a bi-edge or a vertex from $N^{pre}(u) \setminus \{v\}$ or (ii) every path that starts at u and ends at v uses a bi-edge or a vertex from $N^{succ}(v) \setminus \{u\}$, then replace G by $G - (v, u)$.*

Additionally, while enumerating all uncovered cycles is generally not feasible due to their potentially exponential number, it is often possible in practice. This allows the following reduction.

Reduction 5.3 (ALLCYCLES)**.** *If there is an arc $(v, u) \in E(G)$ such that every cycle that visits v immediately after u is covered, then replace G by $G - (v, u)$.*

Fact 5.1. *DOME++ and ALLCYCLES are sound if G is loop-free.*

Apart from the DFVSP reductions, we also lifted and generalized VCP reductions.

[1] Proofs for all theorems are in Appendix A.2.3

5.4.2 VCP Reductions

As noted already in the preliminaries, VCP and DFVSP are related. We formalized this intuition as follows:

Lemma 5.2. *Let G be a digraph, then $S \subseteq V(G)$ is a DFVS if and only if*

- *S is a VC of $\Pi(G)$, and*
- *S is a DFVS of $G - E(\Pi(G))$.*

Thus, we can treat a DFVSP instance as a combination of a VCP instance $\Pi(G)$ and a smaller DFVSP instance without bi-edges. Hence, given sufficient preconditions, it suggests itself to apply VCP reductions to DFVSP. We derive these preconditions using boundary reductions similar to those of (Fellows et al., 2018), which intuitively locally replace one part the a graph with another.

Definition 5.3 (Boundary VCP Reduction)**.** *A boundary VCP reduction is a tuple $r = (H, H', B, c)$, where H, H' are graphs, $B \subseteq V(H) \cap V(H')$ is a set of non-isolated vertices in both H and H', and $c \in \mathbb{Z}$, such that for all $X \subseteq B$ it holds that*

$$VC(H - X) = VC(H' - X) + c.$$

A reduction r is applicable *to G, if the vertices in B are (i) not isolated in G, (ii) an independent set in G, and (iii) the overlapping vertices, i.e., $V(G) \cap V(H) = B = V(G) \cap V(H')$.*

Example 5.1. *An example of a boundary reduction $r = (H, H', B, c)$ is given by the boundary $B = \{b_1, b_2\}$, size difference $c = 1$, and the graphs*

$$H = \quad b_1 \text{ — } v_1 \text{ — } v_2 \text{ — } b_2 \qquad H' = \quad b_1 \text{ — } b_2.$$

r is a boundary reduction since for $X \subseteq \{b_1, b_2\}, X \neq \emptyset$, there is always still the edge between v_1 and v_2 in $H - X$, which means that $VC(H - X) = 1$, whereas $H' - X$ is edgeless. On the other hand, if $X = \emptyset$, then $VC(H - X) = VC(H) = 2$ and $VC(H' - X) = VC(H') = 1$.

Importantly, applicability guarantees soundness and the desired locality property (Proof in Appendix A.2.3):

Theorem 5.3. *For every graph G such that (H, H', B, c) is applicable it holds that for every minimum VC S of $G + V(H) + E(H)$ there is a minimum VC S' of $G + V(H') + E(H')$ such that $|S| = |S'| + c$, and $S \cap V(G) = S' \cap V(G)$.*

Not only does this theorem guarantee that the size of a minimum VC changes by c, but it also tells us that the modification only has local effects on the minimum VCs. This locality is particularly interesting for lifting VCP reductions to DFVSP, as it allows their application when a digraph "locally behaves like a VCP instance". This intuition is formalized in the following theorem, whose proof can be found in Appendix A.2.3.

Theorem 5.4. *Let G be a digraph, $r = (H, H', B, c)$ be a boundary VCP reduction and $\Pi(G) = G' + V(H) + E(H)$ such that r is applicable to G'. If*
(i) all edges incident in G to any $v \in V(H) \setminus B$ are bi-edges and
(ii) for every arc $(u, w) \in E(G)$ at least one of the following holds:
(ii.a) (u, w) is a bi-edge or
(ii.b) $|\{u, w\} \cap B| \leq 1$ then

$$DFVS(G) = DFVS(G^*) + c,$$

where $G^* = G - (V(H) \setminus B) - B \times B + V(H') + \{\, (u, v) : \{u, v\} \in E(H') \,\}$.

We can, thus, use many VCP reductions without modification by checking the preconditions of Theorem 5.4. We omit those reductions that we strictly generalize in the next section. They can, however, be found in Appendix A.2.2. This leaves only the following reduction, which we apply without any changes:

Reduction 5.4 (3EMPTY (Stege and Fellows, 1999; Fellows et al., 2018))**.** *If there exists a vertex $v \in V(G)$ such that $|N(v)| = 3$ and $|E(G[N(v)])| = 0$, then let $N(v) = \{\, a, b, c \,\}$ and replace G by*

$$G - v + \{\{a, b\}\}, \{\{b, c\}\} + \{a\} \times N(b) + \{b\} \times N(c) + \{c\} \times N(a).$$

5.4.3 Directed Versions of VCP Reductions

Some VCP reductions can be generalized to DFVSP, even when Theorem 5.4 does not apply. Note that all of the following reductions are strict generalizations of VCP reductions. I.e., when a digraph only has bi-edges, then each of the new DFVS reductions corresponds to a VCP reduction.

A simple example of this is the SUBSET reduction:

Reduction 5.5 (SUBSET)**.** *If there exists $v, u \in V(G)$ such that $(v, u), (u, v) \in E(G)$, $N_{in}(v) \subseteq N_{in}(u) \cup \{u\}$, and $N_{out}(v) \subseteq N_{out}(u) \cup \{u\}$, then replace G by $G - u$.*

Observation 5.5. *Let G be a digraph. After applying SUBSET to vertices v, u resulting in G', it holds that for every minimum DFVS S of G' the set $S \cup \{u\}$ is a minimum DFVS of G.*

Other reductions, such as the MANYFOLD reduction, are more advanced.

Reduction 5.6 (MANYFOLD)**.** *If there exists a vertex $v \in V(G)$ such that $N(v) = N^{bi}(v)$ and there is a partition (C_1, C_2) of $N(v)$, where*

- $|C_1| \geq |C_2|$,
- $G[C_i]$ *is a diclique for* $i = 1, 2$,
- M *is the set of non-arcs of* $G[N(v)]$,
- $(c, d) \in M$ *implies that either* $(d, c) \in M$ *and there is no uncovered path between* c *and* d, *or* $(d, c) \notin M$ *and every uncovered path from* d *to* c *uses the arc* (d, c)
- *for each* $c_1 \in C_1$, *there is exactly one* $c_2 \in C_2$ *(denoted* $c_2(c_1)$*) such that* $(c_1, c_2) \in M$ *or* $(c_2, c_1) \in M$,

then, replace G *by*

$$\begin{aligned} & G - v - C_2 \\ + & \bigcup_{c_1 \in C_1} \{c_1\} \times N^{succ}(c_2(c_1)) \cup N^{pre}(c_2(c_1)) \times \{c_1\} \\ - & \bigcup_{c_1 \in C_1} (c_1, c_1). \end{aligned}$$

We go over the conditions to explain their relevance.

- $N(v) = N^{bi}(v)$ needs to hold to ensure that when a minimum DFVS S does not contain v, then it must be the case that $N(v)$ is a subset of S.
- For each $c_1 \in C_1$ there is exactly one $c_2 \in C_2$ with missing arc (c_1, c_2) or (c_2, c_1). This ensures that when c_1 is not in a minimum DFVS S, then either all vertices from C_2 are in S, or only the uniquely determined vertex c_2 is not in S.
- The conditions on M ensure that when we perform the contraction, we only add cycles for which there exists a corresponding cycle in the original digraph.

Note that the conditions on the arcs in M are NP-hard to check. Later, we discuss alternative tractable and sufficient conditions. First, we state soundness (Proof in Appendix A.2.3).

Theorem 5.6. *Let* G *be a loop-free digraph, such that PIE is not applicable and MANYFOLD is applicable to* $v^* \in V(G)$ *and* G' *be the graph obtained from* G *after MANYFOLD was applied on vertex* v^*, *then* $DFVS(G) = DFVS(G') + |C_2|$ *and given a minimum DFVS of* G', *we can in polynomial time compute a minimum DFVS of* G.

We need to check two possible conditions on the arcs in M. First, if $(d, c) \in E(G)$ we use straightness:

Definition 5.4 (Straightness)**.** *Let* G *be a digraph and* $(d, c) \in E(G)$. *Then* (d, c) *is* straight *if* $(c, d) \notin E(G)$ *and either (i) every arc* $(d, c') \in E(G)$, *such that* $c' \neq c$ *is a bi-edge, or (ii) every arc* $(d', c) \in E(G)$ *such that* $d' \neq d$ *is a bi-edge.*

As desired, if the arc (d, c) is straight, then every uncovered path that contains d after c uses it. If, on the other hand, (d, c) is not in $E(G)$, we need to prohibit the existence of an uncovered path between c and d. Here, we give a sufficient condition using *Strongly Connected Components (SCCs)*. Recall that an SCC is a subset maximal set S of vertices such that for every combination v, u of vertices in S, there is a (directed) path from u to v.

We consider for a digraph G the SCCs of $G - E(\Pi(G))$, i.e., G without bi-edges, and denote for a vertex $v \in V(G)$ by $\mathrm{SCC}_{||}^{G}(v)$ the unique SCC containing it. If G is clear from the context, we may omit the superscript. Then, the PIE reduction (Lin and Jou, 1999) allows us to remove arcs between vertices u, v such that $\mathrm{SCC}_{||}(v) \neq \mathrm{SCC}_{||}(u)$. This entails that when $\mathrm{SCC}_{||}(v) \neq \mathrm{SCC}_{||}(u)$, every path between u and v uses a bi-edge and is thus covered.

Together, these conditions give us a tractable way of guaranteeing the applicability of MANYFOLD regardless of whether both (c, d) and (d, c) are in M or whether only one of them is. Figure 5.2 shows example applications.

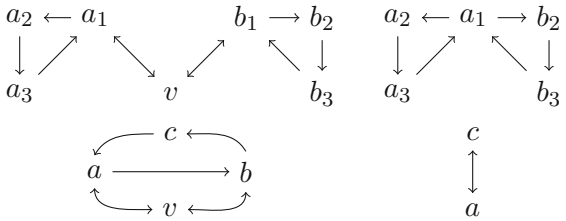


Figure 5.2: MANYFOLDs: Left is before, right is after. The first row shows MANYFOLD, where $\mathrm{SCC}_{||}(a_1) = \{a_1, a_2, a_3\} \neq \{b_1, b_2, b_3\} = \mathrm{SCC}_{||}(b_1)$ and the second row shows MANYFOLD, where the arc (a, b) is straight.

We note that it is not necessary to recompute SCCs at every step since we can update them after each reduction in an approximate but safe manner and only recompute them periodically.

Apart from MANYFOLD, we can also generalize 4PATH in a similar manner by exploiting the lack of uncovered paths between some of the involved vertices.

Reduction 5.7 (4PATH)**.** *If there exists a vertex $v \in V(G)$ such that*

- $N(v) = N^{bi}(v) = \{a, b, c, d\}$,
- $E(G[N(v)]) = \{\, (a, b), (b, a), (b, c), (c, b), (c, d), (d, c) \,\}$,
- *and there is no uncovered path between any pair of vertices from* $\{\, \{a, c\}, \{a, d\}, \{d, b\} \,\}$,

then replace G by

Algorithm 5.2: An algorithm that checks whether a vertex v is unconfined in a graph G.

```
1  A ← {v}
2  N ← { u ∈ V(G) : G[A ∪ {u}] is cyclic }
3  P ← {u ∈ N | |N^succ(u) ∩ A| + |N_pred(u) ∩ A| = 2}
4  if P ≠ ∅ then
5  |  u ← argmin_{u'∈P} |N(u') \ (N ∪ A)| if |N(u) ← (N ∪ A)| = 0 then
6  |  |  return True
7  |  else if |N(u) \ (N ∪ A)| = 1 then
8  |  |  A ← A ∪ {u}
9  |  |  go to 3
10 end
11 return False
```

$$\begin{aligned} G - v + \{(a,c),(c,a),(a,d),(d,a),(b,d),(d,b)\} \\ + \{a,b\} \times N^{succ}(d) + N_{pred}(d) \times \{a,b\} \\ + \{c,d\} \times N^{succ}(a) + N_{pred}(a) \times \{c,d\}. \end{aligned}$$

Again, we practically ensure that every path is covered by requiring $\mathrm{SCC}_{||}(a) \neq \mathrm{SCC}_{||}(c), \mathrm{SCC}_{||}(a) \neq \mathrm{SCC}_{||}(d)$ and $\mathrm{SCC}_{||}(d) \neq \mathrm{SCC}_{||}(b)$.

Theorem 5.7. *Let G be a digraph such that 4PATH is applicable to $v^* \in V(G)$ and G' be the graph obtained from G after 4PATH was applied to the vertex v^*, then*

- *$DFVS(G) = DFVS(G')$, and*
- *given a minimum DFVS of G', we can in polynomial time compute a minimum DFVS of G.*

The proof is in Appendix A.2.3.

Whereas the above reductions all capture fixed graph patterns, the applicability of the following one is determined by the iterative procedure in Algorithm 5.2.

Reduction 5.8 (UNCONFINED)**.** *If there is a vertex $v \in V(G)$ such that* CheckUnconfined*(v, G) returns True, replace G by $G - v$.*

When a vertex v is unconfined, this guarantees us that while there may be minimum DFVSs that do not contain v, there is at least one, which does.

Theorem 5.8. *Let G be a digraph. After applying UNCONFINED to vertex v resulting in G', it holds that for every minimum DFVS S of G' the set $S \cup \{v\}$ is a minimum DFVS of G.*

The proof is in Appendix A.2.3.

This concludes the data reductions that we use and brings us to the solving step.

5.5 MaxSAT Solver

We compute the minimum DFVS for the reduced instance using a MaxSAT solver. Recall from Section 5.2 that we add one disjunction per cycle containing exactly the variables corresponding to the vertices in the cycle. Since there is a 1:1 correspondence between vertices and variables, cycles and clauses, as well as model and DFVS, we treat them synonymously in this section. We also refer to a DFVS candidate that does not break all cycles as *infeasible* and to a DFVS as a *feasible* solution.

Enumerating all cycles for a complete encoding is generally not feasible. A well-established technique that deals with this issue is CEGAR (Clarke et al., 2003). In the context of DFVSP, a CEGAR approach initially gives the solver a small, usually not comprehensive, set of uncovered cycles. Whenever the solver returns a solution that is not a valid DFVS, we add cycles that are not broken by the infeasible solution. This is repeated until the solver returns a feasible solution. Very often, a comparatively small number of constraints, in our case cycles, is sufficient for finding a feasible solution.

The main drawback of this CEGAR approach is the computational overhead. While a solver's decision may quickly imply that the solution is infeasible, the solver may run long past that point until it returns a solution. Further, after an infeasible solution is returned, it is hard to determine which of the solver's decisions caused the infeasibility. Lacking this knowledge, we have to add many cycles that are not necessary for guiding the solver towards a feasible solution.

We propose cycle propagation for improved performance. Cycle propagation adds the feasibility check directly into the MaxSAT solver's logic and adds the necessary cycles at exactly the point where the MaxSAT solver's decision would cause the solution to become infeasible.

We focus on *core-guided* MaxSAT solvers. Here, the MaxSAT solver implements the search for an optimal solution and calls the SAT solver repeatedly. For each SAT call, the MaxSAT solver extends the input CNF by extra clauses related to the search for an optimal solution (Morgado et al., 2014a). We, therefore, add cycle propagation to the SAT solver, as the decisions that lead to infeasibility are made here. In order to introduce cycle propagation, we first discuss the basics of CDCL, the algorithm used by most modern SAT solvers (Silva and Sakallah, 1999; Moskewicz et al., 2001).

5.5.1 Conflict Driven Clause Learning (CDCL)

We limit ourselves to a cursory discussion of CDCL that introduces the necessary concepts to understand cycle propagation. Remember from Section 5.2 that a SAT instance consists of a set of variables V and a set of clauses $\mathcal{C}$. The whole algorithm, including cycle

Algorithm 5.3: The modified CDCL algorithm.

```
1  D ← ∅;
2  while |C| > 0 do
3  |  C ← C \ { C ∈ C : D ∩ C ≠ ∅ };
4  |  D′ ← { ¬ℓ : ℓ ∈ D };
5  |  if |{ C ∈ C : |C \ D′| = 0 }| > 0 then
6  |  |  if No Decisions then
7  |  |  |  return false
8  |  |  end
9  |  |  C ← C∪ analyzeConflict();
10 |  |  C, D ← backtrack();
11 |  else if |{ C ∈ C : |C \ D′| = 1 }| > 0 then
12 |  |  D ← booleanClausePropagation();
13 |  else
14 |  |  V′ = { v ∈ V(G) : ¬v ∈ D };
15 |  |  if G[V′] contains a cycle C then
16 |  |  |  C ← C ∪ C;
17 |  |  else
18 |  |  |  D ← D ∪ { decideLiteral() };
19 |  |  end
20 end
21 return true
```

propagation, is shown in Algorithm 5.3. Ignoring cycle propagation in the block starting at Line 14, the listing shows the basic CDCL algorithm.

CDCL incrementally extends a partial assignment D, assigning values to some of the variables, to a full assignment until it either obtains a model or knows that the formula is unsatisfiable. The algorithm only keeps unsatisfied clauses and removes satisfied ones (Line 3).

Conflicts occur when a clause C cannot be satisfied by any extension of D to a full assignment because D contains the negation of C's literals, as is checked in Line 5. Here, two things can happen. If the conflict occurred without any prior decision, the set of clauses implies a conflict, and the formula is unsatisfiable. Otherwise, CDCL learns a conflict clause: a clause based on the decisions that lead to the conflict and that prevents the solver from making the same set of decisions again. Afterwards, the solver backtracks, where it removes the corresponding literals from D and restores the corresponding removed clauses and literals to $\mathcal{C}$.

Boolean constraint propagation and *decisions* are used by CDCL to extend D. Boolean constraint propagation adds implied literals to D, where a literal is implied if there exists a clause where this literal is the only one remaining that can be satisfied by an extension

of D, as is checked in Line 11. Decisions add a selected literal to D after exhaustively applying Boolean constraint propagation without a conflict, as denoted in Line 18.

This description of CDCL is deliberately conceptual. Modern SAT and MaxSAT solvers are well-engineered pieces of software that use sophisticated data structures and algorithms which are integral to their performance. Particularly conflict analysis, backtracking, and decisions have not been covered here. We refer the interested reader to related literature (Biere et al., 2021) for more details.

With the knowledge of how CDCL works, we discuss the integration of cycle propagation next.

5.5.2 Cycle Propagation

Conflicts are a central concept in CDCL, as they signal the solver that a partial assignment is infeasible. Cycle propagation uses this mechanism to ensure that the solver stops as soon as D implies a cycle. We perform this check after Boolean constraint propagation in Line 14.

Cycles are only implied by negative literals since negative literals indicate that a vertex remains in the graph. Hence, it is sufficient to check whether the set of negative literals $V' = \{ v \in V : \neg v \in D \}$ induces an acyclic graph, i.e., whether $G[V']$ is acyclic. In case a cycle C is found, it is added as a clause to $\mathcal{C}$.

Adding C immediately causes a conflict since by definition D contains the negation of C. Hence, we achieve our goal of immediately stopping the solver. Further, we add a single cycle and corresponding conflict clause, thereby minimizing the number of extra constraints. This usually guides the solver quicker to a feasible solution than adding several cycles after the solver returns an infeasible solution. Note that the solver with cycle propagation never returns an infeasible solution.

The acyclicity check is performed using a DAG implemented as a simple doubly linked data structure. Here, each vertex knows its predecessors, successors, and has an order. The structure preserves two invariants: it is a DAG, and the order of a vertex is the maximum order over its predecessors plus one, or 0 if the vertex has no predecessors. Whenever a new vertex is inserted, its order is recursively propagated to the successors. Recursive calls are only necessary when the propagated order plus one is larger than the successor's order. Should the propagation reach the inserted vertex, we have found a cycle, and we remove the vertex, preserving the invariant that the structure is a DAG. Removal of a vertex requires recursively propagating the change to all successors whose ordering depends on the target vertex. Since the insertion of a vertex can introduce several cycles, we perform the propagations in a breadth-first order, which finds shorter cycles than depth-first order. Shorter cycles lead to shorter clauses and usually increased performance.

Cycle propagation is performed after Boolean constraint propagation for practical reasons. First, modern SAT solvers spend most of their time performing Boolean constraint

propagation and can perform this task very fast. Checking for cycles after each change to D would, therefore, cause a considerable slowdown of the solver. Second, we keep track of the changes to the partial assignment in between cycle propagation runs. This allows us to perform the aforementioned modifications to our data structure in bulk, further speeding up the acyclicity check. With these considerations, the runtime percentage dedicated to cycle propagation shown in the profiler is in the low single digits, as Boolean constraint propagation still takes up almost all of the runtime.

This concludes the conceptual description of our approach. Next, we discuss our empirical evaluation.

5.6 Experiments[2]

Instances

We use instances from the recent *PACE*, the argumentation framework competition *ICCMA*, and *random* graphs. The recent PACE provides 200 dedicated DFVSP instances.[3,4] The 137 ICCMA instances come from a recent argumentation framework competition[5], where we selected those instances with 50 to 1000 vertices. We also generated 1140 random instances using different parameterizations for the number of vertices and the probability an edge exists using the methodology of (Zhou, 2016). We generated 10 instances for each parameterization, which are reported as 114 instances, averaging the results over the respective 10 instances. The number of vertices ranges between 100 and 10000, and the average degree of a vertex varies from 2 to 50.

We preprocessed the instances by removing all self-loops, as the PACE instances met this requirement, and the competition solvers were not able to deal with instances containing self-loops.

Implementation

We implemented the proposed algorithm in our solver DAGer. Our implementation is based on the MaxSAT solver EvalMaxSAT (Avellaneda, 2020b), which uses Glucose 3 in the backend (Audemard and Simon, 2009). We chose EvalMaxSAT because it placed well in the 2021 MaxSAT evaluation[6] and the code base has no dependencies and can easily be modified and integrated.

We initially give up to 25000 short cycles with a maximum length of 4 to the MaxSAT solver. These limits have performed best overall. A lower maximum length does not find

[2]Source code is available at https://github.com/ASchidler/dfvs, results are available at https://doi.org/10.5281/zenodo.7271869

[3]https://pacechallenge.org/2022/tracks/

[4]At the time of writing, details on the origin of the instances have not been released.

[5]https://argumentationcompetition.org/2021/

[6]https://maxsat-evaluations.github.io/2021/

any cycles for some instances, while a higher maximum length seems to slow down the solver, as does a larger number of cycles.

Setup

Our implementation uses C++ and was compiled using gcc 7.5.0. We compared our solver to the second-place PACE solver *grapa-java*[7], which uses a CEGAR-like approach together with an integer linear programming solver and new data reductions.[8] As an additional baseline, we also used a direct DFVSP encoding into *SAT*, based on the transitive closure encoding for acyclicity (Janota et al., 2017), using our data reductions for preprocessing.

We used a time limit of 30 minutes and a memory limit of 8 GB. The experiments were run on servers with two AMD EPYC 7402 CPUs, each with 24 cores running at 2.8 GHz, and using Ubuntu 18.04.

An instance counts as *solved* if it was solved in all five runs, otherwise, if it was solved in at least one run, it is counted as *partially solved*. The given values are averaged over all runs.

5.6.1 Solvers

Comparing DAGer's performance to that of other solvers was the goal of our first experiment. The results, together with different configurations from the next experiment, are shown in Table 5.1 and as a cactus plot in Figure 5.3.

	PACE		ICCMA		Random	
Solver	Solved	Partial	Solved	Partial	Solved	Partial
DAGer	186	2	64	0	13	5
grapa-java	165	0	52	0	10	5
SAT	134	3	59	0	12	5
Configuration	Solved	Partial	Solved	Partial	Solved	Partial
No Cycle Propagation	180	2	62	0	11	6
No Data Reductions	151	6	63	1	13	5
No CP & DR	146	3	62	0	10	7

Table 5.1: Number of solved instances for different solvers and DAGer configurations. *Solved* and *Partial* show the number of solved and partially solved instances respectively.

DAGer performs better than both grapa-java and the SAT encoding for every instance group. Interestingly, the SAT encoding performs better than grapa-java on non-PACE

[7] https://gitlab.informatik.uni-bremen.de/grapa/java

[8] We tried to obtain further solvers for comparison. Unfortunately, for (Bao et al., 2018), we did not manage to get in contact with the authors, for (Fages and Lal, 2006) the source code is lost and the implementation by (Koehler, 2005) did not contain an exact solver

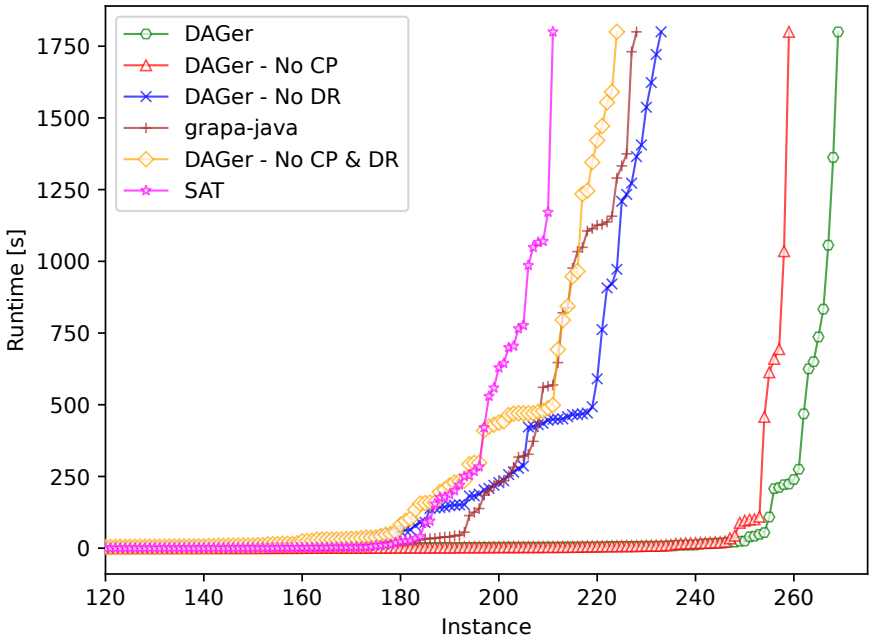


Figure 5.3: Cactus plot for different solvers and DAGer configurations.

instances. The cactus plot shows that instances are either very hard, or very easy, with very few solved instances having a high runtime.

DAGer excels on the PACE instances and solves almost half the ICCMA instances, but random instances seem to be hard for all solvers. We will examine this behavior for DAGer further in subsequent experiments.

5.6.2 Features

We measured the impact of our contributions by disabling cycle propagation (CP) or our new data reductions (DR). Table 5.1 shows the performance of these three additional configurations, and Figure 5.3 shows the runtime behavior. Without cycle propagation, DAGer calls the MaxSAT solver incrementally and whenever the solution D is infeasible, we add disjoint cycles from $G - D$. Hence, our experiment tests precisely the benefit of the integration into the solver.

Cycle propagation has a small impact in terms of the number of instances. While the number is small, the respective instances are hard and contain a very large number of uncovered cycles that we were unable to enumerate within the runtime. Without cycle propagation, DAGer solved 253 instances. The initial solution was infeasible for 132 of those instances, lazily generated clauses were not necessary for the remaining instances.

The impact of the data reduction depends strongly on the instance set. PACE instances are heavily reduced, but the impact on ICCMA and random instances is almost none. The reason for this is that our new reductions rely heavily on structural properties that are unlikely in randomized graphs. For ICCMA instances, the reason is different, which we will explore next.

Overall, without our contributions, DAGer would perform worse than grapa-java, but better than the SAT baseline, showing that the incremental approach is the more promising SAT approach for DFVSP.

5.6.3 Data Reductions

The effectiveness of the data reductions is not well represented by the number of instances the solving algorithm can solve, as in the future, they might be beneficial for instances that are too hard for current solvers. Figure 5.4 shows how much all instances were reduced in size and offers some interesting insights. First, many instances, particularly ICCMA instances are directly solved by our data reductions. The ICCMA instances seem to be either easily reducible and also easy to solve, or they are hard to reduce and solve. Second, the result on random instances shows that the reductions become less effective with increasing density. Lastly, on most instances, the data reductions can significantly decrease the instance's size.

We also wanted to see how much benefit our computationally more expensive reductions have over the simple reductions proposed by Levy and Low (1988). Figure 5.5 shows that our data reductions do not have much benefit over the simple reductions on random instances. For PACE instances, the reductions are very useful, reducing the size of almost all instances and directly solving many of them. For the ICCMA instances, the reductions either solve the instance or are ineffective.

5.6.4 Instance Size

Our last experiment focused on the potential correlation between the graph's size and the MaxSAT solver's ability to find a minimum DFVS. Figure 5.6 shows which instances have been solved, partially solved, or remained unsolved in relation to the number of vertices and density. While the number of uncovered cycles would also have been of interest, we could not enumerate them in a reasonable amount of time for hard instances. Since we were interested in the MaxSAT solver's performance, the figure uses the data from the reduced instances.

The figure shows that increased size and density indeed make the instance harder. Whereas for small graphs up to around 100 vertices, the density does not matter much, the density matters for larger graphs. The size limit for our approach seems to be around 10000 vertices, where the solver fails even for very sparse graphs.

Interestingly, at around 1000 vertices, there is a cluster of instances with high density that the solver solved successfully. It seems that instances, where almost the whole graph

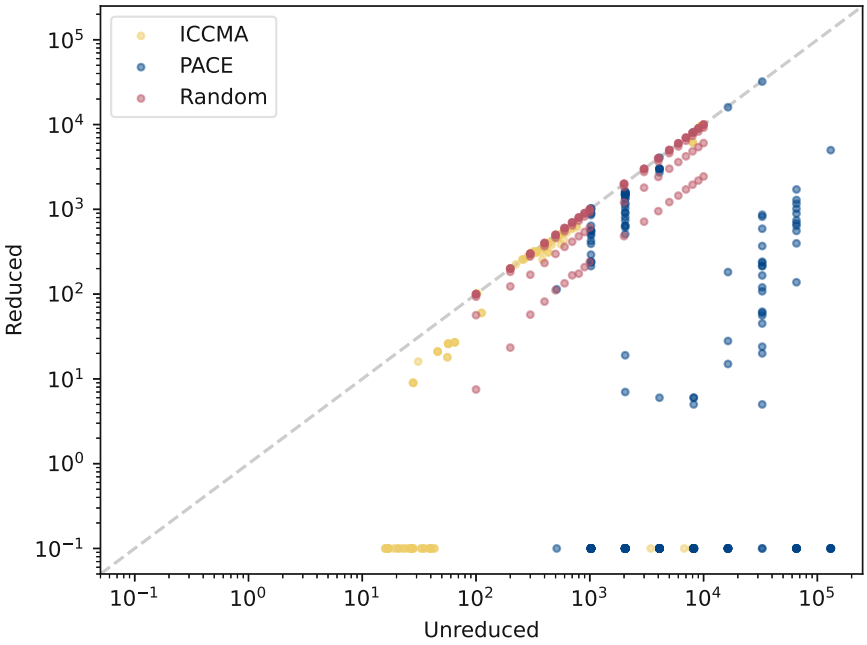


Figure 5.4: Comparison of graph sizes before and after applying data reductions. Due to the use of the logarithmic scale, we treat 0 as 0.1.

is part of a minimum DFVS, are again easier to solve than mid-density instances.

Random instances provide some more insight. For 100 vertices, DAGer solved almost all instances. The only exception is when the minimum DFVS size is around 50, here DAGer only managed to solve half the instances. Hence, they seem to be harder. For the remaining instances, DAGer was not able to solve instances with an average degree higher than 3, but managed to solve random instances with up to 1000 vertices.

5.6.5 Discussion

The results show that the data reductions perform particularly well on the PACE instances, where many instances have a high enough number of bi-edges. While still useful on the ICCMA instances, as there they solve many instances directly, the data reductions are not necessary, as the solver would have solved almost all of them.

Cycle propagation works in a complementary fashion to our data reductions. It works particularly well on instances that have few uncovered short cycles but a large number of uncovered cycles. While not many of the instances in our instance sets fell into this category, cycle propagation helped to solve several hard instances.

In general, a CEGAR approach works well for DFVSP as even without cycle propagation

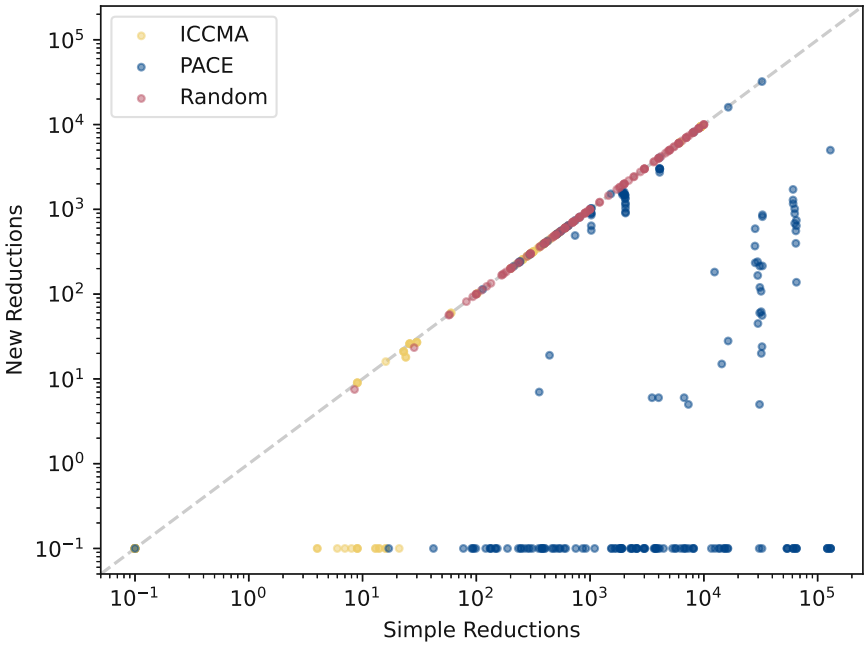


Figure 5.5: Comparison between using only simple reductions and using the data reductions we propose. Due to the use of the logarithmic scale, we treat 0 as 0.1.

and our data reductions the solver outperformed the second-best PACE solver on non-PACE instances.

Particularly challenging for all tested solvers are instances of high edge density, although there is a visible trend that indicates that instances with very high density could in turn become easier.

5.7 Conclusion

In this chapter, we discussed our novel approach to DFVSP. Key features are new data reductions lifted from related problems. Apart from the reductions themselves, we also provided a theoretical basis that can be used to lift further reductions in the future. The other key feature is cycle propagation. While lazily extending the set of constraints to obtain a feasible solution with a limited set of constraints works well, we managed to solve several hard instances by integrating this extension directly into the MaxSAT solver.

