CONJUNCTIVE QUERIES, ARITHMETIC CIRCUITS AND COUNTING COMPLEXITY

Dissertation

zur Erlangung des Doktorgrades
der Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

vorgelegt von

Stefan Mengel

Paderborn, 21. Mai 2013

"We can only see a short distance ahead,
but we can see plenty there that needs to be done."

—Alan Turing [Tur50]

ABSTRACT

This thesis deals with several subjects from counting complexity and arithmetic circuit complexity.

The first part explores the complexity of counting solutions to conjunctive queries, which are a basic class of queries from database theory. We introduce a parameter, called the *quantified star size* of a query ϕ, which measures how the free variables are spread in ϕ. As usual in database theory, we associate a hypergraph to a query ϕ. We show that for classes of queries for which these associated hypergraphs have bounded generalized hypertree width, bounded quantified star size exactly characterizes the subclasses of queries for which counting the number of solutions is tractable. In the case of bounded arity, this allows us to fully characterize the classes of conjunctive queries for which counting the solutions is tractable. Finally, we also analyze the complexity of computing the quantified star size of a conjunctive query.

In the second part we characterize different classes from arithmetic circuit complexity by different means, including conjunctive queries and constraint satisfaction problems, graph polynomials on bounded treewidth graphs, and an extension of the classical arithmetic branching program model by stack memory. In particular, this yields new characterizations of the arithmetic circuit class VP, a class that is central to the area but arguably not well understood.

Finally, the third part studies the complexity of two questions on polynomials given by arithmetic circuits: testing whether a monomial is present and counting the number of its monomials. We show that these problems are complete for different levels of the counting hierarchy, which had few or no known natural complete problems before.

ZUSAMMENFASSUNG

In dieser Arbeit geht es um verschiedene Themen aus der Zählkomplexität und der arithmetischen Schaltkreiskomplexität.

Der erst Teil untersucht die Komplexität des Zählens der Antworten auf Conjunctive Queries, eine grundlegende Klasse von Anfragen aus der Datenbanktheorie. Wir führen den Parameter *quantified star size* einer Query ϕ ein, der misst, wie die freien Variablen in ϕ verteilt sind. Wir ordnen der Query ϕ einen Hypergraphen zu und zeigen, dass für Klassen von Queries mit beschränkter generalized hypertree width der Parameter quantified star size genau die Unterklassen charakterisiert, für die das Zählen von Antworten effizient möglich

ist. Dies erlaubt uns, im Fall beschränkter Arität die Klassen von Conjunctive Queries, für die effizientes Zählen von Antworten möglich ist, vollständig zu charakterisieren. Weiterhin betrachten wir auch die Kompexität der Berechnung der quantified star size von Conjunctive Queries.

Im zweiten Teil der Arbeit charakterisieren wir unterschiedliche Klassen aus der arithmetischen Schaltkreiskomplexität auf verschiedene Arten, und zwar durch Conjunctive Queries and Constraint Satisfaction Probleme, durch Graphpolynome auf Graphen beschränkter Baumweite und durch eine Erweiterung des klassischen Modells der arithmetischen Branchingprogramme durch Stack-Speicher. Insbesondere zeigen wir neue Charakterisierungen der arithmetischen Schaltkreisklasse VP, einer Klasse, die zentral für den Bereich ist aber dennoch nicht gut verstanden.

Der dritte Teil schließlich beschäftigt sich mit zwei Entscheidungproblemen zu Polynomen gegeben durch arithmetische Schaltkreise: Testen ob ein gegebenens Monom vorkommt und Zählen der vorkommenden Monomen. Wir zeigen, dass diese Probleme vollständig sind für unterschiedliche Levels der sogenannten counting hierarchy, für die bisher wenige oder keine natürlichen vollständigen Probleme bekannt waren.

ACKNOWLEDGMENTS

After more than four years of work on this thesis there are many people I would like to thank.

First and most of all I would like to thank my wife Hilke for letting me do this although it was not always easy for her.

I am grateful to my advisor Peter Bürgisser for, on the hand, giving me the freedom to pursue my own research and, one the other hand, being supportive when I needed his opinion or help. In particular, I would also like to thank him for giving me the opportunity to meet many other people in the community. Also, I am thankful to him for relentlessly making me improve the presentation of this thesis (of course I take full responsibility for all shortcomings and mistakes still in it).

I would like to thank Friedhelm Meyer auf der Heide and Luc Segoufin who kindly agreed to act as reviewers of this thesis.

Arnaud Durand has played a crucial role in the creation of this thesis. He patiently explained to me everything I know about logic, introduced me to several areas of computer science, and made key contributions to my research. He also gave me professional and personal advice whenever I needed it. Finally, he made my stays in Paris possible by getting the necessary funding for me. I am very thankful for all of his support and I hope to pay back his kindness one day.

I am very grateful to Guillaume Malod for several reasons. First, without his invitation to Paris to work together this thesis would not be what it is now. Also, he let me share some of his deep understanding of arithmetic circuit complexity. Last but not least I am grateful for his support in my (often hopeless) struggles with French bureaucracy.

I value also a lot the support on a professional and personal level that I received from Hervé Fournier during my several stays in Paris.

My colleagues Dennis Amelunxen, Jesko Hüttenhain and Christian Ikenmeyer have been great companions during the last four years. Our discussions on math, computer science and life in general have sometimes been heated, but I value their support, their friendship and their opinions a lot. Moreover, I am thankful for the TikZ-support by Dennis and the LaTeX-support by Jesko.

I would like to thank the numerous people who I have shared offices with during the last few years for making my time at work so enjoyable. In particular, I will miss Maik Ringkamp's cheerful company when our common time is over.

I am thankful to Sandra Pelster and Inga Gill for their support and their friendliness.

I would like to thank Yann Strozecki for getting me and my family an apartment in Paris for one of my stays there. Moreover, I am grateful to Yanis Langeraert for letting me stay at his place for several months.

I would like to thank the members of the Équipe de Logique Mathématique of the Institut de Mathématiques de Jussieu at Université Paris 7 and the numerous people associated to this group for making me feel very welcome during my several stays in Paris.

I would like to thank Hubie Chen for his hospitality during my short stay in San Sebastián.

My stays in Paris would have been far less enjoyable if I had not learnt French before. I would like to thank my French teacher Sigrid Behrent for making this such a pleasure.

I am also grateful to Barbara and Jens-Peter Kempkes for taking care of our son Jakob on several days during the final phase of writing this thesis.

I would like to thank the organizers of the Dagstuhl Seminars 10481 and 13031 on Computational Counting. Some of the results in this thesis were conceived during these workshops. Moreover, meeting several people there has proved invaluable.

Sébastien Tavenas pointed out an error in an earlier version of the proof of Lemma 12.3.7. Later, he and Pascal Koiran helped me find the proof presented in this thesis. I would like to thank both of them for their contribution.

I would like to thank Friedhelm Meyer auf der Heide and his working group for their hospitality during the first phase of my doctoral studies.

The research in this thesis would not have been possible without the generous financial support by the Research Training Group GK-693 of the Paderborn Institute for Scientific Computation (PaSCo) and by the Deutsche Forschungsgemeinschaft (DFG-grants BU 1371/2-2 and BU 1371/3-1). Furthermore, the research leading to the results presented in this thesis has received funding from the [European Community's] Seventh Framework Programme [FP7/2007-2013] under grant agreement n° 238381. I am very grateful for this support.

CONTENTS

1 INTRODUCTION

Computational complexity theory is a subfield of theoretical computer science that aims to determine the amount of resources necessary to solve different computational problems. Which specific resources are considered depends on the computational model and the considered problem; typical examples include computation time, memory or the number of arithmetic operations needed to solve a problem.

The research in computational complexity can be roughly divided into two directions: In the first direction one tries to prove unconditional lower bounds for specific problems. For example there is a long line of research that tries to find lower bounds on the number of arithmetic operations necessary to multiply two $(n \times n)$-matrices. Unfortunately, proving strong lower bounds appears to be very hard. Despite considerable efforts in the last decades, known lower bounds are in most cases either much lower than the conjectured lower bounds or are only true for restricted computational models.

The second direction of computational complexity offers a way to still assess the hardness of specific problems in spite of the absence of unconditional good lower bounds. Instead of trying to show unconditional lower bounds, one compares the relative complexity of computational problems and computational models, typically by organization into different complexity classes. The most prominent and arguably most successful part of this direction is certainly the theory of NP-completeness, which essentially says that all NP-complete problems are roughly equally hard. Since we know several thousand NP-complete problems, and there is no known polynomial time algorithm for any of them, the commonly accepted conjecture is that all of these problems are intractable, i.e., cannot be solved in polynomial time. Computational complexity offers several other models that try to explain the hardness of problems by showing completeness for different complexity classes.

This thesis only deals with this second direction of computational complexity theory. Roughly half of the thesis is devoted to the understanding of so-called conjunctive queries, a subject originating from database theory. The other half deals with arithmetic circuit complexity, a classic subarea of computational complexity theory. While these directions at first sight appear to be very different, we will see that they are in fact closely connected.

In the following sections of this introduction we will briefly describe the considered areas and sketch aims and results of this thesis.

1.1 PART I: COUNTING SOLUTIONS TO CONJUNCTIVE QUERIES

Part i of this thesis aims at understanding the (counting) complexity of so-called conjunctive queries [CM77]. Conjunctive queries (CQs for short) are a fundamental class of queries from database theory. Equivalent to Select-Project-Join queries, they are the most basic class of database queries and at the same time play an important role in practice. Furthermore, as Kolaitis and Vardi [KV00] showed, conjunctive queries are intimitely connected to constraint satisfaction problems, a central area from artificial intelligence. These features make conjunctive queries the best-studied type of database queries.

A CQ-instance $(\mathcal{A}, \phi)$ consists of a *query* ϕ, which is a logical first-order $\{\exists, \wedge\}$-formula, also called primitive positive formula, and a finite structure $\mathcal{A}$. The structure $\mathcal{A}$ takes the role of the database in database theory, the relations of $\mathcal{A}$ are the tables of the database. The *query result* is

$$\phi(\mathcal{A}) := \{a \mid (\mathcal{A}, a) \models \phi(x)\},$$

that is, the set of assignments that make the query ϕ true.

When computational problems about CQ-instances are considered, the structure $\mathcal{A}$ is most of the time assumed to be encoded in the so-called explicit encoding by explicitly listing all tuples of all relations. This is justified because of the origin in database theory: Databases are usually stored by storing all tuples in all tables explicitly, which corresponds to the explicit encoding of CQ-instances.

Mainly three computational problems on conjunctive queries have been studied in the literature:

- The most widely considered problem is the so-called Boolean conjunctive query problem, denoted CQ, which is to decide for an instance $(\mathcal{A}, \phi)$ if $\phi(\mathcal{A})$ is non-empty, i.e., if the query ϕ has satisfying assignments with respect to $\mathcal{A}$. CQ in particular is very important in query minimization [CM77], where the task is, given a query ϕ, to compute a query ϕ' as small as possible such that ϕ and ϕ' yield the same query result on all databases.

