

Parallel Parameterized Algorithms

Max Bannach

From the Institute of Theoretical Computer Science
of the Universität zu Lübeck
Director: Prof. Dr. math. Rüdiger Reischuk

Parallel Parameterized Algorithms

Dissertation
for Fulfillment of
Requirements
for the Doctoral Degree
of the Universität zu Lübeck

from the Department of Computer Sciences

Submitted by

Max Bannach
from Halle (Saale)

Lübeck, 2019

First referee Prof. Dr. Till Tantau
Second referee Prof. Dr. Heribert Vollmer
Date of oral examination 18. 12. 2019
Approved for printing 06. 01. 2020

ABSTRACT

Fixed-parameter tractability is one of the key methodologies of modern algorithm design and complexity theory. As of today, the most studied resource in this field is sequential time. In contrast, in classical complexity theory there is a rich literature concerning parallel processing. Identifying suitable parameters as well as accelerating computations through parallelization both have the same goal: Increase the solvable fraction of an otherwise intractable problem. It is therefore natural to bring both fields together by studying parallel parameterized algorithms. In this thesis I present a rich framework of parallel parameterized complexity classes and develop a toolbox of basic parallel parameterized algorithms. It will be shown how the core techniques of parameterized complexity theory can be implemented in parallel – including color coding, bounded search trees, kernelization, structural decomposition of graphs, and algorithmic meta-theorems. Especially the latter two methods lead to deep insights into the complexity of well-known problems – but I also illustrate how they can be utilized in practice by presenting two corresponding software libraries: One for computing optimal tree compositions and one for model checking a fragment of monadic second-order logic.

ZUSAMMENFASSUNG

Die parametrisierte Algorithmik ist ein Schlüsselbereich des modernen Algorithmenentwurfes sowie der Komplexitätstheorie. Die fast ausschließlich untersuchte Ressource in diesem Bereich ist dabei die sequentielle Zeit, obwohl die Parallelverarbeitung ein zentrales und vielfach untersuchtes Teilgebiet der klassischen Algorithmik ist. Sowohl das Identifizieren eines geeigneten Parameters als auch die direkte Beschleunigung durch Parallelisierung verfolgen das gleiche Ziel: möglichst viele Instanzen eines an sich nicht effizient lösbaren Problems dennoch zu lösen. Es ist daher naheliegend, beide Forschungsgebiete miteinander zu verbinden – und genau diese Art von Integration ist das Ziel dieser Arbeit. Ich präsentiere eine Vielzahl von parametrisierten Komplexitätsklassen und entwickle eine Sammlung von parallelen parametrisierten Basisalgorithmen. Dabei werden nahezu alle Techniken, die die parametrisierte Komplextheorie zu bieten hat, von einem parallelen Standpunkt aus untersucht – unter anderem Color Coding, beschränkte Suchbäume, Kernelisierung, strukturelle Zerlegungen von Graphen sowie algorithmische Metatheoreme. Außerdem illustriere ich, wie die letzten zwei Techniken in der Praxis genutzt werden können, indem ich zwei Software-Bibliotheken vorstelle: eine zum Berechnen optimaler Baumzerlegungen und eine für die Modellprüfung eines Fragmentes der monadischen Prädikatenlogik zweiter Stufe.

CONTENTS

Abstract	vii
Zusammenfassung	ix

1 Introduction	1
1.1 Why Parallel Parameterized Algorithms	1
1.2 Results of This Dissertation	2
1.3 Related Work and History	7
1.4 Organization of This Thesis	9
1.5 Acknowledgement	10
2 Structures, Graphs, and Logic	11
2.1 Relational Structures	11
2.2 Graphs and Decompositions	13
2.3 First- and Second-Order Logic	16
3 Background in Complexity	21
3.1 Classic Complexity Theory	22
3.2 Parameterized Complexity Theory	26
3.3 Differentiation of Parameterized Complexity	30

I THEORY OF PARALLEL PARAMETERIZED ALGORITHMS

4 A Toolbox of Basic Parallel Parameterized Algorithms	35
4.1 Finding Maximal Independent Set in Graphs of Bounded Degree . .	36
4.2 Graph Traversal	39
4.3 Color Coding	44
5 Parallel Bounded Search Trees	49
5.1 A Short Review of Bounded Search Trees	50
5.2 Modulators and Editing	51
5.3 Feedback-Vertex Set	57

6	Parallel Kernelization	63
6.1	A Short Review of Kernelizations	64
6.2	Parallel Parameterized Algorithms Equal Parallel Preprocessing . .	67
6.3	Kernelizations for Vertex Cover and Matching	72
6.4	Parallel Kernelizations for Problems Parameterized by Vertex Cover	77
6.5	Computing Hitting Set Kernels in Parallel	81
7	Parallel Decomposition of Graphs	91
7.1	Crown Decompositions	92
7.2	Treedepth Decompositions	94
7.3	Tree Decompositions	95
8	Parallel Parameterized Algorithmic Meta-Theorems	101
8.1	First-Order Model Checking	102
8.2	Second-Order Model Checking	104
9	Outlook and Further Directions	111

II TOWARDS PRACTICE AND BACK

10	Jdrasil: A Modular Library for Computing Tree Decompositions	117
10.1	The Design Philosophy of Jdrasil	118
10.2	A High-Level View on the Library	119
10.3	A SAT-Based Exact-Solver	123
10.4	Exact Solving via Positive Instance Driven Dynamic Programming .	126
10.5	Parallelization Through Splitting	139
10.6	Experiments and Analysis	143
11	Jatatosk: A Lightweight Model Checker for a Fragment of MSO	155
11.1	The Aim of Jatatosk	156
11.2	A High-Level View on the Tool	156
11.3	Description of the Fragment	157
11.4	Extensions of the Fragment	159
11.5	Predicting the Run Time and Experiments	161
12	Outlook and Further Directions	171
13	Conclusion	173
	Compendium of Classes and Problems	177
	Experiment Setup	183
	Bibliography	183
	Curriculum Vitae	201

1 INTRODUCTION

Computer science is faced with a huge portfolio of interesting problems, most of which are considered intractable. It lies at the heart of *complexity theory* to study the computations involved in solving such problems in order to provide a fine-grained classification of problems into those that can be solved efficiently and those that cannot. Using such a classification as guideline, it is the task of *algorithm design* to develop algorithms that solve the problems as quickly as possible, and it is the burden of *algorithm engineering* to make these algorithms work in practice. In the following thesis, I try to develop a new subfield of complexity theory, try to demonstrate how it applies to the design of algorithms, and try to present its interaction with algorithm engineering: the field of *parallel parameterized algorithms*.

1.1 WHY PARALLEL PARAMETERIZED ALGORITHMS

What shall we do if we encounter an intractable problem? We could relax our requirements and use heuristics or approximation algorithms. However, this is not feasible whenever a non-optimal solution produces unacceptable costs. Many problems that are intractable from a complexity theoretic point of view can still be solved efficiently in practice via algorithm engineering. A well-known example is the satisfiability problem for propositional logic, for which modern tools can solve instances with millions of variables. How can this be, when the problem in its entirety is so difficult? The reason is that instances that are solved by the practitioners, and thus the instances that arise in “real world” applications, are very structured. This insight is taken back from practice to complexity theory by the field of *parameterized complexity theory*. The central idea is to develop *parameterized algorithms* that try to explicitly utilize such structures in order to be more efficient.

Parameterized complexity theory provides new research directions, but unfortunately loses a bit of the purity of classical complexity theory, as “the landscape of complexity classes becomes much more unwieldy. This means that the natural problems tend to fall into a large number of apparently different classes.” [85] To counteract this effect, Flum and Grohe [85] suggest to use logic, “which can serve as a tool to get a more systematic understanding of such classes.” Large parts of parameterized complexity theory can nowadays be stated completely in logical terms. This is yet another example for “the unusual effectiveness of logic in computer science.” [103]

This characterization of complexity theory in terms of logic, a field called *descriptive complexity*, has another major advantage: “Descriptive complexity is inherently parallel in nature.” [111] Studying parameterized complexity theory from a logician’s point of view is, therefore, nothing else than studying *parallel parameterized complexity theory*. A subject that is very natural to study on its own – just recall that we arrived in the parameterized setting because the problems that we are trying to solve are difficult, and observe that almost all computational devices available today have a parallel architecture that may allow for a parallel speedup.

Having an understanding of parallel parameterized complexity theory, the next step is to turn the gained knowledge into *parallel parameterized algorithms* that we can actually implement. It will be convenient to describe such algorithms in the language of Boolean circuits, as on one hand this is the most natural parallel computational model, and on the other hand such circuits are deeply linked to logic and descriptive complexity. Once we have specified the computational model and have designed parallel parameterized algorithms, we can turn back to algorithm engineering and implement the resulting procedures.

1.2 RESULTS OF THIS DISSERTATION

In this dissertation, I present a rigorous overview of parameterized parallel complexity theory and develop a rich toolbox of parameterized parallel algorithms. In both aspects, I will focus primarily on positive results, that is, we will study parameterized problems that can be solved efficiently in parallel. In contrast, many previous works have studied parameterized circuit complexity of intractable problems to obtain a fine-grained classification of them.

The results split, as this thesis, into two parts. First we concentrate on the design of parameterized algorithms. Besides a collection of basic parallel parameterized algorithms that can be used as subroutines in the design of further algorithms, we study almost all the standard techniques that parameterized complexity theory has in its quiver from a parallel point of view. It will come to light that the technique of color coding is a central concept to execute parameterized algorithms in parallel.

Color coding is a randomized technique used to identify small objects in a larger graph by assigning random colors to the vertices of that graph. The probability that the objects we are searching for get a certain coloring depends only on the size of the objects and the number of used colors and, therefore, color coding naturally leads to randomized parameterized algorithms. In fact, we can check many random colorings in parallel and, thus, color coding also naturally leads to randomized parallel parameterized algorithms. As first addition to our toolbox we will show that we can derandomize color coding in parallel constant time:

▷ Informal Version of Theorem 42.

Color coding can be derandomized in para-AC^0 ◁

Equipped with this powerful subroutine we will develop parameterized counterparts of many basic techniques from parallel processing. Most importantly, we will deal with symmetry breaking by solving multiple versions of the parameterized independent set problem in parallel. In the following table, k refers to the size of the sought independent set, while Δ is the maximum degree of the input graph. The results are proven in Theorem 33, Lemma 46, and Theorem 47.

Problem	Complexity
P_{Δ} -MAXIMAL-INDEPENDENT-SET	para-AC^{o+e}
$P_{k,\Delta}$ -INDEPENDENT-SET	para-AC^0
P_k -PLANAR-INDEPENDENT-SET	para-AC^1 (para-AC^{o1} if planarity is promised)

We will study parameterized reachability and distance problems – establishing a parallel parameterized version of the depth-first, the breadth-first search, and a link between alternating distance and parallel parameterized complexity theory:

▷ Informal Version of Theorem 39.

The parameterized *alternating* distance problem is complete for para-AC^{o1} ◁

Once we have established the toolbox, we will systematically adapt many standard strategies from fixed-parameter tractability theory to a parallel setting – starting with bounded search trees. Algorithms based on this paradigm can naturally be parallelized, as we can handle multiple branches of the search tree in parallel. We formalize this intuition by designing a parallel algorithm for a large family of modulator and editing problems, leading to multiple versions of the following result:

▷ Informal Version of Corollary 57 and Corollary 63.

Let \mathcal{H} be a family of graphs with constant treewidth. There is a family of para-FAC^{o1} -circuits that decides, given graphs $H \in \mathcal{H}$ and G , whether we can delete k vertices from G such that there is no homomorphism (embedding) from H to G . ◁

An interesting problem that will not fit into this framework is the feedback-vertex set problem, and we will craft a dedicated algorithm for it in order to show:

$$P_k\text{-FEEDBACK-VERTEX-SET} \in \text{para-AC}^{1!}$$

To achieve the algorithm we will be forced to develop parallel versions of many well-known preprocessing rules for the feedback-vertex set problem. As it will turn out, this is only possible if we interleave the application of these rules with the parallel search tree – applying them individually is P-complete! Since almost all kernelizations for the feedback-vertex set problem that are discussed in the literature are

build on top of these preprocessing rules, a natural next question is whether there is any hope for a parallel kernelization. Computing kernels in parallel seems, at the first sight, like a tough task: All textbooks present kernelizations as a list of reduction rules that are applied *sequentially* as long as possible. However, we will show that parallel parameterized algorithms are deeply linked to the parallel computation of kernels – similar to the fact that a problem is fixed-parameter tractable if, and only if, it admits a polynomial time computable kernelization (and is decidable):

▷ Informal Version of Theorem 77.

Parallel parameterized algorithms are equivalent to parallel preprocessing, that is, a problem lies in para-AC^i if, and only if, a kernel of it can be computed in FAC^i . ◁

Given the knowledge that many natural problems *have* a parallel computable kernelization, we will start a journey on which we establish a number of upper and lower bounds. For instance, we show that we can compute an exponential kernel for p_k -VERTEX-COVER in FAC^0 , a quadratic kernel in FTC^0 , and we show that the currently best sequential kernel for the problem *cannot* be computed in parallel unless we can compute large matchings in parallel – and whether this is possible is a long-standing open problem in the field.

The technical most challenging – and in my opinion also the most interesting – parallel kernelization that I will present is for the hitting set problem parameterized by the solution size k and the maximum size d of the hyperedges. It was conjectured by Chen, Flum, and Huang that a parallel kernelization for this problem will require time $\Omega(d)$ [53], however, with the *massive* use of color coding we can achieve the kernel in constant parallel time:

▷ Informal Version of Corollary 111.

A kernel for $p_{k,d}$ -HITTING-SET can be computed in FAC^0 . ◁

The technique that we will use to obtain this kernelization (the “massive” use of color coding) is interesting on its own, as it shows that iterated applications of the color coding technique can sometimes be collapsed into a single application. I believe that this trick could be useful for many other parallel parameterized algorithms.

The next and final technique that we will take from fixed-parameter tractability theory and apply it in parallel is the use of algorithmic meta-theorems. To that end, we will develop parallel parameterized algorithms to compute various graph decompositions, including algorithms to compute tree decompositions. Based on these decompositions, I will provide parallel versions of many famous algorithmic meta-theorems. These meta-theorems boil down to efficient algorithms for the model checking problem for various logics. In the following table, each row refers to the model checking problem of a certain logic parameterized by both, the size of the input formula and some structural parameter of the Gaifman graph of the input structure (from top to bottom: its maximum degree, its vertex cover number, its treedepth, and its treewidth).

Logic	Parameter	Complexity	Reference
First-Order	$ \varphi + \Delta$	para-AC ⁰¹	Theorem 130
Monadic Second-Order	$ \varphi + \text{vc}$	para-AC ⁰	Theorem 132
Monadic Second-Order	$ \varphi + \text{td}$	para-AC ⁰¹	Theorem 133
Monadic Second-Order	$ \varphi + \text{tw}$	para-AC ²¹	Theorem 134

In the second part of this thesis we will take the theoretical results from the first part and combine them with algorithmic engineering in order to obtain a tool, which is fast in practice, for the problem in the last row. For this, we will first need a tool that can quickly compute tree decompositions in practice. I will present the Java library *Jdrasil*, which was developed by Sebastian Berndt, Thorsten Ehlers, and myself in the light of the first *Parameterized Algorithms and Computational Experiments Challenge* (PACE 2016). I will, however, only present the parts of the library that were primarily developed by myself – including the architecture, the exact algorithms, as well as the parallel capabilities of the library. In detail, we will first have a look at an improved SAT-encoding for treewidth that is based on an encoding by Berg and Järvisalo [26]. Then we will consider a game theoretic version of a novel *positive instance driven dynamic program* due to Hisao Tamaki [156–158]. This is the currently fastest paradigm for computing optimal tree decompositions in practice, and I will describe in detail how it works. We will compare this algorithm with the SAT-based algorithm and with multiple algorithms that were considered state-of-the-art before the first PACE. Finally, I will describe and experimentally analyze how *Jdrasil* parallelizes the computation of tree decompositions in general (independently of the used algorithm). This is done by computing safe separators, a concept introduced by Bodlaender and Koster [41], with a collection of novel heuristics.

Jdrasil is equipped with an interface that makes it easy to specify and execute dynamic programs over the computed tree decomposition. *Jatatosk* is a model checker for a fragment of monadic second-order logic that is based on this interface. The tool approaches the last result from the previous table from an algorithm engineering point of view by choosing a fragment with an efficient implemen-

tation in mind. The result is a tool that is faster than similar tools on many instances – which will be illustrated with various experiments. Additionally, the architecture of *Jatatosk* will allow us to determine its worst-case run time just from the syntax of the input formula. The table at the right shows the worst-case behavior of *Jatatosk* for some natural formulas for standard problems.

Formula	Run Time
$\varphi_{3\text{col}}$	$O^*(3^k)$
$\varphi_{\text{vc}}(S)$	$O^*(2^k)$
$\varphi_{\text{ds}}(S)$	$O^*(8^k)$
$\varphi_{\text{triangle-minor}}$	$O^*(k^{6k})$
$\varphi_{\text{fvs}}(S)$	$O^*(2^k k^{2k})$

Taking all these results together, I hope that I can convince you with this thesis that studying fixed-parameter tractability in parallel is interesting and fruitful – both, in theory and practice. Preliminary versions of many of the results that I will present within this thesis were previously presented at the following conferences (in chronological order):

- [19] Max Bannach, Christoph Stockhusen, and Till Tantau: *Fast Parallel Fixed-Parameter Algorithms via Color Coding*. In Proceedings of the 10th International Symposium on Parameterized and Exact Computation (IPEC 2015).
- [20] Max Bannach and Till Tantau: *Parallel Multivariate Meta-Theorems*. In Proceedings of the 11th International Symposium on Parameterized and Exact Computation (IPEC 2016).
- [16] Max Bannach, Sebastian Berndt, and Thorsten Ehlers: *Jdrasil: A Modular Library for Computing Tree Decompositions*. In Proceedings of the 16th International Symposium on Experimental Algorithms (SEA 2017).
- [21] Max Bannach and Till Tantau: *Computing Hitting Set Kernels By AC^0 -Circuits*. In Proceedings of the 35th Symposium on Theoretical Aspects of Computer Science (STACS 2018).
- [22] Max Bannach and Till Tantau: *Computing Kernels in Parallel: Lower and Upper Bounds*. In Proceedings of the 13th International Symposium on Parameterized and Exact Computation (IPEC 2018).
- [13] Max Bannach and Sebastian Berndt: *Practical Access to Dynamic Programming on Tree Decompositions*. In Proceedings of the 26th Annual European Symposium on Algorithms (ESA 2018).
- [14] Max Bannach and Sebastian Berndt: *Positive-Instance Driven Dynamic Programming for Graph Searching*. In Proceedings of 16th Algorithms and Data Structures Symposium (WADS 2019).

The second last paper was awarded *Best Student Paper* at the European Symposium on Algorithms 2018. A complete list of my publications can be found in the Curriculum Vitae on page 201. In particular [17], [18], and [23] are built on top of results of this thesis and provide interesting further directions.

1.3 RELATED WORK AND HISTORY

Parameterized complexity theory is a very active field of research. It was first introduced in a series of papers by Downey and Fellows [1, 66–68]. This interesting field is growing so rapidly that alone in the last decade seven (!) textbooks have been published to cover all its different aspects. Downey and Fellows have published two introductory books [69, 70]. Flum and Grohe focus on the complexity theoretic point of view and describe logical characterizations of parameterized classes [85]. A higher focus on algorithmic techniques can be found in the books by Niedermeier [137] and Cygan, Fomin, Kowalik, Lokshantov, Marx, Pilipczuk, Pilipczuk, and Saurabh [59]. A collection of surveys that cover many areas on which the “multivariate algorithmic revolution” has an impact is presented in [34]. Fomin, Lokshantov, Saurabh, and Zehavi have dedicated a book to parameterized preprocessing and kernelization [90]. Although the field is not old, it contains many ideas that were studied long before. For instance, preprocessing was always a heuristic tool frequently used by practitioners, it just did not have a clean mathematical characterization.

The idea of parallelization is almost as old as the dream of automatic computation. The first note is attributed to Luigi Federico Menabrea and his “Sketch of the Analytical Engine Invented by Charles Babbage” back in 1842. Today, almost every computational device contains multiple cores, and highly parallel systems with hundreds of processors are available. With cheap parallel hardware such as field-programmable gate arrays (FPGAs), and massive improvements in parallel computations with general purpose graphical units (GPGPUs), the theory of parallel algorithms is now more important than ever. The theory behind parallel processing is deeply linked to the circuit complexity class NC (for “Nick’s Class”) that was introduced by Nicholas Pippenger [141]. An overview over many classical results in this field was assembled by Cook [55]. Alternatively, one can study parallel algorithms with parallel random access machines – a detailed comparison can be found in [155]. An introduction to many basic parallel algorithms can be found in the textbook of JáJá [114]. Particularly important are results concerning symmetry breaking, such as the parallel algorithm for computing maximal independent sets in general graphs due to Michael Luby [130], or the algorithm due to Goldberg, Serge, Plotkin, and Shannon that works faster on graphs of bounded degree [100]. One of the independent set algorithms that we will study is directly based on the later. That parallel algorithms for symmetry breaking can be accelerated via randomization was observed by Alon, Babai, and Itai [5]. Alon, Yuster, and Zwick later studied randomization in the form of color coding [6], a technique that we will apply in the parameterized parallel setting as well.

Combining both, parameterized complexity theory and parallel computation, was first done by Cai, Chen, Downey, and Fellows with an investigation of parameterized logspace [46]. Later, Flum and Grohe defined the “parameterized counterpart” for any classical complexity class and, thus, a parameterized analogue of the NC-hierarchy [84]. The machine model used within this thesis was derived from this general formulation and was – in this form – first described in [19]. The first to study parallel parameterized algorithms were Cesati and Ianni [47]. Elberfeld, Stockhusen, and Tantau provide a rich framework of parameterized space and circuit classes; and they identify many natural problems for these classes [76]. They focused to a large degree on classes with bounded nondeterminism, and studied in this context also p_k -FEEDBACK-VERTEX-SET with an algorithm similar to the one I will present.

Concerning parallel kernelization, Cai, Chen, Downey, and Fellows did implement the well-known Buss kernelization for p_k -VERTEX-COVER in logspace [46], which was later improved by Elberfeld, Stockhusen, and Tantau to an FTC° -kernelization [76]. For the later result the authors already sketched the idea that a decidable problem has a kernel computable in FTC° if, and only if, it can be solved in para-TC° . Chen, Flum, and Huang studied the parallel complexity of p_k -HITTING-SET for hypergraphs with hyperedges of constant size and provided a parallel kernel that requires time $\Omega(d)$ while producing polynomial work [53]. A result, on which the constant time kernelization for $p_{k,d}$ -HITTING-SET that is presented within this thesis is build.

Parameterized circuit complexity was also used to introduce parameterized lower bounds: Chen and Flum introduced a para-AC° -version of the famous Clique Switching Lemma, which shows that any fpt -approximation of p_k -CLIQUE is unconditionally not in para-AC° [52]. Besides parameterized circuit complexity, there is also a growing body of literature that considers parameterized parallel random access machines and practical implementations of the resulting algorithms [3, 49].

Logic is far older than computer science. The idea of moving from infinite-model theory to finite-model theory in order to link logic to complexity theory has its roots in Fagin’s famous theorem – the class NP is precisely captured by all properties expressible in existential second-order logic [78]. Descriptive complexity is the originating field that did get its momentum by the subsequent work of Immerman [111]. The connection to parameterized complexity is highlighted by Flum and Grohe [85].

Algorithmic meta-theorems find application in many areas of complexity theory and modern algorithm design. An overview can be found in the survey paper by Stephan Kreutzer [123]. The meta-theorem presented in this thesis that works on graphs of bounded degree is based on a result of Flum and Grohe, who have proven the theorem for parameterized logspace [84]. The other meta-theorems that I will present are variations of Courcelle’s famous theorem that states that every property expressible in monadic second-order logic can be tested in linear time on graphs of bounded treewidth [57]. This algorithm was also studied intensively from a parallel point of view for the case that the treewidth is bounded by a constant – in contrast, we will

study such algorithms while considering the treewidth as parameter. Bodlaender partly parallelized the result by providing individual NC-algorithms for many problems on graphs of constant treewidth [32]; subsequently Elberfeld, Jakoby, and Tantau implemented Courcelle’s theorem completely in logspace (and, thus, in AC^1) [74], as well as in NC^1 when the tree decomposition is part of the input [75]. From a practical point of view, the most promising efforts to implement Courcelle’s theorem are based on a game theoretic characterization [117, 118, 126], or by implementing it in a declarative framework for dynamic programming on tree decompositions [31].

1.4 ORGANIZATION OF THIS THESIS

Following this introduction, there are two chapters on preliminaries. In Chapter 2 I introduce the primary objects that we will study: relational structures and graphs. The following chapter provides the necessary background in circuit and complexity theory. Here, we define the complexity classes that we use throughout this thesis.

After the preliminaries, the thesis is partitioned into two parts. Part I is the primary part and deals with the design of parallel parameterized algorithms. The first chapter there, Chapter 4, introduces a toolbox of basic parallel parameterized algorithms and should be read before the others. The remaining chapters are largely independent and can be read in any order. They deal with parallelization of bounded search trees (Chapter 5), parallel preprocessing in the form of parallel kernelization (Chapter 6), the parallel decomposition of structures (Chapter 7), and the implementation of meta-theorems on top of such decompositions (Chapter 8).

The second part combines the theory from Part I with algorithm engineering in order to develop algorithms that are fast in practice. It contains two chapters corresponding to two software libraries: Chapter 10 introduces the library *Jdrasil* for computing tree decompositions; Chapter 11 introduces *Jatatosk*, a lightweight model checker for a fragment of monadic second-order logic. *Jatatosk* performs dynamic programming over tree decompositions and uses *Jdrasil* internally to find a suitable tree decomposition. However, besides this dependency the two chapters are independent and can be read in any order.

Each part ends in a chapter discussing the results, open problems, and further research directions. After Part II, both parts are set in relation and all results of this thesis are summarized in the conclusion in Chapter 13. At the end of this thesis on page 177, you will find a complete compendium of all complexity classes and problems that we discuss in this thesis. A description of the hardware and the test sets used in the various experiments can be found on page 183.

1.5 ACKNOWLEDGEMENT

I like to thank all the wonderful people that supported me during my time as a doctoral student. Without them, this thesis would not have been possible!

First of all, I like to thank my advisor Till Tantau who made an awesome job of always pointing me right to the next gripping puzzle. He was not just a good advisor, but a great teacher and an inspiration alike. Thank you!

Secondly, I would like to thank Rüdiger Reischuk who has placed his trust in me and has given me the opportunity to prepare this dissertation at his institute. He had always the right advice for a young researcher who is trying to make his first steps in the academic world. Thank you, too!

Next I would like to thank my colleagues and coauthors for countless fruitful discussions. In alphabetical order they are: Katharina Dannenberg, Thorsten Ehlers, Tom Hartmann, Tim Kunold, Alexandra Lassota, Maciej Liśkiewicz, Matthias Lutter, Martin Middendorf, Martin Schuster, Malte Skambath, Florian Thaeter, and Oliver Witt. A special thanks goes to Christoph Stockhusen who has aroused my interest in theoretical computer science; and Sebastian Berndt who has introduced me to the PACE challenge and has explored all the treewidth topics with me.

Finally, I would like to thank my friends and family for all their love and support. In particular, I would like to thank my former fellow students Tobias Mende, Albert Piek, and Folke Will for many various conversations about (not necessarily theoretical) computer science. My friend Pascal Geerdsen for cheering me up whenever it was necessary. My parents Mark and Anke for paving the way to study computer science after all. And, of course, my beloved wife Jacqueline, who has shielded me from all the difficulties of the “real world,” allowing me to fully focus on this dissertation.

2 STRUCTURES, GRAPHS, AND LOGIC

Before we start with the design of parallel algorithms, we shall define the objects that the algorithms will handle. In this thesis, these objects are mainly *graphs* and the more general *relational structures*. In this chapter we gather the essential definitions of relational structures and I provide a brief introduction to graph theory. Furthermore, I introduce the language that we will use to describe properties of structures and graphs: *first- and second-order logic*. For a more comprehensive introduction to structures and logic I refer the reader to the textbook of Immerman [111]. The notation concerning graph theory follows the standard textbook of Diestel [65].

2.1 RELATIONAL STRUCTURES

A *relational structure* is a set together with a collection of relations defined on it. In order to grasp these relations and to talk about the structure, we use a *vocabulary* that defines the relations that are available. A structure gives meaning to such a vocabulary by *interpreting* the defined relations.

► Definition 1 (Vocabulary)

A (relational) *vocabulary* is a finite set $\tau = \{R_1^{\alpha_1}, R_2^{\alpha_2}, \dots, R_k^{\alpha_k}\}$ consisting of *relational symbols* R_i of arity $\text{ar}(R_i) = \alpha_i$. ◁

► Definition 2 (Structure)

A (finite, relational) *structure* over some vocabulary $\tau = \{R_1^{\alpha_1}, R_2^{\alpha_2}, \dots, R_k^{\alpha_k}\}$, also called a τ -*structure*, is a tuple $S = (V(S), R_1^S, R_2^S, \dots, R_k^S)$ consisting of a non-empty finite set $V(S)$, called the *universe*, and an *interpretation* $R_i^S \subseteq V(S)^{\alpha_i}$ of every relational symbol R_i . ◁

For a fixed vocabulary τ , we denote with $\text{STRUC}[\tau]$ the set of all τ -structures. If it is clear from the context, we will refer to $V(S)$ by V and we drop the superscript in the relations of a structure, that is, we will identify relations with their relational symbols. We call the elements of the universe *vertices* and refer to unary relations as *colors*. Furthermore, we call tuples contained in binary relations (*directed*) *edges* and tuples contained in relations of higher arity (*directed*) *hyperedges*. In both cases, we drop the term “directed” if the interpretation of the relation is symmetric. Accordingly, we call the relation itself *edge-relation* or *hyperedge-relation*. This terminology is motivated from graph theory, as our primary objects will be graphs. Precise definitions for graphs and graph theory are provided in Section 2.2.

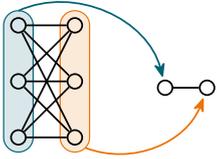
► Example 3 (Strings)

Let Σ be a fixed alphabet. We model strings $w \in \Sigma^*$ as structures over the vocabulary $\tau_{\Sigma\text{-string}}$ that contains the binary relation \leq^2 and a unary relation P_σ for every symbol $\sigma \in \Sigma$. A relational structure S representing w contains the universe $V = \{1, \dots, |w|\}$ and interprets \leq^S as the natural order on V . The relations P_σ^S indicate which symbol is at a given index. For instance, let $\Sigma = \{0, 1\}$ and let $w = 0110$, a corresponding structure would be $S = (\{1, 2, 3, 4\}, \leq^S, P_0^S, P_1^S)$ with $P_0^S = \{1, 4\}$ and $P_1^S = \{2, 3\}$. The set of all strings over Σ^* is naturally represented by $\text{STRUC}[\tau_{\Sigma\text{-string}}]$. ◁

We will often deal with general structures that serve as hosts for many other structures. Let $S = (V, R_1^S, R_2^S, \dots, R_k^S)$ be a structure and $X \subseteq V$. We denote with $S \setminus X = (V \setminus X, R_1^S \cap (V \setminus X)^{\text{ar}(R_1)}, \dots, R_k^S \cap (V \setminus X)^{\text{ar}(R_k)})$ the structure obtained by *deleting* the elements of X , and with $S[X] = S \setminus (V \setminus X)$ the substructure of S *induced* by X . With other words, a structure B is an induced substructure of a structure A if there is a sequence of element deletions that transforms A into B . If this sequence additionally contains the deletion of single tuples we say B is a *substructure* of A , or that A *contains* B . Finally, if the sequence also contains the *contraction* of tuples we say B is a *minor* of A . The contraction of a tuple (x_1, \dots, x_r) deletes the tuple and replaces every occurrence of x_1, \dots, x_r with a single new element x .

Let A and B be two τ -structures, we call a function $\varphi: V(A) \rightarrow V(B)$ a *homomorphism* if for all relational symbols R in τ and all $(x_1, \dots, x_{\text{ar}(R)}) \in V(A)^{\text{ar}(R)}$ we have:

$$(x_1, \dots, x_{\text{ar}(R)}) \in R^A \implies (\varphi(x_1), \dots, \varphi(x_{\text{ar}(R)})) \in R^B$$



We write $A \rightarrow B$ if there is at least one homomorphism from A to B . For an example, observe that there is a homomorphism from every bipartite graph to graphs that contain at least one edge, see the figure at the margin. An *injective* homomorphism is called an *embedding* from A into B . We denote the fact that there is any embedding from A to B with $A \rightarrow\rightarrow B$. We call φ a *strong* homomorphism (embedding) if it satisfies the following stronger condition:

$$(x_1, \dots, x_{\text{ar}(R)}) \in R^A \iff (\varphi(x_1), \dots, \varphi(x_{\text{ar}(R)})) \in R^B$$

Finally, an *isomorphism* is a bijective strong embedding, and two structures A and B are called *isomorphic*, denoted by $A \simeq B$, if there is an isomorphism from A to B .

With relational structures we have a notation to describe the objects we are interested in. Now we need a way to present a structure to a computational model, that is, we need a suitable encoding for them. This is often a matter of taste, as many encodings can be translated into each other quite easily. We present a standard encoding that we will always use, unless explicitly stated otherwise. Note that in the moment in which we define an encoding of a structure, we indirectly define an order of the elements of the universe, even if the structure itself is unordered. We call this order the *lexicographical order* and some algorithms presented within this thesis will explicitly use it.

► Definition 4 (Encoding of a Structure)

Let $S = (V, R_1^S, R_2^S, \dots, R_k^S)$ be a structure and let $\text{lex}: V \rightarrow \{0, 1, \dots, |V| - 1\}$ be an arbitrary but fixed bijection. The *encoding* of S (with respect to lex) is the binary string $\text{code}(S) \subseteq \{0, 1\}^*$ that contains a binary vector of length $|V|$ for every unary relation of S , a $|V| \times |V|$ adjacency matrix for every binary relation, and a $|V| \times |R_i^S|$ incidence matrix for every relation R_i^S of higher arity. In each vector and each matrix, the elements are sorted by lex . ◁

2.2 GRAPHS AND DECOMPOSITIONS

As mentioned in the previous section, we are primarily interested in graphs, which are simple relational structures with a single binary relation. Graphs inherit the concepts of *induced subgraph*, *subgraph*, and *minor* directly from relational structures.

► Definition 5 (Digraphs and Graphs)

A *digraph* is a relational structure over the vocabulary $\tau_{\text{graph}} = \{E^2\}$, a (undirected) *graph* is a digraph with a symmetric interpretation of E . We say a digraph or graph is *simple* if its interpretation of E is irreflexive. ◁

We will, in slight abuse of notation, sometimes denote the edges of a graph as sets $e = \{x, y\} \in E$, meaning an object e that represents both tuples (x, y) , (y, x) in the symmetric relation. For a graph $G = (V, E)$ we denote with $|V| = n$ the number of vertices and with $|E|/2 = m$ the number of *undirected* edges. For a vertex $v \in V$ we let $N(v) = \{w \mid \{v, w\} \in E\}$ be the *neighborhood* of v and define the *closed neighborhood* as $N[v] = N(v) \cup \{v\}$. For a vertex set $C \subseteq V$ we abbreviate $N(C) = (\bigcup_{v \in C} N(v)) \setminus C$. The *degree* of v is defined as $\delta(v) = |N(v)|$, and the *maximum degree* of G is $\Delta(G) = \max_{v \in V} \delta(v)$. We say two vertices v and w are *connected* if there is a sequence $(v = p_1, p_2, \dots, p_q = w)$ of vertices with $\{p_i, p_{i+1}\} \in E$ for all $1 \leq i < q$. A set $C \subseteq V$ is *connected* if all pairs of vertices in C are connected. A (*connected*) *component* of G is a subset of V that is inclusion-wise maximal with respect to this property. A set $S \subseteq V$ is called *separator* if $G[V \setminus S]$ contains more components than G . The components C_1, \dots, C_k of $G[V \setminus S]$ are *associated* with S , and we say C_i is a *full component* associated with S if $N(C_i) = S$. A *minimal separator* is a separator with at least two full components associated with it, and an *inclusion minimal separator* is a separator for which all associated components are full.

The *Gaifman graph* of a structure is a graph that represents its relations. Structures inherit graph theoretic terminology, such as separators, via the Gaifman graph.

► Definition 6 (Gaifman Graph)

The *Gaifman graph* of a structure S , denoted by $\text{Gaif}(S)$, is the (undirected) graph $G = (V, E)$ in which V is the universe of S , and in which E contains an edge $\{v, w\}$ if, and only if, $v \neq w$ and there is a relation R^S in S that contains a tuple $(x_1, \dots, x_{\text{ar}(R)})$ such that $v, w \in \{x_1, \dots, x_{\text{ar}(R)}\}$. ◁

► Example 7

Consider the vocabulary $\tau = \{R^2, G^1, B^3\}$ and a corresponding relational τ -structure $S = (\{1, 2, \dots, 10\}, R^H, G^H, B^H)$ with:

$$R^H = \{ (1, 2), (2, 3), (3, 4), (4, 7), (7, 8), (8, 7) \},$$

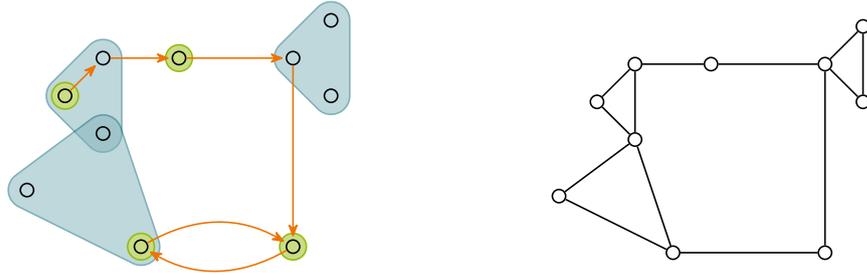
$$G^H = \{ 1, 3, 7, 8 \},$$

$$B^H = \{ (1, 2, 10), (1, 10, 2), (2, 1, 10), (2, 10, 1), (10, 2, 1), (10, 1, 2) \\ (8, 9, 10), (8, 10, 9), (9, 8, 10), (9, 10, 8), (10, 8, 9), (10, 9, 8) \\ (4, 5, 6), (4, 6, 5), (5, 4, 6), (5, 6, 4), (6, 4, 5), (6, 5, 4) \}.$$

The Gaifman graph of S is the graph $\text{Gaif}(S) = (\{1, \dots, 10\}, E)$ with the following edge relation:

$$E = \{ \{1, 2\}, \{1, 10\}, \{2, 3\}, \{2, 10\}, \{3, 4\}, \{4, 5\}, \{4, 6\}, \{4, 7\}, \\ \{5, 6\}, \{7, 8\}, \{8, 9\}, \{8, 10\}, \{9, 10\} \}.$$

The following figure illustrates the structure S on the left, and the corresponding Gaifman graph $\text{Gaif}(S)$ on the right.



◁

A fundamental technique of modern algorithm design, especially in the field of parameterized algorithms, is dynamic programming on certain structural decompositions of graphs. As we will apply similar techniques in this thesis, we need these graph theoretic terminology as well.

► Definition 8 (Tree Decomposition)

A *tree decomposition* of a graph $G = (V, E)$ is a pair (T, ι) in which T is a rooted tree and ι a mapping from the nodes of T to subsets of V (which we call *bags*) such that:

1. for every $u \in V$ the set $\{x \mid u \in \iota(x)\}$ is non-empty and connected in T ;
2. for every $\{v, w\} \in E$ there is a node y in T with $\{v, w\} \subseteq \iota(y)$.

The *width* of a tree decomposition is the maximum size of one of its bags minus one; its *depth* is the maximum of its width and the length of the longest root-leaf-path.

◁

► Definition 9 (Treewidth, Pathwidth, and Treedepth)

Let $G = (V, E)$ be a graph. The *treewidth* $\text{tw}(G)$ of G is the minimum possible width of a tree decomposition of G . The *pathwidth* $\text{pw}(G)$ of G is the minimum width over all tree decompositions (T, ι) for G in which T is a path. Finally, the *treedepth*, denoted by $\text{td}(G)$, of G is the minimum depth over all tree decompositions (T, ι) for G in which for all nodes x, y of T we have $\iota(x) \subsetneq \iota(y)$ if y is a descendant of x . ◁

Intuitively, the treewidth of a graph measures how similar the graph is “to being a tree,” therefore the name. For instance, a tree has treewidth 1 ($\text{tw}(\text{tree}) = 1$); but cliques have treewidth $n - 1$ ($\text{tw}(K_n) = n - 1$). Other graphs that are very unlike a tree are for instance $n \times n$ grids, which contain many cycles and have a treewidth of n , that is, $\text{tw}(\text{grid}) = n$. Likewise, the pathwidth measures how similar a graph is “to being a path,” for instance we have $\text{pw}(\text{path}) = 1$, but already for simple trees the pathwidth increases ($\text{pw}(\text{tree}) = 2$). Finally, the treedepth measures the similarity of a graph to “being a star,” for instance $\text{td}(\text{star}) = 2$, but even simple paths are very unlike a star and have a treedepth of $\log n$, that is, $\text{td}(\text{path}) = \log n$. It is well known that we have $\text{tw}(G) \leq \text{pw}(G) \leq \text{td}(G) \leq \text{tw}(G) \cdot \log_2 n$ for every undirected graph $G = (V, E)$ on n vertices [135].

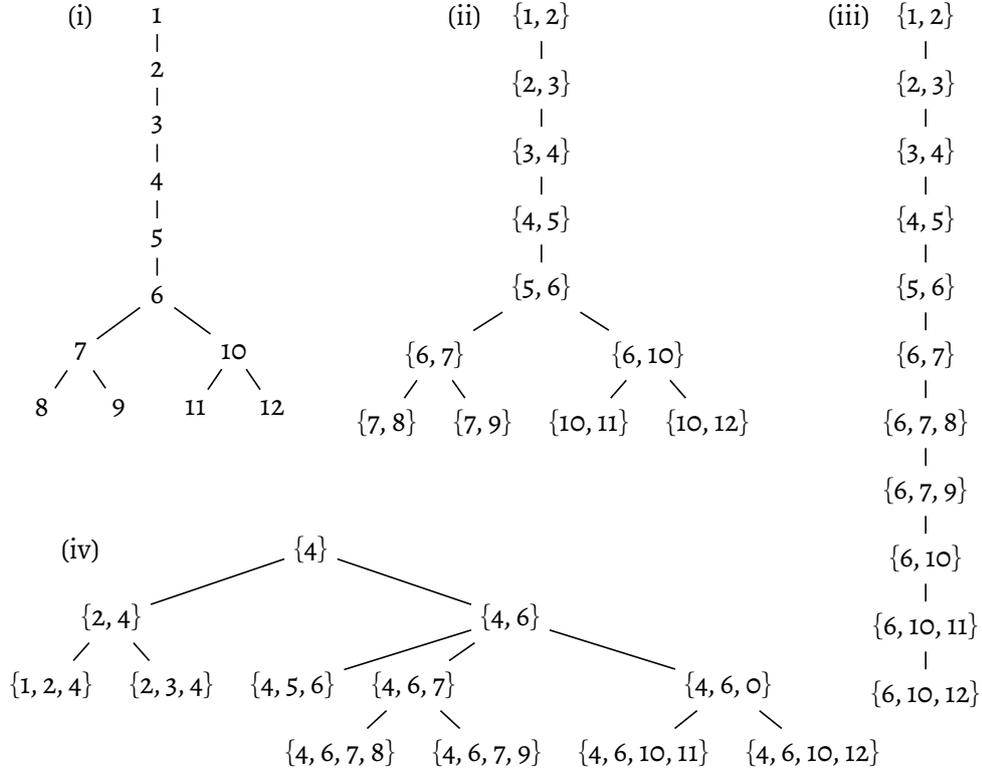
For many algorithms it is useful to have a certain form of a tree decomposition: A *nice* tree decomposition is a tuple (T, ι, η) such that (T, ι) is a tree decomposition, and $\eta: V(T) \rightarrow \{\text{leaf}, \text{introduce}, \text{join}, \text{forget}\}$ is a labeling function of the nodes. The nodes that are labeled as *leaf* are exactly the leaves of T , and the bags of these nodes are empty. Furthermore, the bag associated with the root of T is empty as well. *Introduce-* and *forget-*nodes n have one child x such that there is one $v \in V$ with $v \notin \iota(x)$ and $\iota(n) = \iota(x) \cup \{v\}$, or $v \in \iota(x)$ and $\iota(n) = \iota(x) \setminus \{v\}$, respectively. *Join-*nodes n have two children x and y with $\iota(n) = \iota(x) = \iota(y)$.

If a nice tree decomposition does not provide enough structure, we may also work on *very nice* tree decompositions, which are nice tree decompositions that additionally have exactly one *edge-bag* for every edge $e \in E$. These bags “virtually” introduce the corresponding edge. In particular, we assume that introduce-bags present “isolated” vertices that are later connected to other vertices by edge-bags.

We say a tree decomposition (T, ι) is *balanced* if T is a balanced tree, that is, if for every node n of T the heights of the subtrees rooted at the children of n differ by at most one. A nice tree decomposition (T, ι, η) is *balanced* if the tree obtained from T by contracting introduce and forget nodes is balanced. It is well known that every tree decomposition can be transformed into a (very) nice tree decomposition without increasing the width [59]. Furthermore, any tree decomposition of width w can be transformed into a balanced one of width at most $4w + 3$ [75].

► Example 10

Various tree decompositions of an undirected graph $G = (V, E)$ shown at (i). The decompositions justify (ii) $\text{tw}(G) \leq 1$, (iii) $\text{pw}(G) \leq 2$, and (iv) $\text{td}(G) \leq 4$.



◁

2.3 FIRST- AND SECOND-ORDER LOGIC

With relational structures we have a formal tool to describe all kinds of objects, but we cannot talk about these objects yet. We require a formal way to describe and evaluate properties of a structure. Take the graphs from the previous section as example: Both, digraphs and graphs, are defined over the vocabulary $\{E^2\}$, but the later are a subset of the former. To describe properties such as “is undirected” precisely, we will use mathematical logic.

► Definition 11 (Syntax of First-Order Logic)

Let $\tau = \{R_1^{\alpha_1}, \dots, R_k^{\alpha_k}\}$ be a vocabulary. Strings of the form x_0, x_1, x_2, x_3 , and so forth are called *variables*. The *first-order language* $\mathcal{L}(\tau)$ is inductively defined: It contains the *atomic formulas*, which are the strings $x_i = x_j$ for $i, j \in \mathbb{N}$ and, for a relational symbol R_ℓ of τ and variables $x_1, \dots, x_{\alpha_\ell}$, the string $R_\ell(x_1, \dots, x_{\alpha_\ell})$. Inductively, the language $\mathcal{L}(\tau)$ contains for all strings $\alpha, \beta \in \mathcal{L}(\tau)$ and all $i \in \mathbb{N}$ the strings $\neg(\alpha)$, $(\alpha \wedge \beta)$, and $\exists x_i(\alpha)$. The elements of $\mathcal{L}(\tau)$ are called *first-order formulas*. ◁

Let us denote the set of all first-order formulas by FO. From a computer science point of view, it will often be natural to consider *arithmetic structures*, that is, structures defined over a vocabulary that contains \leq^2 , $+$ ³ and \times ³ which *have to be* interpreted as a total order of the universe, addition, and multiplication, respectively. The set of first-order formulas over arithmetic structures is denoted by FO[+, \times]. We extend first-order formulas by the usual abbreviations:

$$\begin{aligned}x \neq y &\equiv \neg(x = y) \\ \alpha \vee \beta &\equiv \neg(\neg\alpha \wedge \neg\beta), \\ \alpha \rightarrow \beta &\equiv \neg\alpha \vee \beta, \\ \alpha \leftrightarrow \beta &\equiv (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha), \\ \forall x_i(\alpha) &\equiv \neg\exists x_i(\neg\alpha).\end{aligned}$$

To increase readability, we use all lowercase Latin letters with or without subscript to refer to variables (such as x_1, x_2, y, z) and we drop unnecessary braces by using the standard *operator precedence* instead, which is in decreasing order: $\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \exists, \forall$. Furthermore, we use the *dot notation*: we denote a dot instead of an opening brace. This virtual brace is closed after the longest formal correct formula. For instance, we may write $\exists x . \alpha$ instead of $\exists x(\alpha)$. We say a variable x is *free* in φ if it is not in the scope of a quantifier, and we denote the set of free variables of φ by $\text{free}(\varphi)$. Formulas without free variables are also called *sentences*, and formulas with $\text{free}(\varphi) = \{x_1, \dots, x_k\}$ are denoted by $\varphi(x_1, \dots, x_k)$. Variables that are not free are bounded (by a quantifier), and we denote the set of bounded variables by $\text{bound}(\varphi)$. The set of variables of φ is the set $\text{var}(\varphi) = \text{free}(\varphi) \cup \text{bound}(\varphi)$. Finally, we define the *quantifier rank* of a formula φ , denoted by $\text{qr}(\varphi)$, as the maximum nesting depth of quantifiers in φ :

$$\text{qr}(\varphi) = \begin{cases} 0 & \text{if } \varphi \text{ is atomic,} \\ \max(\text{qr}(\alpha), \text{qr}(\beta)) & \text{if } \varphi = (\alpha \wedge \beta), \\ \text{qr}(\alpha) & \text{if } \varphi = \neg(\alpha), \\ \text{qr}(\alpha) + 1 & \text{if } \varphi = \exists x(\alpha). \end{cases}$$

► **Definition 12 (Semantic of First-Order Logic)**

Let $\tau = \{R_1^{\alpha_1}, \dots, R_k^{\alpha_k}\}$ be a vocabulary, $\varphi(x_1, \dots, x_\ell)$ be a first-order formula with free-variables x_1, \dots, x_ℓ , and let $S = (V, R_1^S, \dots, R_k^S)$ be a τ -structure. We inductively define the relation $(S, \alpha) \models \varphi$, which states that S is a *model* for φ under an assignment $\alpha: \text{free}(\varphi) \rightarrow V$: We have $(S, \alpha) \models \varphi$ if...

1. φ is an atomic formula $x = y$ and $\alpha(x) = \alpha(y)$;
2. $\varphi(x_1, \dots, x_\ell) = R_i(x_1, \dots, x_\ell)$ and $(\alpha(x_1), \dots, \alpha(x_\ell)) \in R_i^S$;
3. $\varphi(x_1, \dots, x_\ell) = (\psi(x_1, \dots, x_\ell) \wedge \chi(x_1, \dots, x_\ell))$ and $(S, \alpha) \models \psi(x_1, \dots, x_\ell)$ as well as $(S, \alpha) \models \chi(x_1, \dots, x_\ell)$ holds;

4. $\varphi(x_1, \dots, x_\ell) = \neg(\psi(x_1, \dots, x_\ell))$ and $(S, \alpha) \not\models \psi(x_1, \dots, x_\ell)$ holds;
5. $\varphi(x_1, \dots, x_\ell) = \exists y(\psi(x_1, \dots, x_\ell, y))$ and there is an element $u \in V$ such that $(S, \alpha') \models \psi(x_1, \dots, x_\ell, y)$ for

$$\alpha'(x) = \begin{cases} \alpha(x) & \text{if } x \in \text{free}(\varphi), \\ u & x = y. \end{cases}$$

If S is a model for φ under all assignments we abbreviate $S \models \varphi$. In this case, we also say that “ φ is true in S ,” or that “ S satisfies φ .” \triangleleft

► **Example 13**

An $\{E^2\}$ -structure is a simple graph if it is a model for the following sentence:

$$\varphi_{\text{simple}} = \forall x \forall y E(x, y) \rightarrow (x \neq y \wedge E(y, x)).$$

A digraph is called a *tournament* if between every pair of vertices exactly one of the directed edges exists. With other words, a digraph is a tournament if it is a model for the following sentence:

$$\varphi_{\text{tournament}} = \forall x \forall y (x = y \wedge \neg E(x, y)) \vee (x \neq y \wedge (E(x, y) \leftrightarrow \neg E(y, x))).$$

Finally, let us consider for a graph $G = (V, E)$ the following formula:

$$\varphi_{\text{vc}}(x_1, \dots, x_k) = \forall x \forall y E(x, y) \rightarrow \bigvee_{i=1}^k (x = x_i \vee y = x_i).$$

Observe that for every possible assignment α with $(G, \alpha) \models \varphi_{\text{vc}}$ we may define the set $X = \{\alpha(x) \mid x \in \text{free}(\varphi)\}$ with $|X| \leq k$ such that $G[V \setminus X]$ contains no edges. We call such a set X a *vertex cover* of G , a structure that we will encounter frequently in the rest of this thesis. \triangleleft

If we extent first-order logic by relational variables of arbitrary arity, and if we also allow quantifying over such relational variables, we obtain *second-order formulas*. We stipulate that relational variables are denoted by uppercase Latin letters, and we denote the set of all second-order formulas by SO . We say a relational variable is *monadic* if its arity is one. Accordingly, a second-order formula is *monadic* if all its relational variables are monadic – we denote the set of these formulas by MSO . The remaining definitions for second-order formulas are similar to the definitions of first-order formulas. Instead of going into the details here, I will refer the interested reader to standard textbooks [71, 72].

► Example 14

A natural property that can be expressed in second-order logic is that a set $X \subseteq V$ is connected in a given graph $G = (V, E)$. To write down a formula for this statement, we require a characterization of “being connected” that we can grasp with logic. Observe that for a connected set X the following holds: there is an edge between every non-empty proper subset Y of X and $X \setminus Y$. Also observe that this property is not true in unconnected sets. We start with the simple formula that states that a given set is non-empty: $\varphi_{\text{non-empty}}(A) = \exists x \cdot A(x)$.

The next ingredient we need is a way of describing that a set A is a proper subset of another set B :

$$\varphi_{\text{proper-subset}}(A, B) = (\forall x \cdot A(x) \rightarrow B(x)) \wedge (\exists x \cdot \neg A(x) \wedge B(x)).$$

Summarizing, we can write down the following formula, which precisely states that X is connected in G :

$$\begin{aligned} \varphi_{\text{connected}}(X) = \forall Y \left(\varphi_{\text{non-empty}}(Y) \wedge \varphi_{\text{subset}}(Y, X) \right. \\ \left. \rightarrow (\exists x \exists y \ X(x) \wedge \neg Y(x) \wedge Y(y) \wedge E(x, y)) \right). \end{aligned}$$

The cautious reader may observe that this formula is, of course, a bit wordy in order to illustrate the concept. We can condense it to the following equivalent formula:

$$\begin{aligned} \varphi_{\text{connected}}(X) = \forall Y (\exists x \exists y \ X(x) \wedge X(y) \wedge Y(x) \wedge \neg Y(y)) \\ \rightarrow (\exists x \exists y \ X(x) \wedge X(y) \wedge Y(x) \wedge \neg Y(y) \wedge E(x, y)). \quad \triangleleft \end{aligned}$$

► Example 15

We say a graph $G = (V, E)$ is *3-colorable* if there is a mapping $\lambda: V \rightarrow \{1, 2, 3\}$ such that $\lambda(u) \neq \lambda(v)$ for every edge $\{u, v\} \in E$. This property can be expressed by the following existential second-order formula:

$$\varphi_{\text{3col}} = \exists R \exists G \exists B \left(\forall x \ R(x) \vee G(x) \vee B(x) \right) \wedge \left(\forall x \forall y \ E(x, y) \rightarrow \bigwedge_{C \in \{R, G, B\}} \neg C(x) \vee \neg C(y) \right).$$

For instance, we have $\mathfrak{K}_3 \models \varphi_{\text{3col}}$, as this graph is clearly 3-colorable. But the clique on 4 vertices is not 3-colorable, that is, $\mathfrak{K}_4 \not\models \varphi_{\text{3col}}$. ◁

3 BACKGROUND IN COMPLEXITY

Complexity theory is an area of theoretical computer science that studies the algorithmic complexity of *preserving functions* $\rho: \text{STRUC}[\tau] \rightarrow \text{STRUC}[\sigma]$. In simple words that means we wish to state whether it is easy or hard to compute ρ . Formally, we measure the *resources* that a *computational model* requires for the evaluation of $\rho(A)$ for a structure $A \in \text{STRUC}[\tau]$. A *preserving function* here is defined as follows:

► Definition 16 (Preserving Functions)

A function $\rho: \text{STRUC}[\tau] \rightarrow \text{STRUC}[\sigma]$ is called *preserving* if it satisfies the following conditions for all $A, B \in \text{STRUC}[\tau]$:

1. $A \simeq B \implies \rho(A) \simeq \rho(B)$,
2. $|\text{code}(A)| = |\text{code}(B)| \implies |\text{code}(\rho(A))| = |\text{code}(\rho(B))|$. ◁

The standard computational model for parallel algorithms are *uniform circuit families*, in which ρ is evaluated by Boolean circuits. The resources used by this model are the *depth* and the *size* of these circuits. Loosely speaking this corresponds to the parallel time and work we need on a real parallel machine. We will formally introduce this computational model in Section 3.1.

In many scenarios it is too general to study the complexity of an arbitrary preserving function $\rho: \text{STRUC}[\tau] \rightarrow \text{STRUC}[\sigma]$ and, instead, we will often restrict ourselves to the characteristic function of a set $Q \subseteq \text{STRUC}[\tau]$. Here we will consider only such sets Q with the property that all isomorphic structures are either simultaneously contained in Q or not. In other words, the characteristic function of Q must be preserving as well. In complexity theory we call Q a *decision problem* as we have to decide, given a structure $A \in \text{STRUC}[\tau]$, whether we have $A \in Q$. Recall for instance Example 13 where we have defined the formula φ_{vc} , and where we have considered the set $Q = \{ G \in \text{STRUC}[\tau_{\text{graph}}] \mid G \models \varphi_{\text{vc}} \}$. Then Q is exactly the well-known vertex cover problem. We refer to decision problems by small-caps words, for instance, we would identify Q with `vertex-cover`. Instead of stating the exact set representation, we define problems in the following more convenient way, omitting details about the precise definition of the input structure:

► Problem 17 (`vertex-cover`)

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: Is there a set $X \subseteq V$ with $|X| \leq k$ such that $G[V \setminus X]$ contains no edge? ◁

In *parameterized complexity theory* we try to tighten the analysis of a preserving function ρ by taking a *parameterization* $\kappa: \text{STRUC}[\tau] \rightarrow \mathbb{I}$ into account, which itself is a preserving function that assigns to every structure a *parameter value* from some *ordered* and *countable* index set \mathbb{I} . The intuition is that $\kappa(A)$ describes structural properties of A , which we may explicitly use in a refined complexity analysis. We call the tuple (ρ, κ) a *parameterized function* and, similar to decision problems, we call (ρ, κ) or (Q, κ) a *parameterized problem* if ρ is the characteristic function of $Q \subseteq \text{STRUC}[\tau]$. We denote such problems by strings with a leading “p-” with an index that indicates the parameterization. For instance, $p_k\text{-VERTEX-COVER}$ is the vertex cover problem parameterized by the number k . Formal definitions are provided in Section 3.2. It should be noted, however, that these definitions are far less standardized in the literature than they are for classic complexity – we discuss differences between these definitions in Section 3.3.

3.1 CLASSIC COMPLEXITY THEORY

As mentioned in the introduction of this chapter, our primary computational model is the *Boolean Circuit*. We define this computational model in graph theoretical terms.

► **Definition 18 (AC-Circuit)**

An AC-circuit is a relational structure $C = (V(C), E^C, \leq^C, \text{AND}^C, \text{OR}^C)$ over the signature $\tau_{\text{circ}} = (E^2, \leq^2, \text{AND}^1, \text{OR}^1)$ in which $G = (V(C), E^C)$ is an acyclic digraph and \leq^C is a total order of $V(C)$. The elements of $V(C)$ are called *gates*. The relations AND^C and OR^C constitute a partition of the gates that have at least two incoming edges into *and-gates* and *or-gates*. Gates with exactly one incoming edge are called *not-gates*, gates without any incoming edge are called *input-gates*, and the gates without any outgoing edges are called *output-gates*. ◁

The *size* of a circuit is the number of gates, that is, $\text{size}(C) = |V(C)|$. The *depth* of a circuit, denoted by $\text{depth}(C)$, is the maximum length of a path from an input-gate to an output-gate. An AC-circuit with n input- and m output-gates naturally computes a Boolean function $C: \{0, 1\}^n \rightarrow \{0, 1\}^m$.

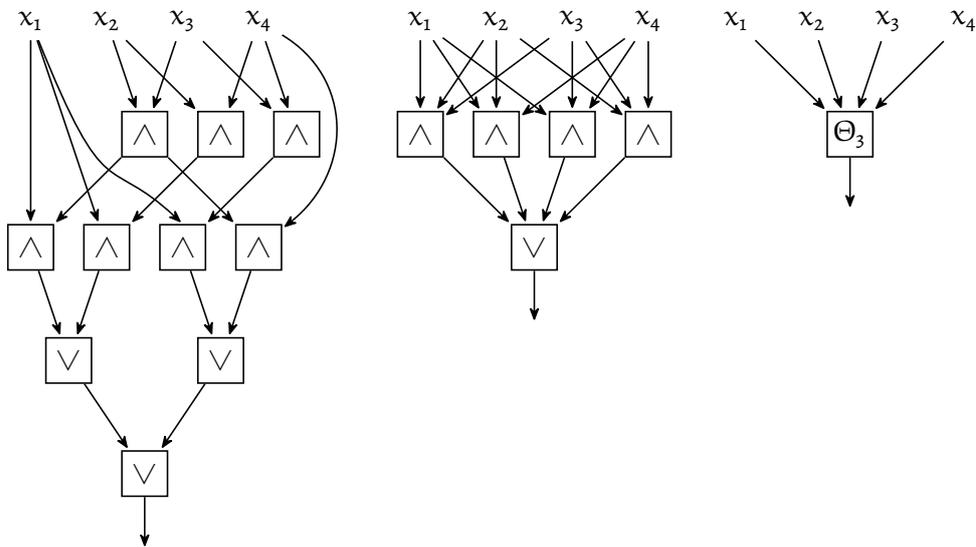
► **Definition 19 (Computation of an AC-Circuit)**

For an input $w = b_1 b_2 \dots b_n$, we inductively define a Boolean labeling $\lambda: V \rightarrow \{0, 1\}$ as $\lambda(x_i) = b_i$ for the input-gates x_1, x_2, \dots, x_n (recall that the vertices are ordered). Define $\lambda(u) = (\lambda(v) + 1) \bmod 2$ if u is a not-gate (that is, u has in-degree one); and for vertices u with in-degree at least two define

$$\lambda(u) = \begin{cases} \min\{\lambda(v) \mid (v, u) \in E^C\} & \text{if } u \in \text{AND}^C \\ \max\{\lambda(v) \mid (v, u) \in E^C\} & \text{if } u \in \text{OR}^C \end{cases}$$

The output of the computation is the bitstring $w' = \lambda(y_1)\lambda(y_2) \dots \lambda(y_m)$ for the output-gates y_1, \dots, y_m . ◁

If we restrict the in-degree of C to be at most two, then we call C an NC-circuit. A TC-circuit is an AC-circuit extended by Θ_c -gates, which evaluate to 1 if at least c predecessors of the gate evaluate to 1. The abbreviation NC stands for “Nick’s Class,” as the corresponding complexity class (a precise definition follows) was named after Nicholas Pippenger [141]. The names of the other classes are based on this choice—the “A” in AC stands for “alternating,” referring to the connection of these circuits to alternating Turing machines; the “T” in TC stands for “threshold,” which is exactly the functionality of the added Θ_c -gates. Different circuits are illustrated in the following graphic: It shows from left to right an NC-, an AC-, and a TC-circuit. For readability, the output-gates are highlighted with an outgoing-edge.



Since circuits can only compute functions for a fixed input length, we need a circuit family $(C_n)_{n \in \mathbb{N}}$ if we wish to compute functions with a variable input length. We will use such families to study the complexity of preserving functions. This will, however, be difficult if the circuits of the family are pairwise very different. In fact, we wish that all circuits of a family “look the same.” To formalize this property, we consider *uniform* families.

► Definition 20 (AC^i and FAC^i)

A preserving function $\rho: \text{STRUC}[\tau] \rightarrow \text{STRUC}[\sigma]$ is in uniform FAC^i if there is a constant $c \in \mathbb{N}$ and a family $(C_n)_{n \in \mathbb{N}}$ of AC-circuits such that:

1. $C_{|\text{code}(A)|}(\text{code}(A)) = \text{code}(\rho(A))$ for all $A \in \text{STRUC}[\tau]$;
2. $\text{depth}(C_n) \leq c \cdot \log^i n$;
3. $\text{size}(C_n) \leq n^c$;

4. There is a Turing machine that on input of $\text{bin}(i)\#\text{bin}(n)$, where $\text{bin}(\cdot)$ is the binary encoding of natural numbers, outputs the i th bit of $\text{code}(C_n)$ in at most $O(\log n)$ steps.