We have shown that cycle propagation works well in practice. We think that there are two avenues where we might further improve its performance: (i) there are several SAT solver details that might be used to further improve performance, particularly adapting

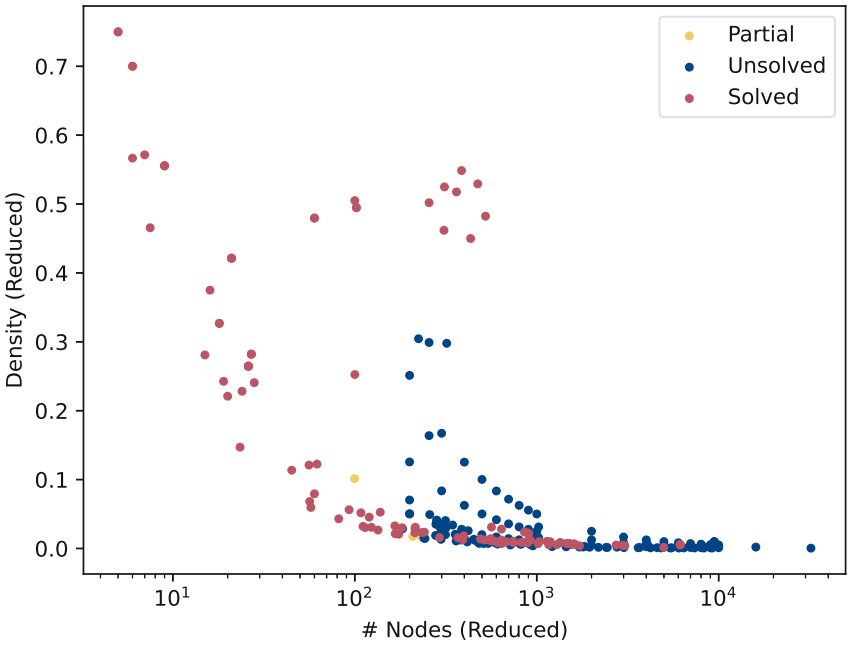


Figure 5.6: Solved instances in relation to the graph size and the density.

inprocessing and decision heuristics to incorporate domain-specific knowledge about DFVSP, and (ii) we used a core-guided MaxSAT solver for our implementation; it would be interesting to see how well cycle propagation performs integrated into an implicit hitting set based MaxSAT solver.

In the context of scalability, the results presented in this chapter show how effective a lazy approach can be in scaling a SAT encoding to larger instances. Even without integrating cycle propagation, this approach outperformed the runner-up solver from PACE. Furthermore, using a full encoding was not feasible for many hard instances. We could not enumerate all uncovered cycles for many instances, which rules out the encoding we used. Further, given that we solved instances with over 1000 vertices, any encoding cubic in the size of the vertices would use more than 10^9 many clauses, exceeding the solver's capabilities if the instance is not very easy to solve.

CHAPTER 6

Graph Coloring

6.1 Introduction

Graph coloring is the fundamental computational problem of coloring the vertices of a given undirected graph with as few colors as possible, avoiding monochromatic edges, or, equivalently, partitioning the graph's vertex set into as few independent sets as possible. Graph coloring arises naturally in many applications, including scheduling, register allocation, pattern matching, and computational geometry. The decision version of the problem—where the number of colors is given, and one asks whether a coloring exists—can be naturally cast as a constraint satisfaction problem: The graph's vertices are variables that range over a finite domain of colors, and each edge represents a binary inequality constraint. Graph coloring is one of Karp's 21 fundamental NP-hard problems (Karp, 1972).

For decades, much research has been devoted to developing algorithmic methods for graph coloring. One can distinguish between *exact methods* that search for a coloring with the smallest number of colors possible and *heuristic methods* that possibly yield suboptimal colorings.

Exact methods for graph coloring include constraint programming (CP), propositional satisfiability (SAT), and integer linear programming (ILP) formulations (Burke et al., 2010; Glorian et al., 2019; Hebrard and Katsirelos, 2019b). Here, the problem is expressed in terms of constraints, propositional logic, or linear constraints over integer domains, respectively, and then solved by a general solver. Generally, these exact methods do not scale to graphs with more than a few thousand vertices, as these encodings become prohibitively large. In our experiments, the largest graph successfully colored by a SAT encoding had around 14000 vertices and was comparatively easy to color due to the graph's sparsity.

Heuristic graph coloring methods include various forms of greedy colorings combined with local search, especially tabu search, reducing the number of colors used by the greedy coloring (Blöchliger and Zufferey, 2008; Brélaz, 1979; Hao and Wu, 2012). Such heuristic methods scale to very large graphs and find good colorings for sparse graphs but struggle with large, dense graphs.

The CG:SHOP Challenge 2022[1] posed a problem that is reducible to graph coloring. Instances for the competition were crafted such that they are noticeably different from well-known graph coloring instances (Fekete et al., 2022) and yield graph coloring instances that are comparatively large and dense graphs. Since the aforementioned methods do not perform well on them, new approaches for graph coloring were developed, one of which is this chapter's subject.

In this chapter, we propose an approach which is hybrid between exact and heuristic techniques, following the SLIM method (see Section 2.4) Our idea is to enhance tabu search by applying SAT encodings locally. Our hybrid algorithm *GC-SLIM* incrementally improves a candidate coloring by repeatedly selecting small subgraphs (local instances) and coloring them optimally with a SAT solver. The problem solved by the SAT solver is a list coloring problem, where each vertex has a list of available colors. The lists ensure that the subgraph's coloring is consistent with the colors of the vertices outside the selected subgraph. GC-SLIM's most essential ingredients include strategies for *local instance selection*, the *SAT-based solution* of the local instance, and a technique called *chain propagation*.

GC-SLIM scales to dense graphs with several hundred thousand vertices and over 1.5 billion edges. Our experimental evaluation shows that our hybrid algorithm beats state-of-the-art methods on large, dense graphs.

6.1.1 Related Work

Since the work on graph coloring is extensive, we discuss only the most relevant work for this chapter. We refer to Sun's dissertation (Sun, 2018) for a more exhaustive survey on graph coloring algorithms.

Greedy colorings are the most common and easy heuristics for graph coloring. Given an ordering of the vertices, each vertex gets assigned the smallest color that avoids monochromatic edges in the given order. Different heuristics use different orderings. *DSatur* (Brélaz, 1979) is one of the most successful greedy heuristics, and we use it in our approach. DSatur always chooses as the next vertex one that is most constrained, i.e., one with the fewest colors available.

Tabu search has been successfully used for graph coloring. Most relevant to this paper is *Partialcol* (Blöchliger and Zufferey, 2008) that we discuss in more detail in Section 6.2.3.

Iterated-DSatur (I-DSatur) (Hebrard and Katsirelos, 2019a) is a SAT-based extension of DSatur that combines DSatur with extensive pre-processing and SAT-solving to a new

[1] https://cgshop.ibr.cs.tu-bs.de/competition/cg-shop-2022/

method that can compute optimal colorings for small graphs. Used as a heuristic, it scales to sparse graphs with several million vertices. I-DSatur adds a reordering mechanism to DSatur invoked whenever the current uncolored vertex v cannot be colored with any of the existing colors, i.e., the current partial k-coloring would become a partial $(k+1)$-coloring. At this point, I-DSatur tries to find a better coloring for all the vertices colored so far and v. If successful, no new color is required; if unsuccessful, the best lower bound on the number of required colors known to I-DSatur can be increased. The main difference between GC-SLIM and I-DSatur is that GC-SLIM tries to reduce the number of colors by improving several smaller local instances, while I-DSatur tries to find improvements for a single local instance that is as large as possible. The former scales better on dense graphs, while the latter performs better on sparse graphs, as we will further discuss in our experimental evaluation.

Large graphs yield a prohibitively large encoding size when the standard SAT encodings for graph coloring are used. Recently, a new approach based on clause learning has been proposed (Glorian et al., 2019; Hebrard and Katsirelos, 2019b), which can circumvent the size issue for many instances using a lazy approach, similar to ours in Chapter 5. This approach is also used in I-DSatur (Hebrard and Katsirelos, 2019a).

Much research in recent years focused on very large and sparse graphs. The advantage of sparse graphs is that they can often be colored with a small number of colors relative to their size and are easily reducible to smaller graphs. State-of-the-art approaches use these and other structural properties of sparse graphs to scale to graphs with millions of vertices (Lin et al., 2017; Rossi and Ahmed, 2014; Verma et al., 2015). We compare GC-SLIM to the most recent such algorithms FastColor (Lin et al., 2017) and I-DSatur (Hebrard and Katsirelos, 2019a).

The top three submissions to the CG:SHOP Challenge 2022 used different variations of the same idea. They perform local search guided by a conflict score, i.e., how often a vertex has been recolored (Crombez et al., 2022; Fontan et al., 2022; Spalding-Jamieson et al., 2022). This strategy performed better on the competition instances than other established local search strategies. We will further discuss this strategy in Section 6.2.3.

6.2 Preliminaries

6.2.1 Graphs and Colorings

We consider connected simple graphs G as introduced in Section 2.1.

Let $k \geq 1$ be an integer, we denote the set $\{1, \ldots, k\}$ by $[k]$. A *partial k-coloring* of a graph G is a mapping $c : V(c) \to [k]$ defined on a set $V(c) \subseteq V(G)$ such that $c(u) \neq c(v)$ for every $\{u, v\} \in E(G)$ with $u, v \in V(c)$. If $V(c) = V(G)$, then c is a *full k-coloring* or simply a *k-coloring* of G.[2] The *chromatic number* $\chi(G)$ of a graph G is the smallest k

[2]Some authors use the term *k-coloring* to refer to a mapping $c : V(G) \to [k]$ that allows monochromatic edges (edges $\{u, v\} \in E(G)$ with $c(u) = c(v)$) and call c a *proper* k-coloring if it has no monochromatic edges.

such that G has a k-coloring. We say a k-coloring is *optimal* if $k = \chi(G)$.

For a (partial) k-coloring c of G, we call the integers $[k]$ *colors* and the sets $c_\ell(G) = \{ v \in V(G) : c(v) = \ell \}$, $\ell \in [k]$, the *color classes* of c. Observe that each color class is an independent set of G and that color classes are pairwise disjoint. We also write $c_0(G) = V(G) \setminus V(c)$ for the set of *uncolored* vertices and write $c(v) = 0$ for a vertex $v \in V(G) \setminus V(c)$. We write c_ℓ, instead of $c_\ell(G)$, if G is clear from the context. Since its color classes uniquely determine a partial k-coloring, we will often specify a k-coloring this way. Further, we write $N_{G,c,\ell}(X) = N_G(X) \cap c_\ell(G)$, the *$\ell$-colored neighborhood*. Whenever G and c are clear from context, we drop the subscript and use $N_\ell(X)$. The *prevalence* of a color ℓ is $|c_\ell|$, and the prevalence of a color ℓ with respect to a vertex v is $|N_{G,c,\ell}(v)|$. Therefore, a least prevalent color of a k-coloring c in the neighborhood of v is $\arg\min_{\ell \in [k]} |N_{G,c,\ell}(v)|$.

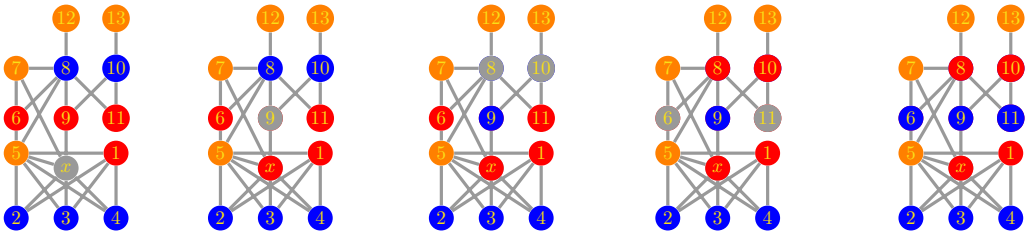
The *graph coloring problem* takes an undirected graph G as input; the task is to produce a coloring of G that uses the least possible number of colors. The decision version of the problem takes as input G and an integer k; the task is to decide whether G admits a k-coloring.

6.2.2 Minimum Partition into Plane Subgraphs Problem (MPPS)

The *Minimum Partition into Plane Subgraphs Problem (MPPS)* takes as an instance a geometric graph G, with vertices $V(G)$ represented by points in the plane, and edges $E(G)$ by straight-line connections between vertices. The task is to find a partitioning of E into as few classes $E_1, \ldots, E_k$ as possible, such that each subgraph G_i, with $V(G_i) = V(G)$ and $E(G_i) = E_i$, is plane.

In this chapter, we consider the MPPS problem in terms of graph coloring. There is a natural reduction from the MPPS problem to graph coloring, which reduces an MPPS instance G to the conflict graph G', containing a vertex for each line segment and where two vertices are adjacent if the corresponding line segments intersect. Evidently, G admits a partitioning into k plane subgraphs if and only if G' has a k-coloring.

6.2.3 Tabu Search



(a) Initial (b) Swap $x \to 9$ (c) $9 \to \{8, 10\}$ (d) $10 \to 11$, $8 \to 6$ (e) Color 6, Color 11

Figure 6.1: Tabu Search example that shows how vertex x is colored through a series of swaps. Note that for the last two graphs, two swaps are performed at once.

Tabu search is a very successful local search approach to graph coloring. We use Partialcol's search strategy (Blöchliger and Zufferey, 2008). Starting from a (non-optimal) $(k+1)$-coloring c of the given graph G, Partialcol selects a color $e \in [k+1]$ to eliminate. The vertices in c_e are then removed from c and considered uncolored, making c a partial k-coloring. Partialcol now tries to complete c and color the vertices in c_0 by performing *swaps*: For a partial k-coloring c, a vertex $v^* \in c_0$, and a color $\ell \in [k]$, a (color) *swap* of v^* to ℓ is obtained from c by setting $c(v^*) := \ell$, and $c(w) := 0$ for all $w \in N_\ell(v^*)$. The swap is a *p-swap* if $|N_\ell(v^*)| = p$. Let $u = |c_0|$ be the number of uncolored vertices before the swap, then $|c_0| = u + p - 1$ after a p-swap.

In each iteration, the algorithm performs a p-swap with smallest p. The choice of color ℓ for the p-swap is restricted by a *tabu* list for vertex v^*: a list of the colors assigned to v^* in the last few iterations. This mechanism ensures that vertices do not get re-assigned the same colors within a certain number of iterations and forces the algorithm to explore more of the search space. Figure 6.1 shows how a series of swaps can empty c_0.

Partialcol terminates if $c_0 = \emptyset$, in which case c is now a full k-coloring, or when it reaches a prescribed number of iterations. Usually, tabu search is run repeatedly, choosing different colors to eliminate.

Conflict Scores

The winning submissions (Crombez et al., 2022; Fontan et al., 2022; Spalding-Jamieson et al., 2022) to the CG:SHOP Challenge are based on heuristic algorithms that utilize a different selection criterion. While Partialcol picks the vertex that minimizes p of the necessary p-swap, these algorithms minimize a conflict score based on $q(v)$, defined as follows: whenever a vertex v is removed from c_0, i.e., whenever the vertex is colored, $q(v)$ is incremented. The different algorithms use different functions to calculate the conflict score. We follow the approach by Spalding-Jamieson et al. (2022) due to its simplicity: the solver picks a random vertex $v \in c_0$ and swaps it to the color ℓ that minimizes $\sum_{u \in N_\ell(v)} 1 + q(u)^2$.

6.3 SAT-Based Local Improvement for Graph Coloring

Our new SLIM (see Section 2.4) approach to graph coloring, *GC-SLIM*, tries to eliminate one color at a time in a fashion similar to Partialcol. Starting from a heuristically computed $(k+1)$-coloring, GC-SLIM selects a color $e \in [k+1]$, removes e from c, and tries to iteratively recolor subgraphs using a SAT solver until all vertices are colored, and c gives rise to a k-coloring.

We first discuss the core of every SLIM algorithm: a method to extract local instances such that their improvement eventually translates to an overall improvement. First, we discuss how we define local instances, i.e., we show how we can color subgraphs of G with a SAT solver while maintaining replacement consistency with the coloring of the

remaining graph. Then, we discuss how we find good local instances. We also discuss further additions to GC-SLIM that enhance its performance.

6.3.1 Local Instances and SAT

Let G be the input graph and c a partial k-coloring of G. Since G is too large to be encoded as a whole to SAT, we select a subset $X \subseteq V(G)$, based on a process described in the next subsection, limiting the size of X in terms of a budget parameter b. The goal is now to find a partial k-coloring for the induced subgraph G', with $V(G') = X$ and $E(G') = \{ \{u, v\} \in E(G) : u, v \in X \}$.

However, a newly found k-coloring of G' will, in general, not be compatible with the coloring c of the vertices outside X. We consider the vertices adjacent to X as extra constraints by defining the local instance in terms of the *list coloring* problem: Let L be a mapping that assigns each vertex $v \in X$ a set $L(v) \subseteq [k]$, called the *list* of v. Here in particular, we let

$$L(v) = [k] \setminus \{ c(u) : u \in N_G(v) \setminus X \}.$$

A *partial list coloring* of (G', L) is partial k-coloring c' of G' with the additional property that $c'(v) \in L(v)$ for each $v \in V(c')$. Let $c \circ c'$ denote the partial k-coloring obtained by composing c and c':

$$(c \circ c')(v) = \begin{cases} c(v) & \text{if } v \in V(c) \setminus X; \\ c'(v) & \text{if } v \in V(c'). \end{cases}$$

The following lemma provides an important link between colorings and list colorings.

Lemma 6.1. *Given a graph G and $X \subseteq V(G)$. Let c be a partial k-coloring of G, (G', L) be the local instance for X, and c' be a partial list coloring of (G', L). Then, $c \circ c'$ is a partial k-coloring of G.*

Proof. Consider an edge $\{u, v\} \in E(G)$. If $u, v \in V(c) \setminus X$, then $(c \circ c')(u) = c(u) \neq c(v) = (c \circ c')(v)$. If $u, v \in X \cap V(c')$, then $(c \circ c')(u) = c'(u) \neq c'(v) = (c \circ c')(v)$. If $u \in V(c) \setminus X$ and $v \in V(c')$, then $c(u) \notin L(v)$ since $u \in N_G(v)$, hence $(c \circ c')(u) = c(u) \neq c'(v) = (c \circ c')(v)$. □

We note in passing that the list coloring problem is a proper generalization of the graph coloring problem. For instance, graph coloring is fixed-parameter tractable in the graph's treewidth, while list coloring is W[1]-hard (Fellows et al., 2011).

Our general aim is to increase the number of colored vertices. Ideally, we would find a full k-coloring for (G', L). While this is often not possible, it turns out that it is still useful to obtain a partial list coloring c' of (G', L), which colors all previously uncolored vertices and minimizes the number of newly introduced uncolored vertices.

We achieve this by a slight tweak of the local instance. For all $v \in X \setminus c_0$, we add 0 to $L(v)$ and thereby allow them to become uncolored. The problem is now a minimization problem: find a partial list coloring c' for (G', L) that minimizes $|c'_0|$.

We encode the existence of a partial list coloring of G' that minimizes the number of uncolored vertices. To this end, for $r \leq |X|$, we define a propositional formula $F(G', L, r)$ which is satisfiable if and only if (G', L) has a partial list coloring c' where $|c'_0| \leq r$. We can minimize the number of uncolored vertices by solving $F(G', L, r)$ for different values of r.

The encoding requires one set of variables and two sets of clauses. For each $v \in X$ and $\ell \in L(v)$, the variable $c_{v,\ell}$ is true if and only if $v \in V(c')$ and $c'(v) = \ell$ or $v \notin V(c')$ and $\ell = 0$.

The first set of clauses encodes that each vertex $v \in X$ is assigned at least one color $\ell \in L(v)$ or is set to 0:

$$\bigwedge_{v \in X} \bigvee_{\ell \in L(v)} c_{v,\ell}.$$

Hence $c_{v,0}$ is true if and only if $v \notin V(c')$.

The second set of clauses encodes that adjacent vertices in G' must not have the same color:

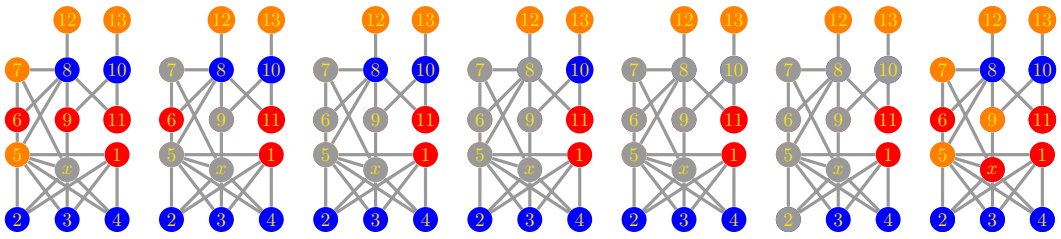
$$\bigwedge_{\substack{\{v,w\} \in E(G'), v<w, \\ \ell \in L(u) \cap L(v), \ell \neq 0}} \neg c_{v,\ell} \vee \neg c_{w,\ell}.$$

Note that $c_{v,0}$ and $c_{w,0}$ can both be true even if $\{v, w\} \in E(G)$. Finally, we use a *totalizer encoding* (Bailleux and Boufkhad, 2003) to express the cardinality constraint

$$|\{\, v \in X : c_{v,0} = \text{true} \,\}| \leq r.$$

The constraint adds $\Theta(|X| \cdot \log |X|)$ many variables and $\Theta(|X|^2)$ many clauses.

6.3.2 Local Instance Selection



(a) X_1 (b) X_2 (c) 5 first branch (d) 7 first branch (e) 9 first branch (f) 5 second branch, budget hit, X_3 (g) A possible solution.

Figure 6.2: Example for local instance selection with branching factor 2 and a budget of 8. The selected component is indicated by discolored (gray) vertices.

In this section, we describe how GC-SLIM constructs local instances for the SAT encoding described in the previous section. Let G be the input graph, and c a partial k-coloring

of G. Our goal is to select a suitable subset $X \subseteq V(G)$ that defines our local instance. The overall approach is to start at a single uncolored vertex and perform a breadth-first search among the least prevalent colors in the neighborhoods where the size of X is limited by a *budget* b and the breadth by a *branching factor* f.

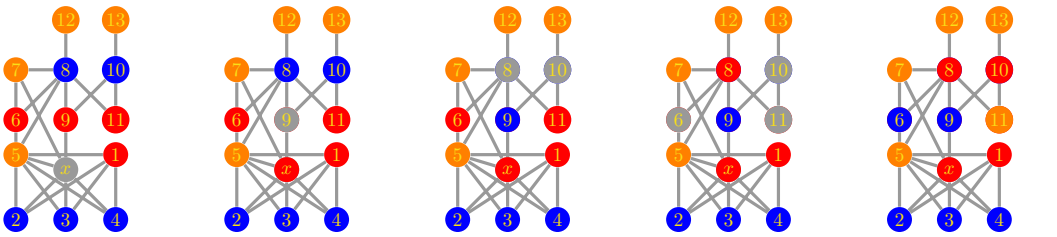
We first select an uncolored vertex $v^* \in c_0$. We initially put $X_0 = \emptyset$, $X_1 = \{v^*\}$ and continue computing a chain of sets $X_0 \subsetneq X_1 \subsetneq \cdots \subsetneq X_s$ as long as $|X_s| \leq b$. If no further addition is possible, we stop as we have found the set $X = X_s$.

Assume we have constructed X_i, $i \geq 1$. We now construct X_{i+1} by starting from $X_{i+1} := X_i$ and incrementally extending X_{i+1}. Let $S = X_i \setminus X_{i-1}$, for each $w \in S$, we find the smallest non-empty set $N_\ell(w), \ell \in [k]$ such that $N_\ell(w) \cap X_{i+1} = \emptyset$. If $|N_\ell(w)| + |X_{i+1}| \leq b$, we add $N_\ell(w)$ to X_{i+1} and in any case, we proceed to the next vertex in S. We repeat this step at most f times, i.e., for each vertex in S we add at most f colors from the neighborhood of the vertex to finish constructing X_{i+1}. We observe that $X \setminus V(c) = \{v^*\}$, i.e., v^* is the only uncolored vertex in X. Figure 6.2 illustrates local instance selection on a simple graph.

The goal of the budget b is to keep the size of the local instance small enough such that the SAT solver can solve it within the local timeout. In practice, the best budget varies greatly with the instance, so we automatically adjust it. Whenever a specified number of consecutive SAT solver calls time out, the budget is decreased, and conversely, whenever the same number of consecutive SAT solver calls return a result, the budget is increased.

We described the process such that we always expand X_{i+1} using the color ℓ such that $N_\ell(w)$ is minimal. Alternatively, we can also use the conflict score discussed in Section 6.2.3. We discuss both options in our experimental section.

6.3.3 Chain Propagation



(a) Initial (b) 1-Swap (c) 2-Swap (d) 2-Swap (e) Three 0-Swaps

Figure 6.3: Example of a chain propagation sequence coloring vertex x.

In this section, we describe *chain propagation*, which is a powerful technique that allows us to determine whether we can quickly color a given uncolored vertex v^* by using a *chain*, or sequence, of swaps and sequence, of swaps and propagating the impact of the swaps in the chain until hopefully finding a 0-swap. This concept is inspired by *s-chain* tabu search (Morgenstern, 1993), where chains up to a length s are explored, and by the consideration of a single *flat chain* in I-DSatur (Hebrard and Katsirelos, 2019a), where a

chain of 1-swaps is applied within a single iteration whenever available. Another way to view chain propagation is as a lookahead for the actions that Partialcol would perform.

We start with the set of uncolored vertices $U = \{v^*\}$, and try to empty this set by applying the following rules until either $U = \emptyset$ or no rule is applicable. Whenever we find a chain of swaps that empties U, we have found a chain that successfully colors v^*. Figure 6.3 illustrates these rules using our running example.

Rule 6.1 (0-swap)**.** *Take a vertex $w \in U$ and a color $\ell \in [k]$ such that $N_\ell(w) = \emptyset$. Swap the color of w to ℓ and remove w from U.*

The immediate goal of local search is finding a 0-swap, as a 0-swap decreases the number of uncolored vertices. The problem with 0-swaps is that they only consider the immediate neighborhood of the vertex.

Therefore, local search may miss possible 0-swaps if they are not included in our local instance or hidden behind larger swaps. We remedy this issue by extending chain propagation beyond 0-swaps, and exploring all chains of limited complexity with the goal of coloring v^*.

A slightly more elaborate case prevails when we apply the following rule multiple times, keeping the number of uncolored vertices constant, completed by a final application of Rule 6.1.

Rule 6.2 (1-swap)**.** *Take a vertex $w \in U$, such that for a color $\ell \in [k]$ and a vertex u we have $N_\ell(w) = \{u\}$. Swap the color of w to ℓ, make u uncolored, and replace w with u in set U.*

We call such a sequence of rule applications a *1-swap chain*, sometimes called a flat chain (Hebrard and Katsirelos, 2019a).

Even more powerful but also more costly is a *p-swap chain*, where $p > 1$ is a fixed constant. It uses the following generalization of Rule 6.2.

Rule 6.3 (p-swap)**.** *Take a vertex $w \in X$, such that for a color $\ell \in [k]$ we have $|N_\ell(v)| \leq p$. Swap the color of w to ℓ, make all the vertices in $N_\ell(w)$ uncolored, and replace w with $N_\ell(w)$ in X.*

Chain propagation explores the possible chains exhaustively. Bookkeeping is necessary to avoid re-applying the same series of swaps, as this leads to cycles, and consequently, chain propagation may not terminate. Further, we apply the rules in order, as it is faster to explore chains without p-swaps.

Two hyperparameters regulate the complexity of the chains. Since Rule 6.3 increases the number of uncolored vertices, it is the main factor for the complexity of the chains explored and, therefore, the main factor on the runtime of chain propagation. We limit the applications of the rule in two ways: (i) we limit p and thereby how much the number

of uncolored vertices can increase within one rule application, and (ii) we limit how often Rule 6.3 can be applied within one chain for the same reason. Together, the two hyperparameters regulate how much the number of uncolored vertices can increase within a single chain.

6.3.4 Putting it all together

Algorithm 6.1 combines the main ingredients of GC-SLIM that we have discussed. Initially, we compute a k-coloring c with a heuristic like DSatur and then repeatedly call GC-SLIM with different colors as the elimination goal. Each call either succeeds, reducing the number of colors, or fails, in which case we restore c to its initial state. For each call, we pick the least prevalent color we have not yet tried as the elimination goal, breaking ties arbitrarily.

Algorithm 6.1: GC-SLIM

```
1  iteration ← 1
2  Remove color e from c.
3  while c_0 ≠ ∅ and iteration < iteration_limit do
4  |  Update tabu list.
5  |  Pick vertex v* ∈ c_0 to color.
6  |  if chain propagation for v* is not successful then
7  |  |  m ← min_{ℓ∈[k]} |N_ℓ(v)|
8  |  |  I ← construct_local_instance(v*)
9  |  |  Changes ← call_sat(I, m, local_timeout)
10 |  |  if finding a list coloring I with at most m uncolored vertices fails then
11 |  |  |  Perform m-swap of v*.
12 |  |  |  Check if budget should decrease.
13 |  |  else
14 |  |  |  Check if budget should increase.
15 |  |  end
16 |  end
17 end
18 if c_0 = ∅ then
19 |  return c
20 else
21 |  Restore c. return Failed
22 end
```

In Algorithm 6.1, GC-SLIM starts with adjusting c according to the given elimination goal and then tries to complete c for a prescribed number of iterations. In each iteration, it picks a vertex v^* and first tries to color it using chain propagation. If this fails, the algorithm creates a local instance based on v^* and tries to color it using the SAT encoding. The number of uncolored vertices in the solution to the local instance is limited

to m, where m is the prevalence of the least prevalent color in the neighborhood of v^*. This limit of m ensures that GC-SLIM will not perform worse than a p-swap. If the local instance is (partially) colored successfully, GC-SLIM proceeds to the next vertex; otherwise, it defaults to a p-swap as Partialcol would perform.

The algorithm contains several hyperparameters, which we will discuss next.

6.3.5 Hyperparameters

The hyperparameters controlling Algorithm 6.1 are the *iteration limit*, the *local timeout for the SAT solver*, and the *choice of SAT solver*.

The iteration limit controls how much time the algorithm spends on eliminating a single color. Lower iteration limits cause shorter runtimes. Therefore, one can try to eliminate more colors in the same amount of time at the price of possibly missing some improvements: sometimes GC-SLIM will run many iterations with very few uncolored vertices until eventually finding the 0-swaps that complete the coloring. A low iteration limit will miss these improvements. Omitted in the listing is a mechanism that grants GC-SLIM an extra 10% of the iteration limit whenever the number of uncolored vertices decreases. Thus, GC-SLIM runs as long as it reduces the number of uncolored vertices, no matter the iteration limit.

The local timeout for the SAT solver follows a similar tradeoff. Lower values lead to quicker search space exploration by trying many different local instances, while larger values may discover new improvements. While the iteration limit regulates how often GC-SLIM generates a local instance, the local timeout strongly influences the budget for the local instances.

The SAT solver can also impact the performance of GC-SLIM, both in terms of memory usage and speed, and different solvers may perform very differently for different instances.

The hyperparameters from this section, together with the branching factor, budget, and p-limit for chain propagation as discussed above, control GC-SLIM. As we will discuss next, some further options can significantly impact GC-SLIM's performance. We will further explore this impact in our experiments.

6.3.6 Further Options

In this section, we discuss several minor options that can affect GC-SLIM's efficiency positively or negatively, depending on the instance. We will explore their effects further in the next section.

Prerun Tabu Search

Partialcol iterations are much faster than GC-SLIM iterations and can often reduce the number of colors quicker, while GC-SLIM can find improvements that Partialcol misses.

Running Partialcol for several iterations before starting GC-SLIM tries to take the best from both worlds.

Flexible Vertices

We say that a vertex $v \in V(G)$ is *flexible* with respect to a (partial) k-coloring c if $v \in V(c)$ and there is a color $\ell \in [k] \setminus \{c(v)\}$ such that $N_\ell(v) = \emptyset$. Thus, we can change the color of a flexible vertex and still have a (partial) k-coloring. We let $F_c \subseteq V(G)$ be the set of all vertices that are flexible w.r.t c. Flexible vertices provide an additional option when choosing a color: instead of simply choosing the least prevalent color, we redefine the prevalence of a color ℓ as $|c_\ell \setminus F_c|$ and the prevalence w.r.t. the neighborhood of a vertex analogously. This can lead to a more accurate estimation since flexible vertices allow immediate 0-swaps. This calculation is heuristical, as adjacent flexible vertices can block each other's color options, so we actually can only change the color of one of them.

Parallelization

GC-SLIM can run in parallel with minimal synchronization: each thread runs GC-SLIM, and whenever one thread successfully eliminates a color, GC-SLIM is restarted in each thread with the improved coloring. This introduces the new hyperparameter *thread count*. Generally, more threads are better as they enable faster search space exploration. Threads can either try to eliminate different colors, the same colors with different hyperparameter settings, or a mixture of both.

6.4 Experiments

The aim of this chapter is not to determine the fastest graph coloring method but to investigate how SAT/CP methods can be utilized for the coloring of large, dense graphs.

We conduct three sets of experiments, one that evaluates the impact of the different hyperparameter settings and, by extension, the different features of GC-SLIM. The second experiment compares GC-SLIM to the state-of-the-art graph coloring methods FastColor and I-DSatur[3]. In the last experiment, we look at GC-SLIM's performance on the whole set of CG:SHOP instances.

Setup

We ran our experiments on a cluster where each server had two Intel Xeon E5-2640 v4 CPUs with 10 cores to 2.4 GHz for the first and second experiment, and two AMD EPYC 7402 CPUs, each with 24 cores running at 2.8 GHz, for the last experiment. The servers ran on Ubuntu 18.04 and used gcc 7.5.0. The runs were limited to 64 GB of memory.

[3]Code is available at https://github.com/ASchidler/coloring/ and results at https://doi.org/10.5281/zenodo.7271869

We implemented our approach in C++ and used Glucose 3[4] and Cadical 1.5.0[5] as SAT solvers.

Our implementation of DSatur (Brélaz, 1979) computed the initial colorings. We compare GC-SLIM against FastColor[6] (Lin et al., 2017) and I-DSatur[7] (Hebrard and Katsirelos, 2019a), representing state-of-the-art methods for coloring massive graphs. FastColor and I-DSatur runs were limited to 128 GB, as lower memory limits were insufficient for large and dense graphs.

We used an initial budget of 300 for the local instance. Whenever three consecutive SAT solver calls timed out, we decreased the budget by 60 vertices; whenever three consecutive calls succeeded, we increased the budget by 60. In practice, the budget varied between 60 for very dense graphs and over 2000 for sparse graphs.

Instances

We used four sets of instances: (i) instances from the CG:SHOP 2022 competition (CG)[8], (ii) large random graphs (Random), (iii) large graphs from the Snap repository (Snap)[9] (Leskovec and Sosič, 2016; Leskovec and Krevl, 2014; Zitnik et al., 2018), and (iv) hard DIMACS instances (DIMACS)[10]. An overview of the number of vertices, edges, and densities are in Table 6.2.

The CG:SHOP competition[11] instances are very dense and have more structure than random graphs. The instances have up to 73000 vertices and 1.5 billion edges. We picked 10 instances of the 225 used in the competition for our second experiment: five instances are from the largest instances in the set with over 73000 vertices; the other five were chosen such that density and size vary.

We used random graphs as it was hard to find large benchmark instances that were not also very sparse. Therefore, we generated 14 Erdos-Renyi random graphs, which vary in size between 10000 and 100000 nodes and in density between 0.05 and 0.5.

The instances from the Snap repository[12] and 10th DIMACS instances[13] contain large graphs with up to several million vertices and have been used in related work (Hebrard and Katsirelos, 2019a; Lin et al., 2017). We preprocessed the instances by removing all vertices with degrees smaller than the lower bound on the chromatic number, determined in related work (Hebrard and Katsirelos, 2019a). We picked the 11 instances with more

[4] https://www.labri.fr/perso/lsimon/glucose/
[5] http://fmv.jku.at/cadical/
[6] https://lcs.ios.ac.cn/~caisw/Color.html
[7] https://bitbucket.org/gkatsi/gc-cdcl/
[8] https://cgshop.ibr.cs.tu-bs.de/competition/cg-shop-2022/
[9] https://snap.stanford.edu/snap/
[10] https://www.cc.gatech.edu/dimacs10/
[11] https://cgshop.ibr.cs.tu-bs.de/competition/cg-shop-2022/
[12] https://snap.stanford.edu/snap/
[13] https://www.cc.gatech.edu/dimacs10/

than 10000 and fewer than 280000 vertices from these preprocessed instances. The 280000 limit was due to memory constraints since we focused on supporting dense graphs, and adjacency matrices become very memory intensive for larger graphs. The mentioned sizes refer to preprocessed instances.