- The problem of computing the complete query result is of great practical interest. As the query result can be of exponential size in the size of the input, it is clear that there can be no polynomial time algorithm in the input size. To remedy this, one considers output-polynomial algorithms, i.e., algorithms whose runtime is polynomial in the size of the input *and* the size of the output. A more restrictive setting are enumeration algorithms with polynomial delay. These algorithms print out the elements of the query result one after the other without repetitions, with only a polynomial delay in the size of the input between printing out two elements.

- The third computational problem on CQ-instances is computing the size of the query result. Since most practical database query languages like SQL have a counting operator, this is also a very natural question. This counting problem, denoted #CQ, is in the center of Part i of this thesis.

When the query ϕ is not allowed to have existential quantifiers—so all variables are free—, the problems above are also known as different versions of the constraint satisfaction problem (CSP). Constraint satisfaction is an important problem from artificial intelligence and has been extensively studied in that area. Observe that for CQ, existential quantification does not make a difference, so the names CSP and CQ are used almost synonymously. For enumeration and counting though, existential quantification makes a critical difference in the complexity (see e.g. [BDG07, PS13]). We denote the counting problem for CSP by #CSP.

By reduction from 3-SAT, it is straightforward to show that CQ is NP-complete, and thus all problems discussed above are in general considered as intractable. Because of their great practical importance, research has concentrated on working around this general hardness result by searching for tractable subclasses, which are primarily found in two directions:

The first approach is restricting the relations of the finite structure $\mathcal{A}$. This was pioneered by Schaefer in a seminal paper [Sch78], where he showed a dichotomy theorem for CQ with Boolean domain, i.e., all relations are subsets of $\{0,1\}^*$. Schaefer showed that there is a set S of relations such that if instances may only be built from relations in S, then CQ for this class of instances is polynomial time decidable. Furthermore, if even a single relation not in S is allowed as a building block for instances, the resulting class is NP-complete. This result has spawned a huge amount of followup work in which similar results were shown for different settings, recently also using tools from universal algebra (see e.g. [Bul11]).

While the results of the last paragraph are very strong, they are not fully satisfying from a database perspective. Remember that the relations in CQ-instances correspond to database tables. But in a typical database setting one does not have control over the data. Also, the tractable classes of relations are typically very small, so one is unlikely to be in a tractable class of CQ-instances of the above type in a database setting. This has lead the database theory community to isolate so-called "islands of tractability" by restricting the queries of CQ-instances. To this end, one assigns a hypergraph $\mathcal{H}$ to the query ϕ,

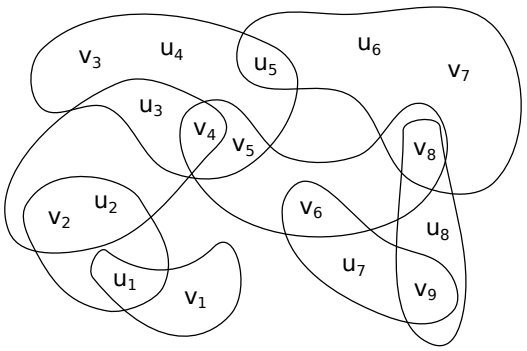

Figure 1: The hypergraph associated to the query ϕ.

in which the variables of ϕ are the vertices and the edges are the variables scopes of the atoms of ϕ. Consider for example the query

$$\begin{aligned}\phi \;:=\; & \exists u_1 \exists u_2 \exists u_3 \exists u_4 \exists u_5 \exists u_6 \exists u_7 \exists u_8 \\ & P_1(v_1, u_1) \wedge P_2(v_2, u_1, u_2) \wedge P_3(v_2, v_4, u_2, u_3) \\ & \wedge P_4(v_3, v_4, v_5, u_3, u_4, u_5) \wedge P_5(v_4, v_5, v_6, v_8) \\ & \wedge P_6(v_7, v_8, u_5, u_6) \wedge P_2(v_6, v_9, u_7) \wedge P_2(v_8, v_9, u_8).\end{aligned}$$

The associated hypergraph is illustrated in Figure 1.

The general approach is to restrict the hypergraphs associated to the queries to isolate tractable subclasses. More formally, for a class $\mathcal{G}$ of hypergraphs, we say that the decision problem CQ on $\mathcal{G}$ is tractable if CQ restricted to instances whose queries have hypergraphs in $\mathcal{G}$ is tractable. For #CQ and #CSP an analogous wording is used. The aim is to find classes $\mathcal{C}$ as big as possible on which CQ, resp. #CQ, are tractable.

The basic observation is that CQ and #CSP are tractable on the class $\mathcal{C}_{tree}$ that consists of all trees. Most research has aimed at extending this result to graphs and hypergraphs that are "nearly" trees. To this end, many different decomposition techniques for graphs and hypergraphs have been proposed. The general idea is to organize vertices and/or edges into clusters and to organize these clusters into a tree, satisfying properties whose exact formulation defines the decomposition technique. The most well-known decomposition technique is certainly the notion tree decompositions of graphs:

A *tree decomposition* of a graph $G = (V, E)$ is a pair $(\mathcal{T}, (\chi_t)_{t \in T})$ where $\mathcal{T} = (T, F)$ is a rooted tree and $\chi_t \subseteq V$ for every $t \in T$ satisfying the following properties:

1. For every $v \in V$ there is a $t \in T$ with $v \in \chi_t$.
2. For every $e \in E$ there is a $t \in T$ such that $e \subseteq \chi_t$.

3. For every $v \in V$ the set $\{t \in T \mid v \in \chi_t\}$ induces a subtree of $\mathcal{T}$.

We define $\max_{t \in T}(|\chi_t|) - 1$ to be the *width* of the tree decomposition $(\mathcal{T}, (\chi_t)_{t \in T})$. The *treewidth* of G is defined as the minimum width over all tree decompositions of G.

The *treewidth* of a hypergraph $\mathcal{H} = (V, E)$ defined to be that of its primal graph, i.e., the graph $\mathcal{H}_P = (V, E')$ with $E' := \{uv \mid \exists e \in E : \{u, v\} \in e\}$.

It turns out that CQ on graphs of bounded treewidth is tractable, but unfortunately, this result is only helpful if the arity of the relations symbols in the considered queries are bounded by a constant. To capture tractable classes of queries with unbounded arity, one has to consider hypergraphs: the study of their primal graphs is not sufficient. To this end, numerous decomposition techniques for hypergraphs were proposed. Most of them mimic treewidth by organizing vertices and edges into clusters and then organizing these clusters in a tree with certain properties. How the clusters are formed and how they are organized in a tree then defines the specific decomposition technique. Let us illustrate this basic idea with the fairly general notion of generalized hypertree decompositions.

A *generalized hypertree decomposition* of a hypergraph $\mathcal{H} = (V, E)$ consists of a triple $(\mathcal{T}, (\lambda_t)_{t \in T}, (\chi_t)_{t \in T})$ where $\mathcal{T} = (T, F)$ is a rooted tree and $\lambda_t \subseteq E$ and $\chi_t \subseteq V$ for every $t \in T$ satisfying the following properties:

1. For every $e \in E$ there is a $t \in T$ such that $e \subseteq \chi_t$.
2. For every $t \in T$ we have $\chi_t \subseteq \bigcup_{e \in \lambda_t} e$.
3. For every $v \in V$ the set $\{t \in T \mid v \in \chi_t\}$ induces a subtree of $\mathcal{T}$.

The *width* of a decomposition $(\mathcal{T}, (\lambda_t)_{t \in T}, (\chi_t)_{t \in T})$ is defined to be $\max_{t \in T}(|\lambda_t|)$. The generalized hypertree width of $\mathcal{H}$ is defined as the minimum width over all generalized hypertree decompositions of $\mathcal{H}$. A generalized hypertree decomposition of the graph from Figure 1 is illustrated in Figure 2.

Given a query ϕ, whose associated hypergraph has generalized hypertreewidth k, CQ on input $(\mathcal{A}, \phi)$ can be solved in time $(|\phi| + |\mathcal{A}|)^{O(k)}$ and thus in polynomial time for fixed k [CD05]. Similar results hold true for many other decomposition techniques (see e.g. [GLS00, CJG08]).

1.1.1 *Structural restrictions for tractable* #CQ

While the decision complexity of conjunctive queries has been studied extensively and huge effort has been invested in finding and understanding more and more general decomposition techniques and associated width measures, the complexity of the associated counting

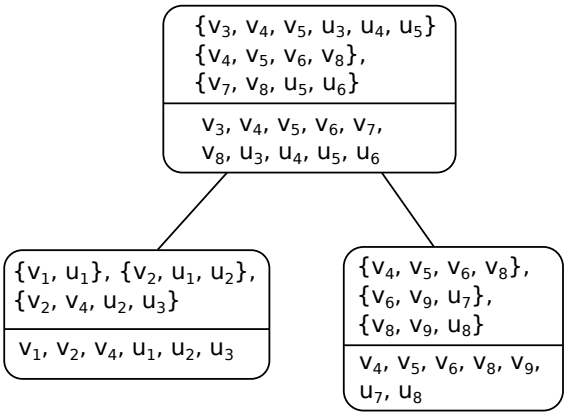

Figure 2: A generalized hypertree decomposition of width 3 for the hypergraph from Figure 8. The boxes are the guarded blocks. In the upper parts the guards are given while the lower parts show the blocks.

problem #CQ was not well understood prior to the research presented in this thesis. This is somewhat surprising, because a counting operator is standard in all practical database query languages like SQL. The complexity of #CSP is comparatively better understood: The general observation here is that structural classes that allow efficient decision in most cases also allow efficient counting for constraint satisfaction, i.e., for quantifier free queries [PS13].

While this is nice, it is not fully satisfying, because quantifiers—which correspond to projections in database theory—are very natural and essential in database queries. While introducing projections does not make any difference for the complexity of CQ, the situation for #CQ is dramatically different. In [PS13] it is shown that even one single existentially quantified variable is enough to make counting answers to CQ-instances #P-hard, even when the associated hypergraph of the query is a tree (which implies width 1 for all commonly considered decomposition techniques). It follows that the decomposition techniques used for CQ are not enough to guarantee tractability for counting.

In Part i of this thesis we will see a way out of this dilemma for counting, by introducing a parameter called *quantified star size* for conjunctive queries. This parameter measures how the free variables are spread in the query. We associate to a query ϕ not only its hypergraph $\mathcal{H} = (V, E)$ as described before, but additionally include the set $S \subseteq V$ of its free variables into our analysis. The *quantified star size* of ϕ is defined as the size of a maximum independent set consisting of vertices from the set S in some specified subhypergraphs of $\mathcal{H}$.

We will see that quantified star size is the appropriate parameter when considering #CQ. On the one hand, bounded quantified star size allows tractable #CQ, when combined with decomposition

techniques such as treewidth or generalized hypertree width. On the other hand, we will prove that unbounded quantified star size yields intractable counting problems (under suitable assumptions from complexity theory, of course). Thus quantified star size is necessary and sufficient for tractable #CQ.