The class AC^i contains all decision problems whose characteristic functions are preserving and contained in FAC^i \triangleleft

We further define $\text{AC} = \bigcup_{i=0}^{\infty} \text{AC}^i$ and we define NC^i and NC as well as TC^i and TC analogously. There are other definitions of uniformity that can be derived if the power of the Turing machine is altered. The uniformity definition we use here, in the literature known as DLOGTIME -uniformity, is the strongest form of uniformity commonly considered [24]. It has the following well-known property:

► **Fact 21** ([111, 164])

The set of decision problems in uniform AC^0 is exactly the set of decision problems that can be defined in $\text{FO}[+, \times]$. \triangleleft

A direct consequence of this fact is the following useful lemma:

► **Lemma 22**

There are uniform families $(C_n^+)_n \in \mathbb{N}$, $(C_n^\times)_n \in \mathbb{N}$, $(C_n^{\text{mod}})_n \in \mathbb{N}$ of FAC^0 -circuits that have $2n$ inputs $x_1, \dots, x_n, y_1, \dots, y_n$ and n^2 outputs z_1, \dots, z_{n^2} such that:

1. each circuit expects that exactly one input x_i and one input y_j is set to 1, while the others are set to 0;
2. all outputs of C_n^+ are 0, with the sole exception of z_{i+j} ;
3. all outputs of C_n^\times are 0, with the sole exception of $z_{i \cdot j}$;
4. all outputs of C_n^{mod} are 0, with the sole exception of $z_{i \bmod j}$.

Proof. Since the given numbers are encoded in *unary* (and in particular bounded by the size of the universe), the existence of $(C_n^+)_n \in \mathbb{N}$ and $(C_n^\times)_n \in \mathbb{N}$ follows in principle directly by the equivalence of uniform AC^0 and $\text{FO}[+, \times]$. However, we have to take a little care about the encoding of the structure, and we require a way to parse the first and second input bit – but both can easily be achieved with first-order formulas equipped with the relations $+^3$, \times^3 and \leq^2 . To obtain the family $(C_n^{\text{mod}})_n \in \mathbb{N}$, we have to describe the unary modulo operation within $\text{FO}[+, \times]$, which is a standard exercise solved by the following formula:

$$\text{mod}(x, y, z) = \exists a \exists b \left(\times(a, y, b) \wedge +(b, z, x) \wedge \leq(z, y) \wedge z \neq y \right). \quad \square$$

Observe that a problem that lies in AC can efficiently be implemented on a *parallel machine*, because a circuit C_n can be evaluated in parallel. In order to do so, we

layer the circuit and evaluate all gates of one layer in parallel. The *parallel time* of the algorithm is then bounded by $O(\log^i n)$, as each layer can be evaluated in constant parallel time and since there are at most $O(\log^i n)$ layers. The *work* of the algorithm, that is, the total number of computational steps, is bounded by the number of gates, and therefore by $O(n^c)$.

Based on this observation we say a decision problem is *parallel tractable* if it lies in AC, and we say it is *parallel intractable* otherwise. Parallel intractable problems may lie in the class P, which contains problems decidable by a uniform family of AC-circuits of polynomial size (but without a further depth restriction). Such circuits can probably not be simulated in parallel, but they can be simulated in polynomial time – problems in P are therefore called (*sequentially*) *tractable*. Unfortunately, many interesting problems do not lie in P and are therefore considered *intractable*. There is a broad range of further complexity classes to study such problems. An important one is NP, the set of problems decidable by a uniform family of AC-circuits of polynomial size that has access to nondeterministic input bits. Based on these definitions, we obtain the following well-known hierarchy:

$$\text{NC}^0 \subsetneq \text{AC}^0 \subsetneq \text{TC}^0 \subseteq \text{NC}^1 \subseteq \text{AC}^1 \subseteq \text{TC}^1 \subseteq \dots \subseteq \text{NC} = \text{AC} = \text{TC} \subseteq \text{P} \subseteq \text{NP}.$$

Observe that it is unknown for almost all inclusions in this hierarchy, whether they are proper or not. Furthermore, almost all of these results are trivial – the sole exception being $\text{AC}^0 \subsetneq \text{TC}^0$, which was shown by Furst, Saxe, and Sipser [93].

We can formulate statements about the complexity of a problem by “sorting” it into the hierarchy: If we show a problem lies in a complexity class, we essentially provide an *upper bound* on the complexity of the problem. To provide a *lower bound*, we need the concept of *reduction* and *hardness*.

► Definition 23 (ACⁱ-Reduction)

An ACⁱ-*reduction* from a decision problem $Q_1 \subseteq \text{STRUC}[\tau]$ to $Q_2 \subseteq \text{STRUC}[\sigma]$ is a mapping $R: \text{STRUC}[\tau] \rightarrow \text{STRUC}[\sigma]$ with $R \in \text{FAC}^i$ and $A \in Q_1 \iff R(A) \in Q_2$ for all structures $A \in \text{STRUC}[\tau]$. ◁

► Definition 24 (C-Hardness)

Let \mathcal{C} be a complexity class. A decision problem $Q \subseteq \text{STRUC}[\tau]$ is said to be *C-hard* if all problems in \mathcal{C} reduce to Q via AC⁰-reduction. ◁

A problem that is hard for some complexity class can be seen as the most “difficult” one of this class, as we can solve all other problems of the class if we can solve this single problem. Hardness can therefore be seen as a lower bound – at least if we assume that the hierarchy does not collapse. Finally, we say that a problem is complete for a complexity class if the lower and upper bound match.

► Fact 25

Let \mathcal{C}_1 and \mathcal{C}_2 be two complexity classes with $\text{AC}^0 \subseteq \mathcal{C}_1 \subseteq \mathcal{C}_2$ and let $Q \subseteq \text{STRUC}[\tau]$ be \mathcal{C}_2 -hard. We have $Q \in \mathcal{C}_1$ if, and only if, $\mathcal{C}_1 = \mathcal{C}_2$. ◁

► Definition 26 (\mathcal{C} -Completeness)

A decision problem $Q \subseteq \text{STRUC}[\tau]$ is *complete* for \mathcal{C} (or \mathcal{C} -complete) if Q is \mathcal{C} -hard and $Q \in \mathcal{C}$. ◁

3.2 PARAMETERIZED COMPLEXITY THEORY

Similar to circuits that are a computational model for parallel algorithms, we will use parameterized circuits that serve as a computational model for the computation of parameterized functions. Recall that a parameterized function is a tuple (ρ, κ) consisting of two preserving functions $\rho: \text{STRUC}[\tau] \rightarrow \text{STRUC}[\sigma]$ and $\kappa: \text{STRUC}[\tau] \rightarrow \mathbb{I}$. The idea is to analyze the complexity of computing $\rho(A)$ by considering $|\text{code}(A)|$ and $\kappa(A)$, instead of measuring the consumed resources only with respect to the encoding length. Intuitively, smaller parameters (recall that \mathbb{I} is ordered) represent “easier” instances, and we will develop algorithms that exploit this circumstance. Note that parameterized functions are a generalization of functions, as every function is a parameterized function with the *trivial parameterization*. This trivial parameterization simply maps to a set \mathbb{I}_o that contains only a single element. In general, there is nothing special about \mathbb{I} and we will often simply have $\mathbb{I} = \mathbb{N}$.

When we study the complexity of ρ from a parameterized point of view, we have to be careful not to “hide” the complexity of computing ρ in the second function κ . In simple words, the computation of κ should be “easier” than the computation of ρ . As we study small parallel circuit classes, we stipulate this condition for this thesis as follows. We will discuss the issue of computing κ in a bit more detail in Section 3.3.

► Proviso 27

We request for all parameterizations $\kappa: \text{STRUC}[\tau] \rightarrow \mathbb{I}$ considered within this thesis that (i) the set \mathbb{I} is a set of ι -structures for some arbitrary but fixed vocabulary ι , in symbols: $\mathbb{I} \subseteq \text{STRUC}[\iota]$; and that (ii) we have $\kappa \in \text{FAC}^o$. ◁

The following definition provides our basic parameterized computational model. It also reveals the reason behind the choice of the term “index set” for the set \mathbb{I} . Recall that this index set is ordered and countable, a fact that we use in the definition.

► Definition 28 (para-ACⁱ and para-FACⁱ)

A parameterized function $(\rho: \text{STRUC}[\tau] \rightarrow \text{STRUC}[\sigma], \kappa: \text{STRUC}[\tau] \rightarrow \mathbb{I})$ is said to be in uniform para-FACⁱ for some $i > 0$ if there is a constant $c \in \mathbb{N}$, a computable function $f: \mathbb{I} \rightarrow \mathbb{N}$, and a family $(C_{n,k})_{n \in \mathbb{N}, k \in \mathbb{I}}$ of AC-circuits such that:

1. $C_{|\text{code}(A)|, \kappa(A)}(\text{code}(A)) = \text{code}(\rho(A))$ for all $A \in \text{STRUC}[\tau]$;
2. $\text{depth}(C_{n,k}) \leq f(k) + c \log^i n$;
3. $\text{size}(C_{n,k}) \leq f(k) \cdot n^c$;

4. There is a Turing machine that on input $\text{bin}(i)\#\text{code}(k)\#\text{bin}(n)$ outputs the i th bit of $\text{code}(C_{n,k})$ in at most $f(k) + c \log(n)$ steps. Here, $\text{bin}(\cdot)$ is the binary encoding of natural numbers.

The class para-AC^i contains all parameterized problems that have a characteristic function that lies in para-FAC^i \triangleleft

Additionally, we define para-AC^0 as above, but require the depth to be constant (independent of n and k). The classes para-NC^i , para-NC , para-TC^i , and para-TC are defined analogously to the previous section. These classes inherit their inclusion structure from the classical classes, so we have

$$\begin{aligned}
& \text{para-NC}^0 \subsetneq \text{para-AC}^0 \subsetneq \text{para-TC}^0 \\
& \subseteq \text{para-NC}^1 \subseteq \text{para-AC}^1 \subseteq \text{para-TC}^1 \\
& \subseteq \text{para-NC}^i \subseteq \text{para-AC}^i \subseteq \text{para-TC}^i \\
& \subseteq \text{para-NC} = \text{para-AC} = \text{para-TC} \\
& \subseteq \text{FPT}.
\end{aligned}$$

Here FPT (for *fixed-parameter tractable*) is the parameterized analogue of P, that is, the class of problems decidable by a uniform family of AC-circuits of size $f(k) \cdot n^c$, but without any further depth restriction. As in the previous section, we can grasp problems in para-AC as *parallel fixed-parameter tractable*. Alternatively, we could define that a problem is parallel fixed-parameter tractable if there is a parallel algorithm that solves the problem and that is allowed to invest $f(k)$ time at each parallel step. This results in a class of functions that is slightly more powerful than para-AC^i (here, the $f(k)$ term is just additive), but which we can embed into the previous hierarchy.

► Definition 29 (para-AC^{i^t} and para-FAC^{i^t})

A parameterized function (ρ, κ) lies in uniform para-FAC^{i^t} (pronounced “para-f-a-c-i-up”) for any $i \geq 0$ if there is a family $(C_{n,k})_{n \in \mathbb{N}, k \in \mathbb{I}}$ of AC-circuits defined as for para-FAC^i , but with $\text{depth}(C_{n,k}) \leq f(k) \cdot c \log^t n$. In particular, circuits of para-FAC^{0^t} have a depth of $f(k)$. The class para-AC^{i^t} contains all parameterized problems that have a characteristic function that lies in para-FAC^{i^t} \triangleleft

The same definition can be used for NC- and TC-circuits. Note that by definition we have $\text{para-AC}^i \subseteq \text{para-AC}^{i^t} \subseteq \text{para-AC}^{i^t + \epsilon}$ for all $i \geq 0$. We will further see in Chapter 4.2 (Corollary 37) that we have the proper inclusion $\text{para-AC}^0 \subsetneq \text{para-AC}^{0^t}$

In order to compare parameterized problems we require a new (parameterized) reduction. In comparison to an AC^0 -reduction, such a reduction has to fulfill an additional property: As we measure the complexity with respect to the input length *and* the parameter, we should not let the parameter grow arbitrarily during the reduction process. The following definition fulfills this property. Definitions for hardness and completeness can be derived analogously.

► Definition 30 (para-ACⁱ-reduction)

A para-ACⁱ reduction from a parameterized problem (Q, κ) to a problem (Q', κ') with $Q \subseteq \text{STRUC}[\tau]$, $\kappa: \text{STRUC}[\tau] \rightarrow \mathbb{I}$, $Q' \subseteq \text{STRUC}[\sigma]$, $\kappa': \text{STRUC}[\sigma] \rightarrow \mathbb{I}$, is a mapping $R: \text{STRUC}[\tau] \rightarrow \text{STRUC}[\sigma]$ such that for all $A \in \text{STRUC}[\tau]$ we have:

1. $A \in Q \iff R(A) \in Q'$;
2. $\kappa'(R(A)) \leq f(\kappa(A))$ for a computable function $f: \mathbb{I} \rightarrow \mathbb{I}$.

Additionally, it is required that $(R, \kappa) \in \text{para-FAC}^i$ ◁

We will require only very little of the machinery of *parameterized intractability*, as we are primarily interested in subclasses of FPT and, thus, in problems that are highly tractable from a parameterized perspective. There are actually many classes “above” FPT that can all be considered as intractable. However, one hierarchy, called the *weft-hierarchy*, is usually sufficient to express intractability. Fortunately, the definitions of the classes within this hierarchy are in terms of circuit complexity and, thus, fit nicely into our framework. The weft-hierarchy is defined in terms of the following restricted version of the *weighted circuit satisfiability* problem:

► Problem 31 (WEIGHTED-CIRCUIT-SATISFIABILITY)

Instance: An AC-circuit C and a number $k \in \mathbb{N}$,

Question: Is there a string $w \in \{0, 1\}^*$ with $\sum_{i=1}^{|w|} w[i] = k$ and $C(w) = 1$? ◁

This problem is quite powerful, as we can easily encode problems such as the satisfiability problem of propositional logic into it. To describe the hierarchy, we restrict the problem to a smaller family of circuits. Let $\mathcal{C}_{t,d}$ be the family of AC-circuits in which every circuit has depth d and contains on any path from an input-gate to an output-gate at most t vertices with more than two incoming edges. The value t is called the *weft* of the circuits. For every $t \geq 1$, the t th-level of the weft-hierarchy, denoted by $W[t]$, is the class of problems that can be reduced via an FPT-reduction to p_k -WEIGHTED-CIRCUIT-SATISFIABILITY restricted to circuits from $\mathcal{C}_{t,d}$ for some arbitrary $d \geq 1$. An FPT-reduction is defined as in Definition 30, but without a depth restriction for the circuits. It follows from the definition that the classes form the following inclusion structure:

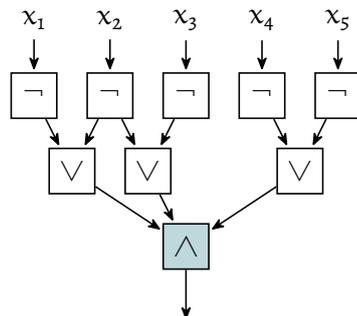
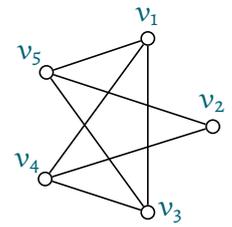
$$\text{FPT} \subseteq W[1] \subseteq W[2] \subseteq \dots \subseteq W[t],$$

and the conjecture used to express intractability is $\text{FPT} \subsetneq W[1]$. This definition is best understood with an example.

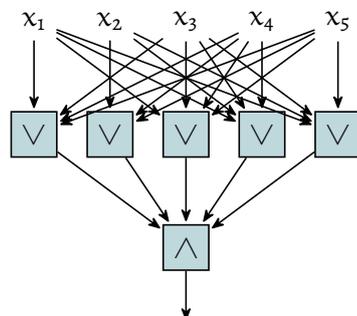
► Example 32

Assume we are given a graph $G = (V, E)$ on n vertices and we wish to check whether G contains a clique of size k (that is, a set $S \subseteq V$ with $|S| = k$ such that all vertices in S are pairwise adjacent). We can reduce this problem to WEIGHTED-CIRCUIT-SATISFIABILITY with circuits of depth three and weft one, which implies that the

problem lies in $W[1]$. In order to do so, we construct a circuit with n input-gates x_1, \dots, x_n and the obvious meaning that setting x_i to 1 corresponds to taking vertices v_i into the clique. The circuit has to verify that for every non-edge $\{v_i, v_j\} \notin E$ at least one element of $\{v_i, v_j\}$ is not contained in the clique, that is, either x_i or x_j (or both) is set to 0. This test can easily be implemented by negating every input and using or-gates with two incoming edges for every non-edge. That all these tests are affirmative can be tested with a single and-gate of high fan-in (this is the gate that increases the weft of the circuit). To illustrate the reduction, consider the graph at the margin. The circuit that we have just sketched is illustrated in the following figure, where the gate that is relevant for the weft is highlighted.



Now assume we are given another graph $G = (V, E)$ on n vertices, but this time we seek a dominating set of size k (a set $S \subseteq V$ with $|S| = k$ such that for all $v \in V$ we have $N[v] \cap S \neq \emptyset$). We can reduce this problem to **WEIGHTED-CIRCUIT-SATISFIABILITY** with circuits of depth and weft two. The reduction is quite similar to the previous one: The circuit again has n input-gates x_1, \dots, x_n indicating which vertices are part of the solution S . For dominating set, the circuit has to test for every vertex if either itself or one of its neighbors is contained in S (meaning that the corresponding input-gate is set to one). These tests can be realized by n or-gates of high fan-in – since they can be used in parallel, they increase the depth and the weft of the circuit by one. Finally, the circuit has to check whether all these tests are affirmative with an additional high degree and-gate, which increased the depth and weft to two. The reduction shows that the problem lies in $W[2]$. For the previous example graph, the resulting circuit is the following:



◁

3.3 DIFFERENTIATION OF PARAMETERIZED COMPLEXITY

We note that there are different definitions of parameterized problems, and especially of the class FPT, in the literature. Considering FPT, these differences seem small and the choice of definition is mainly a matter of taste. However, considering smaller classes like para-AC^0 as we do, reveals more technical differences, which are, as I believe, worth discussing.

The classical definition is due to Downey and Fellows [70], who define a parameterized problem to be a language $L \subseteq \Sigma^* \times \mathbb{N}$. Downey and Fellows distinguish three definitions of FPT: a problem is said to be (i) in *strongly uniform* FPT if an instance (w, k) can be solved in time $f(k) \cdot |w|^c$ for a computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ and a constant $c \in \mathbb{N}$; (ii) in *uniform* FPT if we drop the restriction that f must be computable; and (iii) in *nonuniform* FPT if for every parameter value k there is an extra algorithm solving just the instances with this value. It is known that these three definitions lead to distinct classes [69], from which the first is the most commonly used definition in the literature [59].

Another definition that is commonly used was given by Flum and Grohe [85], who defined a parameterized problem to be a tuple (Q, κ) with $Q \subseteq \Sigma^*$ and $\kappa: \Sigma^* \rightarrow \mathbb{N}$. This definition is a bit more natural, as we can use a classical language Q and just add a parameterization to it. For instance if Q is VERTEX-COVER, we may consider many parameterizations without changing the language. This was also the main reason why I chose to use this definition in this thesis. However, the definition comes with a drawback: κ has to be computed. This is crucial, as we study the complexity of Q and may not want to “hide” some of this complexity in the evaluation of κ . Therefore, Flum and Grohe required the parameterization to be computable in polynomial time [85]. This seems generally reasonable for the study of FPT, but already has some issues there. In particular, some standard parameters, such as the treewidth of the input structure, do not seem to be polynomial-time computable. This phenomenon gets worse if we study subclasses of FPT, for instance para-AC^0 as a polynomial time computable parameterization could implement functions that are not in para-FAC^0 . The result is, essentially, that such small classes are not closed under natural reduction (what they are in the Downey and Fellows definition). This was first observed by Chen and Flum with the example of p -PARITY [52]. It should be noted that the closeness property is especially important in the context of kernelization, which in essence is a self-reduction that is fundamentally entangled with parameterized complexity – we will study it in Chapter 6.

There are two possible ways out of this misery: One could adapt the definition of parameterized reductions, which was suggested by Chen and Flum [52]. However, the result is an unnatural reduction, which additionally is not well suited for kernelizations. Alternatively, one could require that κ is easier to compute, say in logarithmic space as suggested by Elberfeld, Stockhusen, and Tantau [76]. Since we

wish to study para-AC° , we have to require that $\kappa \in \text{FAC}^\circ$ holds, as my coauthors and myself required [19–22]. However, we should note that we have not much freedom if the parameter must be FAC° -computable, as this class is very restrictive. For instance, consider any graph problem and use as parameterization the maximum degree. This parameter is not computable in FAC° . To fix this new misery, Tantau and myself have suggested to “patch” the language by adding an upper bound $d \in \mathbb{N}$ to the problem instance [20]. The new problem is then the original one *together with* the question whether the maximum degree is smaller than d , which we use as parameter (and which then is easily computable in FAC°). With this fix, the definition is actually quite close to the original definition by Downey and Fellows – however, there is still an advantage if the parameter is in fact FAC° -computable.

Concerning subclasses of FPT, it should be noted that the first complete definition was given by Flum and Grohe [85], who have generally defined $\text{para-}\mathcal{C}$ for any complexity class \mathcal{C} . This definition is related to a similar definition by Cai et al., who have defined subclasses of FPT in terms of classical complexity classes extended by an advice function [46]. In contrast, the definition by Flum and Grohe states that a parameterized problem is in $\text{para-}\mathcal{C}$ if an instance can be solved in \mathcal{C} after an arbitrary precomputation on the parameter. This definition has primarily model checking in mind: Given a structure S and a formula φ , which is the parameter, we wish to know whether we have $S \models \varphi$. Such problems can be solved for various types of structures by translating φ into an automaton that can be simulated on input S (compare Chapter 8). Unfortunately, in most scenarios other than model checking this definition is not so natural and leads often to a simple padding argument. Therefore, Elberfeld, Stockhusen, and Tantau provided equivalent definitions with concrete computational models for parameterized logarithmic space and some parameterized circuit classes [76]. Based on this work, Stockhusen, Tantau and myself provided a rigorous definition of parameterized circuit classes [19]. These definitions obtained some refinements in follow-up works by various authors, and the resulting definitions are used in this thesis [20, 52, 140]. The only difference is that we use relational structures instead of strings, but this is a matter of taste as both representations are equivalent by Example 3 and Definition 4.

Part I

Theory of Parallel Parameterized Algorithms

In this first and primary part of the thesis, we will study parallel parameterized algorithms and parameterized circuit complexity. Most of the results are formulated from an algorithm design point of view and constitute a rich toolbox that can be used to explore this area further. I hope the reader will find it both, useful and enjoyable.

Preliminary versions of many results of this part were previously presented at the following conferences:

- [19] Max Bannach, Christoph Stockhusen, and Till Tantau: *Fast Parallel Fixed-Parameter Algorithms via Color Coding*. In Proceedings of the 10th International Symposium on Parameterized and Exact Computation (IPEC 2015).
- [20] Max Bannach and Till Tantau: *Parallel Multivariate Meta-Theorems*. In Proceedings of the 11th International Symposium on Parameterized and Exact Computation (IPEC 2016).
- [21] Max Bannach and Till Tantau: *Computing Hitting Set Kernels By AC^0 -Circuits*. In Proceedings of the 35th Symposium on Theoretical Aspects of Computer Science (STACS 2018).
- [22] Max Bannach and Till Tantau: *Computing Kernels in Parallel: Lower and Upper Bounds*. In Proceedings of the 13th International Symposium on Parameterized and Exact Computation (IPEC 2018).

4 A TOOLBOX OF BASIC PARALLEL PARAMETERIZED ALGORITHMS

We will develop a broad range of parallel parameterized algorithms within this thesis. The design of parallel algorithms is usually more challenging than the development of the corresponding sequential counterpart, as we have to discover structures in the problem that we can handle in parallel. This does not change in a parameterized point of view. In fact, things even become a bit more challenging in this young field, as we do not have a collection of standard algorithms on which we can rely on. We will therefore start by assembling a toolbox of basic parallel parameterized algorithms in this chapter. The first ingredient in our toolbox is a way to break symmetries. We achieve this with parallel algorithms for the independent set problem in graphs of bounded degree.

- ▷ Informal Version of Theorem 33 and Lemma 46.

Parameterized by the maximum degree of the graph, a *maximal* independent set can be computed in $\text{para-FAC}^{\circ+\epsilon}$. Parameterized by both, the maximum degree and the size of the solution, a *maximum* independent set can be computed in para-AC° ◁

The second ingredient that we will add to our toolbox is a collection of algorithms that can answer reachability and distance queries on graphs.

- ▷ Informal Version of Lemma 34, Lemma 35, Theorem 36, and Theorem 39.

Given a graph $G = (V, E)$ and a parameter $k \in \mathbb{N}$, a $\text{para-FAC}^{\circ 1}$ -circuit can simulate a depth-first (breadth-first) search starting at some vertex $s \in V$ up to distance k . In fact, the parameterized *alternating* distance problem is complete for $\text{para-AC}^{\circ 1}$ ◁

By combining the theorem with a result by Beame, Impagliazzo, and Pitassi [25] we will unconditionally deduce $\text{para-AC}^{\circ} \subsetneq \text{para-AC}^{\circ 1}$

The last ingredient that we will add to the toolbox may not appear too sparkling at the first sight. However, as it will turn out, it is the most fundamental result I will present in this chapter, which will serve as engine for many algorithms that we study in the rest of this thesis – I would even go as far as to say that it is a cornerstone of parallel parameterized constant-time computation. We will derandomize the color coding technique due to Alon, Yuster, and Zwick [6]:

- ▷ Informal Version of Theorem 42.

Color coding can be derandomized in para-AC° ◁

4.1 FINDING MAXIMAL INDEPENDENT SETS IN GRAPHS OF BOUNDED DEGREE

The most elemental step in many parallel algorithms is *symmetry breaking*, that is, the detection of parts of the input that can be handled in parallel. Formally, we may model this task as finding a *maximum* independent set in the conflict graph: Every task is represented by a vertex and there is an edge between two vertices if, and only if, the corresponding tasks cannot be executed at the same time. Unfortunately, the INDEPENDENT-SET problem is NP-hard in general [115] and solving such a problem as preprocessing seems a bit exaggerated. Furthermore, the problem is also $W[1]$ -hard parameterized by the solution size [59] and, thus, even FPT-power will not allow an “efficient” preprocessing. To circumnavigate this difficulty, the parallel community is usually contended with the *maximal* version of the problem. This version admits a simple $O(n + m)$ sequential algorithm. The first parallel algorithm due to Karp and Wigderson runs in $O(\log^4 n)$ parallel time [116]. This bound was improved to $O(\log^2 n)$ parallel time independently by Luby [130], and by Alon, Babai, and Itai [5]. To this day, it is still an open problem whether one can find a maximal independent set in parallel logarithmic time.

From a parameterized point of view, we hope to improve this bounds with respect to n in exchange for a higher time bound with respect to the parameter. In the setting of maximal independent sets, the parameter solution size does not make sense, so we have to consider other natural parameters. If we expect that the task that we want to solve is well suited for parallelization, we may hope that the degree of the conflict graph is small. Fortunately, this actually will be the case whenever we wish to find maximal independent sets in the rest of this thesis. Thus, we use the maximum degree of the input graph as parameter and formulate the following result, where $\log^*(\cdot)$ is the iterated logarithm defined as:

$$\log^*(x) = \begin{cases} 1 + \log^*(\log(x)) & \text{for } x > 1, \\ 0 & \text{for } x \leq 1. \end{cases}$$

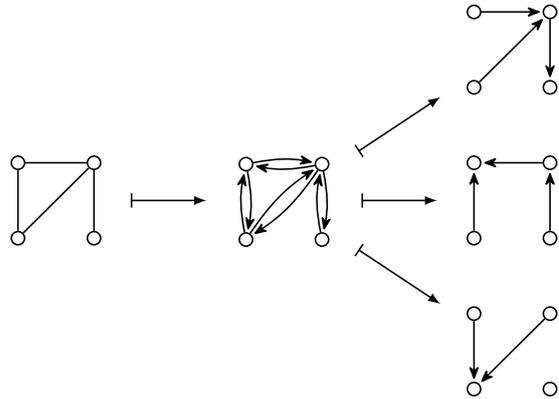
► **Theorem 33**

There is a uniform family of FAC-circuits of depth $f(k) + \log^* |V|$ and size $f(k) \cdot |V|^c$ that, on input of an undirected graph $G = (V, E)$ and an integer k , outputs either that the maximum degree of G exceeds k or a maximal independent set I of G .

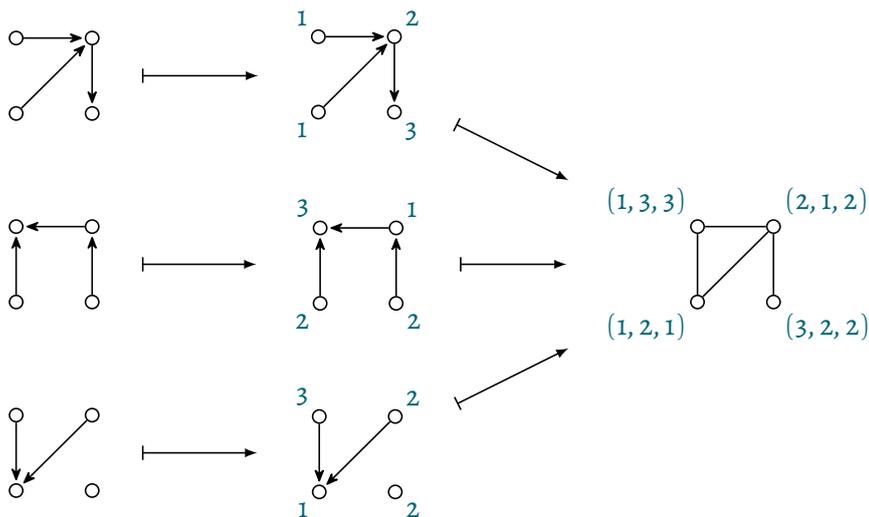
Proof. Let us for simplicity assume that AC-circuits of size $f(k) \cdot n^c$ may count up to $f(k)$ and that, thus, the circuit can check the degree of every vertex and can immediately reject if any degree exceeds k . That this is indeed possible will follow by another basic algorithm that we present in Section 4.3.

The circuit we present here implements the algorithm from Goldberg, Plotkin, and Shannon to compute maximal independent sets in graphs of bounded degree [100].

The circuit interprets G as directed graph \vec{G} by considering each edge $\{u, v\}$ as two directed edges (u, v) and (v, u) . The edge set of this graph is partitioned into k sets E_1, \dots, E_k such that each of the graphs $\vec{G}_i = (V, E_i)$ has only vertices of out-degree at most 1. This partition can be computed in depth $f(k)$ as the circuit has essentially to count up to k . The following figure illustrates an exemplary run of the procedure.



The circuit now performs the following operations on all \vec{G}_i in parallel: First, in constant depth, an initial coloring of every \vec{G}_i is computed by assigning to each vertex v_i the color $i \in \mathbb{N}$, which needs at most $\log |V|$ bits. This coloring can be improved to a coloring with $\log |V|$ colors in constant depth: Replace the color c of each vertex v by $2k + b$, where k is the position of the lowest bit on which c differs from the color of the unique successor of v , and where b is the value of this bit. Computing this improvement consecutively $\log^* |V|$ times yields a coloring with 6 colors [100]. Given the colorings of the k graphs \vec{G}_i , we can compute a 6^k coloring of G by assigning to each vertex the k -tuple of colors that this vertex has in the different \vec{G}_i .



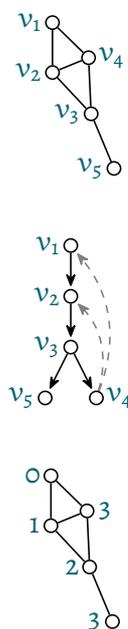
Finally, the circuit initializes a set $I = \emptyset$, iterates over the colors and, in parallel, adds all vertices of the current color, which do not have a neighbor in I , to I . As each step can be performed in a constant number of AC-layers, the set I can be computed in $f(k)$ AC-layers. The circuit outputs I , which is a maximal independent set. The total depth of the circuit is $f(k) + \log^* |V|$. \square

From the point of view of parameterized complexity classes, the aforementioned lemma yields a $\text{para-FAC}^{\circ+\epsilon}$ -circuit for the computation of maximal independent sets on graphs of bounded degree. The result raises the question whether we can improve it to para-FAC° . Unfortunately, this seems unlikely – at least with an algorithm that is similar to the algorithm by Goldberg, Plotkin, and Shannon, since the color trick used by the algorithm requires $\log^* |V|$ iterations to converge. However, if we use the size of the sought independent set as additional parameter, we can even solve the *maximum* version of the problem in para-AC° . We require more machinery to prove this fact – it will be presented at the end of this chapter in Theorem 46.

4.2 GRAPH TRAVERSAL

As mentioned earlier, most of the problems we study in this thesis are graph problems. It will therefore be convenient to have algorithms at hand that can answer basic questions on graphs. The primitive operations on graphs are *reachability queries* (“is there a path from s to t ?”) and *distance queries* (“is there a path from s to t of length at most d ?”). The former is a classical L-complete problem [143], while the later is known to be NL-complete [160]. Therefore, we can solve both problems in AC! In fact, we can even compute the full distance matrix in FAC!

In the parameterized setting we may hope to improve these results for a suitable parameter. Natural candidates are the actual distance between s and t , or the length of the longest path in G . Instead of just providing a decision procedure for the parameterized distance problem, we will actually implement parameterized parallel versions of the depth-first search and the breadth-first search algorithms. This in return will additionally allow us to use properties of these algorithms throughout the rest of this thesis. To formalize this idea, we need a suitable representation of a depth-first search (breadth-first search) run. Let $G = (V, E)$ be a graph with $s \in V$, and let T be a depth-first search tree of G starting at s , a *depth-first search labeling* is a mapping $\lambda_s : V \rightarrow \mathbb{N}$ such that $\lambda_s(v)$ is the distance from s to v in T . The figure in the margin shows from top to bottom: an example graph, a depth-first search tree starting at v_1 , and a corresponding depth-first search labeling. Similarly, we can define a *breadth-first search labeling* with respect to a breadth-first search tree. Notice that in this case the labeling is actually the (path) distance from s to the other vertices. We first handle the computation of breadth-first search labelings, which will yield a natural parallel algorithm.



► **Lemma 34**

There is a uniform family of FAC-circuits of depth $O(k)$ and size $f(k) \cdot |G|^c$ that, on input of a graph $G = (V, E)$, a vertex $s \in V$, and an integer k , outputs a breadth-first search labeling for the vertices in G that are at a distance of at most k to s .

Proof. Our circuit starts by assigning color 0 to s . The circuit is build up of layers, where layer $i + 1$ assigns color $i + 1$ to each vertex that is not colored yet and that has at least one vertex of color i as neighbor. The algorithm stops if all vertices are colored, or at the latest after k layers. After a run of the algorithm, each vertex that has obtained a color is in the same connected component as s and, furthermore, the colors constitute a breadth-first search labeling starting at s . \square

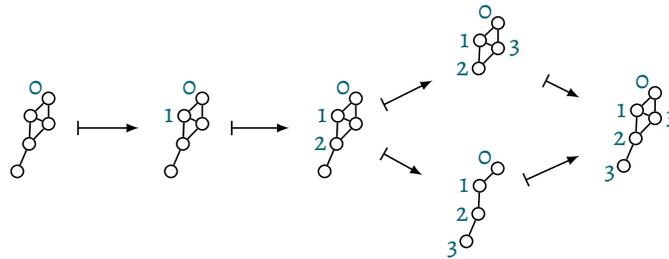
Computing a depth-first search labeling turns out to be more complicated, since an AC-circuit of the desired depth cannot simply follow a path of the search tree and “backtrack” once it reaches a leaf, as in this case the depth of the circuit would not be bounded by the longest path of the input graph.

► Lemma 35

There is a uniform family of FAC-circuits of depth $f(k)$ and size $f(k) \cdot |G|^c$ that, on input of a graph $G = (V, E)$, a vertex $s \in V$, and $k \in \mathbb{N}$, either detects that the longest path in G exceeds 2^k , or outputs a depth-first search labeling starting at s .

Proof. In order to achieve a depth-first search labeling in parallel, we will start a classical depth-first search at s . However, we have not enough time to “backtrack” and, instead, we have to identify all branches of the depth-first search starting at some vertex v in the moment in which we explore v for the first time. These branches can be found by computing the connected components of the unexplored graph via a breadth-first search and Lemma 34.

In detail, we test whether the longest path in G is bounded by 2^k using Lemma 34. If this is not the case, we immediately reject. Secondly, we check whether G is connected (again, using Lemma 34) and, if not, reduce G to the connected component that contains s . Afterwards, the following algorithm, which we call a *phase*, is executed with color $c = 0$ as argument. Each phase does nothing if all vertices are colored, and this is the end of the recursion. If $c = 0$, vertex s is colored with c , otherwise an arbitrary vertex v that is not colored, but that has a neighbor w of color $c - 1$, is selected and colored with c . We set $\lambda(v) = c$ and mark w as the *predecessor* of v . At the end of a phase the vertices of G are partitioned into the colored vertices C and the uncolored vertices $V \setminus C$. The circuit computes the connected components of $G[V \setminus C]$ (using Lemma 34), which we denote by $V_1, \dots, V_\ell \subseteq V \setminus C$. Afterwards, new phases are started recursively and in parallel on each graph $G[V_i \cup C]$ with argument $c + 1$. When all phases have been terminated, the labeling λ is a depth-first search labeling starting at s . This fact is witnessed by the depth-first search tree $T = (V, \{(v, w) \mid v \text{ is a predecessor of } w\})$. The following figure illustrates a run of the algorithm.



Since this algorithm never performs backtracking, the number of consecutive phases is bounded by the length of the longest path, which is bounded by 2^k . For each phase, a circuit of depth $f(k)$ is sufficient, since the most expensive part is clearly the computation of the connected components. Thus, a depth-first search labeling can be computed by an AC-circuit of depth $f(k)$. \square

A direct consequence of Lemma 34 is the following theorem, where DISTANCE asks whether there is a path of length at most d between two given vertices s and t :

► Theorem 36

p_d -DISTANCE \in para-AC⁰

Proof. Compute all vertices of distance at most d from s using Lemma 34 and check whether t is one of them. □

This theorem will serve as a crucial building block in the design of many para-AC⁰-algorithms. It also reveals the fact that para-AC⁰ is *unconditionally* a proper superset of para-AC⁰, since it is known that for $d \leq \log n$ there is a constant c such that any AC-circuit of depth δ that decides whether a given graph contains an s - t -path of length d requires size at least $n^{c \cdot k^\epsilon}$ for $\epsilon = \phi^{-2\delta}/3$, where ϕ is the golden ratio [25].

► Corollary 37

para-AC⁰ \subsetneq para-AC⁰ ◁

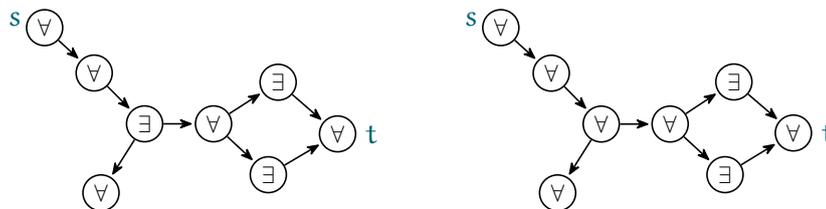
We may also note that para-AC⁰ can solve a notably more general version of the distance problem: Given a directed graph $G = (V, E)$ and a partition $V = V_\exists \cup V_\forall$, an *alternating path* from s to t is a set S of paths in G , all of which end at t , such that (i) exactly one of them starts at s ; (ii) when a path in S starts at some $v \in V_\exists \setminus \{t\}$, then for some w with $(v, w) \in E$ there is a path in S starting at w ; and (iii) when a path in S starts at some $v \in V_\forall \setminus \{t\}$, then for all w with $(v, w) \in E$ there is a path in S starting at w (and there is at least one such w). The *length* of an alternating path is the maximum length of any path in the set S . The *alternating distance* between two vertices is the minimum distance of any alternating path between them.

► Problem 38 (ADISTANCE)

Instance: A directed graph $G = (V, E)$, a partition $V = V_\exists \cup V_\forall$, two vertices $s, t \in V$, a distance d .

Question: Is the alternating distance from s to t in G at most d ? ◁

Two example instances of the problem are illustrated in the following figure. In the left graph there is an alternating path from s to t of length 5, while in the right graph there is no such path. This problem is a classical P-complete problem [110] and, thus, there is no parallel algorithm that solves it unless NC = P. Parameterized by the distance d , however, the problem lies in para-AC⁰. In fact, it is a natural representative for this class in the sense that it is also hard for it.

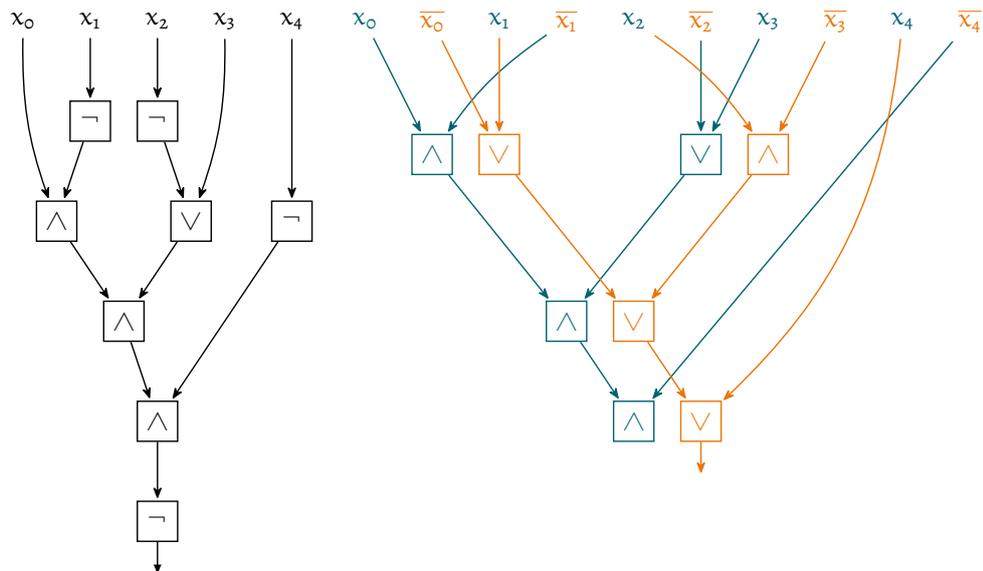


► Theorem 39

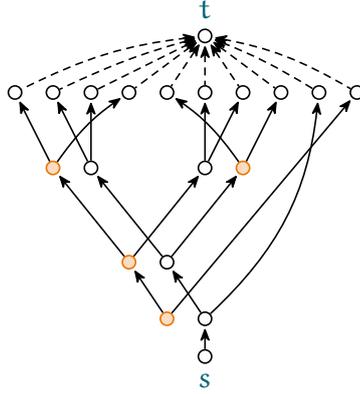
p_d -ADISTANCE is complete for para-AC^{O1} under para-AC^O -reductions.

Proof. For containment consider a circuit that performs a backward breadth-first search starting at t , similar to Lemma 34. The circuit processes the graph in d layers, computing in layer i the vertices that have alternating distance i to t . In the first layer, vertex t is colored. In layer i , all vertices $x \in V_{\exists}$ that have one colored neighbor, and all $y \in V_{\forall}$ that have only colored neighbors (and at least one) are colored. There is an alternating path of distance at most d from s to t if, and only if, s is colored after d layers. The correctness of the circuit follows by a simple induction: in layer 1 we color exactly the vertices with alternating distance 1, and it can easily be seen that coloring a vertex in layer i is only possible if it has a neighbor (or all its neighbors) with alternating distance $i - 1$.

For completeness let us reduce any problem $(L, \kappa) \in \text{para-AC}^{O1}$ to p_d -ADISTANCE. As (L, κ) is in para-AC^{O1} , there is a fixed family of circuits deciding L . Let C be such a circuit. We may assume that C is monotone since we can always replace a non-monotone circuit by a monotone one (using the standard argument used for showing that the circuit value problem reduces to its monotone version [101]): The idea is to use “double-railed” logic that computes the negation of any gate “on the fly.” This technique is illustrated in the following graphic, where the circuit on the left is the non-monotone input circuit. The circuit on the right uses double-railed logic to simulate negation without using negation gates – here the blue wires and gates are the original ones (or the “positive” ones), while the orange wires and gates are the “negated” ones. Note that the monotone circuit has twice as many inputs, as it expects the negation of the original input bits as additional input.



We translate the monotone circuit C into an alternating graph as follows: The vertices of the graph are the gates, and the wires are edges directed from the unique output gate towards the input bits. For each input bit there is a vertex as well. We label the output gate as s , add a new vertex t , and we add edges from all input bits that are set to 1 towards t . We then partition the vertices such that V_{\exists} is the set of OR-gates joined by s and t , and the input bits; and such that V_{\forall} is the set of AND-gates. In the following figure the construction for the aforementioned circuit is illustrated. The vertices of V_{\exists} are uncolored while the vertices in V_{\forall} are colored orange. The dotted edges only exist if the corresponding input-gate is set to 1.



The constructed graph with s and t , and with d as distance, constitutes an instance of p_d -ADISTANCE. An alternating path from s to t corresponds to wires that are set to true during the evaluation of the circuit and, hence, such a path can only exist if the circuit evaluates to true. Since, furthermore, the depth of the circuit is bounded by d , such a path has length at most d as well.

We are left with the task of arguing that the described reduction can be performed by a *uniform* family of para- AC^0 -circuits. Keep in mind that we reduce from a problem $(L, \kappa) \in \text{para-}AC^0$ with, say, $L \subseteq \text{STRUC}[\tau]$ and $\kappa: \text{STRUC}[\tau] \rightarrow \mathbb{I}$. The circuit-family $(R_{n,k})_{n \in \mathbb{N}, k \in \mathbb{I}}$ that we construct obtains some structure $S \in \text{STRUC}[\tau]$ as input and shall output an instance for p_d -DISTANCE, that is, $(S, \kappa) \in L$ if, and only if, $R_{|\text{code}(S)|, \kappa(S)}(S) = \text{code}(A)$ with $(A, \kappa') \in p_d$ -DISTANCE, where κ' maps to the value d . Further recall that $(L, \kappa) \in \text{para-}AC^0$ is witnessed by a uniform family $(C_{n,k})_{n \in \mathbb{N}, k \in \mathbb{I}}$. Observe that it is easy to construct a uniform family of para- AC^0 -circuits that, given the structure S and $\text{code}(C_{|\text{code}(S)|, \kappa(S)})$, outputs the result of the reduction, since the transformations used by the reduction can be expressed by simple first-order interpretations. However, the circuit $R_{|\text{code}(S)|, \kappa(S)}$ does not obtain $\text{code}(C_{|\text{code}(S)|, \kappa(S)})$ as input. Instead, we hard-wire $\text{code}(C_{|\text{code}(S)|, \kappa(S)})$ into a single AC-layer of $R_{|\text{code}(S)|, \kappa(S)}$. This can be done by the Turing machine M that constructs $R_{|\text{code}(S)|, \kappa(S)}$, since $C_{|\text{code}(S)|, \kappa(S)}$ is itself uniform and, thus, M can simulate the Turing machine that is used to construct $C_{|\text{code}(S)|, \kappa(S)}$. \square

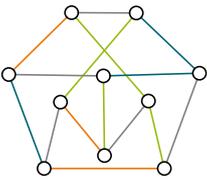
4.3 COLOR CODING

The color coding technique due to Alon, Yuster, and Zwick [6] is actually an advanced strategy and not necessarily “basic.” However, it is *surprisingly well* suited for parameterized parallel constant time computations. In fact, it is the heart of almost all constant time algorithms presented in this thesis and, thus, I believe it is fair to classify it as a basic parameterized parallel algorithm. The technique is best understood by applying it to a concrete problem:

► Problem 40 (RAINBOW-MATCHING)

Instance: An edge-colored graph $G = (V, E, \chi)$ with $\chi: E \rightarrow \{1, \dots, k\}$.

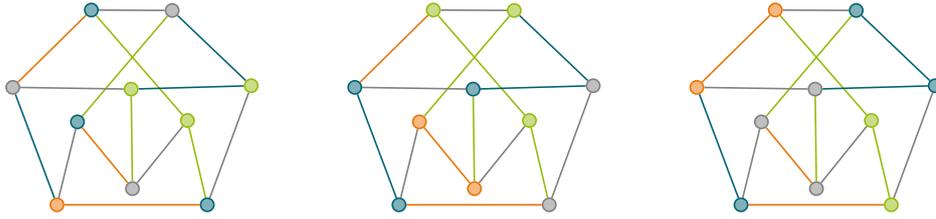
Question: Is there a matching $M \subseteq E$ with $|M| = k$ that contains an edge of every color, that is, all edges in M have distinct colors? ◀



An example instance is shown in the margin, the reader is asked to find a solution for it. This problem is a generalization of the classical matching problem and has interesting applications, for instance in the theory of Latin squares [129]. It is known that RAINBOW-MATCHING is NP-complete even restricted to bipartite graphs [112], and it is APX-complete even on properly edge-colored paths [129].

To understand the idea of the color coding technique let us assume that we have a coin, and let us further assume we flip that coin for every vertex in the graph. The crucial observation that Alon, Yuster, and Zwick had is that whenever we search for a small structure in the graph (here the rainbow matching of size k), the probability that all vertices that participate in this structure obtain “head” is bounded by some function in k (and, especially, is independent of the size of the graph). In fact, if we replace the coin by a die, or for that matter by a random coloring of the vertices of the graph, the probability that the vertices participating in the structure we seek obtain a certain coloring is still bounded solely by a function in k .

This observation can easily be turned into an efficient randomized FPT-algorithm for p_k -RAINBOW-MATCHING: On input of $G = (V, E, \chi)$ with $\chi: E \rightarrow \{1, \dots, k\}$ we “roll” a random coloring $\lambda: V \rightarrow \{1, \dots, k\}$. We say an edge $e = \{v, w\} \in E$ is *compatible with χ* if $\lambda(v) = \lambda(w) = \chi(e)$. Observe that we can test in polynomial time if there is a rainbow matching that contains only compatible edges – for each color we simply search for a compatible edge, the matching property then is guaranteed by the coloring. Furthermore, if G actually contains a real rainbow matching, the probability that all edges in it become compatible with λ is bounded by a function in k and, thus, can be arbitrarily increased by repeating the algorithm $f(k)$ times for some computable function $f: \mathbb{N} \rightarrow \mathbb{N}$. The following graphic on the next page shows three “random” colorings of the instance from above, the reader may identify the one with a compatible solution:



The final ingredient we need to turn the above algorithm into an FPT-algorithm, or actually into a parameterized parallel constant time algorithm, is a way to derandomize color coding. This can be done with *universal coloring families*:

► Definition 41 (Universal Coloring Families)

For natural numbers n , k , and c , an (n, k, c) -universal coloring family is a set Λ of functions $\lambda: \{1, \dots, n\} \rightarrow \{1, \dots, c\}$ such that for every subset $S \subseteq \{1, \dots, n\}$ of size $|S| = k$ and for every mapping $\mu: S \rightarrow \{1, \dots, c\}$ there is at least one function $\lambda \in \Lambda$ with $\forall s \in S: \mu(s) = \lambda(s)$. ◁

It is well known that such families (of a suitable size) can be generated efficiently using hash functions [85]. This in turn allows the use of color coding to design deterministic FPT-algorithms. The following theorem shows that we can compute such families in para-FAC^o:

► Theorem 42

There is a computable function f and a uniform family $(C_{n,k,c})_{n,k,c \in \mathbb{N}}$ of FAC-circuits without inputs such that each $C_{n,k,c}$

1. outputs an (n, k, c) -universal coloring family (coded as a sequence of function tables),
2. has constant depth (independent of n , k , or c), and
3. has size at most $f(k, c) \cdot \text{poly}(n)$.

Proof. Let us first assume that n is sufficiently larger than k , in particular such that $f(k) < n$, and define

$$\begin{aligned} \lambda_{p,a}(x) &= (a \cdot x \bmod p) \bmod k^2 \\ \Lambda'_{n,k} &= \{ \lambda_{p,a} \mid p < k^2 \log n \text{ and } a \in \{0, \dots, p-1\} \}, \\ \Lambda_{n,k,c} &= \{ \omega \circ \lambda \mid \omega: \{0, \dots, k^2-1\} \rightarrow \{1, \dots, c\} \text{ and } \lambda \in \Lambda'_{n,k} \}. \end{aligned}$$

It is well known that $\Lambda'_{n,k}$ is a family of k -perfect hash functions, that is, for every subset $S \subseteq \{1, \dots, n\}$ with $|S| = k$ it contains a function that is injective on S , see [85]. Therefore, given a subset S and a function $\mu: S \rightarrow \{1, \dots, c\}$, some members of $\lambda_{p,a} \in \Lambda'_{n,k}$ will map the members of S injectively to a subset S' of $\{0, \dots, k^2-1\}$

and, then, some function $\omega: \{0, \dots, k^2 - 1\} \rightarrow \{1, \dots, c\}$ will map S' in such a way that $\omega \circ \lambda_{p,a}$ equals μ on S . Consequently, the set $\Lambda_{n,k,c}$ is an (n, k, c) -universal coloring family. The sizes of the two sets can be bounded by $|\Lambda'_{n,k}| \leq (k^2 \cdot \log n)^2$ and $|\Lambda_{n,k,c}| \leq c^{k^2} \cdot (k^2 \cdot \log n)^2 = c^{k^2} k^4 \log^2 n$. Each function in $\Lambda_{n,k,c}$ can clearly be encoded in $n \log_2 c$ bits.

For the construction of the circuit $C_{n,k,c}$ observe that the input length is n and that all numbers of the above definitions are smaller than n . Therefore, we essentially work with *unarily encoded* numbers. For them, addition, multiplication, as well as the modulo operation are in uniform FAC° by Lemma 22. In conclusion, there is a uniform family of FAC° -circuits $(H_n)_{n \in \mathbb{N}}$ that obtains as input three unary numbers p, a, x , and outputs $\lambda_{p,a}(x)$. The circuit $C_{n,k,c}$ consists of $n \cdot (k^2 \log n)^2$ copies of H_n , where all combinations of $0 \leq x \leq n$ and $0 \leq a < p \leq k^2 \log n$ are hard-wired to the different copies of H_n . The concatenated output of these subcircuits almost equals the function table of $\Lambda_{n,k,c}$. The only part missing is the mapping ω , which is applied by $C_{n,k,c}$ in a constant number of additional AC-layers.

Observe that (i) the depth of the circuit $C_{n,k,c}$ is constant as the depth of H_n is constant, and (ii) the size of $C_{n,k,c}$ is bounded by $O(n \cdot (k^2 \log n)^2 \cdot |H_n| \cdot c^{k^2} \log c)$. To see that the family is uniform, just observe that $(H_n)_{n \in \mathbb{N}}$ is uniform and that a Turing machine on input $\text{bin}(i) \# \text{code}(k, c) \# \text{bin}(n)$ can compute $k^2 \log n$ in time $f(k) + \log n$ as $\log n$ is a $\log \log n$ -bit number: Either $k^2 < \log n$ and the multiplication can be performed “in the $\log n$ part,” or $k^2 > \log n$ and the multiplication can be performed in time $f(k)$.

For the remaining case that k is too large, we have $f(k) > n$. Therefore, we may hard-wire *any* family of k -perfect hash functions (whose size may arbitrarily depend on n and k) directly into the circuit. Since the uniformity Turing machine is allowed to run for $f(k)$ steps for some computable function f , this hard-wired version is clearly uniform as well. \square

The theorem has interesting consequences. For instance, it should be immediately clear that p_k -RAINBOW-MATCHING lies in para-AC° and, thus, also the classical matching problem lies in this class. Note that, in contrast, the parallel complexity of the matching problem is still not fully resolved. It is only known that the matching problem can be solved in randomized NC [132] and quasi NC [79] (which is defined as NC, but the circuits are allowed to have size $O(n^{\log^i n})$). Only very recently, these results were improved by Anari and Vazirani to an algorithm that runs in pseudo-deterministic randomized NC [8].

► Corollary 43

p_k -RAINBOW-MATCHING $\in \text{para-AC}^\circ$

◁

Another consequence that we will heavily use, and in fact already have used in the proof of Theorem 33, is the observation that we can “count” with the help of color coding. More precisely we can solve the following problem:

► Problem 44 (THRESHOLD)

Instance: A bitstring $b \in \{0, 1\}^n$ and a number $t \in \mathbb{N}$.

Question: Are there at least t many 1’s in b ? ◁

Clearly, the unparameterized version is complete for TC° , but parameterized by t we obtain the following result.

► Lemma 45

$p_t\text{-THRESHOLD} \in \text{para-AC}^\circ$

Proof. On input of a bitstring b of length n and $t \in \mathbb{N}$, we use Theorem 42 to compute an (n, t, t) -universal coloring family. If b contains at least t many 1’s, then there is a coloring of the positions of b such that each color class contains at least one 1. Thus, it is sufficient to test in parallel for all colorings whether this is the case. □

It should be noticed that this was already known by a result from circuit complexity [136], as AC° can solve the problem for polylogarithmic values of t . In fact, the techniques used to prove this result are similar to the techniques we have used to prove Theorem 42: The input is hashed to a small domain using suitable hash functions and, then, the problem is solved via “brute-force.” Therefore, Theorem 42 can be seen as a generalization of the results from [136] by an extension of color coding. In return, this allows to prove Lemma 45 in just a few lines.

Another useful application of color coding is an extension of our result for independent sets in graphs of bounded degree. If the problem is additionally parameterized by the solution size k , we can actually find an *optimal* solution in parallel constant time. The attentive reader may observe in the following proof that a similar “stamp argument” will work for many other graph problems on graphs of bounded degree. We will formalize this idea in Chapter 8 by adapting a result of Flum and Grohe [84] to the parallel parameterized setting: All problems definable in first-order logic can be solved in $\text{para-AC}^{\text{ol}}$ on structures of bounded degree.

► Lemma 46

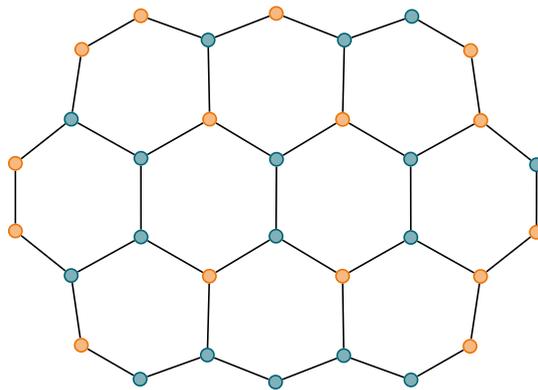
$p_{k,\Delta}\text{-INDEPENDENT-SET} \in \text{para-AC}^\circ$

Proof. We will directly “stamp” the independent set into the graph. To make this idea work, we need the property that the size of the border of the structure we search (which is bounded in size by the parameter) is bounded by the parameter as well. In a graph of bounded degree this is obviously the case, as any of the k vertices in the structure adds at most Δ vertices to the border.

Since *both*, the structure we search and its border, are small, we may use color coding to color them both in any way we want. In the “stamp technique” we use two colors (say blue and orange) and we hope for a coloring in which the structure we search (here the independent set) becomes colored orange, while its border becomes colored blue. We now just have to search for k orange vertices that have a blue neighborhood and induce the structure we are looking for. In the case of the independent set problem, we search for k orange vertices that all have only blue neighbors.

Testing if a given vertex is colored orange and has only blue neighbors can easily be implemented in a constant number of AC-layers. The coloring can be obtained in constant depth by Theorem 42 and the correctness follows by the properties of a universal coloring family. \square

The following figure illustrates the stamp technique used in the proof of Lemma 46. The vertices are colored with blue and orange, and an independent set of size 6 was successfully stamped such that it is orange and its neighborhood is blue – can the reader spot it?



5 PARALLEL BOUNDED SEARCH TREES

In this chapter we study parallel parameterized algorithms based on the bounded search tree technique. I start by presenting a short review of the basic terminology of bounded search trees in Section 5.1. Afterwards, we will study a generic problem: The modulator problem for graphs (given a graph $G = (V, E)$ and a number $k \in \mathbb{N}$, can we delete k vertices such that G belongs to some family \mathcal{F} of graphs). This will allow us to solve many natural problems with parallel bounded search trees. The first result, presented in Section 5.2, handles this problem for the case that \mathcal{F} is the family of H -free graphs.

▷ Informal Version of Theorem 51.

For every fixed graph H , there is a family of para-FAC⁰-circuits that decides, given a graph G and a number k , whether we can delete k vertices such that G is H -free. ◁

In the remainder of Section 5.2 we will devote ourselves to generalize this result to more complex families \mathcal{F} . We will study the family that forbids a homomorphic copy of H (there is no homomorphism from H to a member of \mathcal{F}) and the family of graphs that does not contain a copy of H as embedding. Our goal is to develop an algorithm that can handle both cases, even if the graph H is not fixed (but a parameter).

▷ Informal Version of Corollary 57 and Corollary 63.

Let \mathcal{H} be a family of graphs with constant treewidth. There is a family of para-FAC⁰-circuits that decides, given graphs $H \in \mathcal{H}$ and G , whether we can delete k vertices from G such that there is no homomorphism (embedding) from H to G . ◁

Finally, we close the chapter by studying the feedback-vertex set problem in Section 5.3. This problem does not fit into the framework that we develop in Section 5.2 and, thus, we have to design a new parallel algorithm. We will adapt a classical sequential search tree in order to obtain such a parallel algorithm. However, we will see that this is not trivial, since the sequential search tree applies inherently sequential reduction rules repeatedly.

▷ Informal Version of Theorem 68.

There is a family of para-FAC¹-circuits that, given a graph G and a number k , outputs a feedback-vertex set of size k of G – if such a set exists. ◁

5.1 A SHORT REVIEW OF BOUNDED SEARCH TREES

The bounded search tree method was one of the first tools to obtain fixed-parameter algorithms [69]. Fortunately, it is conceptually one of the easiest methods, and it is well suited for parallelization in a natural way. Intuitively, we will build a search tree with a depth that is bounded (usually linearly) by the parameter. If the branch-number of the tree is also bounded, the size of the whole tree is bounded by a function in the parameter. A parallel algorithm can handle a whole level in a single step and, thus, requires only time depending on the depth of this tree and the time needed to identify the children of a node within the tree. This concept is best understood with a concrete example – for our purposes we will use the independent set problem on planar graphs, that is, we wish to know if a given planar graph contains k vertices that are pairwise not adjacent.

► **Theorem 47**

There is a uniform family of FAC-circuits of depth $f(k)$ and size $f(k) \cdot |V|^c$ that, on input of a graph $G = (V, E)$ and a number $k \in \mathbb{N}$, either outputs an independent set of size k , or correctly detects that such a set does not exist, or correctly detects that G is not planar.

Proof. The proof is based on the proof in [85] for showing that the problem is in FPT. We first observe that for a vertex $v \in V$ at least one vertex of $N[v]$ will be part of any maximal independent set (if no vertex of $N(v)$ can be added, we can add v). Next we use the fact that a planar graph contains a vertex v of degree at most 5, that is, a vertex with $|N[v]| \leq 6$. This follows directly from the fact that planar graphs contain at most $3|V| - 6$ edges.

The circuit works as follows: First it computes the degree of every vertex using the circuit from Lemma 45. If all vertices have degree greater than 5 the circuit safely reports the input graph is not planar. Otherwise, the circuit uses the lexicographical smallest vertex of degree at most 5 and branches over $N[v]$, that is, for every vertex $w \in N[v]$ a subcircuit is used to check if $G[V \setminus N[w]]$ contains an independent set of size at most $k - 1$.

If any of these branches reaches the value $k = 0$, the circuit has found the desired independent set and presents it as output. If a branch creates the empty graph while k is still greater than 0, this branch rejects. If all branches reject, the circuit safely reports that the graph does not contain an independent set of size k .