The DIMACS instances have been used in many papers for graph coloring and are included for reference, as GC-SLIM was not designed for such small instances. We used 10 instances that are considered hard (Hao and Wu, 2012).

6.4.1 Hyperparameter Impact

We explore how the different hyperparameter settings change the results in the first set of experiments. We use a base configuration and vary the setting of one parameter at a time. Each run for each instance was limited to seven hours.

As the base configuration, we use a local timeout of 10 seconds, 300 iterations, no chain propagation, a branching factor for local instances of 3, no multithreading, and no prerun tabu search. We count the instances for which a hyperparameter value finds the best coloring. We also count the instances where it does so uniquely, i.e., no other setting for this hyperparameter found an equally good or better coloring.

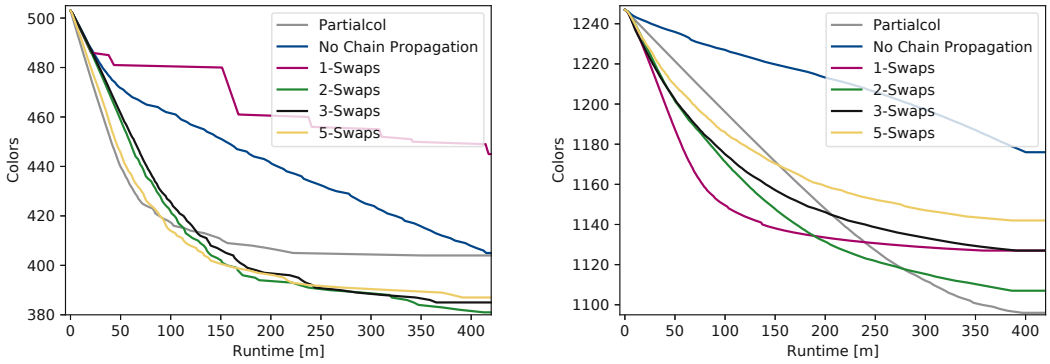
Local Timeout

We try different values for the local timeout: 5, 10 (default), 30, and 60 seconds. A timeout of 5 seconds found the best result for 37 of the 44 instances, where it reached the unique best result on 15 of the instances. The results quickly deteriorate with higher timeouts: while a timeout of 10 seconds found the unique best result for 4 instances, the higher timeouts achieved the same for only one instance.

A closer look at the number of SAT calls and the size of the local instances explains the results. While local instances contain 1511 vertices on average with a 5 seconds timeout, they only increase to 1610 vertices on average with a 60 seconds timeout. This is in stark contrast to the number of SAT calls which decreases from 13910 to 6764. Depending on the instance, 25% to 50% of the SAT calls eventually time out, leading to a significant decrease in the number of possible SAT calls with higher timeouts. Therefore, higher timeouts should be reserved for later stages when lower values fail to find improvements.

Chain Propagation

We use different limits for the maximum size of the swaps in the chains we propagate. We use values of 0 (default), 1, 2, 3, and 5. A limit of 2 achieves the overall best result, reaching the best result on 34 of the 44 instances and the unique best on 18 instances. Limiting the chains to 1-swaps or using no chain propagation at all performs very poorly. Higher limits can be beneficial for some instances, as, for example, a limit of 5 performs slightly better on the Snap instances. This indicates that higher limits might be beneficial for large and very sparse graphs.



(a) CG instance vispecn74166 (b) Random instance rnd_100000_10

Figure 6.4: Comparison of how many colors are eliminated over time with different swap limits for chain tracing. Time is on minutes on the x-axis and the number of colors is on the y-axis.

Figure 6.4 shows how chain propagation impacts GC-SLIM for large, dense instances. We can see that the number of improvements over time speeds up significantly and becomes comparable to Partialcol in terms of speed, sometimes surpassing it.

Impact of the SAT solver

We consider Glucose and Cadical as SAT solvers due to good results and their capability for incremental solving. Overall, Cadical achieves the unique best result on 19 of the 44 instances and Glucose on 10. This makes Cadical the best default choice, while Glucose may be better suited for individual instances. Glucose generally performs better on random instances and worse on the other instances in our experiments.

Flexible Vertices

Using flexible vertices does not give a clear advantage in the number of best results. Considering flexible vertices achieves the unique best result on 17 of the instances and not considering them on 14. While this does not seem like a clear advantage, the reduction in colors is significant, up to 100 colors for the instances where it performs better. For instances where it performs worse, the relative increase in the number of colors is never worse than 6. Considering flexible vertices performs consistently better for the CG instances.

Prerun tabu search

The benefits of running tabu search prior to GC-SLIM are very instance-specific. It achieves consistently better results on the DIMACS and Snap instances and worse results on the random and CG instances. This indicates that this configuration is beneficial

for small and sparse graphs. One possible explanation is that the tabu search iterations become slower for dense graphs, reducing the efficiency gain over GC-SLIM.

Conflict Score

Another option is using a conflict score to determine swaps and selecting local instances instead of simply picking the smallest colored neighborhood. This achieves the unique best result for 16 instances, while not using this option gives the unique best result on 15, making its benefits very instance specific. Using prerun tabu search with the conflict score gives the unique best result on 19 instances, in contrast to 17 instances where the basic configuration finds the unique best result. It performs consistently bad for the random instances, mixed for the DIMACS instances, and consistently good on the CG and snap instances.

Iteration Limit

We try iteration limits of 100, 300 (default), 500, 1000, and 5000. None of the limits performs significantly better than the others, with 500 and higher performing better and 5000 performing overall best with 7 uniquely best results. While 1000 iterations is a good default setting, different settings may perform better for different instances. Furthermore, higher iteration limits may be necessary if the number of colors is close to optimal, and it becomes harder to eliminate a color.

Branching

We try branching factors of 2, 3 (default), 5, 10, and 15. The overall best is a branching factor of 2, which achieves the best result on 31 of the 44 instances and finds 13 uniquely best instances. The results worsen with higher branching factors, except for the Snap instances, where a branching limit of 10 performs best. This matches the results for chain propagation, where higher limits also perform better for the Snap instances, suggesting that a search focused on breadth over depth may be a generally good strategy for sparse instances.

Initial Node Limit

When creating the local instance, we treat the initial vertex as a special case: instead of limiting the number of colors we choose, we limit the number of neighbors we add for the initial vertex. This system chooses more different colors if there are many low-prevalence colors and fewer if not. We try limits of 10, 25, 50 (default), 75, and 100.

Each limit leads to the best result on about 16 to 21 instances and a unique best result on 4 to 5 instances. Therefore, there is no discernible good default, and choosing the right value always depends on the instance at hand. A pattern similar to chain propagation and the branching limit emerges here: the lower the density, the better a higher initial limit, i.e., more focus on breadth, works.

Instance Set	TO	Chain	Flex	Iterations	Prerun	TS	Conflict	Branching	Initial	Solver
CG	5	2	N	1000	Y	Y	2	75	Cadical
DIMACS	5	2	N	1000	Y	N	3	10	Cadical
Random	10	2	Y	1000	N	N	3	50	Glucose
Snap	5	5	Y	5000	Y	Y	10	100	Cadical

Table 6.1: Best hyperparameter settings for different instance sets.

6.4.2 Comparison and Parallelism

We use the results from the first experiment to discern a *best configuration* for each of the four sets of instances; the configurations are shown in Table 6.1. In our comparison, we use different GC-SLIM configurations, FastColor, I-DSatur, and Partialcol. Each was run with a 24-hour time limit. We additionally run other methods and use their output as GC-SLIM's input, i.e., we *prerun* our own implementations of I-DSatur (IS) and/or Partialcol (PC). Partialcol runs either 7 hours alone or four hours together with I-DSatur. We also apply multithreading with 4 threads and varying the configuration parameters. The multithreading runs for only six hours and has twice the memory. The results are in Table 6.2.

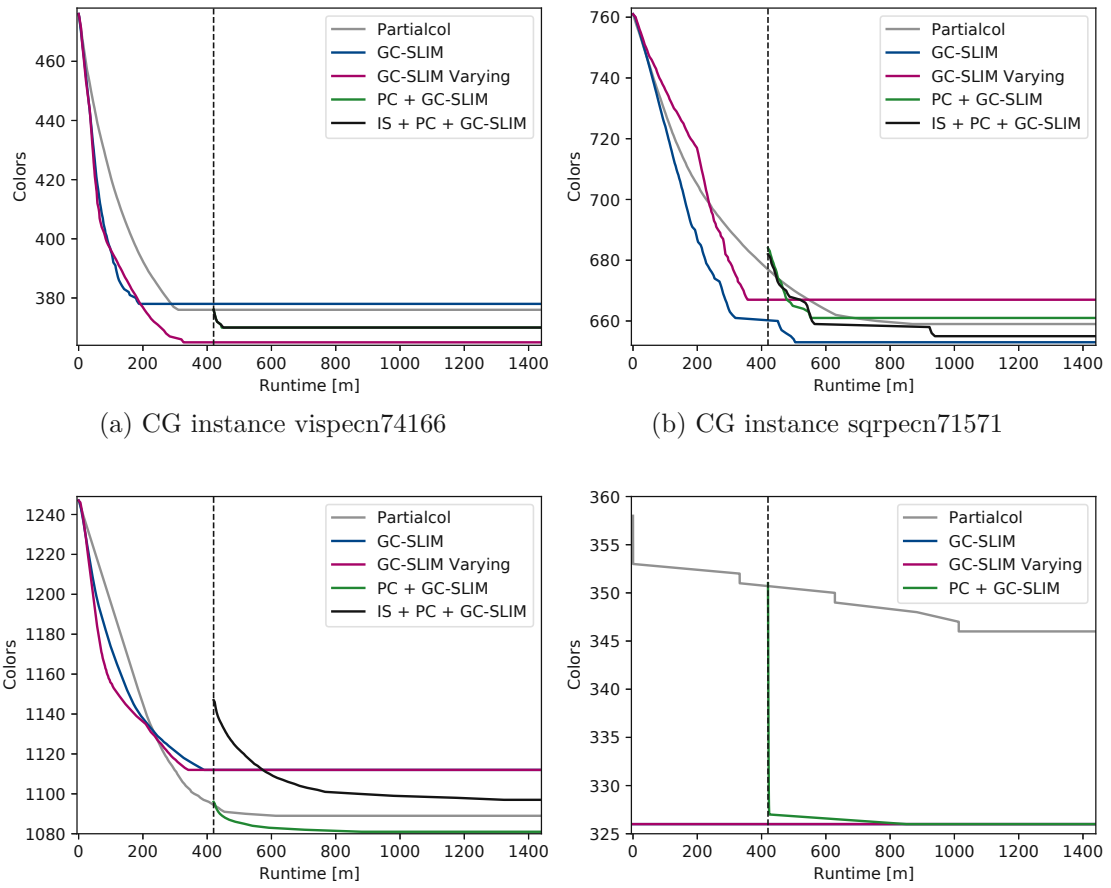
The results show how well GC-SLIM performs on large and dense graphs. On the CG and Random instances, it significantly outperforms FastColor and I-DSatur. Interestingly, for random instances, a Partialcol prerun performs consistently better than any other configuration. For the CG instances, using a portfolio of varied configurations performs best, and the overall best configuration performs comparatively poorly.

While GC-SLIM also outperforms both algorithms on the small DIMACS instances, it does not come close to reaching the best-known value on almost all instances. This shows that specialized algorithms for these small instances work better.

FastColor and I-DSatur shine on the Snap instances where they benefit from the structure of sparse graphs, which GC-SLIM does not particularly exploit. Still, FastColor, which performs better than I-DSatur, finds the best solution for six of the instances, as does the varied configuration. Note that our goal was not to compare FastColor and I-DSatur, but to compare these approaches to GC-SLIM on various graphs.

The GC-SLIM configurations perform very differently. Figure 6.5 shows the progression over time for selected instances.

Overall, the results show that a varied configuration is better than a single, tuned configuration. This finding is strengthened by the fact that multithreading incurs an overhead, as each thread has to restart once a color has been eliminated.



(a) CG instance vispecn74166

(b) CG instance sqrpecn71571

(c) Random instance rnd_100000_10

(d) Snap instance gplus_combined

Figure 6.5: Comparison of how many colors are eliminated by different configurations over time. The x-axis shows the time in minutes and the y-axis the number of colors. The vertical dotted line indicates when Partialcol preruns end and for multithreaded runs, the times are multiplied by the number of threads.

6.4.3 CG:SHOP Instances

The competition used 225 instances in total, separated in different classes, where instances into each class are created by the same process but with different sizes and densities. The instances were created such that then-available graph coloring solvers failed to produce good results. This is supported by the fact that these solvers would have placed very low in the competition (Fekete et al., 2022).

We started our submission by implementing Partialcol. This implementation was run for several days for each instance, and we only started implementing GC-SLIM, when Partialcol failed to find improvements. During that time, we varied and randomized every

parameter and decision in our implementation. This gives us the possibility to compare GC-SLIM to these long Partialcol runs in Figure 6.6. The figure shows that GC-SLIM is able to significantly improve the colorings, even after the long Partialcol runs.

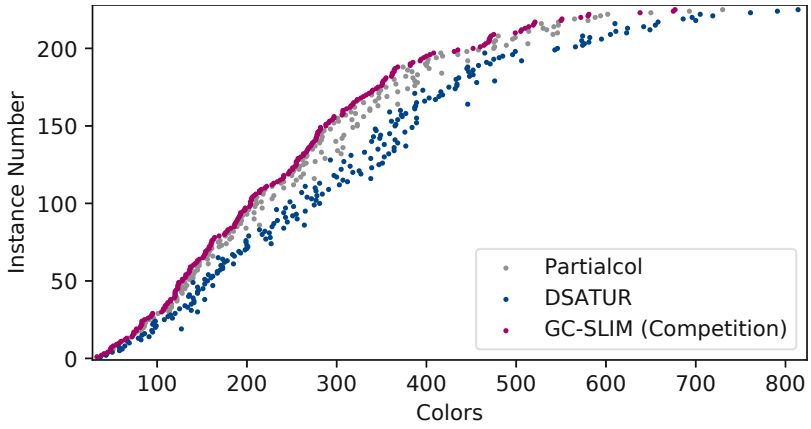


Figure 6.6: The CG:SHOP 2022 instance results. For each instance, we show the initial solution using DSatur, the long-term improved coloring using Partialcol, and the eventually submitted solution obtained using GC-SLIM.

We also have results from a more controlled experiment, where GC-SLIM runs for 24 hours using the best configuration. Figure 6.7 shows that the final GC-SLIM implementation achieves almost the same results as the results from the long runs of Partialcol and GC-SLIM during development as described above. Our final implementation achieves these results in a fraction of the time. This further shows how well GC-SLIM performs on these instances. Finally, Figure 6.8 shows the comparison between the best GC-SLIM run and the best results from the competition.

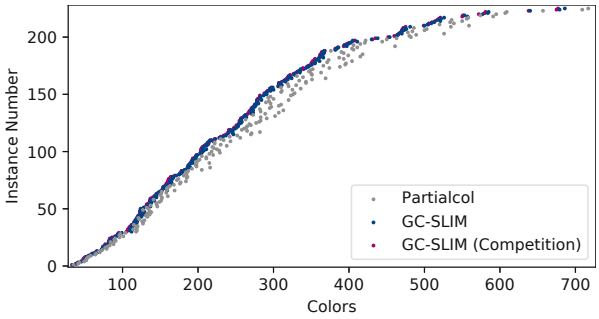


Figure 6.7: The results from a 24 hour run of GC-SLIM.

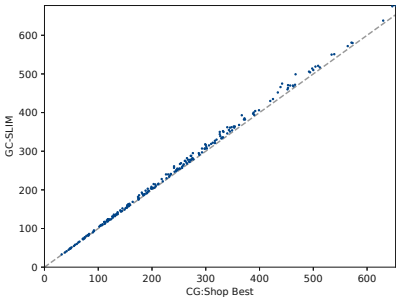


Figure 6.8: Comparison of a long GC-SLIM run with the best result from the CG:SHOP Challenge 2022.

6.5 Conclusion

With GC-SLIM, we have presented a new hybrid approach to graph coloring that enhances tabu search with SAT-based local improvement. Key elements of this combination are the selection method for local instances, the SAT-based solution for local instances, and chain propagation. Further improvements are due to hyperparameter tuning, the prerunning of tabu search, and different metrics for selecting vertices for color elimination. We also proposed and tested a parallel version of GC-SLIM. Our experimental evaluation shows that GC-SLIM complements existing methods and can find colorings with significantly fewer colors than other methods on large dense graphs.

GC-SLIM scales to instances that are out of reach for other SAT-based methods. While the largest instances in our experiments are far beyond the capabilities of SAT encodings, GC-SLIM also shows excellent performance compared to I-DSatur. While both heuristic SAT-based methods are comparable on sparse graphs, GC-SLIM significantly outperforms I-DSatur on dense graphs, making it the more versatile of the two approaches. GC-SLIM's performance shows how well the SLIM framework is suited for achieving scalability.

For future work, we see two main paths. The first one is improving the selection of local instances, as we expect a better criterion than using the least prevalent color. The other path is adapting GC-SLIM to more general graphs. We have seen that FastColor excels even for large random graphs and can handle even larger graphs. GC-SLIM can be adapted to handle large and sparse graphs. This would also require implementing features from FastColor and I-DSatur that exploit the structural properties of sparse graphs, as well as preprocessing. Integrating these features may also lead to a better method for local instance selection.

Table 6.2: Comparison of 24 hour runs. *DS* shows the DSatur run used as input for GC-SLIM. *Best* shows the best known result for the instance from the literature. For comparison, we list *Fast* Color, Iterated DSatur (*IS*), and our Partialcol (*PC*) implementation. GC-SLIM configurations start with *G*, *V* denotes varying parameters, *I* indicates a IDSatur Prerun, and *P* a Partialcol Prerun.

	Instance	$\|V\|$	$\|E\|$	Dens.	DS	Best	Fast	IS	PC	G	GV	GP	GI	GPI
CG	reecn56737	56k	308m	0.19	316	258	314	307	270	266	**265**	**265**	271	268
	reecn73116	73k	610m	0.23	433	349	426	438	372	371	**362**	367	375	365
	rvisp20601	20k	19m	0.09	99	81	93	115	**85**	87	86	**85**	87	99
	sqrp49981	49k	621m	0.5	353	270	345	389	280	274	**272**	278	281	275
	sqrp73525	73k	1560m	0.58	418	329	451	504	352	348	343	**341**	356	342
	sqrpecn71571	71k	1272m	0.5	761	630	752	826	659	656	667	659	667	**655**
	visp40191	40k	53m	0.07	125	104	119	130	**110**	113	**110**	**110**	115	125
	visp70702	70k	192m	0.08	186	145	184	186	159	166	**158**	159	168	186
	vispecn26914	26k	30m	0.08	264	176	234	243	195	207	**190**	**190**	195	191
	vispecn74166	74k	205m	0.07	476	332	444	450	376	378	**365**	367	382	366
Random	rnd_100000_10	100k	499m	0.1	1247	-	1256	1247	1089	1091	1112	**1081**	1096	1097
	rnd_100000_5	100k	250m	0.05	665	-	676	663	580	577	584	**571**	581	573
	rnd_10000_10	10k	4995k	0.1	169	-	171	170	145	144	145	**142**	145	**142**
	rnd_10000_25	10k	12m	0.25	399	-	398	396	333	337	340	**329**	339	330
	rnd_10000_5	10k	2499k	0.05	93	-	94	93	80	79	93	**78**	80	**78**
	rnd_10000_50	10k	24m	0.5	844	-	837	840	703	715	722	**698**	715	699
	rnd_25000_10	25k	31m	0.1	371	-	377	371	319	317	321	**314**	321	315
	rnd_25000_25	25k	29m	0.1	247	-	234	234	246	**195**	197	196	196	196
	rnd_25000_5	25k	15m	0.05	202	-	204	200	173	172	173	**170**	173	**170**
	rnd_25000_50	25k	156m	0.5	1896	-	1877	1899	1613	1630	1646	**1604**	1636	1605
	rnd_50000_10	50k	124m	0.1	679	-	687	677	585	586	592	**578**	590	579
	rnd_50000_5	50k	62m	0.05	363	-	371	364	314	314	316	**309**	316	310
	rnd_75000_10	75k	281m	0.1	968	-	975	968	839	842	852	**830**	847	834
	rnd_75000_5	75k	140m	0.05	517	-	525	517	449	447	452	**443**	450	444
Snap	G_n_pin_pout	99k	500k	0.0	6	5	6	6	6	**5**	**5**	6	**5**	6
	HR_edges	23k	328k	0.0	14	13	**13**	14	**13**	**13**	**13**	**13**	**13**	14
	WikiTalk	13k	728k	0.01	50	48	**48**	49	50	50	50	50	50	50
	artist_edges	18k	606k	0.0	23	19	**19**	21	20	20	20	20	20	23
	com-youtube	47k	670k	0.0	25	23	**23**	24	25	25	25	25	25	25
	gplus_combined	13k	6766k	0.08	326	326	327	337	346	**326**	**326**	**326**	**326**	**326**
	kron_g500-s-l16	17k	1495k	0.01	156	145	152	155	151	148	150	148	149	**147**
	p2p-Gnutella31	24k	100k	0.0	5	5	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**
	smallworld	100k	499k	0.0	8	6	7	8	**6**	**6**	**6**	**6**	**6**	8
	sx-stackoverflow	131k	10m	0.0	69	66	**66**	70	69	69	69	69	69	69
	wave	155k	1057k	0.0	9	7	8	9	8	8	**7**	8	8	9
DIMACS	C2000.5	2000	999k	0.5	208	145	205	207	168	170	173	**167**	170	**167**
	C2000.9	2000	1799k	0.9	555	408	534	547	429	425	441	429	**423**	428
	C4000.5	4000	4000k	0.5	377	259	376	376	312	313	317	**311**	312	**311**
	dsjc1000.1	1000	49k	0.1	26	20	25	25	**22**	**22**	**22**	**22**	**22**	26
	dsjc1000.5	1000	249k	0.5	116	83	112	114	93	93	97	**92**	93	**92**
	dsjc1000.9	1000	449k	0.9	303	222	287	297	232	229	241	231	**228**	231
	flat1000_50_0	1000	245k	0.49	114	50	111	112	**50**	69	88	**50**	**50**	114
	flat1000_60_0	1000	245k	0.49	116	60	111	113	90	76	115	90	**60**	78
	flat1000_76_0	1000	246k	0.49	114	81	111	112	92	93	94	92	**90**	114
	latin_square_10	900	307k	0.76	127	97	118	125	**102**	106	103	**102**	104	127
	r1000.1c	1000	485k	0.97	102	98	103	103	99	100	**98**	99	99	102
	r1000.5	1000	238k	0.48	242	234	**235**	240	245	236	**235**	239	236	236

CHAPTER 7

Decision Trees

7.1 Introduction

Decision trees are indispensable tools for the description, classification, and generalization of data (Larose, 2005; Murthy, 1998; Quinlan, 1986). Since decision trees are easy to understand, they are particularly attractive for providing interpretable models for the data they represent. This aspect has been emphasized in recent years (Darwiche and Hirth, 2020; Doshi-Velez and Kim, 2017; Goodman and Flaxman, 2017; Lipton, 2018; Monroe, 2018). In this context, one prefers trees of *low complexity*, which are trees of low depth (length of a longest path from the root to a leaf) and/or of small size (total number of nodes). A decision tree of low depth guarantees a low number of tests required to classify a sample (desired if tests are expensive or even intrusive (Podgorelec et al., 2002); a decision tree of small size limits the overall effort of understanding its decision-making and thus promotes transparency and interpretability. Decision trees are also used as part of larger implementations like functional synthesis, where the size of the decision tree can have a large impact on the runtime of the application (Golia et al., 2020, 2021). However, inducing decision trees of the lowest complexity is intractable (Hyafil and Rivest, 1976). Therefore, several exact methods (SAT and CP-based) have been proposed for that purpose (Bessiere et al., 2009; Narodytska et al., 2018; Avellaneda, 2020a; Janota and Morgado, 2020; Hu et al., 2020).

Scalability is often an issue with exact methods. For this reason, we propose a novel approach to learning decision trees of low complexity. We combine the scalability of heuristic methods with the strength of encoding-based exact methods following the principle of *SAT-based Local Improvement (SLIM)* (see Section 2.4).

Key to our approach is a suitable notion of a *local instance* which is based on using the decision tree's subtrees. Given a subtree of the decision tree, a SAT solver tries to find a subtree of lower complexity that correctly classifies all the samples the selected subtree

classifies correctly. Whenever the SAT solver is successful, the improved subtree can then replace the selected subtree. This, in turn, may lower the complexity of the whole decision tree. By adding weights and new classification categories to the local instance, we ensure that this replacement does not introduce misclassifications or increase the decision tree's complexity.

Because of the new classification categories and weights, a local instance poses a more complex classification problem. We show how SAT encodings can be generalized to accommodate non-binary weighted classification instances and propose a subtree selection strategy that avoids weights (Corollary 7.3). We further propose a new encoding based on a characterization of decision trees in terms of partitions (Theorem 7.4), which allows us to handle local instances of higher depth than it is possible with known encodings.

We establish a prototype implementation of our approach (DT-SLIM) and empirically evaluate it on data sets from the UCI Machine Learning Repository and prior work. Our experimental results are very encouraging: we can lower the complexity of heuristically obtained decision trees in almost all cases, in some cases significantly, without sacrificing much in terms of accuracy on unseen data.

We further apply DT-SLIM in a more realistic setting, where we apply pruning after reducing the complexity of the decision tree using DT-SLIM. Pruning is a method where parts of the decision tree are removed that will most likely not generalize well to unseen data. Our results are again very encouraging, showing SLIMed and pruned decision trees often outperform pruned decision trees in both complexity and accuracy.

This chapter is structured as follows. We will first give some necessary background and notation in Section 7.2. Next, we discuss the key contribution: how SLIM works in the context of decision trees (Section 7.3). In Section 7.4, we discuss how SAT encodings can induce decision trees, which includes a description of our new encoding DT_pb. Data reductions are necessary to solve many instances and are the topic of Section 7.5. We combine the previously introduced concepts in our approach DT-SLIM in Section 7.6 and give some practical considerations in Section 7.7. The last remaining piece in our approach is pruning, which is applied after SLIM and is briefly introduced in Section 7.8. Finally, we thoroughly test our approach against the state-of-the-art by a set of experiments in Section 7.9.

7.2 Preliminaries

7.2.1 Classification problems

A *classification instance* over a set F of *features* is a pair $\mathcal{I} = (E, c)$ where E is a finite set of *samples* and c is a mapping that assigns each sample $e \in E$ a *class* $c(e)$. $C_\mathcal{I} = \{\, c(e) : e \in E \,\}$ is the set of *classes* of the instance $\mathcal{I}$. A sample $e \in E$ is a mapping that assigns each feature $f \in F$ a *value* $e(f)$. The *domain* of a feature $f \in F$ is denoted by $D(f)$ and the feature's domain relative to E is the set $D_\mathcal{I}(f) = \{\, e(f) : e \in E \,\}$. We call $\mathcal{I}$ a *binary classification instance* if $|C_\mathcal{I}| = 2$. We assume w.l.o.g. that the classes

are ordered, particularly that we can determine a maximum over the set of classes. We say a feature is *numerical* if its domain contains only real numbers and otherwise we say it is *categorical*. We denote the set of numerical features by $N(F)$ and the categorical features by $R(F)$.

7.2.2 Decision Trees

A *decision tree* is a rooted binary tree T with node set $V(T)$, arc set $A(T)$, and root r; T_v denotes the subtree of T rooted at $v \in V(T)$. The *depth* $d_T(v)$ of a node v in T is the length (i.e., number of edges) of the path from T's root r to v; thus r has depth 0. T's depth $d(T)$ is the largest depth over all its nodes; T's *size* is its number of nodes.

Each internal node $v \in V(T)$ is labeled by a feature $\mathrm{feat}(v)$, a *threshold* $\mathrm{thresh}(v) \in D(\mathrm{feat}(v))$, and a *comparison operator* $\circ_v \in \{=, \leq\}$. If $\mathrm{feat}(v)$ is numerical, then $\circ_v$ is $\leq$; if $\mathrm{feat}(v)$ is categorical, then $\circ_v$ is $=$.[1] Each internal node has exactly two children, a left one and a right one. $L(T)$ denotes the set of T's leaves. We write $\mathrm{feat}(T) = \{\, \mathrm{feat}(v) : v \in V(T) \setminus L(T) \,\}$.

Consider a classification instance (E, c) over a set F of features and a decision tree T with $\mathrm{feat}(T) \subseteq F$. For each $v \in V(T)$, we define a set $E_T(v) \subseteq E$: if r is the root of T, we put $E_T(r) = E$; if v_1 is the left and v_2 the right child of a node u, we put $E_T(v_1) = \{\, e \in E_T(u) : e(\mathrm{feat}(u)) \circ_u \mathrm{thresh}(u) \,\}$ and $E_T(v_2) = E_T(u) \setminus E_T(v_1)$. We define the *classification* $c_{\mathcal{I},T}(\ell)$ of a leaf $\ell \in L(T)$, with respect to the classification instance $\mathcal{I} = (E, c)$ as

$$c_{\mathcal{I},T}(\ell) = \arg\max_{c \in C_{\mathcal{I}}} |\{\, e \in E_T(\ell) : c(e) = c \,\}|,$$

ties are broken arbitrarily. We say T *correctly classifies* a sample $e \in E$ if for the unique leaf $\ell \in L(T)$ with $e \in E_T(\ell)$ we have $c(e) = c_{\mathcal{I},T}(\ell)$. T correctly classifies $\mathcal{I} = (E, c)$ if it correctly classifies all the samples $e \in E$; in that case, we simply say T *is a decision tree for* $\mathcal{I}$.

We use a *training set* (E, c) to induce the decision tree and a *test set* (E', c') to determine the *accuracy* of the tree, where $E \cap E' = \emptyset$. The accuracy of a decision tree on a given instance is the fraction of samples from E' it correctly classifies. We use the term *training accuracy* if the accuracy is measured on E instead of E'.

7.2.3 Related Work

We limit the discussion in this section to the induction of optimal decision trees and refer the reader to the survey by Costa and Pedreira (2022) for a comprehensive summary of state-of-the-art methods for different decision tree induction tasks and algorithms. We discuss two different concepts of optimality: *complexity-optimal* decision trees and *accuracy-optimal* decision trees.

[1] Note that in practice, one can enforce any of the two comparison operator for a specific feature by preprocessing. I.e., ordering a categorical domain and using the order as values, or use non-real representations of the real values.

°C	Weather	Day	Length	Hike
29.1	Sunny	Sat	3 h	Yes
22.3	Thunder	Mon	2 h	No
21.5	Rain	Thu	1 h	Yes
23.7	Rain	Fri	3 h	Yes
14.3	Rain	Wed	4 h	No
14.7	Sunny	Tue	3 h	Yes

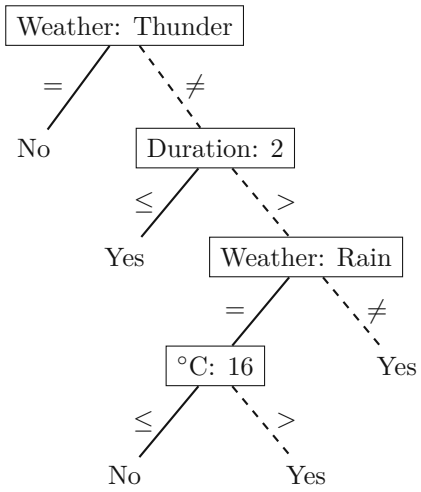


Figure 7.1: Left: A classification instance on when to hike and when not, with $|E| = 6$, $|F| = 4$, using numeric and categorical features. Right: a decision tree of depth 4 and size 9 for the instance on the left.

Complexity-optimal decision trees correctly classify all training set samples and have minimal depth and/or size. Hence, inducing complexity-optimal decision trees is a knowledge compilation task. Pruning is usually required for good accuracy on unseen data, as correctly classifying all training samples often causes overfitting. Inducing decision trees that correctly classify the training set and then pruning the decision tree follows the non-optimal approach of widely used decision tree heuristics like CART and C4.5 (Quinlan, 1993; Breiman et al., 1984). We discuss pruning in Section 7.8.

Several SAT-based methods for inducing complexity-optimal decision trees have been proposed. Initially, Bessiere et al. (2009) explored the use of a SAT encoding, but it performed poorly compared to their constraint programming approach. The first success was achieved by Narodytska et al. (2018): their proposed SAT encoding scaled to small instances. This success sparked a series of further research into SAT encodings for complexity-optimal decision trees (Hu et al., 2020; Avellaneda, 2020a; Janota and Morgado, 2020; Schidler and Szeider, 2021a; Ignatiev et al., 2021; Shati et al., 2021), and theoretical work (Ordyniak and Szeider, 2021). We discuss more details about these approaches in Section 7.4.

An accuracy-optimal decision tree has a constrained structure and maximizes the training accuracy. The structural constraints are different for different approaches, but most approaches impose a limit on the size and the depth of the tree. Maximizing the training accuracy of small trees follows the idea that good accuracy on the training samples correlates with good accuracy on unknown samples for small decision trees. Many approaches for inducing accuracy-optimal decision trees have been proposed: using SAT (Hu et al., 2020; Shati et al., 2021), constraint programming (Verhaeghe et al.,

2020b), mixed integer programming (Bertsimas and Dunn, 2017; Verwer and Zhang, 2019), and dynamic programming/branch & bound (Hu et al., 2019; Aglin et al., 2020; Lin et al., 2020; Demirovic et al., 2022).

Accuracy-optimal methods have the advantage that they can handle huge instances: when the size bound is very small, we can easily enumerate all possible decision trees. Hence, it is easy to find an accuracy-optimal decision tree of very small size, but the training accuracy of such a tree can be rather low. In contrast, complexity-optimal methods fail to induce any decision tree whenever the optimal decision tree becomes too large for the used method. Nonetheless, the accuracy of accuracy-optimal decision trees on unseen data depends on several instance-dependent factors. Even if an accurate and small decision tree exists, the accuracy-optimal approach has to find it among all possible decision trees with maximal training accuracy. We revisit this discussion in Section 7.9 when we compare the accuracy-optimal method BinOCT (Verwer and Zhang, 2019) to our method.

7.3 Local Improvement

Our approach follows the SLIM framework discussed in Section 2.4. Assume we are given a classification instance $\mathcal{I} = (E, c)$, which is too large for inducing a decision tree of smallest depth using an exact method such as a SAT encoding. We can use a heuristic method to compute a non-optimal decision tree T for $\mathcal{I}$. The idea of local improvement is to repeatedly select subtrees T' of T that induce a *local instance* $\mathcal{I}'$ that is small enough (possibly after further simplification and reduction) to be solved by an exact method. Once we have found a decision tree T'' for $\mathcal{I}'$ of smallest depth (or at least a depth that is smaller than the depth of T'), we can *replace* T' in T with the new T'', obtaining a new decision tree T^* for $\mathcal{I}$. We will focus on depth reduction for the theoretical part. Size reduction works analogously and will be discussed explicitly in Section 7.7.

We need to develop a suitable concept of a local instance to instantiate this general idea. This concept must guarantee replacement consistency: after replacing T' with T'', the new decision tree T^* correctly classifies all samples. This task becomes harder if we allow that some leaves of T' are internal nodes of T, i.e., $L(T') \setminus L(T) \neq \emptyset$. In this case, after replacement, the nodes in $L(T'') \setminus L(T)$ must be extended with parts of T to complete T^* into a decision tree for $\mathcal{I}$. The key to our solution is based on the introduction of new classes, as follows.