For queries whose arity is bounded by a constant, we will then refine these results. Instead of considering hypergraphs associated to conjunctive queries, we directly consider classes of queries. We call two queries ϕ and ϕ' equivalent, if for all finite structures $\mathcal{A}$ we have $\phi(\mathcal{A}) = \phi'(\mathcal{A})$. Our considerations are based on the fundamental observation made by Chandra and Merlin in a seminal paper [CM77] that conjunctive queries may have much smaller subqueries that are equivalent to them. For example consider the query

$$\phi := \bigwedge_{i,j \in [n]} E(x_i, y_j).$$

It is easy to see that for each structure $\mathcal{A}$ the query result $\phi(\mathcal{A})$ is nonempty if for $\phi' := E(x_1, y_1)$ we have that $\phi'(\mathcal{A})$ is non-empty. So CQ for ϕ and ϕ' are equivalent. Generalizing this observation leads to the notion of the *core* ϕ' of a query ϕ, which is a unique (up to isomorphism) subquery that is equivalent ϕ. As the core ϕ' is equivalent but may be much smaller and structurally simpler, it is of course preferable to work with ϕ' instead of ϕ. Unfortunately, this is in general not possible, because computing cores of conjunctive queries is intractable.

Somewhat surprisingly, cores can still help to guarantee tractability of CQ. Dalmau, Kolaitis and Vardi [DKV02] showed that if ϕ is assumed to have a core of bounded treewidth, then CQ can be solved efficiently—even without computing the core! Grohe [Gro07] showed that this result is optimal: Assuming a widely believed assumption from parameterized complexity, for every class $\mathcal{C}$ of conjunctive queries, we have that CQ restricted to queries from $\mathcal{C}$ is tractable if and only if the cores of $\mathcal{C}$ are of bounded treewidth.

We will analyse the role of cores for #CQ and use them to fully characterize the classes of queries of bounded arity that allow tractable counting of solutions to CQ-instances.

1.2 PART II: UNDERSTANDING ARITHMETIC CIRCUIT CLASSES

Part ii and Part iii of this thesis deal with different aspects of arithmetic circuit complexity, a very classical but at the same time active subarea of complexity theory. The main difference between arithmetic circuit complexity and the more common Boolean circuit complexity is that instead of considering operations on bits, one considers arithmetic operations over fields or sometimes other algebraic structures like rings or algebras. This is motivated by two facts: On the one

hand, when solving computational problems from e.g. linear algebra or algebra, using arithmetic operations directly in the formulation of algorithms instead of translating into the Boolean setting is a very natural abstraction. On the other hand, computation over fields instead of bits allows to use techniques from rich areas of mathematics like linear algebra, representation theory and algebraic geometry. This additional machinery makes showing lower bounds on the complexity of some computational problems more approachable for arithmetic circuits than for other models. This is also the general idea behind the geometric complexity theory program, an approach that is generally seen as one most promising to proving lower bounds (see e.g. [BLMW11, Mul12] for two recent surveys).

An *arithmetic circuit* over a field $\mathbb{F}$ is a labeled directed acyclic graph (DAG) consisting of vertices or gates with indegree or fanin 0 or 2. The gates with fanin 0 are called input gates and are labeled with constants from $\mathbb{F}$ or variables $X_1, X_2, \ldots, X_n$. The gates with fanin 2 are called computation gates and are labeled with $\times$ or $+$. The polynomial computed by an arithmetic circuit is defined in the obvious way: An input gate computes the value of its label, a computation gate computes the product or the sum of its children's values, respectively. We assume that a circuit has only one sink which we call output gate. We say that the polynomial computed by the circuit is the polynomial computed by the output gate. The *size* of an arithmetic circuit is the number of gates. A circuit is called *multiplicatively disjoint* if, for each $\times$-gate, its two input subcircuits are disjoint. Figure 3 gives an example of an arithmetic circuit.

Clearly, all polynomials can be computed by arithmetic circuits, so the basic question of arithmetic circuit complexity is determining how many arithmetic operations are necessary to compute specific polynomials. As usual in complexity theory, the focus is on asymptotic behaviour, so instead of individual polynomials one considers families or sequences of polynomials, e.g. $(\det_n)_{n\in\mathbb{N}}$, where $\det_n = \sum_{\sigma\in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} X_{i\sigma(i)}$ is the determinant of an $(n \times n)$-matrix of variables X_{ij}.

Efficient computation in the arithmetic circuit setting is captured by the complexity class VP, which consists of families of polynomials of polynomial degree that are computed by arithmetic circuits of polynomial size. For example, the family $(\det_n)_{n\in\mathbb{N}}$ lies in VP. In the definition of VP, the condition on the degree is made, because natural polynomial families like the determinant, the permanent or graph polynomials in general have polynomial degree. Thus, the degree condition makes sure that the actual interesting polynomials are captured by the definitions in arithmetic circuit complexity. Unfortunately, the degree bound is a semantic condition that made working with VP cumbersome for many years, until the situation was remedied by Malod and Portier [MP08]. They showed that VP consists

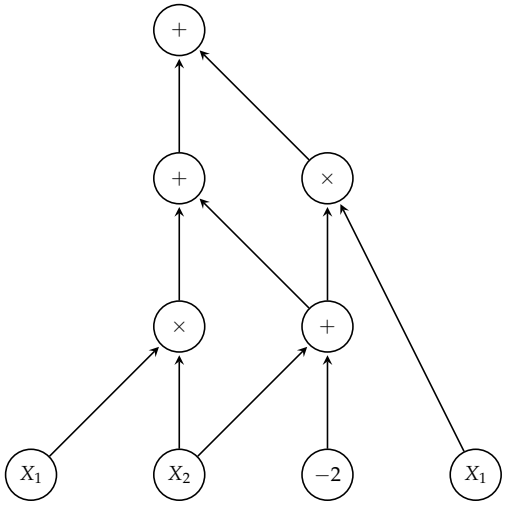

Figure 3: An arithmetic circuit computing the polynomial $X_1X_2 + X_2 - 2 + (X + 2 - 2)X_1$.

exactly of those polynomial families that can be computed by multiplicatively disjoint arithmetic circuits of polynomial size. Since multiplicative disjointness is a syntactical property, this has caused an upswing in the understanding of VP.

An arithmetic circuit is called *skew*, if for each of its multiplication gates, one of its inputs is a variable or a constant. A circuit is called a formula if its underlying graph is a tree. By VP_{ws}, respectively VP_e, we denote the families of polynomials computed by polynomial size skew circuits, respectively formulas[1]. Finally, a family (f_n) of polynomials is in VNP, if there is a family $(g_n) \in \mathsf{VNP}$ and a polynomial p such that $f_n(X) = \sum_{e \in \{0,1\}^{p(n)}} g_n(e, X)$ for all n where X denotes the vector $(X_1, \ldots, X_{q(n)})$ for some polynomial q.

We have

$$\mathsf{VP}_e \subseteq \mathsf{VP}_{ws} \subseteq \mathsf{VP} \subseteq \mathsf{VNP},$$

where all containments are commonly conjectured—but none of them proved—to be strict.

The complexity class VP is, in contrast to its Boolean counterpart P, arguably not very well understood. There are no known natural complete problems and before the work presented in this thesis, there had been very few characterizations of VP that were not slight modifications of the original circuit definition. In my opinion, this lack of different perspectives on VP severely hampered progress on its understanding. Thus the general aim of Part ii to find new natural characterizations of VP and other classes. Let us sketch the results of

1 The "ws" in VP_{ws} stands for "weakly skew". The reason for this notation is that VP_{ws} can also be defined by weakly skew circuits which, in contrast to what their name suggests, are equivalent to skew circuits. Since we will not consider weakly skew circuits in this thesis, we will not introduce them but nevertheless stick to the usual notation VP_{ws} for the complexity class.

the different chapters individually. More detailed introductions can be found in the respective chapters.

1.2.1 *Conjunctive queries and arithmetic circuits*

In Chapter 10 we will apply the findings of Part i to arithmetic circuit complexity: To a word $a = a_1 \dots a_n$ over a domain A one can in a natural way assign a monomial $m(a)$. To do so, we introduce a set of variables $\{X_d \mid d \in A\}$ and define $m(a) := \prod_{i=1}^{n} X_{a_i}$. Now, given a CQ-instance $(\mathcal{A}, \phi)$, this induces a polynomial

$$Q(\mathcal{A}, \phi) := \sum_{a \in \phi(\mathcal{A})} m(a).$$

To $\{0,1\}$-words $e = e_1 \dots e_n$ one can assign another monomial $m'(e) := X_1^{e_1} \dots X_n^{e_n}$ in the variables $X_1 \dots X_n$. Given a CQ-instance $(\phi, \mathcal{A})$ with domain $\{0,1\}$, this yields the polynomial

$$P(\mathcal{A}, \phi) := \sum_{e \in \phi(\mathcal{A})} m'(e).$$

While at first these definitions look quite arbitrary, they in fact give an interesting perspective: When one considers the restrictions on conjunctive queries presented in Part i, it turns out that the resulting polynomials characterize different arithmetic circuit classes like VP. For example, VP can be characterized by the polynomials $Q(\mathcal{A}, \phi)$ of polynomial size families of CQ-instances $(\mathcal{A}_n, \phi_n)$ of bounded quantified star size and bounded treewidth. If we consider families $(\mathcal{A}_n, \phi_n)$ of CQ-instances of bounded *pathwidth* and bounded quantified star size, the resulting polynomials $Q(\mathcal{A}_n, \phi_n)$ characterize the subclass VP_{ws} of VP. Finally, families $(\mathcal{A}_n, \phi_n)$ of polynomial size CQ-instances of bounded generalized hypertree width with the additional condition that every relation in every $\mathcal{A}_n$ is bounded by a constant ℓ, yield a characterization of VP_e. This gives a connection between constraint satisfaction and conjunctives queries and arithmetic circuit classes.

These results were the first characterizations of some arithmetic circuit classes that did not in one way or the other depend on arithmetic circuits themselves. As conjunctive queries are a very flexible tool to encode different combinatorial problems, this shift in perspective may give one a new intuition.

1.2.2 *Graph polynomials on bounded treewidth graphs*

In Chapter 11 we consider graph polynomials of graphs of bounded treewidth, a subject already consider by Courcelle et al. [CMR01] and Flarup et al. [FKL07]. The former authors showed that a huge class of graph polynomials that are hard in general—those definable in monadic second order logic—become tractable on bounded treewidth

graphs. The latter authors then proved that some specific polynomials, in particular the permanent, characterize the class VP_e when restricted to such graphs. Lyaudet [Lya07] then conjectured that all monadic second order definable graph polynomials, that are VNP-complete for a general family of graphs, capture VP_e on bounded treewidth graphs. We show that this conjecture is false by exhibiting graph polynomials that fail to be expressive enough to capture VP_e on the restricted graphs. Despite this, we show that many graph polynomials characterize VP_e on bounded treewidth graphs by using a new notion of reductions tailored specifically to this application.