The claimed depth of the circuit follows directly from the fact that each branch has length at most k . The size of the circuit is bounded by the size of the traversed search tree, that is, by $6^k \cdot \text{poly}(n)$. □

This result implies that p_k -PLANAR-INDEPENDENT-SET lies in para-AC^0 if we promise that the input graph is planar. If this promise is not given, then the circuit needs to check if the graph is actually planar (in the decision version the circuit may only accept if the input contains a size- k independent set *and* is planar). The smallest circuit class for which it is known that it can perform a planarity test is AC^1 [4] and, hence, we have p_k -PLANAR-INDEPENDENT-SET $\in \text{para-AC}^1$.

5.2 MODULATORS AND EDITING

In this section we seek to establish a general result about the parallel evaluation of search trees. For that matter, we will study *modulator* and *editing* problems. Informally, we are given a host graph G and are asked if we can transform it into a graph of some family \mathcal{F} of graphs by just a few modifications. We either want a modulator, that is, a set of vertices whose removal will transform G into a graph contained in \mathcal{F} , or we may edit G by adding or removing edges from it. Formally, we consider the following problems for a fixed family \mathcal{F} of graphs:

► **Problem 48 (MODULATOR(\mathcal{F}))**

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: Is there a set $X \subseteq V$ with $|X| \leq k$ such that $G[V \setminus X] \in \mathcal{F}$? ◁

► **Problem 49 (EDITING(\mathcal{F}))**

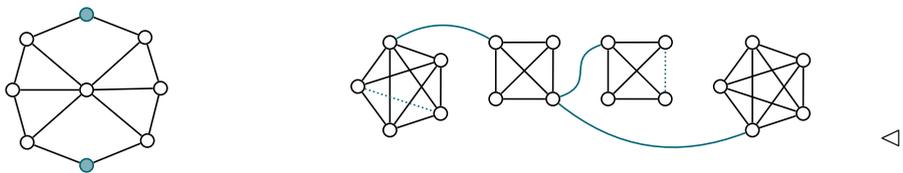
Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: Are there sets $R \subseteq E$ and $A \subseteq \bar{E}$ with $|R \cup A| \leq k$ such that we have $G' = (V, (E \setminus R) \cup A) \in \mathcal{F}$? ◁

The simplest, but still quite powerful, version of this problem is the one for \mathcal{F} being the family of H -free graphs for some fixed graph H . In detail, we define for some fixed graph H the family $\mathcal{F} = \{G \mid G \text{ does not contain } H \text{ as induced subgraph}\}$. Natural use-cases are for instance the *vertex cover* problem ($H = \text{---}$); *cluster editing* ($H = \text{---}$); and *distance to a co-graph* ($H = \text{---}$).

► **Example 50**

In the left figure, there is a modulator of size 2 to --- -free graphs, while in the right graph we can edit 5 adjacencies to obtain a --- -free graph.



► Theorem 51

For every fixed graph H and the family \mathcal{F} of H -free graphs we have:

1. p_k -MODULATOR(\mathcal{F}) \in para-AC^o;
2. p_k -EDITING(\mathcal{F}) \in para-AC^o.

Proof. We prove the first item. Since H is fixed, an AC-circuit of constant depth and size roughly $|V(G)|^{|V(H)|}$ can check, given the input graph $G = (V, E)$, whether there is a set $O \subseteq V$ such that $G[O]$ is isomorphic to H . If no such O exists, we have $G \in \mathcal{F}$ and are done. Otherwise, O is an *obstruction* to be H -free and at least one vertex of O must be added to the solution. The circuit in construction branches over all possibilities (this is just a constant number) and repeats the whole procedure. After k layers of such AC-circuits, we may either have found the sought modulator, or may have correctly decided that there is no such modulator of size k . Hence, the total depth of the circuit is $f(k)$ and its size is $f(k) \cdot |V|^c$ for some computable function f and constant c .

The editing case works equivalently, the only difference is that we do not branch over vertices of O , but over edges and non-edges in $G[O]$. \square

An alternative approach to Theorem 51 is a reduction to the hitting set problem with small hyperedges. We will see in Section 6.5 that we can flatten the search tree to constant depth (using *a lot more* machinery), implying p_k -MODULATOR(\mathcal{F}) \in para-AC^o. This reduction, however, does *not* work for p_k -EDITING(\mathcal{F}). The main obstacle here is that in the editing problem it is not sufficient to “hit” all obstructions, since adding or deleting an edge can create new obstructions. It remains open whether or not p_k -EDITING(\mathcal{F}) can be placed in para-AC^o; however, Stockhusen, Tantau, and myself showed that some special cases (such as editing to cluster graphs) are in para-AC^o [19].

We can naturally extend the result by studying more complex families \mathcal{F} . For instance, we may study $\mathcal{F} = \{G \mid H \not\rightarrow G\}$ for some fixed graph H , where $H \not\rightarrow G$ denotes the fact that there is no homomorphism from H to G . Actually, we can even handle the more general case that H is not fixed, but part of the input. This leads to the following problem:

► Problem 52 (HOM-MODULATOR(\mathcal{H}))

Instance: Two graphs $H = (V(H), E^H) \in \mathcal{H}$ and $G = (V(G), E^G)$, and a number $k \in \mathbb{N}$.

Question: Is there a set $X \subseteq V(G)$ with $|X| \leq k$ such that $H \not\rightarrow G[V \setminus X]$? \triangleleft

It should be clear that this problem is, for arbitrary families \mathcal{H} , more complex than the previous problems, as we have to find a homomorphism from H to G before we can think about the modulator. In other words, we now have to deal with the following problem:

► Problem 53 (HOMOMORPHISM(\mathcal{H}))

Instance: Two graphs $H = (V(H), E^H) \in \mathcal{H}$ and $G = (V(G), E^G)$.

Question: $H \rightarrow G$? ◁

Naturally, the parameter here is H . If H is the complete graph on k vertices and G has no self-loop, this task is exactly the parameterized clique problem and, thus, $W[1]$ -complete [59]. As a consequence, we have to prohibit complete graphs in the family \mathcal{H} if we wish to obtain efficient parallel algorithms. In particular, \mathcal{H} may not be the set of all graphs and it will not be sufficient to bound the cliquewidth of the graphs in it. Instead, we will focus on families \mathcal{H} that contain graphs of bounded treewidth or treedepth. It is known by results of Chen and Müller [50] that the problem (i) lies in para-L when \mathcal{H} has bounded treedepth; (ii) lies in the para-L-reduction closure of the distance problem (parameterized by the distance) if \mathcal{H} has bounded pathwidth; and (iii) lies in para-L-reduction closure of the embedding of trees if \mathcal{H} has bounded treewidth. The problem has also been studied with respect to classical circuit complexity – here Amano showed that the unparameterized problem, in which the graphs in \mathcal{H} are of constant size, lies in AC^0 [7]. We extend these results with parameterized parallel algorithms and in particular we improve the first result of Chen and Müller considerably by showing that p_H -HOMOMORPHISM(\mathcal{H}) actually lies in para- AC^0 if \mathcal{H} has bounded treedepth.

► Theorem 54

Fix two numbers $w, d \in \mathbb{N}$ with $w < \infty$ (but with $d = \infty$ being explicitly allowed) and consider $\mathbb{I} = \{H = (V(H), E^H) \mid \text{there is a tree decomposition } (T, \iota) \text{ of } H \text{ that has width at most } w \text{ and that can be rooted such that } T \text{ has depth at most } d\}$. Furthermore, define $d(H) = \min(d, |V(H)|)$ and let $c \in \mathbb{N}$ be a fixed constant. There is a uniform family $(C_{n,H})_{n \in \mathbb{N}, H \in \mathbb{I}}$ of para-FAC-circuits such that for all pairs $(G = (V(G), E^G), H = (V(H), E^H))$ of graphs with $H \in \mathbb{I}$ we have:

1. $C_{|\text{code}(G,H)|,H}(\text{code}(G,H))$ outputs a homomorphism from H to G encoded as function table, if such a homomorphism exists;
2. $\text{depth}(C_{n,H}) \leq c \cdot d(H)$;
3. $\text{size}(C_{n,H}) \leq d(H) \cdot |V(H)|^c \cdot n^{c \cdot w}$.

Proof. First observe that the parameter is easily computable in FAC^0 as we just have to extract H from a given pair (G, H) . The circuit $C_{n,H}$ will apply dynamic programming over a tree decomposition (T, ι) of H (of width at most w and depth at most $d(H)$). This tree decomposition is hard-wired into the circuit. In order to see that this does not conflict the uniformity, recall that we require a Turing machine that, on input of $\text{bin}(i) \# \text{code}(H) \# \text{bin}(n)$ outputs the i th bit of $\text{code}(C_{n,H})$ in at most $f(H) + \log(n)$ steps. Since f is an arbitrary computable function, this machine has enough time to find a suitable tree decomposition and hard-wire it into the circuit – note that the existence of such a tree decomposition is guaranteed by the choice of \mathbb{I} .

Let us now describe the dynamic programming procedure on (T, ι) . Initially, we consider for every leaf l of T all assignments $\varphi' : \iota(l) \rightarrow V(G)$. We can think of these assignments as colorings of H , where the “colors” are the vertices of G . We call such an assignment *good* if φ' is locally a homomorphism. Observe that there are at most $|V(G)|^{w+1}$ potentially good assignments per leaf and, thus, a circuit of the claimed size and *constant* depth can check whether they are good.

For the inductive step let us consider a node n of T . We consider again all possible assignments $\varphi' : \iota(n) \rightarrow V(G)$ and, this time, we call φ' good if:

1. $\varphi' : \iota(n) \rightarrow V(G)$ is locally a valid homomorphism;
2. for every child m of n in T there is a good assignment $\psi' : \iota(m) \rightarrow V(G)$ such that for every vertex $v \in \iota(n) \cap \iota(m)$ we have $\varphi'(v) = \psi'(v)$.

Clearly, this test can also be implemented by a constant number of AC-layers of size $O(|V(G)|^{w+1})$. Therefore, the overall depth of the circuit will be $c \cdot d(H)$, and its size is bounded by $d(H) \cdot |V(H)|^c \cdot |V(G)|^{c \cdot w}$.

We are left with the task of showing that there is a homomorphism φ from H to G if, and only if, there is a good assignment φ' for the root r of T . To see this, first observe that, if we have found a good assignment φ' for r , then there is a local homomorphism for the vertices in the root bag. Furthermore, by the second property of “being good,” we have found good assignments for every child, and these assignments coincide with φ' in the intersection of the bags. Since, in a tree decomposition, all bags that contain the same vertex form a connected subtree, we can extend φ' along the children of r while ensuring that a fixed vertex $x \in V(H)$ gets mapped to the same vertex $\varphi(x) \in V(G)$ by good assignments in all branches. In other words, we can extend the partial homomorphism φ' to a homomorphism by recursively unite it with good assignments of its children. For the other direction assume that there is a homomorphism $\psi : V(H) \rightarrow V(G)$. Then it is easy to see that for every node n of T the assignment $\varphi' : \iota(n) \rightarrow V(G)$ with $\varphi'(v) = \psi(v)$ for all $v \in \iota(n)$ is good. Therefore, if there is a homomorphism, the algorithm will actually find it. \square

► Corollary 55

Let \mathcal{H} be the class of all graphs of treewidth at most t for some constant t . Then $\text{P}_H\text{-HOMOMORPHISM}(\mathcal{H}) \in \text{para-AC}^0$ \triangleleft

Proof. Set $w = t$ and $d = \infty$ in Theorem 54. \square

► Corollary 56

Let \mathcal{H} be the class of all graphs of treedepth at most t for some constant t . Then $\text{P}_H\text{-HOMOMORPHISM}(\mathcal{H}) \in \text{para-AC}^0$ \triangleleft

Proof. Set $w = t$ and $d = t$ in Theorem 54. \square

The reader may observe that the algorithm of Theorem 54 will also work for general relational structures if they have bounded treewidth (treedepth). The result can, of course, directly be applied to $\text{p}_H\text{-HOM-MODULATOR}(\mathcal{H})$. We proceed as in Theorem 51, but instead of finding the obstruction via “brute-force,” we simply apply Theorem 54 to find it.

► Corollary 57

Let \mathcal{H} be the class of all graphs of treewidth at most t for some constant t . Then $\text{p}_H\text{-HOM-MODULATOR}(\mathcal{H}) \in \text{para-AC}^{\text{ol}}$ ◁

The last version of the modulator problem that we will study in this section is for the family $\mathcal{F} = \{G \mid H \not\rightarrow G\}$. In words, the family of graphs that does not contain some graph H as embedding.

► Problem 58 ($\text{EMB-MODULATOR}(\mathcal{H})$)

Instance: Two graphs $H = (V(H), E^H) \in \mathcal{H}$ and $G = (V(G), E^G)$, and a number $k \in \mathbb{N}$.

Question: Is there a set $X \subseteq V(G)$ with $|X| \leq k$ such that $H \rightarrow G[V \setminus X]$? ◁

We will consider it again in the version in which H is part of the input and, thus, we will have to solve the following problem:

► Problem 59 ($\text{EMBEDDING}(\mathcal{H})$)

Instance: Two graphs $H = (V(H), E^H) \in \mathcal{H}$ and $G = (V(G), E^G)$.

Question: $H \rightarrow G$? ◁

We will adapt the algorithm from Theorem 54 to find embeddings instead of homomorphisms (recall that an embedding is an injective homomorphism). The idea is to assign to every vertex of H a unique color and to apply color coding with exactly these colors to G .

► Theorem 60

Fix two numbers $w, d \in \mathbb{N}$ with $w < \infty$ (but with $d = \infty$ being explicitly allowed) and consider $\mathbb{I} = \{H = (V(H), E^H) \mid \text{there is a tree decomposition } (T, \iota) \text{ of } H \text{ that has width at most } w \text{ and that can be rooted such that } T \text{ has depth at most } d\}$. Furthermore, define $d(H) = \min(d, |V(H)|)$ and let $f: \mathbb{I} \rightarrow \mathbb{N}$ be a computable function and $c \in \mathbb{N}$ be a fixed constant. There is a uniform family $(C_{n,H})_{n \in \mathbb{N}, H \in \mathbb{I}}$ of para-FAC-circuits such that for all pairs $(G = (V(G), E^G), H = (V(H), E^H))$ of graphs with $H \in \mathbb{I}$ we have:

1. $C_{|\text{code}(G,H)|, H}(\text{code}(G, H))$ outputs an embedding from H into G encoded as function table, if such an embedding exists;
2. $\text{depth}(C_{n,H}) \leq c \cdot d(H)$;
3. $\text{size}(C_{n,H}) \leq f(H) \cdot n^{c \cdot w}$.

Proof. We interpret the vertices $V(H)$ of H as colors, or equivalently assign a unique color to every vertex of H . Then we color G with a $(|V(G)|, |V(H)|, |V(H)|)$ -universal coloring family using Theorem 42. Finally, for every coloring λ of the universal coloring family we run the algorithm of Theorem 54, with the only modification that the partial assignments may only map vertices v of H to vertices w in G with $\lambda(w) = v$. Observe that a solution must be injective, as every vertex in H has its own color. Furthermore, if there exists an embedding φ from H to G , there will be a member in the universal coloring family that colors the image of φ with the correct colors. \square

► Corollary 61

Let \mathcal{H} be the class of all graphs of treewidth at most t for some constant t . Then $\text{P}_H\text{-EMBEDDING}(\mathcal{H}) \in \text{para-AC}^{\text{O}1}$ \triangleleft

► Corollary 62

Let \mathcal{H} be the class of all graphs of treedepth at most t for some constant t . Then $\text{P}_H\text{-EMBEDDING}(\mathcal{H}) \in \text{para-AC}^{\text{O}}$ \triangleleft

► Corollary 63

Let \mathcal{H} be the class of all graphs of treewidth at most t for some constant t . Then $\text{P}_H\text{-EMB-MODULATOR}(\mathcal{H}) \in \text{para-AC}^{\text{O}1}$ \triangleleft

Note that for a graph both, the treewidth and the treedepth, equal the maximum treewidth (treedepth) of its connected components. Therefore, building the disjoint union of graphs of bounded treewidth (treedepth) will in turn create a graph of bounded treewidth (treedepth). In this sense, Theorem 60 and its corollaries generalize to the *packing* version, in which we try to find k disjoint copies of H .

An application for Corollary 63 is the following generalization of VERTEX-COVER: Instead of seeking a small set of vertices that “hits” every edge (that is, every path of length 2), we now seek a set that hits every path of length c (for some $c \geq 2$).

► Problem 64 (PATH-VERTEX-COVER)

Instance: A graph $G = (V, E)$ and two numbers $k, c \in \mathbb{N}$.

Question: Is there a set $S \subseteq V$ with $|S| \leq k$ such that each path P of length c in G contains at least one vertex of S ? \triangleleft

This problem was first introduced by Bresar, Kardos, Katrenic, and Semanisin [45], and is applied in wireless sensor networks [138] and traffic control [162]. It is not surprising that it is NP-complete for every fixed $c \geq 2$, as the case $c = 2$ is obviously exactly VERTEX-COVER [45]. We will see in Section 6.5 that the problem lies in $\text{para-AC}^{\text{O}}$ for constant c by a simple reduction to $\text{P}_{k,d}\text{-HITTING-SET}$. This reduction does, unfortunately, not work if c is a parameter. However, we can still use Corollary 63 by setting $\mathcal{H} = \{P_c\}$:

► Corollary 65

$\text{P}_{k,c}\text{-PATH-VERTEX-COVER} \in \text{para-AC}^{\text{O}1}$ \triangleleft

Proof. Follows by the fact that paths have constant treewidth. \square

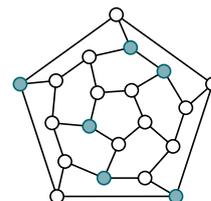
5.3 FEEDBACK-VERTEX SET

In the previous two sections the computation of the branches in the search tree was possible in constant parallel time, or at least in parallel time that is bounded only by the parameter. This is not necessarily always the case. In such scenarios we would like to find parallel algorithms that run in polylogarithmic time with respect to the instance size and detect the possible branches. If we are able to find a bounded search tree with this property, we can at least place the corresponding problem in para-AC^i for some $i \geq 1$. A problem with this property is for instance $\text{p}_k\text{-FEEDBACK-VERTEX-SET}$:

► **Problem 66 (FEEDBACK-VERTEX-SET)**

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: Is there a set $X \subseteq V$ with $|X| \leq k$ such that $G[V \setminus X]$ is a forest? ◁



An example instance with an optimal solution of $k = 6$ is shown in the margin. I will present an algorithm that runs in parallel time $O(k \cdot \log |V|)$, that is, we show $\text{p}_k\text{-FEEDBACK-VERTEX-SET} \in \text{para-AC}^1$. The algorithm is based on the following preprocessing rules that are applied in all k layers (each of which will consist of $\log n$ sublayers) of the circuit.

Leaf Rule Delete a vertex v of degree 1.

Chain Rule Contract a vertex v of degree 2 to one of its neighbors.

Loop Rule Delete a vertex v with $v \in N(v)$, reduce k by 1.

We first show that we can apply each of the above rules individually exhaustively in FAC^1 ; that is, in parallel time $O(\log n)$.

► **Lemma 67**

There is a uniform family of FAC^1 -circuits that, on input of a tuple (G, k) , outputs a tuple (G', k') that results from repeatedly applying (only) the Leaf Rule as long as possible. The same holds for the Chain Rule and for the Loop Rule.

Proof. The claim follows immediately for the Loop Rule as we may delete all such vertices in parallel and since the deletion of a vertex cannot create new vertices with a self-loop. For the other two rules observe that an “exhaustive application” equals either the deletion of attached trees (for the Leaf Rule), or the contraction of induced paths (for the Chain Rule). For the first case, the circuit must be able to detect if a vertex v becomes a leaf at some point of the computation (of course, the circuit cannot sequentially delete degree-1 vertices). The following observation provides a locally testable property that allows precisely such a detection: A vertex v is contained in an attached tree if, and only if, it is possible to delete a single edge such that (i) the graph

decomposes into two components and such that (ii) the component of v is a tree [76]. Both properties can be tested in logspace (and hence in AC^1), and an AC^1 -circuit can test them for all vertices and all edges in parallel. Finally, for the Chain Rule, observe that an AC^1 -circuit can mark all degree-2 vertices in parallel and that such a circuit, afterwards, only has to connect the two endpoints of highlighted paths – which is again a logspace task. \square

Using this circuit as blackbox, we will design a parallel bounded search tree algorithm that uses the preprocessed graph to quickly find branch-points.

► Theorem 68

p_k -FEEDBACK-VERTEX-SET \in para- AC^1

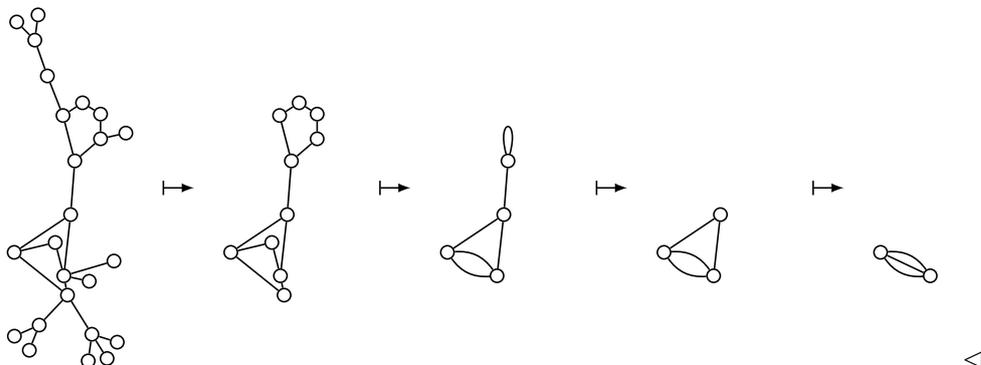
Proof. We have to construct a family of AC -circuits of depth $f(k) \cdot \log n$ and size $f(k) \cdot n^c$. The circuits will consist of k layers such that every layer finds a set of at most $3k$ vertices to branch on (which will be done for the next layer). Note that layer i contains at most $3k$ as many subcircuits as layer $i - 1$.

Each layer consists of multiple AC^1 -circuits that work independently of each other on different possible graphs (depending on the branches of the previous layer). Each of these circuits first checks if the input is a yes-instance (input is a tree and $k \geq 0$), or a no-instance ($k < 0$) – in the first case it just globally signals this circumstance. In the second case it truncates this path of the computation. If the subcircuit has not decided yet, it applies first the Leaf Rule exhaustively, and then the Chain Rule exhaustively – both are possible due to Lemma 67. The circuit now applies the Loop Rule (again, using Lemma 67). If the rule has an effect (that is, k was reduced by at least one) the circuit is done and pipes the result to the next layer. Otherwise, the circuit tests in parallel if there are two vertices v and u that are connected by a multi-edge (that is, by at least two edges). If this is the case, any feedback vertex set must contain either v or u and, hence, the circuit branches on these two vertices and pipes the two resulting graphs to the next layer. Otherwise, we know that we have no vertex with a self-loop, no vertices with multi-edges, and a minimum vertex-degree of at least three. The circuit then uses the simple fact that any size k feedback vertex set in such graph must contain at least one vertex of the $3k$ vertices with the highest degree and, hence, may simply branch over these [59].

Since each layer reduces k in each branch by at least one, after at most k layers every branch has decided if it deals with a yes- or a no-instance. Since each layer is implemented by an AC^1 -circuit, the claim follows. \square

► Example 69

An exemplary run of the algorithm is illustrated in the following figure:



The example run shows that we can eventually only apply one of the reduction rules after we have applied another: We can not directly apply the Chain Rule to the third graph, but only *after* we have applied the Loop Rule to it. Since the Loop Rule reduced k by one, we may hope to identify many such configurations in advance in order to speed up the algorithm. This would be interesting with respect to preprocessing and kernelization – a topic that we cover in the next chapter – as many standard preprocessing algorithms for FEEDBACK-VERTEX-SET apply all three rules exhaustively in advance [59]. Unfortunately, this seems not to be possible in parallel as the following theorem shows.

► Theorem 70

Deciding whether a specific vertex of a given graph will be removed by an exhaustive application of the Leaf Rule, the Chain Rule, and the Loop Rule (jointly in arbitrary order and not separately as in Lemma 67) is P-hard under NC^1 -reductions.

For clarity, let us stipulate that a self-loop contributes two to the degree of a vertex, similarly multi-edges increase the degree by their multiplicity. Therefore, the Chain Rule may not be applied to a leaf with a self-loop. We further stipulate that the Chain Rule may not be applied to a self-loop, that is, it has to contract two distinct vertices (and hence, self-loops may only be handled by the Loop Rule).

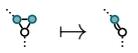
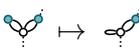
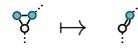
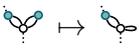
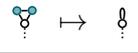
Before we work out the details, let me briefly sketch the proof idea: We will reduce from the monotone circuit value problem (MCVP), which is known to be P-complete under NC^1 -reductions [101]. The input to this problem is a monotone circuit (it consists only of AND-gates and OR-gates of indegree 2, and it has a single gate marked as output) and an assignment of the input gates, the question is whether or not the output gate evaluates to true. We will transform the input circuit into a multi-graph by replacing each gate with a small gadget. Every gadget will have two vertices marked as “input” and one marked as “output.” The “input” vertices are incident to exactly one edge outside of the gadget (which connects them to the “output” vertex of another gadget), the “output” vertex of the gadget may have edges to an arbitrary number of other “input” vertices. The semantic then is as follows: The edge of an “input”

vertex that leaves the gadget will be removed by the reduction rules when the corresponding wire of the circuit would have the value true for the given assignment; similarly the “output” vertex of the gadget will be removed if the corresponding gate would evaluate to true under the given assignment (this in turn removes the edges to other “input” vertices and propagates the computation of the circuit).

Proof. We start with a formal description of the transformation. For the input gates, we use $\mathfrak{1}$ and $\mathfrak{0}$ to describe assignments 1 and 0, respectively. Observe that the former can be removed by the Loop Rule, while the later is immune to all rules.

For AND-gates, we use the gadget , and for OR-gates . In these figures, the two highlighted vertices at the top are the ones we call “input,” while the bottom vertex is the “output” vertex. The dotted lines indicate edges that leave the gadget. For every “input” vertex there will be exactly one outgoing edge, as any gate has exactly two incoming wires. The “output” vertex may have edges to an arbitrary number of successor gates; to ensure that there is at least some edge, we fully connect such vertices to cliques of size three (that is, the “output” vertex is part of a clique of size four) – this ensures that the degree of “output” vertices is always greater than two.

We first prove that these gadgets work locally as intended. Observe that all vertices have a degree of at least three and no self-loop, that is, no rule can be applied unless one “incoming” edge gets removed. Since the “output” vertex of the gadget is fully connected to a clique of size three,

Assignment	Behaviour of the	
	AND-gadget	OR-gadget
(1, 0)		
(0, 1)		
(1, 1)		

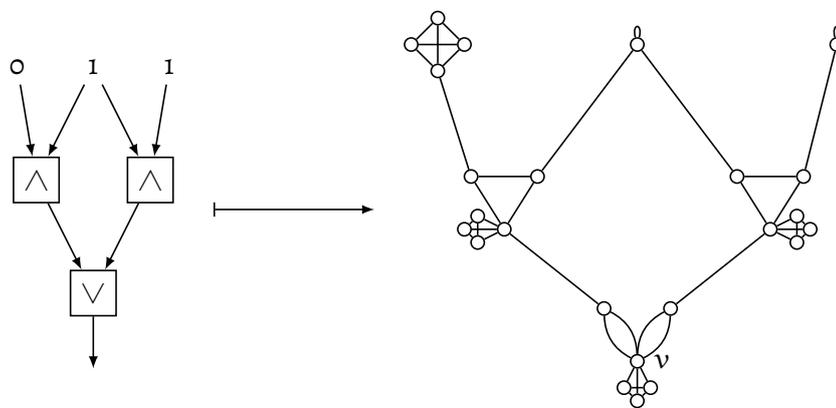
this can only happen if an edge connected to one of the “input” vertices gets removed. Therefore, we see directly that the gadget works as intended for the assignment (0, 0), as no edge connected to the “input” vertices gets removed and no rule can be applied to the “output” vertex. The case distinction illustrated in the table shows that the gadgets also work for the other assignments.

We now show the correctness of the construction by an induction over the gates of the circuit in topological order. The induction hypothesis is that the gadget corresponding to the current gate gets modified by the Leaf Rule, the Loop Rule and the Chain Rule in the same way as the gate gets evaluated. The base case is given as this is true for the input gates by construction. For the inductive step consider the gadget corresponding to any gate g , and let it have the vertices x , y , and z , where z is the “output” vertex. By the induction hypothesis the vertices x and y lose an incident edge for input wires that evaluate to 1 (as the gates corresponding to these gadgets precede g in the topological order), the above table then states that the gadget works correctly. The only pitfall we need to address is that the simulation does not “work backwards,” that is, that a reduction rule in g triggers a reduction rule for the “out-

put" vertex v of a gadget that corresponds to a gate that precedes g in the topological order. This is, however, not possible due to the clique attached to v – even if all edges that are incident to v get removed, v has still a degree of at least three. \square

► Example 71

The following figure illustrates the construction used in the proof. The circuit on the left evaluates to 1 if, and only if, the vertex labeled v in the right graph gets removed by a repeated application of all three reduction rules. Since the circuit clearly computes the value 1, the vertex v gets removed. However, if we replace the second input bit by 0, the circuit evaluates to 0, and we can see in the graph that the reduction rules do not propagate up to v .



◁

6 PARALLEL KERNELIZATION

Preprocessing is a fundamental technique used by practical tools that solve computational hard problems on large real world instances. It has a variety of applications in different domains such as (i) in modern SAT-solvers, which try to eliminate variables and clauses before the actual solving begins [73, 133]; (ii) as a tool to simplify ILP-instances [48]; and (iii) in the design of CSP-solvers, which try to optimize the instance for certain strategies like local search [62, 148]. Despite its impact in practice, preprocessing is rather hard to grasp from a theoretical point of view – at least in the sense of classical complexity. The reason is that even a polynomial time algorithm that just guarantees to reduce the input instance of an NP-hard problem by a single bit already implies $P = NP$ – as we can repeat the algorithm a linear number of times to obtain a trivially small instance that can be solved via exhaustive search.

With respect to preprocessing, the parameterized complexity theory shines, as we can use structural information about the instance to provide a reduction guarantee. In this chapter we will develop a variety of parallel algorithms which provide such guarantees. After a short review of kernelizations in Section 6.1, where I provide the basic definitions and some simple examples, we formulate the first main result of the current chapter in Section 6.2:

▷ Informal Version of Theorem 77.

Parallel parameterized algorithms are equivalent to parallel preprocessing, that is, a problem lies in para-AC^1 if, and only if, a kernel of it can be computed in FAC^1 ◁

After proving this interesting equivalence, we concentrate on concrete kernelizations in Section 6.3 and 6.4. We will present multiple results, which are similar in spirit – as representative example:

▷ Informal Version of Theorem 80.

The problem p_k -VERTEX-COVER admits a kernel of polynomial size computable in FTC^0 and it admits an exponential kernel computable in FAC^0 ◁

I present similar results for the matching problem parameterized by the solution size, as well as for the problems of computing a tree, path, or a treedepth decomposition parameterized by the vertex cover number of the input graph. On the negative side, we establish lower bounds for kernel sizes that are achievable in parallel: We prove that computing certain kernels of linear size for p_k -VERTEX-COVER is equivalent to computing large matchings – and it is a long-standing open problem whether this is possible in parallel.

- ▷ Informal Version of Fact 82 and Theorem 83.
Computing a “Nemhauser–Trotter fashioned” kernel is as hard as computing maximal matchings in bipartite graphs. ◁

At the end of the chapter I present what I call “a little gem of parameterized kernelization”: under the *massive* use of color coding, we will turn a *very sequential* kernelization into a constant-time computable one. This demonstrates, on one hand, the power of color coding in parallel parameterized computations and kernelizations, and will on the other hand place $p_{k,d}$ -HITTING-SET in para-AC^o. This in turn equips us with a powerful tool that will serve as a working-horse in the design of many further parallel parameterized algorithms.

- ▷ Informal Version of Corollary 111.
A kernel for $p_{k,d}$ -HITTING-SET can be computed in FAC^o. ◁

6.1 A SHORT REVIEW OF KERNELIZATIONS

As mentioned in the introduction to this chapter, for most problems there is probably no algorithm that can reduce any instance arbitrarily. However, there might be an algorithm that guarantees to reduce any instance to a smaller instance of size bounded by some function in the parameter. This idea is formalized through *kernelization* and is one of the cornerstones of parameterized complexity theory.

- ▶ Definition 72 (Kernelization and Kernel)
Let (Q, κ) be a parameterized problem with $Q \subseteq \text{STRUC}[\tau]$ and $\kappa: \text{STRUC}[\tau] \rightarrow \mathbb{I}$. A *kernelization* is a function $K: \text{STRUC}[\tau] \rightarrow \text{STRUC}[\tau]$ such that for all $S \in \text{STRUC}[\tau]$ and some computable function $f: \mathbb{I} \rightarrow \mathbb{N}$ we have:

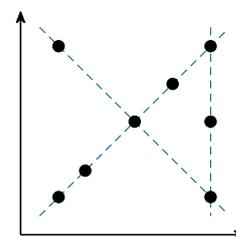
1. $S \in Q \iff K(S) \in Q$;
2. $|\text{code}(K(S))| \leq f(\kappa(S))$.

The image $K(S)$ of S under K is called the *kernel* of S . We say a kernelization is a \mathcal{C} kernelization for some functional complexity class \mathcal{C} if $K \in \mathcal{C}$. Finally, we say K is a *linear, polynomial, or exponential kernelization* if the function f in the definition above is linear, polynomial, or exponential, respectively. ◁

Note that in the above definition K is *not* a parameterized function, that is, K has to evaluate κ by itself if it wants to use the value $\kappa(S)$. Further observe that a kernelization can be seen as a self-reduction of Q with the additional requirement that the size of the produced instance is bounded by the parameter. As a note of caution it should be pointed out that the term “reduction” is to be understood in the “classical” computer science manner – besides the fact that almost all kernelizations use so called “reduction rules” to directly reduce the size of an instance, the image $K(S)$ in principle has nothing to do with S except for membership-equivalence.

For our running example of p_k -PLANAR-INDEPENDENT-SET, we make the following observation to obtain a kernel with $4k$ vertices [90]: On input of $G = (V, E)$ and $k \in \mathbb{N}$, we first check whether G is actually planar. If not, we simply output a trivial no instance (for instance $(\circ \rightarrow, 2)$). Using the famous Four Color Theorem [144], we observe that, as G is planar, it can be colored with four colors such that the vertices of every color class constitute an independent set. Therefore, if $|V| \geq 4k$ we know that G contains an independent set of size at least k and output a trivial yes-instance (for instance $(\circ \rightarrow, 1)$). Otherwise, we know $|V| < 4k$ and G itself is our desired kernel. Observe that all operations can be performed by a uniform family of FAC^1 -circuits (the only non-trivial operation is the planarity test, which is possible in AC^1 [4]) and, thus, p_k -PLANAR-INDEPENDENT-SET admits an FAC^1 -kernelization with $4k$ vertices.

The above example is simple, as the problem itself states that it makes no sense to study large instances. Usually, we will need much more machinery in order to reduce huge problem instances to small kernels. In order to get used to the notation of kernelization, let us study the following problem from computational geometry. We would like to know, given a *huge* set of points, whether we can cover them all with *just a few* straight lines. The following definition precisely describes the input for this NP-complete problem [122]. An example instance with $k = 3$ is shown in the margin, where the dashed lines depict a solution that is of course not part of the input.



► Problem 73 (POINT-LINE-COVER)

Instance: A set of points $p_1, \dots, p_n \in \mathbb{Z}^d$ for a fixed $d \geq 2$ and a number $k \in \mathbb{N}$. Both, the points and k , are encoded as binary numbers.

Question: Can we cover all points by at most k straight lines? ◁

With the concept of kernelization in mind, we would like to get rid of as many points as possible *before* we start to actually solve the problem. The following simple observation due to Kratsch, Philip, and Ray leads to a kernel with at most k^2 elements, which turns out to be optimal (unless $\text{coNP} \subseteq \text{NP/poly}$) [122]: Consider any line that covers *more* than k points, then this line *must* be in any solution. Assume for a contradiction that we would not take the line, then we would need a unique line for each of the points (since we have to cover them all) – however, since the line we try to replace did cover more than k points, we would require more than k replacement lines – a contradiction. We call the process of taking such lines into the solution and, thus, reducing the size of the instance, a *reduction rule*; and we have just argued that the presented rule is *safe*, meaning that it produces an equivalent instance. A typical pattern in the design of kernelizations is to apply such a safe reduction rule exhaustively and, afterwards, to *count* the remaining elements of the instance. Assume the aforementioned rule cannot be applied anymore, then every possible line covers at most k points. Furthermore, we are allowed to use at most k lines and, hence, if the instance has still more than k^2 points we can safely “reject,” which means “map to a trivial no-instance” in the language of kernelization. The following theorem shows that we can compute this simple kernelization quickly in parallel:

► Theorem 74

There is a uniform family of FTC° -circuits that, on input of a set of distinct points $p_1, \dots, p_n \in \mathbb{Z}^d$ and a number k , outputs a p_k -POINT-LINE-COVER kernel with at most k^2 points.

Proof. First observe that the reduction rule “for a line covering at least $k + 1$ points, remove all points on this line and reduce k by 1” can be applied in parallel, as removing all points from a line removes at most one point from any other line. To complete the proof, note that it is sufficient to check all n^2 line segments defined by pairs of points in parallel; and that a TC° -circuit can check if another point lies on such a line segment as it can multiply and divide binary numbers [106]. \square

From a circuit complexity point of view we may ask to improve the result of Theorem 74 in terms of circuit classes. Precisely, we would like to know if it is possible to compute the same kernel in FAC° . The following lemma answers this in the negative. In fact, since it is known that $\text{AC}^\circ \subsetneq \text{TC}^\circ$ [93], the lemma shows *unconditionally* that no kernel of *any size* can be computed for p_k -POINT-LINE-COVER in FAC° . Note that the result only holds under *constant-depth reductions*: We say a preserving function $f: \text{STRUC}[\tau] \rightarrow \text{STRUC}[\sigma]$ constant-depth reduces to another preserving function $g: \text{STRUC}[\rho] \rightarrow \text{STRUC}[\tau]$ if there is a uniform family $(C_n)_{n \in \mathbb{N}}$ of $\text{FAC}^\circ[g]$ -circuits such that $C_{|\text{code}(A)|}(\text{code}(A)) = \text{code}(f(A))$ for all $A \in \text{STRUC}[\tau]$. Here, an $\text{FAC}^\circ[g]$ -circuit is an FAC° -circuit that is equipped with additional “ g -gates,” which naturally compute the function g .

► Lemma 75

For every fixed k , the k th slice of the problem p_k -POINT-LINE-COVER is TC° -complete under constant-depth reductions.

Proof. We start with the case $k = 1$ and $d = 2$, which is clearly in TC° as in this case an instance is a yes-instance if, and only if, the input points are colinear. To see that the problem is TC° -hard we reduce from DIVISION defined as: Given three numbers x , y , and z , is it true that $x/y = z$? This is a classical TC° -complete problem (under constant-depth reductions) [106]. For the reduction let x , y , z be the DIVISION-instance, we construct the instance $a = (0, 0)$, $b = (x, z)$, $c = (y, 1)$ of 1-POINT-LINE-COVER. This is a yes-instance if the points are colinear, that is, if we have $(b - a) \cdot (c - a) = 0$ or, equivalently: $\frac{x-0}{y-0} = \frac{z-0}{1-0} \iff x/y = z$. Since the cases $k > 1$ and / or $d > 2$ are generalizations, they remain TC° -hard. To see that these cases are also in TC° observe that we have to consider at most n^2 line segments from which we have to pick k , that is, there are at most $\binom{n^2}{k} \leq n^{2k}$ solution candidates. For fixed k , these candidates can be checked in parallel by a TC° -circuit and can be evaluated as in the case of $k = 1$. \square

Intuitively, Lemma 75 states that p_κ -POINT-LINE-COVER is complete for para-TC^o. And indeed, this intuition can be formalized by a result of Flum and Grohe that states that a problem is complete for a parameterized class if finitely many slices of the problem are complete for the corresponding classical complexity class [84, 85] (see also Section 3.3 in [154] for further discussions). However, applying the result to our problem would require us to restate all the technical definitions in the light of constant-depth reductions. We will save ourselves the trouble at this point as the gain is relatively small compared to the required effort – Lemma 75 is already strong enough to serve as the sought lower bound.

6.2 PARALLEL PARAMETERIZED ALGORITHMS EQUAL PARALLEL PREPROCESSING

Kernelization is not just a useful tool for preprocessing, it is also a natural alternative definition for the whole parameterized complexity theory. In particular, it is known that a decidable problem is in FPT if, and only if, it admits a polynomial time computable kernelization.

► Fact 76 (for instance [85])

A decidable parameterized problem (Q, κ) is in FPT if, and only if, there is a kernelization K of (Q, κ) with $K \in \text{FP}$. ◀

We will show in the rest of this section that the same relation holds in the parallel setting, that is, a decidable parameterized problem admits a fast parallel parameterized algorithm if, and only if, it admits a fast parallel kernelization. More precisely, a problem is in a parallel subclass of FPT if, and only if, it has a kernelization in some parallel subclass of FP, that is, somewhere within the FNC-hierarchy. I think this claim is, prior to the results of this thesis, somehow surprising as almost all kernelizations – at least in the way they are stated in the literature – have a very “sequential touch.” However, the previous chapters have already provided parallel parameterized algorithms for a variety of problems and, thus, by the following theorem they all obtain a parallel kernelization. We will study more natural examples for parallel kernelizations in the following sections.

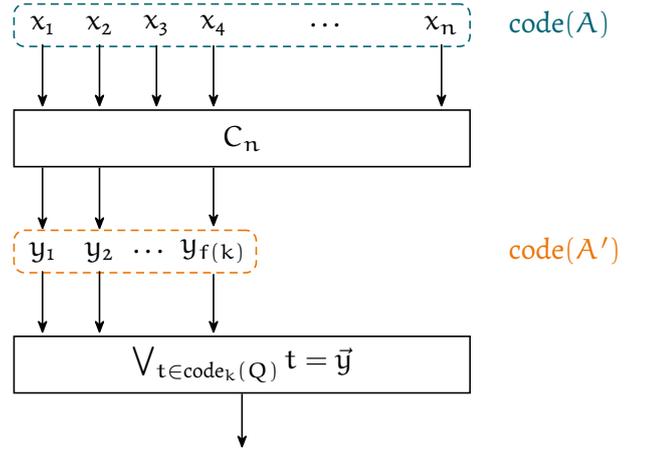
Before we state the main theorem of this section, let us be more specific about what we mean by an FAC^i -kernelization. It is, of course, a circuit-family $(C_n)_{n \in \mathbb{N}}$ that computes the kernelization function K such as in Definition 72. More precisely, C_n has n input-gates and $2n$ output-gates. The input-gates expect a structure A in form of $\text{code}(A)$, and the first n output-gates will output $\text{code}(K(A))$ padded with 0s. The second block of n output-gates will output a *bitmask* that indicates which of the first n output-gates are relevant for the kernel – there will be at most $f(\kappa(A))$ such bits. We stipulate that these output-bits must be sorted in the following way: The kernel is presented in a continuous block at the beginning of the first n output-gates, that is,

the bitmask presented at the second n output-gates consists of a block of 1s followed by a block of 0s. Note that this is a small restriction of Definition 72 as, in principle, a kernelization could output a kernel that in fact is *larger* than the input. However, in such scenarios we can always use the non-modified input as alternative kernel.

► Theorem 77

A decidable parameterized problem (Q, κ) is in para-AC^i if, and only if, it admits a kernelization computable in uniform FAC^i

Proof. For the first direction let (Q, κ) be decidable and let $(C_n)_{n \in \mathbb{N}}$ be a family of uniform FAC^i -circuits that computes a kernelization of (Q, κ) . Recall that for any structure A with $|\text{code}(A)| = n$ the circuit C_n will output the code of a structure A' such that (i) $A \in Q \Leftrightarrow A' \in Q$ and (ii) $|\text{code}(A')| \leq f(\kappa(A))$ for some computable function $f: \mathbb{I} \rightarrow \mathbb{N}$. Note that the output of C_n is a padded string together with a bitmask. Let us define the code words of Q as $\text{code}_k(Q) = \{\text{code}(A) \mid A \in Q \wedge |\text{code}(A)| \leq f(k)\}$.

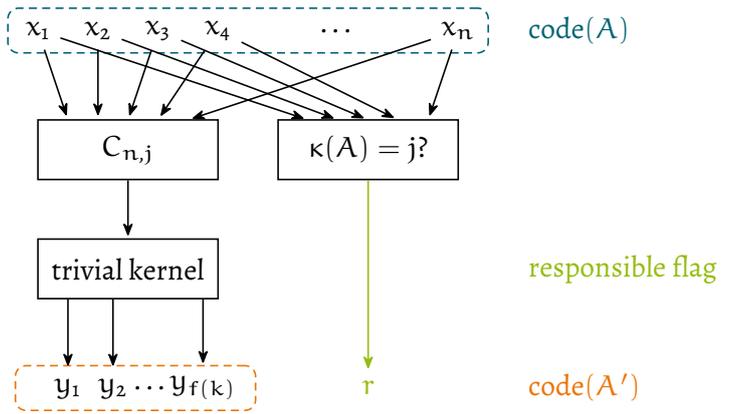


We construct a family $(C_{n,k})_{n,k \in \mathbb{N}}$ of para-AC^i -circuits that decide (Q, κ) . The circuit $C_{n,k}$ is sketched in the figure and works as follows: First, it uses C_n as a subcircuit in order to reduce A to an equivalent instance A' of size at most $f(k)$. Second, the circuit extracts $\text{code}(A')$ using the bitmask. Afterwards, it tests for all $t \in \text{code}_k(Q)$ in parallel if any of them equals $\text{code}(A')$. If this is the case, the circuit accepts, otherwise it rejects. The correctness of the circuit is immediate. For the size and depth observe that C_n itself is an FAC^i -circuit and, thus, fulfills the size and depth requirements. The attached test is performed by an AC -circuit of constant depth and size $g(k) = 2^{f(k)}$. Observe that $(C_{n,k})_{n,k \in \mathbb{N}}$ is uniform as $(C_n)_{n \in \mathbb{N}}$ is uniform and since Q is decidable. The decidability is required by the uniformity Turing machine, which computes $\text{code}_k(Q)$ and hard-wires its elements into $C_{n,k}$.

For the other direction let us assume $(Q, \kappa) \in \text{para-AC}^i$ witnessed by a uniform family $(C_{n,k})_{n,k \in \mathbb{N}}$ of para-AC^i -circuits, and let us first assume $i > 0$.

We construct a family $(C_n)_{n \in \mathbb{N}}$ of FAC^i -circuits that compute the kernelization. The circuit C_n consists of multiple subcircuits C_n^j , which are sketched in the figure on the next page and work as follows: On input of $\text{code}(A)$ they test whether we have $\kappa(A) = j$ (which is possible since κ can be evaluated in FAC^0 by Proviso 27) and set a flag that indicates whether this is the case or not.

If the flag is set, C_n^j is *responsible* for A , otherwise it is not. Parallel to this operation, C_n^j uses $C_{n,j}$ to test $A \in Q$ and produces, using this information, a trivial kernel as output – that is, a fixed yes-or-no-instance. Note that the computation of $C_{n,j}$ and, thus, the produced kernel of C_n^j is not sensibly defined if C_n^j is not responsible.



We now describe the circuit C_n , which is sketched in the figure on the next page (the bitmask-gates are omitted). Define $\ell \in \mathbb{N}$ to be the maximum k with $f(k) \leq c \log^i n$, that is, $f(\ell) \leq c \log^i n$ and $f(\ell + 1) > c \log^i n$. The circuit contains C_n^0, \dots, C_n^ℓ as subcircuits and evaluates them all in parallel. If any C_n^j is responsible, C_n presents the kernel produced by C_n^j as output. If all C_n^j signal that they are not responsible, C_n can conclude that $f(k) > c \log^i n$ and, thus, the whole instance is already a kernel. Therefore, C_n may in this case simply pipe the input to the output.

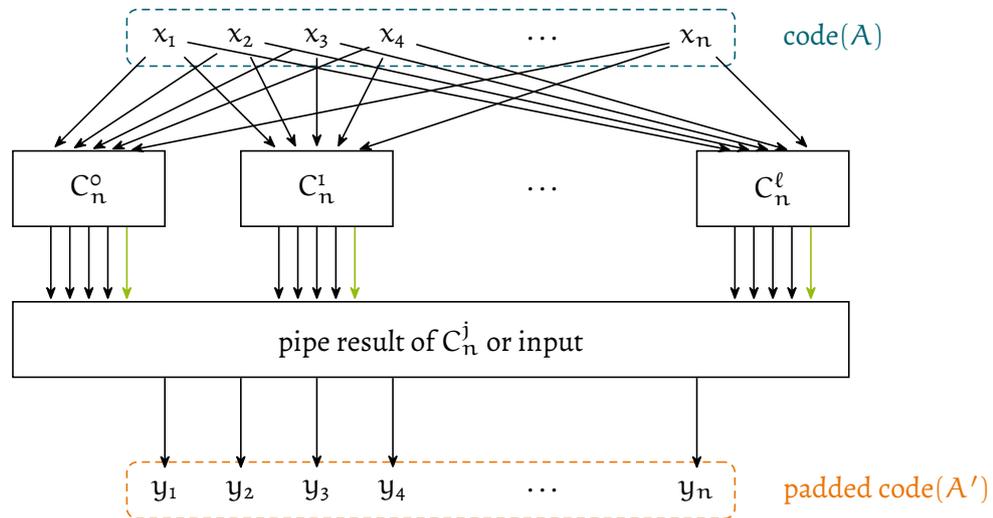
To see that the resulting circuit is an FAC^i -circuit, just observe that there is a constant c' such that $C_{n,j}$ has, by definition, depth at most

$$f(j) + c \log^i n \leq (c + 1) \log^i n \leq c' \log^i n$$

and size at most $f(j) \cdot n^c \leq c \log^i n \cdot n^c \leq n^{c'}$. Again, we are left with the task of arguing that $(C_n)_{n \in \mathbb{N}}$ is uniform. The C_n^j are uniform since $C_{n,j}$ is uniform and since we consider them only for j with $f(j) \leq c \log^i n$. The tricky part is the computation of ℓ , which is required to provide a description of C_n . This means we have to construct a Turing machine that, in time $O(\log n)$, finds the maximum ℓ with $f(\ell) \leq c \log^i n$. First observe that such a machine can compute the value $c \log^i n$, because $\log n$ is a $\log \log n$ -bit number, and because c and i are constants – in particular, $c \log^i n$ is a $2^i \log \log n \in O(\log \log n)$ -bit number. The challenging part is the search for ℓ and the evaluation of $f(\ell)$.

By replacing the family of para-AC^i -circuits by an equivalent family, we may assume that f is monotonically increasing with $f(x) > x$ for all $x \in \mathbb{N}$. By another replacement of this family, we may further assume that a Turing machine can compute $f(x)$ on input $\text{bin}(x)$ in time $O(\log f(x))$: To see this, observe that f is computable and, thus, there is some Turing machine that computes $f(x)$ in time $T(x)$ such that $T(x)$ is monotonically increasing with $T(x) > x$ for all $x \in \mathbb{N}$. We replace $f(x)$ by the function $g(x) = 2^{T(x)}$. Note that a Turing machine can now compute $T(x)$ in time $\log g(x)$. Since computing a power of two is a simple bit operation, the Turing machine can also compute $g(x)$ within the same time bound.

Given the modified family of circuits, the uniformity Turing machine can find ℓ via binary search: Since $f(x) > x$ we know $\ell \leq c \log^i n$ and since f is monotonically increasing, a binary search can be applied. Therefore, the Turing machine has to test only $\log(c \log^i n) \in O(\log \log n)$ possible values for ℓ .



For the remaining case of $i = 0$, we perform the same construction, but choose ℓ such that $f(\ell) \leq n^c$; that is, we bound the subcircuits by size and not by depth. \square

Note that the “replace the circuit family with an equivalent family” operation used in the theorem is a formal way of stating “use the same family of circuits, but replace the function f in the definition with one that is well behaved.” As long as the function g used to replace f is computable and fulfills $g(x) \geq f(x)$ for all $x \in \mathbb{N}$, the resulting family still satisfies all properties of para-AC^i and is clearly equivalent to the original family.

Observe that the above theorem also holds if we replace AC-circuits with either NC- or TC-circuits. The sole exception is NC^0 , as this class may not be powerful enough to compute κ . A nice consequence of this theorem is that “parallel parameterized preprocessing” equals “parallel preprocessing,” meaning that a para-FAC^i -kernelization can be turned into a “real” FAC^i -kernelization:

► Corollary 78

Let Q be decidable and let (Q, κ) have a kernelization that is computable in para-FAC^i . Then (Q, κ) has a kernelization that can be computed in FAC^i . \triangleleft

Proof. By the assumption of the statement, it follows that (Q, κ) lies in para-AC^i as such a family of circuits can compute the kernelization and then, in a second step, solve the problem via “brute-force.” Given $(Q, \kappa) \in \text{para-AC}^i$, Theorem 77 directly implies the FAC^i -kernelization. \square

Similar to the fact that we may always obtain an FPT-algorithm with a run time of the form $f(k) + n^c$, we may also adapt the definition para-AC^i to have size bounds of this form, while at the same time removing the parameter dependency from the depth. It should be noted, however, that this is a purely theoretical result as the produced function g may grow exponentially faster than the original function f . It shows, however, that we can always search for parameterized parallel algorithms that run in polylogarithmic time and whose work is polynomial plus an *additive* term depending only on the parameter.

► Lemma 79

Let (Q, κ) be a parameterized problem with $(Q, \kappa) \in \text{para-AC}^i$. Then there are a computable function $g: \mathbb{I} \rightarrow \mathbb{N}$ and a constant c' such that there is a uniform family $(C'_{n,k})_{n,k \in \mathbb{N}}$ of para-AC^i -circuits that decides (Q, κ) and in which every $C'_{n,k}$ has depth at most $c' \log^i n$ and size at most $g(k) + n^{c'}$.

Proof. Since $(Q, \kappa) \in \text{para-AC}^i$ there is a uniform family $(C_{n,k})_{n,k \in \mathbb{N}}$ of para-AC^i -circuits that decides (Q, κ) . By Theorem 77 there is a constant c' and a uniform family $(C_n)_{n \in \mathbb{N}}$ of FAC^i -circuits such that every C_n has depth at most $c' \log^i n$ and size at most $n^{c'}$ and produces a kernel of size at most $f(\kappa(x))$. We construct the desired family $(C'_{n,k})_{n,k \in \mathbb{N}}$ as follows: The circuit $C'_{n,k}$ first applies the circuit C_n to an input x and obtains an instance x' of size at most $f(\kappa(x))$, then the circuit uses a constant number of AC layers to check $x' \in Q$ by testing in parallel for all $w \in Q$ with $|w| \leq f(\kappa(x))$ whether $w = x'$ holds.

The depth of $C'_{n,k}$ equals (up to a constant) the depth of C_n , and the size of $C'_{n,k}$ is the sum of the size of C_n and the size of the “brute force” circuit applied at the end, that is, there is a computable function $g: \mathbb{N} \rightarrow \mathbb{N}$ such that size of $C'_{n,k}$ can be bounded by $g(\kappa(x)) + n^{c'}$. \square

6.3 KERNELIZATIONS FOR VERTEX COVER AND MATCHING

In this section we study parallel kernelizations for p_k -VERTEX-COVER. Recall that in this problem we are given a graph, and we try to identify a small set of vertices such that every edge is incident to at least one vertex in this set. The vertex cover problem is a prime example of parameterized complexity theory in both, the design of fast parameterized algorithms and the design of kernelizations that produce small kernels. In fact, the problem is a prime example in the development of parallel kernelizations. The first parallel kernelization (actually, it is a logspace kernelization) is due to Cai et al. [46]. Later on, this kernelization was implemented in FTC° by Elberfeld, Stockhusen, and Tantau [76].

Both algorithms are based on the following two simple reduction rules, known as the *Buss kernelization* in the literature: (i) any vertex of degree at least $k + 1$ must be in any size- k vertex cover, and (ii) any isolated vertex is not needed for a vertex cover. The first rule may appear familiar, as it is quite similar to the rule we used for p_k -POINT-LINE-COVER. The argument that the rule is correct is similar: just assume we would not take the vertex, then we would have to take all $k + 1$ neighbors into the solution – and this is obviously too much for a size- k solution. The correctness of the second rule is even more obvious, there is no need to select an element in a minimization problem that does not give any benefit. To see that an exhaustive application of these rules result in a kernel of size $O(k^2)$ we have to count again. In the resulting graph every vertex has degree at most k and, hence, any vertex that we add to the solution may cover at most k edges. In conclusion, a size- k vertex cover may cover at most k^2 edges and, thus, if the resulting graph has more than k^2 edges we may reject it. Finally, since the graph has no isolated vertices, it may have at most $2k^2$ vertices (in fact, we can count more carefully to obtain a bound of $k^2 + k$), which provides the claimed kernel size. The following theorem shows that we can “push” the kernelization by Elberfeld et al. from FTC° to FAC° if we are willing to pay an exponential increase in the kernel size. Note that such an improvement was not possible for p_k -POINT-LINE-COVER, even for larger increases of the kernel size.

► Theorem 80

There is a uniform family of FAC° -circuits that, on input of a tuple (G, k) , outputs a p_k -VERTEX-COVER kernel.

In order to prove the theorem, we will first prove the following more general statement: We can simulate para-TC° -circuits with para-AC° -circuit if the threshold of all threshold-gates is bounded by a function in the parameter.

► Lemma 81

Let $f: \mathbb{I} \rightarrow \mathbb{N}$ be a computable function and $(C_{n,k})_{n \in \mathbb{N}, k \in \mathbb{I}}$ be a uniform family of para-FTC° -circuits such that in every $C_{n,k}$ the maximum threshold used by any threshold-gate is bounded by the value $f(k)$. There is an equivalent family of uniform para-FAC° -circuits that compute exactly the same function.

Proof. Let $C_{n,k}$ be a para- FTC° -circuit as in the statement, and let its size be bounded by $f(k) \cdot n^c$. Replace all occurrences of threshold-gates by para- FAC° -circuits that implement the circuit of Lemma 45. Since they all have constant depth, the overall depth of the circuit increases only by a constant factor. Furthermore, the size of the resulting circuit is bounded by $g(f(k)) \cdot n^{c'}$, where $g: \mathbb{I} \rightarrow \mathbb{N}$ is the size bounding function used in Lemma 45 and $c' \in \mathbb{N}$ a constant. Observe that the resulting family is uniform as both, $(C_{n,k})_{n \in \mathbb{N}, k \in \mathbb{I}}$ and the family from Lemma 45, are uniform. \square

Proof of Theorem 80. We start with the FTC° -kernelization from Elberfeld et al. [76]. There is obviously a para- FTC° -circuit family that implements the same function. These circuits require their threshold-gates only “to count up to k ,” as the difficult part is to identify high-degree vertices. By Lemma 81 we have an equivalent family of uniform para- FAC° -circuits that compute the desired kernel. By Corollary 78 this implies that we can compute a kernel within uniform FAC° . \square

We shall remark that the lemma that we used to transform an FTC° -kernelizations into an FAC° -kernelization will work in many other cases. We will see further examples in Section 6.4.

For now, we will stick to kernelizations for p_k - VERTEX-COVER a little longer. The situation looks pretty good: we have a *quadratic* kernel in FTC° and an *exponential* kernel in FAC° – what more could we hope for? The *best sequential* kernelization due to Chen et al. [51] achieves a *linear* kernel of size $2k$; and since the kernel will usually be fed into circuits of exponential size, in all practical situations we would prefer a linear kernel computed even somewhere in FNC over a quadratic kernel computed in FTC° . A natural next step is, thus, an attempt to parallelize this linear kernelization. Unfortunately, we can link the complexity of computing the kernelization by Chen et al. rather tightly to the computation of large matchings – and whether we can find such matchings in parallel is a long-standing open problem [79, 132]. The linear kernel is based on the following fact, known as the Nemhauser–Trotter Theorem.

► **Fact 82 (The Nemhauser–Trotter Theorem [134])**

Let $G = (V, E)$ be a graph and $I = \{x_v \mid v \in V\}$ be a set of variables. Consider any optimal solution $\beta: I \rightarrow \mathbb{R}$ for the following linear program (LPVC):

$$\begin{aligned} \min \quad & \sum_{v \in V} x_v \\ x_u + x_v \geq 1 \quad & \text{for all } \{u, v\} \in E \\ x_v \geq 0 \quad & \text{for all } v \in V \end{aligned}$$

Let $V_0 = \{v \mid \beta(x_v) < 1/2\}$, $V_{1/2} = \{v \mid \beta(x_v) = 1/2\}$, $V_1 = \{v \mid \beta(x_v) > 1/2\}$ be a partition of V . There is a minimum vertex cover S of the input graph G that satisfies $V_1 \subseteq S \subseteq V_1 \cup V_{1/2}$. \triangleleft

Chen et al. have observed that there are at most $2k$ vertices in the set $V_{1/2}$, and that this set directly yields the desired kernel [51]. Hence, to compute the kernel we have to compute a solution to the linear program used in the Nemhauser–Trotter Theorem, and we will see that this is a difficult task.

► Theorem 83

Computing a solution for LPVC is NC-equivalent to computing a maximum matching in bipartite graphs.

Proof. The first direction is essentially the standard way of efficiently solving LPVC: Given an instance of LPVC we construct a bipartite graph $H = (\{v_1, v_2 \mid v \in V\}, \{\{u_1, v_2\}, \{u_2, v_1\} \mid \{u, v\} \in E\})$ and compute a minimum vertex cover S of it. It is known that the following assignment is an optimal solution for LPVC [59]:

$$\beta(x_v) = \begin{cases} 0 & \text{for } |\{v_1, v_2\} \cap S| = 0, \\ 1/2 & \text{for } |\{v_1, v_2\} \cap S| = 1, \text{ and} \\ 1 & \text{for } |\{v_1, v_2\} \cap S| = 2. \end{cases}$$

Since H is bipartite, computing a minimum vertex cover is equivalent to computing a maximum matching due to König's Theorem [120]. More precisely: To obtain the vertex cover S , we compute a maximum matching in H and this matching constitutes an optimal solution to the dual program of LPVC. Due to the Complementary Slackness Theorem, we can derive an optimal solution for the primal program from an optimal solution of the dual program by solving a linear system of equations, which is possible in NC [114]. Note that the matrices of both LPVC and its dual program are totally unimodular, as the incidence matrix of a bipartite graph is totally unimodular, and since the transpose of a totally unimodular matrix is so as well. Therefore, Cramer's Rule states that the solution that we obtain for the dual program with the algorithm from above is integral as well [58, 121]. This completes this part of the proof.

For the other direction, the input is a bipartite graph $G = (V, E)$ in which we search for a maximum matching. Let β be an optimal real solution of LPVC for G . We can transform β into a (still optimal) half-integral solution β' by simple rounding:

$$\beta'(x_v) = \begin{cases} 0 & \text{if } \beta(x_v) < 1/2, \\ 1/2 & \text{if } \beta(x_v) = 1/2, \text{ and} \\ 1 & \text{if } \beta(x_v) > 1/2. \end{cases}$$

This well-known fact is based on [134], and can be shown by the following procedure that successively transforms the assignment β into a refined optimal solution, ending at β' . To refine β we define the two sets $V_+ = \{x_v \mid 0 < \beta(x_v) < 1/2\}$ and $V_- = \{x_v \mid 1/2 < \beta(x_v) < 1\}$.

We now define for a suitable small $\epsilon > 0$ the two assignments

$$\beta_+(x_v) = \begin{cases} \beta(x_v) & \text{if } x_v \notin V_+ \cup V_-, \\ \beta(x_v) + \epsilon & \text{if } x_v \in V_+, \text{ and} \\ \beta(x_v) - \epsilon & \text{if } x_v \in V_-, \end{cases}$$

and

$$\beta_-(x_v) = \begin{cases} \beta(x_v) & \text{if } x_v \notin V_+ \cup V_-, \\ \beta(x_v) - \epsilon & \text{if } x_v \in V_+, \text{ and} \\ \beta(x_v) + \epsilon & \text{if } x_v \in V_-. \end{cases}$$

Observe that both, β_+ and β_- , are still feasible solutions, as for any edge $\{u, v\}$ the constraint $x_u + x_v \geq 1$ is satisfied (either one of the variables is already 1, or they are both $1/2$, or we add ϵ to at least one of them). Compared to β , the value of the target function changes by $\epsilon|V_+| - \epsilon|V_-|$ and $\epsilon|V_-| - \epsilon|V_+|$, respectively. Since β is optimal, neither β_+ nor β_- may reduce the value of the target function compared to β ; consequently we have $|V_+| = |V_-|$, and β_+ and β_- are both optimal solutions. Conclusively, by repeating this process successively, we will end up at β' .