Let r' be the root of T', let $\ell_1, \ldots, \ell_k \in L(T') \setminus L(T)$ be those leaves of T' that are not leaves of T, and let $s = \max_{e \in E} c(e)$. The *local instance associated with* T' is the pair $\mathcal{I}' = (E', c')$ where $E' = E_T(r')$ and c' is the mapping defined by

$$c'(e) = \begin{cases} c'(e) = s + i & \text{if } e \in E_T(\ell_i) \text{ for some } 1 \leq i \leq k; \\ c'(e) = c(e) & \text{otherwise.} \end{cases}$$

Although $|E_T(r')|$ is independent of the subtree below r', the local instance size can be further reduced by the reduction methods discussed in Section 7.5. The effect of these reductions is highly dependent on the structure of T'.

Let T'' be any decision tree for $\mathcal{I}'$. Obviously, T'' will contain for each $i \in \{1, \ldots, k\}$ at least one leaf m such that $c_{\mathcal{I},T}(m) = \{s + i\}$. We call such a leaf m a *special leaf with classification* $s + i$.

To describe how the new decision tree T^* is constructed, we need the following operation on decision trees: Let T_1, T_2 be decision trees for the same instance, x a leaf of T_1 and y the root of T_2. The *extension of* T_1 *at* x *with* T_2 is the decision tree T_3 obtained from T_1 and T_2 by taking the vertex-disjoint union of the two trees and identifying x with y.

To construct T^*, we start with the decision tree T_0 obtained from T by deleting all descendants of the root r' of T'. From T_0 we obtain T_1 by extending it at r' by T''. Finally, from T_1 we obtain T^* by extending each special leaf m with classification $s + i$ with a new copy $T^m_{\ell_i}$ of T_{ℓ_i}. Figure 7.2 shows an example of this process.

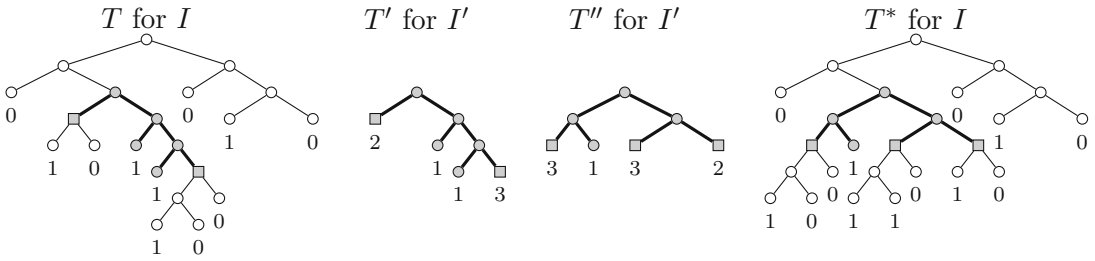


Figure 7.2: Local improvement workflow. The numbers indicate the leaves' classes; squares indicate special leaves.

The next theorem states that this replacement process is sound.

Theorem 7.1. *T^* correctly classifies $\mathcal{I}$.*

Proof. For showing the claim, let ℓ^* be any leaf of T^*. We will show that $|\{\, c(e) : e \in E_T(\ell^*)\,\}| \leq 1$. Let P be the unique path in T^* from the root of T^* to ℓ^*. We distinguish several cases.

Case 1: P does not run through r'. Hence ℓ^* is also a leaf of T. Since T correctly classifies $\mathcal{I}$ by assumption, $1 \geq |\{\, c(e) : e \in E_T(\ell^*)\,\}| = |\{\, c(e) : e \in E_{T^*}(\ell^*)\,\}|$.

Case 2: P runs through r'.

Subcase 2.1: ℓ^* is a leaf of T''. Since ℓ^* is also a leaf of T^*, it isn't a special leaf. Since T'' correctly classifies $\mathcal{I}'$, the latter implies that $c_{\mathcal{I},T^*}(\ell^*) = c_{\mathcal{I}',T''}(\ell^*)$ and, therefore, we have $|\{\, c(e) : e \in E_{T''}(\ell^*)\,\}| \leq 1$, hence again $|\{\, c(e) : e \in E_{T^*}(\ell^*)\,\}| \leq 1$.

Subcase 2.2: ℓ^* is not a leaf of T''. Consequently, P runs through a special leaf m of T''. Let $s + i$ be the classification of m. By construction, the subtree T^*_m of T^* is a

copy of the subtree T_{ℓ_i} of T, and the leaf ℓ^* of T^*_m is the copy of a leaf ℓ of T_{ℓ_i}. Since $c_{\mathcal{I}',T''}(m) = \{s+i\}$, we have $E_{T^*}(m) \subseteq E_T(\ell_i)$. Consequently $E_{T^*}(\ell^*) \subseteq E_T(\ell)$. Since T correctly classifies $\mathcal{I}$, $|\{\, c(e) : e \in E_T(\ell) \,\}| \leq 1$, and from $E_{T^*}(\ell^*) \subseteq E_T(\ell)$ we thus get $|\{\, c(e) : e \in E_{T^*}(\ell^*) \,\}| \leq 1$. □

Let us now turn to the question of decreasing the depth of the input decision tree T employing such a local replacement. This does not work out of the box: Even when $d(T'') < d(T')$ it still can happen that $d(T^*) > d(T)$ since the depth of a special leaf v of T'' of classification $s+i$ can be larger than the depth of the corresponding leaf ℓ_i of T', resulting in a larger depth of T^* if the subtree attached to v at T^* is large.

To overcome this problem, we enrich the local instance with additional information, defining a weighted version of the classification problem.

7.3.1 Weighted classification.

A *weighted classification instance* is a tuple $\mathcal{I}_w = (E, c, d)$ where $\mathcal{I} = (E, c)$ is a classification instance, and d is a mapping that assigns each $c \in C_{\mathcal{I}}(E)$ a positive integer $d(c)$. $\mathcal{I}$ and $\mathcal{I}_w$ have the same decision trees, just the depth for decision trees are defined differently for $\mathcal{I}$ and $\mathcal{I}_w$. Consider a decision tree T for $\mathcal{I}_w$. For a leaf ℓ of T with classification c (i.e., $c_{\mathcal{I},T}(\ell) = c$), we define the *weighted depth of* ℓ *in* T as $d_{w,T}(\ell) = d_T(\ell) + d(c)$. The *weighted depth* $d_w(T)$ of T is the maximum weighted depth over all its leaves.

We will show how locally decreasing the weighted depth of the weighted local instance within our local improvement setting allows us to decrease the depth of the global decision tree.

Let $\mathcal{I} = (E, C)$, $\mathcal{I}' = (E', C')$, T, T', T'' and T^* as above, and let $\mathcal{I}'_w = (E', C', d)$ denote the *weighted local instance*, where the weights for $c \in c'(E')$ are defined as follows: if $c = s + i$ then $d(c) = d(T_{\ell_i})$; if $c \leq s$, then $d(c) = 0$. We note that T' is a decision tree of the weighted local instance, and hence $d_w(T')$ is defined.

Theorem 7.2. *If* $d_w(T'') \leq d_w(T')$ *then* $d(T^*) \leq d(T)$.

Proof. Assume $d_w(T'') \leq d_w(T')$ and consider a longest path P^* in T^* between the root of T^* and a leaf ℓ^* of T^*. We denote the length of a path P^* by $\mathrm{len}(P^*)$

If P^* does not pass through r'', the root of T'', then it is also a root-to-leaf path of T, and so $d(T^*) = \mathrm{len}(P^*) \leq d(T)$, and the claim of is established.

It remains to consider the case where P^* passes through r''. Let P be a longest path in T which passes through r. Consequently, $\mathrm{len}(P) \leq d(T)$.

We can write $\mathrm{len}(P^*) = \mathrm{len}^*_0 + \mathrm{len}^*_1 + \mathrm{len}^*_2$ where len^*_0 is the length of the part of P^* between the root of T^* and r'', len^*_1 is the length of the part of P^* between r'' and a leaf of T'', and len^*_2 is the length of the part of P^* between a leaf of T'' and ℓ^*. It is possible that $\mathrm{len}^*_2 = 0$.

Similarly, we can write $\text{len}(P) = \text{len}_0 + \text{len}_1 + \text{len}_2$, where the three integers are defined similarly, using len_1 for the length of the part of P inside T'.

By the definition of the weights, we have $\text{len}_1 + \text{len}_2 = d_w(T')$, and $\text{len}_1^* + \text{len}_2^* = d_w(T'')$. Since $d_w(T'') \leq d_w(T')$, $\text{len}_1^* + \text{len}_2^* \leq \text{len}_1 + \text{len}_2$. Since $\text{len}_0^* = \text{len}_0$ by construction, this gives $d(T^*) = \text{len}(P^*) \leq \text{len}(P) \leq d(T)$, as claimed. □

We now identify a special case of Theorem 7.2 where we only need to consider the unweighted local instance and still ensure that $d(T^*) \leq d(T)$. Let us call a subtree T' of T to be *safe* if for every leaf ℓ of T' it holds that $d(T') \leq d_w(T') - d_T(\ell)$.

Corollary 7.3. *If T' is safe and $d(T'') \leq d(T')$ then $d(T^*) \leq d(T)$.*

Proof. Let T' be a safe subtree with $d(T'') \leq d(T')$. Let ℓ'' be a leaf of T'' with $d_{w,T''}(\ell'') = d_w(T'')$ and let c be the classification of ℓ'' in T''. From the definitions we get $d_w(T'') = d_{w,T''}(\ell'') = d(c) + d_{T''}(\ell'') \leq d(c) + d(T'') \leq d(c) + d(T')$. Since T' is safe, we have $d(T') \leq d_w(T') - d(c)$, and so we get from $d_w(T'') \leq d(c) + d(T')$ that $d_w(T'') \leq d(c) + d_w(T') - d(c) = d_w(T')$. By Theorem 7.2, $d(T^*) \leq d(T)$ follows. □

This concludes our concept of local improvement for decision trees. Next, we discuss how SAT-based decision tree induction works.

7.4 Encodings

The subsequent SAT encodings address the problem *Bounded-Depth Decision Tree Induction*: Given a classification instance $\mathcal{I}$ over a classification scheme, find a decision tree of minimal depth that correctly classifies $\mathcal{I}$. The encodings also allow for minimizing the size for a given depth.

A SAT approach to Bounded-Depth Decision Tree Induction, given a classification instance $\mathcal{I}$ and an integer d, entails formulating a propositional formula $F(\mathcal{I}, d)$, called the *encoding*, which is satisfiable if and only if there exists a decision tree of depth at most d that correctly classifies $\mathcal{I}$. The encoding is then tested by a SAT solver with increasing values for d until it becomes satisfiable, i.e., represents an (optimal) decision tree. Given a size z, we can extend this idea to a formula $F(\mathcal{I}, d, z)$, which is satisfiable if and only if there exists a decision tree of depth at most d that has at most z many nodes and correctly classifies $\mathcal{I}$. Introducing the size as a second minimization goal offers the possibility for different optimization strategies, prioritizing one or the other.

Different encodings have been proposed that mainly differ in two respects: (i) how they model the decision tree and (ii) how they assign thresholds to internal nodes. A common way to represent thresholds is via binary encodings, where each possible threshold value is represented by one binary variable. This requires $\sum_{f \in F} |D_{\mathcal{I}}(f)|$ many binary variables, resulting in large encodings for instances with large feature domains.

We will first discuss our encoding DT_pb, and then discuss the main ideas for encodings from related work.

7.4.1 DT_pb

The idea behind DT_pb is to formulate the problem in terms of partitions. This approach has been used successfully for different graph-related problems and was introduced by Heule and Szeider (2015) for clique-width computation. We first reformulate the problem of finding a decision tree of a given depth for a classification instance $\mathcal{I}$ by partitioning the set of samples (Theorem 7.4). We then directly convert this definition into a propositional CNF formula $F(\mathcal{I}, d)$, that is satisfiable if and only if a decision tree of depth d that correctly classifies $\mathcal{I}$ exists (Theorem 7.5).

Let $\mathcal{I} = (E, c)$ be a classification instance and $\mathcal{S} = (S_0, \ldots, S_d)$ a sequence of partitions of E. We refer to the classes as *groups* and partitions $S_0, \ldots, S_d$ as the *levels* $0, \ldots, d$. $\mathcal{S}$ is a *DT-sequence that correctly classifies* $\mathcal{I}$ if the following conditions hold.

DT1 $S_0 = \{E\}$.

DT2 For all $1 \leq m \leq d$ it holds that, for each group $g \in S_{m-1} \setminus S_m$, there are groups $g', g'' \in S_m$ with $g = g' \cup g''$, such that for some $f \in \text{feat}(E)$ and $t \in D_{\mathcal{I}}(f)$ it holds that $e'(f) \circ t$ for all $e' \in g'$, and there exists no $e'' \in g''$, such that $e''(f) \circ t$, where $\circ$ is $\leq$ if f is numerical and $=$ if f is categorical.

DT3 For each $g \in S_d$ it holds that for all $e_1, e_2 \in g$ we have $c(e_1) = c(e_2)$.

We note that the definition implies that S_m is a refinement of S_{m-1}, for $1 \leq m \leq d$. The definition of DT-sequences corresponds to the definition of $E_T(v)$ and it is easy to see that a decision tree of depth d can be converted into a DT-sequence of length $d + 1$; and the other way around. This leads us to the following theorem.

Theorem 7.4. *A classification instance can be classified by a decision tree of depth d if and only if it can be classified by a DT-sequence of length d.*

We encode a DT-sequence of length d for $\mathcal{I} = (E, c)$ where $E = \{e_1, \ldots, e_n\}$ and $F = \text{feat}(E) = \{f_1, \ldots, f_k\}$. The result of our encoding is a propositional formula $F(\mathcal{I}, d)$. This formula is satisfiable if and only if there exists a DT-sequence of length d, and therefore a decision tree of depth d that correctly classifies $\mathcal{I}$.

We use the variables

- $g_{i,j,m}$, for $1 \leq i < j < n$, $0 \leq m \leq d$, with the semantics that $g_{i,j,m}$ is true if and only if samples e_i and e_j are in the same group at level m,
- $s_{i,m,\ell}$ for $1 \leq i \leq n$, $0 \leq m < d$, $1 \leq \ell \leq k$, where $s_{i,m,\ell}$ is true if and only if the group of sample e_i at level m is split into two groups using feature f_ℓ.

- $l_{i,m}$ for $1 \leq i \leq n$, $0 \leq m < d$, that is true if and only if the sample e_i at level m is in the group where $e(f) \circ t$.

At the start, i.e., the first set of partitions in the sequence, all samples belong to the same group (DT1). We add the unary clauses

$$\bigwedge_{1 \leq i < j \leq n} g_{i,j,0}.$$

At the last level, all samples in one group must belong to the same class (DT3). We enforce this by adding the unary clauses

$$\bigwedge_{\substack{1 \leq i < j \leq n \\ c(e_i) \neq c(e_j)}} \neg g_{i,j,d}.$$

The remaining clauses enforce DT2. As S_m is a refinement of S_{m-1}, we have to ensure that samples in different groups cannot be in the same group at a higher level. We state this by adding the clauses

$$\bigwedge_{\substack{1 \leq i < j \leq n \\ 0 \leq m < d}} g_{i,j,m} \vee \neg g_{i,j,m+1}.$$

Next, we ensure that at each level m, for every sample i there exists a corresponding feature to satisfy. For this purpose, we add the clauses

$$\bigwedge_{\substack{1 \leq i \leq n \\ 0 \leq m < d}} \bigvee_{1 \leq \ell \leq k} s_{i,m,\ell},$$

and ensure consistency within groups by adding the clauses

$$\bigwedge_{\substack{1 \leq i < j \leq n, \\ 0 \leq m < d, \\ 1 \leq \ell \leq k}} \neg g_{i,j,m} \vee \neg s_{i,m,\ell} \vee s_{j,m,\ell}.$$

The next step in encoding DT2 is determining in which group the sample is in the next level. We distinguish between numerical and categorical features and add the following clauses. First, we handle categorical features and samples that agree on this feature value:

$$\bigwedge_{\substack{f_\ell \in R(F), \\ 1 \leq i,j \leq n, i \neq j \\ e_i(f) \neq e_j(f), \\ 0 \leq m < d}} \neg g_{i,j,m} \vee \neg s_{i,m,\ell} \vee \neg l_{i,m} \vee l_{j,m}.$$

Next, we handle those samples that disagree on the feature value:

$$\bigwedge_{\substack{f_\ell \in R(F), \\ 1 \leq i < j \leq n \\ e_i(f) = e_j(f), \\ 0 \leq m < d}} \neg g_{i,j,m} \vee \neg s_{i,m,\ell} \vee \neg l_{i,m} \vee \neg l_{j,m}.$$

For numerical features, we handle the sample based on the ordering of the feature value. We use the shorthand notation g^* as follows:

$$g^*_{i,j} = \begin{cases} g_{i,j} & \text{if } i < j; \\ g_{j,i} & \text{otherwise.} \end{cases}$$

We can now state that the split in two groups has to be consistent with the ordering of the feature values:

$$\bigwedge_{\substack{f_\ell \in N(F),\\ 1 \leq i,j \leq n\\ e_i(f) \leq e_j(f),\\ 0 \leq m < d}} \neg g^*_{i,j} \vee \neg s_{i,m,\ell} \vee \neg l_{j,m} \vee l_{i,m}.$$

For the case the two features have the same feature value, we add the following clauses:

$$\bigwedge_{\substack{f_\ell \in N(F),\\ 1 \leq i < j \leq n\\ e_i(f) = e_j(f),\\ 0 \leq m < d}} \neg g^*_{i,j} \vee \neg s_{i,m,\ell} \vee \neg l_{i,m} \vee l_{j,m}.$$

We finalize the encoding of DT2 by ensuring that the refinement into groups is according to the values in l:

$$\bigwedge_{\substack{1 \leq i < j \leq n,\\ 0 \leq m < d}} g_{i,j,m+1} \leftrightarrow (g_{i,j,m} \wedge (l_{i,m} \leftrightarrow l_{j,m})).$$

By construction of the formula and from Theorem 7.4 we obtain the following result.

Theorem 7.5. *$F(\mathcal{I}, d)$ is satisfiable if and only if there exists a decision tree of depth at most d that correctly classifies $\mathcal{I}$.*

The number of clauses in $F(\mathcal{I}, d)$ is $O(|E|^2 \cdot |\text{feat}(E)| \cdot d)$. While most of these clauses are short, the number of literals per clause is in $O(|\text{feat}(E)|)$. Therefore, the main factor determining the encoding size is the number of samples and not the depth.

Encoding Weights

We can encode weights with DT_pb by using different maximum depths for the different classes. Let $d_{\min}$ be the lowest weight among all classes. Given a class c, for all $e_i, e_j \in E$ such that $C(e_i) = c, C(e_j) \neq c$, we add the clauses $\neg g_{i,j,w}$, where $w = d - d(c) + d_{\min}$ is the allowed depth in regards to the weight of c.

Minimizing Size

We define the formula $F(\mathcal{I}, d, z)$ for limiting the size of the decision tree. The following modification does not guarantee that the decision tree has at most z nodes but that the decision tree has at most z leaves. For this purpose, we add variables z_i for $1 \leq i \leq n$, where z_i is true if e_i is the first sample in its group, i.e., e_i is not in any leaf that correctly classifies any e_j with $1 \leq j < i$. We express these semantics with the following clauses:

$$\bigwedge_{\substack{1 \leq j \leq n, \\ 0 \leq m < d}} z_j \vee \bigvee_{\substack{1 \leq i < j, \\ c(e_i) = c(e_j)}} g_{i,j,d}.$$

The number of leaves can then be restricted using a cardinality constraint encoding (see Section 3.4.1).

7.4.2 Further Encodings

We discuss four representative encodings from the literature.

The encoding by Narodytska et al. (2018) defines a propositional Formula $F(\mathcal{I}, z)$, which is satisfiable if and only if there exists a decision tree for $\mathcal{I}$ with at most z nodes. In the encoding, the structure of the z many nodes is not fixed, and the solver can arrange them as needed. Thresholds are encoded in a one-hot fashion: for each feature $f \in F$ and value $t \in D_{\mathcal{I}}(f)$, one variable is used to encode for each node v whenever $\text{feat}(v) = f$ and $\text{thresh}(v) = t$. While in every encoding, features and thresholds have to be matched to nodes, the dynamic structure of the decision tree introduces an extra degree of freedom. Consequently, finding a model for $F(\mathcal{I}, z)$ is comparatively hard, as our experiments show (Section 7.9). The advantage of this encoding is that it is the only one that minimizes the size of the decision tree without fixing the depth.

Avellaneda (2020a) achieves better scalability by fixing the structure of the decision tree as a perfect tree of a given depth. The thresholds are represented by a one-hot encoding, as in the encoding by Narodytska et al. (2018). While the fixed decision tree structure significantly increases performance, large domain sizes are a weak point of this encoding, as they require many extra variables and clauses. Further, the encoding requires $\Omega(2^d)$ many clauses, making it unsuitable for instances that require deep decision trees.

Shati et al. (2021) proposed a more efficient variation of the encoding. In this encoding, thresholds are encoded implicitly, and therefore, the size depends on the number of features but not on their domain sizes. This improves the performance but still suffers from the exponential dependency on the maximum depth of the tree.

Our encoding DT_pb circumvents both problems: it can handle deep decision trees and large domain sizes. Its weakness is instances with many samples where the other encodings perform better. This makes DT_pb particularly interesting for smaller instances that require complex decision trees, e.g., instances that require one very deep branch in a comparatively small decision tree.

We discuss the empirical evaluation of how different encodings perform in Section 7.9.

7.5 Instance Size Reduction

As we have seen in the previous section, the number of samples is one of the main factors determining encoding size. Encoding size, in turn, often correlates with the solving time. Hence, as we try to reduce the encoding size, and since we cannot reduce the required depth, we have to reduce the number of samples using the data reduction methods proposed subsequently. Additionally, our methods often remove features, further decreasing the encoding size.

The main idea behind our methods is that we can ignore duplicate samples: given two samples with the same values for all features, we can remove one of them and still obtain a classifying decision tree for the original instance. Hence, our novel data reduction methods strategically modify the samples to reduce the number of values in each feature's domain. This, in turn, increases the chance that two samples have the same values. The drawback of these data reduction methods is that we lose optimality: a lowest depth/smallest decision tree for the reduced instance might be deeper and/or larger than a lowest depth/smallest decision tree for the unreduced instance.

We formalize this idea using *active domains* $D^A_{\mathcal{I}}(f) \subseteq D_{\mathcal{I}}(f)$: the set of domain values of a feature f that we consider as possible thresholds. We first show how small active domains help to reduce the number of samples and then introduce our methods for finding small active domains.

7.5.1 Removing Samples

Given active domains $D^A_{\mathcal{I}}(f) \subseteq D_{\mathcal{I}}(f)$ for each feature f, we create the reduced instance $\mathcal{I}' = (E', c)$ as follows. First, we remove all features f with $D^A_{\mathcal{I}}(f) = \emptyset$. Next, we add for each numerical feature f the value $\max D_{\mathcal{I}}(f)$ to the feature's active domain. Finally, we add for each $e \in E$ a reduced sample e' to E' defined as

$$e'(f) := \begin{cases} e(f), & \text{if } e(f) \in D^A_{\mathcal{I}}(f) \\ \min\{\, t \in D^A_{\mathcal{I}}(f) : t > e(f) \,\}, & \text{if } f \text{ is numerical} \\ \text{unused}, & \text{otherwise.} \end{cases}$$

Here, the *unused* value for categorical features is a special value that is not used as a threshold when inducing a decision tree for $\mathcal{I}'$. Hence, at every node that uses the feature, the corresponding samples will always take the right path. The reduction in the number of samples is then automatically achieved whenever several samples in E map to one sample in E'.

Since in a decision tree for $\mathcal{I}'$ each sample's path is, by construction, the path that the unreduced sample would take, we observe the following:

Observation 7.6. *A decision tree that correctly classifies the reduced instance $\mathcal{I}'$ also correctly classifies $\mathcal{I}$.*

Next, we discuss our methods for computing small active domains.

7.5.2 Threshold Reduction

This method aims to reduce the active domain size of a single feature independent of the other features. This can already drastically reduce the encoding size for encodings that use a binary representation for the possible thresholds.

For each numerical feature $f \in F$, we denote by $E_f[i]$ the i-th sample in E sorted by the values for f. We now define

$$D^A_{\mathcal{I}}(f) := \{\, E_f[i](f) : 1 \leq i \leq |E| - 1, c(E_f[i]) \neq c(E_f[i+1]) \,\}.$$

In other words: for each numerical feature, we only consider those values that separate the ordered samples into partitions of the same class and ignore all other values.

The runtime of this method is in $O(|F| \cdot |E| \cdot \log |E|)$, as we have to sort the samples according to each feature. However, for most non-trivial instances, it holds that $|F| \ll |E|$. Although using these methods' active domains may lead to deeper and/or larger decision trees, we could not observe this sub-optimality in our experiments.

7.5.3 Feature Reduction

Classification instances with fewer features result in smaller SAT encodings, and it is, therefore, useful to remove features as long as we do not misclassify samples in the original instance. A *support set*, for an instance $\mathcal{I} = (E, c)$ over a feature set F, is a subset $F' \subseteq F$, such that for any two samples that have a different class, the support set contains at least one feature the two samples disagree on (Ibaraki et al., 2011; Ordyniak and Szeider, 2021). Removing features as long as the remaining features form a support set guarantees that the induced decision tree will still classify the original instance. However, we observed that it is rarely possible to remove features this way for instances with few features and large domains, as in this case, the set of features is often already a minimal support set.

We, therefore, refine the notion of support sets to *threshold support sets*, which gives us more potential for data reduction. A threshold support set S for $\mathcal{I}$ is a set of pairs (f, t) with $f \in F$ and $t \in D_{\mathcal{I}}(f)$. Further, for any two samples $e_1, e_2 \in E$ with $c(e_1) \neq c(e_2)$, S contains a tuple (f, t) such that t *separates* e_1 *and* e_2 *on feature* f, which means that, if f is numerical, then $e_1(f) \leq t < e_2(f)$ or $e_1(f) > t \geq e_2(f)$, and if f is categorical, then $e_1(f) = t \neq e_2(f)$ or $e_1(f) \neq t = e_2(f)$. Given a threshold support set S, we define the active domain as follows:

$$D^A_{\mathcal{I}}(f) := \{\, t : (f', t) \in S, f' = f \,\}.$$

We can compute a subset-minimal threshold support S set by starting with $S = \emptyset$, and by comparing all samples of different classes, adding a new pair to S whenever none of

the existing thresholds separate the two samples. Let $D = \max_{f \in F} |D_{\mathcal{I}}(f)|$, then this algorithm runs in time $O(|E|^2 \cdot |F| \cdot D)$ where the actual running time strongly depends on the instance. Which features and thresholds are chosen for S in what order can determine the quality of the result. We choose these elements randomly.

In general, threshold support sets result in smaller active domains compared to the simple threshold reduction discussed before. Nonetheless, it is useful to apply the simpler method, as it is usually much faster, and it speeds up and informs the second method, providing overall better threshold support sets.

We now combine all discussed components in our approach DT-SLIM, the topic of the next section.

7.6 DT-SLIM

In this section, we describe DT-SLIM, the overall algorithm that facilitates the SAT-based local improvement, building upon the theoretical results of Section 7.3, the encodings described in Section 7.4, and using the data reductions from the previous section.

As before, let T be a decision tree for a classification instance $\mathcal{I} = (E, c)$. Our aim is to select a subtree T' which gives rise to a local instance $\mathcal{I}' = (E', c')$ and a weighted local instance $\mathcal{I}'_w = (E', c', d)$. Since we will try to find a better decision tree T'' for $\mathcal{I}'_d$ with a SAT encoding, we need to select T' in such a way that the encoding size remains feasible.

We ensure that the local instance is small enough using a *budget* given by integer-valued parameters $\hat{d}$ and $\hat{c}$. The maximum depth of T' is limited by $\hat{d}$, thereby limiting one of the main factors for the encoding size. The other main factor, the number of samples, is limited by $\hat{c}$. Further, since the depth of T' gives us an upper bound, we search for an improved decision tree starting with $F(\mathcal{I}', d(T') - 1)$ and incrementally decrementing the depth limit. Therefore, with each successful SAT call, we improve upon T', even if we cannot find the optimal depth.

7.6.1 Operations

Given an internal node r' that is the root of T', we have three different operations we can perform:

Leaf select: Whenever $d(T_{r'}) \leq \hat{d}$ and $|E_T(r')| \leq \hat{c}$, we can simply create a local instance using all the samples in $E_T(r')$ as they are.

Leaf reduce: In case $d(T_{r'}) \leq \hat{d}$ and $|E_T(r')| > \hat{c}$, we need to reduce the number of samples in $E_T(r')$ using the reduction methods discussed in Section 7.5. We can use the local instance if the number of samples after the reduction does not exceed $\hat{c}$.

Mid reduce: In case both $d(T_{r'}) > \hat{d}$ and $|E_T(r')| > \hat{c}$, we need to select a subtree of $T_{r'}$ as T' in a way such that $d(T') \leq \hat{d}$. Afterwards, we introduce special leaves as discussed in Section 7.3, and modify $E_T(r')$ accordingly to obtain our local instance. Finally, we apply Section 7.5's data reductions to the local instance and thereby try to get the size of the local instance to conform to $\hat{c}$.

We try to apply the three operations in the above order, and whenever one is applicable, after performing the respective reductions, we do not need to try the others for the same r'.

7.6.2 Subtree Selection

We propose the following search strategy for local instances. We find a leaf ℓ of maximum depth and proceed on the path $P(\ell)$ from the root to ℓ. We try each internal node in turn as the root r' until we reach ℓ or we can apply leaf selection or reductions, at which point we know that no further improvements on this path are possible. We then proceed with the next leaf and never use a node in $P(\ell)$ as a possible root r' again. In general, we never choose a node for r' that has already been tried unsuccessfully in a prior iteration. Once we tried all nodes, we exhausted our options. Whenever we find an improvement, we discard $P(\ell)$ and start again, as there might be a different leaf of maximum depth after the improvement. The reason why we proceed from top to bottom is that changes in the top affect the path that samples take later on, and the top-down order requires fewer tries.

The structure of T' for leaf selection and leaf reduction is clear; for mid reduction we propose the following strategy. We start with T' consisting only of the root r' and then incrementally grow T' as follows: in each iteration, we add all nodes v from $V(T_{r'}) \setminus V(T')$ where $d(T_v)$ is maximal until we either hit $\hat{d}$ or after data reductions $|E'| > \hat{c}$. Maximizing the depth has two advantages: (i) T' is usually unbalanced, which creates more opportunities for depth reduction, and (ii) T' is always safe. We, therefore, do not have to use weights.

Data Reductions

The heuristic for computing threshold support sets discussed in Section 7.5.3 is quadratic in the number of samples and therefore becomes prohibitively slow when the number of samples becomes too high. We use the following observation in case the local instance has too many samples to apply the heuristic.

Observation 7.7. *For a given decision tree T' the set $\{ (feat(v), thresh(v)) : v \in V(T') \setminus L(T') \}$ is a threshold support set.*

While this method is significantly faster than the heuristic, it limits the possible improvements to a rearrangement of nodes. We, therefore, only use it when the local instance has too many samples, and the quadratic worst-case runtime is infeasible.

7.6.3 Algorithm

We can now formulate the entire algorithm, which we refer to as DT-SLIM($\mathcal{H}$), where $\mathcal{H}$ denotes the heuristic used to generate the initial decision tree T. The pseudo-code for DT-SLIM($\mathcal{H}$) is shown in Algorithm 7.1. It iteratively selects a leaf v with maximum depth, ignoring those in the set D of completed nodes.

The three operations are stated as an if-then-else construct. Leaf select in Line 8, leaf reduce in Line 12, and mid reduce in Line 17. The SAT solver call is triggered by *induce_dt* and instance size reduction by the call to *reduce*.

Algorithm 7.1: DT-SLIM($\mathcal{H}$)

Data: An instance $\mathcal{I} = (E, c)$, a decision tree $T = (V, A)$ with root r induced using $\mathcal{H}$, a depth limit $\hat{d}$, a sample limit $\hat{c}$.
Result: A new decision tree T^* with $d(T^*) \leq d(T)$.

```
1  D ← ∅
2  T* = T
3  while V(T*) \ D ≠ ∅ do
4      ℓ ← arg max_{ℓ∈L(T)\D} d(v)
5      P ← path(r, ℓ)
6      foreach v ∈ P do
7          if v ∉ D then
8              if d(T*_v) ≤ d̂ and |E_{T*}(v)| ≤ ĉ then
9                  I' ← (E_{T*}(v), c)
10                 T'' ← induce_dt(I', d(T_v) − 1)
11                 D ← D ∪ P
12             else if d(T*_v) ≤ d̂ then
13                 I' ← (E_{T*}(v), c)
14                 I' ← reduce(I')
15                 if |E'_{T*}(v)| ≤ ĉ then
16                     T'' ← induce_dt(I', d(T_v) − 1)
17             else
18                 T', E', c' ← construct_subtree(v)
19                 I' ← (E'_{T'}(v), c')
20                 I' ← reduce(I')
21                 if |E'_{T'}(v)| ≤ ĉ then
22                     T'' ← induce_dt(I', d(T') − 1)
23             if any T'' found then
24                 T* = replace(T*, T', T'')
25                 break
26         D ← D ∪ {v}
27     end
28 end
```

7.7 Implementation

So far, we have discussed our approach conceptually. In this section, we will discuss the practical aspects: which encoding we use, why we use it, and how we determine good values for the budget.

7.7.1 Choosing an Encoding

We tested all the encodings we discussed so far (see Section 7.9). Overall, the encoding by Shati et al. (2021) performed best. As expected from the asymptotics, at higher depths, when the encoding can handle only a limited number of samples, our encoding DT_pb starts to perform better. For this reason, we use a *Dynamic Encoding* in our implementation: up to depth 10, we use the encoding by Shati et al. (2021), and for higher depths, we use DT_pb.

7.7.2 Determining the Budget

With the encodings in hand, the next question is, which value to use for $\hat{d}$, $\hat{c}$, and at which depth should we switch encodings.

We used randomly generated subsets of increasing size from different instances (see Section 7.9 for details on the instances). We increased the size until the respective encoding failed.

Further, we also used a heuristically computed decision tree T and used different nodes $v \in V(T)$ to generate local instances from $E_T(v)$. Our results show that instead of an absolute sample limit, it works best to define for each $1 \leq m \leq \hat{d}$ a specific $\hat{c}_m$. With increasing m, the value of $\hat{c}_m$ decreases until it reaches 0 when $m > \hat{d}$.

The depth and number of samples provide a crude estimate for the solver's running time, as the solving time can greatly vary for the same depth and number of samples. Therefore, we picked our limits such that it is reasonable to expect that the solver will finish within a local timeout of five minutes. While choosing a higher local timeout allows DT-SLIM to try more improvements, it also means that the timeout will be exceeded very often, and, therefore, a lot of time is spent on SAT-solver runs that will not find a reduction in depth or size.

We use a sample limit of 25000 samples for low depths and incrementally lower it to 250 until the maximal depth of 14.

7.7.3 Size Minimization

Minimizing only the depth can severely increase the decision tree's size, as the solver balances the tree and often requires more nodes to achieve lower depths. The use of special leaves aggravates this problem, as any duplication of a special leaf effectively duplicates a whole subtree. We propose several options that directly address that issue and limit the size increase in a single DT-SLIM iteration.

The core technique uses the size minimization offered by all the encodings. This gives us the choice of size as an additional optimization goal, either before or after depth. Alternatively, we can enforce that the size does not increase without minimizing it. Special leaves can be constrained such that they can only occur once by stating that all samples whose class corresponds to the special leaf have to be classified by the same leaf. These options highlight another benefit of using a SAT-based approach: adding extra constraints that constrain the decision tree to fit application-specific requirements is easy.