1.2.3 *Modifying arithmetic branching programs*

Arithmetic branching programs (ABPs) are one of the most widely studied computational models in arithmetic circuit complexity. An ABP is a directed acyclic graph $G = (V, E)$ with an edge weight function $w : E \to \mathbb{F} \cup \{X_1, X_2, \ldots, X_n\}$ and two distinguished vertices $s, t \in V$. The weight of a path $P = e_1 e_2 \ldots e_k$ from s to t in G is $w(P) := \prod_{i=1}^{k} w(e_i)$. The polynomial computed by the ABP is defined as $\sum_P w(P)$ where the sum is over all paths P from s to t in G.

The weights of paths through ABPs can be easily computed by iterated matrix multiplication and thus polynomials computed by polynomial size ABPs can also be computed by polynomial size arithmetic circuits. It is equivalent to the standard conjecture $\mathsf{VP}_{ws} \neq \mathsf{VP}$ that the other direction is not true, i.e., there are families of polynomials that can be computed by polynomial size arithmetic circuits but not by polynomial size ABPs. Nevertheless, ABPs are an important computational model that has contributed greatly to progress in arithmetic circuit complexity (see e.g. [MP08, Koi12]).

In Chapter 12 we will modify ABPs by giving them memory during their computations. We will see that this yields robust new models of computation that characterize different arithmetic circuit classes for different types of memory. In particular, we get a new, natural characterization of VP by branching programs with stacks. This also allows to adapt techniques from the so-called NAuxPDA-characterization of LOGCFL to the arithmetic circuit setting.

1.3 PART III: MONOMIALS IN ARITHMETIC CIRCUITS

In Part iii we take a different perspective on arithmetic circuits. We do not see them as a computational model anymore, but view them as inputs for different computational problems. Arithmetic circuits are a very succinct way of describing polynomials, that can be much more efficient than giving the list of monomials. Unfortunately, this succinctness comes at the price that properties of the computed polynomials become harder to decide. Consider for example the widely

studied polynomial identity testing problem that consists of deciding if a given polynomial f is zero. This problem is trivial if f is given as a list of monomials, but if f is given as an arithmetic circuit, it is not known if $f \equiv 0$ can be decided in deterministic polynomial time. Several other natural decision problems on arithmetic circuits have been considered in the literature, see e.g. [ABKPM09, KP07].

In Part iii we will mainly analyze the complexity of two fundamental problems: The first one, called ZMC for *zero monomial coefficient*, is to decide whether a given monomial m appears in the polynomial given by an arithmetic circuit. The second problem considered is to count the number of monomials in the polynomial computed by a circuit. We show that these two problems on different restricted classes of arithmetic circuits are complete for different levels of the counting hierarchy, a hierarchy of complexity classes that was introduced by Wagner [Wag86] and recently played a prominent role in several papers in arithmetic circuit complexity [Bür09, JS11, KP11]. Although the counting hierarchy was introduced for classifying the complexity of combinatorial problems, there are very few problems known to be complete for its levels, and thus our results can also be seen as a contribution to the study of the counting hierarchy.

1.4 OVERVIEW OVER THE THESIS

Part i is devoted to the counting complexity of conjunctive queries. Chapter 2 gives the necessary preliminaries on conjunctive queries, parameterized complexity, and graph and hypergraph decomposition techniques in a rather condensed form. Chapter 3 presents some basic results on on the complexity of counting solutions to conjunctive queries, introduces the parameter *quantified star size* and states the main results of Chapter 4 and Chapter 5. In Chapter 4 we deal with the complexity of computing the quantified star size of conjunctive queries. In Chapter 5 we consider the impact of quantified star size on the complexity of #CQ. The Chapters 3-5 are a combined and more coherent presentation of the results of [DM11] and [DM13]. The Chapters 6 and 7 deal with conjunctive queries of bounded arity, a case in which a finer understanding of tractable #CQ-instances is possible. The results of Chapter 6 are partly joint work with Arnaud Durand [DM13]. The results of Chapter 7 are unpublished joint work with Hubie Chen. Chapter 8 concludes Part i and outlines some possible directions for future research.

Part ii presents different characterizations of complexity classes commonly considered in arithmetic circuit complexity. We begin with a presentation of the necessary background on arithmetic circuit complexity in Chapter 9. In Chapter 10 we connect the findings of Part i to arithmetic circuit complexity to characterize complexity classes of the arithmetic circuit model by conjunctive queries and constraint satis-

faction problems. This chapter is a largely extended version of the results that appeared in [Men11], some of it depending on the joint work with Arnaud Durand [DM11, DM13]. Chapter 11 deals with the expressivity of graph polynomials on bounded treewidth graphs. Chapter 12 presents the results of [Men13] on arithmetic branching programs with memory. In contrast to Part i, the chapters of Part ii are largely independent from each other and can also be read individually.

The comparatively short Part iii presents joint work with Hervé Fournier and Guillaume Malod [FMM12], which deals with the complexity of some decision problems on arithmetic circuits and the connection to the counting hierarchy.

Part I

COUNTING SOLUTIONS TO CONJUNCTIVE QUERIES

2

PRELIMINARIES

In this chapter we introduce the basic definitions and the notation we will use in Part i of this thesis. We start off with a formal definition of conjunctive queries, introduce some notions from parameterized complexity and then survey the graph and hypergraph decompositions we will consider.

2.1 CONJUNCTIVE QUERIES

In this section we give a very brief introduction into conjunctive queries. Conjunctive queries lie in the intersection of database theory, artificial intelligence and finite model theory, a subarea of logic. Thus terms and notation differ greatly depending on the background of different authors. In this thesis, we mainly follow a notation that is based on logic. Our presentation can only give a very small impression of the basics of database theory and finite model theory. Much more on these rich fields can be found in [AHV95, Lib04].

A *relational vocabulary* is defined to be a set of relation symbols $\tau := \{\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_\ell\}$ where each $\mathcal{R}_i$ has an arity r_i which we denote by $\text{arity}(\mathcal{R}_i)$. A *relational structure* $\mathcal{A}$ over τ is a tuple $(A, \mathcal{R}_1^{\mathcal{A}}, \ldots, \mathcal{R}_\ell^{\mathcal{A}})$ where A is a set called the *domain* of $\mathcal{A}$ and $\mathcal{R}_i^{\mathcal{A}} \subseteq A^{r_i}$ is a relation of arity r_i. We call a structure $\mathcal{A}' = (A', \mathcal{R}_1^{\mathcal{A}'}, \ldots, \mathcal{R}_\ell^{\mathcal{A}'})$ over τ a *substructure* of $\mathcal{A}$ if $A' \subseteq A$ and $\mathcal{R}_i^{\mathcal{A}'} \subseteq \mathcal{R}_i^{\mathcal{A}}$ for $i \in [\ell]$. We call $\mathcal{A}'$ a *proper substructure* of $\mathcal{A}$ if $\mathcal{A}'$ is a substructure of $\mathcal{A}$ and $\mathcal{A}' \neq \mathcal{A}$.

All vocabularies and structures in this thesis are assumed to be relational, so we mostly drop the adjective "relational" and only speak of vocabularies and structures. Furthermore, all structures in this thesis have finite domains. This implies that all relations and all structures are finite as well. We denote structures by calligraphic letters, e.g. $\mathcal{A}, \mathcal{B}, \ldots$. For the corresponding domains we use the corresponding roman letters, i.e., A is the domain of $\mathcal{A}$, B the domain of $\mathcal{B}$ and so on.

Example 2.1.1. One type of finite structures appearing several times in this thesis are graphs. Here the vocabulary consists of only one binary symbol $\mathcal{E}$. A graph $G = (V, E)$ is then a structure $\mathcal{A} = (A, \mathcal{E}^{\mathcal{A}})$ where domain $A := V$ is the vertex set of G and $\mathcal{E}^{\mathcal{A}} := E$ is the edge relation of G. The graph G is undirected if and only if $\mathcal{E}^{\mathcal{A}}$ is symmetric. Otherwise G is directed. Note that there are other ways to encode graphs as structures which we will discuss in Chapter 11.□

If $\mathcal{R}$ is a relation symbol of arity r and $(z_{i_1}, \ldots, z_{i_r})$ is a sequence of (not necessarily distinct) variables, then the expression $\mathcal{R}(z_{i_1}, \ldots, z_{i_r})$

is called an *atomic formula* or *atom* for short. Furthermore, the *scope* $\mathsf{var}(\mathcal{R}(z_{i_1}, \ldots, z_{i_r}))$ of the atom $\mathcal{R}(z_{i_1}, \ldots, z_{i_r})$ is defined as the set of variables appearing in $(z_{i_1}, \ldots, z_{i_r})$.

A *quantifier free conjunctive query* ϕ over a vocabulary τ is a logical formula of the form

$$\phi = \mathcal{R}_{i_1}(\bar{z}_1) \wedge \ldots \wedge \mathcal{R}_{i_s}(\bar{z}_s),$$

where $\mathcal{R}_{i_j}(\bar{z}_1)$ are atomic formulas and the $\mathcal{R}_{i_j}$ are relation symbols of τ of the matching arity. We denote the set of variables of ϕ by $\mathsf{var}(\phi) := \bigcup_{j \in [s]} \mathsf{var}(\mathcal{R}_{i_j}(\bar{z}_j))$.

A *conjunctive query* ϕ over τ is a formula $\phi = \exists z_{i_1} \ldots \exists z_{i_t} \phi'$ where ϕ' is a quantifier free conjunctive query over τ and $x_{i_j} \in \mathsf{var}(\phi)$ for all $j \in t$. The x_{i_j} are called *quantified variables*. The set of variables of ϕ is defined as $\mathsf{var}(\phi) := \mathsf{var}(\phi')$. The set of *free variables* of ϕ is defined as $\mathsf{free}(\phi) := \mathsf{var}(\phi) \setminus \{x_{i_1}, \ldots, x_{i_t}\}$. We sometimes write $\phi(x)$ where $x = \mathsf{free}(\phi)$, to stress the free variables. The set of all atoms of ϕ is denoted by $\mathsf{atom}(\phi)$. All queries in this thesis will be conjunctive, so we often drop the adjective "conjunctive".

A *conjunctive query instance* over τ, short CQ-instance, is a pair $\Phi = (\mathcal{A}, \phi)$ where $\mathcal{A}$ is a finite structure over τ and ϕ is a conjunctive query over τ. Φ is called quantifier free if the query ϕ is quantifier free. We call a CQ-instance $\Phi = (\mathcal{A}, \phi)$ *binary* if all atoms of ϕ have arity at most 2.