To conclude this part of the proof, we will now turn β' into an integral solution. To achieve this, we construct an auxiliary graph G' by deleting all vertices with value 1 in G (as these must be in the vertex cover). Since all vertices with value 0 are now isolated, we may remove them too. We end up with a bipartite graph G' with n' vertices, which are all assigned with the value $1/2$ by β' . We claim β' is an optimal solution for LPVC on G' . For a contradiction assume otherwise, that is, assume there is an assignment γ with $\sum_{v \in V(G')} \gamma(x_v) < \sum_{v \in V(G')} \beta'(x_v)$. We can infer a new assignment β'' for G by "plugging" γ into β' :

$$\beta''(x_v) = \begin{cases} \beta'(x_v) & \text{if } x_v \notin V(G'); \\ \gamma(x_v) & \text{if } x_v \in V(G'). \end{cases}$$

This is a feasible solution for LPVC on G , since for all edges $\{u, v\}$ we have:

$$\beta''(x_u) + \beta''(x_v) = \begin{cases} \gamma(x_u) + \gamma(x_v) \geq 1 & \text{if } u, v \in V(G'); \\ \beta'(x_u) + \beta'(x_v) \geq 1 & \text{if } u, v \notin V(G'); \\ \beta'(x_u) + \gamma(x_v) \geq 1 & \text{if } u \notin V(G') \text{ and } v \in V(G'). \end{cases}$$

The first two lines follow by the fact that γ and β' are feasible; the last line follows by the construction of G' , as an edge $\{u, v\}$ with $u \notin V(G')$ and $v \in V(G')$ only appears if we have $\beta'(x_u) = 1$ (we have deleted isolated vertices and vertices with value 1, and here u was deleted and is not isolated). By the construction of β'' we end up with $\sum_{v \in V(G)} \beta''(x_v) < \sum_{v \in V(G)} \beta'(x_v)$, which is a contradiction as β' is an optimal solution for LPVC on G . Consequently, β' must be an optimal solution for LPVC on G' as well.

Since β' assigns $1/2$ to all vertices in G' , a minimal vertex cover of G' has size at least $n'/2$. Therefore, G' has to consist of two shores of equal size, as otherwise the smaller one would be a vertex cover of size smaller than $n'/2$. We can, thus, greedily select one shore into the vertex cover, that is, we set β' for one shore to 1 and for the other shore to 0. The obtained optimal integral solution of LPVC can be turned, as in the first direction, into a solution for the dual program in NC, that is, into a maximum matching of G . \square

Note that other kernels that are based on the Nemhauser–Trotter Theorem, such as the one by Soleimanfallah and Yeo [153], or the one by Lampis [125], do also not bypass Theorem 83. Finally, a known $3k$ -kernel, which is based on crown decompositions (we will define them in Section 7.1), also requires the computation of large matchings [59]. Since the computation of matchings turns out to be the main obstacle in the computation of small vertex cover kernels, and since it is unknown how to compute such matchings in parallel, it is a natural first step in the context of this thesis to study if we can compute kernels for p_k -MATCHING in parallel.

Of course, we have p_k -MATCHING \in para-AC⁰ by Corollary 43 and, thus, can compute an exponential kernel in FAC⁰. Our aim therefore is a polynomial kernel somewhere within FNC.

► Theorem 84

There is a uniform family of FTC⁰-circuits that, on input of a tuple (G, k) , outputs a p_k -MATCHING kernel with at most $O(k^2)$ vertices.

Proof. The circuit first computes a set $S = \{v \in V \mid |N(v)| > 2k\}$ of “high-degree” vertices. If we have $|S| \geq k$, the circuit can output a trivial yes-instance since for such a set S we can greedily match any vertex $v \in S$ with a vertex $u \in N(v) \setminus S$, reducing the available matching mates of all other vertices in S by at most two – and since they have degree at least $2k$, there are still enough mates left to match every vertex of S .

If the circuit has not finished yet, we compute a set S' consisting of S and the $2k$ lexicographically smallest neighbors of every vertex in S . Note that we have $|S'| \leq 2k^2$. Consider the graph $G' = G[V \setminus S']$. Since S was the set of high-degree vertices, G' has maximum-degree $d \leq 2k$. Our circuit now removes all isolated vertices from G' , resulting in G'' and then checks if we have $|V(G'')| \geq k \cdot 2d$. If so, we can output a trivial yes-instance since a graph with maximum degree d and minimum degree 1 always contains a matching of size $|V(G'')|/2d \geq k$. If, on the other hand, we have $|V(G'')| \leq k \cdot 2d \leq 4k^2$, the circuit outputs $G[S' \cup V(G'')]$ together with the unchanged number k .

The output clearly has size at most $O(k^2)$. To see that $G[S' \cup V(G'')]$ is a kernel, we only have to show that if G has a size- k matching M , so does $G[S' \cup V(G'')]$ (the

other direction is trivial). To see this, first note that any edge in M that does not have an endpoint in S must lie in G'' and, hence, is also present in $G[S' \cup V(G'')]$. Next, all other edges in M must have an endpoint in S and, thus, there can be at most $|S|$ such edges. We can greedily construct a matching of size $|S|$ in $G[S']$ (by the same argument as the one from the beginning of this proof for $|S| \geq k$). This means that we find a matching of size $|M|$ in $G[S' \cup V(G'')]$. \square

A neat corollary of Theorem 84 is the following, and I could imagine that it will actually find application in practice. Since the result does not really fit into the hierarchies that we study within this thesis, we formulate the result in terms of a parallel algorithm running on a PRAM, without going further into the details.

► Corollary 85

The problem p_k -MATCHING can be solved in parallel time $\text{polylog}(n) + \text{poly}(k)$ and $\text{poly}(n)$ work on a PRAM. \triangleleft

6.4 PARALLEL KERNELIZATIONS FOR PROBLEMS PARAMETERIZED BY VERTEX COVER

Not all decidable problems have polynomial-size kernels, even if we allow sequential polynomial-time to compute them. This is usually the case for graph problems in which the parameter is not the sum but rather the maximum over the parameters of all connected components of the input. In order to develop some intuition, let us assume we have such a problem that is NP-hard – which essentially means that we can reduce SAT to it. Now assume we reduce *many* instances of SAT to our problem, obtaining graphs G_1, \dots, G_ℓ . We can create a new instance G of our problem by building the disjoint union of all G_i . Observe that the size of G depends on ℓ and the maximum size of some G_i , while the parameter $\kappa(G)$ is bounded by $\max_{i=1}^{\ell} \kappa(G_i)$. If we choose ℓ much larger than the size of the individual G_i , the resulting graph G will be much larger than $\kappa(G)$. Thus, a potential kernelization for the problem would be forced to reduce G . In fact, if ℓ is large enough, the kernelization has to remove large parts of G . However, again intuitively, to achieve this, the kernelization algorithm has to reason about the individual G_i and, in fact, it will eventually be forced to discard some G_i entirely. This seems to be a tough task for a polynomial-time algorithm, as the problem is NP-hard after all. Accordingly, it seems unlikely that such a kernelization can exist. The technical details to prove such a statement are, of course, more complicated and the resulting theorems are of the form “problem (Q, κ) has no polynomial kernel unless $\text{NP} \subseteq \text{coNP/poly}$ ” [35]. Typical problems that suffer from this property are the decision versions of the graph parameters that we have encountered in Section 2.2, such as p_k -TREEWIDTH, p_k -PATHWIDTH, and p_k -TREEDEPTH.

If a problem suffers from the above result, one usually tries to achieve polynomial kernels with respect to more structural parameters. In particular, of course, parameters that will grow by taking the disjoint union of multiple instances. A common parameter in this line of research is the *vertex cover number* of the graph [38, 39, 119], that is, we wish to solve problems (such as TREEWIDTH) on graphs that have small vertex covers (we denote the resulting problem by $p_{\text{vc-TREEWIDTH}}$).

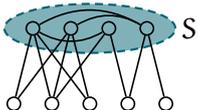
In order to be helpful, the kernelization algorithms require access to an actual vertex cover S of the input graph. Of course, we may not want to solve an NP-hard problem exactly as a preprocessing step for another computational hard problem. Instead, we will rely on an *approximation* algorithm. Fortunately, this turns out to be quite easy by using the already presented kernelization for vertex cover:

► Lemma 86

There is a uniform family of FTC° -circuits that, on input of a graph $G = (V, E)$ and a number $k \in \mathbb{N}$, outputs a set $S \subseteq V$ with $|S| \leq k^2 + 2k$ such that $G[V \setminus S]$ is edgeless, or correctly reports that no such set of size at most k exists.

Proof. The circuit uses the FTC° -implementation of the Buss kernel from Elberfeld, Stockhusen, and Tantau [76] in a slightly modified manner. Instead of outputting the $k^2 + k$ kernel, it outputs these vertices together with all vertices selected to the vertex cover (the high-degree ones), which are at most k . The result is a set S of size $k^2 + 2k$ that clearly is a vertex cover of G , which is presented as approximate solution by the circuit. Of course, if the Buss kernelization “rejects” by outputting a trivial no-instance, the circuit reports that the graph has no solution of size k . ◻

We will consider the input to $p_{\text{vc-TREEWIDTH}}$ (and the other problems) as triples $(G = (V, E), k, S)$ where $S \subseteq V$ is a vertex cover of G , as shown in the margin. Our goal will be to measure the kernel size with respect to S . This definition is justified by Lemma 86 and allows us to concentrate on the concrete kernelization techniques.



We will describe FTC° -kernelizations for $p_{\text{vc-TREEWIDTH}}$, $p_{\text{vc-PATHWIDTH}}$, as well as $p_{\text{vc-TREEDPTH}}$ based on known kernelization algorithms for these problems. In all cases the result requires the threshold gates “only for counting up to the parameter,” as it was the case in Theorem 80 and, therefore, they can be adapted to FAC° kernelizations that produce exponential-size kernels by Lemma 81. We will start with a kernel for $p_{\text{vc-TREEWIDTH}}$, which is based on [39] and the following two facts.

► Fact 87 ([38, 39, 119])

Let $G = (V, E)$ be a graph with treewidth, pathwidth, or treedepth at most k and with $u, v \in V$, $\{u, v\} \notin E$, and $|N(u) \cap N(v)| > k$. Then adding the edge $\{u, v\}$ to G will not increase the treewidth, pathwidth, or treedepth of G , respectively. ◁

► Fact 88 ([33])

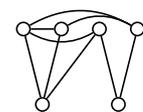
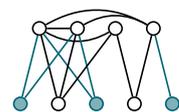
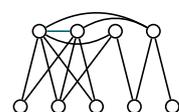
Let $G = (V, E)$ be a graph and $v \in V$ be a *simplicial* vertex, then $\text{tw}(G) \geq |N(v)|$. A vertex v is said to be simplicial if $N(v)$ is a clique. ◁

► Theorem 89

There is a uniform family of FTC° -circuits that, on input of a triple (G, k, S) , outputs a $p_{\text{vc}}\text{-TREEWIDTH}$ kernel with at most $O(|S|^3)$ vertices.

Proof. On input (G, k, S) the circuit can check if S is a vertex cover and if we have $k < |S|$. If not, it outputs a trivial no-instance in the first case and a yes-instance in the second case (a tree decomposition of width $|S|$ can be obtained from S).

The circuit now checks in parallel for every pair $u, v \in S$ with $\{u, v\} \notin E$ if we have $|N(u) \cap N(v) \cap (V \setminus S)| > k$, that is, if the two vertices have more than k common neighbors in $V \setminus S$. If this is the case, the circuit adds the edge $\{u, v\}$, as shown in the first figure. Note that this operation is safe by Fact 87 and can be applied in parallel as we consider only neighbors in $V \setminus S$ while adding only edges in S . Finally, the circuit considers all simplicial vertices $v \in V \setminus S$ (they are highlighted in the second figure) in parallel: if we have $|N(v)| > k$, the circuit safely outputs a trivial no-instance by Fact 88, otherwise the circuit can safely remove v from the input graph by standard arguments [38].



We now argue that, if the circuit has not decided yet, the remaining graph has at most $O(|S|^3)$ vertices: it consists of the vertices in S , and the nonsimplicial vertices $I \subseteq (V \setminus S)$. We have $|I| \leq |S|^3$ as any vertex $u \in I$ must have at least two neighbors v, w in S with $\{v, w\} \notin E$ (as otherwise u would be simplicial), however, every pair of nonadjacent vertices in S can have at most k common neighbors (as otherwise the circuit would have added the edge). Since we have at most $|S|^2$ such pairs, the claim follows by $k \leq |S|$. ◻

► Corollary 90

There is a uniform family of FAC° -circuits that, on input of a triple (G, k, S) , outputs a $p_{\text{vc}}\text{-TREEWIDTH}$ kernel. ◁

► Corollary 91

$p_{\text{vc}}\text{-TREEWIDTH} \in \text{para-AC}^\circ$ ◁

A similar proof works for $p_{\text{vc}}\text{-PATHWIDTH}$ and $p_{\text{vc}}\text{-TREEDEPTH}$, however, we cannot use Fact 88 for those problems and have to rely on a different way to handle simplicial vertices:

► Fact 92 ([38])

Let $G = (V, E)$ be a graph, $k \in \mathbb{N}$ be a number, and $v \in V$ be a simplicial vertex. If the degree $|N(v)|$ of v is 1 and the neighbor of v has another degree-1 neighbor, or if we have $2 \leq |N(v)| \leq k$ and for each pair $x, y \in N(v)$ there is a simplicial vertex $w \in N(x) \cap N(y)$ with $w \notin N[v]$, then we have $\text{pw}(G) \leq k$ if, and only if, $\text{pw}(G[V \setminus \{v\}]) \leq k$. ◁

► Fact 93 ([119])

Let $G = (V, E)$ be a graph, $k \in \mathbb{N}$ be a number, and let $v \in V$ be a simplicial vertex with $1 \leq |N(v)| \leq k$. If every neighbor of v has degree at least $k + 1$, then we have $\text{td}(G) \leq k$ if, and only if, $\text{td}(G[V \setminus \{v\}]) \leq k$. \triangleleft

With Fact 92 and Fact 93 we can obtain cubic kernels similar to the one of Theorem 89. The following parallel kernelizations are based on the sequential kernelization by Bodlaender, Jansen, and Kratsch [38] and the kernelization by Kobayashi and Tamaki [119]. However, the more involved rules to handle simplicial vertices cannot be parallelized as easily as the rule from Fact 88. Instead, we will use a two-phase marking scheme that makes sure that all decisions are globally conflict-free.

► Theorem 94

There is a uniform family of FTC^0 -circuits that, on input of a triple (G, k, S) , outputs a p_{vc} -PATHWIDTH kernel with at most $O(|S|^3)$ vertices.

Proof. The circuit works as in Theorem 89 and differs only in the last step, that is, the handling of simplicial vertices. We have to identify the vertices for which Fact 92 applies in constant parallel time, which is not trivial since we have dependencies between these vertices. The circuit *marks* simplicial vertices to which Fact 92 does not apply or which we will use as conditions when applying the fact to other vertices as follows: The circuit first marks for every $v \in S$ the lexicographically smallest degree-1 neighbor of v . Then for every simplicial vertex $v \in V \setminus S$ of degree at least 2, the circuit marks for every pair of neighbors x, y of v the lexicographically smallest simplicial vertex $w \in (N(x) \cap N(y)) \setminus N[v]$. If for any pair such a vertex does not exist, v marks itself. Note that all simplicial vertices that are not marked can safely be removed by Fact 92, and since the safeness is witnessed by marked vertices, the circuit can remove them all in parallel.

We are left with the task to show that there are at most $O(|S|^3)$ marked vertices left (the other vertices can be counted as in Theorem 89). We have at most $|S|$ marked vertices of degree 1 (one for each vertex in S), and at most $|S|^2$ marked vertices of degree greater than 1: each such vertex v has a pair of neighbors in S that has v as sole simplicial neighbor. \square

► Theorem 95

There is a uniform family of FTC^0 -circuits that, on input of a triple (G, k, S) , outputs a p_{vc} -TREEDEPTH kernel with at most $O(|S|^3)$ vertices.

Proof. We proceed again as in Theorem 89 and only differ in the way we handle simplicial vertices. In particular, we argue how we can apply Fact 93 in parallel constant time. The circuit starts by marking for every vertex $v \in S$ with $|N(v)| > k$ the $k + 1$ lexicographically smallest neighbors of v , then the circuit marks every simplicial vertex $v \in V \setminus S$ that has at least one neighbor of degree less than k . Note that every

simplicial vertex that is not marked can safely be removed by Fact 93 and, since this safeness is witnessed by marked vertices, these vertices can be removed in parallel.

The amount of remaining vertices can be computed as in Theorem 89, we will end the proof by counting the number of marked vertices. There are at most $|S|^2 + |S|$ vertices that were marked in the first step, as every vertex in S marks only $k + 1$ neighbors. Additionally, we may have some simplicial vertices that are marked because they have a neighbor of degree at most k . Since every degree k vertex in S can produce at most k such vertices, the number of these vertices can be bounded by $|S|^2$ as well. \square

► Corollary 96

There are uniform families of FAC° -circuits that, on input of a triple (G, k, S) , output a p_{VC} -PATHWIDTH or p_{VC} -TREEDEPTH kernels. \triangleleft

► Corollary 97

p_{VC} -PATHWIDTH \in para- AC° and p_{VC} -TREEDEPTH \in para- AC° \triangleleft

6.5 COMPUTING HITTING SET KERNELS IN PARALLEL

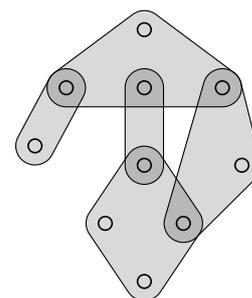
I want to close this chapter with a little gem of parallel kernelization, which demonstrates that a classical and *very sequential* kernelization can be turned into a *constant time* parallel one. A second property that makes the presented result a real gem is its generality: we will provide an FAC° -kernelization for the *hitting set* problem – a well-known generalization of the vertex cover problem to hypergraphs.

► Problem 98 (HITTING-SET)

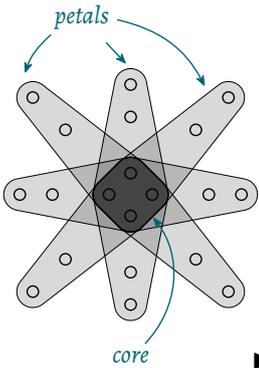
Instance: A hypergraph $H = (V, E)$ with $\max_{e \in E} |e| = d$ and a number $k \in \mathbb{N}$.

Question: Is there a set $X \subseteq V$ with $|X| \leq k$ and $e \cap X \neq \emptyset$ for all $e \in E$? \triangleleft

An example instance with $k = 3$ and $d = 4$ is visualized in the margin. It is well known that this problem is $W[2]$ -complete for parameter k and, thus, we may not hope for a kernelization in this setting [85]. Instead, we will focus on the combined parameter $k + d$ and show that, in this setting, the problem lies in para- AC° . Note that such a result is not even obvious if d is a constant, as already for $d = 3$ the problem is a generalization of the vertex cover problem.



We will present the kernelization in three steps: first we discuss the underlying sequential kernelization as described in [85]; second we will introduce a parallel version that requires $O(d)$ time following the ideas of Chen, Flum, and Huang [53]; and third we reduce the parallel time to $O(1)$ by an intensive use of color coding.



The sequential kernelization is based on a famous result by Erdős and Rado (the *Sunflower Lemma*), which states that hypergraphs of a certain size have to contain certain structures (namely, *sunflowers*) [77]. A sunflower is pictorially a collection of hyperedges that all intersect at the same position and, thus, can be drawn like a sunflower (see the graphic at the margin).

Formally the definition and the lemma is as follows:

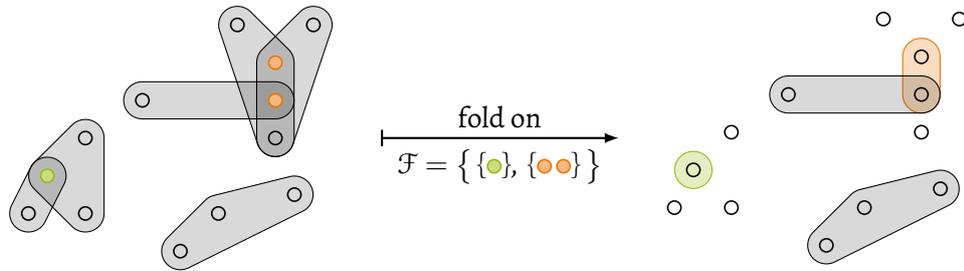
► Definition 99 (Sunflower)

Let $H = (V, E)$ be a hypergraph. A *sunflower* (s_1, \dots, s_k) of size k is a k -tuple of hyperedges $s_i \in E$ such that there is a set $C \subseteq V$ (called the *core* of the sunflower) with $s_i \cap s_j = C$ for all $1 \leq i \neq j \leq k$. ◁

► Fact 100 (Sunflower Lemma [77])

Every hypergraph $H = (V, E)$ with more than $k^d \cdot d!$ hyperedges contains a sunflower of size $k + 1$. ◁

The lemma itself directly infers a blueprint for a kernelization: as long as there are large sunflowers, remove them without changing optimality. As long as we can find the sunflowers and as long as we can safely remove them, the Sunflower Lemma will provide us with a bound on the size of the kernel. In order to turn this blueprint into an actual kernelization for $p_{k,d}$ -HITTING-SET, let us define a *fold* of a hypergraph $H = (V, E)$ into a family \mathcal{F} of sets $C \subseteq V$ as the operation that deletes every edge e from H for which there is a $C \in \mathcal{F}$ with $C \subseteq e$ and which, afterwards, adds all elements of \mathcal{F} as additional edges to H .



► Lemma 101

There is a family of FAC^0 -circuits that, on input of a hypergraph $H = (V, E)$ and a family $\mathcal{F} \subseteq 2^V$, outputs the result of the fold operation on H and \mathcal{F} .

Proof. The circuit checks in parallel for every edge whether there is a set in \mathcal{F} that is a subset of this particular edge and, if this is the case, marks the edge. Afterwards it presents all unmarked edges together with \mathcal{F} as the new hypergraph. ◻

In order to utilize the fold operation, we require another operation, called *harvest*, which obtains as input a hypergraph H and a number k , and which outputs a “suitable” family of sets $C \subseteq V$ for the fold operation – suitable will refer to the fact that a fold of H into the produced family will be safe with respect to hitting set. Given an implementation of the fold and the harvest operation, the procedure in the margin will be our working horse for a hitting set kernelization. It obtains a hypergraph H and a number k as input and, as long as $\text{harvest}(H, k)$ outputs a non-empty family \mathcal{F} , it will simply fold H into \mathcal{F} .

```

F ← harvest(H, k)
while F ≠ ∅ do
  H ← fold(H, F)
  F ← harvest(H, k)
end
return H

```

Of course, we cannot expect the algorithm to do anything useful if we fold H into arbitrary sets. However, if the harvest operation selects the sets cautiously, the fold operation will be safe with respect to hitting set, and if we care even more about the selection of such sets, the fold operation will produce the desired kernel. In the light of the Sunflower Lemma we wish, of course, to fold sunflowers within H .

► Lemma 102

Let $H = (V, E)$ be a hypergraph and \mathcal{F} be a set of sets $C \subseteq V$ such that each C is the core of a sunflower of size $k + 1$ in H . Then H and $\text{fold}(H, \mathcal{F})$ have the same size- k hitting sets.

Proof. For the first direction let X be a size- k hitting set of H . We argue that X is a hitting set of $\text{fold}(H, \mathcal{F})$. Note that every hyperedge of $\text{fold}(H, \mathcal{F})$ that is contained in H is hit by definition of a hitting set. Furthermore, for each $C \in \mathcal{F}$ we have $X \cap C \neq \emptyset$ as C is the core of a sunflower (s_1, \dots, s_{k+1}) : If X would not hit C , it would need to hit every s_i and, since the s_i intersect only in C , would require size at least $k + 1$.

For the other direction let X be a size- k hitting set of $\text{fold}(H, \mathcal{F})$. Then X is a hitting set for H as every hyperedge of H is either contained in $\text{fold}(H, \mathcal{F})$, or contains a subset $C \in \mathcal{F}$. In both cases, X hits the hyperedge in H . \square

If the operation $\text{harvest}(H, k)$ outputs a set that contains exactly one core of an arbitrary sunflower of size $k + 1$ (if such a core exists, and an empty set otherwise), the presented algorithm will replace sunflowers by their cores until no sunflowers remain. The original proof of the Sunflower Lemma is constructive and provides such an implementation of $\text{harvest}(H, k)$ in polynomial time and, together with Fact 100 and Lemma 102, this yields the sequential kernelization for $p_{k,d}$ -HITTING-SET we mentioned earlier. This implementation of the kernelization is *very sequential* and in order to parallelize it, we have to adapt the harvest operation to collect more than one core. Obviously, the more cores we find per round, the faster is the algorithm. Ultimately, we would like to fold on all possible cores at the same time – which is, surprisingly, possible due to the following lemma:

► Lemma 103

There is a family of para-FAC^o-circuits that, on input of a hypergraph $H = (V, E)$ and a number $k \in \mathbb{N}$, outputs the set of all cores of sunflowers of size $k + 1$.

Proof. Let us first observe that any core of a sunflower is a subset of some edge and, thus, there are at most $2^d \cdot |E|$ “possible cores.” The circuit in construction may check all these cores in parallel and, thus, our task reduces to check whether there is a sunflower (s_1, \dots, s_k) in H that contains a given set $C \subseteq V$ as core.

To perform this test, the circuit constructs an auxiliary graph $A = (V(A), E^A)$ with the following vertex and edge set:

$$\begin{aligned} V(A) &= \{e \setminus C \mid e \in E \wedge C \subseteq e\}, \\ E^A &= \{\{e_i, e_j\} \mid e_i \cap e_j \neq \emptyset\}. \end{aligned}$$

Observe that an independent set of size $k + 1$ in this graph is a sunflower of size $k + 1$ that contains C as core, and vice versa that any such sunflower corresponds to an independent set of size $k + 1$. Unfortunately, this graph has neither bounded degree nor is planar and, hence, we cannot apply the independent set algorithms that we have previously developed. Fortunately, the graph is still structured enough to find the independent set in parallel constant time. To achieve this, we use color coding and color the vertices of H (not of A !) with $k + 1$ colors using Theorem 42. Let $\lambda: V(H) \rightarrow \{0, \dots, k\}$ be the current coloring. We introduce an additional coloring $\chi: V(A) \rightarrow \{0, \dots, k + 1\}$ for A with one fresh color by setting for all $e \in V(A)$

$$\chi(e) = \begin{cases} c & \text{if } \lambda(v) = c \text{ for all } v \in e; \\ k + 1 & \text{otherwise.} \end{cases}$$

In words, we color a vertex of A with a fresh color if the vertices in the corresponding edge are not colored in a monochromatic way by λ , otherwise we color it with the same color that the vertices of the edge have obtained by λ . We claim that a set x_0, x_1, \dots, x_k with $\chi(x_i) = i$ constitutes an independent set of size $k + 1$. For a contradiction assume otherwise, that is, assume there are two x_i, x_j with $i \neq j$ and $x_i \cap x_j \neq \emptyset$. Then consider $v \in x_i \cap x_j$ and let $\lambda(v) = c$. Observe that we have either $c \neq i$ or $c \neq j$ and, thus, either the vertices in x_i or in x_j are not colored in a monochromatic way, implying $\chi(x_i) = k + 1$ or $\chi(x_j) = k + 1$ – the contradiction we have sought. Now for the other direction that a size $k + 1$ independent set will be colored in this way by some coloring, observe that there are at most $d \cdot (k + 1)$ vertices in H that are involved in the coloring for the independent set. Hence, a $(|V(H)|, d \cdot (k + 1), k + 1)$ -universal coloring family guarantees to find it. \square

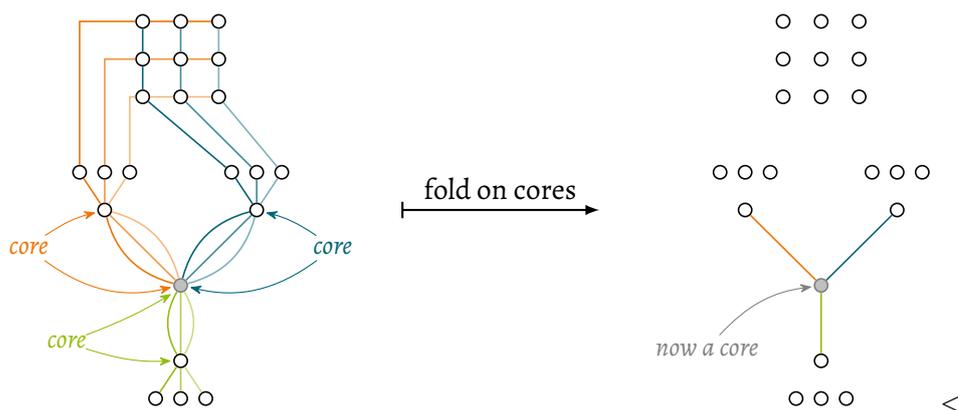
We can now run all parts of our kernelization in parallel constant time, the fold operation in FAC° and the harvest operation in para-FAC° . But for how many rounds will the whole kernelization algorithm run? Unfortunately, more than one! As the following example illustrates, replacing all sunflowers by their cores may create new sunflowers that were not present in the original graph. To get rid of these, we have to run another round and, since this may again create new sunflowers, eventually even

a third and a fourth. On the positive side, d rounds will be sufficient, as the edges we add (the cores) are always real subsets of some edges and, thus, the maximum size of an edge that participates in a sunflower will decrease in every round.

► Example 104

Consider the following hypergraph, which we illustrate for this example in metro-representation rather than set-representation. In this representation, every hyperedge has its own color and is represented by a line that “touches” every vertex contained in the hyperedge. In the hypergraph on the left, there are three sunflowers of size 3, indicated by hyperedges with the same color shade (that is, we have an orange-, a blue-, and a green-sunflower). Observe that all three sunflowers have a core of size two (they are labeled in the figure) and that all these cores share the gray vertex. However, the gray vertex alone is *not* a core of any sunflower of size 3.

In the right figure we see the hypergraph obtained by folding on all cores of sunflowers of size 3. Here, the gray vertex now is a core of a sunflower of size 3. Consequently, if we wish to bound the size of the hypergraph using the Sunflower Lemma, we have to apply the fold operation to all cores again.



The key idea of improving the parallel time of the kernelization from $O(d)$ to $O(1)$ is the observation that we are not limited to fold on cores of sunflowers. We may fold on other structures, as long as the fold is safe and as long as we can guarantee to remove sunflowers, the result will still be a kernel. To utilize this idea, we would like to identify sets that are not just “cores of sunflowers,” but also sets that are “cores of cores,” that is, sets that would become a core after one round of the algorithm. Similarly, we would like to identify “cores of cores of cores” and “cores of cores of cores of cores,” and so on. The crucial observation for the following definition is that the information whether a set is a “core of cores” is already encoded in the hypergraph, it is just hard “to see” this fact by searching for normal sunflowers. Instead, we will directly search for a structure that is a “sunflower at the border” and a “core of cores” in the center. More precisely and more formally, we introduce the notation of *pseudo-sunflowers* and *pseudo-cores*.

► Definition 105 (Pseudo-Sunflowers and Pseudo-Cores)

Let $H = (V, E)$ be a hypergraph with $\max_{e \in E} |e| = d$ and let $k \in \mathbb{N}$ be fixed. A *k-pseudo-sunflower* with *pseudo-core* $C \subseteq V$ in H is a triple (T, r, S) in which T is a k -nary tree rooted at $r \in V(T)$ and S is a mapping $S: \text{leaves}(T) \times \{0, \dots, d\} \rightarrow 2^V$ such that for all leaves $l, m \in \text{leaves}(T)$ at $\text{depth}(l)$ and $\text{depth}(m)$ we have:

1. $S(l, 0) = C$.
2. $S(l, 0) \cup S(l, 1) \cup \dots \cup S(l, \text{depth}(l)) \in E$.
3. $S(l, i) \cap S(l, j) = \emptyset$ for all $0 \leq i < j \leq \text{depth}(l)$, but $S(l, i) \neq \emptyset$ for all $i \in \{1, \dots, \text{depth}(l)\}$ and $S(l, i) = \emptyset$ for $i > \text{depth}(l)$.
4. Let $z \in \{1, \dots, \min(\text{depth}(l), \text{depth}(m))\}$ be the depth where the path from r to l and the path from r to m diverge for the first time. Then $S(l, z) \cap S(m, z)$ must be empty, that is, $S(l, z) \cap S(m, z) = \emptyset$ must hold. \triangleleft

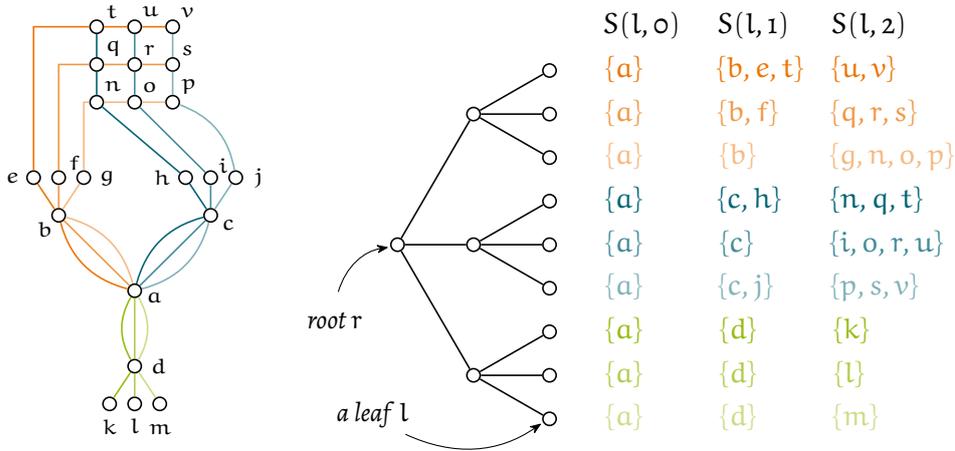
Let us develop some intuition about this technical definition. As the name suggests, the k -nary tree is something like a sunflower. Instead of petals, however, we have paths in the tree T . Condition 2 and 3 ensure that there are edges in the graph that get mapped to these paths (such that they are partitioned along the path). In a sunflower the petals are not allowed to intersect outside the core. A pseudo-sunflower weakens this requirement: By Condition 4 hyperedges may not intersect at the node where the two corresponding paths diverge for the first time – but they may intersect before and after this point.

As the following lemmas will show, this requirement is strong enough for the kernelization and works similar to the cores of sunflowers: We will argue that any hitting set of size $k - 1$ has to hit the pseudo-sunflower at such diverging points, as otherwise there will always be at least k disjoint sets after that point, which the hitting set would need to hit. By an induction, we will see that this will force any hitting set of size $k - 1$ to hit the pseudo-core.

Before we start to prove many useful properties of pseudo-sunflowers, let us first strengthen our intuition about these structure further with the following example:

► Example 106

We consider the same hypergraph as in Example 104, but this time with named vertices. The hypergraph is shown in the very left in the figure on the next page. To the right, a 3-pseudo-sunflower (T, r, S) is illustrated. We have the tree T with root r in the center, and at the very right the mapping S from leaves l of T to subsets of the vertices. As these subsets have to form a hyperedge along a path in the tree, the mapping is colored with the color of the corresponding hyperedge.



Verifying that (T, r, S) satisfies all properties of a 3-pseudo-sunflower for pseudo-core $C = \{a\}$, is a good exercise left for the reader. \triangleleft

To obtain the claimed constant-time kernelization, we will require three results: (i) we have to prove that it is safe to fold on pseudo-cores; (ii) we have to argue how to find all pseudo-cores at once; and (iii) we have to show that folding on all these pseudo-cores will not generate new pseudo-sunflowers.

► Lemma 107

Let $H = (V, E)$ be a hypergraph and \mathcal{F} be a set of sets $C \subseteq V$ such that each C is a pseudo-core of a $(k + 1)$ -pseudo-sunflower. Then H and $\text{fold}(H, \mathcal{F})$ have the same size- k hitting sets.

Proof. We argue as in the proof of Lemma 102, the only difference is that we have to prove that every size k hitting set X has to hit every pseudo-core in \mathcal{F} . Let us fix a $(k + 1)$ -pseudo-sunflower with pseudo-core $C \in \mathcal{F}$. We argue that $X \cap C \neq \emptyset$.

We say that X hits a node v of T if there is a leaf l of T such that v is at depth β on a path from r to l and $X \cap (S(l, 0) \cup S(l, 1) \cup \dots \cup S(l, \beta)) \neq \emptyset$. We prove by a reverse induction on the depth β that all nodes of T get hit, which implies that the root gets hit as well and, hence, $X \cap C \neq \emptyset$. For the base case we consider the leaves of T . Since $S(l, 1) \cup \dots \cup S(l, \text{depth}(l)) = e$ is an edge of H , the leaves get hit by X as X is a hitting set of H .

For the inductive step consider a node v at depth $\beta - 1$ with children w_1, \dots, w_{k+1} . By the induction hypothesis w_1, \dots, w_{k+1} get hit by X , which means that for each of them there is a leaf l_i such that w_i is at depth β on the unique path from r to l_i and $X \cap (S(l_i, 0) \cup S(l_i, 1) \cup \dots \cup S(l_i, \beta)) \neq \emptyset$. Since the w_i are children of v , it follows by the forth property of a pseudo-sunflower that $S(l_i, \beta) \cap S(l_j, \beta) = \emptyset$ for all $1 \leq i \neq j \leq k + 1$. Therefore, a hitting set of size k cannot hit them all at depth β , but must hit at least one of them at depth $\beta - 1$, implying that X hits v . \square

► Lemma 108

There is a family of para-FAC^o-circuits that, on input of a hypergraph $H = (V, E)$ and a number $k \in \mathbb{N}$, outputs the set \mathcal{F} of all pseudo-cores of k -pseudo-sunflowers.

Proof. Every pseudo-core is contained in some hyperedge and, thus, there are only $2^d \cdot |E|$ candidates for such pseudo-cores. Furthermore, in every k -pseudo-sunflower the tree T has depth at most d by property two and three of pseudo-sunflowers. The circuit in construction may test all possible pseudo-cores and all possible trees T in parallel. Effectively, this reduced the lemma to the claim that there is a circuit as in the lemma that, on input of a set $C \subseteq V$, a number $k \in \mathbb{N}$, and a tree T with root r , decides whether there is a k -pseudo-sunflower (T, r, S) with pseudo-core C .

The mapping S of the pseudo-sunflower essentially maps hyperedges to paths from the root r of T to its leaves. We say a vertex $v \in V(H)$ *participates* in the pseudo-sunflower if there is a leaf l at depth q and a value $\beta \in \{0, \dots, q\}$ such that we have $v \in S(l, \beta)$. The tuple (l, β) is a *witness* for the fact that v participates in the pseudo-sunflower. Observe that a vertex may have multiple witnesses as it can be contained in multiple hyperedges that are mapped to T . Let $W = \text{leaves}(T) \times \{0, \dots, d\}$ be the set of all possible witnesses. It is easy to check for a given set $c \subseteq W$ whether these witnesses respect the third and fourth property of a pseudo-sunflower. Let $\mathcal{C} \subseteq 2^W$ be the set of such sets of witnesses. We will apply color coding to identify the way hyperedges (and thus vertices) get assigned to T by S . For this task we will color $V(H)$ with \mathcal{C} and the following interpretation: if some vertex v obtains a color $c = \{(l_1, \beta_1), \dots, (l_\alpha, \beta_\alpha)\}$, then it may only participate in the pseudo-sunflower in the form of $v \in S(l_i, \beta_i)$ with $(l_i, \beta_i) \in c$. Observe that $|\mathcal{C}|$ is bounded by a function in k and d as T is k -nary and of depth at most d and, thus, $|W| \leq k^d \cdot d$. Finally, let us fixate the colors for the elements of the given pseudo-core C such that these must be assigned for every leaf to level 0 .

We argue that, if all colors are correct, we can find a pseudo-sunflower in this colorful version. Since the number of all colors is bounded by k and d , we can use a universal coloring family and Theorem 42 to find a pseudo-sunflower in the uncolored hypergraph. Fix a coloring λ as above. For every leaf l of T we consider all edges e , and we say e is *compatible* with l if for each $v \in e$ there is some β such that $(l, \beta) \in \lambda(v)$ – in other words, an edge is compatible with a leaf if every vertex of the edge is assigned to the path from the root of T to this leaf by the coloring.

To conclude the proof, we claim that we have found the sought pseudo-sunflower if we find a compatible hyperedge e_l for every leaf l . We define the mapping S for the pseudo-sunflower as $S(l, \beta) = \{v \mid v \in e_l \wedge (l, \beta) \in \lambda(v)\}$ for all $l \in \text{leaves}(T)$ and $\beta \in \{0, \dots, \text{depth}(l)\}$. The first property of pseudo-sunflowers is guaranteed as we have fixed the color for the pseudo-core; the second property holds as e_l is compatible with l ; and the third and fourth properties are implied by the choice of λ (or rather the choice of the colors \mathcal{C}). \square

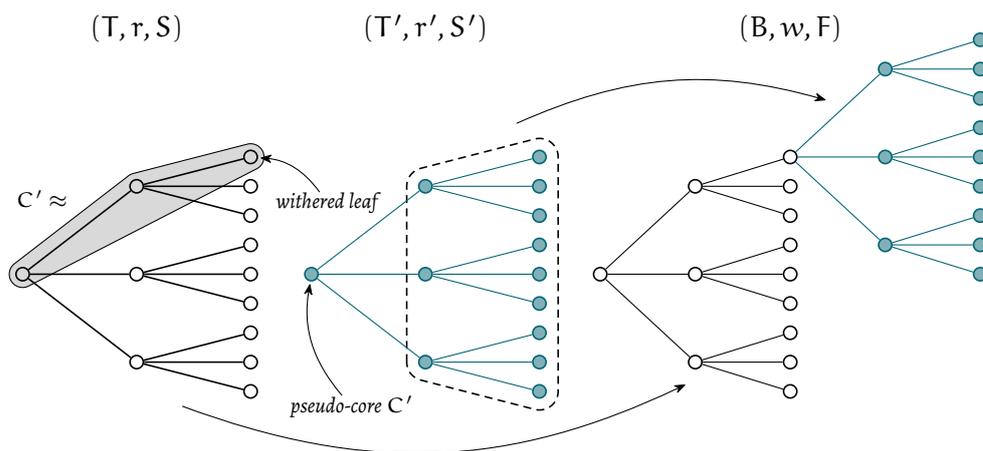
► Lemma 109

Let $H = (V, E)$ be a hypergraph, $k \in \mathbb{N}$ a number, and $\mathcal{F} \subseteq 2^V$ the set of all pseudo-cores of $(k+1)$ -pseudo-sunflowers in H . Then $\text{fold}(H, \mathcal{F})$ does not contain a $(k+1)$ -pseudo-sunflower.

Proof. For a contradiction let us assume otherwise, that is, let us assume that there is some $(k+1)$ -pseudo-sunflower (T, r, S) with pseudo-core C in $\text{fold}(H, \mathcal{F})$. We will explicitly construct a $(k+1)$ -pseudo-sunflower (B, w, F) with pseudo-core C in H . This implies that the fold operation would have removed (T, r, S) and, thus, this is the contraction we have sought.

Initially, we start by setting $B = T$, $w = r$, and $F = S$. Notice that the triple (B, w, F) is not necessarily a valid pseudo-sunflower in H , as the second property of pseudo-sunflowers may be invalidated in the form of some leaves $l \in V(B)$ at some depth q with $\bigcup_{i=0}^q S(l, i) \notin E^H$. We call such leaves *withered*, and we will show by an induction over the number p of such leaves that we can modify (B, w, F) to be a valid $(k+1)$ -pseudo-sunflower with pseudo-core C in H . This is trivial for $p = 0$, as we already have a valid pseudo-sunflower in this case. So assume the claim holds for p and let us consider (B, w, F) with $p+1$ withered leaves.

Fix any of the withered leaves l at depth q and let $\bigcup_{i=0}^q S(l, i) = C'$. Since $C' \notin E^H$ (as l is withered), there must be a $(k+1)$ -pseudo-sunflower (T', r', S') with pseudo-core C' in H . In order to remove the withered leaf l from B , we will identify l with r' , that is, we “glue” the tree T' to B at the withered leaf. Since B and T' are both $(k+1)$ -nary trees, this operation will result in a $(k+1)$ -nary tree again. The following figure illustrates this operation:



We now redefine the mapping F for every leaf l_i of B , where l is the withered leaf of T at depth q :

$$F(l_i, \beta) = \begin{cases} S(l_i, \beta) & \text{if } l_i \in \text{leaves}(T); \\ S(l, \beta) & \text{if } l_i \in \text{leaves}(T') \text{ and } \beta \leq q; \\ S'(l_i, \beta) & \text{if } l_i \in \text{leaves}(T') \text{ and } \beta > q. \end{cases}$$

Observe that the first property of pseudo-sunflowers holds inherited from (T, r, S) as we have not changed the assignment at level 0 ; the third property holds as it holds for all remaining leaves of (T, r, S) , and for the new leaves it holds inherited from the fact that it is true in (T', r', S') and by $\bigcup_{j=0}^q F(l, \beta) = C' = S'(l, 0)$; finally for the fourth property observe that all branch points at which we enforce disjointness are witnessed by the corresponding disjointness in either (T, r, S) or (T', r', S') .

On the other hand, the triple (B, w, F) may still invalidate the second property of pseudo-sunflowers and may still have withered leaves. However, we have decreased the number of these leaves by one, and therefore the induction hypothesis tells us that we can find a $(k+1)$ -pseudo-sunflower with the same pseudo-core C in H . \square

► Corollary 110

A kernel for $p_{k,d}$ -HITTING-SET can be computed in para-FAC° \triangleleft

Proof. The circuit can implement the harvest operation for pseudo-cores and can also fold on them. By the results of this section, the general kernelization algorithm will only run for one round and the circuit can therefore simulate it in constant time. Finally, since every sunflower of size $k+1$ is a $(k+1)$ -pseudo-sunflower, the result does not contain any sunflower and, thus, is a kernel by Fact 100. \square

► Corollary 111

A kernel for $p_{k,d}$ -HITTING-SET can be computed in FAC° \triangleleft

Proof. Combine Corollary 78 with Corollary 110. \square

Many problems have para-AC° -reductions to the hitting set problem and, thus, lie in para-AC° . Natural examples are the dominating set problem in graphs of bounded degree, as well as the modulator problem to H -free graphs:

► Corollary 112

$p_{k,\Delta}$ -DOMINATING-SET $\in \text{para-AC}^\circ$ \triangleleft

Proof. Build a hypergraph that contains for every vertex v the hyperedge $N[v]$. \square

► Corollary 113

Let \mathcal{F} be the family of H -free graphs, then p_k -MODULATOR(\mathcal{F}) $\in \text{para-AC}^\circ$ \triangleleft

Proof. Construct a hyperedge for every induced copy of H in G . \square

7 PARALLEL DECOMPOSITION OF GRAPHS

Graph decomposition techniques lie at the heart of modern algorithmic graph theory. Such decompositions reveal structural information of the input graph. A prominent example is the concept of tree decompositions, which we have encountered in Section 2.2. Usually, such structures can be used to guide an algorithm that solves an otherwise computational hard problem. Sometimes we can even solve a problem by “just looking” at the structural information. For instance, a graph that is similar to a tree may not have a large feedback vertex set and, thus, if we wish to solve p_k -FEEDBACK-VERTEX-SET and figure out that the input graph has large treewidth, we may directly reject the given instance.

Of course, in order to make the structure of a graph algorithmically usable, we have to discover it first – that is, we have to compute a suitable decomposition. We did this already in Section 5.2 for the embedding problem. However, in this case the graph was very small and we did “not really care” how exactly we can find a suitable decomposition. In this chapter we will change this situation and study the precise complexity of computing various decompositions in parallel. We start by introducing *crown decompositions* in Section 7.1. These are comparably simple decompositions, which we can compute them in parallel constant time:

- ▷ Informal Version of Theorem 114.

Crown decompositions can be computed in para-FAC^o. The parameter is the size of the used matching. ◁

After this introduction to the parallel computation of graph decompositions, we will consider more complex decompositions. Namely, tree and treedepth decompositions. For the later, we modify a folklore approximation algorithm to run in parallel. The main observation is that all we have to do is to compute a certain depth-first search tree – and we can do so via Theorem 35.

- ▷ Informal Version of Theorem 119.

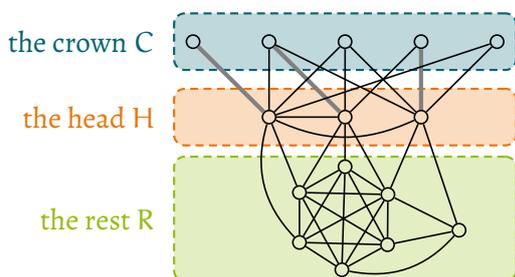
A treedepth decomposition of width $O(2^{\text{td}(G)})$ can be computed in para-FAC^{o!} ◁

Finally, in Section 7.3, we precisely analyze each subroutine of an algorithm by Bodlaender and Hagerup for computing optimal tree decompositions. This analysis will reveal that all steps of the algorithm can be executed within para-FAC:

- ▷ Informal Version of Theorem 120.

An optimal tree decomposition can be computed in para-FAC^{2!} ◁

7.1 CROWN DECOMPOSITIONS



We start with a rather simple decomposition that defines the structure of a graph quite tightly. This technique is linked to large matchings in the graph and can, for instance, be used to sequentially compute a $3k$ -kernel of p_k -VERTEX-COVER [59].

A *crown decomposition* of a graph $G = (V, E)$ is a partition (C, H, R) of V such that C is an independent set, H separates C from R , and there is a matching from H into C . We call C the *crown* that is attached to G (and separated from the *rest* denoted by R) via the set H , which is called the *head*. The figure illustrates the decomposition and justifies the naming convention. The matching from H into C is highlighted.

It is well understood how such a decomposition can be computed *sequentially* [59]. However, from a parallel point of view the situation is far less clear. In particular, it is not known how a matching (that connects the head and the crown) could be computed efficiently in parallel. However, if we assume that the head is not too big (that is, we use $|H|$ as parameter), the matching will not be too big either. This allows us to develop a parameterized parallel algorithm for computing crown decompositions without resolving the parallel complexity of the matching problem.

► **Theorem 114**

There is a uniform family of FAC-circuits of constant depth and size $f(k) \cdot |G|^c$ that, on input of an integer k and a graph $G = (V, E)$, either detects that G has less than $3k + 1$ non-isolated vertices, outputs a matching $M \subseteq E$ with $|M| = k + 1$, or outputs a crown decomposition (C, H, R) with $|H| \leq k$ and $|R| \leq 3k$ of G .

Proof. The circuit first checks whether the number of non-isolated vertices is at least $3k + 1$. This is possible by simulating threshold-gates using Lemma 45. If this is not the case, the circuit is done. Otherwise, the circuit tests for all values $k' \leq k + 1$ in parallel whether there is a matching of size k' in G . This can be done using the circuit of Corollary 43. We either find a matching of size $k + 1$ or a maximum matching M of G . In the first case the circuit outputs the matching and is done.

Let $P = \{v \mid \text{there is a } u \in V \text{ with } \{u, v\} \in M\}$ and $Q = V \setminus P$, note that Q is an independent set as M is the largest matching in the graph. Consider the bipartite graph $G' = (V, E' = \{\{u, v\} \in E \mid u \in P \text{ and } v \in Q\})$. By König's Theorem [120], the vertex cover number of G' is at most k , as G' contains no larger matching. Consequently, the circuit can use another para-FAC^o-circuit to compute a *minimum* vertex cover S of G' using Theorem 80. We have $S \cap P \neq \emptyset$, as otherwise we would have $S \subseteq Q$ and Q would contain at most k non-isolated vertices, which implies that G has at most $3k$ non-isolated vertices. Let I be the set of isolated vertices of G . It is easy

to verify that $H = S \cap P$, $C = \{v \mid v \notin S \text{ and there is a } u \in H \text{ with } \{u, v\} \in E'\} \cup I$, and $R = V \setminus (H \cup C)$ constitute a crown decomposition of G . Note that we have $|C| \geq |V| - 3k$, since we have $|P| \leq 2k$, $|Q| \geq |V| - 2k$ and $|S| \leq k$ – therefore, we also have $|R| \leq 3k$. \square

An example application for crown decompositions is a kernelization for the graph coloring problem parameterized by the number of colors that “have to be saved.”

► **Problem 115 (DUAL-COLORING)**

Instance: A graph $G = (V, E)$ and a number $q \in \mathbb{N}$.

Question: Is there a proper coloring of G with at most $|V| - q$ colors? \triangleleft

In this problem a graph $G = (V, E)$ and a parameter $k \in \mathbb{N}$ are given, and the question is whether or not we can “save” k colors when coloring G , that is, if we can color G with at most $|V| - k$ colors.

The following parallel kernelization is based on a sequential kernel due to Chor, Fellows, and Juedes that runs in quadratic time [54].

► **Lemma 116**

There is a uniform family of FAC-circuits of constant depth and size $f(k) \cdot |G|^c$ that, on input of a graph $G = (V, E)$ and an integer $q \in \mathbb{N}$, outputs p_q -DUAL-COLORING kernel with at most $3q$ vertices.

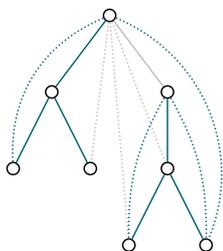
Proof. Let \bar{G} be the edge-complement graph of G , which can be computed by an FAC-circuit of constant depth. The circuit applies Theorem 114 and either gets informed that the graph has less than $3k + 1$ non-isolated vertices, obtains a matching M of size $q + 1$, or obtains a crown decomposition (C, H, R) with $|R| \leq 3q$. In the first case the circuit outputs the set of non-isolated vertices, as isolated vertices of \bar{G} are connected to all vertices in G and, hence, need a unique color. In the second case the circuit can output a trivial yes-instance, as we can save $q + 1$ colors. For the last case we observe that, since C is an independent set in \bar{G} , it is a clique connected to R in G and, thus, every vertex in C needs a unique color. However, because of the matching from H to C , the vertices of H can be colored with the same colors as the vertices in C . Therefore, the circuit can reduce the instance to $G' = G[V \setminus (C \cup H)] = G[R]$ and $q' = q - |H|$. Since $|R| \leq 3q$, this is the desired kernel. \square

► **Corollary 117**

p -DUAL-COLORING \in para-AC $^\circ$ \triangleleft

7.2 TREEDEPTH DECOMPOSITIONS

In this section we study the graph invariant treedepth. The corresponding decomposition is a tree decomposition of bounded width *and* depth. However, graphs of low treedepth actually have many different decompositions that witness the low treedepth and that are useful in different algorithmic scenarios. Besides the characterization via tree decompositions, the most common one reads as follows [135]: “the treedepth of a graph G is the minimum height of a rooted forest F such that G is contained in the closure of F .” For instance, consider the graph $G =$  and the rooted forest F (which is just a tree) shown at the margin. The closure of the forest is indicated by dotted lines and the embedding of G is highlighted. The forest F witnesses that the treedepth of G is at most 4.



Since any graph is contained in the closure of a depth-first search tree of it, we can deduce that the treedepth of a graph is bounded by the minimum depth of any such tree. Note that this observation implies that the treedepth of a graph is bounded by the longest path in the graph.

► **Fact 118** ([135])

The length of the longest path in a graph $G = (V, E)$ is bounded by $2^{\text{td}(G)}$. ◁

This leads to the convenient property that the treedepth of a graph can be approximated “just by a depth-first search.” In conjunction with Lemma 35 we can thus formulate the following theorem:

► **Theorem 119**

There is a uniform family of FAC-circuits of depth $f(k)$ and size $f(k) \cdot |G|^c$ that, on input of a graph $G = (V, E)$ and an integer k , either determines $\text{td}(G) > k$ or outputs a rooted tree decomposition (T, ι) of G of width $O(2^{\text{td}(G)})$ and such that for all nodes x, y of T we have $\iota(x) \subsetneq \iota(y)$ if y is a descendant of x .

Proof. Let us assume G is connected and let T be a depth-first search tree rooted at an arbitrary start vertex $r \in V$. For all vertices $v \in V$ we define the vertex set $\iota(v) = \{w \mid w \text{ lies on the unique path from } v \text{ to } r \text{ in } T\}$. Due to Fact 118, the depth of T is bounded by $2^{\text{td}(G)}$. Therefore, we also have $|\iota(v)| \leq 2^{\text{td}(G)}$ for each $v \in V$. Since bags extend along the paths from the root to the leaves of T , all conditions of a tree decomposition are satisfied by (T, ι) .

A circuit with the desired size and depth can compute a depth-first search labeling using Lemma 35, and either conclude that the length of the longest path exceeds 2^k (and therefore $\text{td}(G) > k$), or obtain the depth-first search labeling $\lambda: V \rightarrow \mathbb{N}$. In the later case, the circuit can compute the bags of the decomposition in parallel: For each $v \in V$ it initializes the bag $\iota(v) = \{v\}$ and, as long as $r \notin \iota(v)$, repeats the following procedure sequentially: let $w \in \iota(v)$ the vertex that minimizes $\lambda(w)$ in $\iota(v)$, add the unique $w' \in N(w)$ that satisfies $\lambda(w') = \lambda(w) - 1$ to $\iota(v)$.

To complete the proof, we have to handle the case that G is not connected. The circuit can compute all connected components of G using a breadth-first search labeling (Lemma 34). Afterwards, the circuit can apply the aforementioned algorithm to each connected component. Finally, the circuit adds a new empty root bag that is connected to the roots of all constructed tree decompositions. This operation does not increase the width and increases the depth only by one. \square

7.3 TREE DECOMPOSITIONS

In contrast to treedepth, the initial situation for treewidth looks a little better, as there are already parallel algorithms known in the literature. The first attempt was done by Bodlaender [32]. However, the resulting algorithm produces too much work and is only suitable for graphs of *constant* treewidth. The result was improved by Lagergren with a CRCW-algorithm that runs in $O(\log^3 n)$ parallel time using only $O(n)$ processors [124]. An algorithm with this time and work *can* be translated (or “seen as”) a para-AC-algorithm and, hence, implies $p_k\text{-TREewidth} \in \text{para-AC}$. Two years after the discovery of the algorithm by Lagergren, the result was again improved by Bodlaender and Hagerup to an EREW-algorithm with optimal speedup that runs in time $O(\log^2 n)$ using $O(n)$ operations [37]. This algorithm was translated to different parallel models, for instance to the parallel external memory model in a work by Jacob, Lieber, and Mních [113]. In the remainder of this section we will do the same and translate the algorithm by Bodlaender and Hagerup to our circuit model. The term “translate” stands for a careful analysis of each subroutine of the algorithm, revealing that it can be implemented in para-AC.²¹ We should note that, in contrast to the previous section, the presented algorithm is an exact algorithm rather than an approximation. Therefore, we may hope to obtain better results by implementing one of the many approximation algorithms for treewidth in parallel – such as the one by Robertson and Seymour [146]. However, none of these algorithms achieves a better parallel run time than the algorithm we present here.

► **Theorem 120**

There is a uniform family of FAC-circuits of depth $f(k) \cdot \log^2 |G|$ and width $f(k) \cdot |G|^c$ that, on input of a graph $G = (V, E)$ and an integer k , either determines $\text{tw}(G) > k$ or outputs a tree decomposition of G of width at most k .

We first provide a high-level sketch of the algorithm of Bodlaender and Hagerup, which runs in $O(\log n)$ rounds. Afterwards, we provide a series of lemmas that state that we can implement all operations performed in one round in para-AC.²¹ Putting all the pieces together, we will obtain the algorithm claimed in the theorem.

The idea of the algorithm is as follows: If $G = (V, E)$ is small enough, we can compute an optimal tree decomposition via “brute-force,” otherwise we try to reduce the graph until it has a suitable size. We call two vertices $u, v \in V$ *reduction partners* if

they are adjacent or twins (). We can reduce the size of G by 1 if we *contract* the two vertices, that is, if we remove v from G after connecting all neighbors of v to u (without creating multi-edges: ). Let G' be the resulting graph, and let (T', ι') be a recursively computed tree decomposition of G' of width at most k (). We compute a tree decomposition (T, ι) of G of width at most $k + 1$ by *injecting* v into (T', ι') , that is, by adding v to all bags that contain u (). The resulting tree decomposition is most likely not optimal, but its width is bounded by a function in k . This decomposition can be used to compute an implicit representation of an optimal tree decomposition of G . This implicit representation is a binary tree T together with a collection \mathcal{P}_v of simple paths for every $v \in V(G)$ – that is, the vertices of V correspond to paths in T (). The paths are encoded as triples (x, y, v) with $x, y \in V(T)$, $v \in V(G)$, and with the natural meaning that there is a path in \mathcal{P}_v from x to y . Note that there can be multiple triples that start or end at some node x , but of course never more than $k + 1$. Given such an implicit representation, we can compute a tree decomposition of width k ().

Our plan for proving Theorem 120 is to implement the above sketched algorithm such that (i) we execute roughly $\log n$ rounds of it, and such that (ii) each round can be implemented by para-FAC-circuits of depth roughly $\log n$. The first item consists of reducing the instance to a smaller one by contracting *reduction partners*, which are pairs of vertices that are adjacent or twins. And, as usual in the design of parallel algorithms, we require *many* reduction partners that we can contract at once. The following fact due to Bodlaender and Hagerup guarantees that, in principle, there are always enough reduction partners. For that matter, let us call a vertex v *d-small* if we have $|N(v)| \leq d$.

► Fact 121 ([37])

Let $G = (V, E)$ be a graph with $\text{tw}(G) \leq k$, and let $d = 2^{k+4}(54k + 54)$ and $c = 1/(8(27k + 27)^2)$. Then there are at least $c|V|/2$ distinct pairs $\{u, v\}$ of d -small vertices that are reduction partners, that is, $\{u, v\} \in E$ or $N(u) = N(v)$. ◁

Unfortunately, the reduction partners provided by Fact 121 can still be in conflict. However, since all involved vertices are d -small, each pair can only be in conflict with a few other pairs. A maximal independent set in a corresponding conflict graph will, thus, equip us with enough *conflict free* reduction partners.

► Lemma 122

There is a computable and monotonically increasing function $g: \mathbb{N} \rightarrow \mathbb{N}$ such that there is a uniform family of FAC-circuits of depth $f(k) \cdot \log |V|$ and size $f(k) \cdot |V|^c$ that, on input of a graph $G = (V, E)$ and $k \in \mathbb{N}$, outputs a set I of $1/g(k) \cdot |V|$ pairs of vertices that can be contracted in parallel, or that concludes $\text{tw}(G) > k$.

Proof. Let $d = 2^{k+4}(54k+54)$ and $c = 1/(8(27k+27)^2)$. If $\text{tw}(G) \leq k$, then there are at least $c|V|/2$ distinct pairs $\{u, v\}$ of d -small reduction partners by Fact 121. Since a circuit of the desired size can check all pairs of vertices in parallel, it can compute a set S of reduction partners. Furthermore, since the circuit has logarithmic depth, it can check whether $|S| \geq c|V|/2$ holds – and it can report $\text{tw}(G) > k$ otherwise.

We cannot contract all pairs in S simultaneously, as pairs may share a vertex, may be adjacent, or may have a common neighbor. Since all these properties can easily be checked by an AC^0 -circuit, the circuit in construction can check for each pair of reduction partners if they are in conflict. By doing so, the circuit computes a *conflict graph* C whose vertex set is S and whose edges indicate conflicts. As the degree of each vertex appearing in a pair in S is bounded by d , the degree of C is bounded by a function in k , that is, by $g(k)$. Since each maximal independent set I in a graph of maximum degree Δ has size at least $|V|/(\Delta + 1)$, it is sufficient to use the reduction partners that constitute a maximal independent set in C . The circuit can compute such a set using Theorem 33. \square

Once we have a set of conflict free reduction partners, we can contract them to obtain a smaller graph. Via recursion, we will obtain an optimal tree decomposition for this reduced graph and, thus, our next task is to undo the contractions. We will do this in three steps: First we undo the contraction to obtain a tree decomposition that is not optimal (but which has a width that is still bounded by k). Then, secondly, we increase the width of this decomposition a little more, which in return allows us to make the tree decomposition balanced (and in particular such that the corresponding tree has depth $\log n$). Finally, we will compute a new and optimal tree decomposition. In order to execute the second step, we use the following fact from Elberfeld, Jacoby, and Tantau – but we have to be careful about the used *encoding*, as it differs from our standard encoding (Definition 4):

► Fact 123 ([75])

There are uniform families $(C_n^1)_{n \in \mathbb{N}}$, $(C_n^2)_{n \in \mathbb{N}}$, $(C_n^3)_{n \in \mathbb{N}}$ of FTC^0 -circuits that perform the following tasks:

- $(C_n^1)_{n \in \mathbb{N}}$ expects a graph G and a width- w tree decomposition (T, ι) in *term representation* as input, and outputs a width- $(4w+3)$ tree decomposition (T', ι') in *term representation* such that T' is a balanced binary tree;
- $(C_n^2)_{n \in \mathbb{N}}$ expects a tree T in *term representation* as input, and outputs it in *ancestor representation*;
- $(C_n^3)_{n \in \mathbb{N}}$ expects a tree T in *ancestor representation* as input, and outputs it in *term representation*. \triangleleft

The *term representation* of a tree is a string of brackets that encodes the ancestor relation of the tree. For instance, the term representation of $\begin{array}{c} \circ \\ \diagup \quad \diagdown \\ \circ \quad \circ \\ \diagup \quad \diagdown \quad \diagup \quad \diagdown \\ \circ \quad \circ \quad \circ \quad \circ \end{array}$ is the string $[[[]][[]]]$.