These options directly tackle the size of a decision tree after a single DT-SLIM iteration. It is important to mention that this does not automatically lead to smaller decision trees after a full DT-SLIM run for several iterations. For example, duplicating special leaves may be beneficial, as it might enable more improvements in the subtrees, decreasing the overall size later on. Similar effects occur when minimizing the size first, where later improvements are made impossible. In fact, the only configuration that showed an overall benefit, compared to only minimizing the depth, was minimizing the size as a secondary objective. For this reason, we only consider the minimization of depth and depth then size in our experiments.

7.7.4 Unknown values

Sometimes $e(f)$ is unknown for a feature f and a sample e, i.e., e is a partial function. In such a case, we use the most prevalent value for f among the other samples. In case this causes inconsistencies, i.e., using the most prevalent value for sample e_i causes it to be equal to another sample e_j that was previously different, but $c(e_i) \neq c(e_j)$, we ignore this sample when inducing the decision tree.

7.8 Decision Tree Pruning

Decision tree pruning is well known to be an essential tool for generalization, i.e., high accuracy on unseen data. Surprisingly, it has not been considered in the literature in conjunction with SAT-based methods. We provide the first integration of SAT-based methods and state-of-the-art pruning techniques. We use established post-pruning methods, i.e., methods that are applied after inducing a classifying decision tree. We first briefly discuss these techniques and then their integration with SAT-based methods.

Pruning heuristically removes subtrees from a decision tree to achieve a better generalization. In other words, pruning is a method that trades an increased error rate on the training data for a lower estimated error rate on unseen data. Pruning of a decision tree T is performed relative to a specific non-leaf node $v \in V(T) \setminus L(T)$; there are two *pruning operations* illustrated in Figure 7.3: make v a leaf and thereby omit T_v (subtree-replacement), or, let u and w be the children of v in any order, replace v by u, thereby lifting T_u and removing T_w (subtree-lifting). We consider two different *pruning methods*. Each method defines its metric and checks the tree's nodes in turn. A pruning

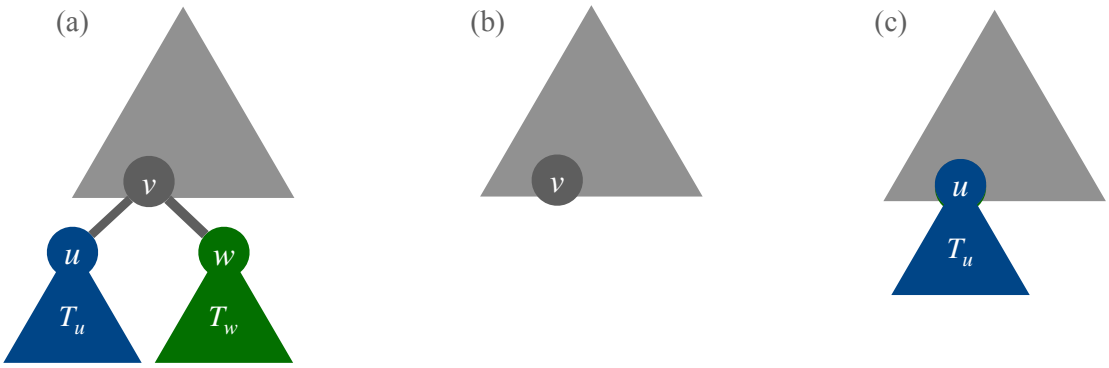


Figure 7.3: The illustration shows the original tree (a) and result of subtree-replacement (b) and subtree-lifting (c).

operation is then performed whenever it improves the metric. More specifically, we use the following methods.

Cost-complexity Pruning (Breiman et al., 1984)

We prune a node v if the pruning decreases $R_\alpha(v) = f_{T_v}(E_T(v), C) + \alpha|L(T_v)|$. The parameter α controls the trade-off between error rate and the number of leaves; the higher α, the smaller the resulting tree.

C4.5's Pruning (Quinlan, 1993)

We prune a node v if $e_T(v) < e'_T(v)$, where $e'_T(v)$ and $e_T(v)$ are the estimates for error rates before and after pruning, respectively, defined as follows:

$$e_T(v) = \left(f + \frac{z^2}{2E_T(v)} + z \cdot \sqrt{\frac{f}{E_T(v)} - \frac{f^2}{E_T(v)} + \frac{z^2}{4E_T(v)^2}}\right) \cdot \left(1 + \frac{z^2}{E_T(v)}\right)^{-1}$$

where f is the error rate after pruning and z is the percent point function value for confidence c (Tan et al., 2019). The error estimate for the current subtree is calculated by $e'_T(v) = \left(\sum_{\ell \in L(T_v)} (E_T(\ell) \cdot e_T(\ell))\right) \cdot (E_T(v))^{-1}$. C4.5's pruning uses another parameter m: all leaves $\ell \in L(T)$ with $|E_T(\ell)| < m$ are pruned immediately.

Cost-complexity pruning and C4.5's pruning come with hyperparameters α and (c, m) respectively. These hyperparameters have to be tuned to good values using for example cross validation. We assume that the hyperparameter settings are part of the input.

Pruning for SAT-based Methods

SAT-based methods offer several options for pruning. The straightforward approach is to induce a decision tree first and then prune it, using the same process as widely used

heuristics like C4.5 and CART. There is also the option to encode the pruning itself. Shati et al. (2021) showed that their SAT encoding, could be modified such that not the depth but the training accuracy is minimized. Yu et al. (2021) encoded cost-complexity pruning for decision lists and sets. The two ideas could, therefore, be combined to directly apply cost-complexity pruning for SAT encodings for inducing decision trees.

Pruning for DT-SLIM

The options for DT-SLIM look different. Since we run the SAT encoding many times on different parts of the decision tree and pruning decreases the training accuracy, the pruned decision tree does not necessarily correctly classify the instance. Non-classifying decision trees are generally not an issue to the SLIM approach, and DT-SLIM handles them by correctly classifying the samples that were correctly classified before, thereby not decreasing the training accuracy. Therefore, it is possible to integrate pruning directly into DT-SLIM, but it is not trivial, as there are different ways of handling non-classifying decision trees, applying pruning, handling hyperparameters, developing selection strategies, picking termination criteria, etc., which would exceed the scope of this chapter. We, therefore, keep to the established process of improving the decision tree, keeping it classifying, and then pruning it afterwards.

In the final section, we will evaluate DT-SLIM with and without pruning.

7.9 Experiments

Instances

We collected a comprehensive set of instances used in related work (Bessiere et al., 2009; Olson et al., 2017; Narodytska et al., 2018; Verwer and Zhang, 2019; Avellaneda, 2020a; Schidler and Szeider, 2021a), totaling 69 base instances. Eight of these instances come with a test set. We split the other 61 instances into five folds for cross validation, resulting in 313 instances in total. This section reports results for the 69 base instances, averaged over the folds. The instances provide a good variety in the number of samples, features, and classes, as Table 7.1 shows. Features vary from all-numerical to mixed and to all-categorical. As we will discuss below, SAT encodings were able to optimally solve 37 of the base instances, and we only consider the remaining 32 base instances for our DT-SLIM experiments.

Setup

Our implementation[2] uses Python 3.9 and PySAT 0.1.7[3] (Ignatiev et al., 2018). We use an adapted version of the Glucose 3.0[4] (Audemard and Simon, 2009) SAT solver, as it is

[2]https://github.com/ASchidler/decision_tree, results under https://doi.org/10.5281/zenodo.7271869
[3]https://pysathq.github.io/
[4]https://www.labri.fr/perso/lsimon/glucose/

Instance	$\|E\|$	$\|F\|$	$\|C\|$	Solved	Instance	$\|E\|$	$\|F\|$	$\|C\|$	Solved
anneal	798	38	5		iris	150	4	3	x
appendicitis	106	7	2	x	irish	500	5	2	x
audiology	200	70	24	x	kr-vs-kp	3196	36	2	
australian	690	14	2		letter recognition	20000	16	26	
backache	180	32	2	x	lymphography	148	18	4	x
balance-scale	625	4	3	x	magic04	19020	10	2	
banknote	1372	4	2	x	messidor	1151	19	2	
bank conv	4521	51	2		meteo	14	4	2	x
biodeg	1055	41	2		monks-1	124	6	2	x
breast-cancer-wisconsin	699	10	2	x	monks-2	169	6	2	x
cancer	683	9	2	x	monks-3	122	6	2	x
car	1728	6	2	x	mouse	70	5	2	x
ccdefault	30000	23	2		mushroom	8124	22	2	x
cleve	303	13	2	x	musk1	476	166	2	
cleveland	303	13	5	x	musk2	6598	166	2	
colic	368	22	2	x	mux6	128	6	2	x
compas-scores	11752	16	11		new-thyroid	215	5	3	x
corral	160	6	2	x	objectivity	1000	59	2	
german-credit	1000	24	2		pendigits	7494	16	10	
haberman	306	3	2	x	primary-tumor	339	17	21	x
hand posture	78095	36	5		promoters	106	58	2	x
heart-statlog	270	13	2	x	segment	2310	19	7	
heloc dataset v1	10459	23	2		seismic bumps	2584	18	2	
hepatitis	155	19	2	x	shuttleM	14500	9	2	x
hiv 1625	1625	8	2		soybean-large	307	35	19	x
hiv 746	746	8	2		spambase	4601	57	2	
hiv impens	947	8	2		spect	267	22	2	x
hiv schilling	3272	8	2		splice	3190	60	3	
house-votes-84	435	16	2	x	Statlog satellite	4435	36	6	
HTRU 2	17898	8	2		tic-tac-toe	958	9	2	
hungarian	294	13	2	x	vehicle	846	18	4	
hypothyroid	3163	25	2	x	wine	178	13	3	x
ida	60000	170	2		yeast	1484	8	10	
IndiansDiabetes	768	8	2		zoo	101	16	7	x
Ionosphere	351	34	2	x					

Table 7.1: A list of all instances considered for our experiments. *Solved* indicates that a depth optimal decision tree for the instance could be induced by a SAT encoding.

among PySAT's solvers that performed best and has a low memory profile. We induce heuristic initial decision trees using C4.5 implemented in Weka 3.8.5[5] (Frank et al., 2005) and CART implemented in scikit-learn 0.24.1[6] (Pedregosa et al., 2011). Additionally, we compare DT-SLIM to the mixed-integer programming tool BinOCT (Verwer and Zhang, 2019)[7]. We ran the experiments with a memory limit of 12 GB on servers with two Intel Xeon E5-2640 v4 CPUs running at 2.40 GHz and using Ubuntu 18.04. DT-SLIM was limited to 12 hours per run.

[5] https://www.cs.waikato.ac.nz/~ml/weka/
[6] https://scikit-learn.org/
[7] https://github.com/SiccoVerwer/binoct

7.9.1 Encodings

In our first experiment, we were interested in the performance of the encodings themselves. We can see in Table 7.2 that the encodings alone can already solve a majority of the instances. The encoding by Shati et al. (2021) performed best overall. Besides solving the most instances, it was also the fastest, solving many of the instances within the first minute. Our encoding DT_pb performed similarly to the encoding by Avellaneda (2020a). The encoding by Narodytska et al. (2018) solved the fewest instances but is the only encoding that minimizes the size. None of the instances had few samples but required a deep decision tree. Hence, the results do not fully reflect the usefulness of DT_pb within DT-SLIM.

	<1m		<5m		<10m		<1h		<3h		<6h	
Instance	S	P	S	P	S	P	S	P	S	P	S	P
Narodytska et al. (2018)	8	0	10	0	10	0	11	1	12	1	12	2
Avellaneda (2020a)	18	0	19	0	20	0	27	0	29	1	30	2
Avellaneda (2020a) + Size	16	0	17	0	18	0	21	1	25	2	26	2
DT_pb	15	0	20	0	20	0	26	0	29	1	30	2
DT_pb + Size	11	0	14	0	16	0	18	0	20	1	20	1
Shati et al. (2021)	23	0	26	0	27	0	31	0	35	0	36	2
Shati et al. (2021) + Size	18	0	20	0	20	1	24	1	26	2	27	2
Dynamic Encoding	23	0	27	0	27	0	32	0	35	0	35	2
Dynamic Encoding + Size	17	0	20	0	21	1	24	1	26	2	27	2

Table 7.2: Solved instances by encoding grouped by runtime. For each encoding, S (Solved) indicates the number of instances where all slices were solved and P (Partial) the number of instances where at least one but not all slices were solved. For each depth-minimizing encoding, the table also shows the variant, where the size is minimized as a secondary objective.

Adding size minimization, after finding the optimal depth, makes the problem considerably harder for each of the depth-minimizing encodings, severely decreasing the number of solved instances. Interestingly, with size minimization, the encodings by Avellaneda (2020a) and Shati et al. (2021) perform very similarly.

Depth Limit

To gauge the maximum depth any of the encodings can handle, we devised a small experiment. We incrementally created an artificial instance with $d+1$ samples, such that any decision tree needs exactly a branch with depth d to correctly classify the instance. Hence, it is the smallest instance that requires depth d. We gave each encoding up to one hour per instance. The encoding by Avellaneda (2020a) managed depth 11 in 635 seconds. Further, the encoding by Shati et al. (2021) was able to handle depth 12 in 2247 seconds. The encoding by Narodytska et al. (2018) had an advantage here, as although

the depth was high, the number of nodes was low. This encoding managed 29 nodes or depth 14 in 1263 seconds. Finally, DT_pb was able to handle depth 15 in 342 seconds. This shows the unique property of our encoding to handle high depths comparatively well, as long as the number of samples is low.

7.9.2 DT-SLIM

In this experiment, we tested how well DT-SLIM can improve the decision trees induced by the state-of-the-art decision tree heuristics, C4.5 and CART, and the results are listed in Tables 7.3 and 7.4. We consider plain DT-SLIM, where only the depth is minimized, and DT-SLIM + Size, where in each iteration, after trying to improve the depth, the solver gets the same time for improving the size. The tables are sorted by the number of samples. In both tables, it is noteworthy that for the largest initial decision trees, the size after running DT-SLIM is actually higher than without DT-SLIM.

Figures 7.4 and 7.5 show why this is the case, illustrated by two instances *HTRU 2* and *ida*, plotting the trajectories of depth, size, and accuracy over time. While the first figure shows one of the larger initial decision trees in our experiments, the second figure shows the decision tree for one of the largest instances in our set in terms of the number of samples. In both cases, the depth decreases monotonically, but the size goes up and down in waves. The reason for this is that DT-SLIM balances the tree since we minimize the depth (first), and might require more nodes than before to achieve this. This also means that after the improvement, the same number of samples is spread over more branches. This, in turn, gives DT-SLIM more chances for improvement. Hence, the number of nodes goes up when the subtrees are balanced and down once DT-SLIM improves the new branches.

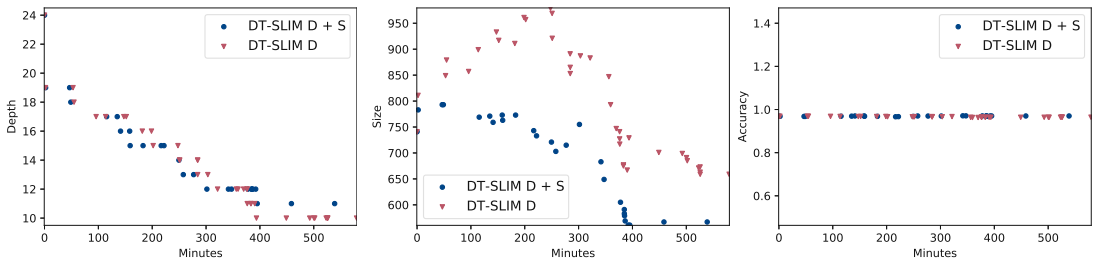


Figure 7.4: The DT-SLIM progression for instance *HTRU 2*.

In comparison, Figure 7.6 shows the progression for the largest initial decision tree. Here, DT-SLIM never got to the phase where it reduces the branches, as the run did not finish. Extending the runtime for this instance indeed results in a smaller decision tree, but given the large number of possible improvements given the large decision tree size, the runtime for an entire DT-SLIM run can extend over several days. The obvious solution, immediately searching the subtree after each improvement, turns out to perform worse, as it drastically increases the overall runtime.

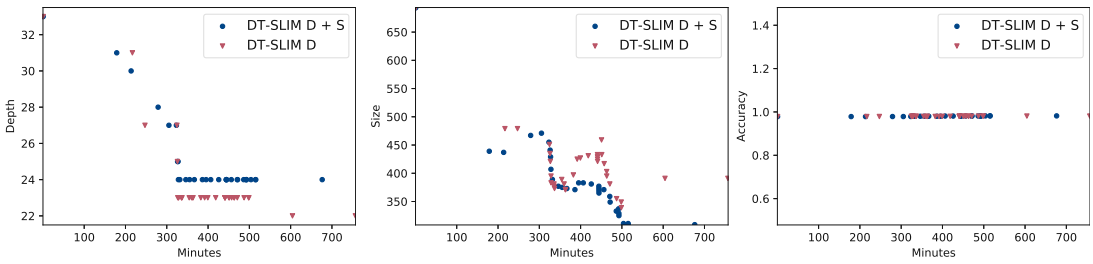


Figure 7.5: The DT-SLIM progression for instance *ida*.

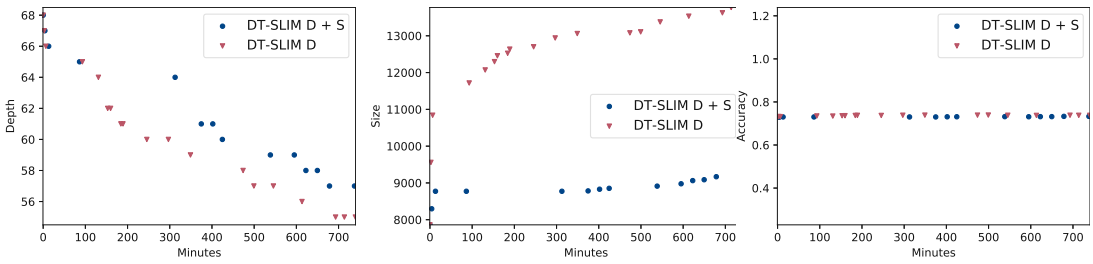


Figure 7.6: The DT-SLIM progression for instance *ccdefault*.

DT-SLIM works well for initial decision trees induced by C4.5. The depth is almost always significantly reduced, and even without minimizing the size, DT-SLIM is often able to reduce the size. With size minimization, DT-SLIM reduces the size of almost all C4.5 decision trees. In terms of accuracy, C4.5 performs a little better, although, on the instances where the accuracy decreases, it does so only slightly. Overall, the accuracy stays relatively stable, while depth and size are often greatly reduced.

The results are different for CART. When comparing CART and C4.5, the decision tree sizes are similar—with CART decision trees being slightly smaller—but the depth differs significantly. CART produces more balanced decision trees, hence, we expected that DT-SLIM would not work as well here since, as we have seen above, balancing the subtrees is an important mechanism, and CART trees do not give many opportunities for it. Indeed, while DT-SLIM still manages to improve the depth of almost all instances, the size reduction was not as successful. In terms of size, DT-SLIM can achieve better size, if it minimizes the depth and size. Particularly, again, on the largest instances, where DT-SLIM does not finish, CART achieves better results. Interestingly, although CART achieves, in general, a better accuracy on most instances, for the largest instances, the accuracy using DT-SLIM is better.

7.9.3 Pruning

C4.5 and CART both perform their pruning after inducing the decision tree, in a separate step. This allows us to insert DT-SLIM between the decision tree induction and the pruning. In this experiment, we compare the decision trees obtained from the heuristics with pruning to the decision trees obtained by running DT-SLIM before pruning. Pruning

Instance	C4.5 Size	C4.5 Depth	C4.5 Acc.	DT-SLIM(C4.5) Size	DT-SLIM(C4.5) Depth	DT-SLIM(C4.5) Acc.	DT-SLIM(C4.5)+Size Size	DT-SLIM(C4.5)+Size Depth	DT-SLIM(C4.5)+Size Acc.
musk1	67.4	18.6	**0.74**	51.4	6.0	0.54	**37.4**	**5.2**	0.61
australian	139.8	16.0	**0.82**	145.8	**7.0**	0.80	**114.6**	**7.0**	0.79
hiv 746	168.6	47.0	0.73	172.6	**24.0**	0.76	**152.6**	25.6	**0.79**
IndiansDiabetes	225.4	18.6	**0.71**	227.0	7.8	0.66	**195.4**	**7.4**	0.69
vehicle	238.2	34.8	**0.71**	272.2	9.4	0.68	**188.2**	**9.0**	0.68
anneal	123.0	23.0	0.72	109.0	**7.0**	0.80	**79.0**	**7.0**	**0.90**
hiv impens	183.4	21.6	0.83	201.0	**15.6**	0.84	**176.6**	16.2	**0.85**
tic-tac-toe	125.4	11.6	0.77	139.4	**7.0**	**0.80**	**115.8**	**7.0**	0.79
german-credit	349.4	22.6	**0.68**	344.6	10.0	0.66	**259.8**	**9.6**	0.67
objectivity	201.0	18.4	0.73	203.0	**8.4**	**0.74**	**157.0**	**8.4**	0.72
biodeg	195.4	19.6	**0.79**	187.8	**7.8**	0.78	**159.4**	8.8	0.78
messidor	370.2	32.4	**0.62**	376.6	**10.6**	0.59	**311.4**	11.6	0.60
yeast	831.0	26.8	**0.47**	925.0	13.4	**0.47**	**753.4**	**13.2**	**0.47**
hiv 1625	208.2	21.8	**0.87**	220.6	**11.6**	**0.87**	**176.6**	**11.6**	**0.87**
segment	125.4	13.6	**0.96**	121.4	**7.6**	0.95	**96.6**	8.2	**0.96**
seismic bumps	360.2	24.2	0.82	339.4	**11.4**	**0.83**	**274.6**	**11.4**	0.82
splice	253.0	14.4	0.91	308.2	10.0	0.90	**237.0**	**9.8**	**0.92**
kr-vs-kp	**87.4**	15.2	**0.98**	139.4	**10.0**	0.93	106.6	**10.0**	0.95
hiv schilling	539.0	30.8	0.73	601.8	**18.0**	**0.74**	**497.8**	**18.0**	**0.74**
Statlog satellite	610.6	32.8	**0.81**	675.8	12.8	0.79	**503.4**	**11.8**	0.79
bank conv	664.2	27.8	**0.87**	640.2	16.4	0.86	**502.2**	**16.0**	**0.87**
spambase	483.8	29.2	**0.88**	455.0	21.6	0.87	**364.6**	**21.2**	0.87
musk2	**233.8**	19.8	**0.66**	**233.8**	**11.6**	0.60	248.2	12.4	0.64
heloc dataset v1	**3107.4**	44.8	**0.66**	5406.2	**36.2**	0.65	4105.0	36.6	**0.66**
pendigits	507.0	17.0	**0.92**	457.0	11.0	**0.92**	**407.0**	**10.0**	0.91
compas-scores	**3846.2**	58.8	0.76	5487.4	**33.6**	**0.77**	4611.8	37.0	**0.77**
HTRU 2	702.2	23.2	**0.97**	661.8	**10.6**	0.96	**563.4**	**10.6**	0.96
magic04	**3120.2**	38.0	**0.82**	6029.8	**27.8**	**0.82**	4219.0	28.4	**0.82**
letter recognition	**3751.8**	27.0	**0.87**	5088.6	**22.0**	**0.87**	4243.4	22.2	**0.87**
ccdefault	**7800.2**	67.8	0.73	12036.2	**55.4**	**0.74**	9362.6	57.6	0.73
ida	693.0	33.0	**0.98**	391.0	**22.0**	**0.98**	**309.0**	24.0	**0.98**
hand posture	**3010.6**	36.4	**0.72**	3561.4	31.0	**0.72**	3226.2	**30.4**	**0.72**

Table 7.3: Size, depth, and accuracy of unpruned decision trees induced by C4.5 and improved by DT-SLIM(C4.5).

without DT-SLIM is performed by the respective heuristics implementations, and the pruning after DT-SLIM is performed by our implementations of C4.5 pruning and cost-complexity pruning.

Hyperparameters are required by both pruning methods discussed in Section 7.8, and we tune them as follows: in addition to the test set, we withhold an additional slice, the validation set[8]. We induce the decision tree using the remaining slices. Afterwards, we

[8]For the eight instances that provided a test set, the validation set is created by splitting off 25% of the samples.

Instance	CART Size	CART Depth	CART Acc.	DT-SLIM(CART) Size	DT-SLIM(CART) Depth	DT-SLIM(CART) Acc.	DT-SLIM(CART)+Size Size	DT-SLIM(CART)+Size Depth	DT-SLIM(CART)+Size Acc.
musk1	73.4	15.4	**0.70**	67.8	7.2	0.65	**58.6**	**6.8**	0.63
australian	143.4	12.0	**0.81**	147.4	**7.0**	0.79	**119.8**	**7.0**	**0.81**
hiv 746	**137.4**	14.6	**0.81**	172.2	**10.2**	0.75	151.4	**10.2**	0.79
IndiansDiabetes	217.8	15.0	**0.72**	221.4	**7.6**	0.66	**190.6**	7.8	0.66
vehicle	230.2	15.4	**0.71**	262.2	**8.4**	0.69	**209.8**	8.8	0.69
anneal	129.0	20.0	0.92	147.0	**7.0**	**0.93**	**101.0**	**7.0**	0.83
hiv impens	**170.2**	18.2	0.81	208.6	13.6	**0.82**	177.8	**13.4**	**0.82**
tic-tac-toe	**112.6**	10.6	0.79	131.8	**7.0**	**0.81**	115.0	**7.0**	0.80
german-credit	333.0	16.4	**0.68**	335.4	**8.6**	0.65	**293.4**	9.2	0.66
objectivity	207.4	19.2	**0.75**	195.8	**8.4**	0.72	**161.8**	8.8	0.73
biodeg	195.8	14.8	**0.80**	195.8	8.0	0.78	**153.8**	**7.4**	0.78
messidor	357.0	20.4	**0.62**	369.8	**10.4**	0.59	**292.2**	**10.4**	0.60
yeast	825.0	25.2	**0.50**	926.2	**11.8**	0.46	**820.6**	12.2	0.45
hiv 1625	188.2	14.0	**0.86**	228.2	**10.8**	0.84	**181.8**	11.0	**0.86**
segment	127.4	15.6	**0.96**	119.0	8.6	**0.96**	**107.0**	**8.0**	**0.96**
seismic bumps	351.4	19.8	**0.81**	336.6	9.8	**0.81**	**267.0**	**9.6**	0.80
splice	**245.0**	14.2	**0.92**	297.8	**9.8**	0.89	255.0	10.0	0.90
kr-vs-kp	**94.6**	14.6	**0.98**	163.8	**9.6**	0.94	133.8	**9.6**	0.93
hiv schilling	**502.6**	20.0	**0.77**	588.6	**16.6**	**0.77**	507.4	**16.6**	0.76
Statlog satellite	618.6	22.2	**0.80**	607.8	**11.6**	0.78	**527.0**	11.8	**0.80**
bank conv	**623.8**	26.8	**0.87**	807.4	**15.2**	0.86	658.2	16.2	0.86
spambase	465.8	32.0	**0.89**	463.0	**17.6**	0.87	**437.4**	19.2	0.87
musk2	226.6	18.4	**0.68**	225.8	**10.4**	0.67	**189.0**	11.4	0.67
heloc dataset v1	**2913.4**	32.4	0.61	4586.2	**23.0**	**0.62**	3560.2	23.8	**0.62**
pendigits	479.0	17.0	0.91	501.0	**10.0**	0.90	**373.0**	11.0	**0.92**
compas-scores	**3659.8**	32.4	0.76	5075.0	23.0	**0.77**	4340.6	**22.8**	**0.77**
HTRU 2	685.4	21.8	**0.97**	665.0	**9.8**	**0.97**	**578.2**	10.0	0.96
magic04	**3210.2**	34.4	**0.82**	5351.4	**24.0**	**0.82**	4025.8	25.2	**0.82**
letter recognition	**3897.0**	27.2	**0.88**	5077.8	**22.8**	0.87	4556.2	**22.8**	0.87
ccdefault	**7505.0**	42.4	0.73	10380.2	**34.0**	**0.74**	8954.6	34.4	0.73
ida	**659.0**	32.0	**0.99**	1261.0	23.0	**0.99**	1189.0	**20.0**	**0.99**
hand posture	**3226.6**	43.4	**0.70**	4041.0	**30.8**	**0.70**	3667.8	31.6	**0.70**

Table 7.4: Size, depth, and accuracy of unpruned decision trees induced by CART and improved by DT-SLIM(CART).

vary the hyperparameters and pick the values that have the highest accuracy on the validation set. In case of DT-SLIM, we first perform DT-SLIM and then pruning, and in this section, we give the results of the pruning method with the hyperparameter settings that performed best according to the validation set. The accuracy in the results is then again measured on the test set that was never visible to the heuristic, DT-SLIM, or pruning method during the whole process.

Table 7.5 shows the results for C4.5. We can see that the DT-SLIM's depth reductions are maintained even after pruning. In terms of size, the results are more mixed. Interestingly,

Instance	C4.5 Size	C4.5 Depth	C4.5 Acc.	DT-SLIM(C4.5) Size	DT-SLIM(C4.5) Depth	DT-SLIM(C4.5) Acc.	DT-SLIM(C4.5)+Size Size	DT-SLIM(C4.5)+Size Depth	DT-SLIM(C4.5)+Size Acc.
musk1	**30.2**	9.4	0.65	39.8	**5.2**	**0.69**	36.2	5.6	0.58
australian	16.6	6.2	0.84	16.2	**4.0**	0.84	**14.2**	4.6	**0.86**
hiv 746	36.2	16.0	**0.82**	**30.2**	9.0	0.75	32.6	**6.8**	0.76
IndiansDiabetes	**13.0**	**4.4**	**0.75**	16.6	**4.4**	0.73	27.0	5.0	0.73
vehicle	**61.4**	9.2	**0.71**	92.6	**7.0**	0.67	100.2	7.8	0.66
anneal	**53.0**	12.0	**0.98**	59.0	**6.0**	0.86	59.0	**6.0**	0.77
hiv impens	44.2	10.4	**0.85**	64.2	9.4	**0.85**	**43.8**	**8.0**	**0.85**
tic-tac-toe	43.4	6.0	**0.80**	**41.0**	5.2	0.76	**41.0**	**5.0**	0.72
german-credit	49.0	12.0	**0.73**	50.6	7.8	0.70	**48.2**	**5.8**	0.71
objectivity	47.0	7.6	**0.79**	**16.2**	4.2	0.76	23.0	**3.4**	0.78
biodeg	**29.8**	8.0	**0.82**	44.6	**5.0**	0.75	60.2	6.8	0.79
messidor	**31.0**	7.8	**0.64**	37.4	**5.4**	0.62	66.6	7.8	**0.64**
yeast	55.0	8.6	**0.56**	**33.4**	**6.0**	0.53	63.8	7.4	0.53
hiv 1625	53.4	12.0	**0.84**	**39.8**	**6.8**	0.83	60.6	8.2	**0.84**
segment	**69.8**	11.6	**0.96**	92.6	**7.2**	0.95	81.0	**7.2**	**0.96**
seismic bumps	**33.8**	6.6	0.91	39.4	**4.8**	**0.93**	55.8	6.2	**0.93**
splice	56.6	10.4	**0.94**	**33.0**	**6.8**	0.93	40.6	7.0	0.93
kr-vs-kp	**41.4**	10.4	**0.97**	54.2	**7.8**	0.94	61.0	8.6	0.93
hiv schilling	57.0	8.6	**0.87**	**21.0**	**4.2**	**0.87**	27.0	4.8	**0.87**
Statlog satellite	123.8	12.8	**0.82**	186.6	10.2	0.79	**115.0**	**9.8**	0.81
bank conv	**35.0**	7.8	**0.90**	76.2	**7.0**	**0.90**	45.4	7.6	**0.90**
spambase	**64.2**	12.8	**0.89**	91.4	**10.6**	**0.89**	89.4	**10.6**	0.88
musk2	38.2	9.2	0.81	40.6	5.0	0.77	**28.2**	**3.6**	**0.83**
heloc dataset v1	92.2	13.2	**0.74**	66.2	**9.8**	0.73	**44.2**	**9.8**	**0.74**
pendigits	325.0	15.0	**0.92**	317.0	**11.0**	0.91	**281.0**	**11.0**	**0.92**
compas-scores	**101.8**	**11.8**	**0.81**	445.0	16.4	**0.81**	117.0	12.2	**0.81**
HTRU 2	**20.6**	6.0	**0.98**	25.0	**5.4**	**0.98**	49.4	7.4	**0.98**
magic04	**325.8**	**15.6**	**0.85**	659.8	20.6	0.84	445.0	20.8	**0.85**
letter recognition	**2721.8**	24.8	**0.86**	4185.8	21.6	0.85	3555.0	**21.0**	0.85
ccdefault	78.2	9.8	**0.82**	17.8	5.0	**0.82**	**15.8**	**4.8**	**0.82**
ida	179.0	21.0	**0.99**	249.0	**16.0**	**0.99**	**169.0**	20.0	**0.99**
hand posture	**713.0**	21.8	**0.68**	788.2	**19.2**	0.67	717.4	19.4	0.67

Table 7.5: Size, depth, and accuracy of pruned decision trees induced by C4.5 and processed by DT-SLIM(C4.5).

Instance	CART Size	CART Depth	CART Acc.	DT-SLIM Depth Size	DT-SLIM Depth Depth	DT-SLIM Depth Acc.	DT-SLIM Depth+Size Size	DT-SLIM Depth+Size Depth	DT-SLIM Depth+Size Acc.
musk1	43.4	8.6	0.62	33.4	4.2	**0.63**	**27.0**	**4.0**	0.62
australian	35.0	5.4	0.82	18.6	4.0	0.83	**10.2**	**3.4**	**0.85**
hiv 746	41.0	7.2	0.77	36.6	7.0	0.77	**21.8**	**6.2**	**0.79**
IndiansDiabetes	53.4	9.6	**0.74**	28.6	4.6	0.71	**21.4**	**4.4**	**0.74**
vehicle	101.4	12.0	0.68	85.8	6.6	**0.69**	**79.4**	**6.4**	**0.69**
anneal	123.0	19.0	**0.91**	**41.0**	**6.0**	0.87	69.0	7.0	**0.91**
hiv impens	**75.0**	10.0	0.85	75.4	**7.8**	**0.86**	82.2	8.0	0.85
tic-tac-toe	**53.4**	6.6	0.76	59.8	6.6	0.79	57.4	**6.4**	**0.80**
german-credit	59.4	9.4	0.72	31.4	**5.0**	**0.73**	**29.8**	6.0	0.71
objectivity	43.4	8.2	**0.80**	42.6	5.2	0.78	**14.6**	**3.6**	0.79
biodeg	87.8	11.2	**0.80**	54.2	**5.6**	0.79	**50.6**	6.0	**0.80**
messidor	**34.2**	8.6	**0.65**	176.6	9.0	0.63	111.8	**6.8**	0.63
yeast	**45.8**	8.4	**0.55**	82.6	8.2	0.54	73.0	**7.6**	0.54
hiv 1625	58.2	8.4	0.81	**37.0**	**6.0**	**0.84**	57.8	7.0	**0.84**
segment	102.2	15.4	**0.95**	93.4	**7.8**	**0.95**	**72.6**	8.0	**0.95**
seismic bumps	**31.4**	7.2	**0.93**	49.4	4.4	**0.93**	47.0	**3.8**	**0.93**
splice	45.8	8.0	**0.95**	**42.6**	**7.4**	0.94	43.4	**7.4**	0.94
kr-vs-kp	**48.6**	10.0	**0.97**	**48.6**	**7.8**	0.93	54.6	8.4	0.95
hiv schilling	88.6	**6.2**	**0.87**	99.0	**6.2**	**0.87**	**44.6**	6.4	**0.87**
Statlog satellite	**88.2**	**9.0**	0.79	107.8	9.4	**0.80**	96.2	9.6	**0.80**
bank conv	**35.8**	7.8	**0.90**	47.0	7.4	0.89	36.6	**7.0**	**0.90**
spambase	173.8	17.4	**0.88**	110.6	9.8	**0.88**	**79.8**	**9.2**	**0.88**
musk2	**36.2**	8.0	**0.77**	45.0	**6.4**	0.74	49.0	6.6	0.67
heloc dataset v1	**50.2**	**7.8**	**0.73**	61.4	11.8	**0.73**	77.0	10.4	0.72
pendigits	393.0	14.0	0.90	371.0	**10.0**	0.91	**303.0**	11.0	**0.92**
compas-scores	137.5	10.5	**0.81**	460.5	15.2	0.80	**55.0**	**8.8**	**0.81**
HTRU 2	**17.4**	**3.8**	**0.98**	26.6	6.0	**0.98**	28.6	7.2	**0.98**
magic04	**249.4**	**13.2**	**0.85**	514.6	17.8	0.84	344.2	16.4	0.84
letter recognition	**3128.2**	28.8	**0.86**	4141.8	**21.8**	**0.86**	3581.8	23.2	**0.86**
ccdefault	22.2	**5.2**	**0.82**	30.6	6.4	**0.82**	**15.0**	5.4	**0.82**
ida	**67.0**	**9.0**	**0.99**	247.0	15.0	**0.99**	313.0	16.0	**0.99**
hand posture	**535.0**	15.2	**0.67**	741.8	**14.6**	0.64	841.4	19.2	0.63

Table 7.6: Size, depth, and accuracy of pruned decision trees induced by CART and processed by DT-SLIM(CART).

using size minimization does not automatically achieve a smaller size after pruning, hinting that balancing the subtrees may actually help pruning. Overall, DT-SLIM improved decision trees are overall smaller and less deep than C4.5 decision trees, as C4.5 achieved the smallest size on 13 and the smallest depth on 3 of the 32 instances. In terms of accuracy, C4.5 decision trees do a little better. While the difference is mostly small, for instances with many unknown values, like *anneal*, the combination of DT-SLIM and pruning is less accurate. Here, DT-SLIM with size minimization achieves higher accuracy than DT-SLIM without size minimization.