Let $\Phi = (\mathcal{A}, \phi)$ be a quantifier free CQ-instance. An *assignment* to Φ is a mapping $a : \mathsf{var}(\phi) \to A$. A partial assignment to Φ is a mapping $a : X \to A$ for a subset X of $\mathsf{var}(\phi)$. Let $a : X \to A$ and $b : Y \to A$ be two partial assignments. We call a and b *compatible*, in symbols $a \sim b$, if they agree on their common variables, i.e., for all $x \in X \cap Y$ we have $a(x) = b(x)$. Let $\mathcal{R}(z_{i_1}, \ldots, z_{i_r})$ be an atom of ϕ. We say that a *satisfies* $\mathcal{R}(z_{i_1}, \ldots, z_{i_r})$ if $(a(z_{i_1}), \ldots, a(z_{i_r})) \in \mathcal{R}^{\mathcal{A}}$. We say that a satisfies Φ if it satisfies all of its atoms. In this case we write $(\mathcal{A}, a) \models \phi$.

An assignment to a general, not necessarily quantifer free, CQ-instance $\Phi = (\mathcal{A}, \phi)$ is a mapping $a : \mathsf{free}(\phi) \to A$. An assignment a will alternatively also be seen as a tuple of dimension $|\mathsf{free}(\phi)|$ indexed by the variables $\mathsf{free}(\phi)$. Consequently, relations will also be seen either as sets of tuples or as list of assignments. An assignment $a : \mathsf{free}(\phi) \to A$ satisfies Φ if there is an assignment $a' : \mathsf{var}(\phi) \to A$ with $a \sim a'$ such that the quantifier free query instance $(\mathcal{A}, \phi')$, where we get ϕ' by deleting all quantifiers from ϕ, is satisfied by a'. Again we write $(\mathcal{A}, a) \models \phi$. Observe that a' is in general not unique.

The *query result* $\phi(\mathcal{A})$ of a CQ-instance $\Phi = (\mathcal{A}, \phi)$ is defined as

$$\phi(\mathcal{A}) := \{a \mid (\mathcal{A}, a) \models \phi(x)\}.$$

The elements of the query result are called *solutions* of the query instance or *satisfying assignments* or *query answers*. Observe that the query result $\phi(\mathcal{A})$ is a relation and that for an atomic subformula ϕ' of ϕ with relation symbol $\mathcal{R}$ we have $\phi'(\mathcal{A}) := \mathcal{R}^{\mathcal{A}}$.

Example 2.1.2. We consider again graphs as discussed in Example 2.1.1. Consider the query $\bigwedge_{i,j\in[k],i\neq j} \mathcal{E}(v_i, v_j)$. If $\mathcal{A}$ is the structure of a graph G, then $\phi(\mathcal{A})$ contains all maps $a : \{v_1, \dots, v_k\} \to V$ such that the image of a is the vertex set of a k-clique of G.

For $\phi = \exists w \mathcal{E}(v, w) \wedge \mathcal{E}(u, w)$ the query result $\phi(\mathcal{A})$ contains the (ordered) pairs of vertices (v_1, v_2) that have a common neighbor w.□

We call two instances $\Phi = (\mathcal{A}, \phi), \Phi' = (\mathcal{A}', \phi')$ *solution equivalent*, if $\mathsf{free}(\phi) = \mathsf{free}(\phi')$ and $\phi(\mathcal{A}) = \phi'(\mathcal{A}')$.

It is often helpful to interpret relations as tables (indeed, they represent database tables in database theory). The rows of the tables are tuples of the relation and the columns are indexed by the variables of the atoms that the relations belong to.

Throughout this thesis we will make use of the following classical database operations on relations.

Definition 2.1.3. *Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two relations indexed by the variables $(x_1, \dots, x_n, y_1, \dots, y_m)$ and $(y_1, \dots, y_m, z_1, \dots, z_\ell)$, respectively, with $\{x_1, \dots, x_n\} \cap \{z_1, \dots, z_\ell\} = \emptyset$. The* natural join *of $\mathcal{R}_1$ and $\mathcal{R}_2$ is*

$$\mathcal{R}_1 \bowtie \mathcal{R}_2 := \{(a_1, \dots, a_n, b_1, \dots, b_m, c_1, \dots, c_\ell) \mid (a_1, \dots, a_n, b_1, \dots, b_m) \in \mathcal{R}_1, (b_1, \dots, b_m, c_1, \dots, c_\ell) \in \mathcal{R}_2\}$$

. □

Let a be an assignment and $Y \subseteq \mathsf{free}(\phi)$. By $a|_Y$ we denote the restriction of a onto Y.

Definition 2.1.4. *Let $\mathcal{R}$ be a relation indexed by the set X of variables where each $a \in \mathcal{R}$ is interpreted as an assignment $a : X \to A$. Let $Y \subseteq X$, then the* projection *of $\mathcal{R}$ onto Y, denoted by $\pi_Y(\mathcal{R})$ is defined as*

$$\pi_Y := \{a|_Y \mid a \in \mathcal{R}\}.$$

In the tuple view this corresponds to deleting all entries from the tuples of $\mathcal{R}$ that correspond to variables not *in Y and then deleting duplicate tuples from the relation.* □

Definition 2.1.5. *Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two relations. Let $\mathcal{R}_1$ be indexed by the variable set Y. The* semi-join *of $\mathcal{R}_1$ and $\mathcal{R}_2$ is defined as*

$$\mathcal{R}_1 \ltimes \mathcal{R}_2 := \pi_Y(\mathcal{R}_1 \bowtie \mathcal{R}_2).$$ □

2.1.1 *Model of computation and encoding of instances*

The underlying model of computation for our algorithms will be the RAM model with unit costs. We assume the relations of a finite structure $\mathcal{A}$ to be encoded by listing their tuples. Apart from this convention we will not specify an encoding but only give estimates on its size in O-notation that will be satisfied by any reasonable encoding.

Let $\mathcal{A}$ be a structure over a vocabulary τ. For a relation $\mathcal{R}^{\mathcal{A}}$ let $|\mathcal{R}^{\mathcal{A}}|$ denote the cardinality of $\mathcal{R}^{\mathcal{A}}$. Then we define for $\mathcal{R}^{\mathcal{A}}$

$$\|\mathcal{R}^{\mathcal{A}}\| := \operatorname{arity}(\mathcal{R}) \cdot |\mathcal{R}^{\mathcal{A}}|.$$

For the vocabulary τ let $|\tau|$ be the number of predicate symbols. Finally, let $|A|$ be the cardinality of a domain A. Then for a structure $\mathcal{A}$ over the vocabulary τ with domain A we define

$$\|\mathcal{A}\| := |\tau| + |A| + \sum_{\mathcal{R} \in \tau} \|\mathcal{R}^{\mathcal{A}}\|.$$

Furthermore, we define for a conjunctive query

$$|\phi| := \sum_{\substack{\phi' \in \mathsf{atom}(\phi), \\ \mathcal{R} \text{ relation symbol of } \phi'}} \operatorname{arity}(\mathcal{R}).$$

Finally, for a CQ-instance $\Phi = (\mathcal{A}, \phi)$ we define

$$\|\Phi\| := |\phi| + \|\mathcal{A}\|.$$

Note that for any reasonable encoding, up to constant factors, $\|\mathcal{A}\|$ is the size of an encoding of $\mathcal{A}$, $|\phi|$ is the size of an encoding of ϕ and $\|\Phi\|$ is the size of an encoding of Φ. For a detailed discussion and justification of these conventions see [FFG02, Section 2.3].

The following lemma states that the basic database operations we considered above can be performed efficiently.

Lemma 2.1.6. *Given relations $\mathcal{R}_1$ and $\mathcal{R}_2$ and $Y \subseteq \mathsf{var}(\mathcal{R})$, one can compute*

- $\mathcal{R}_1 \bowtie \mathcal{R}_2$ *in time*
$$O(\|\mathcal{R}_1\| \log(\|\mathcal{R}_1\|) + \|\mathcal{R}_2\| \log(\|\mathcal{R}_2\|) + \|\mathcal{R}_1\|\|\mathcal{R}_2\|),$$
- $\pi_Y(\mathcal{R}_1)$ *in time*
$$O(\|\mathcal{R}_1\| \log(\|\mathcal{R}_1\|)),$$
- $\mathcal{R}_1 \ltimes \mathcal{R}_2$ *in time*
$$O(\|\mathcal{R}_1\| \log(\|\mathcal{R}_1\|) + \|\mathcal{R}_2\| \log(\|\mathcal{R}_2\|)).$$

Proof. To compute $\mathcal{R}_1 \bowtie \mathcal{R}_2$, first sort both relations lexicographically by the entries of the rows indexed by $\mathsf{var}(\mathcal{R}_1) \cap \mathsf{var}(\mathcal{R}_2)$ in time $O(\|\mathcal{R}_1\| \log(\|\mathcal{R}_1\|) + \|\mathcal{R}_2\| \log(\|\mathcal{R}_2\|))$. Then traverse both ordered relations in parallel to construct the up to $\|\mathcal{R}_1\| \cdot \|\mathcal{R}_2\|$ tuples of $\mathcal{R}_1 \bowtie \mathcal{R}_2$.

To compute $\pi_Y(\mathcal{R}_1)$, sort $\mathcal{R}_1$ lexicographically by the columns indexed by Y in time $O(\|\mathcal{R}_1\| \log(\|\mathcal{R}_1\|))$. Then traverse the ordered relation, saving for each tuple t the projection $\pi_Y(\mathcal{R}_1)$ skipping potential double occurences.

To compute $\mathcal{R}_1 \ltimes \mathcal{R}_2$, we again sort both relations lexicographically by the entries of the rows indexed by $\mathsf{var}(\mathcal{R}_1) \cap \mathsf{var}(\mathcal{R}_2)$ in time $O(\|\mathcal{R}_1\| \log(\|\mathcal{R}_1\|) + \|\mathcal{R}_2\| \log(\|\mathcal{R}_2\|))$. Then traverse both ordered relations in parallel and save the tuples $t \in \mathcal{R}_1$ that are compatible with a tuple in $\mathcal{R}_2$. ■

We will use Lemma 2.1.6 throughout this thesis, most of the time without explicitly referencing it. A finer analysis with slightly better runtime bounds can be found in the appendix of [FFG02].

2.1.2 *Query problems*

The basic computational question on CQ-instances is the *Conjunctive query answering problem*.

CONJUNCTIVE QUERY ANSWERING PROBLEM
Input: a conjunctive query ϕ and a structure $\mathcal{A}$.
Problem: Compute $\phi(A)$.

Clearly, $\|\phi(\mathcal{A})\|$ can be exponential in $\|\Phi\|$ and thus we cannot have a polynomial time algorithm. Thus much research has concentrated on the *Boolean conjunctive query problem*.

CQ
Input: a conjunctive query ϕ and a structure $\mathcal{A}$.
Problem: Decide if $\phi(A)$ is empty or not.

The main focus in this thesis will be on the associated counting problem #CQ.

#CQ
Input: a conjunctive query ϕ and a structure $\mathcal{A}$.
Problem: Compute $\|\phi(A)\|$.