In contrast, the *ancestor representation* is a string consisting of a sequence of tuples $(v\#w)$ for all $v, w \in V(T)$ with v being an ancestor of w . Note that both encodings contain more information about reachability relations of vertices in the tree than the encoding of Definition 4. Therefore, it is not surprisingly that many problems on trees and forests that are L-complete when the input is given with the standard encoding [56], become solvable in TC^0 or NC^1 if the input is given in term or ancestor representation [75]. On the other hand, it is also easy to see that we can switch between the encoding of Definition 4 and the ancestor representation whenever we can answer reachability queries – which we can in AC^1 .

► Corollary 124

There is a uniform family of FAC^1 -circuits that, on input of a graph $G = (V, E)$ and a width- w tree decomposition (T, ι) of G , outputs a tree decomposition (T', ι') of width $4w + 3$ such that T' is a balanced binary tree. ◁

► Lemma 125

There is a uniform family of FAC -circuits of depth $f(k) \cdot \log |V|$ and size $f(k) \cdot |V|^c$ that, on input of a graph $G = (V, E)$, a set of conflict free pairs of vertices I , a graph $G' = (V', E')$ that is obtained from G by contracting the pairs in I , and a tree decomposition (T', ι') of G' of width k , outputs a balanced and nice tree decomposition (T, ι, η) of G of width at most $8k + 3$ and depth $(16k + 6) \cdot \log |V| + 1$.

Proof. Let (T', ι') be the given tree decomposition. An FAC^0 -circuit can compute (T, ι) by adding for each pair $\{u, v\} \in I$ the vertex v to every bag that contains u . This can be done in parallel for all vertices and all bags. Since the number of vertices in each bag is at most doubled, (T, ι) has width at most $2k$. This decomposition can be transformed into a balanced one of width at most $8k + 3$ by Corollary 124.

The last thing we have to do is to transform this decomposition into a nice decomposition (T, ι, η) . In order to do so, the circuit first adds an empty bag to each leaf, which is labeled as *leafnode*. Then, each node n with two children x and y is replaced by nodes $n, n_l,$ and n_r such that n_l, n_r are the children of n, x is a child of $n_l,$ and y a child of n_r . The node n is labeled as *join node*. This operation doubles the depth of the decomposition. Finally, for every node x with child y , the circuit in construction computes a chain of *forget nodes* from x to a new node z with $\iota(x) \cap \iota(y) = \iota(z)$, and a chain of *introduce nodes* from z to y . This will increase the depth of the decomposition at most by a factor of $8k + 3$.

Since making a balanced tree decomposition nice will result in a balanced decomposition again, the above algorithm produces a nice and balanced tree decomposition of width at most $2k$ and depth at most $(16k + 6) \log |V| + 1$. ◻

By putting all the previous results together we can compute a suboptimal tree decomposition of width, say, ℓ . Our task is to compute an optimal tree decomposition

of width k . The algorithm by Bodlaender and Hagerup [37] that we try to implement, as well as the famous linear time algorithm by Bodlaender [33], use the following algorithm due to Bodlaender and Kloks [40] as subroutine – and we will do the same.

► **Fact 126** (The Bodlaender–Kloks Algorithm. Implicit in [40], see also [37] for details.) There is a computable function $f: \mathbb{N} \rightarrow \mathbb{N}$ and an algorithm that, on input of a graph $G = (V, E)$, a width- ℓ nice tree decomposition (T, ι) of G , and an integer $k \in \mathbb{N}$, either detects $\text{tw}(G) > k$ or outputs a binary tree T' and a set $\{\mathcal{P}_v \mid v \in V(G)\}$ of collections of simple paths such that:

1. $|\mathcal{P}_v| > 0$ for all $v \in V(G)$;
2. $|\{v \mid \text{there is a } p \in \mathcal{P}_v \text{ with } n \in p\}| \leq k + 1$ for all $n \in V(T)$;
3. $\{n \mid \text{there is a } p \in \mathcal{P}_v \text{ with } n \in p\}$ is connected in T for all $v \in V(G)$;
4. the set $\{\mathcal{P}_v \mid v \in V(G)\}$ is encoded as set of triples (x, y, v) with $x, y \in V(T)$ and $v \in V(G)$;
5. the algorithm requires time $f(\ell)$ per node of T , and a node can be processed when his children have already been processed. \triangleleft

► **Lemma 127**

There is a uniform family of FAC-circuits of depth $f(k) \cdot \log |V|$ and size $f(k) \cdot |V|^c$ that, on input of a graph $G = (V, E)$, an integer k , and of a balanced and nice tree decomposition (T, ι, η) of G of width at most $\ell \leq f(k)$, outputs either $\text{tw}(G) > k$ or a width- k tree decomposition of G .

Proof. The circuit starts by either detecting $\text{tw}(G) > k$ or by computing a binary tree T' and a set $\{\mathcal{P}_v \mid v \in V(G)\}$ of collections of simple paths, encoded as set of triples (x, y, v) with $x, y \in V(T')$ and $v \in V(G)$. In order to do so, the circuit “bubbles up” the nice tree decomposition and spends $f(k)$ time on every node to simulate the algorithm from Fact 126. Since the depth of T is $f(k) \cdot \log |V|$, the desired circuit can implement this algorithm without modification.

If the algorithm has not reported $\text{tw}(G) > k$, the circuit has access to an optimal tree decomposition in the implicit form of a binary tree T' and a set of triples. Since the “real” tree decomposition we try to extract from this implicit representation uses the same tree T' the rest of the lemma boils down to the following algorithmic task: Given a tree $T' = (V, E)$ and three nodes $s, x, t \in V$, decide whether x lies on the unique path between s and t . This property can be expressed in monadic second-order logic with following formula, where $\varphi_{\text{connected}}(X)$ is a formula that expresses that the subgraph induced on X is connected (recall Example 14).

$$\begin{aligned} \varphi(s, x, t) = & \left[\exists P \cdot P(s) \wedge P(x) \wedge P(t) \wedge \varphi_{\text{connected}}(P) \right] \\ & \wedge \left[\forall P \cdot (P(s) \wedge \neg P(x) \wedge P(t)) \rightarrow \neg \varphi_{\text{connected}}(P) \right] \end{aligned}$$

Since T' is a tree (of treewidth 1), we can check the formula and, thus, decide the problem, in AC^1 [74]. \square

Proof of Theorem 120. The circuit first checks whether the size of the graph is bounded by k . If this is the case, an optimal tree decomposition can be computed via “brute-force.” Otherwise, the circuit computes a set of $1/f(k) \cdot |V|$ reduction pairs using Lemma 122, or concludes that the treewidth of G exceeds k . The circuit reduces G to G' by contracting the reduction pairs (the lemma guarantees that this is possible in parallel) and recursively computes a tree decomposition of G' . This tree decomposition can be transformed to a nice and balanced decomposition of G of width bounded by a function in k using Lemma 125. Finally, the circuit can reduce the width of the decomposition to k or conclude $\text{tw}(G) > k$ using Lemma 127.

Since Lemma 122 provides us with $1/f(k) \cdot |V|$ reduction pairs, $f(k) \cdot \log |V|$ rounds of the algorithm are sufficient to reduce the graph to a size depending only on the parameter. Considering each round as a subcircuit, each subcircuit has to execute the algorithms from the lemmas 122, 125, and 127. Since each lemma can be implemented in depth $f(k) \cdot \log |V|$, the complete circuit has a total depth of $f(k) \cdot \log^2 |V|$ and is, hence, a para-FAC²¹-circuit. Finally, the correctness follows from the correctness of the original algorithm [37]. \square

Note that this algorithm is, unfortunately, of pure theoretical interest – as are the algorithms of Bodlaender [33] and Bodlaender and Hagerup [37]. The reason is that the function f used by the Bodlaender–Kloks Algorithm (Fact 126) grows highly exponentially [40]. And indeed, attempts to turn Bodlaender’s algorithm into a practical implementation have revealed that this is rather hopeless [147]. However, by circumnavigating the Bodlaender–Kloks Algorithm by either skipping it, or by replacing it with heuristics, one obtains a general framework for treewidth heuristics [95]. Practical algorithms that compute optimal tree decompositions, however, require a fundamental different algorithmic strategy. We will discuss and analyze such strategies theoretically and practically in the second part of this thesis in Chapter 10.

8 PARALLEL PARAMETERIZED ALGORITHMIC META-THEOREMS

In the previous sections we developed many parallel parameterized algorithms for various problems. We did this on a case-by-case basis, that is, for every problem we carefully crafted a parallel algorithm for that particular problem and analyzed the correctness and the resource consumption of that particular algorithm. This has led to results such as $p_k\text{-VERTEX-COVER} \in \text{para-AC}^0$ (via the kernelization of Theorem 80). While this approach has the advantage that it gives us an algorithmic insight into the way parameterized problems can be solved in parallel, it has the drawback that we have to repeat the whole procedure for every “new” problem. For instance, if we just change the definition of VERTEX-COVER slightly such that we now ask to cover *at least* a given amount of edges (rather than *all*), the kernelization techniques used for VERTEX-COVER do not apply anymore.

► Problem 128 ($\text{PARTIAL-VERTEX-COVER}$)

Instance: A graph $G = (V, E)$ and two numbers $k, t \in \mathbb{N}$.

Question: Is there a set $S \subseteq V$ with $|S| \leq k$ and $|\{\{u, v\} \mid u \in S \vee v \in S\}| \geq t$? ◀

For the sole parameter k , this problem is known to be $W[1]$ -hard [102]. But this does, of course, not rule out an efficient parallel algorithm for the combined parameter $k + t$. However, instead of crafting *yet another vertex cover algorithm*, it would be preferable to derive such an algorithm from a general result that “just tells us” what the parallel complexity of $p_{k,t}\text{-PARTIAL-VERTEX-COVER}$ is.

To free us from the burden of crafting new algorithms on a case-by-case basis, we will use *algorithmic meta-theorems*. Such meta-theorems generally state that *all problems* that can be described in a *certain logic* can be solved in *some complexity class* for all instances with a *certain structure*. The most prominent example is Courcelle’s Theorem, which states that all problems expressible in monadic second-order logic can be solved in polynomial time on structures of bounded treewidth [57]. A bit more formally, we are interested in the parameterized complexity of the following decision problem for which we fix a logic \mathcal{L} :

► Problem 129 ($\text{MODEL-CHECKING}(\mathcal{L})$)

Instance: A relational structure S and an \mathcal{L} -formula φ .

Question: $S \models \varphi$? ◀

In all its generality, this problem is, of course, very difficult. Recall for instance the sentence $\varphi_{3\text{col}}$ from Section 2.3:

$$\varphi_{3\text{col}} = \exists R \exists G \exists B \left(\forall x R(x) \vee G(x) \vee B(x) \right) \wedge \left(\forall x \forall y E(x, y) \rightarrow \bigwedge_{C \in \{R, G, B\}} \neg C(x) \vee \neg C(y) \right).$$

It describes that a given graph has a proper coloring with three colors. Therefore, even if the size of φ is bounded by a constant and if we consider only monadic second-order logic, the model checking problem is already NP-hard. For a weaker logic – such as first-order logic – a natural parameter is the formula φ . However, the following sentence φ_{ds} for p_k -DOMINATING-SET already shows that the problem p_φ -MODEL-CHECKING(FO) is W[1]-hard:

$$\varphi_{\text{ds}} = \exists x_1 \dots \exists x_k \forall y \exists z \bigvee_{i=1}^k (x_i = y \vee (E(y, z) \wedge z = x_i)).$$

In order to obtain useful results, we will add a structural parameterization on the given relational structure S in addition to the formula φ . First, in Section 8.1, we use the maximum degree Δ of the Gaifman graph of S as a parameter. Later on, in Section 8.2, we use more restrictive parameterizations such as the treedepth or the treewidth of the Gaifman graph.

8.1 FIRST-ORDER MODEL CHECKING

Our first result considers the case that \mathcal{L} is the class of first-order formulas and that the maximum degree Δ of the Gaifman graph of the input structure is a parameter. Our main result is that this model checking problem lies in para-AC^0 :

- Theorem 130
 $p_{\varphi, \Delta}$ -MODEL-CHECKING(FO) \in para-AC^0

We rely strongly on a previous result by Flum and Grohe [84], who showed that this model checking problem lies in para-L (that is, it can be decided by a Turing machine that uses at most $f(k) + O(\log n)$ space), but we differ in three regards: First, we use color coding in our proof, which simplifies the argument somehow, second, we identify the parameterized distance problem on bounded degree graphs as the only part of the computation that is presumably not in para-AC^0 and, third, we observe that the degree of these graphs can be made a parameter and needs not be considered constant. Note that the result of Flum and Grohe, and the claim of Theorem 130 are incomparable in the sense that the relation of para-AC^0 and para-L is unclear. While para-AC^0 contains the parameterized distance problem (Theorem 39), it does certainly not contain undirected s - t -reachability. In contrast, para-L contains the undirected s - t -reachability problem due to Reingold's algorithm [143], but it does not contain the parameterized distance problem unless $\text{para}_\beta\text{-L} \subseteq \text{para-L}$ [154].

Proof of Theorem 130. Let φ be a formula given as input. For simplicity of presentation, we assume that the structure S is actually an undirected graph $G = (V, E)$ of maximum degree Δ . Let $d(a, b)$ denote the distance of two vertices in G and let $N_r(a) = \{b \in V \mid d(a, b) \leq r\}$ be the ball around a of radius r in G . By Gaifman's Theorem [94] we can rewrite φ as a Boolean combination of formulas of the following form:

$$\exists x_1 \cdots \exists x_k \left(\bigwedge_{i=1}^k \bigwedge_{j \neq i} \psi_{\text{dist} > 2r}(x_i, x_j) \wedge \bigwedge_{i=1}^k \psi(x_i) \right)$$

where $\psi_{\text{dist} > 2r}(x_i, x_j)$ is a standard formula expressing that $d(x_i, x_j) > 2r$ and ψ is r -local, meaning that for all $a \in V$ we have $G \models \psi(a) \iff G[N_r(a)] \models \psi(a)$. What remains is to determine whether there are k vertices a_1 to a_k in G such that the balls $N_r(a_i)$ do not intersect and $G[N_r(a_i)] \models \psi(a_i)$ holds for them.

We use color coding to determine the existence of such a_i . Let us introduce colors 1 to k . Since the maximum degree Δ is a parameter and r depends only on $|\varphi|$, the maximum size M of any $N_r(a)$ is bounded by the parameter. This means that there is a $(|V|, M \cdot k, k + 1)$ -universal coloring family that contains a coloring that colors all vertices of $N_r(a_i)$ with color i . In other words, there will be a coloring such that the balls are contained in monochromatic connected components of color i .

It remains to test whether for each color i there is a vertex a_i such that $N_r(a_i)$ has color i and $G[N_r(a_i)] \models \psi(a_i)$ holds. For this, let some candidate a_i be given. We need to determine for a given vertex b whether $d(a_i, b) \leq r$, where the distance is computed in the subgraph of G induced by the vertices of color i . Meaning, we need to solve parameterized distance problems parameterized by d (and Δ), which is possible in para-AC⁰ due to Theorem 39. Once the set $N_r(a)$ of vertices reachable from a vertex a in at most r steps has been determined, we can create an isomorphic copy of $G[N_r(a)]$ consisting just of a $|N_r(a)| \times |N_r(a)|$ adjacency matrix in para-AC⁰: Number the vertices of G in lexicographical order, which also induces an ordering on the vertices of $N_r(a)$. The entry in row i and column j of the matrix is a 1 if the i th and the j th vertex in $N_r(a)$ are connected by an edge in E . Determining which vertex is the i th vertex of $N_r(a)$ can be done by a para-AC⁰-circuit by Lemma 45.

Given the adjacency matrix of $G[N_r(a)]$, we can decide in para-AC⁰ whether we have $G[N_r(a)] \models \varphi$, since the size of $G[N_r(a)]$ depends only on the input parameters. \square

Equipped with Theorem 130 we can now provide an upper bound on the complexity of $p_{k,t}$ -PARTIAL-VERTEX-COVER:

► Lemma 131

$p_{k,t}$ -PARTIAL-VERTEX-COVER \in para-AC⁰

Proof. On input of a graph G , we first test whether there is a vertex of degree at least t . If so, we can accept since this vertex alone already constitutes the desired cover. Otherwise, we know that the *graph has a maximum degree bounded by the parameter*, and we can apply Theorem 130 to the following first-order formula, which depends only on k and t :

$$\underbrace{\exists x_1 \cdots \exists x_k}_{\text{the size-}k \text{ cover}} \underbrace{\exists a_1 \exists b_1 \cdots \exists a_t \exists b_t}_{\text{the } t \text{ covered edges}} \left(\varphi_{\text{dist}}(a_1, b_1, \dots, a_t, b_t) \wedge \bigwedge_{i=1}^t (E(a_i, b_i) \wedge \bigvee_{j=1}^k a_i = x_j) \right).$$

Here, φ_{dist} is a standard formula expressing that $\{a_1, b_1\}, \dots, \{a_t, b_t\}$ are distinct sets. □

8.2 SECOND-ORDER MODEL CHECKING

In the second part of this chapter, we switch from first-order logic to second-order logic. This will, of course, increase the complexity of the model checking problem, as this logic is more powerful. To compensate this circumstance, we have to use structural parameters that are stronger than the maximum degree. The vertex cover number, the treedepth, and the treewidth of the Gaifman graph of the input structure will fill this role. In detail, we will prove the following three theorems in the course of this section:

► Theorem 132

$p_{\varphi, \text{vc}}$ -MODEL-CHECKING(MSO) \in para-AC⁰

► Theorem 133

$p_{\varphi, \text{td}}$ -MODEL-CHECKING(MSO) \in para-AC⁰

► Theorem 134

$p_{\varphi, \text{tw}}$ -MODEL-CHECKING(MSO) \in para-AC²

These results are a parallel version of the theorem of Courcelle. Similar work was done by Elberfeld, Jakoby, and Tantau, who showed these result for *constant* treedepth or treewidth and a constant sized formula [74, 75]. An interesting observation is that in the constant case, the parameters vertex cover number and treedepth coincide – in both cases the corresponding model checking problem is in AC⁰. In the parameterized setting, however, a difference in their complexity is revealed.

The standard approach to solve problems on structures of small treewidth is to apply dynamic programming on a tree decomposition of the structure (or rather of its Gaifman graph) while using the fact that graphs of small treewidth have small balanced separators [59]. We applied such a technique in Section 5.2, where we showed that we can find a homomorphism from a graph H to another graph G efficiently if H is small and has constant treewidth. The main ingredient to prove the aforementioned theorems is to show that such a strategy works whenever the problem can be defined in MSO-logic. Before we jump directly into the proof, it will be helpful to develop some intuition about dynamic programming on tree decompositions.

We will, for now, assume that we are already given a tree decomposition (T, ι) , and we assume that this decomposition is very nice. Recall that this means that T is a rooted binary tree, and that we are additionally given some labeling function of its nodes $\eta: V(T) \rightarrow \{\text{leaf}, \text{introduce}, \text{join}, \text{forget}, \text{edge}\}$. We also consider the children of a node to be ordered, for instance by the lexicographical order of $V(T)$. A dynamic program on (T, ι) is just a run of a (nondeterministic) tree automaton: The mapping ι can be seen as a function that maps the nodes of T to symbols from some alphabet Σ . A naive approach to manage ι would yield a huge alphabet (depending on the size of the graph). We thus define the so called *tree-index*, which is a map $\text{idx}: V(G) \rightarrow \{0, \dots, \text{tw}(G)\}$ such that no two vertices that appear in the same bag share a common tree-index. The existence of such an index follows from the property that every vertex is forgotten exactly once: We can traverse T from the root to the leaves and assign a free index to a vertex v when it is forgotten; we release the used index once we reach an introduce bag for v . The symbols of Σ then only contain the information for which tree-index there is a vertex in the bag. From a theoretician's perspective this means that $|\Sigma|$ depends only on the treewidth; from a programmer's perspective the tree-index makes it easier to manage data structures for the dynamic program (this will be discussed further in the second part of this thesis).

► Definition 135 (Tree Automaton)

A nondeterministic bottom-up *tree automaton* is a tuple $A = (Q, \Sigma, \Delta, F)$ where Q is a set of *states* with a subset $F \subseteq Q$ of *accepting states*, Σ is a non-empty *alphabet*, and $\Delta \subseteq (Q \cup \{\perp\}) \times (Q \cup \{\perp\}) \times \Sigma \times Q$ is a *transition relation* in which $\perp \notin Q$ is a special symbol to treat nodes with less than two children. The automaton is *deterministic* if for every $x, y \in Q \cup \{\perp\}$ and every $\sigma \in \Sigma$ there is exactly one $q \in Q$ with $(x, y, \sigma, q) \in \Delta$. ◁

► Definition 136 (Computation of a Tree Automaton)

The *computation of a tree automaton* $A = (Q, \Sigma, \Delta, F)$ on a labeled tree (T, ι) with $\iota: V(T) \rightarrow \Sigma$ and root $r \in V(T)$ is an assignment $q: V(T) \rightarrow Q$ such that for all $n \in V(T)$ we have (i) $(q(x), q(y), \iota(n), q(n)) \in \Delta$ if n has two children x, y ; (ii) $(q(x), \perp, \iota(n), q(n)) \in \Delta$ or $(\perp, q(x), \iota(n), q(n)) \in \Delta$ if n has one child x ; and (iii) $(\perp, \perp, \iota(n), q(n)) \in \Delta$ if n is a leaf. The computation is *accepting* if $q(r) \in F$. ◁

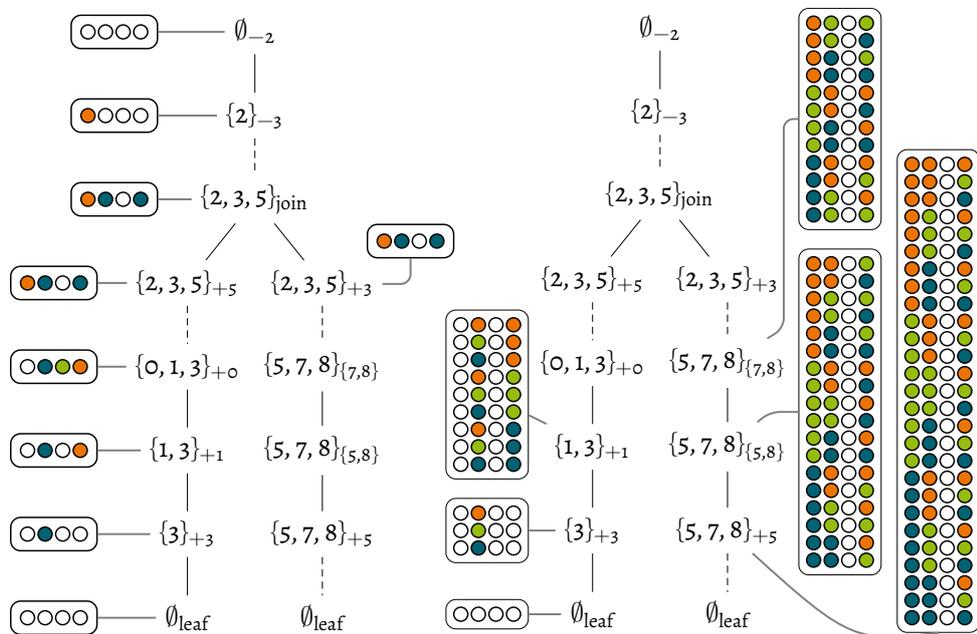
► Example 137 (A Dynamic Program for Graph Coloring)

We want to decide whether a given graph $G = (V, E)$ can be colored with three colors. Given a very nice tree decomposition (T, ι, η) , a nondeterministic tree automaton can process T as follows: On introduce-bags, the automaton *guesses* a color for the introduced vertex; on forget-bags the automaton clears the information for the removed vertex from its current state; at edge-bags the automaton *rejects* in case the two endpoints of that edge have the same color in the current state of the automaton; and, finally, in join-bags the automaton *rejects* when it was in different states for the two children. In case the automaton reaches the root of T without rejecting, it will accept T . Using the properties of a tree decomposition, it is easy to check that G is indeed colorable with three colors in this case.

The following figure illustrate a run of the automaton. The *left* figure shows a part of a tree decomposition of the grid graph $\begin{matrix} \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ \end{matrix}$ with vertices $\{0, \dots, 8\}$. The index of a bag shows the type of the bag: a positive sign means “introduce,” a negative one “forget,” a pair represents an “edge”-bag, and text is self-explanatory. Solid lines represent real edges of the decomposition, while dashed lines illustrate a path (that is, there are some bags skipped). On the left branch of the decomposition a run of a nondeterministic tree automaton for 3-COLORING is illustrated for the tree-index:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 3 & 0 & 1 & 2 & 3 & 0 & 1 & 0 \end{pmatrix}.$$

To increase readability, states of the automaton are connected to the corresponding bags with gray lines, and for some nodes the states are omitted. In the *right* figure, the same automaton is simulated deterministically.



◁

Given some intuition about dynamic programming on tree decompositions and tree automata, we can now turn to prove Theorem 132, 133, and 134. The classical way of proving variants of Courcelle's Theorem is as follows: On input of a structure S and an MSO-formula φ , we compute a tree decomposition (T, ι) of S . This tree decomposition is then translated into an s -tree-structure \mathcal{T} and φ is translated to a new MSO-formula ψ such that $S \models \varphi \Leftrightarrow \mathcal{T} \models \psi$. To decide $\mathcal{T} \models \psi$, the s -tree-structure \mathcal{T} is transformed into a labeled tree (\mathfrak{T}, λ) and ψ is turned into a tree automata \mathfrak{A} such that we have $\mathcal{T} \models \psi \Leftrightarrow (\mathfrak{T}, \lambda) \in L(\mathfrak{A})$. An s -tree-structure is a structure $\mathcal{T} = (V, E^{\mathcal{T}}, P_1^{\mathcal{T}}, \dots, P_s^{\mathcal{T}})$ over the signature $\tau_{s\text{-tree}} = (E^2, P_1^{\mathcal{T}}, \dots, P_s^{\mathcal{T}})$ where $(V, E^{\mathcal{T}})$ is a directed tree.

If we would aim for a class like FPT, we could directly implement the sketched strategy. But since we aim for parallel classes, things are a bit more tricky. The constructed tree decomposition could be degenerated (for instance, T could actually be a path), which makes it difficult to simulate a tree automaton in parallel. In order to keep the tree flat, we will have to drop our assumption that our decomposition is nice, in fact, we will need nodes in T that have high degree. This, unfortunately, requires us to step away from classical tree automata and to step towards the (slightly more technical) multiset automata introduced by Elberfeld, Jakoby, and Tantau [75].

A *multiset* M is a set S together with a multiplicity function $\#_M: S \rightarrow \mathbb{N}$. The *multiplicity* of M is the value $\max_{e \in S} \#_M(e)$. We denote by $\mathcal{P}_\omega(S)$ the class of all multisets of S , and by $\mathcal{P}_m(S)$ the class of all multisets of multiplicity at most $m \in \mathbb{N}$ of S . Notice that $\mathcal{P}_1(S)$ is just the standard power set of S . We define for a multiset $M \in \mathcal{P}_\omega(S)$ and a number $m \in \mathbb{N}$ the *capped version* $M|_m$ of M by setting the multiplicity to $\#_{M|_m}(e) = \min(\#_M(e), m)$ for all $e \in S$.

► Definition 138 (Multiset Automaton)

A *nondeterministic bottom-up multiset automaton* is a tuple $\mathcal{A} = (\Sigma, Q, Q_\alpha, \Delta, m)$ consisting of an *alphabet* Σ , a *state set* Q with *accepting states* $Q_\alpha \subseteq Q$, a *state transition relation* $\Delta \subseteq \Sigma \times \mathcal{P}_m(Q) \times Q$, and a *multiplicity bound* $m \in \mathbb{N}$. The automaton is *deterministic* if for every $\sigma \in \Sigma$ and every $M \in \mathcal{P}_m(Q)$ there is exactly one $q \in Q$ with $(\sigma, M, q) \in \Delta$. ◁

► Definition 139 (Computation of a Multiset Automaton)

Let (\mathfrak{T}, λ) be a labeled tree, where $\lambda: V(\mathfrak{T}) \rightarrow \Sigma$ is the labeling function, and let $\mathfrak{A} = (\Sigma, Q, Q_\alpha, \Delta, m)$ be a multiset automaton. A *computation* of \mathfrak{A} on (\mathfrak{T}, λ) is a partial assignment $q: V(\mathfrak{T}) \rightarrow Q$ such that for every node $n \in V(\mathfrak{T})$ for which $q(n)$ is defined, we have that (i) the value $q(c)$ is defined for each child c of n in \mathfrak{T} and (ii) for the multiset $M = \{q(c) \mid c \text{ is a child of } n\}$ we have $(\lambda(n), M|_m, q(n)) \in \Delta$. A computation is *accepting* if $q(r) \in Q_\alpha$ holds for the root node r of \mathfrak{T} . The *tree language* $L(\mathfrak{A})$ contains all labeled trees accepted by \mathfrak{A} . ◁

► Fact 140 ([75])

The following statements hold and are constructive:

1. For all multiset automata \mathfrak{A} and \mathfrak{B} there is another multiset automaton \mathfrak{C} with $L(\mathfrak{C}) = L(\mathfrak{A}) \cap L(\mathfrak{B})$;
2. For every nondeterministic multiset automaton \mathfrak{A} there is a deterministic multiset automaton \mathfrak{B} with $L(\mathfrak{A}) = L(\mathfrak{B})$;
3. For every multiset automaton \mathfrak{A} there is a multiset automaton \mathfrak{B} accepting the complement of $L(\mathfrak{A})$. \triangleleft

Fortunately for us, many steps of the translation of the model checking problem to the evaluation of a multiset automaton work in the same way for structures of bounded treewidth as for such with constant treewidth. This is reflected by the following fact that we can use:

► Fact 141 (Implicit in [75])

There are functions h_1, h_2, h_3 , and h_4 performing the following mappings:

1. The input for h_1 are a structure S together with a width- w tree decomposition (T, ι) of S and an MSO-formula φ . The output is an s -tree-structure \mathcal{T} .
2. The input for h_2 are an MSO-formula φ and a tree width w . The output is an MSO-formula ψ .
3. The input for h_3 are an s -tree-structure \mathcal{T} and an MSO-formula ψ . The output is a labeled tree (\mathfrak{T}, λ) of the same depth.
4. The input for h_4 is an MSO-formula ψ . The output is a multiset automaton \mathfrak{A} .

The following holds for the values computed by these functions:

$$S \models \varphi \iff \mathcal{T} \models \psi \iff (\mathfrak{T}, \lambda) \in L(\mathfrak{A}).$$

All h_i are computable and h_1 and h_3 are even computable by uniform FAC-circuits of depth $O(1)$ and size $f(\varphi, w) \cdot |S| \cdot |T|$. \triangleleft

To prove Theorem 132, 133, and 134, we are essentially left with the task of simulating a multiset automaton (whose size is bounded by a function in the parameter). The following lemma will serve as workhorse in all three cases:

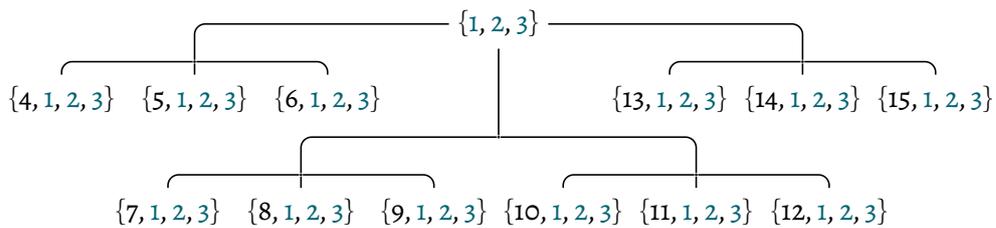
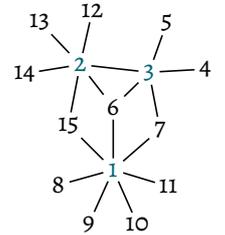
► Lemma 142

Let S_d be the set of labeled trees (\mathfrak{T}, λ) of maximum depth d . There is a uniform family of FAC-circuits of depth $O(d)$ and size $f(|\mathfrak{A}|) \cdot |\mathfrak{T}|^c$ that, on input of a labeled tree $(\mathfrak{T}, \lambda) \in S_d$ and a multiset automata $\mathfrak{A} = (\Sigma, Q, Q_a, \Delta, m)$, decides whether or not $(\mathfrak{T}, \lambda) \in L(\mathfrak{A})$ holds.

Proof. Since the size of the circuit depends on the size of \mathfrak{A} by an arbitrary computable function f , we can assume that \mathfrak{A} is deterministic, as otherwise we can compute an equivalent deterministic automaton in a constant number of AC-layers using Fact 140. The circuit has d “main”-layers, each of which consists of circuits of constant depth. The i th layer will assign states to the nodes of the $(d - i)$ th layer of \mathfrak{T} . The first layer simply assigns states to the leaves of \mathfrak{T} . Layer i then has access to the assigned states of layer $i - 1$. In order to compute the state $q(n)$ for a node n , the circuit computes the multiset $M = \{q(c) \mid c \text{ is a child of } n\}$ using the result of the last layer. Now the circuit has to cap M to compute $M|_m$. In order to do so, the circuit has to count up to m . Since we have $m \leq |\mathfrak{A}|$, the value m is bounded by the parameter and, therefore, we can use the para-AC⁰-circuit from Lemma 45. Once $M|_m$ is computed, the circuit can compute $q(n)$ by a lookup of $(\lambda(n), M|_m)$ in the description of Δ . The circuit outputs 1 if, and only if, after the evaluation of the d layers the root r of \mathfrak{T} is assigned with $q(r) \in Q_a$. \square

There are three special cases of Lemma 142 for the simulation of multiset automata, which can be performed (i) in para-FAC⁰ for trees of constant depth, (ii) in para-FAC⁰ for trees of depth bounded by the parameter, and (iii) in para-AC¹ for balanced trees of logarithmic depth. In all cases, the size of the automaton is the parameter.

Proof of Theorem 132. On input of a structure S and an MSO-formula φ , a para-AC⁰ can approximate a vertex cover X of the Gaifman graph of S of size at most $k^2 + 2k$ using Lemma 86 in combination with Lemma 81. From a given vertex cover X , it is easy to construct a tree decomposition (T, ι) of width at most $|X|$ and depth 2: The root bag r contains the set X , and for every $v \in V \setminus X$ there is one child of r containing $X \cup \{v\}$. For instance, consider the graph at the margin and the vertex cover X . A corresponding tree decomposition is illustrated below:



Given the tuple $(S, (T, \iota), \varphi)$, the circuit in construction computes a labeled tree (\mathfrak{T}, λ) and a multiset automaton \mathfrak{A} using Fact 141. The depth of the tree \mathfrak{T} is bounded by the depth of (T, ι) and, hence, is bounded by a constant. Furthermore, we have $|\mathfrak{A}| \leq f(|\psi| + |X|)$ for some computable function f . Hence, a para-AC⁰-circuit can invoke Lemma 142 and output the result. \square

Proof of Theorem 133. As Theorem 132, but (T, ι) is computed using Theorem 119. \square

Proof of Theorem 134. The proof is almost identical to the proof of Theorem 132. On input of a structure S and an MSO-formula φ , a para-AC^{2t}-circuit computes a tree decomposition (T, ι) of the Gaifman graph of S using Theorem 120. However, the circuit can not directly invoke Lemma 142 as the depth of T is not bounded. This can be overcome as follows: Let the width of (T, ι) be w , then an FAC¹-circuit can compute a balanced tree decomposition of logarithmic depth and width at most $4w + 3$ using Corollary 124. Given this decomposition, we proceed as in Theorem 132. \square

9 OUTLOOK AND FURTHER DIRECTIONS

In the first part of this thesis we studied parallel parameterized algorithms and have explored parameterized circuit complexity. We started by providing a general toolbox of basic algorithms, which I hope will prove useful in the design of further parallel parameterized algorithms. It has turned out that the fascinating technique of color coding is perfectly suited to develop parameterized parallel constant time algorithms and it lies at the heart of many of our results.

- ⊢ *Open Problem:* Since we used the technique so heavily, it would be interesting for practical purposes to study whether it can be implemented by circuits that are asymptotically smaller (but perhaps a little deeper) than the current ones. ⊣
- ⊢ *Open Problem:* It should be explored in which particular cases we can use color coding to parallelize algorithms – Till Tantau and myself have made first steps in this direction by investigating the descriptive complexity of color coding [23]. ⊣
- ⊢ *Open Problem:* In contrast, it would be interesting to have other techniques for parallel parameterized constant time algorithms. In particular: Are there natural problems in para-AC° that do not require color coding for solving them? ⊣

After the warm-up, we dealt with parallel bounded search trees and parallel kernelizations in Chapter 5 and 6. For the bounded search trees we provided a general framework in the form of parallel algorithms for certain modulator problems. It should be easy to extend this approach in further directions, as almost any search tree can (theoretically) be evaluated in parallel.

- ⊢ *Open Problem:* The parallelization of search trees differs a lot in theory and practice. Algorithm engineering is therefore a promising direction to study the parallelization of such search trees. In this setting, the notation of *work optimal* algorithms (roughly: parallel algorithms that do not perform more computational steps than their sequential counterpart) becomes relevant. Skambath, Tantau, and myself have made first steps in this direction – primarily with vertex cover in mind [18]. ⊣

We translated the famous allegory that “kernelization is the same as fixed-parameter tractability” to the parallel setting: “parallel preprocessing equals fast parallel parameterized algorithms.” In this regard, we observed complex trade-offs between kernel size and the depth of the circuits needed to compute them. On the negative side, we proved that the problem of finding a maximum matching in a graph turns out to be an obstacle in the parallel computation of kernels, for instance for a linear kernel for p_k -VERTEX-COVER.

- ⊢ *Open Problem:* It is desirable to find alternative kernelizations for these problems. Even without improving the best known kernel bounds, we could improve the kernel size reachable in parallel by circumnavigate the matching problem. ⊣

We concluded the chapter about parallel kernelization with a constant-time kernelization for $p_{k,d}$ -HITTING-SET. This was a rather surprising result, as the problem is very general and its standard kernelization is very sequential. In fact, we did require heavy machinery and a lot of color coding to solve it in constant-time. The reward is a powerful tool that can be used to place many natural problems in para-AC⁰.

- ⊢ *Open Problem:* Are there further applications that obtain fast parallel parameterized algorithms by using $p_{k,d}$ -HITTING-SET as a subroutine? A good starting point in this direction is a result by Chen, Flum, and Huang, who have shown that certain weighted Fagin definable problems can be reduced to it [53]. ⊣

We ended the first part by investigating the parallel decomposition of graphs and logical structures in Chapter 7, and by afterwards implementing meta-theorems on top of these decompositions in Chapter 8. In particular, we studied the parallel complexity of computing crown- and treedepth decompositions, and we analyzed the precise circuit-complexity of a parallel algorithm by Bodlaender and Hagerup [37] to compute exact tree decompositions: it was para-FAC²!

- ⊢ *Open Problem:* Is it possible to compute the treewidth of a graph in para-FAC²? The best lower bound that is currently known is only para-L, the parameterized version of logspace. ⊣

- ⊢ *Open Problem:* Is there a parallel parameterized approximation for the treewidth problem in para-FAC² or below? In the light of a recent result by Fomin, Lokshtanov, Pilipczuk, Saurabh, and Wrochna [89], who have established a polynomial time k^2 -approximation for treewidth, it is interesting to ask whether it is possible to obtain a k^c -approximation within NC. ⊣

In terms of meta-theorems, we examined parallel model checking algorithms for first-order logic on structures of bounded degree, and for monadic second-order logic on structures of bounded treedepth or treewidth. These results quite naturally match their non-parameterized counter parts due to Seese [151] and Elberfeld, Jakoby, and Tantau [74, 75]. It is interesting that the bottleneck for the monadic second-order model checker is the algorithm to compute the tree decomposition, which is currently more expensive than the evaluation of the tree automaton.

- ⊢ *Open Problem:* A promising further task is to extend these results to other logical structures. A first attempt was provided by Pilipczuk, Siebertz, and Toruńczyk, who have shown that model checking for first-order logic on sparse structures can be performed within para-AC¹ [140]. ⊣

Besides the fundamental directions in which we have explored the topic in this part of the thesis, there are many further paths that can be investigated in the light of parameterized circuit complexity. For instance, the textbook application for circuit complexity are lower bounds, and it is thus natural to ask whether we can establish parameterized circuit lower bounds. First progress in this direction was made by Chen and Flum [52].

An interesting further application of parameterized complexity is to tackle problems whose classical complexity is still not fully resolved. In this matter, we have shown that the problem of finding a matching parameterized by the solution size is in para-AC° (while the classical problem is not known to be in NC). Das, Enduri, and Reddy have studied the parameterized parallel complexity of the graph isomorphism problem [61] (for which the exact complexity is not resolved either) by different parameterizations like the vertex cover number or the size of a feedback-vertex set. It would be gripping to further investigate parallel algorithms in this direction with weaker parameterizations.

Part II

Towards Practice and Back

In this second part of the thesis, we will turn from complexity theory and algorithm design to algorithm engineering. We will explore two libraries, called *Jdrasil* and *Jatatosk*, that I have developed during my time as a doctoral student. The former is a tool to compute optimal tree decompositions, while the later is a model checker for a fragment of monadic second-order logic. Both tools are publicly available [12, 15].

Preliminary versions of many results of this part were previously presented at the following conferences:

- [16] Max Bannach, Sebastian Berndt, and Thorsten Ehlers: *Jdrasil: A Modular Library for Computing Tree Decompositions*. In Proceedings of the 16th International Symposium on Experimental Algorithms (SEA 2017).
- [13] Max Bannach and Sebastian Berndt: *Practical Access to Dynamic Programming on Tree Decompositions*. In Proceedings of the 26th Annual European Symposium on Algorithms (ESA 2018).
- [14] Max Bannach and Sebastian Berndt: *Positive-Instance Driven Dynamic Programming for Graph Searching*. In Proceedings of 16th Algorithms and Data Structures Symposium (WADS 2019).

The second paper was awarded *Best Student Paper* at the European Symposium on Algorithms 2018.

10 JDRASIL: A MODULAR LIBRARY FOR COMPUTING TREE DECOMPOSITIONS

In this chapter we will tackle the problem of computing optimal tree decompositions in *practice*, that is, for real world graphs on a real machine. For that end, I will present the library *Jdrasil* and, in the light of this thesis, will highlight its parallel capabilities.

Jdrasil – the name is a portmanteau of “Java” and “Yggdrasil,” a gigantic tree in Norse mythology – is a Java library that is capable of computing tree decompositions both, exactly and heuristically. The goal of *Jdrasil* is to allow other projects to add tree decompositions to their applications as easily as possible. In order to achieve this, *Jdrasil* is designed in a modular way: Every algorithm is implemented as interchangeably as possible. At the same time, algorithms are implemented in a clean object oriented manner, hopefully making it easy to understand and extend the implementation.

Due to its modularity, *Jdrasil* has many facets, some of which require new theory while others are direct implementations of facts that are already known in the literature. This chapter is neither a complete scan through all the features of the library, nor a tutorial for it. Both can be found in *Jdrasil*’s manual in its publicly available GitHub repository [15]. Instead, I will present the design philosophy behind the library in Section 10.1 and provide a high-level view on the library in Section 10.2.

After this insight into the library, we will study some concrete subroutines of *Jdrasil*. I have decided to present two exact algorithms that are implemented in the library. Both perform very well and both require novel ideas from theory. We will start in Section 10.3 with a SAT-encoding of treewidth, which is based on an encoding by Berg and Jarvisalo [26], but adds new ideas to it. As we will see, these ideas improve the performance of the encoding noticeably. Afterwards, in Section 10.4, I present a novel algorithm for computing tree decompositions that was developed by Hisao Tamaki [156, 158] in a game theoretic characterization due to Sebastian Berndt and myself [14]. This algorithm is the currently fastest algorithm in practice [63, 64].

We will study the parallel capabilities of *Jdrasil* in Section 10.5. Due to its modularity, the parallelization in *Jdrasil* is implemented in a coarse and general way: Independently of the used subroutine, *Jdrasil* will automatically identify parts of the graph that can be decomposed in parallel and apply the subroutine to it.

We close the chapter in Section 10.6 with a series of experiments to compare and analyze the algorithms presented within this chapter.

10.1 THE DESIGN PHILOSOPHY OF JDRASIL

When one starts the development of a library that solves a combinatorial problem, one is confronted with general design decisions that one has to make beforehand. Of course, developing a library like *Jdrasil* for computing tree decompositions is no exception. The two most important decisions for *Jdrasil*, which to some extent define the “spirit” of *Jdrasil*, concern the *level of optimization* and the *level of parallelization* we aim for.

With the *level of optimization* we mean to which extent we optimize the implementation of subroutines. One plausible route for a library that computes exact tree decompositions would be the development of a “simple” program that just expects a graph as input and outputs a tree decomposition. Such a program would essentially implement one algorithm and optimize it “to the bone,” for instance by optimizing the implementation for specific architectures. However, when I started the development of *Jdrasil* I noticed that, at this time, it was pretty unclear which algorithm for treewidth would work best in practice. Therefore, I decided against this route. Instead, *Jdrasil* is designed as a modular library – meaning that in *Jdrasil* many algorithms are implemented and exchangeable. The library was developed in a way that makes it easy to extend it by further algorithms, without caring about representation of data, preprocessing, or output. Furthermore, the design of *Jdrasil* as library allowed me to add many more features – such as an easy interface for dynamic programs over tree decompositions.

When thinking about parallelization in practice, there are multiple *levels of parallelization* that we could apply. For instance, we could aim for a very fine level of parallelization by designing an algorithm directly for, say, FPGAs. However, considering the mentioned initial situation, this seems a little premature. Alternatively, we could aim for a medium level of parallelization by designing algorithms that are capable of using many CPU-cores or that work directly on the GPU. The first iteration of the Parameterized Algorithms and Computational Experiments Challenge (PACE) originally contained a parallel track to feature such algorithms. This track was discarded as the best sequential implementation was significantly faster than the best parallel one [63]. This sequential implementation is based on a novel algorithm due to Hiso Tamaki [156]. Recently, there were attempts of computing tree decompositions on the GPU, but the resulting implementation was outperformed by Tamaki’s algorithm as well [163]. Instead of aiming for a fine or middle level of parallelization, we will therefore be content with a course level of parallelization. This means that in *Jdrasil* we do not directly parallelize any subroutine, but instead identify parts of the graph that can be decomposed in parallel. This favors the modular architecture of *Jdrasil*: Independently of the subroutine used to decompose the graph (it may even be a subroutine implemented by the user), *Jdrasil* will automatically utilize parallel architectures by applying the subroutine to many parts of the input graph.

10.2 A HIGH-LEVEL VIEW ON THE LIBRARY

On the most abstract layer, the user can use `Jdrasil` to compute a tree decomposition without knowing anything about the used algorithms and the process as a whole. The following code snippet is the easiest way to compute and output a tree decomposition for the input graph \triangleleft using `Jdrasil`. Here, the `SmartDecomposer` is a class that encapsulates the whole modularity of `Jdrasil` to make it easy to gather some first results. However, the real strength of the library lies in its modularity, which we will discuss afterwards.

```
import jdrasil.graph.*;
import jdrasil.algorithms.*;

public class Main {
    public static void main(String[] args) {

        // create a new empty graph with integer vertices
        Graph<Integer> G = GraphFactory.emptyGraph();

        // make it a triangle
        for (int v = 1; v <= 3; v++) { G.addVertex(v); }
        G.addEdge(1, 2);
        G.addEdge(2, 3);
        G.addEdge(3, 1);

        // output graph in PACE format
        System.out.println(G);

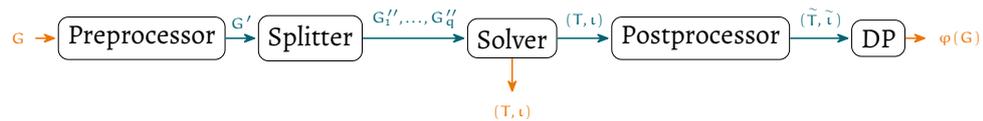
        // compute tree decomposition
        TreeDecomposition<Integer> td = null;

        try {
            td = new SmartDecomposer<Integer>(G).call();
        } catch (Exception e) {
            // something went wrong
        }

        // ouput tree decomposition in PACE format
        System.out.println(td);
    }
}
```

Let us discuss what the `SmartDecomposer` tries to hide from us, and why we refer to `Jdrasil` as a “modular library” – and what this term actually means. In the design process of the library, I have tried to abstract the workflow of computing a tree decomposition (and working with it) from a software engineering point of view. That is, given a graph G and the description of some problem we want to solve on it, what are the steps a program has to run through in order to compute a tree decomposition and solve the problem with it? As usual in software design, I wanted these steps to be defined fine enough to have small and self-contained tasks, but coarse enough to make a useful abstraction and to make them interchangeable.

The result is the following *abstract pipeline* that is used by *Jdrasil* to process an input graph G and to output either a tree decomposition (T, ι) of G or the image $\varphi(G)$ of some user specified function φ . The black boxes are the different interfaces that are involved in the computation of a tree decomposition; the **orange part** describes the input and output; the **blue part** highlights the information flow within the program.



The *Preprocessor* is the first instance that obtains the input graph G . Its task is to identify “easy” parts of the input, which we may safely remove to obtain an equivalent instance G' . Later on, a preprocessor may construct a tree decomposition for G from a tree decomposition of G' . A typical example is the removal of attached trees or the contraction of degree-2 vertices. The standard implementation in *Jdrasil* follows the description by Bodlaender, Koster, and Eijkhof [43] and we will, thus, not discuss it further within this thesis. The following code illustrates the usage of a preprocessor within *Jdrasil*:

```

// generate instance of the preprocessing algorithm
GraphReducer<T> reducer = new GraphReducer<>(G);

// get the preprocessed graph
Graph<T> H = reducer.getProcessedGraph();

// add the decomposition of H
TreeDecomposition<T> td = ...
reducer.addbackTreeDecomposition(td);

// we can now access the final decomposition of the original graph
reducer.getTreeDecomposition();

```

The *Splitter* interface is modeled after the concept of so called *safe separators* by Bodlaender and Koster [41]. In essence, a safe separator allows to split a graph G into multiple graphs G_1, \dots, G_q such that one can obtain an optimal tree decomposition for G by gluing tree decompositions of these graphs. The simplest safe separator is the empty set, which splits the graph into its connected components. It is notable that this concept was underestimated (and thus not used) by almost all participants for PACE 2016, and that it therefore boosted the performance of many submissions for the PACE 2017. In the light of this thesis the splitter interface is especially interesting, as it is an access point for parallelization in practice: *Jdrasil* uses the parallel capabilities of Java to *automatically parallelize* the computation of a tree decomposition if possible. We will give precise definitions in Section 10.5. The following code on the next page shows how to invoke a splitter:

```

// create a splitter for the graph G
GraphSplitter<T> splitter = new GraphSplitter<T>(G, H -> {

    // create a tree decomposition of the atom H
    TreeDecomposition<T> td = ...
    return td;

},lb); // lb is a lower bound on the treewidth of G

// obtain a tree decomposition of G
// this will invoke the splitting process
TreeDecomposition<T> result = splitter.call();

```

The *Solver* interface is at the very heart of the pipeline. Its purpose is quite obvious: Given a graph G , output a tree decomposition (T, ι) of it. Since all other parts of the pipeline are *safe* with respect to treewidth, the solver determines the quality of the tree decomposition. For an exact algorithm, the whole pipeline will produce an optimal decomposition. However, the solver may also be a heuristic or an approximation algorithm. In its current state, *Jdrasil* contains many standard algorithms, for instance the heuristics proposed by Bodlaender and Koster [42], the classical approximation algorithm due to Robertson and Seymour [59, 85], the branch-and-bound algorithm by Gogate and Dechter [99], and many more. Particularly successful was a SAT-based approach, which we will discuss in detail in Section 10.3. It should be noted that, within *Jdrasil*, a solver will usually expect the “hard” core of the problem and, thus, does not invoke any checks for simple solutions – for instance, a solver usually handles a huge tree in the same way as a small structured graph. Therefore, it is essential to apply a preprocessor beforehand. The following code illustrates the usage of a (here SAT-based) solver:

```

// create a SAT-based decomposer for the graph G
TreeDecomposer<T> decomposer = new SATDecomposer<>(G, Encoding.IMPROVED);

// invoke the computation and obtain the tree decomposition
TreeDecomposition<T> td = decomposer.call();

```

A *postprocessor* obtains a tree decomposition (T, ι) and prepares it for a subsequent task. A prominent example is the transformation of (T, ι) into a nice tree decomposition. However, a postprocessor may also improve a non-optimal tree decomposition, for instance with the refinement technique [42] or via local improvements [83]. Another typical task is to optimize (T, ι) for the following dynamic program, for instance by minimizing the number of join-bags. The implementations are standard and are therefore not further discussed within this thesis. The following code illustrates the use of a postprocessor that computes a nice tree decomposition:

```

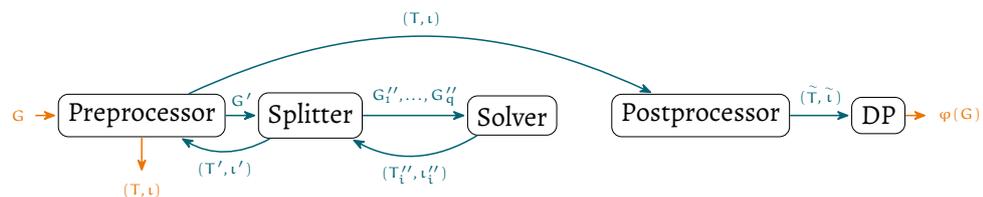
// create nice tree decomposition for the given decomposition td
// the boolean indicates whether the result should be very nice
NiceTreeDecomposition<T> ntd = new NiceTreeDecomposition<>(td, true);

// obtain the nice tree decomposition as tree decomposition
td = ntd.getProcessedTreeDecomposition()

```

The *Dynamic Programming* interface is the last part of the pipeline. It allows the specification of tree automata that are executed on a computed tree decomposition. For instance, the user may define a program that checks whether G can be properly colored with three colors. This is the most involved abstraction, as it has to, on the one hand, guarantee that *Jdrasil* can efficiently run the defined dynamic program, but should on the other hand neither be difficult to program nor restrictive to the user. The interface is closely linked to our point of view of tree automata, as explained in Section 8.2. The manual of *Jdrasil* contains a detailed example of how to implement a graph coloring solver using the interface [15], which is essentially the tree automaton that we have encountered in Example 137. A corresponding implementation is publicly available [11] and was experimentally evaluated in [13].

The attentive reader may have observed that the previous graphic of the pipeline must have simplified the information flow within the library. In reality, it is not quite as simple as in the graphic, since information has to flow *backwards*. For instance, after the solver has computed tree decompositions for the atoms, the splitter has to glue them all together to obtain an actual tree decomposition for the input graph. Similarly, the preprocessor has to enrich a tree decomposition of G' to one of G . Therefore, the “real” information flow looks as follows:



Jdrasil is modular in the sense that there are multiple implementations for every step of the pipeline. These implementations can be swapped arbitrarily due to the strict interfaces described above. This allows rapid development and testing of new algorithms. For instance, a user can implement a new solver using the interface of *Jdrasil* and plug it directly into the pipeline, without caring about pre- or postprocessing at all. This architecture has allowed me to add new algorithms quickly to the library. A notable example is the algorithm by Tamaki that he has developed for PACE 2016 [156] and improved for PACE 2017 [158]. We will discuss a game theoretic characterization of the algorithm, due to Berndt and myself [14], in Section 10.4.

10.3 A SAT-BASED EXACT-SOLVER

A common (theoretical and practical) approach to solve intractable problems is to reduce them to the Boolean satisfiability problem. The formulation we will use is based on the work of Berg and Jarvisalo [26], which in turn is an improved version of a formulation of Samer and Veith [149].

The Concept of Elimination Orders. Encoding treewidth directly into a propositional formula is a rather tough task. Fortunately, there is an alternative representation for treewidth that is ideally suited for a SAT-encoding. An *elimination order* π of a graph $G = (V, E)$ is a bijection $\pi: V \rightarrow \{1, 2, \dots, |V|\}$. The *filled graph* $G_\pi = (V, E_\pi)$ of the elimination order π is a directed graph with edges E_π that are constructed via the following process:

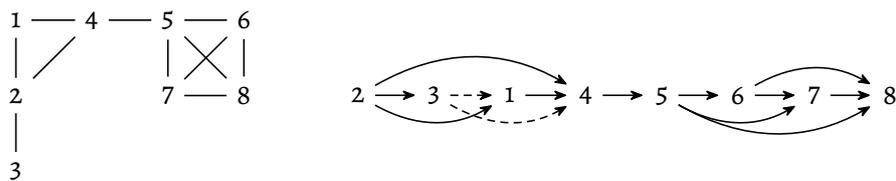
- The first edge set E_π^0 simply equals E , where the edges are directed from the “lower” vertex (according to π) to the “higher” vertex:

$$E_\pi^0 = \{ (u, v) \mid \pi(u) < \pi(v) \wedge \{u, v\} \in E \}.$$

- The next edge set E_π^{i+1} is generated by connecting all vertices u and v with $\pi(u) > i$ and $\pi(v) > i$ if both, u and v , are connected with the vertex $\pi^{-1}(i)$, that is, E_π^i results from E_π^{i-1} by adding the following edges to it:

$$\{ (u, v) \mid \pi(v) > \pi(u) > i \wedge (\pi^{-1}(i), u) \in E_\pi^{i-1} \wedge (\pi^{-1}(i), v) \in E_\pi^{i-1} \}.$$

Finally, E_π is equal to $E_\pi^{|V|}$. The following figure shows an example of a graph G and the corresponding filled graph G_π for $\pi = (2, 3, 1, 4, 5, 6, 7, 8)$. Here, the solid edges represent the edges of the original graph, while the dashed edges are the edges created by eliminating vertex 2.



The *width* of an elimination order π is the largest number of direct successors of a vertex in G_π , that is, $\text{width}(\pi) = \max_i \{ |\{(u_i, v) \in E_\pi\}| \}$. The width of the example is 3, as there exist three outgoing arcs from vertex 2 and 5. The following fact allows us to characterize the treewidth of a graph via an elimination order.

- Fact 143 (for instance [42])
 $\text{tw}(G) = \min_\pi \{ \text{width}(\pi) \}.$

◁

The Encoding of Berg and Jarvisalo. If $G = (V, E)$ is a graph on n vertices, our SAT-formula contains $n(n-1)/2$ variables $\text{ord}_{i,j}$ for each $i \in \{1, \dots, n\}$ and each $j > i$, indicating that the vertex v_i appears before v_j in the elimination order. To simplify notation, let us define $\text{ord}_{i,j}^*$ to be either $\text{ord}_{i,j}$ if $i < j$ or $\neg \text{ord}_{j,i}$ if $j < i$. To ensure that these variables encode a linear order of the vertices, it is sufficient to enforce the transitivity: For all distinct $i, j, k \in \{1, \dots, n\}$, we need to ensure that if $\text{ord}_{i,j}^*$ and $\text{ord}_{j,k}^*$ are true, then $\text{ord}_{i,k}^*$ is also true.

To encode the directed edges of the filled graph G_π , another n^2 variables $\text{arc}_{i,j}$ are introduced. As all original edges of G are present in G_π , for each $\{v_i, v_j\} \in E$, either $\text{arc}_{i,j}$ or $\text{arc}_{j,i}$ has to be set. To be consistent with the ordering implied by $\text{ord}_{i,j}$, we need to enforce that $\text{ord}_{i,j}^*$ implies that $\text{arc}_{j,i}$ is not set.

Finally, let us describe the elimination process and consider i, j, k with $\pi(i) < \pi(j)$ and $\pi(i) < \pi(k)$. Assume that v_i and v_k as well as v_i and v_j are adjacent, respectively. Then the filled graph G_π contains either the arc (v_j, v_k) or the arc (v_k, v_j) . Hence, if $\text{arc}_{i,j}$ and $\text{arc}_{i,k}$ are set and $\text{ord}_{j,k}^*$ is also set, then we need to set $\text{arc}_{j,k}$ as well. The following table summarizes all parts of the formula:

$\forall i, j, k \in \{1, \dots, n\}$	SAT-formulation
$i \neq j, i \neq k, j \neq k$	$\text{ord}_{i,j}^* \wedge \text{ord}_{j,k}^* \rightarrow \text{ord}_{i,k}^*$
$\{v_i, v_j\} \in E$	$\text{arc}_{i,j} \vee \text{arc}_{j,i}$
$i \neq j$	$\text{ord}_{i,j}^* \rightarrow \neg \text{arc}_{j,i}$
$i \neq j, i \neq k, j \neq k$	$\text{ord}_{j,k}^* \wedge \text{arc}_{i,j} \wedge \text{arc}_{i,k} \rightarrow \text{arc}_{j,k}$

Parameterized Cardinality Constraints. To ensure that the width of the produced elimination does not exceed a value $t \in \mathbb{N}$, we also need to make sure that for each v_i , at most t edges (v_i, v_j) exist in G_π . For a fixed t we define the formula $\varphi(G, t)$ as above and add the constraints $\sum_{j=1}^n \text{arc}_{i,j} \leq t$ for every i . Such constraints are called *cardinality constraints* and are usually implemented using a sorting network. Standard implementations, for instance using Batcher's odd-even mergesort, introduce $O(n \log^2 n)$ auxiliary variables. In *Jdrasil* we use *parameterized cardinality constraints* of size $O(t \cdot n)$ whenever t is small enough. They are based on classical sequential counters. An overview of different encodings for cardinality constraints can be found in the following survey papers [10, 92].

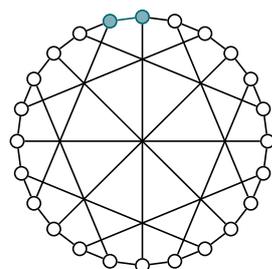
In order to determine $\text{tw}(G)$, the above encoding would be used for $t = n, n-1, \dots$ until the system does not have any solution. We make use of the *iterative* abilities of modern SAT-solvers that allows to add clauses to an already solved formula. This technique was also recommended by Berg and Jarvisalo [26]. The solver starts by solving the formula $\varphi(G, n)$. After the SAT-solver has solved a formula $\varphi(G, t)$ for some $1 \leq t \leq n$, it adds the constraints $\sum_{j=1}^n \text{arc}_{i,j} \leq t-1$ for every i , obtaining the formula $\varphi(G, t-1)$. The solver then tries to solve this new formula and it repeats the whole procedure until it reaches a t for which $\varphi(G, t)$ is not satisfiable.

Adding the Clique Trick. We extend the encoding of Berg and Jarvisalo by a trick that was observed in the context of exact algorithms for treewidth [36]: From the definition of partial k -trees (an equivalent formalism for treewidth), it follows that if $C \subseteq V$ is a clique in G , then there is an optimal elimination order for G that eliminates C at the very end. Therefore, if we know some clique C in G we can hard-wire the $\text{ord}_{j,k}^*$ for it. The new parts that we add to the formula are shown in the table.

$\forall i, j \in \{1, \dots, n\}$	SAT-formulation
$v_i \in V \setminus C, v_j \in C$	$\text{ord}_{i,j}^*$
$v_i, v_j \in C, i < j$	$\text{ord}_{i,j}^*$

This technique is the better the larger the clique is. Of course, in general it is intractable to find a large clique. In our implementation we use the SAT-solver for this task as well, but give it only a limited amount of time. In case the solver does not find a clique, we run a simple heuristic to find at least some clique. I noticed, however, that this event happens rarely as the SAT-solver works well to find cliques on graphs of small treewidth – which is not surprising, as the maximum size of a clique in G is bounded by the treewidth of G .