Comparing CART and C4.5 after pruning, C4.5 decision trees are smaller and more accurate, while CART decision trees remain less deep. The results are shown in Table 7.6. Here, the size varies: sometimes CART does better without DT-SLIM, while the depth is almost always better using DT-SLIM. Overall, CART without DT-SLIM achieves the best size on 11 and the best depth on 6 of the 32 instances. DT-SLIM(CART)-improved decision trees are comparatively accurate. The two DT-SLIM configurations found the most accurate decision tree for all but 8 instances, and on these 8 instances, the accuracy was only slightly lower than that of the CART-induced decision tree.

Pruning is important, as can be seen by comparing the result with and without pruning. Independent of DT-SLIM, the decision trees become much smaller, less deep, and most of the time, much more accurate. Particularly the largest unpruned decision tree for instance *ccdefault* with around 7000 nodes, can achieve better generalization with only 22 nodes after pruning. Therefore, we should always compare our methods with and without pruning. DT-SLIM still reduces the complexity and increases the accuracy of many of the instances, making it a valuable addition before pruning.

7.9.4 CP Methods

CP methods aim for a similar goal as DT-SLIM, small and accurate decision trees, but try to directly compute them instead of applying pruning. Hence, we wanted to compare DT-SLIM to state-of-the-art CP methods. Unfortunately, many of them require instances with binary domains and at most two classes (Hu et al., 2019; Verhaeghe et al., 2020a; Aglin et al., 2020; Demirovic and Stuckey, 2021). The restriction to two classes, limits the number of instances we could compare, and the binary domains cause practical problems, as converting the instances causes the size to increase dramatically. Particularly OSDT (Hu et al., 2019), which seems the most promising comparison, already exceeds the memory bounds on medium sized instances.

We, therefore, used BinOCT (Verwer and Zhang, 2019) for our comparison, as it supports instances with many classes and arbitrary domains. We used BinOCT with depth limits from 2 to 6. Table 7.7 shows a comparison between DT-SLIM-improved decision trees and BinOCT-induced decision trees. For both methods, we picked the decision trees that performed best in terms of accuracy (Virtual Best). It is noteworthy that BinOCT induced the decision trees much faster than DT-SLIM improved them.

The results show that, in general, DT-SLIM achieves better accuracy, but the decision trees are usually more complex. When not picking the virtual best but the best according to the validation set, BinOCT's accuracy compares slightly better. Nonetheless, in this comparison, DT-SLIM is sometimes a few percentage points worse but often significantly better. The underlying idea behind BinOCT and similar methods is that high training accuracy correlates with high testing accuracy. This correlation does not necessarily hold for larger decision trees. Hence, for the instances that require large decision trees, it is not surprising that DT-SLIM performs better.

	Virtual Best						Validation Best					
	DT-SLIM			BinOCT			DT-SLIM			BinOCT		
Instance	Size	Dep	Acc.	Size	Dep	Acc.	Size	Dep	Acc.	Size	Dep	Acc.
musk1	42.6	5.2	**0.73**	**29.0**	**5.0**	0.69	29.4	4.2	0.64	**21.0**	**4.0**	**0.69**
australian	12.6	4.4	**0.87**	**7.0**	**2.0**	0.86	**11.8**	4.0	0.84	15.0	**2.6**	**0.85**
hiv 746	19.8	6.6	**0.81**	**13.0**	**3.0**	0.56	21.8	8.0	**0.76**	**7.0**	**2.0**	0.56
IndiansDiabetes	**33.4**	5.4	**0.75**	44.6	**5.0**	0.73	26.2	5.0	**0.73**	**22.2**	**3.2**	0.72
vehicle	118.2	7.4	**0.71**	**49.8**	**5.0**	0.68	109.4	7.6	**0.69**	**39.0**	**4.4**	0.67
anneal	83.0	8.0	**0.94**	**5.0**	**2.0**	0.82	41.0	6.0	**0.87**	**5.0**	**2.0**	0.82
hiv impens	73.8	9.4	**0.87**	**13.4**	**3.0**	0.85	30.2	7.6	**0.85**	**13.0**	**2.8**	0.84
tic-tac-toe	44.2	6.2	**0.84**	**26.2**	**4.0**	0.73	51.8	6.4	**0.79**	**37.8**	**4.8**	0.67
objectivity	**21.0**	**4.2**	**0.81**	39.8	5.0	**0.81**	26.6	4.4	0.80	**15.8**	**3.0**	**0.81**
german-credit	42.6	6.8	**0.75**	**15.0**	**3.0**	0.72	**32.6**	5.2	**0.71**	36.2	**4.4**	0.70
biodeg	60.6	6.4	**0.82**	**46.2**	**5.0**	0.78	44.6	5.2	0.78	**34.6**	**4.4**	**0.79**
messidor	89.4	7.2	**0.66**	**26.6**	**4.0**	0.64	95.0	8.2	**0.64**	**19.4**	**3.4**	**0.64**
yeast	59.8	7.6	**0.57**	**57.8**	**5.0**	0.56	59.0	7.6	0.53	**30.6**	**3.6**	**0.54**
hiv 1625	49.8	7.2	**0.85**	**37.4**	**5.0**	0.78	36.6	6.4	**0.84**	**10.6**	**2.4**	0.77
segment	81.0	7.4	**0.96**	**29.4**	**6.0**	0.92	73.8	7.2	**0.96**	**19.4**	**5.0**	0.86
seismic bumps	19.4	4.4	**0.93**	**14.6**	**3.0**	**0.93**	**5.8**	**1.6**	**0.93**	8.6	2.2	**0.93**
splice	39.4	7.0	**0.94**	**26.2**	**4.0**	0.75	42.6	7.4	**0.94**	**18.2**	**3.0**	0.74
kr-vs-kp	41.0	7.4	**0.95**	**9.8**	**3.0**	0.58	49.8	7.8	**0.94**	**10.6**	**2.8**	0.58
hiv schilling	**51.8**	7.2	**0.88**	65.0	**6.0**	0.83	19.8	4.4	**0.87**	**11.0**	**2.4**	0.83
Statlog satellite	91.8	8.6	**0.82**	**56.2**	**5.0**	0.79	76.2	8.8	**0.80**	**56.2**	**5.0**	0.79
bank conv	17.0	4.2	**0.90**	**15.0**	**3.0**	**0.90**	29.4	6.2	0.89	**11.8**	**2.4**	**0.90**
spambase	75.8	8.2	**0.90**	**68.2**	**6.0**	0.89	72.2	8.4	**0.88**	**40.2**	**4.6**	**0.88**
musk2	48.6	**5.4**	**0.85**	**45.8**	6.0	0.80	48.6	5.8	0.74	**27.0**	**4.2**	**0.80**
heloc dataset	**31.0**	8.0	**0.74**	61.0	**5.0**	0.73	**40.6**	8.8	**0.73**	**40.6**	**4.2**	**0.73**
pendigits	259.0	11.0	**0.92**	**101.0**	**6.0**	0.80	259.0	11.0	**0.92**	**61.0**	**5.0**	0.77
compas-scores	**75.0**	8.8	**0.81**	117.0	**6.0**	**0.81**	49.4	9.6	**0.81**	**46.6**	**4.4**	0.80
HTRU 2	**23.0**	5.0	**0.98**	30.6	**4.0**	**0.98**	**6.6**	2.8	**0.98**	7.0	**2.0**	**0.98**
magic04	358.6	15.6	**0.85**	**109.8**	**6.0**	0.84	160.2	13.0	**0.84**	**53.8**	**4.8**	0.83
letter rec	3703.4	22.0	**0.86**	**97.0**	**6.0**	0.47	3563.0	22.2	**0.86**	**56.6**	**5.0**	0.37
ccdefault	21.4	5.6	**0.82**	**15.0**	**3.0**	**0.82**	**8.6**	3.0	**0.82**	13.0	**2.6**	**0.82**
ida	181.0	16.0	**0.99**	**55.0**	**5.0**	**0.99**	139.0	12.0	**0.99**	**7.0**	**2.0**	0.98
hand posture	760.2	18.0	**0.70**	**123.8**	**6.0**	0.63	571.8	15.6	**0.68**	**40.6**	**4.2**	0.59

Table 7.7: Comparison between BinOCT and DT-SLIM. The left side shows the virtual best in regards to the testing accuracy and the right hand side the best according to the accuracy on the validation set.

7.10 Conclusion

We introduced DT-SLIM, an anytime method that reduces the complexity of decision trees and scales to very large decision trees and classification instances. Our approach includes the novel SAT encoding DT_pb and novel data reductions.

The experimental results show how effectively DT-SLIM can reduce the size and depth of decision trees induced by a standard heuristic. For very large decision trees, the effectiveness is limited by the speed of DT-SLIM. Hence, improving the efficiency of DT-SLIM could improve the results on these instances. Unsurprisingly, the reduction in size is generally more significant if we minimize not only the depth but also the size. In terms of accuracy, DT-SLIM might slightly reduce the accuracy. Hence, in applications where a smaller or less deep decision tree is essential, DT-SLIM is very applicable.

Pruning is of utmost practical relevance, and the results look very different after applying pruning. Generally, the decision tree's size decreases drastically, independent of whether it is DT-SLIM improved or not. Here, the initial decision tree after pruning can become smaller without DT-SLIM than with DT-SLIM. In terms of accuracy, the results vary greatly, depending on the heuristic used for the initial decision tree. For CART-induced decision trees, DT-SLIM improves the depth consistently, achieves overall good size reduction, and provides comparatively very good accuracy. For C4.5-induced decision trees, DT-SLIM is also able to provide low-depth, small, and accurate decision trees. While DT-SLIM can overall improve CART decision trees very well, it is more instance-dependent for C4.5, where it can nonetheless very often improve the decision tree, not only regarding the complexity but also the accuracy.

DT-SLIM shows the versatility and flexibility of the SLIM framework. The learning context poses some unique challenges among SLIM instantiations. The encoding's size depends on two factors: the depth of the decision tree and the number of samples. This two-dimensionality already makes determining a budget considerably harder. In the encodings, keeping the decision tree consistent with the training data requires many constraints. The large number of constraints, in turn, necessitates the use of data reductions, as otherwise, many local instances would be too large for the solver. Even with data reductions, the local instances become very large, which requires a long local timeout. This long local timeout allows for comparatively few improvements. Hence, we need a good selection strategy that will pick very few local instances that do not admit any improvements, as we would otherwise waste a lot of time. DT-SLIM masters all these challenges and scales far beyond the capabilities of the encodings themselves. Furthermore, other approaches for finding low-complexity decision trees try to maximize the accuracy, given a specific size or depth of a decision tree, like the CP methods. While this approach works well, as our experiments show, CP methods do not work for larger decision trees. To the best of our knowledge, GC-SLIM is the only approach that explicitly scales to large decision trees and large instances.

In short, the results show that DT-SLIM is indeed able to reduce the complexity of decision trees, often drastically. Further, we have shown that when considering pruning,

the benefits carry over for many instances. DT-SLIM is, therefore, a viable method for reducing decision tree complexity in a variety of use cases. Particularly if the depth of the decision tree is of importance, as DT-SLIM provides significant reductions in depth.

7.10.1 Future Work

We see two parts of DT-SLIM that can be improved in future work. The *running time*, as our method still requires a significant time investment and different ways of *pruning* that are more interleaved into the improvement iterations.

Running Time

DT-SLIM, in the current implementation, still offers room for improvement. One avenue of improvement is parallelization, as DT-SLIM can run in parallel on independent subtrees. Particularly on large decision trees, this could significantly improve runtime. Additionally, the current implementation is written in Python, but the supporting code around the SAT solver has become relatively complex. A significant part of the runtime is spent selecting the local instances and there, particularly on instance size reduction. Therefore, re-implementing the supporting code in a native language, e.g., C++, could bring considerable improvement in efficiency.

Pruning

While post-pruning works well, there are other options. As discussed in Section 7.8, pruning can be included in the SAT encoding. In the context of DT-SLIM, it would be possible to apply this method instead of finding classifying decision trees. While this can be applied and tuned in many ways, it could improve the overall accuracy of the decision tree. Furthermore, the pruning results hint at the possibility that DT-SLIM might help post-pruning remove larger parts of the decision tree. The results could be improved when DT-SLIM could be tuned more towards providing a good input for the pruning method.

CHAPTER 8

Conclusion

8.1 Concluding Remarks

We discussed different methods for scaling SAT-based methods to large instances. Our results show that even if the encoded instance seems too large or complex to be solved, several methods may still allow us to harness the power of modern SAT solvers. Whenever a good, but not necessarily optimal, solution is sufficient, SLIM sets a high bar for maximum size and complexity of an instance.

Encoding Refinement

We discussed encodings for two different problems: hypertree width and twin-width. These encodings represent a large group of encodings with two properties: (i) the encoded instance is considerably larger than the original instance, and (ii) they require a dedicated characterization for the SAT encoding. In comparison, the encodings for DFVS and graph coloring in Chapters 5 and 6 follow directly from the definition of the encoded problem. In both cases, there is nothing we can do to make encoded instances smaller besides choosing a different encoding. On the other hand, we showed how we can refine the hypertree width and twin-width encodings to great effect. In the case of hypertree width, the key was a good characterization, while for twin-width we succeeded by finding good encodings based on the same characterization.

The performance of these refinements becomes clearer when considering the whole history of the encodings. Since they were originally introduced in the respective papers (Schidler and Szeider, 2020, 2021b, 2022), they were refined, and the versions presented in this thesis perform much better. The augmented hypertree width encoding (Schidler and Szeider, 2020) originally performed much worse than the pure hypertree width encoding (Schidler and Szeider, 2021b). The refined version discussed in Chapter 3 performs even better than the pure encoding. The story is even more impressive for the absolute twin-width

encoding. Here the original version (Schidler and Szeider, 2022) could solve almost no instances, while the version presented in Chapter 4 performs better than the relative encoding on most instances.

Another success story for encoding refinements is the decision tree encodings discussed in Chapter 7. Table 7.2 shows how much better the refinement of Avellaneda's (2020a) encoding by Shati et al. (2021) performs. Overall, the success story of decision tree encodings is also a success story of refining the encodings.

SAT encodings often work well with little effort, but, as our results show, extra effort for refinement can go a long way towards better scalability.

Lazy Encodings

We showed one lazy encoding approach that is similar to many other lazy approaches. Our results and a history of successful applications show how powerful lazy encodings can be, even if the encoded instance is very large. This scalability can be supported using data reductions, which also supported scalability in Chapter 7. Our implementation discussed in Chapter 5 beats all available methods for DFVSP and is the new state-of-the-art for finding minimum DFVS.

Cycle propagation, the implementation of the lazy clause generation directly into the MaxSAT solver gives our approach a significant performance boost. Indeed, without our data reductions, cycle propagation would have made the difference between winning or not winning PACE 2022. While some solvers offer interfaces that allow such an approach without changing the solver itself, this is not the case for our MaxSAT solver of choice. Hence, we had to modify the solver, and while the results are great, our solution is solver-dependent, removing one of the advantages of a SAT-based approach. We hope that our results, together with other approaches that work similarly, such as for dynamic symmetry breaking (Kirchweger and Szeider, 2021), will lead to the inclusion of the necessary APIs in future solvers.

SLIM

The two SLIM approaches discussed in this thesis highlight two characteristics of SLIM: (i) how effective SLIM is, and (ii) how different SLIM approaches can be in detail.

While most local improvements on a decision tree lower the decision tree's overall complexity—when focusing on deep branches—we need many improvements to eliminate a single color from a graph. This difference in the effectiveness of the improvements allows us to work with a long local timeout for decision trees and requires a short local timeout for graph coloring.

Another big difference is the size of the encoding. While the list coloring encoding is relatively small and becomes smaller the more constrained it is, the decision tree encoding is comparatively large, particularly when considering the depth of the decision tree. The result is that the local instances in DT-SLIM are limited by size and complexity. In

contrast, the local instances in GC-SLIM are usually only limited by complexity, as the size becomes an issue much later. This leads to a static budget in DT-SLIM and a dynamic budget in GC-SLIM.

The last major difference is how elaborate the selection of a local instance is. In GC-SLIM, the construction follows a clear algorithm and will quickly lead to a local instance. Therefore, GC-SLIM's runtime is almost entirely used by the SAT solver. DT-SLIM constructs local instances by relying heavily on data reductions, and the local instance selection, together with the data reductions, take up a considerable amount of the runtime. These differences also affect how we select local instances, as the penalty for choosing an unimprovable local instance in DT-SLIM is higher due to the elaborate selection and the long local timeout.

Overall, both SLIM instantiations and those mentioned in related work show how well SLIM achieves scalability. This scalability does not only compare well to other SAT-based methods: for DT-SLIM there is no comparable non-SAT-based method, and for GC-SLIM none of the other approaches perform as well on such a large variety of graphs.

8.2 Future Work

We have already outlined much future work in the respective chapters. In the more general scope of this thesis, we still see much interesting work that can be done.

Although we discussed both lazy approaches and SLIM, there has not yet been a combined approach, i.e., a SLIM approach that solves the local instances using a lazy encoding. For graph coloring, such a lazy encoding exists (Glorian et al., 2019), and for decision trees, finding a good lazy encoding might be a worthwhile endeavour on its own. We expect that by using lazy encodings, we can increase the size of the local instance and thereby leverage more improvements.

Twin-width is an interesting topic, as due to its novelty, many open questions exist. We can use our encoding to find the twin-width of interesting graphs and expect to scale this approach up further using cube-and-conquer and creates a very good heuristic using a SLIM approach. While the theoretical knowledge about twin-width and how to use it is increasing, little is known about how to compute it. We see this as another avenue for scaling the encoding, as many insights could be included in the encoding in the form of symmetry breaking and a propagation-like mechanism, as discussed in Chapter 5.

In a more general scope, we see many applications for SAT-based methods that can scale. Particularly the area of explainable AI has seen many applications of logic-based methods, among them SAT. In this thesis, we showed that decision tree learning could be tackled using SLIM, and we expect to expand this to more areas in explainable AI. This expectation is founded in the fact that in the context of explainable AI, we often deal with large amounts of data and complex models, which sounds like the ideal use case for a SLIM approach.

APPENDIX A

Appendices

A.1 Hypertree Decompositions (Chapter 3)

In this section, we present the proofs for our two main theorems in Chapter 3, showing the correctness of our two characterizations.

A.1.1 Proof of Theorem 3.2

In this section, we establish Theorem 3.2 in an algorithmic way, by providing polynomial-time algorithms that transform hypertree decompositions into pure/augmented hypertree orderings of the same width, and conversely, polynomial-time algorithms that transform pure/augmented hypertree orderings into hypertree decompositions of the same width. Also for this section, let H be an arbitrary connected hypergraph.

From Hypertree Decompositions to Pure Hypertree Orderings

Let $\mathcal{D} = (T_D, \chi_D, \lambda_D)$ be a hypertree decomposition of H of width W. We will translate $\mathcal{D}$ into a pure hypertree ordering $\mathcal{P}$ of width W.

We assume, w.l.o.g, that the following properties hold for $\mathcal{D}$:

N1 For all $t \in V(T_D)$, $e \in \lambda_D(t)$: $e \cap \chi_D(t) \neq \emptyset$.

N2 For all $\{t, t'\} \in E(T_D)$, if t' is the parent of t then $\chi_D(t) \not\subseteq \chi_D(t')$.

We observe that N1 can be established by removing all violating hyperedges from the respective covers. N2 can be established by contracting edges between violating nodes and retaining the parent's cover and bag. Due to this property, at least one vertex is forgotten at every node (recall the definition of "forgetting a vertex" from Section 3.2).

Next, we define a preorder $\preceq^*$ on $V(H)$ by setting $v \preceq^* w$ if and only if $f(v)$ is in the subtree rooted at $f(w)$ (this includes the case $f(v) = f(w)$). By letting $\prec$ be any total ordering that refines $\preceq^*$ and setting $\lambda_\prec(v) := \lambda_D(f(v))$, we define $\mathcal{P} = (\prec, \lambda_\prec)$. Further, let A be the set of active arcs according to $\prec$ and A^* its transitive closure (see Section 3.3).

We first show some properties of the transformation and then establish its correctness. The following two observation state basic properties of pure hypertree orderings and will be used in the following proofs:

Observation A.1. *If $(v, w) \in A^*$ then there is some $u \in V(H)$ such that $(u, w) \in A$, where possibly $u = v$.*

Lemma A.2. *If $(u, w) \in A$ and $(u, v) \in A^*$, such that $v \prec w$ then $(v, w) \in A$.*

Proof. Let v' be the $\prec$-smallest vertex such that $(u, v'), (v', v) \in A^*$ holds. By the definition of A, $(u, v') \in A$ and therefore $(v', w) \in A^*$. The hypothesis follows inductively by using v' instead of u. □

Lemma A.3. *If $(v, w) \in A$ and $\{v, w\} \notin E(G(H))$ then there is some $u \in V(H)$ such that $\{u, w\} \in E(G(H))$ and v is arc-reachable from u.*

Proof. Given the premise, there must be a $u' \in V(H)$ such that $(u', w) \in A$ and $(u', v) \in A$. Either $\{u', w\} \in E(G(H))$ or there must be a vertex $u'' \in V(H)$ such that $(u'', w) \in A$ and $(u'', u') \in A$. Since $u'' \prec u' \prec v$, this process moves towards the beginning of $\prec$ in a strictly monotonous fashion. We therefore arrive at a vertex u such that $\{u, w\} \in E(G(H))$ (Observation A.1). Since we arrived at u following an arc-path, v is arc-reachable from u. □

Lemma A.4. *If $(u, w) \in A^*$, then $f(u)$ is in the subtree rooted at $f(w)$.*

Proof. From the premise, we know that w is arc-reachable from u. We show the desired property by induction on $|\{\, v : (u, v) \in A^*, (v, w) \in A^* \,\}|$, the length of the arc-path.

Base Case: The path has length 0. In this case the premise implies that $\{u, w\} \in E(G(H))$. Therefore, T2 implies that u and w must occur in a bag together and due to T3 it holds that $u \in \chi_D(f(w))$ or $w \in \chi_D(f(u))$. Therefore, by the definition of $\prec$, it holds that $w \in \chi_D(f(u))$, as otherwise $w \prec u$ which would contradict the premise that $(u, w) \in A^*$. The hypothesis is implied by $w \in \chi_D(f(u))$ and T3.

Induction Step: The path has length $i + 1$. By the definition of A^* there exists a v such that $(u, v) \in A^*$ and $(v, w) \in A^*$. By the induction hypothesis, $f(v)$ is in the subtree rooted at $f(w)$ and $f(u)$ in the subtree rooted at $f(v)$, consequently the induction step holds. □

Lemma A.5. *If $(v, w) \in A$, then $w \in \chi_D(f(v))$.*

Proof. Case 1: $\{v, w\} \in E(G(H))$. By T2, there has to be a $t \in V(T_D)$ such that $\{v, w\} \subseteq \chi_D(t)$. Due to T3, w has to occur in the bag of every node on the path between t and $f(w)$. Since $v \in \chi_D(t)$ and by Lemma A.4, $f(v)$ is in the subtree of T_D rooted at $f(w)$, thus $w \in \chi_D(f(v))$ and the lemma holds.

Case 2: $\{v, w\} \notin E(G(H))$. By Lemma A.3, there exists some $u \in V(H)$ such that $\{u, w\} \in E(G(H))$ and v is arc-reachable from u. By T2, there is a node $t \in V(T_D)$ such that $\{u, w\} \subseteq \chi_D(t)$. Due to Lemma A.4, $f(u)$ is in the subtree of T_D rooted at $f(v)$ and $f(u)$ is in the subtree rooted at $f(w)$. Since $v \prec w$, it holds that $f(v)$ is in the subtree rooted at $f(w)$. Since $w \in \chi_D(t)$, by T3, w must occur in all bags of the nodes on the path between t and $f(w)$, including $f(v)$. Therefore, the lemma follows. □

Proposition A.6. *$\mathcal{P}$ is a pure hypertree ordering of H of width W.*

Proof. We first observe that we do not change any edge cover and therefore, the width does not increase.

Next we show that P1 holds. For every vertex $v \in V(H)$ it holds that $v \in \bigcup \lambda_{\prec}(v)$ by the definition of $\lambda_{\prec}$. Further, for every vertex $w \in V(H)$ such that $(v, w) \in A$ it holds that $w \in \chi_D(f(v))$ by Lemma A.5. This implies that $w \in \bigcup \lambda_D(f(v)) = \bigcup \lambda_{\prec}(v)$ and therefore, P1 holds.

Next, we show that P2 holds by contradiction. We assume for vertices $v, w \in V(H)$ that $v \in B(w)$ and $v \in \bigcup \lambda_{\prec}(w)$. Given the definition of B and Lemma A.4, we know that v occurs in at least one of the bags in the subtree rooted at $f(w)$. Since $\lambda_{\prec}(w) = \lambda_D(f(w))$ and $v \in \bigcup \lambda_{\prec}(w)$ it must hold that $v \in \chi_D(f(w))$ due to T4. Since $v \prec w$, this implies $f(v) = f(w)$, therefore $\lambda_{\prec}(v) = \lambda_{\prec}(w)$. Consequently, every $u \in P(v, w)$ has the same edge cover and thereby $v \in R(w)$, implying $v \notin B(w)$ and contradicting the assumption. □

From Pure Hypertree Orderings to Hypertree Decompositions

We now proceed and show the other direction. The following lemma formally justifies the definition of R.

Lemma A.7. *Let $\mathcal{D} = (T_D, \chi_D, \lambda_D)$ be a hypertree decomposition and let $t, t' \in V(T_D)$, such that $\{t, t'\} \in E(T_D)$ and t' is the parent of t. If $\lambda_D(t) \subseteq \lambda_D(t')$ then we can add all vertices from $\chi_D(t)$ to $\chi_D(t')$ and D remains a valid hypertree decomposition.*

Proof. Since T1–T4 held before the transformation, we show that they also hold after the transformation. Since we did not remove any vertices from the bags, T1 and T2 still hold. Since t and t' are adjacent in the tree, T3 also holds. Since by adding vertices to the bag, we can only reduce the number of omitted vertices, T4 also holds. Finally, since $\lambda_D(t')$ contains at least the hyperedges in $\lambda_D(t)$, it remains a valid edge cover. □

Corollary A.8. *Let $\mathcal{D}$ be a hypertree decomposition (T_D, χ_D, λ_D) and let $t, t' \in V(T_D)$, such that t is in the subtree rooted at t'. If for every node t'' on the path between t and t' (including t') it holds that $\lambda_D(t) \subseteq \lambda_D(t'')$, then we can add all vertices in $\chi_D(t)$ to $\chi_D(t'')$ and $\mathcal{D}$ remains a valid hypertree decomposition.*

Given a pure hypertree ordering $\mathcal{P} = (\prec, \lambda_\prec)$ of H, we define $\mathcal{D} = (T_\prec, \chi_D, \lambda_D)$, where

- $T_\prec$ is the canonical tree,
- $\lambda_D(\tau_\prec(v)) = \lambda_\prec(v)$ for all $v \in V(H)$, and
- $\chi_D(\tau_\prec(v)) := \chi_\prec(v) \cup \bigcup_{u \in R(v)} \chi_\prec(u)$ for all $v \in V(H)$.

This definition implies that for every $v \in V(H)$, $f(v) = \tau_\prec(v)$.

Lemma A.9. *If $v \in \chi_D(\tau_\prec(u))$ and $u \prec v$ then $v \in \chi_\prec(u)$ and $(u, v) \in A$.*

Proof. Since $v \in \chi_D(\tau_\prec(u))$ either (i) $(u, v) \in A$ or (ii) $(u', v) \in A$ with $u' \in R(u)$. Case (i) implies that $v \in \chi_\prec(u)$ by definition of $\chi_\prec$. Case (ii) implies that u is arc-reachable from u'. Let u'' be the successor of u' in the arc-path from u' to u. Since $(u', v) \in A$ and $(u', u'') \in A$ it follows that $(u'', v) \in A$. The same argument holds for u'' and repeating the process eventually shows the desired property for u. □

Corollary A.10. *If $(u, w) \in A$, $(u, v) \in A^*$, and $v \prec w$, then $(v, w) \in A$.*

Proposition A.11. *$\mathcal{D} = (T_\prec, \chi_D, \lambda_D)$ is a hypertree decomposition of H of width W.*

Proof. By definition of the canonical tree, $T_\prec$ is rooted at $\tau_\prec(r) \in V(T_\prec)$, where $r \in V(H)$ is the $\prec$-largest vertex. We argue that every node, except $\tau_\prec(r)$, has exactly one parent, i.e., that $T_\prec$ is indeed a tree. Since H is connected, for each vertex $u \in V(H)$, except r, there exists a vertex $v \in V(H)$ such that $(u, v) \in A$. Due to the definition of $E(T_\prec)$ this implies that $\tau_\prec(u)$ is connected to exactly one $\tau_\prec(v)$ such that $u \prec v$. Therefore, $\tau_\prec(v)$ is the parent of $\tau_\prec(u)$ and $T_\prec$ is connected. Since Property P1 holds, it follows that for every $\tau_\prec(v) \in V(T_\prec)$, $R(v)$ only contains vertices covered by $\lambda_\prec(v)$. Therefore, the preconditions of Corollary A.8 are met and $\lambda_D(v)$ is a valid edge cover for $\chi_D(v)$.

Conditions T1 and T2 hold as well, as we have the following for each $e \in E$. Let u be the $\prec$-smallest vertex in e, then for each $v \in e \setminus \{u\}$ it holds that $(u, v) \in A$. Therefore, by the construction of χ_D, $e \subseteq \chi_D(\tau_\prec(u))$.

Next we show that T3 holds. Let $u, w \in V(H)$ be arbitrary vertices, such that $w \in \chi_D(\tau_\prec(u))$. Whenever $w \prec u$ and therefore $w \notin \chi_\prec(u)$, T3 holds on the path from $\tau_\prec(w)$ to $\tau_\prec(u)$ by construction of R and χ_D. We therefore assume that $u \prec w$. It remains to show that for each node $\tau_\prec(v) \in V(T_\prec)$ on the path between $\tau_\prec(u)$ and $\tau_\prec(w)$ it holds that $w \in \chi_D(\tau_\prec(v'))$. We proceed by induction on the position of $\tau_\prec(v)$ on the path from $\tau_\prec(u)$ to $\tau_\prec(w)$.

Base Case: The first edge on the path is $\{\tau_\prec(u), \tau_\prec(v)\} \in E(T_\prec)$. By Lemma A.9 it holds that $(u, w) \in A$. It then follows from Corollary A.10 that $(v, w) \in A$, thereby showing the base case.

Induction Step: Consider an arbitrary edge $\{\tau_\prec(v'), \tau_\prec(v'')\} \in E(T_\prec)$ on the path between $\tau_\prec(u)$ and $\tau_\prec(v)$. Since we know that $w \in \chi_D(\tau_\prec(v'))$, we can use the same argument as in the base case to show that $w \in \chi_D(\tau_\prec(v''))$. Therefore, the hypothesis and T3 hold.

Next we prove that T4 holds by contradiction. Assume vertices $u, v, w \in V(H)$, such that v is omitted at $\tau_\prec(w)$, $v \in \chi_D(\tau_\prec(u))$ and $\tau_\prec(u)$ is a descendant of $\tau_\prec(w)$ in T. Since $v \in \chi_D(\tau_\prec(u))$, either $(u, v) \in A$ or $(u', v) \in A$, where $u' \in R(u)$. Both cases imply that either (i) $(v, w) \in A^*$ or (ii) $(w, v) \in A^*$. Case (i) implies that $v \in R$ as otherwise P2 would be violated. This contradicts the assumption that $v \notin \chi_D(\tau_\prec(w))$. Case (ii) implies by Lemma A.9 that $(u, v) \in A$. Therefore, by Corollary A.10, $(w, v) \in A$, contradicting the initial assumption that v is omitted at $\tau_\prec(w)$. Therefore, T4 holds.

Since T is a tree, λ_D assigns edge covers, and T1–T4 hold, D is indeed a hypertree decomposition for H of width W. □

Propositions A.6 and A.11 establish Theorem 3.2.

A.1.2 Proof of Theorem 3.3

Similar to the previous section, we prove this theorem in an algorithmic way. As before, let H be an arbitrary connected hypergraph.

For convenience, we write

$$A^{\equiv} := A \cup \{\, (u, v) : u \equiv v \,\},$$

which gives

$$\chi^{\equiv}_\prec(v) = \{v\} \cup \{\, w : (v, w) \in A^{\equiv} \,\}.$$

A.1.3 From Hypertree Decompositions to Augmented Hypertree Orderings

Let $\mathcal{D} = (T_D, \chi_D, \lambda_D)$ be a hypertree decomposition of H of width W. We define $\mathcal{A} = (\prec, \lambda^{\equiv}_\prec, \equiv)$ as follows:

- We let $\equiv$ be the equivalence relation where any two vertices $u, v \in V(H)$ are equivalent (in symbols $u \equiv v$), if $f(u) = f(v)$.
- Let $\prec^*$ be the partial order where $u \prec^* v$ if $f(u)$ is a descendant of $f(v)$ in T_D, for any $u, v \in V(H)$. We let $\prec$ be an arbitrary, but fixed, total order that refines $\prec^*$.

- We let $\lambda^{\equiv}_{\preceq}(u) = \lambda_D(f(u))$ for all $u \in V(H)$.