When the queries of CQ and #CQ are assumed to be quantifier free, the resulting problems are also called *constraint satisfaction problems* CSP, resp. #CSP, and have attracted huge interest in the artificial intelligence community. It is easy to see that CQ and CSP are actually the same problem, an observation first made by Kolaitis and Vardi

[KV00] that has lead to a very fruitful exchange of results and techniques between the database and artificial intelligence communities.

Note that all results in this thesis are for counting with set semantics, i.e., we count each solution only once. Counting for bag semantics in which multiple occurences of identical tuples are counted is considerable simpler to analyze has already been essentially solved in [PS13].

2.2 PARAMETERIZED COMPLEXITY

This section is a very short introduction to some notions from parameterized complexity used in the remainder of this thesis. For more details see [FG06].

A *parameterized decision problem* over an alphabet Σ is a language $L \subseteq \Sigma^*$ together with a computable parameterization $\kappa : \Sigma^* \to \mathbb{N}$. The problem (L, κ) is said to be *fixed-parameter tractable*, or $(L, \kappa) \in \mathsf{FPT}$, if there is a computable function $f : \mathbb{N} \to \mathbb{N}$ such that there is an algorithm that decides for $x \in \Sigma^*$ in time $f(\kappa(x))|x|^{O(1)}$ if x is in L.

Let (L, κ) and (L', κ') be two parameterized decision problems over the alphabets Σ, resp. Π. A *parameterized many-one reduction* from (L, κ) to (L', κ') is a function $r : \Sigma^* \to \Pi^*$ such that for all $x \in \Sigma^*$:

- $x \in L \Leftrightarrow r(x) \in L'$,
- $r(x)$ can be computed in time $f(\kappa(x))|x|^c$ for a computable function f and a constant c, and
- $\kappa'(r(x)) \leq g(\kappa(x))$ for a computable function g.

It is easy to see that FPT is closed under parameterized many-one reductions.

Let p-Clique be the following parameterized problem.

p-Clique
Input: a graph G and $k \in \mathbb{N}$.
Parameter: k.
Problem: Decide if G has a clique of size k.

Here the parameterization κ is simply defined by $\kappa(G, k) := k$. The class W[1] consists of all parameterized problems that are parameterized many-one reducible to p-Clique. A problem (L, κ) is called W[1]-*hard*, if there is a parameterized many-one reduction from p-Clique to (L, κ).

It is widely believed that $\mathsf{FPT} \neq \mathsf{W}[1]$ and thus in particular p-Clique and all W[1]-hard problems are not fixed-parameter tractable.

Parameterized counting complexity theory is developed similarly to decision complexity. A *parameterized counting problem* is a function $F : \Sigma^* \times \mathbb{N} \to \mathbb{N}$, for an alphabet Σ. Let $(x, k) \in \Sigma^* \times \mathbb{N}$, then we call x the input of F and k the parameter. A parameterized counting

problem F is *fixed-parameter tractable*, or $F \in \mathsf{FPT}$, if there is an algorithm computing $F(x,k)$ in time $f(k) \cdot |x|^c$ for a computable function $f : \mathbb{N} \to \mathbb{N}$ and a constant $c \in \mathbb{N}$.

Let $F : \Sigma^* \times \mathbb{N} \to \mathbb{N}$ and $G : \Pi^* \times \mathbb{N} \to \mathbb{N}$ be two parameterized counting problems. A *parameterized parsimonious reduction* from F to G is an algorithm that computes, for every instance (x,k) of F, an instance (y,l) of G in time $f(k) \cdot |x|^c$ such that $l \leq g(k)$ and $F(x,k) = G(y,l)$ for computable functions $f,g : \mathbb{N} \to \mathbb{N}$ and a constant $c \in \mathbb{N}$. A *parameterized T-reduction* from F to G is an algorithm with an oracle for G that solves any instance (x,k) of F in time $f(k) \cdot |x|^c$ in such a way that for all oracle queries the instances (y,l) satisfy $l \leq g(k)$ for computable functions f,g and a constant $c \in \mathbb{N}$.

Let p-#Clique be the following problem.

p-#Clique
Input: a graph G and $k \in \mathbb{N}$.
Parameter: k.
Problem: Compute the number of cliques of size k in G.

A parameterized problem F is in #W[1] if there is a parameterized parsimonious reduction from F to p-#Clique[1]. F is #W[1]*-hard*, if there is a parameterized T-reduction from p-#Clique to F. As usual, F is #W[1]-complete if it is in #W[1] and hard for it, too.

Again, it is widely believed that there are problems in #W[1] (in particular the #W[1]-complete problems) that are not fixed-parameter tractable. Thus, from showing that a problem F is #W[1]-hard it follows that F can be assumed to be not fixed-parameter tractable.

We will mainly deal with two parameterized problems that are versions of CQ and #CQ parameterized by the size of the input query. This parameterization is justified by the origins from database theory. In a typical database application the query is usually far smaller than the database, so it makes sense to use the size of the query as a parameter.

p-CQ
Input: a conjunctive query ϕ and a structure $\mathcal{A}$.
Parameter: $\lvert\phi\rvert$
Problem: Decide if $\phi(A)$ is empty or not.

1 Let us remark that Thurley [Thu06] gives good arguments for defining #W[1] not with parsimonious reductions. He instead defines #W[1] with parameterized T-reductions with only one oracle call. We keep the definition of [FG04, FG06], because we will show no #W[1] upper bounds and thus can avoid these subtleties. We remark though that finding the right reduction notions for counting problems is notoriously tricky to get right (see e.g. [KPZ99, DHK05]).

p-#CQ **Input:** a conjunctive query ϕ and a structure $\mathcal{A}$. **Parameter:** $\lvert\phi\rvert$ **Problem:** Compute $\lvert\phi(A)\rvert$.

2.3 GRAPH AND HYPERGRAPH DECOMPOSITIONS

In this section we will introduce several decomposition techniques for graphs and hypergraphs. We treat tree decompositions independently from the other decompositions, because we will use them again in Part ii and will thus introduce more details on them.

2.3.1 *Treewidth*

We first present some basic facts on treewidth. All proofs can be found in the survey by Bodlaender [Bod98] and the references therein.

Unless stated otherwise all graphs in this thesis are nonempty, finite, undirected and simple, i.e., they have no parallel edges or loops. In contrast, trees are always assumed to be rooted and thus directed.

The treewidth of a graph G is a measure of how similar G is to a tree. There are several equivalent definitions for treewidth of which we will first present the one by Robertson and Seymour [RS86].

Definition 2.3.1. *A* tree decomposition *of a graph $G = (V, E)$ is a pair $(\mathcal{T}, (\chi_t)_{t\in T})$ where $\mathcal{T} = (T, F)$ is a rooted tree and $\chi_t \subseteq V$ for every $t \in T$ satisfying the following properties:*

1. *For every $v \in V$ there is a $t \in T$ with $v \in \chi_t$.*
2. *For every $e \in E$ there is a $t \in T$ such that $e \subseteq \chi_t$.*
3. *For every $v \in V$ the set $\{t \in T \mid v \in \chi_t\}$ induces a subtree of $\mathcal{T}$.*

The third property is called the connectedness condition. *The sets χ_t are called* blocks *or* bags *of the decomposition.*

We call $\max_{t\in T}(\lvert\chi_t\rvert) - 1$ *the* width *of the tree composition $(\mathcal{T}, (\chi_t)_{t\in T})$. The* treewidth $\operatorname{tw}(G)$ *of G is the minimum width over all tree decompositions of G.* □

To ease notation we sometimes identify a vertex $t \in T$ with the corresponding bag χ_t.

We remark that the class of graphs of treewidth 1 consists exactly of all forests, i.e., the graphs that have trees as their connected components. In particular, trees have treewidth 1.

Example 2.3.2. Figure 5 shows a tree decomposition of width 2 of the graph G from Figure 4. Since G is not a forests, it follows that the treewidth of G is 2. □

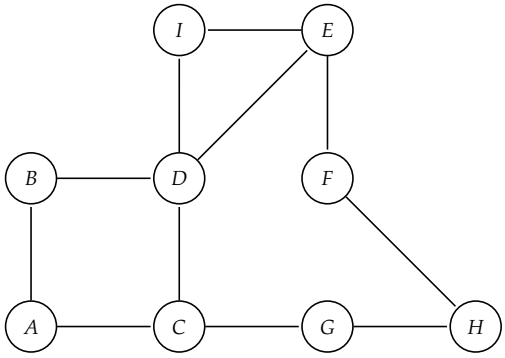

Figure 4: The graph G of Example 2.3.2 and Example 2.3.8.

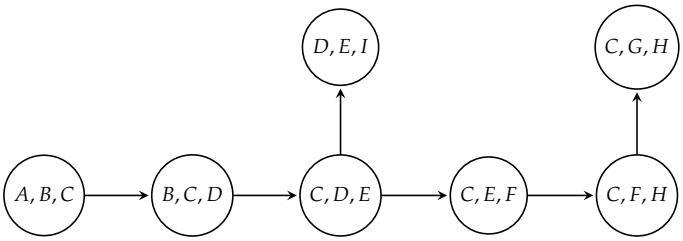

Figure 5: A tree decomposition of width 2 of the graph G of Figure 4.

Given a graph G and an integer k, it is NP-complete to decide if G has treewidth G at most k, but if we take k as a parameter the problem becomes fixed-parameter tractable.

Theorem 2.3.3 ([Bod93]). *There is a polynomial p and an algorithm that, given a graph $G = (V, E)$, computes a tree decomposition of G of width $k := \mathrm{tw}(G)$ in time at most $2^{p(k)}|V|$.*

We will use the following folklore results. A proof can e.g. be found in [Die05, Chapter 12]

Lemma 2.3.4. *Let $G = (V, E)$ be a graph, $C \subseteq V$ a clique in G and $(\mathcal{T}, (\chi_t)_{t \in T})$ a tree decomposition of G. Then there is a bag χ_t such that $C \subseteq \chi_t$.*

Remark 2.3.5. Note that from Lemma 2.3.4 we get that the complete graph K_n on n vertices has treewidth $n - 1$ which is the maximum treewidth any graph on n vertices can have. The corresponding tree decomposition puts all vertices of K_n into a single bag. □

We will also use the following result whose proof can be found e.g. in [FG06, Chapter 11].

Lemma 2.3.6. *Every graph G of treewidth at most k has a vertex of degree at most k.*

We will also use an alternative definition of treewidth by so-called elimination orders.

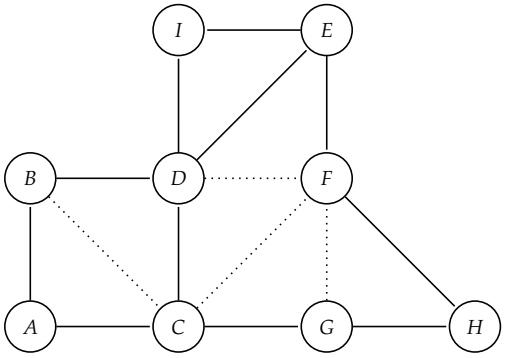

Figure 6: The fill-in graph G_π of Example 2.3.8. The edges added during the construction of G_π from G are represented as dotted lines.