Adding the clique trick can be seen as adding domain specific knowledge to the formula. I suspect that it implies a strong form of symmetry breaking, as the performance boost obtained by the trick is astonishing. For instance, the *McGee graph* (the smallest cubic graph of girth 7) is the highly symmetric graph visualized in the margin. It has only 24 vertices and 36 edges, but a corresponding SAT-formula *without* the clique trick could not be solved in over 5 hours. In contrast, the formula *with* the clique trick can be solved in less than 5 minutes – and that despite the fact that the largest clique has just size 2 (it is highlighted in the figure). A similar positive effect was later observed when an updated version of the clique trick was used for a novel SAT-encoding for fractional hypertreewidth [81].



Adding the Twin Trick. Another trick that we may add is the twin trick, which is based on the following observation: Assume $v, w \in V$ are *twins* (that is, $N(v) = N(w)$) and assume π is an optimal elimination order in which v is eliminated before w ; then the permutation π' that is created from π by swapping v and w is optimal as well. This claim is implied by the following observation: Whenever one of the twins will be eliminated, the other one becomes simplicial (its neighborhood is a clique), and it is well known that such a vertex can safely be eliminated at any time [99].

The twin-relation is in fact an equivalence relation on V , and we may safely fix any or-

$\forall i, j \in \{1, \dots, n\}, \forall \ell \in \{1, \dots, q\}$	SAT-formulation
$v_i, v_j \in P_\ell \setminus C, i < j$	$\text{ord}_{i,j}^*$

der on the vertices in every equivalence class. We only have to be cautious whenever we use this trick in combination with the clique trick: If a twin is part of the clique, it must be ordered behind the other twins within the same equivalence class – otherwise we would encode a contradiction. Formally, we add the parts shown in the table to our encoding, where P_1, \dots, P_q are the non-trivial twin-classes of G (that means $|P_i| > 1$) and $C \subseteq V$ is the clique used for the clique trick.

The combination of preprocessing and splitting followed by a SAT-based approach works surprisingly well in practice. Sebastian Berndt, Thorsten Ehlers, and myself participated with it in the PACE 2016 challenge, the results can be found in [63] and a follow-up analysis was done in [16]. The implementation was also used in a local improvement solver by Fichte, Lodha, and Szeider, where it outperformed other approaches [83]. The same group later extended the approach to other parameters such as treedepth and fractional hypertreewidth [81, 96].

10.4 EXACT SOLVING VIA POSITIVE INSTANCE DRIVEN DYNAMIC PROGRAMMING

In this section we will study an exact algorithm for treewidth based on a novel algorithmic technique called *positive instance driven dynamic programming*. This technique was invented by Hisao Tamaki for his submission to the first *Parameterized Algorithms and Computational Experiments Challenge* (PACE 2016) [156]. In the second iteration of the challenge (PACE 2017) all submissions – including another one by Tamaki [157, 158], the winning submission by Larisch and Salfelder [128], as well as my own submission [14, 15] – were based on positive instance driven dynamic programming. In fact, the currently best way of computing optimal tree decompositions in practice is using this technique, and it seems that it is inherently better than other classical dynamic programs for treewidth [158, 159].

In this section I will present an algorithm due to Sebastian Berndt and myself that is directly based on Tamaki’s first algorithm [156] and that applies positive instance driven dynamic programming to solve a general version of graph searching [14, 86]. Since graph searching is deeply linked to various graph decompositions, we will not only obtain an algorithm for treewidth, but also for other graph parameters such as pathwidth and treedepth.

What is positive instance driven dynamic programming? This technique describes an *execution mode* for a classical dynamic program. Before we go into the details of the definition, let us stipulate what we mean by a “classical dynamic program.” Assume we wish to solve some decision problem, a classical dynamic program is a *recursive procedure* that solves the problem (it returns either `true` or `false`) together with a *memoization table*. Such a program explores its *configuration graph* (also called the *memoization graph*), which is the acyclic graph that contains all possible configurations as vertex set and which has directed edges according to the recursive calls of the procedure. This graph has a unique *start configuration* and its sinks correspond to the configurations at which the recursion stops. These sinks are partitioned into *win configurations* and *lose configurations* (depending on whether the recursive procedure returns `true` or `false`, respectively). The dynamic program labels each non-sink vertex v of the configuration graph with either `true` or `false` using a Boolean combination of the truth values of the children of v . All vertices that are labeled with

true constitute the *winning region* (the *positive instances*), and the decision problem reduces to the question whether or not the start configuration is part of this region.

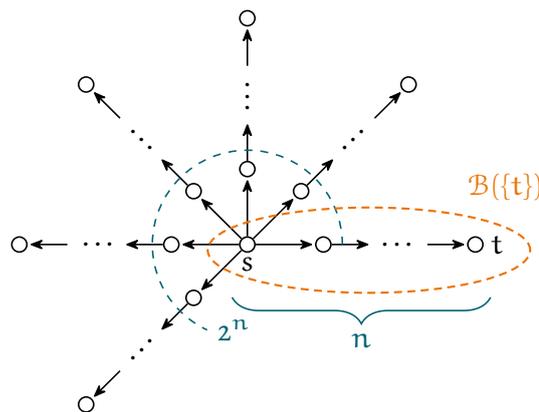
It is a bit tricky to work directly with the configuration graph, as it does not encode any information about the Boolean combination used to label its vertices. Therefore, we will only consider dynamic programs that use Boolean combinations of the following form: Let $H = (V, E)$ be the configuration graph of the program and let $\lambda: V \rightarrow \{\text{true}, \text{false}\}$ be the labeling computed by it, then for all non-sink vertices v there is a (potentially empty) set $I \subseteq N(v)$ such that:

$$\lambda(v) = \bigwedge_{w \in I} \lambda(w) \quad \vee \quad \bigvee_{w \in N(v) \setminus I} \lambda(w).$$

If we consider dynamic programs with this property, then we can encode the way λ is computed by coloring the edges of the configuration graph. An *edge-alternating graph* is a triple $H = (V, E, A)$ consisting of a vertex set V , an existential edge relation $E \subseteq V \times V$, and a universal edge relation $A \subseteq V \times V$. We define the neighborhood of a vertex v as $N_{\exists}(v) = \{w \mid (v, w) \in E\}$, $N_{\forall}(v) = \{w \mid (v, w) \in A\}$, and $N(v) = N_{\exists}(v) \cup N_{\forall}(v)$. An *edge-alternating s-t-path* is a set $P \subseteq V$ such that (i) $s, t \in P$ and (ii) for all $v \in P$ with $v \neq t$ we have either $N_{\exists}(v) \cap P \neq \emptyset$ or $\emptyset \neq N_{\forall}(v) \subseteq P$ or both. We write $s \prec t$ if such a path exists and define $\mathcal{B}(Q) = \{v \mid v \in Q \text{ or there is a } w \in Q \text{ with } v \prec w\}$ for $Q \subseteq V$ as the set of vertices on edge-alternating paths leading to Q . Observe that the winning region is exactly $\mathcal{B}(Q)$ if Q is the set of win configurations. Further, observe that edge-alternating graphs are a generalization of alternating graphs (we studied them in Section 4.2), in which we have for all vertices v either $N_{\exists}(v) = \emptyset$ or $N_{\forall}(v) = \emptyset$. Example 144 on the next page illustrates the concept with a simple two-player game.

As the name suggests, a *positive instance driven dynamic program* is a procedure that mimics a classical dynamic program, but that computes *only* the winning region (the positive instances) without ever “touching” the rest of the configuration graph.

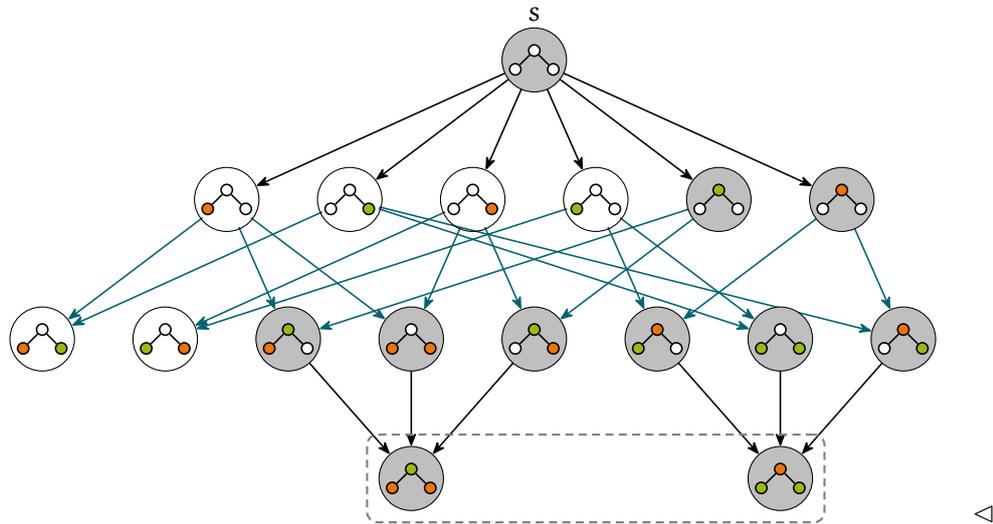
It is not clear at all whether such a procedure exists for a given classical dynamic program. If it exists, however, it will usually require much more time to explore a new configuration than the classical dynamic program would require (because it has to explore the configuration graph in a reversed direction). The hope is, of course, that the winning region is much smaller than the whole configuration graph. It is easy to see that, in principle, the winning region can be exponentially smaller than the configuration graph – consider for instance



a configuration graph that is a spider with 2^n legs of size n , in which only one win configuration exists at the end of a leg (as shown in the figure). How big the difference between the size of the winning region and the size of the configuration graph is in reality, depends highly on the dynamic program and the given input.

► Example 144

Consider the following 2-color-construction game played by Alice and Bob: The input is a two-colorable graph $G = (V, E)$, and Alice begins. In each turn, a player may color an uncolored vertex with one of two colors, as long as the induced coloring is still proper (that is, adjacent vertices have different colors). Alice wins if all vertices of the graph are colored (which implies a proper coloring), otherwise Bob wins. The configuration graph of a simple algorithm, which tries all possible moves, is shown below for the input graph $G = \circ-\circ-\circ$. Existential edges are black and **universal** edges are **blue**. The winning region (the positive instances) is highlighted, and the two win configurations are the configurations in the dashed box.



Computing Tree Decompositions via Graph Searching. If we wish to use positive instance driven dynamic programming to compute optimal tree decompositions, we first require a classical dynamic program for that task. In order to formulate such a dynamic program, we use an equivalent characterization of treewidth in the form of a vertex pursuit-evasion game. We study classical graph searching in a general setting proposed by Fomin, Fraignaud, and Nisse [86]. The input is an undirected graph $G = (V, E)$ and a number $k \in \mathbb{N}$, and the question is whether a team of k searchers can catch an *invisible* fugitive on G by the following set of rules: At the beginning, the fugitive is placed at a vertex of her choice and at any time, she knows the position of the searchers. In every turn she may move with *unlimited speed* along edges of the graph, but may never cross a vertex occupied by a searcher. This implies that

the fugitive does not occupy a single vertex but rather a subgraph, which is separated from the rest of the graph by the searchers. The vertices of this subgraph are called *contaminated*, and at the start of the game all vertices are contaminated. The searchers, trying to catch the fugitive, can perform one of the following operations during their turn:

1. *place* a searcher on a contaminated vertex;
2. *remove* a searcher from a vertex;
3. *reveal* the current position of the fugitive.

When a searcher is placed on a contaminated vertex, the vertex becomes *clean*. When a searcher is removed from a vertex v , the vertex may become *recontaminated* if there is a contaminated vertex adjacent to v . The searchers win the game if they manage to clean all vertices, that is, if they catch the fugitive; the fugitive wins if, at any point, a recontamination occurs, or if she can escape infinitely long. Note that this implies that the searchers have to catch the fugitive in a *monotone* way. *A priori* one could assume that the later condition gives the fugitive an advantage (recontamination could be necessary for the cleaning strategy), however, a crucial result in graph searching is that “recontamination does not help” in all variants of the game that we consider [28, 98, 127, 131, 152].

The *search number* $s(G)$ of a graph $G = (V, E)$ is the minimum number of searchers required to ensure that the searchers can catch the fugitive. It is well known that this number is directly linked to the treewidth of G , and that this connection is constructive in the sense that a winning strategy of the searchers can be turned into a tree decomposition and vice versa.

- Fact 145 (Originally [152], for the version presented here see also [86] and [14].)
 For every graph $G = (V, E)$ it holds that $s(G) = \text{tw}(G) + 1$. ◁

Equipped with the link between treewidth and graph searching, our task of providing a dynamic program for treewidth has reduced to the task of providing a dynamic program that determines whether k searchers have a winning strategy.

We simplify the game to make it more accessible for an algorithmic approach. First of all, we restrict the fugitive in the following sense: Since she is invisible to the searchers and travels with unlimited speed, there is no need for her to take regular actions. Instead, the only moment when she is actually active is when the searchers perform a reveal. If C is the set of contaminated vertices, consisting of the induced components C_1, \dots, C_ℓ , a reveal will uncover the component in which the fugitive hides and, as a result, reduce C to C_i for some $1 \leq i \leq \ell$. The only task of the fugitive is, thus, to answer a reveal with such a number i . We call the whole process of the searchers performing a reveal, the fugitive answering it, and finally of reducing C to C_i a *reveal-move*.

We will also restrict the possible moves of the searchers by the concept of *implicit searcher removal*. Let $S \subseteq V(G)$ be the vertices currently occupied by the searchers, and let $C \subseteq V(G)$ be the set of contaminated vertices. We call a vertex $v \in S$ *covered* if every path between v and C contains a vertex $w \in S$ with $w \neq v$.

► Lemma 146

A covered searcher can be removed safely.

Proof. As we have $N(v) \cap C = \emptyset$, the removal of v will not increase the contaminated area. Furthermore, at no later point of the game v can be recontaminated, unless a neighbor of v gets recontaminated as well (in which case the game would already be lost for the searchers). \square

► Lemma 147

Only covered searchers can be removed safely.

Proof. Since for any other vertex $w \in S$ we have $N(w) \cap C \neq \emptyset$, the removal of w would recontaminate w and, hence, would result in a defeat of the searchers. \square

Both lemmas together imply that the searchers never have to decide to remove a searcher, but rather can do it *implicitly*. We thus restrict the possible moves of the searchers to a combined move of placing a searcher and *immediately* removing all searchers on covered vertices. We call this a *fly-move*. Observe that the sequence of original moves mimicked by a fly-move does not contain a reveal and, thus, may be performed independently of any action of the fugitive. We are now ready to describe the configurations of the game, which we do in the form of *k-blocks*:

► Definition 148

A *k-block*, or simply a *block*, of a graph $G = (V, E)$ is a tuple (S, C) with $S, C \subseteq V$ such that:

1. $S \cap C = \emptyset$;
2. $N(C) \subseteq S$;
3. $|S| \leq k$.

◁

A block (S, C) encodes the position of the searchers (the set S) and the contaminated area (the set C). Observe that in any block (S, C) the set S separates C from the rest of G . We can explore all blocks, and thus all configurations of the game, with the following “Robertson–Seymour fashioned” dynamic program¹ in order to determine whether k searcher have a winning strategy. It is assumed that an input graph $G = (V, E)$ and a target number $k \in \mathbb{N}$ is globally available in memory, and that the procedure is started with the block (\emptyset, V) and executed using memoization.

```

procedure divideAndCatch(S, C)

  // recursion stops
  if |S| > k then // we need too many searchers → lose configuration
    return false
  end
  if C = ∅ then // the searchers cleaned the graph → win configuration
    return true
  end

  // implicit searcher removal
  for v ∈ S do
    if N_G(v) ∩ C = ∅ then
      return divideAndCatch(S \ {v}, C)
    end
  end

  // reveal-move
  C1, ..., Cℓ ← connectedComponents(G[C])
  if ℓ > 1 then
    return  $\bigwedge_{i=1}^{\ell}$  divideAndCatch(S, Ci)
  end

  // fly-move
  return  $\bigvee_{v \in C}$  divideAndCatch(S ∪ {v}, C \ {v})
end

```

Note that, in the case of the procedure returning true, it is easy to obtain a winning strategy using backtracking. In fact, this strategy is a width $k-1$ tree decomposition. Looking at the program a little closer, it is obvious that the only interesting configurations of the program are the blocks with $S = N(C)$. We call such objects *full blocks*. We can rewrite the algorithm such that it works only with full blocks, which will make it easier to analyze its configuration graph.

¹Robertson and Seymour have established an fpt-approximation algorithm for treewidth that essentially is the presented algorithm, but that adds balanced separators to S instead of “brute-forcing” a fly-move [145]. The algorithm as presented here is leaned to the description of Reed [142], details can be found in the standard textbooks [59, 85].

```

procedure divideAndCatchFull(C)

  // recursion stops
  if C = ∅ then // the searchers cleaned the graph → win configuration
    return true
  end

  // reveal-move
  C1, ..., Cℓ ← connectedComponents(G[C])
  if ℓ > 1 then
    return  $\bigwedge_{i=1}^{\ell}$  divideAndCatchFull(Ci)
  end

  // not enough searcher to perform a fly-move
  if |NG(C)| = k then
    return false
  end

  // fly-move
  return  $\bigvee_{v \in C}$  divideAndCatchFull(C \ {v})

end

```

This is exactly the dynamic program to which we will apply the positive instance driven dynamic programming technique. To that end, let us define the configuration graph of the algorithm: For an input graph $G = (V, E)$ and a number $k \in \mathbb{N}$, it is the edge-alternating graph $H = (V(H), E^H, A^H)$ with:

$$\begin{aligned}
V(H) &= \{ C \mid C \subseteq V(G) \text{ and } |N_G(C)| \leq k \}, \\
E^H &= \{ (C, C') \mid C \setminus \{v\} = C' \text{ for some } v \in C \text{ and } |N_G(C)| < k \}, \\
A^H &= \{ (C, C') \mid C' \text{ is a connected component of } G[C] \}.
\end{aligned}$$

Observe that the set Q of win configurations of the graph is exactly the set $\{ \{v\} \mid v \in V(G) \text{ and } |N_G(\{v\})| < k \}$, as these are the positions in which the searchers can catch the fugitive after they have cleaned everything else. The winning region is, thus, $\mathcal{B}(Q)$ and our aim is to develop an algorithm that computes this region. Our algorithm traverses H “backwards” by starting at the set Q of winning configurations and by uncovering $\mathcal{B}(Q)$ layer by layer. In order to achieve this, we need to compute the predecessors of a configuration C . This is easy if C was reached by a fly-move, as we can simply enumerate the at most k possible predecessors (the last searcher that was placed is in $N(C)$, therefore the predecessors of C are exactly the configurations $C' = C \cup \{v\}$ with $v \in N(C)$ and $|N(C')| < k$)². Reversing a reveal-move, that is, finding the universal predecessors, is significantly more involved. A simple approach is to test for every subset of already explored configurations if we can “glue” them together – but this would result in a run time of the form $2^{|V(H)|}$.

²Actually, there is a special case if C contains a vertex that is isolated in $G[C]$. Fortunately, this case is covered by the reverse reveal-moves that we describe next.

Fortunately, we can avoid this exponential blow-up as H has the following useful property:

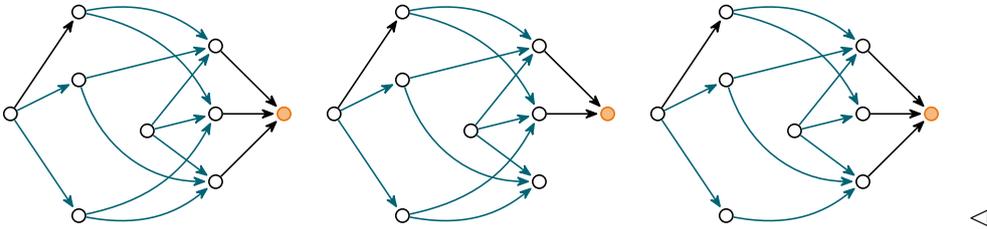
► Definition 149 (Universal Consistent)

We say that an edge-alternating graph $H = (V, E, A)$ is *universal consistent* with respect to a set $Q \subseteq V$ if for all $v \in V \setminus Q$ with $v \in \mathcal{B}(Q)$ and $N_{\forall}(v) = \{w_1, \dots, w_r\}$ we have (i) $N_{\forall}(v) \subseteq \mathcal{B}(Q)$ and (ii) for every $I \subseteq \{w_1, \dots, w_r\}$ with $|I| \geq 2$ there is a vertex $v' \in V$ with $N_{\forall}(v') = I$ and $v' \in \mathcal{B}(Q)$. ◁

Intuitively, this definition implies that for every vertex with high universal-degree there is a set of vertices that we can arrange in a tree-like fashion to realize the same adjacency relation. This allows us to glue only two configurations at a time and, thus, removes the exponential dependency.

► Example 150

Consider the following three edge-alternating graphs, where black edges are existential and the blue edges are universal. The set Q contains a single vertex that is highlighted. From left to right: the first graph is universal consistent; the second and third one are not. The second graph conflicts the condition that $v \in \mathcal{B}(Q)$ implies $N_{\forall}(v) \subseteq \mathcal{B}(Q)$, as the vertex on the very left is contained in $\mathcal{B}(Q)$ by the top path, while its universal neighbor on the bottom path is not contained in $\mathcal{B}(Q)$. The third graph conflicts the condition that $N_{\forall}(v) = \{w_1, \dots, w_r\}$ implies that for every $I \subseteq \{w_1, \dots, w_r\}$ with $|I| \geq 2$ there is a vertex $v' \in V$ with $N_{\forall}(v') = I$ and $v' \in \mathcal{B}(Q)$ as witnessed by the vertex with three outgoing universal edges.



► Lemma 151

For every graph $G = (V, E)$ and number $k \in \mathbb{N}$, the edge-alternating configuration graph H of the algorithm `divideAndCatchFull(V)` is universal consistent.

Proof. For the first property observe that “reveals do not harm”: Searchers that can catch the fugitive without knowing where she hides, certainly can do so if they know.

For the second property consider any configuration $C \in V(H)$ that has universal edges to C_1, \dots, C_ℓ . By definition we have $|N_G(C)| \leq k$ and $N_G(C_i) \subseteq N_G(C)$ for all $1 \leq i \leq \ell$. Therefore we have for every $I \subseteq \{1, \dots, \ell\}$ and $C' = \cup_{i \in I} C_i$ that $N_G(C') \subseteq N_G(C)$ and $|N_G(C')| \leq k$ and, thus, $C' \in V(H)$. ◻

We are now ready to formulate the algorithm for computing the winning region. In essence, the algorithm runs in two phases: First it computes the set Q of winning configurations; second the winning region $\mathcal{B}(Q)$ is computed by the sketched reversed moves.

```

1  procedure catchAndGlue(G,k)
2      K ← ∅
3      initialize empty queue
4
5      // Phase I: compute Q
6      for v ∈ V(G) do
7          offer({v}, k-1)
8      end
9
10     // Phase II: compute B(Q)
11     while queue not empty do
12         extract C from queue
13
14         // reverse fly-moves
15         for v ∈ N_G(C) do
16             offer(C ∪ {v}, k-1)
17         end
18
19         // reverse reveal-moves
20         for C' ∈ K do
21             if not intersect(C, C') then
22                 offer(C ∪ C', k)
23             end
24         end
25
26     end
27
28     // done, we now have K = B(Q)
29     return K
30
31 end

```

```

procedure offer(C,t)
    if C ∉ K and |N_G(C)| ≤ t then
        add C to K
        insert C into queue
    end
end

procedure intersect(C,C')
    if C ∩ C' ≠ ∅ then
        return false
    end
    if N_G(C) ∩ C' ≠ ∅ then
        return false
    end
    if C ∩ N_G(C') ≠ ∅ then
        return false
    end
    return true
end

```

► **Theorem 152**

The algorithm `catchAndGlue(G,k)` finishes in at most $O(|\mathcal{B}(Q)|^2 \cdot |V|^2)$ steps and correctly outputs $\mathcal{B}(Q)$.

Proof. The algorithm is supposed to compute Q in phase I and the rest of $\mathcal{B}(Q)$ in phase II. Observe that Q is correctly computed in phase I by the definition of Q .

To show the correctness of the second phase we argue that the computed set K equals $\mathcal{B}(Q)$. Observe that K is exactly the set of vertices inserted into the queue. We first show $K \subseteq \mathcal{B}(Q)$ by induction over the i th inserted vertex. The first vertex C_1 is in $\mathcal{B}(Q)$ as $C_1 \in Q$. Now consider C_i . As $C_i \in K$, it was either added in Line 16 or Line 22. In the first case there was a vertex $C'_i \in K$ such that $C_i = C'_i \cup \{v\}$ for some $v \in N_G(C'_i)$. By the induction hypothesis we have $C'_i \in \mathcal{B}(Q)$ and by the definition of the configuration graph we have $(C_i, C'_i) \in E^H$ and, therefore, we also have that $C_i \in \mathcal{B}(Q)$. In the second case there were vertices C'_i and C''_i

with $C'_i, C''_i \in K$ and $C_i = C'_i \cup C''_i$. By the induction hypothesis we have again $C'_i, C''_i \in \mathcal{B}(Q)$. Let t_1, \dots, t_ℓ be the connected components of C'_i and C''_i . Since H is universal consistent with respect to Q by Lemma 151, we have $t_1, \dots, t_\ell \in \mathcal{B}(Q)$. By the definition of H we have $N_\forall(C_i) = t_1, \dots, t_\ell$ and, thus, $C_i \in \mathcal{B}(Q)$.

To see $\mathcal{B}(Q) \subseteq K$ consider for a contradiction the vertices of $\mathcal{B}(Q)$ in reversed topological order (recall that H is acyclic) and let C be the first vertex in this order with $C \in \mathcal{B}(Q)$ and $C \notin K$. If $C \in Q$ we have $C \in K$ by phase I and are done, so assume otherwise. Since $C \in \mathcal{B}(Q)$ we have either $N_\exists(C) \cap \mathcal{B}(Q) \neq \emptyset$ or $\emptyset \neq N_\forall(C) \subseteq \mathcal{B}(Q)$. In the first case there is a $C' \in \mathcal{B}(Q)$ with $(C, C') \in E^H$. Therefore, C' precedes C in the reversed topological order and, by the choice of C , we have $C' \in K$. Therefore, at some point of the algorithm C' gets extracted from the queue and, in Line 16, would add C to K , a contradiction.

In the second case, the configuration graph contains vertices t_1, \dots, t_ℓ such that $N_\forall(C) = \{t_1, \dots, t_\ell\}$ and $t_1, \dots, t_\ell \in \mathcal{B}(Q)$. By the choice of C , we have again $t_1, \dots, t_\ell \in K$. Since H is universal consistent with respect to Q , we have for every $I \subseteq \{1, \dots, \ell\}$ that $\bigcup_{i \in I} t_i$ is contained in $\mathcal{B}(Q)$. In particular, the vertices $t_1 \cup t_2, t_3 \cup t_4, \dots, t_{\ell-1} \cup t_\ell$ are contained in $\mathcal{B}(Q)$, and these elements are added to K whenever the t_i are processed (for simplicity assume here that ℓ is a power of 2). Once these elements are processed, Line 22 will also add their union, that is, vertices of the form $(t_1 \cup t_2) \cup (t_3 \cup t_4)$. In this way, the process will add vertices that correspond to increasing subgraphs of G to K , resulting ultimately in adding $\bigcup_{i=1}^\ell t_i = C$ to K , which is the contradiction we have been looking for. \square

Theorem 152 provides us with an efficient algorithm for computing the winning region of the search game. Since the question whether the input graph has treewidth at most $k - 1$ can be answered by a simple lookup that checks whether the start configuration $V(G)$ is contained in this region, we can answer this question in the same time. In fact, we can compute a corresponding winning strategy – and thus a tree decomposition – on the fly, by tracking which operation offers which configurations to K . Another strength of Theorem 152 is that it can easily be adapted to other graph parameters as graph searching is very general: If we forbid reveals, the search number equals $\text{pw}(G) + 1$; if we forbid to remove placed searchers, the search number equals $\text{td}(G)$. In fact, it can be shown that we can compute the corresponding decompositions within the same time bound by a small modification of the algorithm used for Theorem 152 [14]. However, for this section we are content with the computation of optimal tree decompositions.

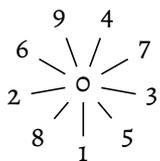
In the rest of this section, we will apply algorithmic engineering in order to speed up the algorithm in practice (without improving its theoretical run time). The main idea of all of the following improvements is that we do not need to know $\mathcal{B}(Q)$ completely, as we only want to know whether the start configuration is contained in it.

The Priority Heuristic. The first observation we use is that the start configuration $V(G)$ is the largest block. Thus, we wish to generate *large* configurations quickly. We can prioritize such configurations by changing the queue to a priority queue in which the priority of a block $(N(C), C)$ is the cardinality of C . In this way, the algorithm will extend larger blocks first and, thus, hopefully finds the start configuration faster. Despite its simplicity, this heuristic has an enormous effect in practice. For instance, the McGee Graph mentioned in the previous section is a hard instance for the algorithm presented in this section as well (there are many configurations to glue, as the graph is very symmetric) – the plain algorithm is not able to solve it within 10 minutes, while using the priority heuristic it is solved in less than a second.

The Fast-Contamination Heuristic. Another trick that we can apply is to contaminate vertices instantaneously whenever this is *safe*. Contaminating a vertex means increasing a currently handled full block $(N(C), C)$ by taking a vertex $v \in N(C)$ and moving it to C . This operation is performed by the reverse fly-move, where we try all possible choices of v . However, this operation is *safe* if $N(C \cup \{v\}) \subseteq N(C)$, that is, if we do not require any additional searchers for the new block. Therefore, if we find such a vertex we can directly contaminate it, and the fast-contamination heuristic does exactly this: Whenever we explore a new full block, we greedily contaminate all vertices that can be contaminated safely.

Additionally, we may always contaminate the remaining graph if it is smaller than k , that is, if we handle a full block $(N(C), C)$ with $|V \setminus C| < k$, then we can contaminate the whole graph at once. In this scenario, the start configuration of the searchers would be $V \setminus C$, from where they then would move to $N(C)$.

The Fast-Glue Heuristic. Star-like graphs are worst-case instances for the presented algorithm. Consider for instance the star shown in the margin and observe that for the optimal search number $k = 2$, the set Q of win configurations is exactly the set of leaves, that is, $Q = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}\}$. Observe that for none of these configurations a reverse fly-move is possible, as the neighborhood of any set containing the vertex o is too large. Therefore, the algorithm can only generate larger configurations by gluing the small configurations together, and it will therefore first generate *all* pairs, then *all* 3-tuples, then *all* 4-tuples, and so on. With other words, for such instances $\mathcal{B}(Q)$ is of exponential size and the algorithm will fully list $\mathcal{B}(Q)$ before it finds the start configuration.



To overcome this issue, the fast-glue heuristic keeps gluing a full block as long as possible, before it handles the next block. This can be achieved by replacing Lines 20–24 in the main algorithm with the code snippet on the next page.

```

init empty stack
push C
while stack is not empty do
  pop C' from stack
  for C'' ∈ K do
    if not intersect(C', C'') then
      offer(C' ∪ C'', k)
      push C' ∪ C'' to stack
    end
  end
end
end

```

Considering the star example again, the heuristic will glue the first handled win configuration, say $\{1\}$, together to $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ before extracting any other block from the queue. If we use the heuristic in combination with the fast-contamination heuristic, this block will be extended to $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and the algorithm finds a solution without extracting any other block from the queue.

Discarding Configurations via Pruning. While we are exploring the winning region $\mathcal{B}(Q)$ we may encounter configurations that are not necessary for an optimal tree decomposition, as we may already have encountered a “better” configuration. In this scenario, we can safely discard this configuration by not adding it to $\mathcal{B}(Q)$ and by not offering it to the queue – which in return will increase the speed of the algorithm.

Assume we encounter a block $(N(C), C)$ for the first time, and assume that we have already found a block $(N(C'), C')$ with $C \subseteq C'$ and $N(C') \subseteq N(C)$. We can safely discard $(N(C), C)$, as any strategy of the searchers that use this block (the searcher visit $N(C)$ to clean C) can instead use $(N(C'), C')$, as this requires fewer searchers at the same spot while cleaning a larger part of the graph (C' rather than C).

The Potential-Maximal-Clique-Heuristic. The last heuristic is based on more involved graph theoretic concepts concerning tree decompositions. A *potential maximal clique* of a graph $G = (V, E)$ is a set of vertices $\Omega \subseteq V$ that is a clique in some minimal triangulation of G . Bouchitté and Todinca have proven the following more accessible local characterization of potential maximal cliques [44], which allows us to efficiently test whether a given set of vertices is a potential maximal clique:

► Fact 153 ([44])

Let $G = (V, E)$ be a graph and $\Omega \subseteq V$, then Ω is a potential maximal clique if and only if:

1. $G[V \setminus \Omega]$ has no full component associated with Ω ;
2. for all $u, v \in \Omega$ we have either $\{u, v\} \in E$ or there is at least one component C associated with Ω such that $u, v \in N(C)$. ◁

One can show that, if k searchers have a winning strategy on G , then they have also a winning strategy with the following more restricted set of rules: Whenever the searchers place new searchers on G , the set S of searchers has to be a potential maximal clique; and whenever the searchers remove some searchers from the graph, the set S must be a minimal separator [88] (see also Section 5.4 in [87]). Of course, in this variant the searchers have to place and remove several searchers at once.

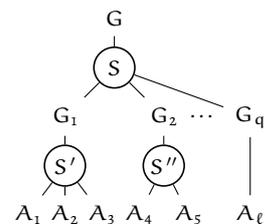
Observe that for a set Ω that has no associated full component and that is no potential maximal clique, no superset $\Omega' \supseteq \Omega$ can be a potential maximal clique. Furthermore, observe that the last move of the searchers (the move in which the fugitive is caught) is a place move on a vertex v and, thus, $N[v]$ must be a potential maximal clique. Therefore, we may discard blocks $(N(\{v\}), \{v\})$ in Line 7 whenever $N[v]$ is not a potential maximal clique.

We can naturally follow this path further and try to reconstruct only such winning strategies in which the searchers always stand on minimal separators or on potential maximal cliques. However, this is highly non-trivial from a positive instance driven point of view, as it is only easy to find minimal separators given a potential maximal clique that contain them; but it is hard to find a potential maximal clique that contains a given minimal separator. Tamaki has proven new structural properties of potential maximal cliques in order to present a positive instance driven algorithm based on such a strategy [158]. He participated with this algorithm at the PACE 17 [64, 157]. An optimized implementation of this algorithm can be found in *Jdrasil* as well, but for this section we leave it at the heuristic that prunes win configurations that are no potential maximal cliques.

10.5 PARALLELIZATION THROUGH SPLITTING

In our architecture for computing a tree decomposition, the *splitter* is the second element in the pipeline. Its task is to *split* an input graph G into many small graphs G_1, \dots, G_q , to which we refer to as *atoms*. While doing so, the splitter must guarantee that it can glue tree decompositions of the G_i into a tree decomposition of G . Furthermore, it must ensure that this tree decomposition is optimal in the event that the tree decompositions for the G_i are optimal.

The cornerstone to the concept of the splitter is the notation of *safe separators* due to Bodlaender and Koster [41]. For a graph $G = (V, E)$, a set $S \subseteq V$ is called a *safe separator* if (i) S is a separator (that is, $G[V \setminus S]$ has more components than $G[V]$) and (ii) completing S into a clique does not increase the treewidth of G . Such safe separators are useful by the simple observation that any tree decomposition (T, ι) of G must contain for every clique $C \subseteq V$ a bag b with $C \subseteq \iota(b)$, which leads to the following recursive scheme illustrated at the margin: On input of a graph G , the splitter finds a safe separator S ; separates G on it; adds S as clique to each resulting component and obtains new graphs G_1, \dots, G_q ; it then recurses on these graphs to obtain tree decompositions for all of them. These decompositions can be glued together to a tree decomposition of G by adding a new bag b with $\iota(b) = S$, which is linked to one bag containing S in each of the obtained tree decompositions. The recursion stops when the splitter reaches a graph in which it cannot find any safe separator – this graph is an atom and a tree decomposition for it is obtained with a solver. It should be clear that this approach is highly desirable from a parallel point of view, as we can handle all the atoms independently of each other in parallel.



Clique and Almost Clique Separators. A clique separator is a separator S such that S is a clique. Obviously, such separators are safe by the above definition. Given a graph $G = (V, E)$, we can find a clique separator, if one exists, in time $O(n \cdot m)$ using the algorithm by Gavril [97]. In fact, in the same time we can actually compute a whole decomposition of G along such clique separators using an improved version of the algorithm due to Tarjan [161]. In the light of parallelization, there is also an $O(\log^3 n)$ time and $O(n \cdot m)$ work algorithm to find clique separators [60]. The implementation in *Jdrasil* is oriented on the description by Berry, Pogorelcnik, and Simonet to find minimal clique separators [27].

Bodlaender and Koster [41] have observed that a separator S is also safe if it is an inclusion minimal *almost clique*. An almost clique is simply a clique plus one additional vertex. We can find almost clique separators by guessing a vertex v and looking for a clique separator in $G[V \setminus \{v\}]$. This operation requires time $O(n^2 \cdot m)$. If we have found an almost clique separator S , we can check if it is inclusion minimal by testing if its associated components are full.

It is worth to mention that there are interesting special cases of clique and almost clique separators: The separators of size zero, one, and two. A single vertex is a clique, two vertices are either an edge (and thus a clique) or a clique and vertex (and thus an almost clique). These separators correspond to connected components, bi-connected components, and triconnected components, which we can identify more efficiently with standard graph theoretic techniques [108, 109]. Jdrasil applies these techniques before it starts to search general clique and almost clique separators.

Minor-Safe Separators. Bodlaender and Koster have identified a large class of safe separators based on a minor characterization [41]. Recall that a graph H is a minor of a graph G if we can create a graph isomorphic to H by applying a sequence of the following operations to G : delete a vertex v ; delete an edge $\{v, w\}$; contract an edge $\{v, w\}$, that is, delete the edge and replace all occurrences of v or w by a single new element x . We say H is a *labeled minor* of G if the vertices have unique labels and, when performing a contraction, we use the label of either v or w for x .

► Fact 154 ([41])

Let $G = (V, E)$ be a graph and $S \subseteq V$ a separator of G with associated components C_1, \dots, C_q , then S is safe if all $G[V \setminus C_i]$ contain a clique on S as labeled minor. ◀

Let us call a separator *minor-safe* if it is safe with respect to Fact 154. On the positive side, the set of minor-safe separators eventually contains more elements than just the clique and almost clique separators, which means more splitting and thus more parallelization. On the negative side, Fact 154 does not provide us with any hint of how to find a minor-safe separator.

As a rule of thumb, everything that involves minors is computationally hard. Since we wish to use the safe separators as preprocessing, we have to rely on heuristics. In Jdrasil, we use the following simple Monte-Carlo algorithm to find a minor-safe separator in a given graph G . The algorithm is based on a similar algorithm presented by Tamaki for his PACE 2017 submission [158].

```
while attempts < threshold do
  attempts ← attempts + 1
  S ← sampleSeparator(G)
  if not isEventuallyUnsafe(G,S) then
    return S
  end
end
```

The algorithm `sampleSeparator` provides a *candidate* S for a minor-safe separator, and shall guarantee that S is indeed a (not necessarily safe) separator. Multiple calls of the algorithm provide multiple candidates. The algorithm `isEventuallyUnsafe` is a Monte-Carlo algorithm with a one-sided error: If S is not safe by Fact 154, then the algorithm will detect this circumstance with certainty; if S is minor-safe, then the algorithm will eventually identify S as safe separator.

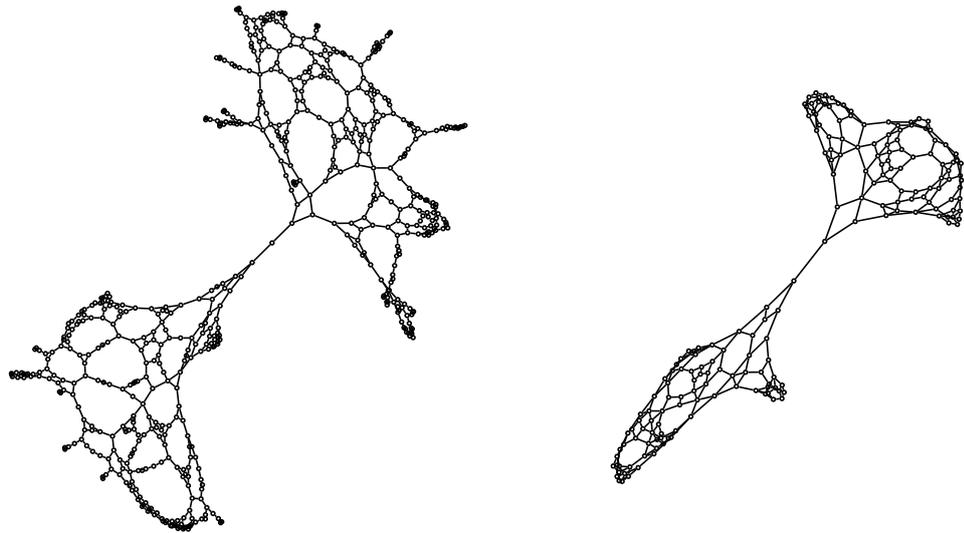
The implementation of `sampleSeparator` in `Jdrasil` follows the approach used in [158]. We manage a pool of separators. If the pool is empty, the algorithm computes a tree decomposition of G using a randomized heuristic and adds all bags that are a separator of G to the pool; if the pool is not empty, `sampleSeparator` just returns and removes an element from the pool.

The implementation of `isEventuallyUnsafe` is more involved. On input of a separator S , the algorithm computes all associated components C_1, \dots, C_q of S and will check whether there is a clique on S in all $G[V \setminus C_i]$ as a labeled minor. To perform the minor test, Tamaki suggests the following approach [158]: Define the set $R = V \setminus (S \cup C)$ and contract the edges in $G[R]$ randomly to obtain a graph B on vertex set $S \cup R'$, where R' contains the vertices that remain after the contractions on R . In a second phase, for each edge $\{u, v\}$ missing in S , a common neighbor $w \in N(u) \cap N(v) \cap R'$ is chosen and contracted either to u or v . The missing edges are processed in order of common neighbors in R' (the less common neighbors, the earlier a missing edge is processed). The choice to which vertex we contract w is done in such a way that the minimal number of common neighbors any remaining edge has is maximized.

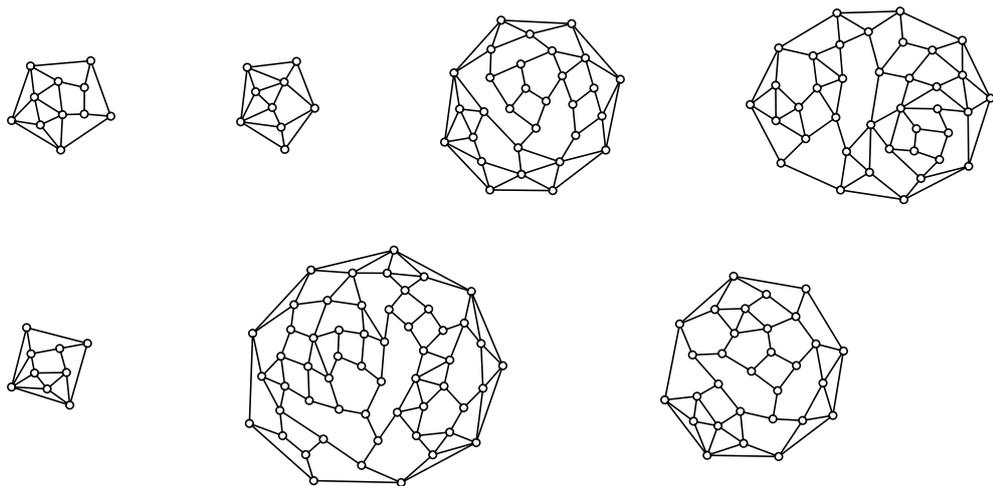
In `Jdrasil` we differ from this approach by trying to find paths that can be contracted. This can be seen as a more greedy and a less random implementation of the above sketched algorithm. The result is a simpler algorithm that performs a little better on the test sets used in Section 10.6: Instead of contracting R , we directly iterate over the missing edges in S in random order. Fixing a missing edge $\{u, v\}$, we compute the set $P = N(u) \cap N(v) \cap C$ of common neighbors of u and v in C . If P is not empty, we choose a random element x of P and contract it at random to either u or v . On the other hand, if P is empty, we compute the shortest path from u to v via breadth-first search and contract it to u . Due to its randomized nature, we repeat the test multiple times for every set S .

► Example 155

The left side of the following graphic shows `ex045.gr`, an instance from PACE 2017 (Testset II) with 600 vertices and 865 edges. On the right, the same graph is shown after the preprocessing routine of `Jdrasil` was applied to it. The reduced graph has still 185 vertices and 342 edges.



Applying the splitter to the reduced graph results in the following seven atoms, which we can handle completely independent of each other. The maximum number of vertices in one of the atoms is 55 and the maximum number of edges 103.



10.6 EXPERIMENTS AND ANALYSIS

In this section we will analyze the performance of *Jdrasil*, and especially of the techniques presented within this chapter, with a series of experiments. We start by studying the performance of the SAT-approach from Section 10.3 with respect to multiple state-of-the-art SAT-solvers on page 144. Once we have decided for a SAT-solver, we will analyze the speedup that we obtain by improving the encoding with the clique and the twin trick on page 145.

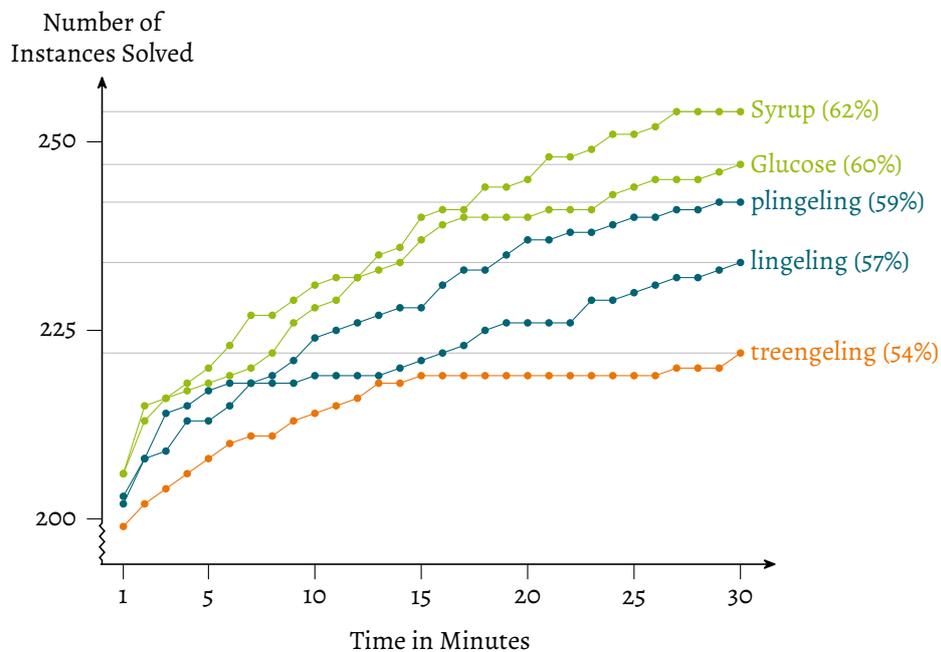
Afterwards, we will examine an implementation of the positive instance driven dynamic program *catchAndGlue* from Section 10.4. We first study the performance of the algorithm if we equip it with the various heuristics presented in Section 10.4 on page 146. Subsequently, we will compare the performance of the SAT-approach and *catchAndGlue* with each other, and with other treewidth algorithms on page 147.

The last two experiments of this section will deal with the parallel capabilities of *Jdrasil*. We will first explore the impact of splitting, both sequentially and in parallel, for various algorithms starting at page 149. Finally, we will measure the speedup and efficiency of the parallel version of *Jdrasil* experimentally with respect to the number of atoms of the input instance on page 152.

Important to note is that for all experiments and all mentioned algorithms (also the ones not discussed within this thesis) we always use the corresponding implementation of that algorithm in *Jdrasil* – no other implementation was used for any experiment. In particular, all algorithms are executed with the exact same set of preprocessing rules: We use the preprocessor of *Jdrasil* in all experiments, but we use splitting only if it is explicitly mentioned. Details about the data sets and the used hardware can be found on page 183, details about other algorithms implemented in *Jdrasil* can be found in [16] and in the manual of *Jdrasil* [15].

SELECTING A SAT-SOLVER

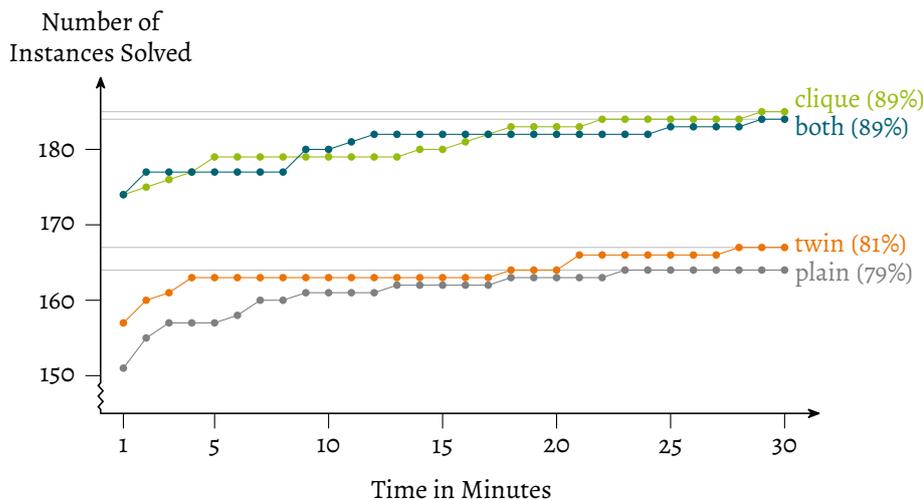
The pleasant thing about encoding our problem into a formula of propositional logic is that, once we have settled the encoding, we can kindly ask a SAT-solver to solve the problem for us. In fact, we can even rely on the expertise of the SAT-community to handle such problems in parallel. The following plot compares the run time of *Jdrasil* using the sequential SAT-solvers *lingeling* and *Glucose*, their parallel versions *plingeling* and *Syrup*, as well as the parallel cube-and-conquer solver *treengeling*, which is based on *lingeling* as well [9, 29, 107]. In all cases, the rest of *Jdrasil* was executed sequentially. The experiment was performed on Machine I with Testset I and II. The following *cactus plot* shows the allowed time t on the x -axis, and the number of instances that the solver was able to solve within t minutes on the y -axis. The number in parentheses is the percentage of solved instances of the test set. All solvers were executed with a timeout of 30 minutes per instance.



The experiment reveals that the selection of the SAT-solver has quite an effect on the overall run time of *Jdrasil*. We can observe that the cube-and-conquer approach of *treengeling* seems not to be feasible for our encoding. On the other hand, the parallel versions *plingeling* and *Syrup* perform notably better than their sequential counter parts *lingeling* and *Glucose*. But more crucially, we can observe that *Glucose* performs better than *lingeling* for our encoding. In fact, the sequential version of it is even better than the parallel solver *plingeling*.

THE GAIN OF ADDING THE CLIQUE AND THE TWIN TRICK

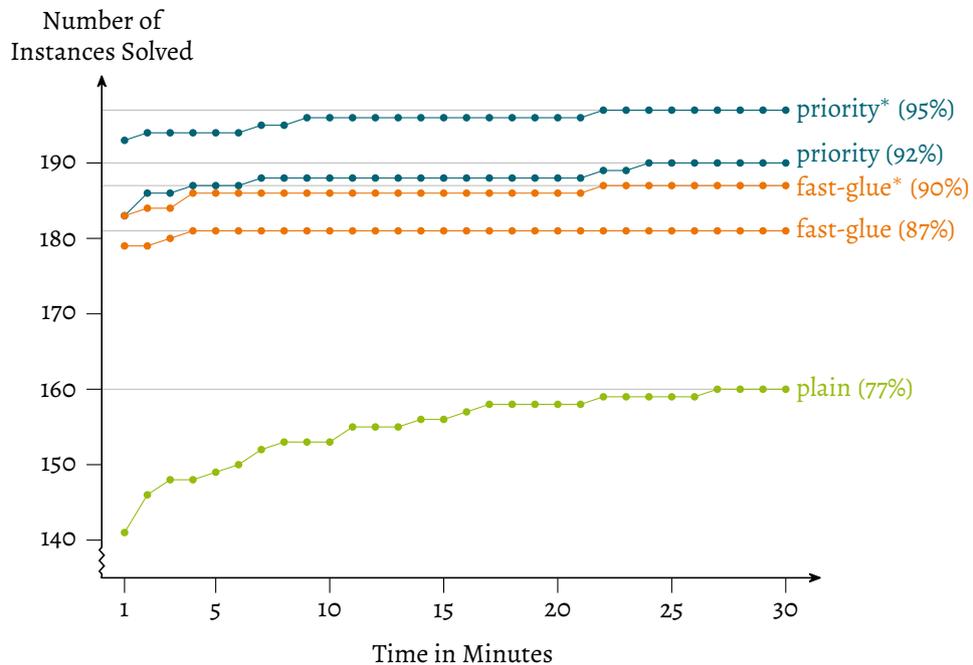
The following cactus plot illustrates the impact of the clique and the twin trick in *Jdrasil* using the SAT-approach with Glucose as underlying SAT-solver (due to the last experiment, this is the best sequential solver for this task). It was obtained from an experiment that was performed on Machine I and Testset I, and with a timeout of 30 minutes per instance. The four curves show the performance of a plain encoding without the tricks, using the clique trick, using the twin trick, and using both tricks.



The experiment reveals that adding the twin trick provides only a slight advantage (but it does overall improve the encoding). In contrast, adding the clique trick boosts the performance of the encoding such that it is able to solve 10% more instances of the test set within the 30 minutes timeout. Interestingly, combining both tricks does not provide any further advantage and, in fact, results in an encoding that performs slightly worse than the encoding that uses just the clique trick. I assume this is because both tricks improve the performance by adding some sort of symmetry breaking to the formula. Since in most instances there are only very few twins, adding these to the encoding does not provide more symmetry breaking than what we already get from the added clique and, thus, by using both tricks we increase the formula without any gain on many instances.

SELECTING A HEURISTIC TO SPEED UP
POSITIVE INSTANCE DRIVEN DYNAMIC PROGRAMMING

In this experiment we study the performance of the catchAndGlue algorithm from Section 10.4 when it is equipped with various heuristics. The experiment was performed on Machine I using Testset I with a timeout of 30 minutes per instance. In the following, we compare the performance of the **plain algorithm**, the algorithm using the **priority heuristic**, and the algorithm using the **fast-glue heuristic**. Adding the fast-contamination heuristic, pruning, or the potential-maximal-clique heuristic alone, increases the performance only slightly – the effect is too small to be visible in the cactus plot and, thus, the corresponding curves are omitted. However, adding all three heuristics to either the priority or the fast-glue heuristic provides an additional acceleration. The corresponding curves are marked with an asterisk.



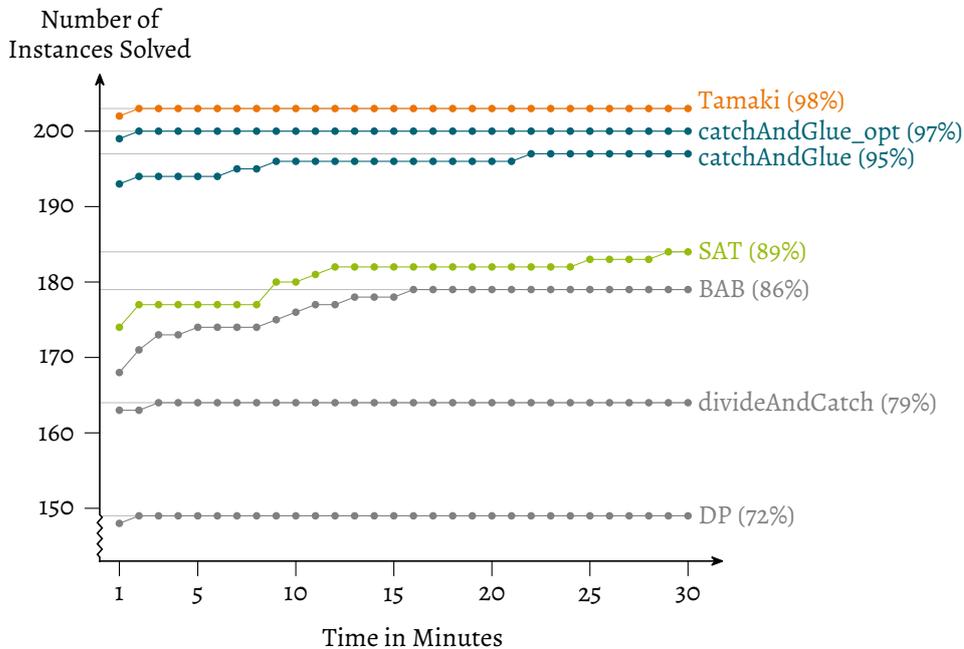
As we can see, the plain algorithm already performs decently. However, adding either the priority or the fast-glue heuristic improves the performance a lot. In fact, by using these heuristics the algorithm from Section 10.4 is able to solve 87% of the instances of the test set within the given timeout of 30 minutes. Since the priority heuristic is simpler to implement and performs slightly better than the fast-glue heuristic, I would recommend to always use it. Combining both heuristics does *not* improve the performance further and results in an algorithm that is actually slightly worse than using just the priority heuristic (not shown in the plot). This is not surprising, as both heuristics have the same goal: Generate large configurations quickly. Since, furthermore, both heuristics achieve this in a similar way, we would add unnecessary overhead by using both heuristics.

COMPARISON OF THE DIFFERENT ALGORITHMS

In this experiment we compare the best configuration (with respect to the previous experiments) of the **SAT-approach** and the positive instance driven dynamic program **catchAndGlue** against each other. The plot contains in gray the algorithms which were considered state-of-the-art *before* the first Parameterized Algorithms and Computational Experiments Challenge: a branch-and-bound algorithm (BAB) [99], the divideAndCatch algorithm on which we based our positive instance driven dynamic program in Section 10.4, as well as a “Held-Karp-like” dynamic program (DP) due to Bodlaender, Fomin, Koster, Kratsch, and Thilikos [36]. The diagram also contains **Tamaki’s** positive instance driven dynamic program that is based on minimal separators and potential maximal cliques [158].

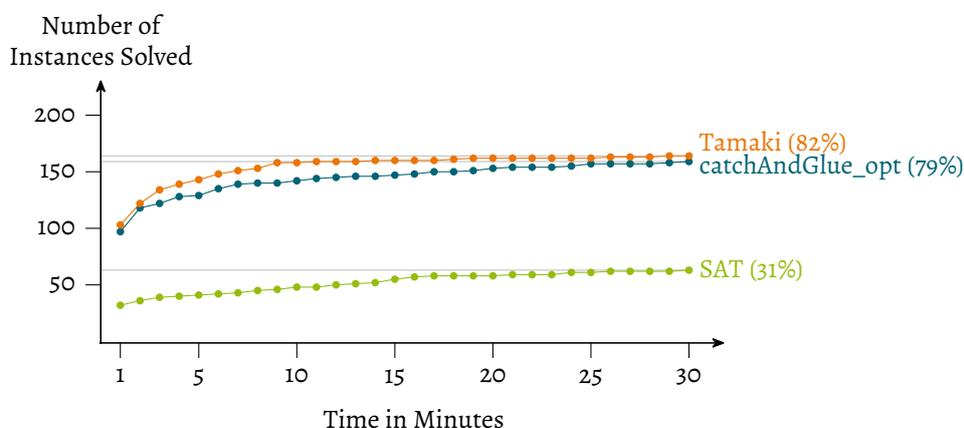
I used for all algorithms the corresponding implementation of *Jdrasil*, which were all highly optimized. However, **catchAndGlue** is not optimized and refers to a direct implementation of the code presented within this thesis. The performance can be improved by carefully choosing data structures – for instance, the set $\mathcal{B}(Q)$ can be managed in a set-trie [150]. The optimized version of **catchAndGlue**, which is actually used by *Jdrasil*, is denoted by **catchAndGlue_opt** in the following plot.

The experiment was performed on Machine I and Testset I with a timeout of 30 minutes per instance. As mentioned above, all tests were performed with *Jdrasil’s* implementation of the corresponding algorithms. For each of them, the preprocessing of *Jdrasil* was applied, but splitting was deactivated.



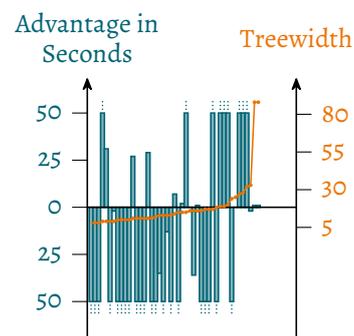
The experiment reveals that both, our algorithm using the improved SAT-encoding and our positive instance driven dynamic program catchAndGlue, outperform all the algorithms that were considered state-of-the-art before the first PACE. However, the experiment shows that the positive instance driven dynamic program by Tamaki solves 4 instances more than the optimized version of catchAndGlue. I assume that these instances contain just a few minimal separators or potential maximal cliques, as Tamaki’s algorithm works directly with these objects.

The following figure illustrates the same experiment for the three fastest algorithms repeated on Testset II, which contains the instances of the second iteration of the PACE. The instances in this data set are larger and more difficult than the instances in Testset I. However, they contain fewer symmetries as they are generated from real world graphs, which makes them more vulnerable to preprocessing.



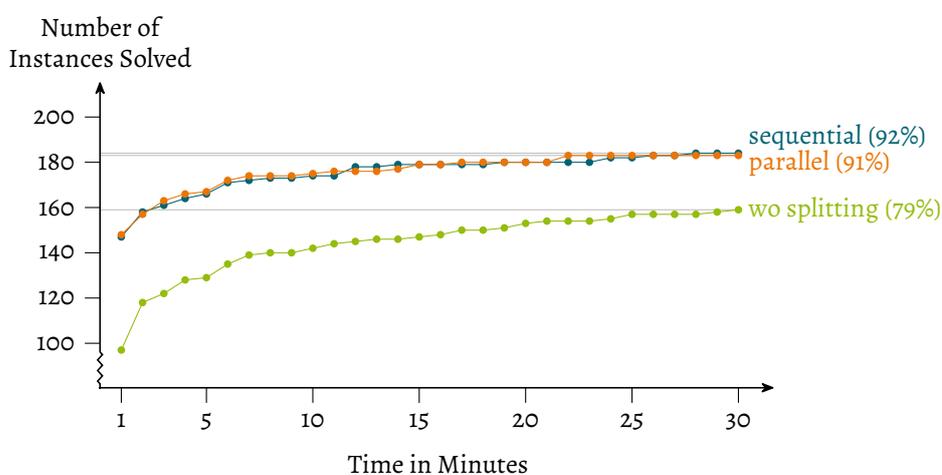
The plot shows that this test set is indeed more challenging, as all solvers only solve a smaller percentage of the instances within the time bound. We can observe that the advantage of the positive instance driven algorithms over the SAT-approach become more apparent compared to the first experiment.

An interesting finding is that there are a couple of instances that are still solved faster by the SAT-approach than by catchAndGlue. In particular, the instances of higher treewidth seem to be good candidates for this effect. The *advantage plot* on the right shows the advantage of the SAT-approach over catchAndGlue. To produce it, I took all instances that were solved by *both* solvers, ordered them by their treewidth, and used the instances as x -coordinate. For each instance, there is a bar that indicates the *advantage*: A positive bar means that the SAT-approach is faster by the length of the bar, while a negative bar means that catchAndGlue is faster. All bars are capped at 50 seconds, and the *curve* visualizes the treewidth of the instances.