Proposition A.12. *$\mathcal{A}$ is an augmented hypertree ordering of H of width W.*

Proof. We show that A1 holds, i.e., that $\lambda^{\equiv}_{\preceq}(u)$ contains edge covers for all bags in $\chi^{\equiv}_{\preceq}$. For this purpose, we assume w.l.o.g., that the following holds for T_D: for any two nodes $t, t' \in V(T_D)$ such that $\{t, t'\} \in E(T_D)$, it holds that $\chi_D(t) \setminus \chi_D(t') \neq \emptyset$ and $\chi_D(t') \setminus \chi_D(t) \neq \emptyset$, i.e., neither of the two sets $\chi_D(t)$ and $\chi_D(t')$ is a subset of the other. This can be achieved by contracting violating edges $\{t, t'\}$ and retaining the larger bag. This does not affect the validity of the decomposition, as T1, T2, T3 and T4 still hold. This assumption implies that at each node in T at least one vertex is forgotten (recall, again, the definition of "forgetting a vertex" from Section 3.2).

We will argue by case distinction that, given an arbitrary vertex u, it holds that $\chi^{\equiv}_{\preceq}(u) \subseteq \chi_D(f(u))$. A vertex v is contained in $\chi^{\equiv}_{\preceq}(u)$ for at least one of the four reasons:

(i) $u = v$,

(ii) $u \equiv v$,

(iii) $\{u, v\} \in E(G(H))$, or

(iv) (u, v) is a fill-in edge in $A^{\equiv}$.

Case (i): the hypothesis $\chi^{\equiv}_{\preceq}(u) \subseteq \chi_D(f(u))$ holds, as $u \in \chi_D(f(u))$.

Case (ii): similarly to Case (i), the hypothesis holds, as by definition $f(u) = f(w)$.

For the remaining cases, we assume that the first two cases do not prevail.

Case (iii): by T2, there exists a bag $\chi_D(t)$ such that $u, v \in \chi_D(t)$. By T3 either $f(u)$ is a descendant of $f(v)$ or $f(v)$ a descendant of $f(u)$. Since $v \in \chi^{\equiv}_{\preceq}(u)$ and we eliminated Cases (i) and (ii) we know that $u \prec v$ and therefore by definition $f(u)$ is a descendant of $f(v)$. By T3 this implies that $v \in \chi_D(f(u))$.

Case (iv): this case follows similarly: from the definition of $A^{\equiv}$ we know that $u \prec v$. Following the definition of fill-in edges, any fill-in edge (u, v) requires (possible fill-in) edges (w, u) and (w, v). From this we also know that $w \prec u \prec v$ and by definition $f(w) = f(u)$, or $f(w)$ is a descendant of $f(u)$. Since fill-in edges require other edges to the same vertex, there has to be at least one (w', v) such that $\{w', v\} \in E(G(H))$ and by definition $f(w') = f(u)$ or $f(w')$ is a descendant of $f(u)$, where $w' = w$ if $\{w, v\} \in E(G(H))$. Now the desired property $v \in \chi_D(f(u))$ holds, either because $w' \equiv u$, or because $f(w')$ is a descendant of $f(u)$ and by T2 v occurs in a bag below $f(u)$ and by T3 in the bag of $f(u)$.

In order to show that A2 (the Special Condition) holds, we assume the contrary, i.e., there are two vertices $u, v \in V(H)$ such that $(u, v) \in A^*$ and there is a vertex $w \in \bigcup \lambda^{\equiv}_{\prec}(v)$, such that $w \notin \chi^{\equiv}_{\prec}(v)$ and $w \in \chi^{\equiv}_{\prec}(u)$. It cannot hold that $u \equiv v$ as then, by definition, $\chi^{\equiv}_{\prec}(u) = \chi^{\equiv}_{\prec}(v)$. Further, $f(w)$ is a descendant of $f(v)$, as otherwise $v \prec w$, and $(v, w) \in A^{\equiv}$, due to fill-in edges, contradicting that $w \notin \chi^{\equiv}_{\prec}(v)$. This would imply that $w \notin \chi_D(f(v))$, which would violate T4. Therefore A2 holds.

Since all vertices that are forgotten at the same node are equivalent and are next to each other in $\prec$, A3 holds as well.

It remains to observe that since we do not change any of the edge covers in λ_D, the width remains the same. This concludes the proof of the proposition. □

From Augmented Hypertree Orderings to Hypertree Decompositions

Let $\mathcal{A} = (\prec, \lambda^{\equiv}_{\prec}, \equiv)$ be an augmented hypertree ordering of H of width W. We define $\mathcal{D} = (T_{\prec}, \chi_D, \lambda_D)$ where

- $T_{\prec}$ is the canonical tree,
- $\chi_D(\tau_{\prec}(v)) = \chi^{\equiv}_{\prec}(v)$, and
- $\lambda_D(\tau_{\prec}(v)) = \lambda^{\equiv}_{\prec}(v)$.

Proposition A.13. *$\mathcal{D}$ is a hypertree decomposition of H of width W.*

Proof. The width of $\mathcal{D}$ is the same as the width of $\mathcal{A}$, as we do not change $\lambda^{\equiv}_{\prec}$, and A1 guarantees that for all $v \in V(H)$, $\lambda_D(\tau_{\prec}(v))$ is an edge cover of $\chi_D(\tau_{\prec}(v))$.

T1 holds as each $v \in V(H)$ has a corresponding node $\tau_{\prec}(v) \in V(T_{\prec})$ and $\tau_{\prec}(v)$'s bag contains v.

For T2, assume an arbitrary $e \in E(H)$ and the $\prec$-smallest vertex v in e; now $e \subseteq \chi_D(v)$ by the definitions of $A^{\equiv}$ and $\chi^{\equiv}_{\prec}$.

To verify Condition T3, assume to the contrary that T3 is violated: i.e., there exist nodes $\tau_{\prec}(u), \tau_{\prec}(u'), \tau_{\prec}(u'') \in V(T_{\prec})$, such that $\tau_{\prec}(u')$ is on the path between $\tau_{\prec}(u)$ and $\tau_{\prec}(u'')$. Further, for some vertex $v \in V(H)$ it holds that $v \in \chi_D(u)$, $v \in \chi_D(u'')$, and $v \notin \chi_D(u')$. Therefore, the set of nodes $\{\, t \in V(T_{\prec}) : v \in \chi_D(t) \,\}$ induces in $T_{\prec}$ the (connected) subtrees $T_1, \ldots, T_k$, where $k > 1$ since T3 is violated. Let T_i be the subtree, such that $\tau_{\prec}(v) \in V(T_i)$; due to A3 $\{\, \tau_{\prec}(w) : w \in [v]_{\equiv} \,\} \subseteq V(T_i)$. Further, let $j \in \{1 \ldots k\} \setminus \{i\}$ and $\tau_{\prec}(r_j)$ be the root of T_j. By assumption, $v \in \chi_D(\tau_{\prec}(r_j))$, which implies by the definition of $A^{\equiv}$ that $(r_j, v) \in A^{\equiv}$ and that $\tau_{\prec}(r_j)$ is not the root of $T_{\prec}$. Hence, there exists a parent $\tau_{\prec}(p_j)$ of $\tau_{\prec}(r_j)$. By construction, $v \notin \chi^{\equiv}_{\prec}(p_j)$, and p_j is the $\prec$-smallest vertex with an incoming arc from r_j, hence $p_j \prec v$. Since $(r_j, v) \in A^{\equiv}$ and $(r_j, p_j) \in A^{\equiv}$, by the definition of $A^{\equiv}$ also $(p_j, v) \in A^{\equiv}$, contradicting $v \notin \chi^{\equiv}_{\prec}(p_j)$. Therefore T3 holds.

T4 holds due to A2.

This concludes the proof of the proposition. □

The conjunction of Propositions A.12 and A.13 establishes Theorem 3.3.

A.2 Directed Feedback Vertex Set (Chapter 5)

In this section, we list reductions from related work that we used in our implementation and give the proofs for the correctness of the novel reductions in Chapter 5.

A.2.1 Standard DFVSP Reductions

Here, we use $G \circ v$ as the digraph, called the exclusion of v from a digraph G by letting $G \circ v := G - v + N_{out}(v) \times N_{in}(v)$. The most well-known reduction rules for DFVSP preprocessing are those of Levy and Low (1988):

Reduction A.1 (LOOP)**.** *If there exists $v \in V(G)$ such that $(v, v) \in E(G)$ replace G by $G - v$.*

Reduction A.2 (IN0/1)**.** *If there exists $v \in V(G)$ such that v has at most one incoming edge, replace G by $G \circ v$.*

Reduction A.3 (OUT0/1)**.** *If there exists $v \in V(G)$ such that v has at most one outgoing edge, replace G by $G \circ v$.*

The latter two rules were later subsumed by Lemaic (2008), using the reductions.

Reduction A.4 (INDICLIQUE)**.** *If there exists $v \in V(G)$ such that the incoming edges of v form a diclique, replace G by $G \circ v$.*

Reduction A.5 (OUTDICLIQUE)**.** *If there exists $v \in V(G)$ such that the outgoing edges of v form a diclique, replace G by $G \circ v$.*

Apart from this, Lemaic introduced two new reductions

Reduction A.6 (DICLIQUE-2)**.** *If there exists $v \in V(G)$ whose neighbors can be partitioned into two disjoint cliques N_1, N_2 such that the bi-edges of v are a strict subset of N_1, replace G by $G \circ v$.*

Reduction A.7 (DICLIQUE-3)**.** *If there exists $v \in V(G)$ without bi-edges whose neighbors can be partitioned into three disjoint cliques $N_1, N_2, N3$, replace G by $G \circ v$.*

Note, that for soundness of all the above reductions it is necessary that LOOP is not applicable to v.

Furthermore, Lin and Jou introduced three further reductions that make use of bi-edges in the digraph.

Reduction A.8 (PIE)**.** *If there is an arc $(u,v) \in E(G)$ such that $(v,u) \notin E(G)$ and every path from v to u in G uses a bi-edge, replace G by $G - (u,v)$.*

Reduction A.9 (DOME)**.** *If there is an arc $(v,u) \in E(G)$ such that $(u,v) \notin E(G)$ and one of the following holds*

- *$\{\, p : (p,v) \in E(G), (v,p) \notin E(G) \,\} \subseteq \{\, p : (p,u) \in E(G) \,\}$, i.e., for every $(p,v) \in E(G)$ that is not a bi-edge there is an arc $(p,u) \in E(G)$.*
- *$\{\, p : (u,p) \in E(G), (p,u) \notin E(G) \,\} \subseteq \{\, p : (v,p) \in E(G) \,\}$, i.e., for every $(u,p) \in E(G)$ that is not a bi-edge there is an arc $(v,p) \in E(G)$.*

then replace G by $G - (v,u)$.

Reduction A.10 (CORE)**.** *If there exists $v \in V(G)$ such that all arcs of v are bi-edges and the neighbor of v form a diclique, replace G by $G \circ v$.*

The CORE reduction is a special case of the INDICLIQUE and OUTDICLIQUE reductions.

A.2.2 Standard VCP Reductions

Reduction A.11 (SUBSET (Stege and Fellows, 1999))**.** *If there exists $v,u \in V(G)$ such that $\{v,u\} \in E(G)$ and $N(v) \subseteq N(u) \cup \{u\}$, then replace G by $G - u$.*

Another reduction by Fellows et al. is more complicated but generalizes many others like the 2FOLD reduction (Xiao and Nagamochi, 2013).

Reduction A.12 (MANYFOLD (Fellows et al., 2018))**.** *If there exists a vertex $v \in V(G)$ such that there is a partition (C_1, C_2) of $N(v)$, where*

- *$|C_1| \geq |C_2|$,*
- *C_i is a clique for $i = 1, 2$, and*
- *for each $c_1 \in C_1$, there is precisely one $c_2 \in C_2$ such that $\{c_1, c_2\} \notin E(G)$.*

Then, replace G by

$$G - v - C_2 + \bigcup_{\{c_1,c_2\} \in M, c_1 \in C_1} \{c_1\} \times N(c_2),$$

where M denotes the set of missing edges from $G[N(v)]$.

Whereas MANYFOLD works well on dense graphs, the following reduction works on sparse graphs.

Reduction A.13 (4PATH (Fellows et al., 2018))**.** *If there exists a vertex $v \in V(G)$ such that*

- $N(v) = \{a, b, c, d\}$, *and*
- $E(G[N(v)]) = \{\{a, b\}, \{b, c\}, \{c, d\}\}$,

then replace G by

$$G - v + \{\{a, c\}, \{a, d\}, \{b, d\}\} + \{a, b\} \times N(d) + \{c, d\} \times N(a).$$

Algorithm A.1: An algorithm that checks whether a vertex v is unconfined in a graph G.

```
1  S ← {v}
2  P ← { u ∈ N(S) : |N(u) ∩ S| = 1 }
3  if P = ∅ then
4  |  u ← argmin_{u'∈P} |N(u') \ (N(S) ∪ S)|
5  |  if |N(u') \ (N(S) ∪ S)| = 0 then
6  |  |  return True
7  |  else if |N(u) \ (N(S) ∪ S)| = 1 then
8  |  |  S ← S ∪ {u}
9  |  |  go to 3
10 end
11 return False
```

While the above reductions all capture a fixed graph pattern, the applicability of the following one is determined by an iterative procedure in Algorithm A.1.

Reduction A.14 (UNCONFINED (Xiao and Nagamochi, 2013; Akiba and Iwata, 2016))**.** *If there is a vertex $v \in V(G)$ such that* CheckUnconfined(v, G)*, replace G by $G - v$.*

All of these reductions induce boundary reductions.

A.2.3 Proofs for DFVSP reductions

Theorem 5.3. *For every graph G such that (H, H', B, c) is applicable it holds that for every minimum VC S of $G+V(H)+E(H)$ there is a minimum VC S' of $G+V(H')+E(H')$ such that $|S| = |S'| + c$, and $S \cap V(G) = S' \cap V(G)$.*

Proof. Let S be a minimum vertex cover of $G + H$ and $X = B \cap S$. We know that $VC(H - X) = VC(H' - X) + c$ and that $S_l = S \cap (V(H) \setminus B)$ is a minimal vertex cover

of $H - X$. Therefore, $|S_l| = VC(H - X)$, which implies that there exists a vertex cover S'_l of $H' - X$ such that $|S_l| = |S'_l| + c$. Then, $S' = S \cap V(G) \cup S'_l$ is a vertex cover of $G + H'$ and it holds that $|S| = |S'| + c$ and $S \cap V(G) = S' \cap V(G)$.

It remains to show that S' is also minimum. Assume that there was another vertex cover C of strictly smaller cardinality. Then we can use the same steps as above to obtain a vertex cover C' of $G + H$ of strictly smaller cardinality than S. This is a contradiction, which implies that S' is a minimum vertex cover. □

Theorem 5.4. *Let G be a digraph, $r = (H, H', B, c)$ be a boundary VCP reduction and $\Pi(G) = G' + V(H) + E(H)$ such that r is applicable to G'. If*
(i) all edges incident in G to any $v \in V(H) \setminus B$ are bi-edges and
(ii) for every arc $(u, w) \in E(G)$ at least one of the following holds:
(ii.a) (u, w) is a bi-edge or
(ii.b) $|\{u, w\} \cap B| \leq 1$ then

$$DFVS(G) = DFVS(G^*) + c,$$

where $G^* = G - (V(H) \setminus B) - B \times B + V(H') + \{\, (u, v) : \{u, v\} \in E(H') \,\}$.

Proof. The proof is analogous to that of Theorem 5.3. □

Theorem 5.6. *Let G be a loop-free digraph, such that PIE is not applicable and MANYFOLD is applicable to $v^* \in V(G)$ and G' be the graph obtained from G after MANYFOLD was applied on vertex v^*, then $DFVS(G) = DFVS(G') + |C_2|$ and given a minimum DFVS of G', we can in polynomial time compute a minimum DFVS of G.*

Proof. The main observation that is used to proof soundness for the MANYFOLD reduction in the VCP case, is that without loss of generality there are only two possible cases we need to consider. For this, first note that if v^* is not contained in a minimum DFVS of G, then the vertices in $C_1 \cup C_2 = N^G_{bi}(v^*)$ are. On the other hand, if v^* is in a minimum DFVS S of G, then still since C_1 and C_2 are dicliques, it holds that $|S \cap C_i| \geq |C_i| - 1$ for $i = 1, 2$. In fact, if $|S \cap C_i| = |C_i|$ for one of $i = 1, 2$ then we can assume that it holds for both $i = 1$ and $i = 2$, since in this case v^* must be in S and we can replace it by the missing vertex without changing the size. Therefore, w.l.o.g. for a minimum DFVS S of G it holds that either

1. $S \cap C_i = C_i$ for $i = 1, 2$ and $v^* \notin S$, or
2. $|S \cap C_i| = |C_i| - 1$ for $i = 1, 2$.

Assume now, that we are given a minimum DFVS S of G such that 1. holds. In this case, we obtain a DFVS of G' as $S' = S \setminus C_2$ and it follows that $DFVS(G) - |C_2| \geq DFVS(G')$. To see that S' is a DFVS of G' observe that every edge that G' has but not G contains a vertex from $C_1 \subseteq S \setminus C_2 = S'$.

If instead we are given a minimum DFVS S of G such that 2. holds, then let $c_i \in C_i \setminus S$. In this case, $S' = S \setminus (C_2 \cup \{v^*\})$ is a DFVS of G' and thus $DFVS(G) - |C_2 \setminus \{c_2\}| + |\{v^*\}| = DFVS(G) - |C_2| \geq DFVS(G')$. To see that S' is a DFVS of G', assume the contrary. Then there must be cycle that is not covered by S'.

We know that at at least one of (c_1, c_2) and (c_2, c_1) is in M and proceed by a case distinction on whether both are in M or not.

Case both are in M: Then there is no uncovered path between c_1 and c_2 since the reduction is applicable. Assume there is a (w.l.o.g.) uncovered cycle. Then this cycle must use the vertex c_1 since the only added arcs, which do not have a vertex in S' use c_1. Furthermore, the cycle must contain an arc between $N^G(c_2)$ and c_1 and between $N^G(c_1)$ and c_1. If only the latter holds true, then the same cycle is also present in $G - S$. If only the former holds true, then there is an equivalent cycle in $G - S$, where c_1 is replaced by c_2. Since the cycle is uncovered, the arcs must go into opposite direction, i.e., be of the form (v_2, c_1) and (c_1, v_1) or (v_1, c_1) and (c_1, v_2), where $v_i \in N^G(c_i), i = 1, 2$. Assume the first form, then we can transform the uncovered cycle into an uncovered path from c_1 to c_2 in G by going from v_2 to c_2 instead of c_1. This is a contradiction to the assumption that there are no uncovered paths between c_1 and c_2. The argument for the latter form is analogous.

Case only (c_1, c_2) is in M. Thus, every uncovered path in G from c_2 to c_1 uses the arc (c_2, c_1). This implies that a cycle $c_1 \dots c_1$ in $G' - S'$ is also a cycle in $G - S$, there is a corresponding cycle with c_1 replaced by c_2 in G, or there is a corresponding cycle $c_1 \dots c_2 c_1$ in $G - S$. This a contradiction to the assumption that S is DFVS of G.

Case only (c_2, c_1) is in M. Thus, every uncovered path in G from c_1 to c_2 uses the arc (c_1, c_2). This implies that a cycle $c_1 \dots c_1$ in $G' - S'$ is also a cycle in $G - S$, there is a corresponding cycle with c_1 replaced by c_2 in G, or there is a corresponding cycle $c_1 c_2 \dots c_1$ in $G - S$. This a contradiction to the assumption that S is DFVS of G.

Thus, it follows that $DFVS(G) \geq DFVS(G') + |C_2|$.

As for the other direction, let S' be a minimum DFVS of G'. Again, we consider two cases, namely $|S' \cap C_1| = |C_1|$ and $|S' \cap C_1| = |C_1| - 1$, which are the only possible cases for the cardinality of the intersection.

Case $|S' \cap C_1| = |C_1|$: Then $S = S' \cup C_2$ is a DFVS of G, which implies $DFVS(G) \leq DFVS(G') + |C_2|$. To see that S is a DFVS of G, observe that $G - S - v^*$ and $G' - S'$ are equal and $N(v^*) \subseteq C_1 \cup C_2 \subseteq S$.

Case $|S' \cap C_1| = |C_1| - 1$: Let $c_1 \in C_1 \setminus S'$ and $c_2 \in C_2$ the unique vertex such that there is no bi-edge between c_1 and c_2 in G. Then $S = S' \cup (C_2 \setminus \{c_2\}) \cup \{v^*\}$ is a DFVS of G, which implies $DFVS(G) \leq DFVS(G') + |C_2|$. It remains to show that S is a DFVS of G. We consider two subcases for the number of arcs that use c_1 and c_2 in M.

Case only one of (c_1, c_2) is in M or (c_2, c_1) is in M. Assume (c_1, c_2) is in M, the other case works analogously. Thus, every uncovered path in G from c_2 to c_1 uses the arc

(c_1, c_2). Assume that there is a (w.l.o.g.) uncovered cycle in $G - S$. We know that this cycle must use c_1 and is thus of the form $c_1 \dots c_1$. Furthermore it must use $c2$, which implies that there is an uncovered path from c_2 to c_1 as a part of the cycle. This the cycle is actually of the form $c_1 \dots c_2 c_1$. Then however, there is a corresponding cycle $c_1 \dots c_1$ in $G' - S'$ since all the arcs of c_2 were added to c_1, which is a contradiction.

Case both (c_1, c_2) and (c_2, c_1) are in M. Assume that there is a (w.l.o.g.) uncovered cycle in $G - S$. As in the previous case, we know that the cycle must use both c_1 and c_2 since it is otherwise also a cycle in $G' - S'$. Thus, this cycle gives us an uncovered path from c_1 to c_2, which is a contradiction to the assumption on M.

Since these are the only cases, we are done and $DFVS(G) \leq DFVS(G') + |C_2|$, meaning that overall $DFVS(G) = DFVS(G') + |C_2|$. Furthermore, the constructions used in the proof are possible in polynomial time, which was the second claim of the theorem. □

Theorem 5.7. *Let G be a digraph such that 4PATH is applicable to $v^* \in V(G)$ and G' be the graph obtained from G after 4PATH was applied to the vertex v^*, then*

- *$DFVS(G) = DFVS(G')$, and*
- *given a minimum DFVS of G', we can in polynomial time compute a minimum DFVS of G.*

Proof. Let S be a minimum DFVS of G. If $N^G(v^*) \subseteq S$, then S is also a DFVS of G' and $DFVS(G) \geq DFVS(G')$ since every added arc uses a vertex from $N^G(v^*)$. Otherwise, we can assume w.l.o.g. that $|N^G(v^*) \cap S| = 2$ and $v \in S$, since for every minimum DFVS of G, with $|N^G(v^*) \cap S| = 3$, there is a minimum DFVS S^* with $N^G(v^*) \subseteq S^*$ of the same size. Furthermore, there cannot be a DFVS of G, which contains less than two elements from $N^G(v^*)$, since the elements in $N^G(v^*)$ form a path of four elements, where every arc is a bi-edge. So let S be a minimum DFVS of G such that $|N^G(v^*) \cap S| = 2$ and $v \in S$. We proceed by case distinction over the two neighbors of v^* that are in S.

Case $N(v^*) \cap S = \{b, c\}$: In this case $S' = (S \setminus \{v^*\}) \cup \{a\}$ is a DFVS of G' and $DFVS(G) \geq DFVS(G')$. Assume that on the contrary, there is a (w.l.o.g.) uncovered cycle. Then this cycle must use the vertex d since the only added arcs, which do not have a vertex in S' use d. Furthermore, the cycle must contain an arc between $N^G(a)$ and d and between $N^G(d)$ and d. If only the latter holds true, then the same cycle is also present in $G - S$. If only the former holds true, then there is an equivalent cycle in $G - S$, where d is replaced by a. Since the cycle is uncovered, the arcs must go into opposite direction, i.e., be of the form (v_a, d) and (d, v_d) or (v_d, d) and (d, v_a). Assume the first form, then we can transform the uncovered cycle into an uncovered path from d to a in G by going from v_a to a instead of d. This is a contradiction to the assumption that there are no uncovered paths between a and d. The argument for the latter form is analogous.

Case $N(v^*) \cap S = \{a, c\}$: In this case $S' = (S \setminus \{v^*\}) \cup \{d\}$ is a DFVS of G' and $DFVS(G) \geq DFVS(G')$. The argument showing that S' is a DFVS of G' is analogous to that of the previous case.

Case $N(v^*) \cap S = \{b, d\}$: In this case $S' = (S \setminus \{v^*\}) \cup \{a\}$ is a DFVS of G' and $DFVS(G) \geq DFVS(G')$. The argument showing that S' is a DFVS of G' is analogous to that of the first case.

As for the other direction, let S' be a minimum DFVS of G'. If $N^G(v^*) \subseteq S'$, then S' is also a DFVS of G and $DFVS(G) \leq DFVS(G')$. Otherwise, we know that $|N^G(v^*) \cap S'| = 3$ since $G'[N^G(v^*)]$ is a diclique.

Case $N^G(v^*) \cap S' = \{a, b, c\}$: Then $S = (S' \setminus \{a\}) \cup \{v^*\}$ is a DFVS of G and $DFVS(G) \leq DFVS(G')$. Assume that on the contrary there is a cycle. Then this cycle must contain a. However, since d has the same neighbors as a in G' this implies the existence of an equivalent cycle with a replaced by d in G', which is a contradiction to $d \notin S'$.

Case $N^G(v^*) \cap S' = \{a, b, d\}$: Then $S = (S' \setminus \{a\}) \cup \{v^*\}$ is a DFVS of G and $DFVS(G) \leq DFVS(G')$ by an analogous argument as above.

The other two cases follow from symmetric arguments. □

In order to prove soundness of UNCONFINED, we use the following definitions inspired by (Xiao and Nagamochi, 2013).

For a set $A \subseteq V(G)$ such that $G[A]$ is acyclic let

$$N_c(A) = \{\, u \in V(G) : G[A \cup \{u\}] \text{ is cyclic} \,\},$$

i.e., the neighbors of any vertex in A such that there is a cycle through A and the vertex. A vertex $u \in N_c(A)$ is called a directed child of A if it has exactly two arcs that are shared with vertices in A (i.e., $|N_{succ}(u) \cap A| + |N_{pred}(u) \cap A| = 2$). The vertices in A that share arcs with u are called its *parents*.

Lemma A.14. *Let A be a set of vertices from G such that*

- *$G[A]$ is acyclic, and*
- *for every minimum DFVS S of G it holds that $A \cap S = \emptyset$.*

Then for each directed child u of A no minimum DFVS of G contains all vertices $w \in N(u) \setminus (N_c(A) \cup A)$.

Proof. Assume that there is a minimum DFVS S of G such that $S \cap (N(u) \setminus (N_c(A) \cup A)) = (N(u) \setminus (N_c(A) \cup A))$ for some directed child $u \in N_c(A)$. The parents p_1, p_2 of u are in A and thus by assumption not in S. Furthermore, since S is a DFVS and $A \cap S = \emptyset$, we know that $N_c(A) \subseteq S$ since $N_c(A)$ contains the vertices v such that $G[A \cup \{v\}]$ is cyclic.

It follows that $N(u) \setminus A \subseteq S$ and thus $N(u) \setminus A \subseteq S'$, where $S' = (S \cup \{p_1\}) \setminus \{u\}$ (or $(S \cup \{p_2\}) \setminus \{u\}$). Recall that u is a directed child of A and therefore only has two arcs that go from or to A, which use the parents. Since we put one of u's parents into S' there is at most one arc that uses u in the graph $G - S'$, implying that u cannot be contained in any cycle in $G - S'$. Thus, the removal of u can be compensated by the addition of p_1 (or p_2) and we see that S' is a minimum DFVS of G, which shares a vertex with A. This is a contradiction. □

Theorem 5.8. *Let G be a digraph. After applying UNCONFINED to vertex v resulting in G', it holds that for every minimum DFVS S of G' the set $S \cup \{v\}$ is a minimum DFVS of G.*

Proof. The result follows from Lemma A.14. The idea of Algorithm 5.2 is the following: We start by assuming that $S = \{v\}$ has an empty intersection with any minimum DFVS. If this leads to a contradiction together with Lemma A.14 then v must be contained in some DFVS and the theorem follows.

In particular, we get a contradiction, when there is a directed child u such that $|N(u) \setminus (N_c(A) \cup S)| = 0$. In this case, we know that there is a DFVS of G that contains v. If there is no immediate contradiction but there is a $u \in N_c(A)$ with $|N(u) \setminus (N_c(A) \cup S)| = 1$, then we know that the unique vertex $v' \in N(u) \setminus (N_c(A) \cup S)$ must also not be contained in any minimum DFVS of G, which means that we can extend S by adding v'. □

Bibliography

Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Learning optimal decision trees using caching branch-and-bound search. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 3146–3153. AAAI Press, 2020. URL https://ojs.aaai.org/index.php/AAAI/article/view/5711.

Jungho Ahn, Kevin Hendrey, Donggyu Kim, and Sang-il Oum. Bounds for the twin-width of graphs. *CoRR*, abs/2110.03957, 2021. URL https://arxiv.org/abs/2110.03957.

Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover. *Theoretical Computer Science*, 609:211–225, 2016. URL https://doi.org/10.1016/j.tcs.2015.09.023.

Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In Oliver Kullmann, editor, *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 167–180. Springer, 2009. doi: 10.1007/978-3-642-02777-2_18. URL https://doi.org/10.1007/978-3-642-02777-2_18.

Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI 2009*, pages 399–404, 2009. URL http://ijcai.org/Proceedings/09/Papers/074.pdf.

Florent Avellaneda. Efficient inference of optimal decision trees. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 3195–3202. AAAI Press, 2020a. URL https://ojs.aaai.org/index.php/AAAI/article/view/5717.

Florent Avellaneda. A short description of the solver EvalMaxSAT. In *MaxSAT Evaluation 2020*, pages 8–9, 07 2020b. URL http://florent.avellaneda.free.fr/dl/EvalMaxSAT.pdf.

Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer*

Science, pages 108–122. Springer, 2003. doi: 10.1007/978-3-540-45193-8_8. URL https://doi.org/10.1007/978-3-540-45193-8_8.

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. doi: 10.4230/LIPIcs.SEA.2017.28. URL https://doi.org/10.4230/LIPIcs.SEA.2017.28.

Yu Bao, Morihiro Hayashida, Pengyu Liu, Masayuki Ishitsuka, Jose C Nacher, and Tatsuya Akutsu. Analysis of critical and redundant vertices in controlling directed complex networks using feedback vertex sets. *Journal of Computational Biology*, 25 (10):1071–1090, 2018. URL https://doi.org/10.1089/cmb.2018.0019.

Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185, pages 825–885. IOS Press, 2009. doi: 10.3233/978-1-58603-929-5-825. URL https://doi.org/10.3233/978-1-58603-929-5-825.

Kenneth E. Batcher. Sorting networks and their applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May 1968*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968. doi: 10.1145/1468075.1468121. URL https://doi.org/10.1145/1468075.1468121.

Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. Benchmarking the chase. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, page 37–52, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341981. doi: 10.1145/3034786.3034796. URL https://doi.org/10.1145/3034786.3034796.

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. doi: 10.1109/ICTAI.2014.57. URL https://doi.org/10.1109/ICTAI.2014.57.

Jeremias Berg, Neha Lodha, Matti Järvisalo, and Stefan Szeider. MaxSAT benchmarks based on determining generalized hypertree-width. In *MaxSAT Evaluation 2017*, page 22. Department of Computer Science, University of Helsinki, 2017.

Jeremias Berg, Fahiem Bacchus, and Alex Poole. Abstract cores in implicit hitting set MaxSat solving. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 277–294. Springer, 2020. doi: 10.1007/978-3-030-51825-7_20. URL https://doi.org/10.1007/978-3-030-51825-7_20.

Pierre Bergé, Édouard Bonnet, and Hugues Déprés. Deciding twin-width at most 4 is NP-complete. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 18:1–18:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.ICALP.2022.18. URL https://doi.org/10.4230/LIPIcs.ICALP.2022.18.

Dimitris Bertsimas and Jack Dunn. Optimal classification trees. *Mach. Learn.*, 106 (7):1039–1082, 2017. doi: 10.1007/s10994-017-5633-9. URL https://doi.org/10.1007/s10994-017-5633-9.

Christian Bessiere, Emmanuel Hebrard, and Barry O'Sullivan. Minimising decision tree size as combinatorial optimisation. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2009. doi: 10.1007/978-3-642-04244-7_16. URL https://doi.org/10.1007/978-3-642-04244-7_16.

Armin Biere. Bounded model checking. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 739–764. IOS Press, 2021. doi: 10.3233/FAIA201002. URL https://doi.org/10.3233/FAIA201002.

Armin Biere, Edmund M. Clarke, Richard Raimi, and Yunshan Zhu. Verifiying safety properties of a power PC microprocessor using symbolic model checking without bdds. In Nicolas Halbwachs and Doron A. Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1999. doi: 10.1007/3-540-48683-6_8. URL https://doi.org/10.1007/3-540-48683-6_8.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Adv. Comput.*, 58:117–148, 2003. doi: 10.1016/S0065-2458(03)58003-2. URL https://doi.org/10.1016/S0065-2458(03)58003-2.

Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53.

University of Helsinki, 2020. URL https://researchportal.helsinki.fi/files/142452772/sc2020_proceedings.pdf.

Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021. ISBN 978-1-64368-160-3. doi: 10.3233/FAIA336. URL https://doi.org/10.3233/FAIA336.

Ivo Blöchliger and Nicolas Zufferey. A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Comput. Oper. Res.*, 35(3):960–975, 2008. doi: 10.1016/j.cor.2006.05.014. URL https://doi.org/10.1016/j.cor.2006.05.014.

Hans L. Bodlaender. Discovering treewidth. In Peter Vojtás, Mária Bieliková, Bernadette Charron-Bost, and Ondrej Sýkora, editors, *SOFSEM 2005: Theory and Practice of Computer Science, 31st Conference on Current Trends in Theory and Practice of Computer Science, Liptovský Ján, Slovakia, January 22-28, 2005, Proceedings*, volume 3381 of *LNCS*, pages 1–16. Springer Verlag, 2005. doi: 10.1007/978-3-540-30577-4_1. URL https://doi.org/10.1007/978-3-540-30577-4_1.

Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *Proc. VLDB Endowment*, 11(2):149–161, October 2017. ISSN 2150-8097. doi: 10.14778/3149193.3149196. URL http://www.vldb.org/pvldb/vol11/p149-bonifati.pdf.

Angela Bonifati, Wim Martens, and Thomas Timm. Navigating the maze of wikidata query logs. In Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia, editors, *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, page 127–138, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366748. doi: 10.1145/3308558.3313472. URL https://doi.org/10.1145/3308558.3313472.

Édouard Bonnet, Colin Geniet, Eun Jung Kim, Stéphan Thomassé, and Rémi Watrigant. Twin-width II: small classes. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 1977–1996. SIAM, 2021a. doi: 10.1137/1.9781611976465.118. URL https://doi.org/10.1137/1.9781611976465.118.

Édouard Bonnet, Colin Geniet, Eun Jung Kim, Stéphan Thomassé, and Rémi Watrigant. Twin-width III: max independent set, min dominating set, and coloring. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 35:1–35:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021b. doi: 10.4230/LIPIcs.ICALP.2021.35. URL https://doi.org/10.4230/LIPIcs.ICALP.2021.35.

Édouard Bonnet, Eun Jung Kim, Stéphan Thomassé, and Rémi Watrigant. Twin-width I: tractable FO model checking. *J. ACM*, 69(1):3:1–3:46, 2022. doi: 10.1145/3486655. URL https://doi.org/10.1145/3486655.

Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984. ISBN 0-534-98053-8. doi: 10.1201/9781315139470. URL https://doi.org/10.1201/9781315139470.

Daniel Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22 (4):251–256, apr 1979. ISSN 0001-0782. doi: 10.1145/359094.359101. URL https://doi.org/10.1145/359094.359101.