Definition 2.3.7. *Let $G = (V, E)$ be a graph with $|V| = n$. A bijection $\pi : V \to [n]$ is called an* elimination order. *We say that u is higher-numbered than v with respect to π if $\pi(u) > \pi(v)$. The* fill-in graph *G_π of G with respect to π is constructed iteratively: Starting from G, for $i = 1, \ldots, |V|$ we add an edge between all pairs u, w of neighbors of $\pi^{-1}(i)$ that are higher-numbered than $\pi^{-1}(i)$.*

The width *of π is the minimum integer k such that in G_π each vertex $v \in V$ has at most k higher-numbered neighbors.*

The elimination width elim-width(G) *of G is the minimum width over all elimination orders of G.* □

Example 2.3.8. We consider again the graph G of Figure 4. The sequence $A, B, I, H, G, C, D, E, F$ defines an elimination order π in the obvious way. The fill-in graph G_π is shown in Figure 6. The width of π is 2. □

Elimination orders give the following characterization of treewidth which appears to be folklore. A proof can be found e.g. in [Bod98].

Lemma 2.3.9. *For every graph G we have* elim-width$(G) = \mathrm{tw}(G)$.

Finally, we give a lemma that lets us construct small depth tree decompositions of binary trees.

Lemma 2.3.10 ([Bod88]). *For every binary tree $G = (V, E)$ one can in polynomial time construct a tree decomposition $(\mathcal{T}, (\chi_t)_{t \in T})$ of width at most 3 such that the depth of $\mathcal{T}$ is $O(\log(|V|))$ and $\mathcal{T}$ is binary.*

In Chapter 10 we will also consider path decompositions, a restricted version of tree decompositions.

Definition 2.3.11. *A tree decomposition $(\mathcal{T}, (\chi_t)_{t \in T})$ is called a* path decomposition *if $\mathcal{T}$ is a path.*

The pathwidth $\mathrm{pw}(G)$ *of a graph is defined as the minimum width over all path decompositions of G.* □

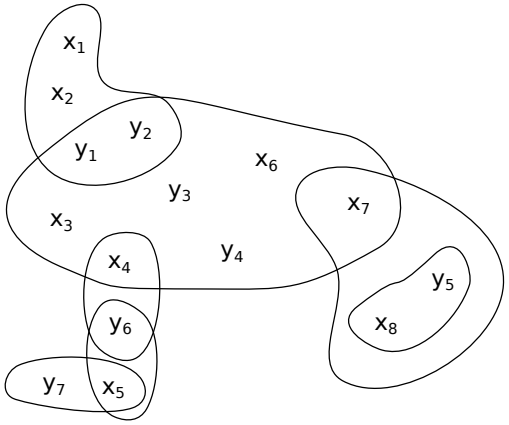

Figure 7: The hypergraph associated to the query from Example 2.3.12.

Note that for every graph G we have $\mathrm{tw}(G) \leq \mathrm{pw}(G)$. Paths have pathwidth 1, but contrary to what one might guess graphs of pathwidth 1 do *not* have to be disjoint unions of paths. In fact the star $G = (V, E)$, $V = (z, v_1, \ldots, v_n)$, $E = \{zv_i \mid i \in [n]\}$ has pathwidth 1. A path decomposition of G is $(\mathcal{P}, (\chi_t)_{t \in T})$ where $\mathcal{P}$ is a path with the vertices $u_1, \ldots, u_n$ and $\chi_{u_i} = \{z, v_i\}$.

2.3.2 *Hypergraph decomposition techniques*

In this section we present some well known hypergraph decompositions techniques. For more details and more decomposition techniques see e.g. [CJGo8, GLSoo, Miko8].

A (finite) hypergraph $\mathcal{H}$ is a pair (V, E) where V is a finite set and $E \subseteq \mathcal{P}(V)$. The *arity* of $\mathcal{H}$ is $\max_{e \in E} |e|$. We associate a hypergraph $\mathcal{H} = \mathcal{H}_\phi = (V, E)$ to a query ϕ by setting $V := \mathsf{var}(\phi)$ and $E := \{\mathsf{var}(\phi_t) \mid \phi_t \in \mathsf{atom}(\phi)\}$.

Example 2.3.12. The query

$$\begin{aligned}\phi(y_1, \ldots y_7) := \exists x_1 \ldots \exists x_8 R(x_1, x_2, y_1, y_2) \wedge T(x_8, y_5) \\ \wedge S(x_3, x_4, x_6, x_7, y_1, y_2, y_3, y_4) \wedge T(x_4, y_6) \\ \wedge T(x_5, y_6) \wedge T(x_5, y_7) \wedge P(x_7, x_8, y_5)\end{aligned}$$

□

has the associated hypergraph illustrated in Figure 7.

Example 2.3.13. Consider the query

$$\begin{aligned}\phi \;:=\; & \exists u_1 \exists u_2 \exists u_3 \exists u_4 \exists u_5 \exists u_6 \exists u_7 \exists u_8 \\ & P_1(v_1, u_1) \wedge P_2(v_2, u_1, u_2) \wedge P_3(v_2, v_4, u_2, u_3) \\ & \wedge P_4(v_3, v_4, v_5, u_3, u_4, u_5) \wedge P_5(v_4, v_5, v_6, v_8) \\ & \wedge P_6(v_7, v_8, u_5, u_6) \wedge P_2(v_6, v_9, u_7) \wedge P_2(v_8, v_9, u_8)\end{aligned}$$

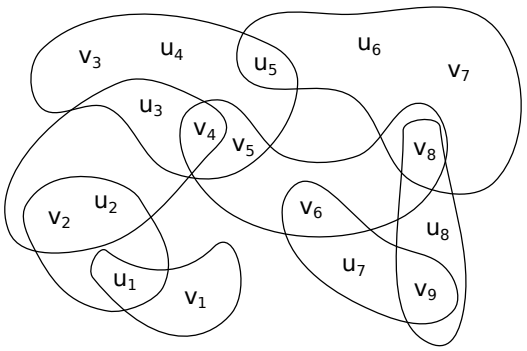

Figure 8: The hypergraph associated to the query ϕ of Example 2.3.13.

The associated hypergraph is illustrated in Figure 8. □

For all decomposition techniques defined below, the width of a CQ-instance $\Phi = (\mathcal{A}, \phi)$ is defined to be the width of the hypergraph associated to ϕ.

The simplest idea to generalize treewidth to hypergraphs is considering primal graphs.

Definition 2.3.14. *The* primal graph *of a hypergraph $\mathcal{H} = (V, E)$ is defined as the graph $\mathcal{H}_P = (V, E_p)$ with*

$$E_p := \{uv \in \binom{V}{2} \mid \exists e \in E : u, v \in E\}.$$

□

Definition 2.3.15. *The* treewidth *of a hypergraph $\mathcal{H}$ is the treewidth of its primal graph $\mathcal{H}_P$.* □

By Lemma 2.3.4, classes of hypergraphs that have unbounded edge size are of unbounded treewidth even when the hypergraphs are intuitively very simple. Consequently, treewidth is, for some considerations on hypergraphs, not the right measure of the complexity of a hypergraph. Thus research focussed on finding decompositions that work with hypergraphs directly and not with their primal graphs. The base class of hypergraphs that roughly corresponds to trees in the setting of treewidth are *acyclic hypergraphs* which are defined with the help of *join trees* which organize the edges of a hypergraph in a tree with a connectivity condition similar to that for treewidth.

Definition 2.3.16. *A* join tree *of a hypergraph $\mathcal{H} = (V, E)$ is a pair $(\mathcal{T}, (\lambda_t)_{t \in T})$ where $\mathcal{T} = (T, F)$ is a rooted tree and for each $t \in T$ we have $\lambda_t \in E$ such that*

- *for each $e \in E$ there is a $t \in T$ such that $\lambda_t = e$,*

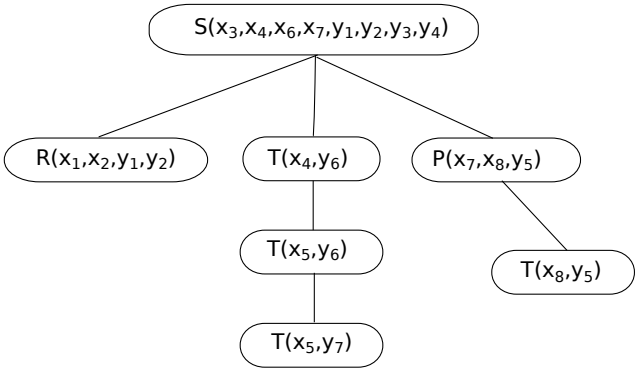

Figure 9: A join tree for the hypergraph of Figure 7.

- *For each $v \in V$ the set $\{t \in T \mid v \in \lambda_t\}$ induces a subtree of T.*

A hypergraph is called acyclic *if it has a join tree.* □

When there is no ambiguity, we often identify vertices $t \in T$ with their edges λ_t.

Lemma 2.3.17 ([Yan81]). *There is a polynomial time algorithm that, given a hypergraph $\mathcal{H}$, decides if $\mathcal{H}$ is acyclic. Moreover, if $\mathcal{H}$ is acyclic the algorithm computes a join tree of $\mathcal{H}$.*

A conjunctive query ϕ is called acyclic if its associated hypergraph is acyclic.

Example 2.3.18. The query $\phi(y_1, \dots, y_7)$ that we considered in Example 2.3.12 is acyclic. Its join tree is depicted in Figure 9. We have $\mathsf{free}(\varphi) = \{y_1, \dots y_7\}$ and $\mathsf{var}(\varphi) = \{x_1, \dots, x_8, y_1, \dots y_7\}$.

The Boolean *acyclic conjunctive query* problem denoted ACQ is the Boolean conjunctive query problem restricted to acyclic instances. We denote by #ACQ the corresponding counting problem.

Acyclic conjunctive queries play an important role in database theory, because of the following result by Yannakakis [Yan81].

Theorem 2.3.19 ([Yan81]). ACQ *can be solved in polynomial time.*

To give the reader a first impression of how decompositions are used algorithmically, we present a proof.

Proof of Theorem 2.3.19. Let $\Phi = (\mathcal{A}, \phi)$ be an ACQ-instance with associated acyclic hypergraph $\mathcal{H}$. W.l.o.g. we assume that ϕ is quantifier free. Let $(\mathcal{T}, (\lambda_t)_{t \in T})$ be a join tree of $\mathcal{H}$ computed with Lemma 2.3.17.

In a slight abuse of notation we identify the λ_t with the atoms in ϕ they represent and denote by $\lambda_t(\mathcal{A})$ the relation of the atom λ_t. For each $t \in T$ we define a relation A_t inductively: If t is a leaf we set

$A_t := \lambda_t(\mathcal{A})$. If t has children $t_1, \dots t_k$ we set (see Definition 2.1.5 for the definition of the semi-join $\ltimes$)

$$A_t := (\dots((\lambda_t(\mathcal{A}) \ltimes A_{t_1}) \ltimes A_{t_2}) \dots \ltimes A_{t_k}(\mathcal{A})).$$

For $t \in T$ let $\mathcal{T}_t$ be the subtree of $\mathcal{T}$ with root t and let T_t the vertex set of $\mathcal{T}_t$. Let furthermore ϕ_t be the subquery of ϕ defined as $\phi_t := \bigwedge_{t' \in T_t} \lambda_{t'}$.