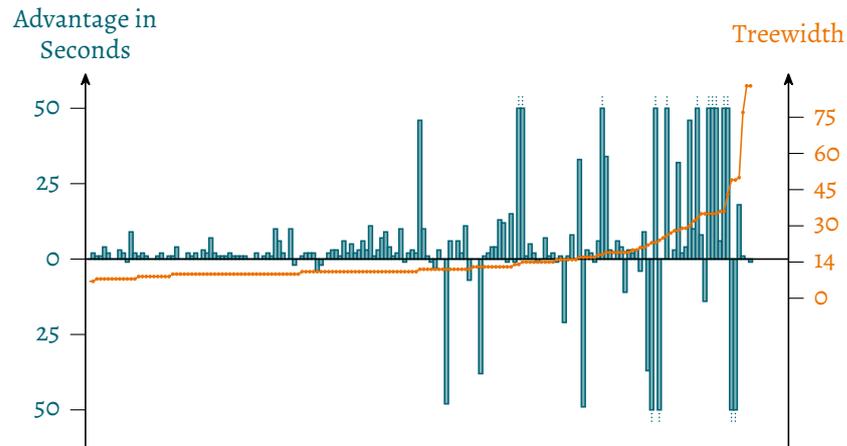
PARALLELIZATION VIA SPLITTING

In this experiment we examine the power of splitting when computing optimal tree decompositions using the optimized catchAndGlue algorithm. In particular, we will study how well these techniques can be used for parallelization. The following experiments were again executed on Machine I, which is equipped with 8 cores. I compared catchAndGlue **with sequential splitting** (atoms are handled sequentially), **with parallel splitting** (all atoms are handled in parallel, but everything else is still sequential), and **without splitting** (sequential algorithm that does not compute atoms at all) on Testset II. As before, I used a 30 minutes timeout per instance. The results are visualized in the following cactus plot:

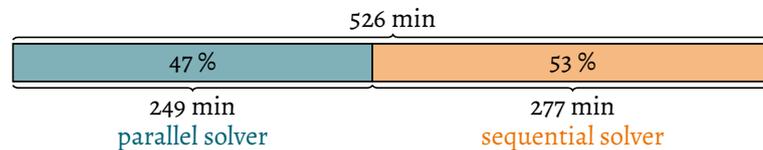


We can see that splitting allows the solver to solve about 20 instances more compared to the solver that does not use splitting. Concerning parallelization, the positive message of the plot is that, even though the parallel version solves one instance less than the sequential version, it is generally equal or faster than the sequential one. That the solver does not become worse when using parallelization is due to the fact that the splitting is implemented in a work optimal way, that is, the sequential solver performs the same amount of operations to apply splitting as the parallel solver. However, the resolution of the cactus plot is not high enough to see whether or not we obtain a speedup using parallelization.

The advantage plot on the next page shows the advantage of the parallel version over the sequential version. As in the experiment in which we compared various solvers, I took only the instances that were solved by both solvers, ordered them by their treewidth, and used them as x -coordinates. For each instance, there is a bar that indicates whether the parallel solver is better (by the amount of a positive bar), or whether the sequential solver is better (by the amount of a negative bar). The **curve** illustrates the treewidth of the instances.

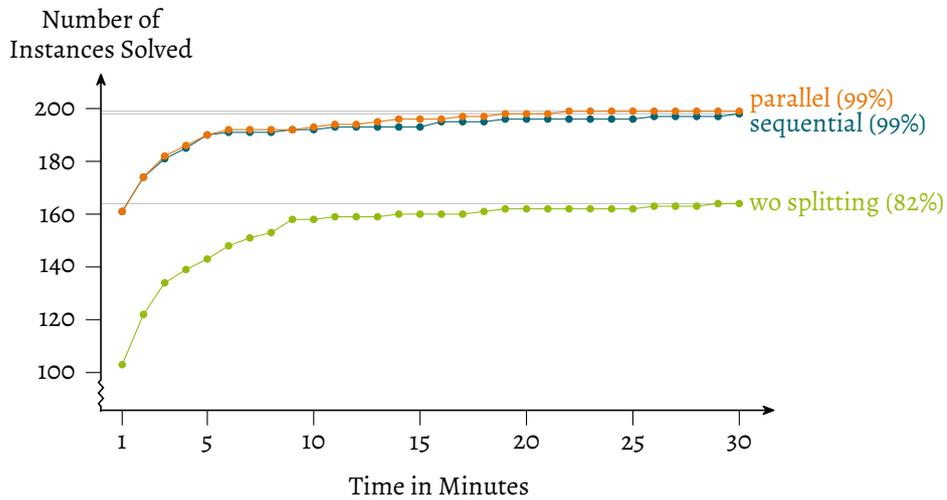


This plot shows that the difference of the sequential and the parallel version is negligible on most of the instances. However, if there is a meaningful difference for some instance then it is in most cases in favor of the parallel algorithm. Interestingly, especially the graphs with higher treewidth seem to be well suited for parallelization. The following *domination plot* visualizes the total time used by the **parallel solver** and used by the **sequential solver**. To generate the plot, I used only the instances that were solved by both solvers. The length (total time) is the sum of the run times of both solvers (that is, the time the whole experiment took), while the individual times are the times of the corresponding solvers.

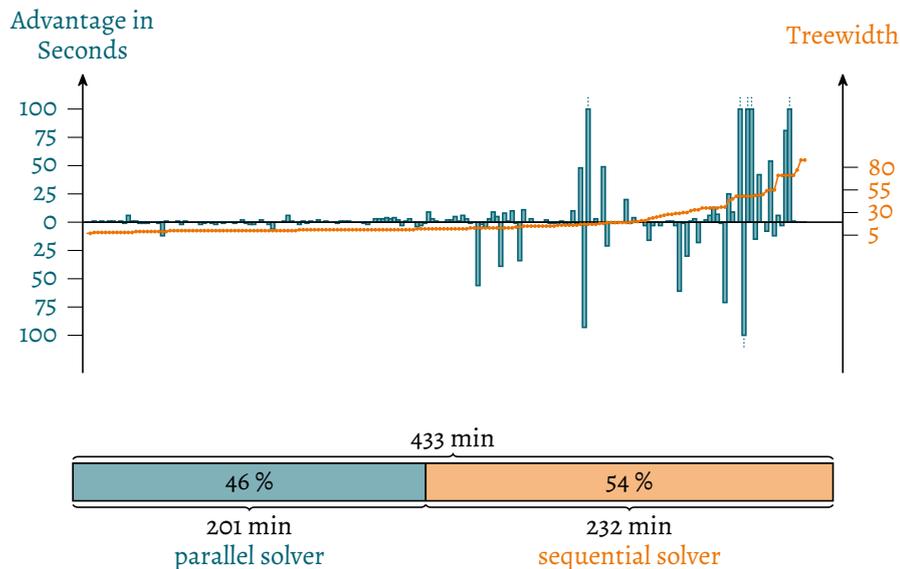


As expected from the previous plot, the figure underpins that the parallel version is overall faster than the sequential version. In detail, the parallel version of the program is able to solve the whole test set 25 minutes faster than the sequential solver.

Due to Jdrasil's architecture, we can not only execute catchAndGlue in parallel, but also any other algorithm. For the following plot I used Tamaki's algorithm with **sequential splitting**, with **parallel splitting**, and **without splitting**. The experiment was performed on Machine I and Testset II, with a timeout of 30 minutes per instance.



We can observe that, using splitting, Tamaki's algorithm is able to solve almost the whole testset within the given time; and that the parallel version is again a bit faster. The following advantage plot and domination plot highlight the speedup obtained in parallel: The parallel version is able to solve the whole test set 30 minutes faster than the sequential one.



SPEEDUP AND EFFICIENCY

In this last experiment we will study experimentally the speedup and efficiency we gain by the parallel version of *Jdrasil*. From a theoretical point of view, the *speedup* we obtain using p processors is usually defined as $S_p(n) = T_1(n)/T_p(n)$, where $T_i(n)$ is the (theoretical worst case) time the algorithm requires to solve an instance of size n when it uses i processors. Accordingly, the *efficiency* of the parallel algorithm is defined as $E_p(n) = T_1(n)/(p \cdot T_p(n))$. However, these definitions are not well suited for our type of parallelization, as the splitting strategy uses a structural property (the number of atoms) in order to apply parallelization. Therefore, for any n there will be instances where we have no speedup up at all (they have just one atom), or a high speedup (they have many atoms). To circumnavigate this problem, we study the speedup and the efficiency with respect to the number of atoms a , rather than with respect to n : $S_p(a) = T_1(a)/T_p(a)$ and $E_p(a) = T_1(a)/(p \cdot T_p(a))$, where $T_i(a)$ is the (experimental) time to solve all instances with a atoms of a given test set. Note that it does not make sense for the splitting approach to consider $p > a$ (we can not parallelize more than handle each atom on an individual processor) and, thus, the best speedup we can hope for is a .

The following table shows the distribution of instances with a certain number of atoms³ in Testset II. It contains, however, only instances with at least two atoms, and only atom classes that contain at least 2% of the instances of the test set. For every atom class, the table shows the time *Jdrasil* requires to solve all instances of this class given a certain number of processors on Machine I.

Number of Atoms	Instances	Time using p Processors						
		1	2	3	4	5	6	7
2	19%	4967s	3534s	–	–	–	–	–
3	13,5%	627s	502s	471s	–	–	–	–
4	3%	85s	81s	81s	79s	–	–	–
5	2%	43s	37s	50s	36s	37s	–	–
7	2%	41s	28s	32s	30s	31s	19s	17s

³The number of atoms refers to the number of atoms generated by *Jdrasil*'s heuristics, and not to the (unknown) optimal number of such objects.

The times collected in the previous table result in the experimental speedups and efficiencies that are illustrated in the following table:

Atoms	Speedup / Efficiency using p Processors					
	2	3	4	5	6	7
2	1,40 / 0,70	–	–	–	–	–
3	1,24 / 0,62	1,50 / 0,44	–	–	–	–
4	1,04 / 0,52	1,04 / 0,34	1,07 / 0,26	–	–	–
5	1,16 / 0,58	0,86 / 0,28	1,19 / 0,29	1,16 / 0,23	–	–
7	1,46 / 0,73	1,28 / 0,42	1,36 / 0,34	1,32 / 0,26	2,15 / 0,35	2,41 / 0,34

The data shows that we obtain a speedup in general, although it is comparable small to the one we might expect from theory – the sole exception is the set of instances with 5 atoms solved using 3 processors, where the parallel version is slower than the sequential one. The reason for the comparatively low speedup is that for an optimal speedup all atoms of an instance would have to have the same size. However, many instances contain a single large atom together with a collection of smaller atoms. This phenomenon also explains the above average efficiency of using two processors: One processor can handle the big atom, while the other one can handle all the small atoms within the same time.

11 JATATOSK: A LIGHTWEIGHT MODEL CHECKER FOR A FRAGMENT OF MSO

In this chapter we will explore a possible path to handle monadic second-order logic model checking in practice. A model checker is a tool that, given a logical structure S and a formula φ , tests whether or not $S \models \varphi$ holds – we have studied such algorithms in Chapter 8. In practice, we usually require that, in the case of $S \models \varphi$, the model checker also outputs an assignment of the free variables of φ . For convenience, we further require an assignment to existentially bounded variables that are not in the scope of a universal quantifier.

A model checker can be seen as convenient interface to dynamic programming over tree decompositions. Such an interface is highly desirable from a parallel point view, as dynamic programs over tree decompositions offer a way to solve combinatorial problems in parallel (which is otherwise often a difficult task). This is particularly true in practice, as such dynamic programs have many points that can be parallelized on different architectures [139]. For instance, if we have multiple CPU cores or even a cluster, we can split the tree decomposition and handle multiple subtrees in parallel. If we have an architecture that allows massive parallelization (such as a GPU), we can parallelize a single step of the dynamic program (transforming the table of one bag to the table of the next bag). The later approach was successfully applied to SAT, a problem that otherwise is hard to parallelize as well [82].

Unfortunately, using the techniques developed in Chapter 8 directly in practice is difficult, as the generated tree automata are huge. Instead, in this chapter I introduce a fragment of MSO that can be checked more directly. In particular, we can model check formulas of this fragment in an “event-driven” way, in which the dynamic program has to perform most of its computations only on edge-bags.

For this fragment I present the model checker *Jatatosk*, which is implemented on top of the dynamic programming interface of *Jdrasil* – therefore the name, which is a portmanteau of “Java” and “Ratatosk,” the squirrel that runs up and down on Yggdrasil to transmit messages from the animals at the crown of the tree to those sitting at its roots. The current implementation of *Jatatosk* is sequential, as it aims to explore the power of the fragment and the viability of the approach. However, it is implemented with an architecture that has parallelization in mind such that it can be parallelized in the future. *Jatatosk* is publicly available at GitHub [12], where the reader will find a manual that guides through the first steps of using this tool.

11.1 THE AIM OF JATATOSK

The primary goal of *Jatatosk* is, of course, a happy marriage of theoretical powerful tools – such as Courcelle’s Theorem – with practice. Similar attempts can be found in the literature. For instance, Sequoia is a full MSO-model checker based on a game theoretic characterization of Courcelle’s Theorem [117, 126]. Another contender is D-Flat, which is an Answer Set Programming (ASP) solver for problems of small treewidth [30], which allows an implementation of Courcelle’s Theorem as well [31].

So why do we need another tool for that task? Both tools, Sequoia and D-Flat, solve the general problem of MSO-model checking, which is arguably a very hard task. While both of them perform decently on real world instances – in particular for problems that require a connected solution – they are not yet fully competitive against modern SAT- or ILP-solvers [118]. In fact, an MSO-model checker can probably not run in time $f(|\varphi| + tw) \cdot \text{poly}(n)$ for any elementary function f [91]. Even worse, it is usually unclear what the concrete run time of such a tool is for a specific formula φ . To that end, the authors of Sequoia have performed a sophisticated analysis of their tool to obtain worst-case bounds for standard formulas [118].

To tackle all these problems, the design principle of *Jatatosk* is the focus on a fragment of MSO. The result is a streamlined solver that is comparatively faster and easier to analyze. In particular, the run time of *Jatatosk* can directly be derived from the *syntax* of the input formula.

11.2 A HIGH-LEVEL VIEW ON THE TOOL

The main idea of the architecture of *Jatatosk* is an *event-driven* evaluation of the formula. This event-driven approach is inspired by the classical *guess-and-verify* approach used in nondeterministic computations. For instance, consider the problem of testing whether a graph can be properly colored with three colors: A nondeterministic computation would first guess the coloring, which means assigning a color to each vertex; and would secondly verify it, which means checking for every edge whether the endpoints have different colors. This strategy can be implemented as a dynamic program over tree decompositions (compare with Example 137): At every introduce-bag we *guess* a color for the introduced vertex, and on every edge-bag we *verify* that the two endpoints have obtained different colors. In this particular case, forget- and join-bags actually do nothing more than a little bookkeeping, and the run time and correctness of the algorithm is almost immediate. *Jatatosk* is event-driven in the sense that it will evaluate the formula only on edge-bags (so these bags trigger an event), while all other bags are handled in a very uniform way.

In order to make the event-driven approach work in general, we will restrict the allowed input structures slightly. In general, *Jatatosk* expects as input a relational vocabulary $\tau = (R_1, R_2, \dots, R_r)$ as well as a τ -structure S . We require, however, that τ contains a binary relation E , and that S interprets E in a symmetric way. That is, S has to contain an undirected graph as substructure. *Jatatosk* will compute a tree decomposition of this graph, and the edges of this graph will trigger events in edge-bags. Note in particular that this stands in contrast to the other standard approach of using the Gaifman Graph (see Definition 6).

Internally, *Jatatosk* computes a tree decomposition using *Jdrasil*, and implements a tree automaton that evaluates the formula using the dynamic programming interface of *Jdrasil*. Every state of the automaton is represented as *bit vector* (sometimes also called *bit set*). The bits of these vectors essentially describe for every vertex in the current bag, to which variables of the formula it is assigned (the details will be explained in the following sections). We will choose the fragment of MSO in a way that allows us to deduce from the syntax of the formula the required size s of the bit vectors as well as the way in which we have to modify the bits on different bag-types – without explicitly constructing the automaton. In this way, we can directly deduce the run time of the algorithm from the syntax of the formula as well: It will be of magnitude 2^s .

11.3 DESCRIPTION OF THE FRAGMENT

We seek a fragment of monadic second-order logic that can be checked with the sketched event-driven approach. First observe that existential second-order quantifiers are “easy,” as we have already seen in the example of graph coloring. Therefore, our fragment contains only formulas of the following form, where Ψ is a first-order formula defined later:

$$\exists X_1 \exists X_2 \dots \exists X_q \Psi.$$

There is an easy implementation for such quantifiers. Assume we work on a tree decomposition of bag-size k , then a state can be represented by $k \cdot q$ bits that simply indicate which vertex in the current bag is in which sets. On introduce-bags we guess the sets X_i to which the introduced vertex shall be assigned, and we set the corresponding bits; in forget-bags we just have to clear the bits of the forgotten vertex; on edge-bags we do not have to do anything; and on join-bags we have to ensure that the bit vectors are the same for both children.

Observe that, from an event-driven perspective, the representation with $k \cdot q$ bit can be very lavish. For instance, consider the following formula:

$$\varphi_{\text{example}} = \exists R \exists B \forall x (R(x) \leftrightarrow \neg B(x)).$$

In this example R and B constitute a partition of the vertices and, thus, one bit per vertex would be sufficient. An event-driven approach will notice this too late, as it will evaluate the formula after it has assigned vertices to R and B. Since Jatoskos has an exponential run time with respect to the number of bits in the states, this is a rather big deal. Therefore, the fragment also contains *partition quantifier*, which have the following semantic and can be implemented by using only $k \cdot \log q$ bits:

$$\exists^{\text{partition}} X_1, \dots, X_q \equiv \exists X_1 \exists X_2 \dots \exists X_q (\forall x \bigvee_{i=1}^q X_i(x)) \wedge (\forall x \bigwedge_{i=1}^q \bigwedge_{j \neq i} \neg X_i(x) \wedge \neg X_j(x)).$$

A second observation about event-driven evaluation is that we can check multiple formulas at every event independently. Therefore, Ψ has the form $\Psi = \bigwedge_{i=1}^{\ell} \psi_i$. The interesting part of the description are the formulas ψ_i . The natural formula for the event-driven approach is:

$$\psi_i = \forall x \forall y E(x, y) \rightarrow \chi_i,$$

where χ_i is some quantifier-free first-order formula. Observe that these are exactly the formulas we need to express graph coloring; and observe that we can check such formulas directly on edge-bags without storing any further information in the state of the automaton. Apart from optimization (which we handle later), this fragment can already express graph coloring, vertex cover, and independent set:

$$\begin{aligned} \phi_{\text{3col}} &= \exists^{\text{partition}} R, G, B \forall x \forall y E(x, y) \rightarrow \bigwedge_{C \in \{R, G, B\}} \neg C(x) \vee \neg C(y); \\ \phi_{\text{vc}} &= \exists S \forall x \forall y E(x, y) \rightarrow (S(x) \vee S(y)); \\ \phi_{\text{is}} &= \exists S \forall x \forall y E(x, y) \rightarrow (\neg S(x) \vee \neg S(y)). \end{aligned}$$

There are many problems for which it is not sufficient to check whether a property holds on every edge. Instead, such problems often require that *some* edges satisfy the property. Consider for instance the dominating set problem in reflexive graphs, which we can express with the following formula (again omitting optimization):

$$\varphi_{\text{ds}} = \exists S \forall x \exists y E(x, y) \wedge S(y).$$

We can still check such formulas in an event-driven way, but we cannot simply reject if some edge does not satisfy the formula. Therefore, we now have to reserve some bits in the state of the automaton to describe whether we have already seen the correct y for a vertex x . In particular, we store k bits and, whenever a vertex is introduced, we set the corresponding bit to 0. For each edge $\{x, y\}$ we check whether or not the formula is true and, if so, set the bit for x . Furthermore, whenever we forget a vertex x we have to check if its corresponding bit was already set to 1, otherwise we have to reject the current state. Finally, observe that in join-bags we cannot simply reject anymore if the bit vectors are different. For the bits reserve for this formula, we have to propagate the logical-or of both bit vectors, as we may have seen the edge in one of the two subtrees.

In a very similar way, we can also check formulas of the form $\exists x \exists y E(x, y) \wedge \chi_i$, $\exists x \forall y E(x, y) \rightarrow \chi_i$, $\forall x \exists y E(x, y) \wedge \chi_i$, $\exists x \chi_i$, and $\forall x \chi_i$. This finalizes the description of the fragment, which contains formulas of the form $\exists X_1 \dots \exists X_q \bigwedge_{i=1}^{\ell} \psi_i$ where the ψ_i are formulas of the following form:

$$\psi_i \in \{ \forall x \forall y E(x, y) \rightarrow \chi_i, \forall x \exists y E(x, y) \wedge \chi_i, \exists x \forall y E(x, y) \rightarrow \chi_i, \\ \exists x \exists y E(x, y) \wedge \chi_i, \forall x \chi_i, \exists x \chi_i \}.$$

Here, all χ_i are quantifier-free first-order formulas. For convenience, we require that they are in conjunctive normal form. Note that in the literature, the atom $E(x, y)$ used in the above formulas is sometimes called a *guard*.

11.4 EXTENSIONS OF THE FRAGMENT

As the formulas from the previous section already indicate, performing model checking alone will not be sufficient to express many natural problems. In fact, every graph is a model of the formula φ_{vc} if the vertex cover simply contains all vertices. On the other hand, many interesting problems can be expressed in monadic second-order logic, but not within our fragment. The solution to both problems is an extension of the fragment by operations that behave “nicely” during the model checking process.

An Extension to Optimization. The optimization version of the model checking problem is usually formulated as follows [59, 85]: Given a logical structure S , a formula $\varphi(X_1, \dots, X_p)$ of the MSO-fragment defined in the previous section with free second-order variables X_1, \dots, X_p , and weight functions $\omega_1, \dots, \omega_p$ with $\omega_i: V \rightarrow \mathbb{Z}$; the task is to find sets S_1, \dots, S_p with $S_i \subseteq V$ such that $\sum_{i=1}^p \sum_{s \in S_i} \omega_i(s)$ is *minimized* under $S \models \varphi(S_1, \dots, S_p)$, or conclude that S is not a model for φ for any assignment of the free variables. We can express the (actually *weighted*) optimization version of vertex cover as follows: $\varphi_{vc}(S) = \forall x \forall y E(x, y) \rightarrow (S(x) \vee S(y))$.

We can, of course, *maximize* the term $\sum_{i=1}^p \sum_{s \in S_i} \omega_i(s)$ by simply multiplying all weights with -1 . In that way, we can turn the formula to find an independent set into a correct one.

The implementation of such an optimization is straightforward: There is an existential quantifier for every free variable X_i of the formula, and we assign the current value of $\sum_{i=1}^p \sum_{s \in S_i} \omega_i(s)$ to every state of the automaton. This value is adapted if elements are “added” to some free variables at introduce nodes. Note that, since we optimize an affine function, this does not increase the state space: Even if multiple computational paths lead to the same state with different values at some node of the tree, it will be well defined which of these values is the optimal one. Therefore, we have to reserve only k bits in the description of the states of the automaton per free variable – independently of the weights.

Quantifier Extensions. Many interesting properties, such as graph connectivity, can be expressed in monadic second-order logic. For instance, that a set X is connected in graph theoretic terms can be expressed by the following formula (compare with Example 14 in Section 2):

$$\begin{aligned} \varphi_{\text{connected}}(X) &= \forall Y (\exists x \exists y X(x) \wedge X(y) \wedge Y(x) \wedge \neg Y(y)) \\ &\rightarrow (\exists x \exists y X(x) \wedge X(y) \wedge Y(x) \wedge \neg Y(y) \wedge E(x, y)). \end{aligned}$$

There are two flaws with this formula. First of all, it is rather long for a “simple” statement, and it is (probably) only for logicians a natural way to express connectivity. In the light of Jatosk, the even bigger problem is that the formula is not part of the fragment we work on. On the other hand, “guessing a connected set” is actually a well understood topic on tree decompositions [59]. In order to make this power available to the user, we introduce a quantifier extension to our fragment. We add the *connected quantifier*, which guesses a set X that is connected in graph theoretic terms with respect to the relation E and, thus, which has the following semantic:

$$\exists^{\text{connected}} X \psi(X) \equiv \exists X (\varphi_{\text{connected}}(X) \wedge \psi(X)).$$

Implementing the connected quantifier is considerably harder than implementing a classical existential quantifier. The automaton has to overcome the difficulty that an introduced vertex may not be connected to the rest of the bag in the moment it got introduced, but may be connected to it when further vertices “arrive.” The solution to this dilemma is to manage a partition of the bag into $k' \leq k$ connected components $P_1, \dots, P_{k'}$, for which we reserve $k \cdot \log_2 k$ bit in the state description. Whenever a vertex v is introduced, the automaton either guesses that it is not contained in X and clears the corresponding bits, or it guesses that $v \in X$ and assigns some P_i to v . Since v is isolated in the bag in the moment of its introduction (recall that we work on a very nice tree decomposition), it requires its own component and is therefore assigned to the smallest empty partition P_i . When a vertex v is forgotten, there are four possible scenarios: (i) $v \notin X$, then the corresponding bits are already cleared and nothing happens; (ii) $v \in X$ and $v \in P_i$ with $|P_i| > 1$, then v is removed and the corresponding bits are cleared; (iii) $v \in X$ and $v \in P_i$ with $|P_i| = 1$ and there are other vertices w in the bag with $w \in X$, then the automaton rejects the configuration, as v is the last vertex of P_i and may not be connected to any other partition anymore; (iv) $v \in X$ is the last vertex of the bag that is contained in X , then the connected component is “done,” the corresponding bits are cleared and one additional bit is set to indicate that the connected component cannot be extended anymore. When an edge $\{u, v\}$ is introduced, components might need to be merged. Assume $u, v \in X$, $u \in P_i$, and $v \in P_j$ with $i < j$ (otherwise, an edge-bag does not change the state), then we essentially perform a classical union-operation from the well-known union-find data structure. Hence, we assign all vertices that are assigned to P_j to P_i . Finally, at a join-bag we may join two states that agree locally on the vertices that are in X (they have assigned the same vertices to some P_i),

however, they do not have to agree in the way the different vertices are assigned to P_i (in fact, there does not have to be an isomorphism between these assignments). Therefore, the transition at a join-bag has to connect the corresponding components analogously to the edge-bags.

Adding the connected quantifier to the fragment allows us to describe many problems, which otherwise require sophisticated formulas, in a natural way. Besides the obvious fact that we can now express problems such as connected vertex cover, we can also describe more involved problems. For instance, the following sentence is true whenever the graph contains a triangle as minor (and thus contains a cycle), and it is easy to see that the sentence can be extended to describe that any fixed graph H is a minor of the input graph:

$$\begin{aligned} \varphi_{\text{triangle-minor}} = & \exists^{\text{connected}} R \exists^{\text{connected}} G \exists^{\text{connected}} B . \\ & (\forall x (\neg R(x) \vee \neg G(x)) \wedge (\neg G(x) \vee \neg B(x)) \wedge (\neg B(x) \vee \neg R(x))) \\ & \wedge (\exists x \exists y E(x, y) \wedge R(x) \wedge G(y)) \wedge (\exists x \exists y E(x, y) \wedge G(x) \wedge B(y)) \\ & \wedge (\exists x \exists y E(x, y) \wedge B(x) \wedge R(y)). \end{aligned}$$

With similar tools as for the connected quantifier, we introduce the *forest quantifier* that guesses a cycle free set X (in graph theoretic terms with respect to the relation E). Its implementation is almost identical to the one of the connected quantifier: We manage a partition of the bag into $P_1, \dots, P_{k'}$ and, at introduce-bags, guess whether the introduced vertex is part of the forest or not (giving it its singleton partition eventually). On edge-bags and join-bags, we perform the same union-find operation, but we additionally reject the state if this operation creates a cycle. Forget-bags can even be handled more easily: Here we just have to clear the bits, as the forgotten vertex may not be part of any cycle in the future.

The main and natural application of the forest quantifier is the problem of finding a feedback-vertex set in the input graph. This task can now be naturally expressed with the following formula:

$$\varphi_{\text{fvs}}(S) = \exists^{\text{forest}} F \forall X (S(x) \vee F(x)).$$

11.5 PREDICTING THE RUN TIME AND EXPERIMENTS

We have defined the fragment and have discussed how to handle optimization and powerful quantifier extensions. For each formula and quantifier that we have introduced, we have reserved some bits in the state description of the automaton. In this section we analyze the exact performance of *Jatatosk* with respect to the number of reserved bits. For that end, let $\text{bit}(\varphi, k)$ be the number of bits we reserve for the input formula φ on a tree decomposition of bag-size k .

A nondeterministic tree automaton will process a labeled tree with n nodes in time $O(n)$. If the automaton has state set Q , one might think that a running time of the form $O(|Q| \cdot n)$ is sufficient to simulate the automaton deterministically, as the automaton could be in any potential subset of the states at some node of the tree. However, there is a pitfall: For every node we have to compute the set of potential states of the automaton depending on the sets of potential states of its children, leading to a quadratic dependency on $|Q|$. In detail, let x be a node with children y and z and let Q_y and Q_z be the set of potential states in which the automaton is at these nodes. To determine Q_x , we have to check for every $q_i \in Q_y$ and every $q_j \in Q_z$ if there is a $p \in Q$ such that $(q_i, q_j, t(x), p) \in \Delta$. Note that the number of states $|Q|$ can be quite large even for moderately sized parameters k , as $|Q|$ is typically of size $2^{\Omega(k)}$. Therefore, we will try to avoid this quadratic blow-up.

The crucial observation is that, often, a tree automaton will only continue if both children are in the same state. For instance, this is the case for the automaton that checks whether a graph can be colored with three colors. We call automata with this property *symmetric*:

► Definition 156 (Symmetric Tree Automaton)

A *symmetric* nondeterministic bottom-up tree automaton is a nondeterministic bottom-up tree automaton $A = (Q, \Sigma, \Delta, F)$ in which all transitions $(l, r, \sigma, q) \in \Delta$ satisfy either $l = \perp$, $r = \perp$, or $l = r$. ◀

Assume as before that we wish to compute the set of potential states for a node x with children y and z . Observe that in a symmetric tree automaton it is sufficient to consider the set $Q_y \cap Q_z$ and that the intersection of two sets can be computed in linear time if we choose the underlying data structures carefully.

► Observation 157

A symmetric tree automaton can be simulated in time $O(|Q| \cdot n)$.

By the above observations it follows that, if the automaton that we construct is symmetric, then J_{atatosk} will run in time $O^*(2^{\text{bit}(\varphi, k)} \cdot n)$, but it will only run in time $O^*((2^{\text{bit}(\varphi, k)})^2 \cdot n)$ otherwise. Unfortunately, not all formulas will yield a symmetric automaton. To overcome this issue, we partition the bits of the state description into “symmetric bits” and “asymmetric bits.” The idea is that, if we would only have symmetric bits, then the automaton would be symmetric as well. In particular, independently of the asymmetric bits, the symmetric bits must be identical in the states of all children. Let $\text{symmetric}(\varphi, k)$ and $\text{asymmetric}(\varphi, k)$ be defined analogously to $\text{bit}(\varphi, k)$. We implement the join of states as in the following lemma, allowing us to deduce the running time of the model checker for concrete formulas.

► Lemma 158

Let x be a node of T with children y and z , and let Q_y and Q_z be sets of states in which the automaton may be at y and z . Then the set Q_x of states in which the automaton may be at node x can be computed in time $O^*(2^{\text{symmetric}(\varphi, k) + 2 \cdot \text{asymmetric}(\varphi, k)})$.

Proof. To compute Q_x , we first split Q_y into B_1, \dots, B_q such that all elements in one B_i share the same “symmetric bits”. This can be done in time $|Q_y|$ using bucket-sort. Note that we have $q \leq 2^{\text{symmetric}(\varphi, k)}$ and $|B_i| \leq 2^{\text{asymmetric}(\varphi, k)}$. With the same technique we identify for every element v in Q_z its corresponding partition B_i . Finally, we compare v with the elements in B_i to identify those for which there is a transition in the automaton. This strategy yields a total running time of the form $|Q_z| \cdot \max_{i=1}^q |B_i| \leq 2^{\text{bit}(\varphi, k)} \cdot 2^{\text{asymmetric}(\varphi, k)} = 2^{\text{symmetric}(\varphi, k) + 2 \cdot \text{asymmetric}(\varphi, k)}$. \square

The table at the right shows all formulas and quantifiers that an input formula may use. For each quantifier and formula the table states the amount of bits that we will reserve in the state description of the tree automaton, as well as the fact if they are symmetric or not.

Given the table at the right, we can, simply by looking at the syntax of the input formula, deter-

Quantifier / Formula	Bits	Sym.
free var. X_1, \dots, X_q	$q \cdot k$	✓
$\exists^{\text{partition}} X_1, \dots, X_q$	$k \cdot \log_2 q$	✓
$\exists^{\text{connected}} X$	$k \cdot \log_2 k + 1$	✗
$\exists^{\text{forest}} X$	$k \cdot \log_2 k$	✗
$\forall x \forall y E(x, y) \rightarrow \chi_i$	0	✓
$\forall x \exists y E(x, y) \wedge \chi_i$	k	✗
$\exists x \forall y E(x, y) \rightarrow \chi_i$	$k + 1$	✗
$\exists x \exists y E(x, y) \wedge \chi_i$	1	✗
$\forall x \chi_i$	0	✓
$\exists x \chi_i$	1	✗

mine the number of symmetric and asymmetric bits that *Jatatosk* will reserve in the state space of the tree automaton – and thus we can directly determine the worst-case run time of *Jatatosk* for that formula. The following table illustrates this for all formulas used within this chapter.

Input Formula φ	symmetric(φ, k) asymmetric(φ, k)	Worst Case Run Time
$\varphi_{3\text{col}}$	$k \cdot \log_2(3)$ 0	$O^*(3^k)$
$\varphi_{\text{vc}}(S)$	k 0	$O^*(2^k)$
$\varphi_{\text{ds}}(S)$	k k	$O^*(8^k)$
$\varphi_{\text{triangle-minor}}$	0 $3k \cdot \log_2(k) + 3$	$O^*(k^{6k})$
$\varphi_{\text{fvs}}(S)$	k $k \cdot \log_2(k)$	$O^*(2^k k^{2k})$

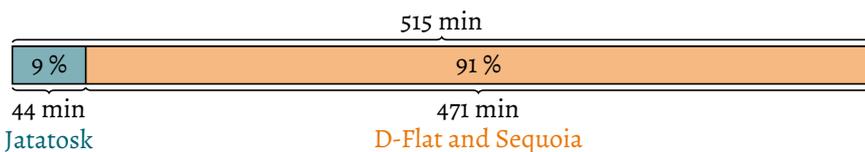
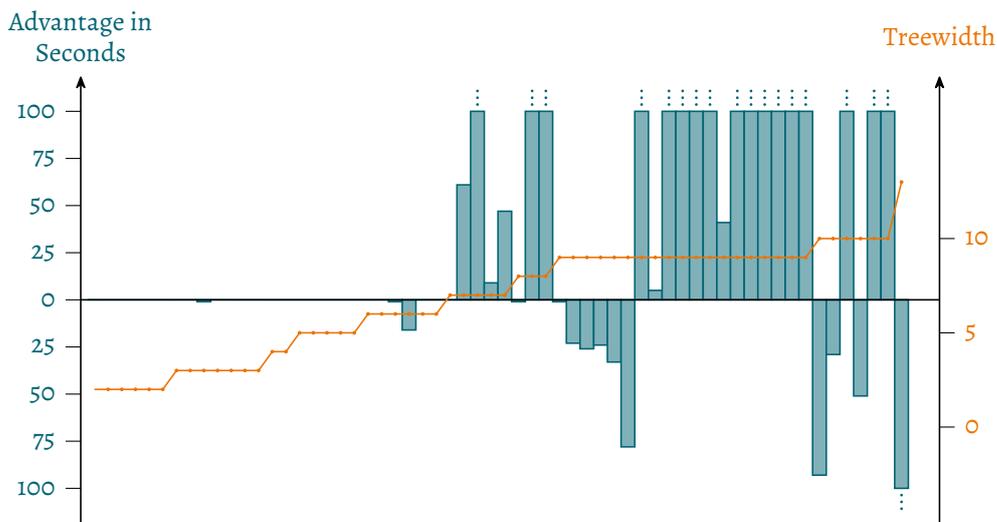
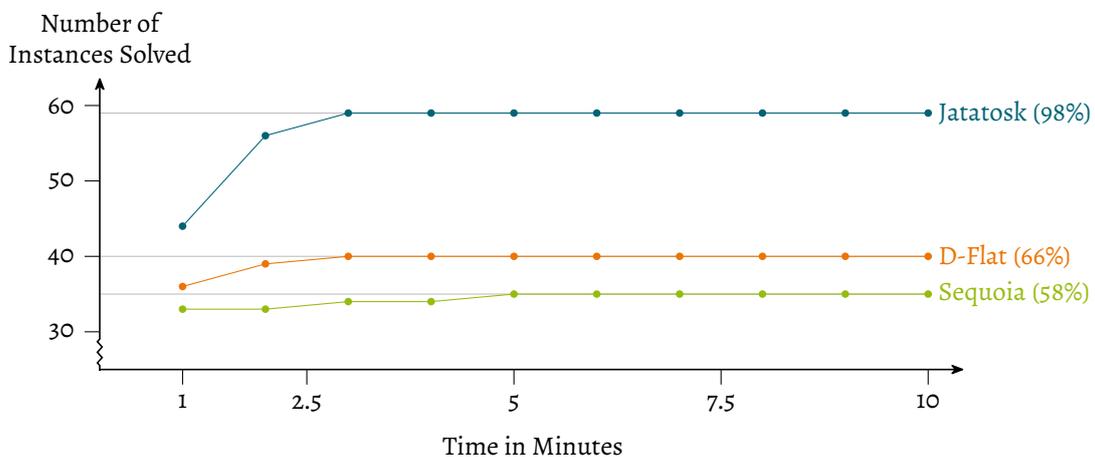
Experiments with Natural Problems. To study the performance of *Jatatosk*, we compare it to *Sequoia* and *D-Flat* for various natural problems. All experiments in this section were performed on Machine II with a timeout of 10 minutes per instance. All solvers were tested on the graphs of Testset III and IV, as well as the graphs of Testset II with treewidth at most 11. Overall, this results in a test set of 61 instances. *Jatatosk* (and underlying *Jdrasil*) were used with Java 1.8, while *Sequoia* and *D-Flat* were both compiled with gcc 7.2.

For every experiment I provide three color coded figures that visualize the results. The first graphic is always a classical *cactus plot* that visualizes how many instances can be solved by each of the tools in x minutes. Therefore, a faster growing function is better. On the right side of the diagram the name of the corresponding solver and its total amount of solved instances (in percent) is shown.

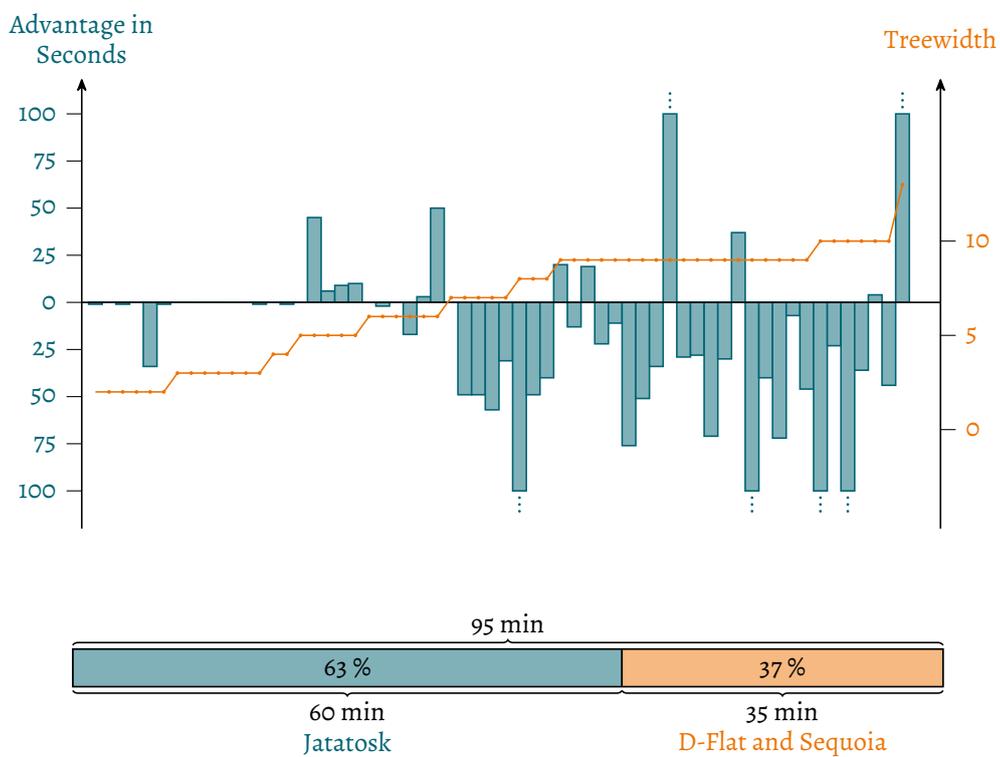
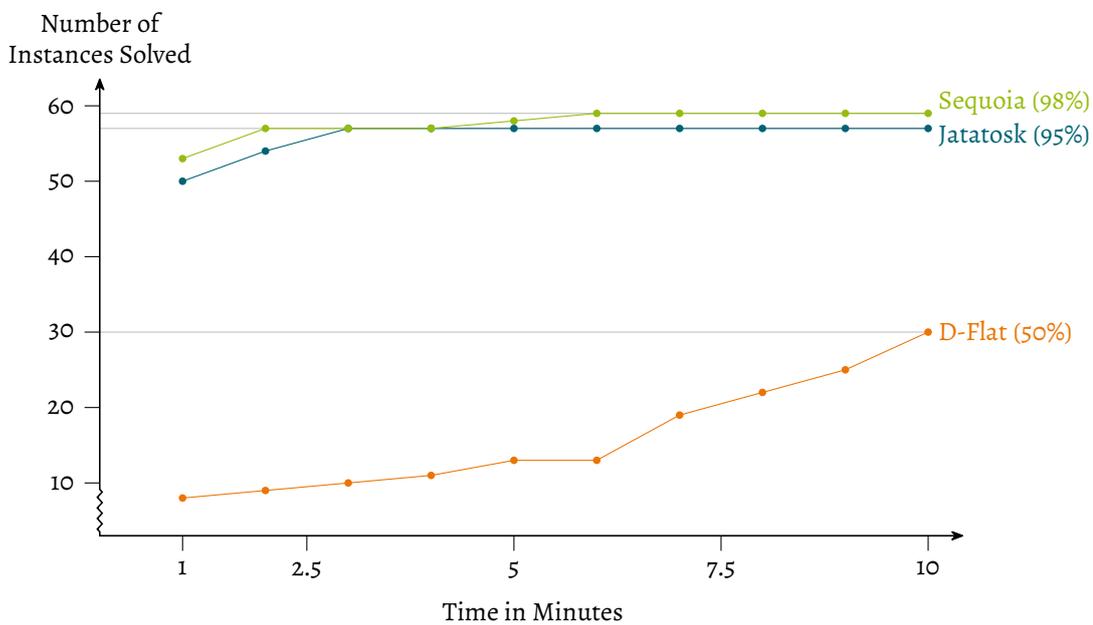
The second graphic is always an *advantage plot* of *Jatatosk* against *D-Flat* and *Sequoia*. For these I have taken all instances of the test set and ordered them by their *treewidth*. An x -coordinate corresponds to one test instance (with respect to the ordering) and for each instance there is an *advantage bar*: A positive bar means that *Jatatosk* is faster than the best of *D-Flat* and *Sequoia* on this instance; a negative bar means that the fastest of *D-Flat* and *Sequoia* is faster than *Jatatosk* by length of the bar. The bars are capped at 100 seconds.

The third and last diagram for each experiment is a *domination plot*, which contains the time *Jatatosk* needed to solve all instances, as well as the time *D-Flat* and *Sequoia* needed to solve them, where for each individual instance the fastest of the two was used. Then the diagram contains the sum of these two values (the complete time required for the experiment) and the percent of this time used for either *Jatatosk* or its *competitors*. Therefore, if both have 50% the solvers used exactly the same amount of time, while otherwise the solver with the *smaller percentage* is *faster*.

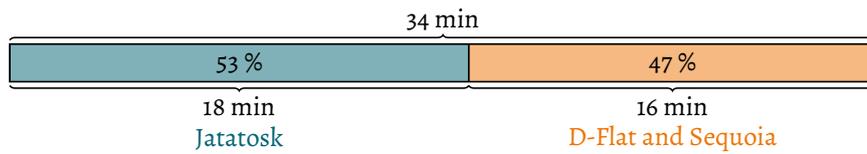
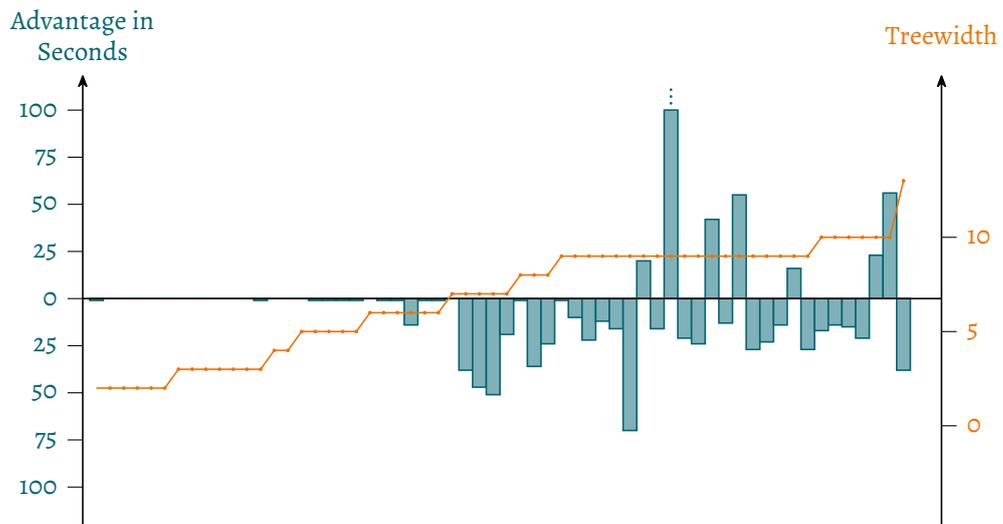
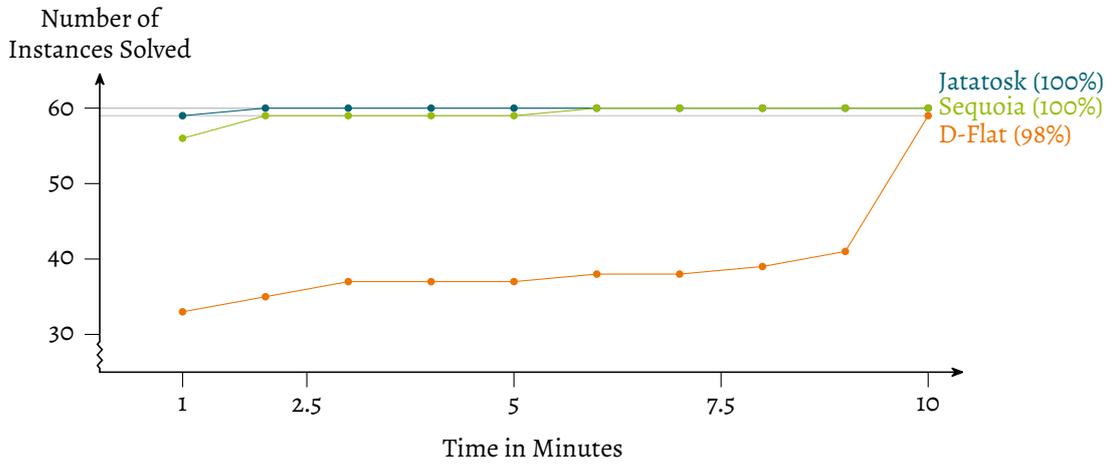
RESULTS FOR 3-COLORING



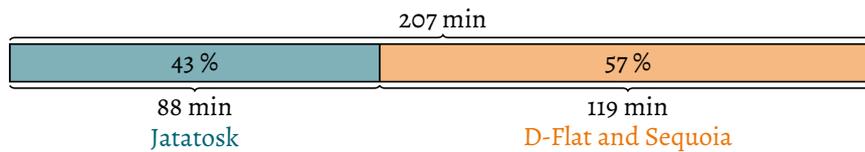
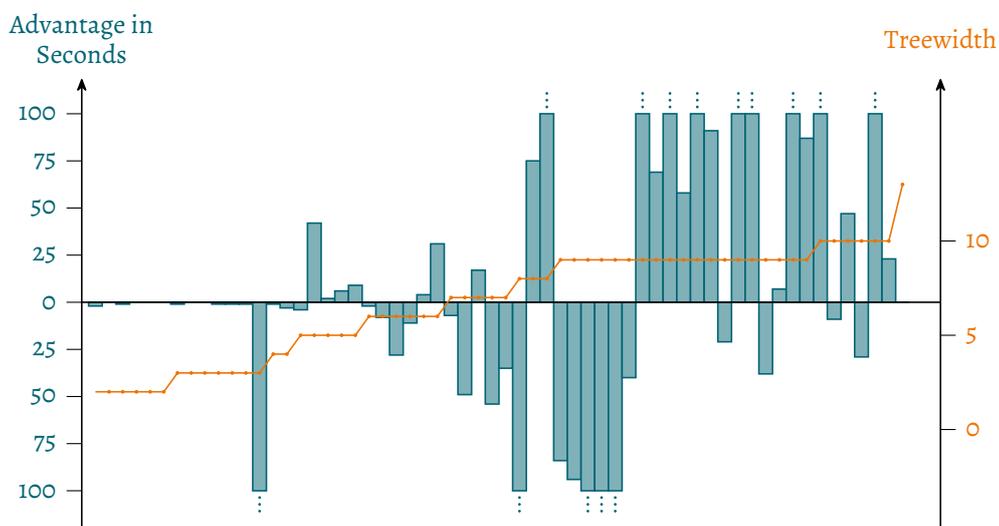
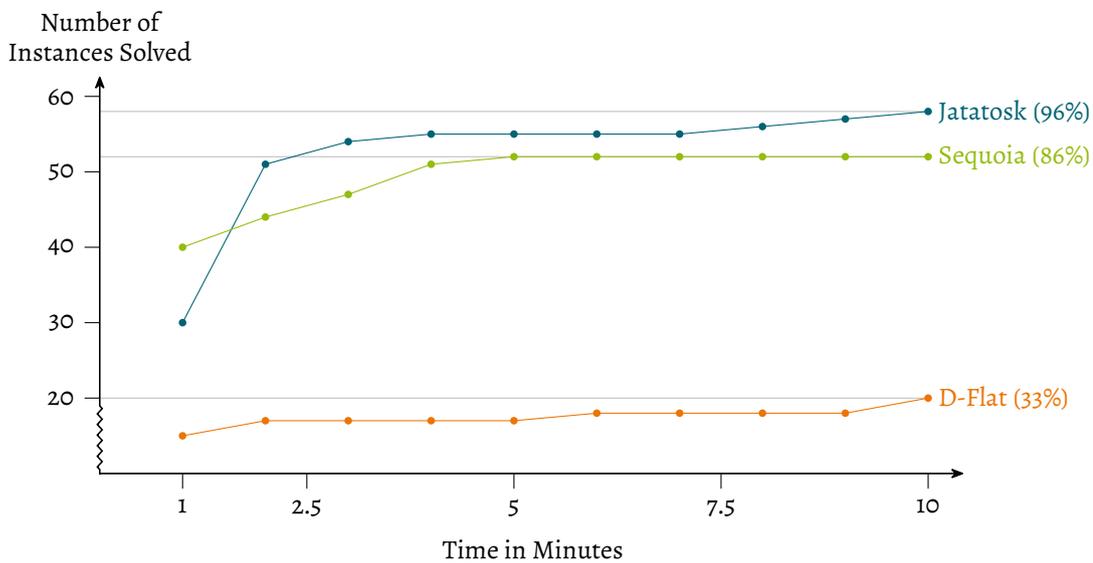
RESULTS FOR VERTEX COVER



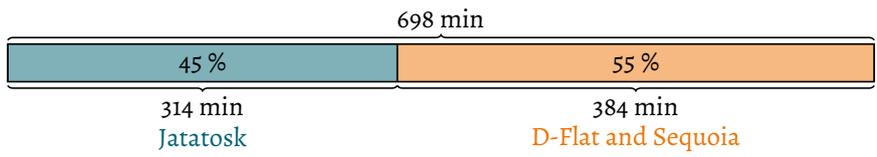
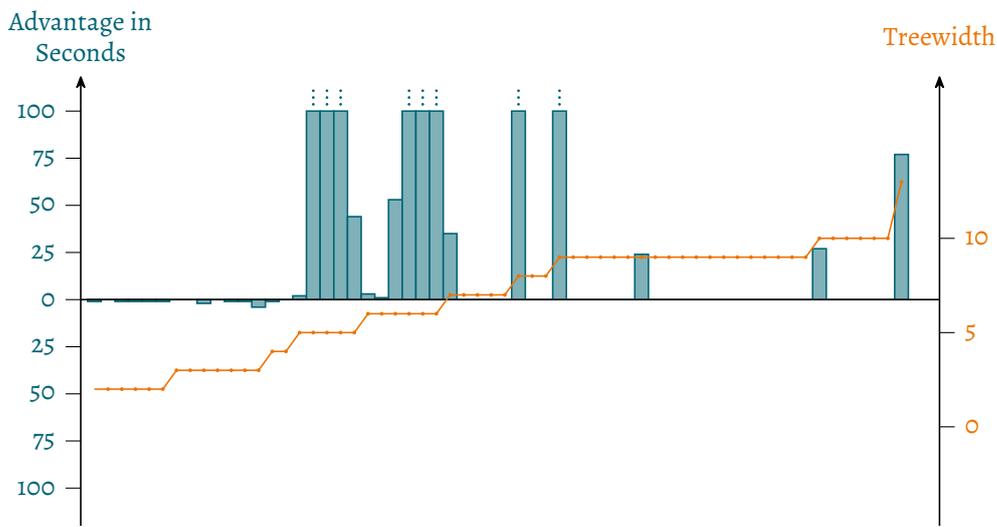
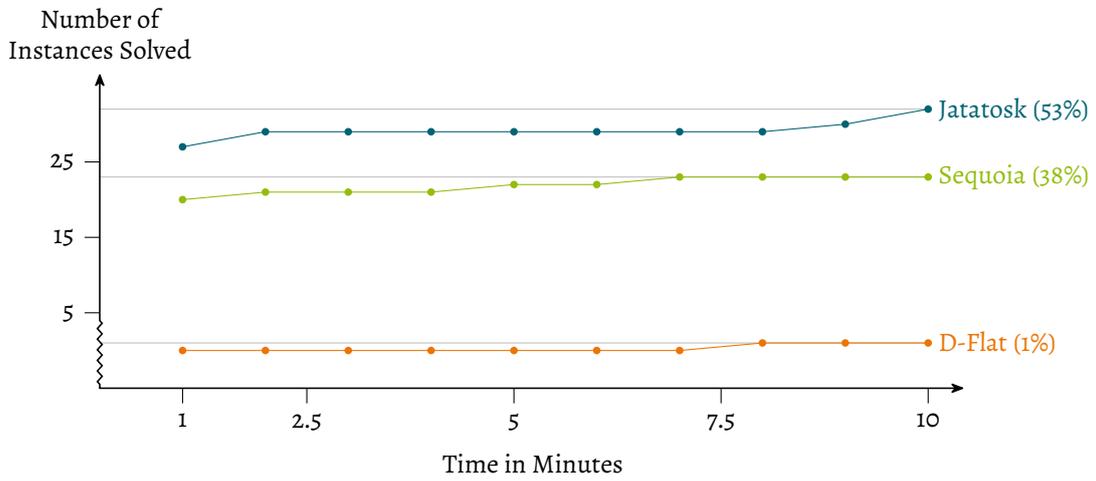
RESULTS FOR INDEPENDENT SET



RESULTS FOR DOMINATING SET



RESULTS FOR FEEDBACK-VERTEX SET



Evaluation of the Experiments. It can be seen that *Jatatosk* is competitive (though not always superior) against its competitors, as it is faster than the faster of the two on many instances. *Jatatosk* outperforms the others for the task of coloring a graph with three colors, but gets outperformed by *Sequoia* for finding a minimum vertex cover. The same holds for the problem of finding an independent set, although the difference is much smaller in this case. For the task of computing a dominating-set, we have a more complicated situation: *Jatatosk* outperforms the others on about half of the instances, and gets outperformed on the other half. Interestingly, the difference is quite high in both halves in both directions.

Solving feedback-vertex set seems to be the hardest task for all tested solvers. In case of *Jatatosk*, this is also reflected by the formula and the expected run time. *Jatatosk* manages to solve only about 50% of the test set for this problem. However, *Sequoia* falls behind a little more, and *D-Flat* does not seem to work for this problem at all. The superiority of *Jatatosk* in solving feedback-vertex set is also reflected by the difference plot, in which almost all non-zero bars are in favor of *Jatatosk*.

All the cactus plots reveal that *Jatatosk* is able to solve more instances in a smaller amount of time in almost all cases. The exception are vertex cover and independent set, where the cactus plot of *Sequoia* looks slightly better. However, the difference in this two cases is very small, while the advantage of *Jatatosk* on the other problems is comparatively large. I assume the similarity for vertex cover and independent set is owed to the circumstance that all solvers compile internally a similar algorithm. The slight advantage of *Sequoia* might be due to a performance difference of Java and C++. That *Jatatosk* outperforms the other solvers for coloring is not surprising, as the *MSO*-fragment used by *Jatatosk* is tailored around the corresponding graph coloring formula. That *Jatatosk* is better at solving feedback-vertex set is probably due to the fact that we have implemented and optimized the forest quantifier directly, while the other solvers have to extract this from the formula as well. I was surprised, however, that *Jatatosk* did well for the dominating set problem, as the promised run time by the fragment is much worse than the best known run time. However, it seems that the other solvers do not reach a better run time neither.

12 OUTLOOK AND FURTHER DIRECTIONS

In the second part of this thesis, we have moved from parameterized algorithm design to parameterized algorithm engineering – an intermediate discipline between theory and practice that explores implementations of theoretical concepts. While we studied parallel parameterized algorithms that guarantee certain worst-case performances in the first part of this thesis, in this part we developed parallel strategies that work well in practice.

There were two main problems that we tackled: The computation of optimal tree decompositions in Chapter 10, and the implementation of a model checker for monadic second-order logic on top of it in Chapter 11. In case of treewidth computations, we examined in detail the library *Jdrasil*. Here, I have decided to use a coarse parallelization strategy, that parallelizes the computation of a tree decomposition independently of the used (exact or heuristic) subroutines. This strategy fits well into the modular architecture of *Jdrasil* and provides at least a small speedup. We have first explored the modular architecture of *Jdrasil* and have then glanced at some of its subroutines. In particular, we have evaluated the SAT-based approach by Berg and Järvisalo and extended it by tricks known from the parameterized complexity community. Afterwards, we screened the concept of splitting and safe separators, which has turned out to be a key technology for computing tree decompositions in practice – and especially for turning these algorithms into parallel ones. This approach leads to interesting further research directions. Since the size of a kernel for the treewidth problem cannot be bounded by a function in the treewidth (unless $\text{NP} \subseteq \text{coNP/poly}$), splitting could provide an alternative to kernelization. Ultimately, a Turing-kernel for the problem is not excluded, but achieving any guarantees for the splitting process would be a welcome first step.

- ⊢ *Open Problem:* Improving the procedure for finding safe separators to provide any guarantee. Either in terms of a probability to find safe separators if they exist; or in terms of a bound for the size of the atoms. ⊣

An important tool to test whether a separator S is safe for treewidth was to check whether a clique on S is contained as labeled minor in the components associated with S . The heuristics that are currently used to perform this check are rather simple. Improving them could lead to faster algorithms for computing tree decompositions.

- ⊢ *Open Problem:* Finding alternative heuristics for the following problem: Given a graph $G = (V, E)$ and a set $S \subseteq V$, is a clique on S contained as labeled minor in G ? ⊣

- ⊢ *Open Problem:* Is there an exponential time or FPT-algorithm for the above problem that works fast in practice? ⊣

In the second chapter of this part, we studied a fragment of MSO that was crafted with the aim of being easy to model check in practice – both, sequentially and in parallel. It was especially selected to overcome the huge automata constructions that we encountered in Chapter 8. To see if the approach is viable, I introduced the model checker *Jatatosk*. While it is implemented sequentially in its current form, the architecture of *Jatatosk* (and of course of the whole fragment) has parallelization in mind. Therefore, it is a natural next step to parallelize *Jatatosk*.

- ⊢ *Open Problem:* Extending *Jatatosk* with parallel capabilities. In particular, multiple subtrees of the tree decomposition can be handled in parallel on multiple CPUs, while single steps of the dynamic program can be outsourced to a GPU. ⊣

Note, however, that although the architecture has parallelization in mind, implementing it in a way that results in an actual speedup will be non-trivial. This is because there are many details that have to be overcome and that will slow down the implementation. For instance, for such dynamic programs both, load balancing and GPU balancing are very difficult tasks.

Since *Jatatosk* was fully development from an algorithm engineering point of view, rather than a pure algorithm design point of view, it lacks some features that could, in theory, be easily added to it. Most importantly: It does not handle whole MSO, but only a fragment of it. Given the success of *Jatatosk*, it is a natural next step to explore implementations of model checkers for further fragments. For instance, it should be possible to push the event-driven evaluation of *Jatatosk* (which uses the binary relation E as guard) to guarded first-order logic (by working on the Gaifman graph).

- ⊢ *Open Problem:* Exploring other fragments of monadic second-order logic for their capability of being model checked quickly in practice. ⊣

Since the initial fragment was quite limited, we extended it with special quantifiers for some problems that can be solved efficiently on graphs of small treewidth. However, the list of problems that can be solved efficiently on graphs of small treewidth is virtually endless and, therefore, there are many further quantifier extensions that are possible. One could, for instance, existentially bind a long path, a long cycle, a vertex cover, or a dominating set.

- ⊢ *Open Problem:* What are useful quantifier extensions that increase the expressiveness of the fragment and that can be implemented fast in practice? ⊣

13 CONCLUSION

In this thesis we have explored the fascinating field of fixed-parameter tractability from a parallel point of view – both, in theory and practice. To that end, we started by establishing a collection of parallel subclasses of FPT. The resulting framework is build on top of earlier work due to Flum and Grohe [84], and Elberfeld, Stockhusen, and Tantau [76]. It is based on parameterized circuit classes that inherit some of their features from classical circuit complexity, but which also confronted us with new technical challenges – we discussed and resolved them in Chapter 3.

Once we had the definitions settled, we were ready to design parallel parameterized algorithms. The first objective, which we engaged in Chapter 4, was the compilation of a toolbox of basic parallel parameterized algorithms. A cornerstone for many algorithms within this box, but also for many other algorithms across this thesis, was the technique of color coding. In fact, one could say that what prefix sum or pointer jumping are for classical parallel algorithms, is color coding for parameterized constant time algorithms – almost all algorithms use it, and it seems unavoidable most of the time.

An elementary technique in the design of parallel algorithms is symmetry breaking in the form of computing independent sets. We encountered this problem multiple times with various parameters throughout this thesis. In particular, we proved the following three results – however, we did only focus on minimizing the *parallel time*, while it is often also important to implement symmetry breaking in a *work optimal way*, which is, thus, an interesting further research direction.

- p_{Δ} -MAXIMAL-INDEPENDENT-SET \in para-AC $^{o+\epsilon}$,
- $p_{k,\Delta}$ -INDEPENDENT-SET \in para-AC o
- p_k -PLANAR-INDEPENDENT-SET \in para-AC $^!$

We continued by adapting many techniques that parameterized complexity has in its quiver to work in parallel. It is not surprising that this works well for bounded search trees, as such trees can naturally be evaluated in parallel. We explored the technique with the help of various interesting problems, including multiple modulator problems as well as the feedback-vertex set problem. In contrast, I was surprised that kernelization is well suited for parallelization as well – after all, the technique is presented in a very sequential way in any textbook. However, we were even able to

prove that the relation “FPT equals kernelization” holds in the parallel setting: “parallel parameterized algorithms equal parallel kernelization.” The by far strongest result that we obtained in this area is a constant-time kernelization for hitting set parameterized by the solution size k and the maximum size d of any hyperedge. By doing so, we have refuted a conjecture by Chen, Flum, and Huang [53], which states that such a kernelization requires parallel time $\Omega(d)$. The following table illustrates the kernel sizes that we achieved for various problems in different circuit classes (the function f is some highly exponential function that results from Theorem 77 and Corollary 78):

Problem	Kernel size achievable in				
	AC^0	TC^0	NC	RNC	P
p-POINT-LINE-COVER	–	k^2	k^2	k^2	k^2
p_k -VERTEX-COVER	2^k	$k^2 + 2k$	$k^2 + 2k$	$2k$	$2k - c \log k$
p_k -MATCHING	2^k	$6k^2$	$6k^2$	1	1
p_{vc} -TREEWIDTH	$2^{ S }$	$ S ^3$	$ S ^3$	$ S ^3$	$ S ^3$
p_{vc} -PATHWIDTH	$2^{ S }$	$ S ^3$	$ S ^3$	$ S ^3$	$ S ^3$
p_{vc} -TREEDEPTH	$2^{ S }$	$ S ^3$	$ S ^3$	$ S ^3$	$ S ^3$
$p_{k,d}$ -HITTING-SET	$f(k, d)$	$f(k, d)$	$f(k, d)$	$f(k, d)$	$k^d \cdot k!$

The results of the table are all positive in the sense that they try to minimize the size of the corresponding circuits. However, circuit complexity is also famous for its power in providing lower bounds. Although we showed that a certain kernel for p_k -VERTEX-COVER cannot be parallelized unless we can compute large matchings in parallel, we did not discuss “real” circuit lower bounds in this thesis. A natural next step is, thus, to rule out that certain kernel sizes can be achieved by circuits of certain size or depth.