Robert Brummayer and Armin Biere. Effective bit-width and under-approximation. In *EUROCAST 2009*, volume 5717 of *Lecture Notes in Computer Science*, pages 304–311. Springer, 2009. URL https://doi.org/10.1007/978-3-642-04772-5_40.

Edmund K. Burke, Jakub Marecek, Andrew J. Parkes, and Hana Rudová. A supernodal formulation of vertex colouring with applications in course timetabling. *Ann. Oper. Res.*, 179(1):105–130, 2010. doi: 10.1007/s10479-010-0716-z. URL https://doi.org/10.1007/s10479-010-0716-z.

Jianer Chen, Yang Liu, Songjian Lu, Barry O'sullivan, and Igor Razgon. A fixed-parameter algorithm for the directed feedback vertex set problem. In *STOC 2008*, pages 177–186, 2008. URL https://doi.org/10.1145/1374376.1374404.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003. URL https://doi.org/10.1145/876638.876643.

Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971. doi: 10.1145/800157.805047. URL https://doi.org/10.1145/800157.805047.

Vinícius G Costa and Carlos E Pedreira. Recent advances in decision trees: An updated survey. *Artificial Intelligence Review*, pages 1–36, 2022.

Alain Cournier and Michel Habib. A new linear algorithm for modular decomposition. In Sophie Tison, editor, *Trees in Algebra and Programming - CAAP'94, 19th International Colloquium, Edinburgh, UK, April 11-13, 1994, Proceedings*, volume 787 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 1994. doi: 10.1007/BFb0017474. URL https://doi.org/10.1007/BFb0017474.

Loïc Crombez, Guilherme Dias da Fonseca, Yan Gerard, and Aldo Gonzalez-Lorenzo. Shadoks approach to minimum partition into plane subgraphs (CG challenge). In Xavier Goaoc and Michael Kerber, editors, *38th International Symposium on Computational Geometry, SoCG 2022, June 7-10, 2022, Berlin, Germany*, volume 224 of *LIPIcs*, pages 71:1–71:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.SoCG.2022.71. URL https://doi.org/10.4230/LIPIcs.SoCG.2022.71.

Adnan Darwiche and Auguste Hirth. On the reasons behind decisions. In *ECAI 2020*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 712–720. IOS Press, 2020. doi: 10.3233/FAIA200158. URL https://doi.org/10.3233/FAIA200158.

Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962. doi: 10.1145/368273.368557. URL https://doi.org/10.1145/368273.368557.

Emir Demirovic and Peter J. Stuckey. Optimal decision trees for nonlinear metrics. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Virtual Event, February 2-9, 2021*, pages 3733–3741. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/AAAI/article/view/16490.

Emir Demirovic, Anna Lukina, Emmanuel Hebrard, Jeffrey Chan, James Bailey, Christopher Leckie, Kotagiri Ramamohanarao, and Peter J. Stuckey. Murtree: Optimal decision trees via dynamic programming and search. *J. Mach. Learn. Res.*, 23:26:1–26:47, 2022. URL http://jmlr.org/papers/v23/20-520.html.

Finale Doshi-Velez and Been Kim. A roadmap for a rigorous science of interpretability. *CoRR*, abs/1702.08608, 2017. URL http://arxiv.org/abs/1702.08608.

Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013. ISBN 978-1-4471-5558-4. doi: 10.1007/978-1-4471-5559-1. URL https://doi.org/10.1007/978-1-4471-5559-1.

Wolfgang Dvorák, Sebastian Ordyniak, and Stefan Szeider. Augmenting tractable fragments of abstract argumentation. *Artif. Intell.*, 186:157–173, 2012. URL https://doi.org/10.1016/j.artint.2012.03.002.

Wolfgang Dvorák, Markus Hecher, Matthias König, André Schidler, Stefan Szeider, and Stefan Woltran. Tractable abstract argumentation via backdoor-treewidth. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Virtual Event, February 22 - March 1, 2022*, pages 5608–5615. AAAI Press, 2022. URL https://ojs.aaai.org/index.php/AAAI/article/view/20501.

M. Ayaz Dzulfikar, Johannes K. Fichte, and Markus Hecher. The PACE 2019 Parameterized Algorithms and Computational Experiments Challenge: The Fourth Iteration (Invited Paper). In Bart M. P. Jansen and Jan Arne Telle, editors, *14th International Symposium on Parameterized and Exact Computation (IPEC 2019)*, volume 148 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:23, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-129-0. doi: 10.4230/LIPIcs.IPEC.2019.25. URL https://drops.dagstuhl.de/opus/volltexte/2019/11486.

Guy Even, B Schieber, M Sudan, et al. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20(2):151–174, 1998. URL https://doi.org/10.1007/PL00009191.

François Fages and Akash Lal. A constraint programming approach to cutset problems. *Computers & Operations Research*, 33(10):2852–2865, 2006. URL https://doi.org/10.1016/j.cor.2005.01.014.

Sándor P. Fekete, Phillip Keldenich, Dominik Krupke, and Stefan Schirra. Minimum partition into plane subgraphs: The CG: SHOP Challenge 2022. *CoRR*, abs/2203.07444, 2022. URL https://arxiv.org/abs/2203.07444.

Michael R. Fellows, Fedor V. Fomin, Daniel Lokshtanov, Frances A. Rosamond, Saket Saurabh, Stefan Szeider, and Carsten Thomassen. On the complexity of some colorful problems parameterized by treewidth. *Inf. Comput.*, 209(2):143–153, 2011. doi: 10.1016/j.ic.2010.11.026. URL https://doi.org/10.1016/j.ic.2010.11.026.

Michael R. Fellows, Lars Jaffke, Aliz Izabella Király, Frances A. Rosamond, and Mathias Weller. What is known about vertex cover kernelization? In *Adventures Between Lower Bounds and Higher Altitudes - Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday*, volume 11011 of *Lecture Notes in Computer Science*, pages 330–356. Springer, 2018. doi: 10.1007/978-3-319-98355-4_19. URL https://doi.org/10.1007/978-3-319-98355-4_19.

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. doi: 10.1007/978-3-319-66263-3_25. URL https://doi.org/10.1007/978-3-319-66263-3_25.

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. doi: 10.1007/978-3-319-98334-9_8. URL https://doi.org/10.1007/978-3-319-98334-9_8.

Johannes Klaus Fichte, Markus Hecher, and Stefan Szeider. Breaking symmetries with rootclique and lextopsort. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 286–303. Springer, 2020. doi: 10.1007/978-3-030-58475-7_17. URL https://doi.org/10.1007/978-3-030-58475-7_17.

Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. General and fractional hypertree decompositions: Hard and easy cases. In Jan Van den Bussche and Marcelo Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 17–32. ACM, 2018a. doi: 10.1145/3196959.3196962. URL https://doi.org/10.1145/3196959.3196962.

Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. General and Fractional Hypertree Decompositions. In Jan Van den Bussche and Marcelo Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 17–32, New York, New York, USA, 2018b. ACM Press. ISBN 9781450347068. doi: 10.1145/3196959.3196962. URL https://doi.org/10.1145/3196959.3196962.

Wolfgang Fischl, Georg Gottlob, Davide Mario Longo, and Reinhard Pichler. Hyperbench: A benchmark and tool for hypergraphs and empirical findings. In Dan Suciu, Sebastian Skritek, and Christoph Koch, editors, *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 464–480. Assoc. Comput. Mach., New York, 2019. doi: 10.1145/3294052.3319683. URL https://doi.org/10.1145/3294052.3319683.

Rudolf Fleischer, Xi Wu, and Liwei Yuan. Experimental study of FPT algorithms for the directed feedback vertex set problem. In *ESA 2009*, pages 611–622. Springer, 2009. URL https://doi.org/10.1007/978-3-642-04128-0_55.

Florian Fontan, Pascal Lafourcade, Luc Libralesso, and Benjamin Momège. Local search with weighting schemes for the CG: SHOP 2022 competition (CG challenge). In Xavier Goaoc and Michael Kerber, editors, *38th International Symposium on Computational Geometry, SoCG 2022, June 7-10, 2022, Berlin, Germany*, volume 224 of *LIPIcs*, pages 73:1–73:6. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.SoCG.2022.73. URL https://doi.org/10.4230/LIPIcs.SoCG.2022.73.

Eibe Frank, Mark A. Hall, Geoffrey Holmes, Richard Kirkby, and Bernhard Pfahringer. WEKA - A machine learning workbench for data mining. In Oded Maimon and Lior Rokach, editors, *The Data Mining and Knowledge Discovery Handbook*, pages 1305–1314. Springer, 2005.

Meinrad Funke and Gerhard Reinelt. A polyhedral approach to the feedback vertex set problem. In *IPCO 1996*, pages 445–459. Springer, 1996. URL https://doi.org/10.1007/3-540-61310-2_33.

Tibor Gallai. Transitiv orientierbare graphen. *Acta Math. Acad. Sci. Hung.*, 18:25–66, 1967. doi: 10.1007/BF02020961. URL https://doi.org/10.1007/BF02020961.

Robert Ganian, Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. SAT-encodings for treecut width and treedepth. In Stephen G. Kobourov and Henning Meyerhenke, editors, *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*, pages 117–129. SIAM, 2019. doi: 10.1137/1.9781611975499.10. URL https://doi.org/10.1137/1.9781611975499.10.

Gael Glorian, Jean-Marie Lagniez, Valentin Montmirail, and Nicolas Szczepanski. An incremental SAT-based approach to the graph colouring problem. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming - 25th*

International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings, volume 11802 of *Lecture Notes in Computer Science*, pages 213–231. Springer, 2019. doi: 10.1007/978-3-030-30048-7_13. URL https://doi.org/10.1007/978-3-030-30048-7_13.

Christopher D. Godsil and Gordon F. Royle. *Algebraic Graph Theory*. Graduate texts in mathematics. Springer, 2001. ISBN 978-0-387-95220-8. doi: 10.1007/978-1-4613-0163-9. URL https://doi.org/10.1007/978-1-4613-0163-9.

Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. Manthan: A data-driven approach for boolean function synthesis. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 611–633. Springer, 2020. doi: 10.1007/978-3-030-53291-8_31. URL https://doi.org/10.1007/978-3-030-53291-8_31.

Priyanka Golia, Friedrich Slivovsky, Subhajit Roy, and Kuldeep S. Meel. Engineering an efficient boolean functional synthesis engine. In *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2021, Munich, Germany, November 1-4, 2021*, pages 1–9. IEEE, 2021. doi: 10.1109/ICCAD51958.2021.9643583. URL https://doi.org/10.1109/ICCAD51958.2021.9643583.

Bryce Goodman and Seth R. Flaxman. European union regulations on algorithmic decision-making and a "right to explanation". *AI Mag.*, 38(3):50–57, 2017. doi: 10.1609/aimag.v38i3.2741. URL https://doi.org/10.1609/aimag.v38i3.2741.

Georg Gottlob and Marko Samer. A backtracking-based algorithm for hypertree decomposition. *J. Exp. Algorithmics*, 13:1:1.1–1:1.19, February 2009. ISSN 1084-6654. doi: 10.1145/1412228.1412229. URL https://doi.org/10.1145/1412228.1412229.

Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. of Computer and System Sciences*, 64(3):579–627, 2002. doi: 10.1006/jcss.2001.1809. URL https://doi.org/10.1006/jcss.2001.1809.

Georg Gottlob, Nicola Leone, and Francesco Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. *J. of Computer and System Sciences*, 66(4):775–808, 2003. doi: 10.1016/S0022-0000(03)00030-8. URL https://doi.org/10.1016/S0022-0000(03)00030-8.

Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In Dieter Kratsch, editor, *Graph-Theoretic Concepts in Computer Science, 31st International Workshop, WG 2005, Metz, France, June 23-25, 2005, Revised Selected Papers*, volume 3787 of *LNCS*, pages 1–15. Springer Verlag, 2005. doi: 10.1007/11604686_1. URL https://doi.org/10.1007/11604686_1.

Georg Gottlob, Gianluigi Greco, and Francesco Scarcello. Treewidth and hypertree width. In Lucas Bordeaux, Youssef Hamadi, and Pushmeet Kohli, editors, *Tractability: Practical Approaches to Hard Problems*, pages 3–38. Cambridge University Press, 2014. doi: 10.1017/CBO9781139177801.002. URL https://doi.org/10.1017/CBO9781139177801.002.

Georg Gottlob, Cem Okulmus, and Reinhard Pichler. Fast and parallel decomposition of constraint satisfaction problems. *Constraints*, 2022. doi: 10.1007/s10601-022-09332-1. URL https://doi.org/10.1007/s10601-022-09332-1.

Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Transactions on Algorithms*, 11(1):Art. 4, 20, 2014. doi: 10.1145/2636918. URL https://doi.org/10.1145/2636918.

Martin Grohe, Stephan Kreutzer, and Sebastian Siebertz. Deciding first-order properties of nowhere dense graphs. *J. ACM*, 64(3):17:1–17:32, 2017. doi: 10.1145/3051095. URL https://doi.org/10.1145/3051095.

Sylvain Guillemot and Dániel Marx. Finding small patterns in permutations in linear time. In Chandra Chekuri, editor, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 82–101. SIAM, 2014. doi: 10.1137/1.9781611973402.7. URL https://doi.org/10.1137/1.9781611973402.7.

Michel Habib and Christophe Paul. A survey of the algorithmic aspects of modular decomposition. *Comput. Sci. Rev.*, 4(1):41–59, 2010. doi: 10.1016/j.cosrev.2010.01.001. URL https://doi.org/10.1016/j.cosrev.2010.01.001.

Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39:297–308, 1985. doi: 10.1016/0304-3975(85)90144-6. URL https://doi.org/10.1016/0304-3975(85)90144-6.

Jin-Kao Hao and Qinghua Wu. Improving the extraction and expansion method for large graph coloring. *Discret. Appl. Math.*, 160(16-17):2397–2407, 2012. doi: 10.1016/j.dam.2012.06.007. URL https://doi.org/10.1016/j.dam.2012.06.007.

Emmanuel Hebrard and George Katsirelos. A hybrid approach for exact coloring of massive graphs. In Louis-Martin Rousseau and Kostas Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, volume 11494 of *Lecture Notes in Computer Science*, pages 374–390. Springer, 2019a. doi: 10.1007/978-3-030-19212-9_25. URL https://doi.org/10.1007/978-3-030-19212-9_25.

Emmanuel Hebrard and George Katsirelos. Clause learning and new bounds for graph coloring. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International*

Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019, pages 6166–6170. ijcai.org, 2019b. doi: 10.24963/ijcai.2019/856. URL https://doi.org/10.24963/ijcai.2019/856.

Marijn Heule and Stefan Szeider. A SAT approach to clique-width. *ACM Trans. Comput. Log.*, 16(3):24, 2015. doi: 10.1145/2736696. URL https://doi.org/10.1145/2736696.

Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, volume 7261 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2011. doi: 10.1007/978-3-642-34188-5_8. URL https://doi.org/10.1007/978-3-642-34188-5_8.

Marijn J. H. Heule. Schur number five. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, pages 6598–6606. AAAI Press, 2018. URL https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16952.

Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016. doi: 10.1007/978-3-319-40970-2_15. URL https://doi.org/10.1007/978-3-319-40970-2_15.

John Hooker. *Logic-Based Benders Decomposition*, chapter 19, pages 389–422. John Wiley & Sons, Ltd, 2000. ISBN 9781118033036. doi: https://doi.org/10.1002/9781118033036.ch19. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118033036.ch19.

Hao Hu, Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. Learning optimal decision trees with MaxSAT and its integration in adaboost. In *Proceedings of IJCAI 2020*, pages 1170–1176. ijcai.org, 7 2020. doi: 10.24963/ijcai.2020/163. URL https://doi.org/10.24963/ijcai.2020/163. Main track.

Xiyang Hu, Cynthia Rudin, and Margo I. Seltzer. Optimal sparse decision trees. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 7265–7273, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/ac52c626afc10d4075708ac4c778ddfc-Abstract.html.

Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is NP-complete. *Inf. Process. Lett.*, 5(1):15–17, 1976. doi: 10.1016/0020-0190(76)90095-8. URL https://doi.org/10.1016/0020-0190(76)90095-8.

Toshihide Ibaraki, Yves Crama, and Peter L. Hammer. *Partially defined Boolean functions*, page 511–563. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2011. doi: 10.1017/CBO9780511852008.013. URL https://www.doi.org/10.1017/CBO9780511852008.013.

Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pages 428–437, 2018. doi: 10.1007/978-3-319-94144-8_26. URL https://doi.org/10.1007/978-3-319-94144-8_26.

Alexey Ignatiev, João Marques-Silva, Nina Narodytska, and Peter J. Stuckey. Reasoning-based learning of interpretable ML models. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 4458–4465. ijcai.org, 2021. doi: 10.24963/ijcai.2021/608. URL https://doi.org/10.24963/ijcai.2021/608.

Mikolás Janota and António Morgado. SAT-based encodings for optimal decision trees with explicit paths. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 501–518. Springer, 2020. doi: 10.1007/978-3-030-51825-7_35. URL https://doi.org/10.1007/978-3-030-51825-7_35.

Mikolás Janota, Radu Grigore, and João Marques-Silva. Counterexample guided abstraction refinement algorithm for propositional circumscription. In Tomi Janhunen and Ilkka Niemelä, editors, *Logics in Artificial Intelligence - 12th European Conference, JELIA 2010, Helsinki, Finland, September 13-15, 2010. Proceedings*, volume 6341 of *Lecture Notes in Computer Science*, pages 195–207. Springer, 2010. doi: 10.1007/978-3-642-15675-5_18. URL https://doi.org/10.1007/978-3-642-15675-5_18.

Mikolás Janota, William Klieber, João Marques-Silva, and Edmund M. Clarke. Solving QBF with counterexample guided refinement. *Artif. Intell.*, 234:1–25, 2016. doi: 10.1016/j.artint.2016.01.004. URL https://doi.org/10.1016/j.artint.2016.01.004.

Mikolás Janota, Radu Grigore, and Vasco M. Manquinho. On the quest for an acyclic graph. In *RCRA@AI*IA 2017*, volume 2011 of *CEUR Workshop Proceedings*, pages 33–44. CEUR-WS.org, 2017. URL http://ceur-ws.org/Vol-2011/paper4.pdf.

Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi: 10.1007/978-1-4684-2001-2_9. URL https://doi.org/10.1007/978-1-4684-2001-2_9.

Rafael Kiesel and André Schidler. A dynamic maxsat-based approach to directed feedback vertex sets. In Gonzalo Navarro and Julian Shun, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2023, Florence, Italy, January 22-23, 2023*, pages 39–52. SIAM, 2023. doi: 10.1137/1.9781611977561.ch4. URL https://doi.org/10.1137/1.9781611977561.ch4.

Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 34:1–34:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPIcs.CP.2021.34. URL https://doi.org/10.4230/LIPIcs.CP.2021.34.

Henning Koehler. A contraction algorithm for finding minimal feedback sets. In *ACSC 38*, pages 165–173, 2005. URL http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV38Koehler.html.

Tuukka Korhonen, Jeremias Berg, and Matti Järvisalo. Solving graph problems via potential maximal cliques: An experimental evaluation of the Bouchitté–Todinca algorithm. *J. Exp. Algorithmics*, 24(1), February 2019. ISSN 1084-6654. doi: 10.1145/3301297. URL https://doi.org/10.1145/3301297.

Daniel Kroening. Software verification. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 791–818. IOS Press, 2021. doi: 10.3233/FAIA201004. URL https://doi.org/10.3233/FAIA201004.

Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 21(12):1377–1394, 2002. doi: 10.1109/TCAD.2002.804386. URL https://doi.org/10.1109/TCAD.2002.804386.

Alexander S. Kulikov, Danila Pechenev, and Nikita Slezkin. SAT-based circuit local improvement. In Stefan Szeider, Robert Ganian, and Alexandra Silva, editors, *47th International Symposium on Mathematical Foundations of Computer Science, MFCS 2022, August 22-26, 2022, Vienna, Austria*, volume 241 of *LIPIcs*, pages 67:1–67:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.MFCS.2022.67. URL https://doi.org/10.4230/LIPIcs.MFCS.2022.67.

Jean-Marie Lagniez, Daniel Le Berre, Tiago de Lima, and Valentin Montmirail. A recursive shortcut for CEGAR: application to the modal logic K satisfiability problem. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 674–680. ijcai.org, 2017. doi: 10.24963/ijcai.2017/94. URL https://doi.org/10.24963/ijcai.2017/94.

Daniel T. Larose. *Discovering knowledge in data*. Wiley-Interscience [John Wiley & Sons], Hoboken, NJ, 2005. ISBN 0-471-66657-2. An introduction to data mining.

Mile Lemaic. *Markov-Chain-Based Heuristics for the Feedback Vertex Set Problem for Digraphs*. PhD thesis, Universität zu Köln, 2008. URL https://kups.ub.uni-koeln.de/2547/.

Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8 (1):1, 2016.

Leonid Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265—266, 1973.

Hanoch Levy and David W Low. A contraction algorithm for finding small cycle cutsets. *Journal of algorithms*, 9(4):470–493, 1988. URL https://doi.org/10.1016/0196-6774(88)90013-2.

Hen-Ming Lin and Jing-Yang Jou. On computing the minimum feedback vertex set of a directed graph by contraction operations. *ICCD 1999*, 19(3):295–307, 1999. URL https://doi.org/10.1109/ICCD.1999.808567.

Jimmy Lin, Chudi Zhong, Diane Hu, Cynthia Rudin, and Margo I. Seltzer. Generalized and scalable optimal sparse decision trees. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 6150–6160. PMLR, 2020. URL http://proceedings.mlr.press/v119/lin20g.html.

Jinkun Lin, Shaowei Cai, Chuan Luo, and Kaile Su. A reduction based method for coloring very large graphs. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 517–523. ijcai.org, 2017. doi: 10.24963/ijcai.2017/73. URL https://doi.org/10.24963/ijcai.2017/73.

Zachary C. Lipton. The mythos of model interpretability. *Commun. ACM*, 61(10):36–43, 2018. doi: 10.1145/3233231. URL https://doi.org/10.1145/3233231.

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 *LNCS*, pages 429–445. Springer Verlag, 2017. doi: 10.1007/978-3-319-66263-3_27. URL https://doi.org/10.1007/978-3-319-66263-3_27.

Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. A SAT approach to branchwidth. *ACM Trans. Comput. Log.*, 20(3):15:1–15:24, 2019. doi: 10.1145/3326159. URL https://doi.org/10.1145/3326159.

Ruben Martins, Saurabh Joshi, Vasco M. Manquinho, and Inês Lynce. Incremental cardinality constraints for MaxSAT. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer, 2014. doi: 10.1007/978-3-319-10428-7_39. URL https://doi.org/10.1007/978-3-319-10428-7_39.

Ross M. McConnell and Jeremy P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. In Daniel Dominic Sleator, editor, *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. 23-25 January 1994, Arlington, Virginia, USA*, pages 536–545. ACM/SIAM, 1994. URL http://dl.acm.org/citation.cfm?id=314464.314641.

Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60(0):94–112, 2014. ISSN 0747-7171. doi: http://doi.org/10.1016/j.jsc.2013.09.003. URL http://www.sciencedirect.com/science/article/pii/S0747717113001193.

David Monniaux. A survey of satisfiability modulo theory. In Vladimir P. Gerdt, Wolfram Koepf, Werner M. Seiler, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing - 18th International Workshop, CASC 2016, Bucharest, Romania, September 19-23, 2016, Proceedings*, volume 9890 of *LNCS*, pages 401–425. Springer Verlag, 2016. doi: 10.1007/978-3-319-45641-6_26. URL https://doi.org/10.1007/978-3-319-45641-6_26.

Don Monroe. AI, explain yourself. *AI Communications*, 61(11):11–13, 2018. doi: 10.1145/3276742. URL https://doi.org/10.1145/3276742.

António Morgado, Carmine Dodaro, and João Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer, 2014a. doi: 10.1007/978-3-319-10428-7_41. URL https://doi.org/10.1007/978-3-319-10428-7_41.

António Morgado, Alexey Ignatiev, and João Marques-Silva. MSCG: robust core-guided MaxSAT solving. *J. Satisf. Boolean Model. Comput.*, 9(1):129–134, 2014b. doi: 10.3233/sat190105. URL https://doi.org/10.3233/sat190105.

Craig A. Morgenstern. Distributed coloration neighborhood search. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*, volume 26

of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 335–357. DIMACS/AMS, 1993. doi: 10.1090/dimacs/026/16. URL https://doi.org/10.1090/dimacs/026/16.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001. doi: 10.1145/378239.379017. URL https://doi.org/10.1145/378239.379017.

Sreerama K. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data Mining and Knowledge Discovery*, 2(4):345–389, 1998. doi: 10.1023/A:1009744630224. URL https://doi.org/10.1023/A:1009744630224.

Nina Narodytska, Alexey Ignatiev, Filipe Pereira, and Joao Marques-Silva. Learning optimal decision trees with SAT. In *Proceedings of IJCAI 2018*, pages 1362–1368. ijcai.org, 7 2018. doi: 10.24963/ijcai.2018/189. URL https://doi.org/10.24963/ijcai.2018/189.

Toru Ogawa, Yangyang Liu, Ryuzo Hasegawa, Miyuki Koshimura, and Hiroshi Fujita. Modulo based CNF encoding of cardinality constraints and its application to maxsat solvers. In *25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2013, Herndon, VA, USA, November 4-6, 2013*, pages 9–17. IEEE Computer Society, 2013. doi: 10.1109/ICTAI.2013.13. URL https://doi.org/10.1109/ICTAI.2013.13.

Randal S. Olson, William La Cava, Patryk Orzechowski, Ryan J. Urbanowicz, and Jason H. Moore. PMLB: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining*, 10(1):36, Dec 2017. ISSN 1756-0381. doi: 10.1186/s13040-017-0154-4. URL https://doi.org/10.1186/s13040-017-0154-4.

Sebastian Ordyniak and Stefan Szeider. Parameterized complexity of small decision tree learning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Virtual Event, February 2-9, 2021*, pages 6454–6462. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/AAAI/article/view/16800.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011. URL http://dl.acm.org/citation.cfm?id=2078195.

David Pisinger and Stefan Ropke. Large neighborhood search. In *Handbook of Metaheuristics*, pages 399–419. Springer Verlag, 2010.

David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986. doi: 10.1016/S0747-7171(86)80028-1. URL https://doi.org/10.1016/S0747-7171(86)80028-1.

Vili Podgorelec, Peter Kokol, Bruno Stiglic, and Ivan Rozman. Decision trees: An overview and their use in medicine. *Journal of Medical Systems*, 26(5):445–463, 2002. doi: 10.1023/A:1016409317640. URL http://doi.org/10.1023/A:1016409317640.

Rachel Pottinger and Alon Halevy. Minicon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10(2–3):182–198, September 2001. ISSN 1066-8888. doi: 10.1007/s007780100048. URL https://doi.org/10.1007/s007780100048.

J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986. doi: 10.1023/A:1022643204877. URL https://doi.org/10.1023/A:1022643204877.

J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993. ISBN 1-55860-238-0. doi: 10.1007/BF00993309. URL https://doi.org/10.1007/BF00993309.

Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. MaxSAT-based postprocessing for treedepth. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 478–495. Springer, 2020. doi: 10.1007/978-3-030-58475-7_28. URL https://doi.org/10.1007/978-3-030-58475-7_28.

Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. Turbocharging treewidth-bounded Bayesian network structure learning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Virtual Event, February 2-9, 2021*, pages 3895–3903. AAAI Press, 2021a. URL https://ojs.aaai.org/index.php/AAAI/article/view/16508.

Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. Learning fast-inference Bayesian networks. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 17852–17863, 2021b. URL https://proceedings.neurips.cc/paper/2021/hash/94e70705efae423efda1088614128d0b-Abstract.html.

Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. Learning large Bayesian networks with expert constraints. In James Cussens and Kun Zhang, editors, *Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence*, volume 180 of *Proceedings of Machine Learning Research*, pages 1592–1601. PMLR, 01–05 Aug 2022. URL https://proceedings.mlr.press/v180/peruvemba-ramaswamy22a.html.

Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. Circuit minimization with qbf-based exact synthesis. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023*. AAAI Press, 2023. to appear.

Ryan A. Rossi and Nesreen K. Ahmed. Coloring large complex networks. *Soc. Netw. Anal. Min.*, 4(1):228, 2014. doi: 10.1007/s13278-014-0228-y. URL https://doi.org/10.1007/s13278-014-0228-y.

Marko Samer and Helmut Veith. Encoding treewidth into SAT. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *LNCS*, pages 45–50. Springer Verlag, 2009. doi: 10.1007/978-3-642-02777-2_6. URL https://doi.org/10.1007/978-3-642-02777-2_6.

André Schidler. SAT-based local search for plane subgraph partitions (CG challenge). In Xavier Goaoc and Michael Kerber, editors, *38th International Symposium on Computational Geometry, SoCG 2022, June 7-10, 2022, Berlin, Germany*, volume 224 of *LIPIcs*, pages 74:1–74:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.SoCG.2022.74. URL https://doi.org/10.4230/LIPIcs.SoCG.2022.74.

André Schidler and Stefan Szeider. Computing optimal hypertree decompositions. In Guy E. Blelloch and Irene Finocchi, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*, pages 1–11. SIAM, 2020. doi: 10.1137/1.9781611976007.1. URL https://doi.org/10.1137/1.9781611976007.1.

André Schidler and Stefan Szeider. SAT-based decision tree learning for large data sets. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Virtual Event, February 2-9, 2021*, pages 3904–3912. AAAI Press, 2021a. URL https://ojs.aaai.org/index.php/AAAI/article/view/16509.

André Schidler and Stefan Szeider. Computing optimal hypertree decompositions with SAT. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, 2021b. doi: 10.24963/ijcai.2021/196. URL https://doi.org/10.24963/ijcai.2021/196.

André Schidler and Stefan Szeider. A SAT approach to twin-width. In Cynthia A. Phillips and Bettina Speckmann, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2022, Alexandria, VA, USA, January 9-10, 2022*, pages 67–77. SIAM, 2022. doi: 10.1137/1.9781611977042.6. URL https://doi.org/10.1137/1.9781611977042.6.

Christian Schulz, Ernestine Großmann, Tobias Heuer, and Darren Strash. PACE 2022, 2022. URL https://pacechallenge.org/2022/.

Roberto Sebastiani and Patrick Trentin. Optimathsat: A tool for optimization modulo theories. *J. Autom. Reason.*, 64(3):423–460, 2020. doi: 10.1007/s10817-018-09508-6. URL https://doi.org/10.1007/s10817-018-09508-6.

Jendrik Seipp and Malte Helmert. Counterexample-guided cartesian abstraction refinement for classical planning. *J. Artif. Intell. Res.*, 62:535–577, 2018. doi: 10.1613/jair.1.11217. URL https://doi.org/10.1613/jair.1.11217.

Pouya Shati, Eldan Cohen, and Sheila A. McIlraith. SAT-based approach for learning optimal decision trees with non-binary features. In *Proceedings of CP 2021*, volume 210 of *LIPIcs*, pages 50:1–50:16. Schloss Dagstuhl, 2021. doi: 10.4230/LIPIcs.CP.2021.50. URL https://doi.org/10.4230/LIPIcs.CP.2021.50.

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, 10th Edition*. Wiley, 2018. ISBN 978-1-118-06333-0. URL http://os-book.com/OS10/index.html.

João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999. doi: 10.1109/12.769433. URL https://doi.org/10.1109/12.769433.

Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Proceedings*, volume 3709 of *LNCS*, pages 827–831. Springer Verlag, 2005. doi: 10.1007/11564751_73. URL https://doi.org/10.1007/11564751_73.

Takehide Soh, Daniel Le Berre, Stéphanie Roussel, Mutsunori Banbara, and Naoyuki Tamura. Incremental SAT-based method with native Boolean cardinality handling for the Hamiltonian cycle problem. In *JELIA 2014*, volume 8761 of *Lecture Notes in Computer Science*, pages 684–693. Springer, 2014. URL https://doi.org/10.1007/978-3-319-11558-0_52.

Jack Spalding-Jamieson, Brandon Zhang, and Da Wei Zheng. Conflict-based local search for minimum partition into plane subgraphs (CG challenge). In Xavier Goaoc and Michael Kerber, editors, *38th International Symposium on Computational Geometry, SoCG 2022, June 7-10, 2022, Berlin, Germany*, volume 224 of *LIPIcs*, pages 72:1–72:6. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.SoCG.2022.72. URL https://doi.org/10.4230/LIPIcs.SoCG.2022.72.

Ulrike Stege and Michael Ralph Fellows. An improved fixed parameter tractable algorithm for vertex cover. *Technical report/Departement Informatik, ETH Zürich*, 318, 1999. URL https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/69332/eth-4359-01.pdf.

Bernardo Subercaseaux and Marijn J. H. Heule. The packing chromatic number of the infinite square grid is at least 14. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 21:1–21:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.SAT.2022.21. URL https://doi.org/10.4230/LIPIcs.SAT.2022.21.

Wen Sun. *Heuristic Algorithms for Graph Coloring Problems. (Algorithmes heuristiques pour des problèmes de coloration de graphes)*. PhD thesis, University of Angers, France, 2018. URL https://tel.archives-ouvertes.fr/tel-02136810.

Pang-Ning Tan, Michael S. Steinbach, Anuj Karpatne, and Vipin Kumar. *Introduction to Data Mining (Second Edition)*. Pearson, 2019. URL https://www-users.cse.umn.edu/%7Ekumar001/dmbook/index.php.

Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983. doi: 10.1007/978-3-642-81955-1_28. URL https://doi.org/10.1007/978-3-642-81955-1_28.

Moshe Y. Vardi. Boolean satisfiability: theory and engineering. *Commun. ACM*, 57(3):5, 2014. doi: 10.1145/2578043. URL https://doi.org/10.1145/2578043.

Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning optimal decision trees using constraint programming. *Constraints An Int. J.*, 25(3-4):226–250, 2020a. doi: 10.1007/s10601-020-09312-3. URL https://doi.org/10.1007/s10601-020-09312-3.

Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning optimal decision trees using constraint programming. *Constraints An Int. J.*, 25(3-4):226–250, 2020b. doi: 10.1007/s10601-020-09312-3. URL https://doi.org/10.1007/s10601-020-09312-3.

Anurag Verma, Austin Buchanan, and Sergiy Butenko. Solving the maximum clique and vertex coloring problems on very large sparse networks. *INFORMS J. Comput.*, 27(1):164–177, 2015. doi: 10.1287/ijoc.2014.0618. URL https://doi.org/10.1287/ijoc.2014.0618.

Sicco Verwer and Yingqian Zhang. Learning optimal classification trees using a binary linear program formulation. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 1625–1632. AAAI Press, 2019. doi: 10.1609/aaai.v33i01.33011624. URL https://doi.org/10.1609/aaai.v33i01.33011624.

Eric Weisstein. MathWorld online mathematics resource, 2021. retrieved Jan 17, 2021.

Mingyu Xiao and Hiroshi Nagamochi. Confining sets and avoiding bottleneck cases: A simple maximum independent set algorithm in degree-3 graphs. *Theor. Comput. Sci.*, 469:92–104, 2013. doi: 10.1016/j.tcs.2012.09.022. URL https://doi.org/10.1016/j.tcs.2012.09.022.

Jinqiang Yu, Alexey Ignatiev, Peter J. Stuckey, and Pierre Le Bodic. Learning optimal decision sets and lists with SAT. *J. Artif. Intell. Res.*, 72:1251–1279, 2021. doi: 10.1613/jair.1.12719. URL https://doi.org/10.1613/jair.1.12719.

Hai-Jun Zhou. A spin glass approach to the directed feedback vertex set problem. *Journal of Statistical Mechanics: Theory and Experiment*, 2016(7):073303, 2016. URL https://doi.org/10.1088/1742-5468/2016/07/073303.

Marinka Zitnik, Rok Sosič, Sagar Maheshwari, and Jure Leskovec. BioSNAP Datasets: Stanford biomedical network dataset collection. http://snap.stanford.edu/biodata, August 2018.