In order to develop parallel parameterized algorithms for a broad range of problems at once, I ended the first part of this thesis by presenting parallel versions of famous algorithmic meta-theorems. To that end, we needed a parallel way of decomposing a graph with respect to various graph parameters. The following table summarizes the corresponding results from Chapter 7:

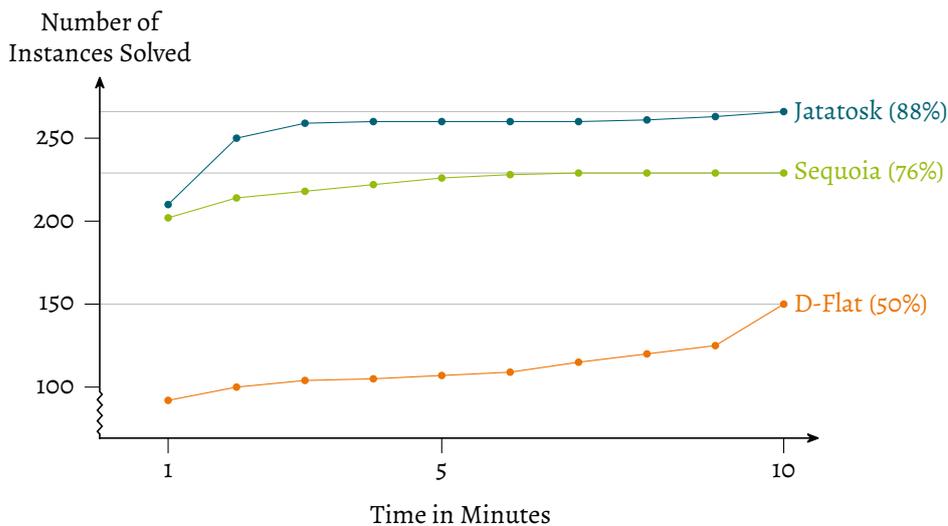
Decomposition	Complexity	Note
Crown Decomposition	para- AC^0	optimal
Treedepth Decomposition	para- AC^{0t}	approximation
Tree Decomposition	para- AC^{2t}	optimal

Equipped with these algorithms, we were able to establish parallel algorithmic meta-theorems in Chapter 8. I presented such theorems for first- and monadic second-order logic with respect to various graph parameters. The results are summarized in the table on the next page.

	Logic	Parameter	Complexity
	First-Order	$ \varphi + \Delta$	para-AC ⁰¹
Monadic Second-Order		$ \varphi + vc$	para-AC ⁰
Monadic Second-Order		$ \varphi + td$	para-AC ⁰¹
Monadic Second-Order		$ \varphi + tw$	para-AC ²¹

The objective of the second part of this thesis was the development of a practical tool for the result in the last line. The foundation for such a tool is a library that is able to compute tree decompositions quickly in practice. Chapter 10 showcases the Java library *Jdrasil*, which was developed for this purpose. I presented some of its features, in particular an improved SAT-encoding as well as a game theoretic characterization of positive instance driven dynamic programming. Furthermore, I highlighted ways to parallelize the computation of tree decompositions through safe separators.

With all the results concerning the computation of tree decompositions and the solving of model checking problems that were discussed in the course of this thesis, it was possible to present a model checker that performs well in practice: *Jatatosk*. The main achievement of Chapter 11 is the elaboration of a fragment of monadic second-order logic that is both, general enough to express many natural problems, but restricted enough to be model checked efficiently in practice. *Jatatosk* is tailored to this fragment, which makes it more efficient than many of its competitors. Additionally, its architecture allows to deduce the precise run time of *Jatatosk* directly from the syntax of an input formula. The following figure illustrates the performance of *Jatatosk* against its competitors over all formulas: It is the sum of the cactus plots discussed in Chapter 11. Although the plot looks positive for *Jatatosk*, I should point out once more that the other two tools can solve a more general problem – in particular, both can be used to model check monadic second-order logic, while *Jatatosk* can only handle a fragment of it.



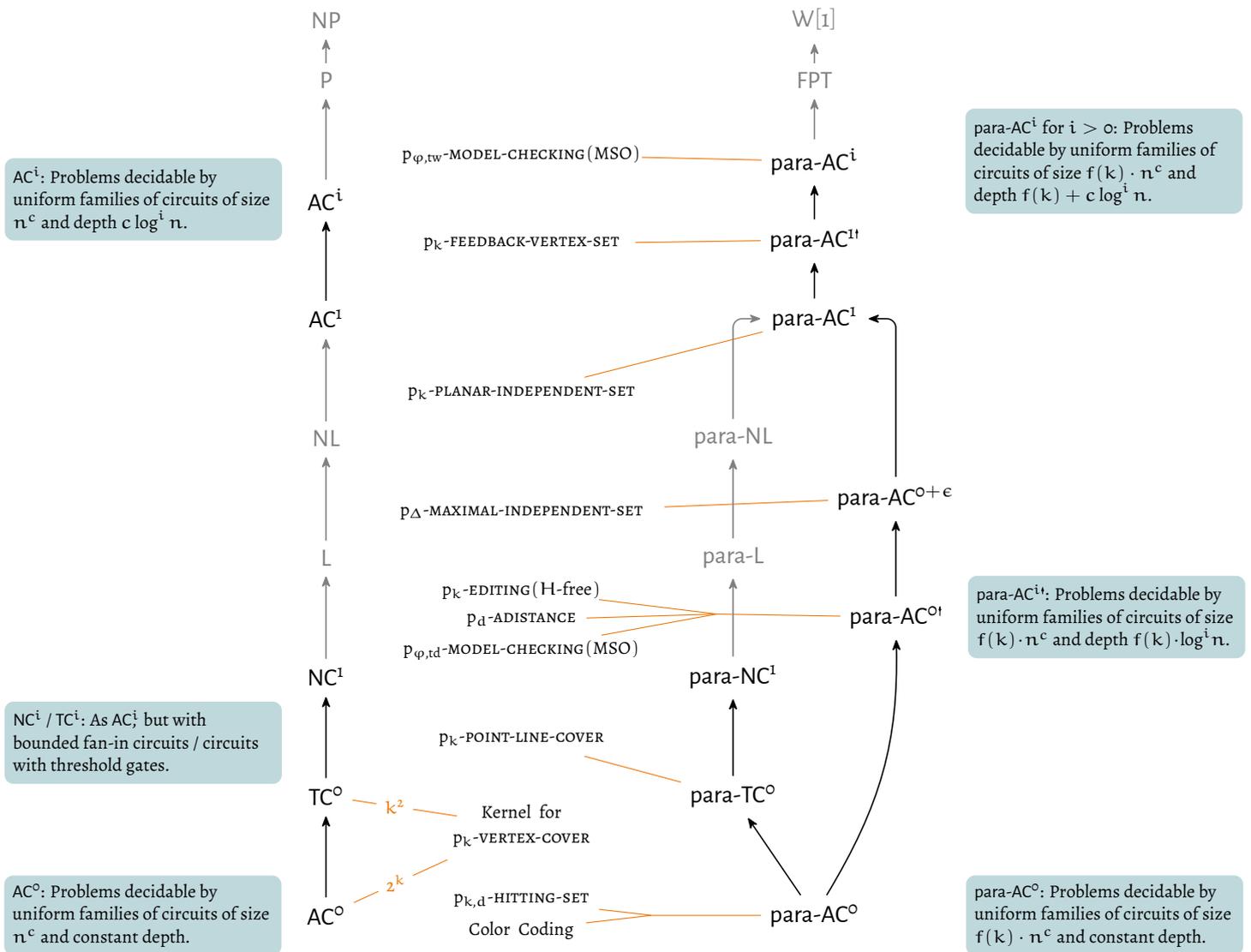
At the end of each part, I already sketched many possible paths for further research. I would like to close this thesis by repeating the three paths that I personally think are the most interesting ones. The first path is the application of parallel color coding in practice. We have seen that many parallel parameterized algorithms are based on color coding, but the algorithms presented within this thesis use the technique quite heavily – probably too heavily for an efficient implementation. It is therefore a desirable path to balance the “amount of used color coding” in order to obtain parallel algorithms that are fast in practice.

The second path I would like to highlight is the exploration of polynomial kernels that are computable within FAC° . Within this thesis, all kernels that we were able to compute in FAC° have exponential size. However, we do not have any lower bound that forbids kernels of polynomial size. It would therefore be interesting to close this gap, either with positive results (natural problems that have kernels of polynomial size that can be computed in FAC°), or with negative results (a technique to prove that such kernels cannot exist for certain problems).

Finally, the third path that should – in my opinion – obtain further attention in the future is the computation of optimal tree decompositions in practice. During the past years – in the light of the Parameterized Algorithms and Computational Experiments Challenge – the available tools already became much better, and the understanding of the community of how to solve this problem did increase a lot as well. However, techniques like positive instance driven dynamic programming are still not fully understood, and we do not have the tools yet to grasp when and why this technique works well. Here, the practice is currently one step ahead of the theory, and it is of course important to catch up. Beyond that, the most challenging task for the future will be to understand whether and how positive instance driven dynamic programming can be parallelized.

COMPENDIUM OF CLASSES AND PROBLEMS

The figure illustrates the complexity classes that we studied. An arrow $A \rightarrow B$ means that A is a subclass of B . Gray classes are known in the literature, but were not further discussed within this thesis. In the center of the figure a collection of “typical” problems that we have encountered for some of the classes is presented. A complete list of all problems and results can be found on the next page.



ADISTANCE

Instance: A directed graph $G = (V, E)$, a partition $V = V_{\exists} \cup V_{\forall}$, two vertices $s, t \in V$, a distance d .

Question: Is the alternating distance from s to t in G at most d ?

Result:

- in para-AC⁰ for parameter d 41

Referenced on pages: 41–43, 177

CLIQUE

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: Is there a set $X \subseteq V$ with $|X| \geq k$ such that $G[X]$ is a complete graph?

Referenced on page: 8

CLUSTER-EDITING

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: Can G be transformed into a graph in which every component is a clique by just k edge-modifications?

Result:

- in para-AC⁰ for parameter k 51

Referenced on page: 51

DISTANCE

Instance: A graph $G = (V, E)$, two vertices $s, t \in V$, and a number $d \in \mathbb{N}$.

Question: Is there a path of length at most d from s to t in G ?

Result:

- in para-AC⁰ for parameter d 41

Referenced on pages: 41, 43

DOMINATING-SET

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: Is there a set $S \subseteq V$ with $|S| \leq k$ and $\{v \in V \mid |N[v] \cap S| \geq 1\} = V$?

Result:

- in para-AC⁰ for parameter $k + \Delta$ 90

Referenced on pages: 90, 102

DUAL-COLORING

Instance: A graph $G = (V, E)$ and a number $q \in \mathbb{N}$.

Question: Is there a proper coloring of G with at most $|V| - q$ colors?

Result:

- in para-AC⁰ for parameter q 93

Referenced on page: 93

EDITING(\mathcal{F})

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: Are there sets $R \subseteq E$ and $A \subseteq \bar{E}$ with $|R \cup A| \leq k$ such that we have

$G' = (V, (E \setminus R) \cup A) \in \mathcal{F}$?

Result:

- in para-AC⁰ for parameter k if \mathcal{F} is the class of H -free graphs 52

Referenced on pages: 51–52

EMB-MODULATOR(\mathcal{H})

Instance: Two graphs $H = (V(H), E^H) \in \mathcal{H}$ and $G = (V(G), E^G)$, and a number $k \in \mathbb{N}$.

Question: Is there a set $X \subseteq V(G)$ with $|X| \leq k$ such that $H \rightsquigarrow G[V \setminus X]$?

Result:

- in para-AC⁰ parameterized by H for \mathcal{H} of constant treewidth 56

Referenced on pages: 55–56

EMBEDDING(\mathcal{H})

Instance: Two graphs $H = (V(H), E^H) \in \mathcal{H}$ and $G = (V(G), E^G)$.

Question: $H \rightsquigarrow G$?

Results:

- in para-AC⁰ parameterized by H for \mathcal{H} of constant treewidth 56
- in para-AC⁰ parameterized by H for \mathcal{H} of constant treedepth 56

Referenced on pages: 55–56

FEEDBACK-VERTEX-SET

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: Is there a set $X \subseteq V$ with $|X| \leq k$ such that $G[V \setminus X]$ is a forest?

Result:

- p_k -FEEDBACK-VERTEX-SET \in para-AC¹ 58

Referenced on pages: 3, 8, 57–59, 91, 177

HITTING-SET

Instance: A hypergraph $H = (V, E)$ with $\max_{e \in E} |e| = d$ and a number $k \in \mathbb{N}$.

Question: Is there a set $X \subseteq V$ with $|X| \leq k$ and $e \cap X \neq \emptyset$ for all $e \in E$?

Results:

- in para-AC⁰ for parameter $k + d$ 90
- kernel in FAC⁰ for parameter $k + d$ 90

Referenced on pages: 4, 8, 56, 64, 81–83, 90, 112, 174, 177

HOM-MODULATOR(\mathcal{H})

Instance: Two graphs $H = (V(H), E^H) \in \mathcal{H}$ and $G = (V(G), E^G)$, and a number $k \in \mathbb{N}$.

Question: Is there a set $X \subseteq V(G)$ with $|X| \leq k$ such that $H \rightsquigarrow G[V \setminus X]$?

Result:

- in para-AC⁰ parameterized by H for \mathcal{H} of constant treewidth 55

Referenced on pages: 52, 55

HOMOMORPHISM(\mathcal{H})

Instance: Two graphs $H = (V(H), E^H) \in \mathcal{H}$ and $G = (V(G), E^G)$.

Question: $H \rightarrow G$?

Results:

- in para-AC⁰ parameterized by H for \mathcal{H} of constant treewidth 54
- in para-AC⁰ parameterized by H for \mathcal{H} of constant treedepth 55

Referenced on pages: 53–55

INDEPENDENT-SET

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: Is there a set $X \subseteq V$ with $|X| \geq k$ such that $G[X]$ is edgeless?

Results:

- maximal solution can be found in para-FAC^{0+ ϵ} on graphs of bounded degree 38
- in para-AC⁰ for parameter $k + \Delta$ 47

Referenced on pages: 3, 36, 38, 47, 173

MATCHING

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: Is there a set $M \subseteq E$ with $|M| = k$ such that no vertex is incident to more than one element of M ?

Results:

- in para-AC⁰ for parameter k 76
- k^2 -kernel in FTC⁰ 76

Referenced on pages: 76–77, 174

MODEL-CHECKING(\mathcal{L})

Instance: A relational structure S and an \mathcal{L} -formula φ .

Question: $S \models \varphi$?

Results:

- in para-AC⁰ for $\mathcal{L} = \text{FO}$ and parameter $|\varphi| + \Delta(S)$ 102
- in para-AC⁰ for $\mathcal{L} = \text{MSO}$ and parameter $|\varphi| + \text{vc}(S)$ 104
- in para-AC⁰ for $\mathcal{L} = \text{MSO}$ and parameter $|\varphi| + \text{td}(S)$ 104
- in para-AC² for $\mathcal{L} = \text{MSO}$ and parameter $|\varphi| + \text{tw}(S)$ 104

Referenced on pages: 101–102, 104, 177

MODULATOR(\mathcal{F})

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: Is there a set $X \subseteq V$ with $|X| \leq k$ such that $G[V \setminus X] \in \mathcal{F}$?

Results:

- in para-AC⁰ for parameter k if \mathcal{F} is the class of H -free graphs 52
- in para-AC⁰ for parameter k if \mathcal{F} is the class of H -free graphs 90

Referenced on pages: 51–52, 90

PARITY

Instance: A binary string $w \in \{0, 1\}^*$.

Question: $\sum_{i=1}^{|w|} w[i] \pmod 2 = 0$?

Referenced on page: 30

PARTIAL-VERTEX-COVER

Instance: A graph $G = (V, E)$ and two numbers $k, t \in \mathbb{N}$.

Question: Is there a set $S \subseteq V$ with $|S| \leq k$ and $|\{\{u, v\} \mid u \in S \vee v \in S\}| \geq t$?

Result:

- in para-AC⁰ for parameter $k + t$ 104

Referenced on pages: 101, 104

PATH-VERTEX-COVER

Instance: A graph $G = (V, E)$ and two numbers $k, c \in \mathbb{N}$.

Question: Is there a set $S \subseteq V$ with $|S| \leq k$ and $S \cap P \neq \emptyset$ for each length- c path P in G ?

Results:

- in para-AC⁰ for parameter k if c is constant 56
- in para-AC⁰ for parameter k and c 56

Referenced on page: 56

PATHWIDTH

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: $\text{pw}(G) \leq k$?

Results:

- $|S|^3$ -kernel in FTC⁰ 80
- in para-AC⁰ parameterized by vc 81

Referenced on pages: 77–81, 174

PLANAR-INDEPENDENT-SET

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: Is G planar and is there a set $X \subseteq V$ with $|X| \geq k$ such that $G[X]$ is edgeless?

Results:

- in para-AC¹ for parameter k 50
- in para-AC⁰ if planarity is promised for parameter k 50
- $4k$ -kernel in FAC¹ 65

Referenced on pages: 50–51, 65

POINT-LINE-COVER

Instance: A set of points $p_1, \dots, p_n \in \mathbb{Z}^d$ for a fixed $d \geq 2$ and a number $k \in \mathbb{N}$. Both, the points and k , are encoded as binary numbers.

Question: Can we cover all points by at most k straight lines?

Results:

- k^2 -kernel in FTC⁰ 65
- every slice is TC⁰-complete 66

Referenced on pages: 65–67, 72, 174, 177

RAINBOW-MATCHING

Instance: An edge-colored graph $G = (V, E, \chi)$ with $\chi: E \rightarrow \{1, \dots, k\}$.

Question: Is there a matching $M \subseteq E$ with $|M| = k$ that contains an edge of every color, that is, all edges in M have distinct colors?

Result:

- in para-AC⁰ for parameter k 46

Referenced on pages: 44, 46

SAT

Instance: A propositional formula φ .

Question: Is there a satisfying assignment β of φ , that is, one with $\beta \models \varphi$?

Referenced on pages: 5, 77, 117, 121, 123–126, 143–145, 147–148, 171, 175

THRESHOLD

Instance: A bitstring $b \in \{0, 1\}^n$ and a number $t \in \mathbb{N}$.

Question: Are there at least t many 1's in b ?

Result:

- in para-AC⁰ for parameter t 47

Referenced on page: 47

TREEDDEPTH

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: $td(G) \leq k$?

Results:

- $|S|^\beta$ -kernel in FTC⁰ 80
- in para-AC⁰ parameterized by vc 81
- in para-AC⁰¹ for parameter k 94

Referenced on pages: 77–81, 94, 174

TREEWIDTH

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: $tw(G) \leq k$?

Results:

- $|S|^\beta$ -kernel in FTC⁰ 79
- in para-AC⁰ parameterized by vc 79
- in para-AC²¹ for parameter k 95

Referenced on pages: 77–79, 95, 174

VERTEX-COVER

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: Is there a set $X \subseteq V$ with $|X| \leq k$ such that $G[V \setminus X]$ contains no edge?

Result:

- in para-AC⁰ for parameter k 72

Referenced on pages: 4, 8, 21–22, 30, 56, 63, 72–73, 92, 101, 111, 174, 177

WEIGHTED-CIRCUIT-SATISFIABILITY

Instance: An AC-circuit C and a number $k \in \mathbb{N}$,

Question: Is there a string $w \in \{0, 1\}^*$ with $\sum_{i=1}^{|w|} w[i] = k$ and $C(w) = 1$?

Referenced on pages: 28–29

EXPERIMENT SETUP

Machine I A MacBook Pro (Retina, 2012) with a 2,7 GHz Intel Core i7 and 16 GB 1600 MHz DDR3. The machine runs macOS Mojave in version 10.14.2.

Machine II A desktop computer equipped with 8 GB RAM and an Intel Core processor that contains four cores of 3.2 GHz each. The machine runs Ubuntu in version 17.10.

Testset I Contains all the graphs that were used for the treewidth track in the PACE 2016 and which were labeled as “solvable in 100s” [63]. This test set shares most of the graphs with the standard test set for graph coloring from the DIMACS graph coloring challenge.

Testset II Contains all graphs of the second iteration of the PACE challenge [64]. It contains both, the public and the hidden graphs (which were released after the challenge). The graphs were handcrafted by the authors of the challenge to generate instances that are difficult, but manageable for the current technology. The approach was to start with an instance from the Probabilistic Inference Challenge (which is reasonable, since many probabilistic inference algorithms first compute a tree-decomposition of the input), then a random center vertex v was chosen, and the graph was reduced to the vertices of distance at most r to v (that is, to the ball of radius r around v).

Testset III A collection of publicly available transit graphs that were generated from GTFS-transit feeds [80]. This test set was also used for experiments in [83].

Testset IV A collection of real-world instances collected in [2].

BIBLIOGRAPHY

- [1] Karl R. Abrahamson, Rodney G. Downey, and Michael R. Fellows. Fixed-Parameter Tractability and Completeness IV: On Completeness for W[P] and PSPACE Analogues. *Annals of Pure and Applied Logic*, 73(3):235–276, 1995. doi : 10.1016/0168-0072(94)00034-Z.
- [2] Michael Abseher, Nysret Musliu, and Stefan Woltran. Improving the Efficiency of Dynamic Programming on Tree Decompositions via Machine Learning. *Journal of Artificial Intelligence Research*, 58:829–858, 2017. doi : 10.1613/jair.5312.
- [3] Faisal N. Abu-Khzam, Michael A. Langston, Pushkar Shanbhag, and Christopher T. Symons. Scalable Parallel Algorithms for FPT Problems. *Algorithmica*, 45(3):269–284, 2006. doi : 10.1007/s00453-006-1214-1.
- [4] Eric Allender and Meena Mahajan. The Complexity of Planarity Testing. In *Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science, STACS 2000, February 2000, Lille, France*, Lecture Notes in Computer Science, pages 87–98. Springer, 2000. doi : 10.1007/3-540-46541-3_7.
- [5] Noga Alon, László Babai, and Alon Itai. A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *Journal of Algorithms*, 7(4):567–583, 1986. doi : 10.1016/0196-6774(86)90019-2.
- [6] Noga Alon, Raphael Yuster, and Uri Zwick. Color-Coding. *Journal of the ACM*, 42(4):844–856, 1995. doi : 10.1145/210332.210337.
- [7] Kazuyuki Amano. k -Subgraph Isomorphism on AC^0 Circuits. *Computational Complexity*, 19(2):1016–3328, 2010. doi : 10.1007/s00037-010-0288-y.
- [8] Nima Anari and Vijay V. Vazirani. A Pseudo-Deterministic RNC Algorithm for General Graph Perfect Matching. *CoRR*, abs/1901.10387, 2019. URL: <http://arxiv.org/abs/1901.10387>.
- [9] Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009, July 11–17, 2009, Pasadena, California, USA*, pages 399–404, 2009.

- [10] Olivier Bailleux and Yacine Boufkhad. Efficient CNF Encoding of Boolean Cardinality Constraints. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming, CP 2003, September 29 – October 3, Kinsale, Ireland*, Lecture Notes in Computer Science, pages 108–122. Springer, 2003. doi:10.1007/978-3-540-45193-8_8.
- [11] Max Bannach. Jdrasil for graph coloring.
<https://github.com/maxbannach/Jdrasil-for-GraphColoring>.
 Accessed: 23.01.2019; Commit: a5e52a8.
- [12] Max Bannach and Sebastian Berndt. Jatatosk.
<http://www.github.com/maxbannach/jatatosk>.
 Accessed: 08.04.2019; Commit: 45e306c.
- [13] Max Bannach and Sebastian Berndt. Practical Access to Dynamic Programming on Tree Decompositions. In *Proceedings of the 26th Annual European Symposium on Algorithms, ESA 2018, August 20–22, 2018, Helsinki, Finland*, LIPIcs, pages 6:1–6:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ESA.2018.6.
- [14] Max Bannach and Sebastian Berndt. Positive-Instance Driven Dynamic Programming for Graph Searching. In *Proceedings of the 16th Algorithms and Data Structures Symposium, WADS 2019, August 5–7, 2019, Edmonton, Canada*, Lecture Notes in Computer Science. Springer, 2019.
- [15] Max Bannach, Sebastian Berndt, and Thorsten Ehlers. Jdrasil.
<http://www.github.com/maxbannach/jdrasil>.
 Accessed: 05.06.2019; Commit: dfa1eee.
- [16] Max Bannach, Sebastian Berndt, and Thorsten Ehlers. Jdrasil: A Modular Library for Computing Tree Decompositions. In *Proceedings of the 16th International Symposium on Experimental Algorithms, SEA 2017, June 21–23, 2017, London, UK*, LIPIcs, pages 28:1–28:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.SEA.2017.28.
- [17] Max Bannach, Sebastian Berndt, Thorsten Ehlers, and Dirk Nowotka. SAT-Encodings of Tree Decompositions. In *SAT Competition 2018: Solver and Benchmark Descriptions*, 2018.
- [18] Max Bannach, Malte Skambath, and Till Tantau. Towards Work-Efficient Parallel Parameterized Algorithms. In *Proceedings of the 13th International Conference on Algorithms and Computation, WALCOM 2019, February 27 – March 2, 2019, Guwahati, India*, Lecture Notes in Computer Science, pages 341–353. Springer, 2019. doi:10.1007/978-3-030-10564-8_27.

- [19] Max Bannach, Christoph Stockhusen, and Till Tantau. Fast Parallel Fixed-parameter Algorithms via Color Coding. In *Proceedings of the 10th International Symposium on Parameterized and Exact Computation, IPEC 2015, September 16–18, 2015, Patras, Greece*, volume 43 of *LIPICs*, pages 224–235. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi : 10.4230/LIPICs.IPEC.2015.224.
- [20] Max Bannach and Till Tantau. Parallel Multivariate Meta-Theorems. In *Proceedings of the 11th International Symposium on Parameterized and Exact Computation, IPEC 2016, August 24–26, 2016, Aarhus, Denmark*, volume 63 of *LIPICs*, pages 4:1–4:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi : 10.4230/LIPICs.IPEC.2016.4.
- [21] Max Bannach and Till Tantau. Computing Hitting Set Kernels By AC^0 -Circuits. In *Proceedings of the 35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, February 28 to March 3, 2018, Caen, France*, volume 96 of *LIPICs*, pages 9:1–9:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi : 10.4230/LIPICs.STACS.2018.9.
- [22] Max Bannach and Till Tantau. Computing Kernels in Parallel: Lower and Upper Bounds. In *Proceedings of the 13th International Symposium on Parameterized and Exact Computation, IPEC 2018, August 22–24, 2018, Helsinki, Finland*, *LIPICs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.
- [23] Max Bannach and Till Tantau. On the Descriptive Complexity of Color Coding. In *Proceedings of the 36th International Symposium on Theoretical Aspects of Computer Science, STACS 2019, March 13–16, 2019, Berlin, Germany*, *LIPICs*, pages 11:1–11:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi : 10.4230/LIPICs.STACS.2019.11.
- [24] David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1 . In *Proceedings of the 3th Annual Structure in Complexity Theory Conference, CoCo 1988, June 14-17, 1988, Washington, D. C., USA*, pages 47–59. IEEE, 1988. doi : 10.1109/SCT.1988.5262.
- [25] Paul Beame, Russell Impagliazzo, and Toniann Pitassi. Improved Depth Lower Bounds for Small Distance Connectivity. *Computational Complexity*, 7(4):325–345, 1998. doi : 10.1007/s000370050014.
- [26] Jeremias Berg and Matti Järvisalo. SAT-Based Approaches to Treewidth Computation: An Evaluation. In *Proceedings of the 26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, November 10–12, 2014, Limassol, Cyprus*, pages 328–335. IEEE Computer Society, 2014. doi : 10.1109/ICTAI.2014.57.

- [27] Anne Berry, Romain Pogorelcnik, and Geneviève Simonet. An Introduction to Clique Minimal Separator Decomposition. *Algorithms*, 3(2):197–215, 2010. doi:10.3390/a3020197.
- [28] Daniel Bienstock and Paul D. Seymour. Monotonicity in Graph Searching. *Journal of Algorithms*, 12(2):239–245, 1991.
- [29] Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In *Proceedings of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.
- [30] Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. D-flat: Progress report, 2014. URL: <https://www.dbai.tuwien.ac.at/research/report/dbai-tr-2014-86.pdf>.
- [31] Bernhard Bliem, Reinhard Pichler, and Stefan Woltran. Implementing Courcelle’s Theorem in a Declarative Framework for Dynamic Programming. *Journal of Logic and Computation*, 27(4):1067–1094, 2017. doi:10.1093/logcom/exv089.
- [32] Hans L. Bodlaender. NC-Algorithms for Graphs With Small Treewidth. In *Proceedings of the 15th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 1989, June 14-16, 1989, Castle Rolduc, The Netherlands*, pages 1–10. Springer, 1989. doi:10.1007/3-540-50728-0_32.
- [33] Hans L. Bodlaender. A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996. doi:10.1137/S0097539793251219.
- [34] Hans L. Bodlaender, Rod Downey, Fedor V. Fomin, and Dániel Marx, editors. *The Multivariate Algorithmic Revolution and Beyond - Essays Dedicated to Michael R. Fellows on the Occasion of His 60th Birthday*, volume 7370 of *Lecture Notes in Computer Science*. Springer, 2012. doi:10.1007/978-3-642-30891-8.
- [35] Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, and Danny Hermelin. On Problems Without Polynomial Kernels. *Journal of Computer and System Sciences*, 75(8):423–434, 2009. doi:10.1016/j.jcss.2009.04.001.
- [36] Hans L. Bodlaender, Fedor V. Fomin, Arie M.C.A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. On Exact Algorithms for Treewidth. *ACM Transactions on Algorithms*, 9(1):12:1–12:23, 2012. doi:10.1145/2390176.2390188.
- [37] Hans L. Bodlaender and Torben Hagerup. Parallel Algorithms with Optimal Speedup for Bounded Treewidth. *SIAM Journal on Computing*, 27(6):1725–1746, 1998. doi:10.1137/S0097539795289859.

- [38] Hans L. Bodlaender, Bart M. P. Jansen, and Stefan Kratsch. Kernel Bounds for Structural Parameterizations of Pathwidth. In *Proceedings of the 13th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2012, July 4–6, 2012, Helsinki, Finland*, Lecture Notes in Computer Science, pages 352–363. Springer, 2012. doi:10.1007/978-3-642-31155-0_31.
- [39] Hans L. Bodlaender, Bart M. P. Jansen, and Stefan Kratsch. Preprocessing for Treewidth: A Combinatorial Analysis through Kernelization. *SIAM Journal on Discrete Mathematics*, 27(4):2108–2142, 2013. doi:10.1137/120903518.
- [40] Hans L. Bodlaender and Ton Kloks. Efficient and Constructive Algorithms for the Pathwidth and Treewidth of Graphs. *Journal of Algorithms*, 21(2):358 – 402, 1996. doi:10.1006/jagm.1996.0049.
- [41] Hans L. Bodlaender and Arie M. C. A. Koster. Safe Separators for Treewidth. *Discrete Mathematics*, 306(3):337–350, 2006. doi:10.1016/j.disc.2005.12.017.
- [42] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth Computations I. Upper bounds. *Information and Computation*, 208(3):259–275, 2010. doi:10.1016/j.ic.2009.03.008.
- [43] Hans L. Bodlaender, Arie M. C. A. Koster, Frank van den Eijkhof, and Linda C. van der Gaag. Pre-Processing for Triangulation of Probabilistic Networks. In *Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence, UAI’01, August 2–5, 2001, Seattle, Washington, USA*, pages 32–39. Morgan Kaufmann, 2001.
- [44] Vincent Bouchitté and Ioan Todinca. Treewidth and Minimum Fill-in of Weakly Triangulated Graphs. In *Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science, STACS 99, March 4–6, 1999, Trier, Germany*, Lecture Notes in Computer Science, pages 197–206. Springer, 1999. doi:10.1007/3-540-49116-3_18.
- [45] Bostjan Bresar, Frantisek Kardos, Ján Katrenic, and Gabriel Semanisin. Minimum k -Path Vertex Cover. *CoRR*, abs/1012.2088, 2010.
- [46] Liming Cai, Jianer Chen, Rodney G. Downey, and Michael R. Fellows. Advice Classes of Parameterized Tractability. *Annals of Pure and Applied Logic*, 84(1):119–138, 1997. doi:10.1016/S0168-0072(95)00020-8.
- [47] Marco Cesati and Miriam Di Ianni. Parameterized Parallel Complexity. In *Proceedings of the 4th International Conference on Parallel Processing, Euro-Par 1998, September 1–4, 1998, Southampton, UK*, volume 1470 of *Lecture Notes in Computer Science*, pages 892–896. Springer, 1998. doi:10.1007/BFb0057945.

- [48] Donald Chai and Andreas Kuehlmann. Circuit-Based Preprocessing of ILP and Its Applications in Leakage Minimization and Power Estimation. In *Proceedings of the 22nd IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD 2004)*, 11–13 October 2004, San Jose, CA, USA, pages 387–392. IEEE Computer Society, 2004. doi:10.1109/ICCD.2004.1347951.
- [49] James Cheetham, Frank K. H. A. Dehne, Andrew Rau-Chaplin, Ulrike Stege, and Peter J. Taillon. Solving Large FPT Problems on Coarse-Grained Parallel Machines. *Journal of Computer and System Sciences*, 67(4):691–706, 2003. doi:10.1016/S0022-0000(03)00075-8.
- [50] Hubie Chen and Moritz Müller. The Fine Classification of Conjunctive Queries and Parameterized Logarithmic Space. *ACM Transactions on Computation Theory*, 7(2):7:1–7:27, 2015. doi:10.1145/2751316.
- [51] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex Cover: Further Observations and Further Improvements. *Journal of Algorithms*, 41(2):280–301, 2001. doi:10.1006/jagm.2001.1186.
- [52] Yijia Chen and Jörg Flum. Some Lower Bounds in Parameterized AC^0 . In *Proceedings of the 41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22–26, 2016, Kraków, Poland*, volume 58 of *LIPICs*, pages 27:1–27:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICs.MFCS.2016.27.
- [53] Yijia Chen, Jörg Flum, and Xuanguai Huang. Slicewise Definability in First-Order Logic with Bounded Quantifier Rank. In *Proceedings of the 26th Annual Conference on Computer Science Logic, CSL 2017, August 20–24, 2017, Stockholm, Sweden*, *LIPICs*, pages 19:1–19:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.CSL.2017.19.
- [54] Benny Chor, Mike Fellows, and David W. Juedes. Linear Kernels in Linear Time, or How to Save k Colors in $O(n^2)$ Steps. In *Proceedings of the 30th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2004, June 21–23, 2004, Bad Honnef, Germany*, *Lecture Notes in Computer Science*, pages 257–269. Springer, 2004. doi:10.1007/978-3-540-30559-0_22.
- [55] Stephen A. Cook. A Taxonomy of Problems With Fast Parallel Algorithms. *Information and Control*, 64(1):2 – 22, 1985. doi:10.1016/S0019-9958(85)80041-3.
- [56] Stephen A. Cook and Pierre McKenzie. Problems Complete for Deterministic Logarithmic Space. *Journal of Algorithms*, 8(3):385–394, 1987. doi:10.1016/0196-6774(87)90018-6.

- [57] Bruno Courcelle. Graph Rewriting: An Algebraic and Logic Approach. In *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 193–242. Elsevier, Amsterdam, Netherlands and MIT Press, Cambridge, Massachusetts, 1990. doi : 10.1016/B978-0-444-88074-1.50010-X.
- [58] Gabriel Cramer. *Introduction à l'analyse des lignes courbes algébriques*. Chez les Frères Cramer et Cl. Philibert, 1750.
- [59] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015. doi : 10.1007/978-3-319-21275-3.
- [60] Elias Dahlhaus, Marek Karpinski, and Mark B. Novick. Fast Parallel Algorithms for the Clique Separator Decomposition. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms, 22–24 January 1990, San Francisco, California, USA*, pages 244–251. SIAM, 1990.
- [61] Bireswar Das, Murali Krishna Enduri, and I. Vinod Reddy. On the Parallel Parameterized Complexity of the Graph Isomorphism Problem. In *Proceedings of the 12th International Conference on Algorithms and Computation, WALCOM 2018, March 3–5, 2018, Dhaka, Bangladesh*, Lecture Notes in Computer Science, pages 252–264. Springer, 2018. doi : 10.1007/978-3-319-75172-6_22.
- [62] Rina Dechter and Itay Meiri. Experimental Evaluation of Preprocessing Techniques in Constraint Satisfaction Problems. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence, IJCAI 1989, August 1989, Detroit, MI, USA*, pages 271–277. Morgan Kaufmann, 1989.
- [63] Holger Dell, Thore Husfeldt, Bart M. P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. The First Parameterized Algorithms and Computational Experiments Challenge. In *Proceedings of the 11th International Symposium on Parameterized and Exact Computation, IPEC 2016, August 24–26, 2016, Aarhus, Denmark*, LIPIcs, pages 30:1–30:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi : 10.4230/LIPIcs.IPEC.2016.30.
- [64] Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. In *Proceedings of the 12th International Symposium on Parameterized and Exact Computation, IPEC 2017, September 6–8, 2017, Vienna, Austria*, LIPIcs, pages 30:1–30:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi : 10.4230/LIPIcs.IPEC.2017.30.
- [65] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

- [66] Rod Downey and Michael Fellows. Fixed-parameter Tractability and Completeness III: Some Structural Aspects of the W Hierarchy. In *Complexity Theory*, pages 191–225. Cambridge University Press, New York, NY, USA, 1993.
- [67] Rodney G. Downey and Michael R. Fellows. Fixed-Parameter Tractability and Completeness I: Basic Results. *SIAM Journal on Computing*, 24(4):873–921, 1995. doi : 10 . 1137/S0097539792228228.
- [68] Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness II: on completeness for W[1]. *Theoretical Computer Science*, 141(1&2):109–131, 1995. doi : 10 . 1016/0304-3975(94)00097-3.
- [69] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer, 1999. doi : 10 . 978 . 14612/67980.
- [70] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013. doi : 10 . 978 . 14471/55584.
- [71] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite Model Theory*. Springer, 1995. doi : 10 . 1007/3-540-28788-4.
- [72] Heinz-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas. *Mathematical Logic*. Springer, 1994. doi : 10 . 1007/978-1-4757-2355-7.
- [73] Niklas Eén and Armin Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing, SAT 2005, June 19–23, 2005, St. Andrews, UK*, Lecture Notes in Computer Science, pages 61–75. Springer, 2005. doi : 10 . 1007/11499107_5.
- [74] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace Versions of the Theorems of Bodlaender and Courcelle. In *Proceedings of the 51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, October 23–26, 2010, Las Vegas, USA*, pages 143–152. IEEE Computer Society, Los Alamitos, California, 2010. doi : 10 . 1109/FOCS . 2010 . 21.
- [75] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Algorithmic Meta Theorems for Circuit Classes of Constant and Logarithmic Depth. In *Proceedings of the 29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012, February 29th – March 3rd, 2012, Paris, France*, pages 66–77. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2012. doi : 10 . 4230/LIPIcs . STACS . 2012 . 66.
- [76] Michael Elberfeld, Christoph Stockhusen, and Till Tantau. On the Space and Circuit Complexity of Parameterized Problems: Classes and Completeness. *Algorithmica*, 71(3):661–701, 2015. doi : 10 . 1007/s00453-014-9944-y.

- [77] Paul Erdős and Richard Rado. Intersection Theorems for Systems of Sets. *Journal of the London Mathematical Society*, 1(1):85–90, 1960.
- [78] Ronald Fagin. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. *Complexity of Computation*, 7:27–41, 1974.
- [79] Stephen A. Fenner, Rohit Gurjar, and Thomas Thierauf. Bipartite Perfect Matching is in quasi-NC. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, June 18–21, 2016, Cambridge, MA, USA*, pages 754–763. ACM, 2016. doi : 10 . 1145/2897518 . 2897564.
- [80] Johannes Klaus Fichte. gtfs2graphs – a transit feed to graph format converter. <http://www.github.com/daajoe/gtfs2graphs>. Accessed: 20.04.2018; Commit: 2199448.
- [81] Johannes Klaus Fichte, Markus Hecher, Neha Lodha, and Stefan Szeider. An SMT Approach to Fractional Hypertree Width. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming, CP 2018, August 27–31, 2018, Lille, France*, Lecture Notes in Computer Science, pages 109–127. Springer, 2018. doi : 10 . 1007/978–3–319–98334–9_8.
- [82] Johannes Klaus Fichte, Markus Hecher, Stefan Woltran, and Markus Zisser. Weighted Model Counting on the GPU by Exploiting Small Treewidth. In *Proceedings of the 26th Annual European Symposium on Algorithms, ESA 2018, August 20–22, 2018, Helsinki, Finland*, LIPIcs, pages 28:1–28:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi : 10 . 4230/LIPIcs . ESA . 2018 . 28.
- [83] Johannes Klaus Fichte, Neha Lodha, and Stefan Szeider. SAT-Based Local Improvement for Finding Tree Decompositions of Small Width. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing, SAT 2017, August 28 – September 1, 2017, Melbourne, VIC, Australia*, pages 401–411, 2017. doi : 10 . 1007/978–3–319–66263–3_25.
- [84] Jörg Flum and Martin Grohe. Describing Parameterized Complexity Classes. *Information and Computation*, 187(2):291–319, 2003. doi : 10 . 1016/S0890–5401(03)00161–5.
- [85] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Springer, 2006. doi : 10 . 978 . 36420/67570.
- [86] Fedor V. Fomin, Pierre Fraigniaud, and Nicolas Nisse. Nondeterministic Graph Searching: From Pathwidth to Treewidth. *Algorithmica*, 53(3):358–373, 2009. doi : 10 . 1007/s00453–007–9041–6.

- [87] Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2010. doi:10.1007/978-3-642-16533-7.
- [88] Fedor V. Fomin, Dieter Kratsch, Ioan Todinca, and Yngve Villanger. Exact Algorithms for Treewidth and Minimum Fill-In. *SIAM Journal on Computing*, 38(3):1058–1079, 2008. doi:10.1137/050643350.
- [89] Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, Michal Pilipczuk, and Marcin Wrochna. Fully Polynomial-Time Parameterized Computations for Graphs and Matrices of Low Treewidth. *ACM Transactions on Algorithms*, 14(3):34:1–34:45, 2018. doi:10.1145/3186898.
- [90] Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, and Meirav Zehavi. *Kernelization: Theory of Parameterized Preprocessing*. Cambridge University Press, 2018.
- [91] Markus Frick and Martin Grohe. The Complexity of First-Order and Monadic Second-Order Logic Revisited. *Annals of Pure and Applied Logic*, 130(1-3):3–31, 2004. doi:10.1016/j.apal.2004.01.007.
- [92] Alan M. Frisch and Paul A. Giannaros. SAT Encodings of the At-Most-k Constraint Some Old , Some New , Some Fast , Some Slow. In *Proceedings of the 10th International Workshop on Constraint Modelling and Reformulation, ModRef2011, 12 September, 2011, Perugia, Italy*. EASST, 2011.
- [93] Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, Circuits, and the Polynomial-Time Hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984. doi:10.1007/BF01744431.
- [94] Haim Gaifman. On Local and Non-Local Properties. In *Proceedings of the Herbrand Symposium*, volume 107 of *Studies in Logic and the Foundations of Mathematics*, pages 105–135. Elsevier, 1982. doi:10.1016/S0049-237X(08)71879-2.
- [95] Moses Ganardi. Matching-Based Algorithms for Computing Treewidth. Bachelor thesis, Rheinisch-Westfälische Technische Hochschule Aachen, 2012.
- [96] Robert Ganian, Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. SAT-Encodings for Treecut Width and Treedepth. In *Proceedings of the Proceedings of the 21th Workshop on Algorithm Engineering and Experiments, ALENEX 2019, January 7–8, 2019, San Diego, CA, USA*, pages 117–129. SIAM, 2019. doi:10.1137/1.9781611975499.10.
- [97] Fanica Gavril. Algorithms on Clique Separable Graphs. *Discrete Mathematics*, 19(2):159–165, 1977. doi:10.1016/0012-365X(77)90030-9.

- [98] Archontia C. Giannopoulou, Paul Hunter, and Dimitrios M. Thilikos. LIFO-Search: A Min-Max Theorem and a Searching Game for Cycle-Rank and Tree-Depth. *Discrete Applied Mathematics*, 160(15):2089–2097, 2012. doi : 10.1016/j.dam.2012.03.015.
- [99] Vibhav Gogate and Rina Dechter. A Complete Anytime Algorithm for Treewidth. In *Proceedings of the 20th Conference in Uncertainty in Artificial Intelligence, UAI '04, July 7-11, 2004, Banff, Canada*, pages 201–208. AUAI Press, 2004.
- [100] Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel Symmetry-Breaking in Sparse Graphs. *SIAM Journal on Discrete Mathematics*, 1(4):434–446, 1988. doi : 10.1137/0401044.
- [101] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computations*. Oxford University Press, 1995.
- [102] Jiong Guo, Rolf Niedermeier, and Sebastian Wernicke. Parameterized Complexity of Generalized Vertex Cover Problems. In *Proceedings of the 9th International Workshop on Algorithms and Data Structures, WADS 2005, August 15-17, 2005, Waterloo, Canada*, Lecture Notes in Computer Science, pages 36–48. Springer, 2005. doi : 10.1007/11534273_5.
- [103] Joseph Y. Halpern, Robert Harper, Neil Immerman, Phokion G. Kolaitis, Moshe Y. Vardi, and Victor Vianu. On the Unusual Effectiveness of Logic in Computer Science. *Bulletin of Symbolic Logic*, 7(2):213–236, 2001.
- [104] Tom Hartmann, Max Bannach, and Martin Middendorf. Sorting Signed Permutations by Inverse Tandem Duplication Random Losses. In *Proceedings of the 17th Asia Pacific Bioinformatics Conference, APBC 2019, January 14–16, 2019, Wuhan, China*, 2019.
- [105] Tom Hartmann, Max Bannach, and Martin Middendorf. Sorting Signed Permutations by Inverse Tandem Duplication Random Losses. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pages 1–1, 2019. doi : 10.1109/TCBB.2019.2917198.
- [106] William Hesse. Division Is in Uniform TC^0 . In *Proceedings of the 28th International Colloquium on Automata, Languages, and Programming, ICALP 2001, July 2001, Crete, Greece*, Lecture Notes in Computer Science, pages 104–114. Springer, 2001. doi : 10.1007/3-540-48224-5_9.
- [107] Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. In *Hardware and Software: Verification and Testing - Revised Selected Papers of the 7th International Haifa Verification Conference, HVC 2011, December 6–8, 2011, Haifa, Israel*, Lecture Notes in Computer Science, pages 50–65. Springer, 2011. doi : 10.1007/978-3-642-34188-5_8.

- [108] John E. Hopcroft and Robert Endre Tarjan. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Communiacion of the ACM*, 16(6):372–378, 1973. doi : 10.1145/362248.362272.
- [109] John E. Hopcroft and Robert Endre Tarjan. Dividing a Graph into Triconnected Components. *SIAM Journal on Computing*, 2(3):135–158, 1973. doi : 10.1137/0202012.
- [110] Neil Immerman. Languages which Capture Complexity Classes. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, STOC 1983, 25–27 April, 1983, Boston, Massachusetts, USA*, pages 347–354. ACM New York, NY, 1983. doi : 10.1145/800061.808765.
- [111] Neil Immerman. *Descriptive Complexity*. Graduate texts in computer science. Springer, 1999. doi : 10.1007/978-1-4612-0539-5.
- [112] Alon Itai, Michael Rodeh, and Steven L. Tanimoto. Some Matching Problems for Bipartite Graphs. *Journal of the ACM*, 25(4):517–525, 1978. doi : 10.1145/322092.322093.
- [113] Riko Jacob, Tobias Lieber, and Matthias Mnich. Treewidth Computation and Kernelization in the Parallel External Memory Model. In *Proceedings of the 8th International Conference on Theoretical Computer Science, TCS 2014, September 1–3, 2014, Rome, Italy*, Lecture Notes in Computer Science, pages 78–89. Springer, 2014. doi : 10.1007/978-3-662-44602-7_7.
- [114] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
- [115] Richard M. Karp. Reducibility Among Combinatorial Problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.
- [116] Richard M. Karp and Avi Wigderson. A Fast Parallel Algorithm for the Maximal Independent Set Problem. *Journal of the ACM*, 32(4):762–773, 1985. doi : 10.1145/4221.4226.
- [117] Joachim Kneis and Alexander Langer. A Practical Approach to Courcelle’s Theorem. *Electronic Notes in Theoretical Computer Science*, 251:65 – 81, 2009.
- [118] Joachim Kneis, Alexander Langer, and Peter Rossmanith. Courcelle’s Theorem - A Game-Theoretic Approach. *Discrete Optimization*, 8:568–594, 2011.
- [119] Yasuaki Kobayashi and Hisao Tamaki. Treedepth Parameterized by Vertex Cover Number. In *Proceedings of the 11th International Symposium on Parameterized and Exact Computation, IPEC 2016, August 24–26, 2016, Aarhus, Denmark, LIPIcs*, pages 18:1–18:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi : 10.4230/LIPIcs.IPEC.2016.18.

- [120] Denes König. Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre. *Mathematische Annalen*, 77(4):453–465, 1916. doi : 10 . 1007 / BF01456961.
- [121] Antoni A. Kosinski. Cramer’s Rule Is Due to Cramer. *Mathematics Magazine*, 74, 10 2001. doi : 10 . 2307 / 2691101.
- [122] Stefan Kratsch, Geevarghese Philip, and Saurabh Ray. Point Line Cover: The Easy Kernel is Essentially Tight. *ACM Transactions on Algorithms*, 12(3):40:1–40:16, 2016. doi : 10 . 1145 / 2832912.
- [123] Stephan Kreutzer. Algorithmic Meta-Theorems. *Electronic Colloquium on Computational Complexity (ECCC)*, 16:147, 2009.
- [124] Jens Lagergren. Efficient Parallel Algorithms for Graphs of Bounded Tree-Width. *Journal of Algorithms*, 20:20–44, 1996. doi : 10 . 1006 / jagm . 1996 . 0002.
- [125] Michael Lampis. A Kernel of Order $2k - c \log k$ for Vertex Cover. *Information Processing Letters*, 111(23–24):1089–1091, 2011. doi : 10 . 1016 / j . ipl . 2011 . 09 . 003.
- [126] Alexander Langer. *Fast Algorithms for Decomposable Graphs*. PhD thesis, RWTH Aachen University, 2013.
- [127] Andrea S. LaPaugh. Recontamination Does Not Help to Search a Graph. *Journal of the ACM*, 40:224–245, 1993. doi : 10 . 1145 / 151261 . 151263.
- [128] Lukas Larisch and Felix Salfelder. p17.
<https://github.com/freetdi/p17>.
 Accessed: 02.08.2017; Commit: 552341d.
- [129] Van Bang Le and Florian Pfender. Complexity Results for Rainbow Matchings. *Theoretical Computer Science*, 524:27–33, 2014. doi : 10 . 1016 / j . tcs . 2013 . 12 . 013.
- [130] Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986. doi : 10 . 1137 / 0215074.
- [131] Frédéric Mazoit and Nicolas Nisse. Monotonicity of Non-Deterministic Graph Searching. *Theoretical Computer Science*, 399(3):169–178, 2008. doi : 10 . 1016 / j . tcs . 2008 . 02 . 036.
- [132] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as Easy as Matrix Inversion. *Combinatorica*, 7(1):105–113, 1987. doi : 10 . 1007 / BF02579206.

- [133] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Preprocessing in Incremental SAT. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing, SAT 2012, June 17–20, 2012, Trento, Italy*, Lecture Notes in Computer Science, pages 256–269. Springer, 2012. doi: 10.1007/978-3-642-31612-8_20.
- [134] George L. Nemhauser and Leslie E. Trotter Jr. Properties of Vertex Packing and Independence System Polyhedra. *Mathematical Programming*, 6(1):48–61, 1974. doi: 10.1007/BF01580222.
- [135] Jaroslav Nešetřil and Patrice Ossona de Mendez. *Sparsity*. Springer, 2012. doi: 10.1007/978-3-642-27875-4.
- [136] Ilan Newman, Prabhakar Ragde, and Avi Wigderson. Perfect Hashing, Graph Entropy, and Circuit Complexity. In *Proceedings of the 5th Annual Structure in Complexity Theory Conference, CoCo 1990, July 8–11, 1990, Barcelona, Spain*, pages 91–99. IEEE Computer Society, Los Alamitos, California, 1990. doi: 10.1109/SCT.1990.113958.
- [137] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006. doi: 10.1093/acprof:oso/9780198566076.001.0001.
- [138] Marián Novotný. Design and Analysis of a Generalized Canvas Protocol. In *Proceedings of the 4th International Workshop on Information Security Theory and Practices. Security and Privacy of Pervasive Systems and Smart Devices, WISTP 2010, April 12–14, 2010, Passau, Germany*, Lecture Notes in Computer Science, pages 106–121. Springer, 2010. doi: 10.1007/978-3-642-12368-9_8.
- [139] Participants of Shonan Meeting 144. personal communication.
- [140] Michał Pilipczuk, Sebastian Siebertz, and Szymon Toruńczyk. Parameterized Circuit Complexity of Model-Checking on Sparse Structures. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, July 09–12, 2018, Oxford, UK*, pages 789–798. ACM, 2018. doi: 10.1145/3209108.3209136.
- [141] Nicholas Pippenger. On Simultaneous Resource Bounds (Preliminary Version). In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science, FOCS 1979, 29–31 October 1979, San Juan, Puerto Rico*, pages 307–311. IEEE Computer Society, 1979. doi: 10.1109/SFCS.1979.29.
- [142] Bruce A. Reed. *Algorithmic Aspects of Tree Width*, pages 85–107. Springer, New York, NY, 2003. doi: 10.1007/0-387-22444-0_4.

- [143] Omer Reingold. Undirected ST-Connectivity in Log-Space. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, STOC 2005, May 22–24, 2005, Baltimore, MD, USA*, pages 376–385. ACM, 2005. doi:10.1145/1060590.1060647.
- [144] Neil Robertson, Daniel P. Sanders, Paul D. Seymour, and Robin Thomas. Efficiently Four-Coloring Planar Graphs. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing, STOC '96, May 22–24, 1996, Philadelphia, Pennsylvania, USA*, pages 571–575. ACM, 1996. doi:10.1145/237814.238005.
- [145] Neil Robertson and Paul D. Seymour. Graph Minors. II. Algorithmic Aspects of Tree-Width. *Journal of Algorithms*, 7(3):309–322, 1986. doi:10.1016/0196-6774(86)90023-4.
- [146] Neil Robertson and Paul D. Seymour. Graph Minors. XIII. The Disjoint Paths Problem. *Journal of Combinatorial Theory*, 63(1):65–110, 1995. doi:10.1007/978-3-540-24605-3_37.
- [147] Hein Röhrig. Tree Decomposition: A Feasibility Study. Diploma thesis, Max-Planck-Institut für Informatik in Saarbrücken, 1998.
- [148] Sivan Sabato and Yehuda Naveh. Preprocessing Expression-Based Constraint Satisfaction Problems for Stochastic Local Search. In *Proceedings of the 4th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2007, May 23–26, 2007, Brussels, Belgium*, Lecture Notes in Computer Science, pages 244–259. Springer, 2007. doi:10.1007/978-3-540-72397-4_18.
- [149] Marko Samer and Helmut Veith. Encoding Treewidth into SAT. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, SAT 2009, June 30 – July 3, 2009, Swansea, UK*, pages 45–50, 2009. doi:10.1007/978-3-642-02777-2_6.
- [150] Iztok Savič. Index Data Structure for Fast Subset and Superset Queries. In *Proceedings of the 5th International Cross-Domain Conference on Availability, Reliability, and Security in Information Systems and HCI, IFIP WG 2013, September 2–6, Regensburg, Germany*, pages 134–148, 2013. doi:10.1007/978-3-642-40511-2_10.
- [151] Detlef Seese. Linear Time Computable Problems and Logical Descriptions. *Electronic Notes in Theoretical Computer Science*, 2:246–259, 1995. doi:10.1016/S1571-0661(05)80203-8.
- [152] Paul D. Seymour and Robin Thomas. Graph Searching and a Min-Max Theorem for Tree-Width. *Journal of Combinatorial Theory*, 58:22–33, 1993. doi:10.1006/jctb.1993.1027.

- [153] Arezou Soleimanfallah and Anders Yeo. A Kernel of Order $2k - c$ for Vertex Cover. *Discrete Mathematics*, 311(10–11):892–895, 2011. doi:10.1016/j.disc.2011.02.014.
- [154] Christoph Stockhusen. *On the Space and Circuit Complexity of Parameterized Problems*. PhD thesis, University of Lübeck, Germany, 2017.
- [155] Larry J. Stockmeyer and Uzi Vishkin. Simulation of Parallel Random Access Machines by Circuits. *SIAM Journal on Computing*, 13(2):409–422, 1984. doi:10.1137/0213027.
- [156] Hisao Tamaki. treewidth-exact.
<https://github.com/TCS-Meiji/treewidth-exact>.
 Accessed: 02.08.2017; Commit: d5ba92a.
- [157] Hisao Tamaki. treewidth-exact.
<https://github.com/TCS-Meiji/PACE2017-TrackA>.
 Accessed: 30.05.2019; Commit: 7278390.
- [158] Hisao Tamaki. Positive-Instance Driven Dynamic Programming for Treewidth. In *Proceedings of the 25th Annual European Symposium on Algorithms, ESA 2017, September 4–6, 2017, Vienna, Austria*, LIPIcs, pages 68:1–68:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ESA.2017.68.
- [159] Hisao Tamaki and Hans L. Bodlaender. personal communication.
- [160] Till Tantau. Logspace Optimisation Problems and their Approximation Properties. *Theory of Computing Systems*, 41(2):327–350, 2007. doi:10.1007/s00224-007-2011-1.
- [161] Robert Endre Tarjan. Decomposition by Clique Separators. *Discrete Mathematics*, 55(2):221–232, 1985. doi:10.1016/0012-365X(85)90051-2.
- [162] Jianhua Tu and Wenli Zhou. A factor 2 Approximation Algorithm for the Vertex Cover P_3 Problem. *Information Processing Letters*, 111(14):683–686, 2011. doi:10.1016/j.ipl.2011.04.009.
- [163] Tom C. van der Zanden and Hans L. Bodlaender. Computing Treewidth on the GPU. In *Proceedings of the 12th International Symposium on Parameterized and Exact Computation, IPEC 2017, September 6–8, 2017, Vienna, Austria*, LIPIcs, pages 29:1–29:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.IPEC.2017.29.
- [164] Heribert Vollmer. *Introduction to Circuit Complexity*. Springer, 1999. doi:10.978.36420/83983.

CURRICULUM VITAE

Date of Birth 26.12.1990
Place of Birth Halle (Saale), Germany
Nationality German
Marital Status Married



EDUCATION

- M.Sc. Universität zu Lübeck, December 2014
Title: *On the Space and Circuit Complexity of Certain Parameterized Problems*
Advisor: Prof. Dr. Till Tantau
- B.Sc. Universität zu Lübeck, November 2012
Title: *Berechnungskomplexitäten von Varianten des Subset-Sum-Problems*
Advisor: Prof. Dr. Till Tantau
- Abitur Bismarck Gymnasium Genthin, June 2009
Focus Subjects: *Mathematics, Biology*

FELLOWSHIPS AND AWARDS

- 2018 ESA Track B Best Student Paper Award
2017 Third Place at the PACE Challenge Track A (exact)
2017 Fourth Place at the PACE Challenge Track A (heuristic)
2016 Third Place at the PACE Challenge Track A (exact)
2016 Fifth Place at the PACE Challenge Track A (heuristic)
2015 Stipend for the *Graduate School for Computing in Medicine and Life Science* at the Universität zu Lübeck
2013 Place 54 at the Northwestern Europe Regional Contest (NWERC) in Delft (ACM International Collegiate Programming Contest)

PUBLICATIONS

- [19] Max Bannach, Christoph Stockhusen, and Till Tantau: *Fast Parallel Fixed-Parameter Algorithms via Color Coding*. In Proceedings of the 10th International Symposium on Parameterized and Exact Computation (IPEC 2015).
- [20] Max Bannach and Till Tantau: *Parallel Multivariate Meta-Theorems*. In Proceedings of the 11th International Symposium on Parameterized and Exact Computation (IPEC 2016).
- [16] Max Bannach, Sebastian Berndt, and Thorsten Ehlers: *Jdrasil: A Modular Library for Computing Tree Decompositions*. In Proceedings of the 16th International Symposium on Experimental Algorithms (SEA 2017).
- [21] Max Bannach and Till Tantau: *Computing Hitting Set Kernels By AC^0 -Circuits*. In Proceedings of the 35th Symposium on Theoretical Aspects of Computer Science (STACS 2018).
- [22] Max Bannach and Till Tantau: *Computing Kernels in Parallel: Lower and Upper Bounds*. In Proceedings of the 13th International Symposium on Parameterized and Exact Computation (IPEC 2018).
- [17] Max Bannach, Sebastian Berndt, Thorsten Ehlers, and Dirk Nowotka: *SAT-Encodings of Tree Decompositions*. In Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions.
- [13] Max Bannach and Sebastian Berndt: *Practical Access to Dynamic Programming on Tree Decompositions*. In Proceedings of the 26th Annual European Symposium on Algorithms (ESA 2018).
- [18] Max Bannach, Malte Skambath and Till Tantau: *Towards Work-Efficient Parallel Parameterized Algorithms*. In Proceedings of the 13th International Conference and Workshops on Algorithms and Computation (WALCOM 2018).
- [23] Max Bannach and Till Tantau: *On the Descriptive Complexity of Color Coding*. In Proceedings of the 36th Symposium on Theoretical Aspects of Computer Science (STACS 2019).
- [14] Max Bannach and Sebastian Berndt: *Positive-Instance Driven Dynamic Programming for Graph Searching*. In Proceedings of 16th Algorithms and Data Structures Symposium (WADS 2019).
- [104] Tom Hartmann, Max Bannach, and Martin Middendorf: *Sorting Signed Permutations by Inverse Tandem Duplication Random Losses*. In Proceedings of the 17th Asia Pacific Bioinformatics Conference (APBC 2019).

- [105] Tom Hartmann, Max Bannach, and Martin Middendorf: *Sorting Signed Permutations by Inverse Tandem Duplication Random Losses*. In IEEE/ACM Transactions on Computational Biology and Bioinformatics.

TEACHING AND SCIENTIFIC DUTIES

From 2014 to 2019 I worked as scientific staff at the Institute for Theoretical Computer Science of the Universität zu Lübeck. During this time I was repeatedly a teaching assistant for the following courses:

- Theoretical Computer Science
- Algorithmic, Logic, and Complexity
- Computational Complexity
- Algorithm Design

I wrote reviews for STACS, MFCS, FCT, ESA, IPEC, and JAIR, and I assisted in the supervision of the following bachelor and master theses:

Bachelor Theses

- Alexander Droigk: *Design and Implementation of Edge Routing Algorithms in TikZ*
- Gilian Henke: *Algorithmic Drawing of Automata*
- Keanu Soegiharto: *Semantic Web Technology in the Study of Proteins*
- Thorsten Peinemann: *Efficient Parallel Kernel Algorithms for the Vertex Cover Problem*

Master Theses

- Ruben Beyer: *Algorithmic Drawing of Planar Graphs with TikZ*
- Alexandra Lassota: *Possibilities and Limitations of Parallel Kernel Computation*
- Friederike Bartels: *Constructing Solutions For Unary Subset Sum and Similar Problems in TC⁰*
- Zacharias Heinrich: *Dynamic Kernelisations for Vertex Cover and Hitting Set*

PROFESSIONAL AFFILIATIONS

- 2014–present Association for Computing Machinery (ACM)
 2015–present European Association for Theoretical Computer Science (EATCS)
 2016–present Gesellschaft für Informatik (GI)