Claim 2.3.20. *For every vertex $t \in T$ we have $A_t = \pi_{var(\lambda_t)}(\phi_t(\mathcal{A}))$.*

Proof. If t is a leaf, then the proof is immediate.

Consider now $t \in T$ with children $t_1, \dots t_k$. We first show $A_t \subseteq \pi_{\mathsf{var}(\lambda_t)}(\phi_t(\mathcal{A}))$. So consider $a \in A_t$. By definition of A_t we have for every $i \in [k]$ a tuple $a_i \in A_{t_i}$ with $a \sim a_i$. By induction we have $A_{t_i} = \pi_{\mathsf{var}(\lambda_{t_i})}(\phi_{t_i}(\mathcal{A}))$ for $i \in [k]$ and thus there is an assignment $b_i \in \phi_{t_i}(\mathcal{A})$ with $b_i|_{\mathsf{var}(\lambda_{t_i})} = a_i$. For $i \neq j$ we have by the connectivity condition $\mathsf{var}(\phi_{t_i}) \cap \mathsf{var}(\phi_{t_j}) \subseteq \mathsf{var}(\lambda_t)$ and thus the b_i are all compatible. It follows that we can combine a and the b_i to an assignment $b \in \phi_t(\mathcal{A})$ with $b|_{\mathsf{var}(\lambda_t)} = a$ which completes the first direction of the proof.

For the other direction, consider $b \in \phi_t(\mathcal{A})$. Obviously, $b|_{\mathsf{var}(\lambda_t)} \in \lambda_t(\mathcal{A})$. Moreover, for every $i \in [k]$ we have $b|_{\mathsf{var}(\phi_{t_i})} \in \phi_{t_i}(\mathcal{A})$. Thus by induction $b|_{\mathsf{var}(\lambda_{t_i})} \in A_{t_i}$. It follows with the definition of A_t that $b|_{\mathsf{var}(\lambda_t)} \in A_t$ which completes the proof of the claim. ■

Let r be the root of $\mathcal{T}$. Then $\phi(\mathcal{A}) \neq \emptyset$ if and only if $A_r \neq \emptyset$. Observing that A_r can be computed in polynomial time with Lemma 2.1.6 completes the proof. ■

Theorem 2.3.19 served as a starting point to finding more general classes of hypergraphs on which CQ is tractable, by trying to identify classes of "nearly" acyclic hypergraphs. There are lots of different decomposition techniques and associated width measures for hypergraphs. One of the most general width measures is *generalized hypertree width*.

The approach of generalized hypertree decomposition is similarly to that of tree decompositions: We want to organize a hypergraph into clusters that form a tree with a connectivity condition. Instead of bags that contain vertices and that must cover all edges, the basic clusters of generalized hypertree decompositions are *guarded blocks* (λ_t, χ_t) where λ_t contains edges while χ_t contains vertices. To make sure that the vertices χ_t of a guarded block form a sufficiently simple set, we demand that χ_t is covered by the edges in λ_t and that λ_t is small. To make sure that the decomposition represents the hypergraph well, we require that every edge must be contained in a set χ_t (but not necessarily in a λ_t). We now give the exact definition.

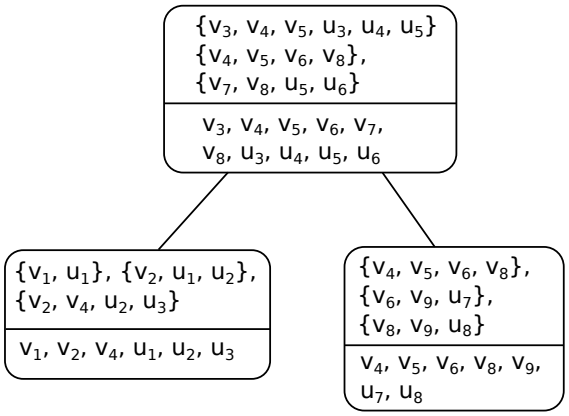

Figure 10: A generalized hypertree decomposition of width 3 for the hypergraph from Figure 8. The boxes are the guarded blocks. In the upper parts the guards are given while the lower parts show the blocks.

Definition 2.3.21. *A* generalized hypertree decomposition *of a hypergraph $\mathcal{H} = (V, E)$ is a triple $(\mathcal{T}, (\lambda_t)_{t\in T}, (\chi_t)_{t\in T})$ where $\mathcal{T} = (T, F)$ is a rooted tree and $\lambda_t \subseteq E$ and $\chi_t \subseteq V$ for every $t \in T$ satisfying the following properties:*

- *For every $e \in E$ there is a $t \in T$ such that $e \subseteq \chi_t$.*
- *For every $t \in T$ we have $\chi_t \subseteq \bigcup_{e\in\lambda_t} e$.*
- *For every $v \in V$ the set $\{t \in T \mid v \in \chi_t\}$ induces a subtree of $\mathcal{T}$.*

The third property is again called the connectedness condition. *The sets χ_t are called* blocks *or* bags *of the decomposition, while the sets λ_t are called the* guards *of the decomposition. A pair (λ_t, χ_t) is called* guarded block.

The width *of a decomposition $(\mathcal{T}, (\lambda_t)_{t\in T}, (\chi_t)_{t\in T})$ is $\max_{t\in T}(|\lambda_t|)$. The* generalized hypertree width *of $\mathcal{H}$ is the minimum width over all generalized hypertree decompositions of $\mathcal{H}$.* □

Again, we sometimes identify a guarded block (λ_t, χ_t) with the vertex t.

Example 2.3.22. Figure 10 shows a generalized hypertree decomposition of width 3 for the hypergraph from Figure 8. □

We give the following very easy upper bound for generalized hypertree width.

Observation 2.3.23. *Let $\mathcal{H} = (V, E)$ be a hypergraph such that there are k edges $e_1, \ldots, e_k \in E$ with $V \subseteq \bigcup_{i=1}^{k} e_i$. Then $\mathcal{H}$ has generalized hypertree width at most k.*

Proof. We will construct a trivial generalized hypertree decomposition $(\mathcal{T}, (\lambda_t)_{t\in T}, (\chi_t)_{t\in T})$ of $\mathcal{H}$ of width k. The tree $\mathcal{T}$ only consists of one single vertex t, the block of t is $\chi_t := V$ and the guard is $\lambda_t := \{e_1, \dots, e_k\}$. It is easily seen that this satisfies all desired properties of a hypertree decomposition. Furthermore, the decomposition has width k. ■

It turns out the generalized hypertree width is strictly more general than treewidth in the following sense.

Lemma 2.3.24 ([GLS00]). *For every hypergraph $\mathcal{H}$ the generalized hypertree width is less than or equal to $1 + \mathrm{tw}(\mathcal{H})$. Moreover, for every ℓ there are hypergraphs of treewidth ℓ and generalized hypertree width 1.*

To get an idea of the different decomposition techniques, we feel that it is instructional to give a proof.

Proof. For the first claim consider a hypergraph $\mathcal{H} = (V, E)$. It is easy to see that isolated vertices change neither the treewidth nor the generalized hypertree width of $\mathcal{H}$, so we assume w.l.o.g. that there are no isolated vertices. It follows that for each $v \in V$ we can choose an edge e_v with $v \in e_v$. Let $(\mathcal{T}, (\chi_t)_{t\in T})$ be a tree decomposition of width $k-1$ of the primal graph $\mathcal{H}_P$ of $\mathcal{H}$. We define for each $t \in T$ the guard $\lambda_t := \{e_v \mid v \in \chi_t\}$.

We claim that $(\mathcal{T}, (\lambda_t)_{t\in T}, (\chi_t)_{t\in T})$ is a generalized hypertree decomposition of $\mathcal{H}$. Each edge $e \in E$ induces a clique in $\mathcal{H}_P$ which by Lemma 2.3.4 lies in a bag χ_t. It follows that $e \subseteq \chi_t$ which proves the first first property of generalized hypertree decompositions. By definition of e_v, we have for each $t \in T$ that $\chi_t \subseteq \bigcup_{v\in\chi_t} e_v = \bigcup_{e\in\lambda_t} e$ which proves the second property of generalized hypertree decompositions. Finally, the connectivity condition is clear because $(\mathcal{T}, (\chi_t)_{t\in T})$ is a tree decomposition. Thus $(\mathcal{T}, (\lambda_t)_{t\in T}, (\chi_t)_{t\in T})$ is a generalized hypertree decomposition of $\mathcal{H}$. Clearly, the decomposition is of width at most k which completes the proof of the first claim.

For the second claim consider $G = (V, E)$ with $V := [\ell + 1]$ and $E := \{V\}$. By Observation 2.3.23 the hypergraph $\mathcal{H}$ has generalized hypertree width 1. But $\mathcal{H}_P$ is a clique with $\ell + 1$ vertices which by Lemma 2.3.4 has treewidth ℓ. ■

Unfortunately, deciding if a hypergraph has generalized hypertree width at most k is NP-complete even for $k = 3$ [GMS09]. This unpleasant result is amended by the fact that there is an approximation algorithm.

Theorem 2.3.25 ([AGG07, GLS02]). *There is an algorithm that, given a hypergraph $\mathcal{H}$ of generalized hypertree width k, constructs a generalized hypertree decomposition of width $O(k)$ of $\mathcal{H}$ in time $|\mathcal{H}|^{O(k)}$.*

The connection between generalized hypertree width and acyclic hypergraphs is given by the following lemma proved by Gottlob et al. [GLS01]:

Lemma 2.3.26 ([GLS01]). *A hypergraph is acyclic if and only if it has generalized hypertree width* 1.

Note that Gottlob et al. state Lemma 2.3.26 for the slightly more restrictive notion of hypertree width instead of generalized hypertree width. It is easily verified that their proof works for generalized hypertree decompositions without any changes. To acquaint the reader with the definitions, we present this proof from [GLS01].

Proof of Lemma 2.3.26. Let $\mathcal{H}$ first be an acyclic hypergraph with a join tree $(\mathcal{T}, (\lambda_t)_{t\in T})$. For each $t \in T$ set $\lambda'_t := \{\lambda_t\}$ and $\chi'_t := \lambda_t$. Then it is easy to check that $(\mathcal{T}, (\lambda'_t)_{t\in T}, (\chi'_t)_{t\in T})$ is a generalized hypertree decomposition of $\mathcal{H}$ of width 1. It follows that $\mathcal{H}$ has generalized hypertree width 1.

Let now $\mathcal{H} = (V, E)$ be a hypergraph with a generalized hypertree decomposition $(\mathcal{T}, (\lambda_t)_{t\in T}, (\chi_t)_{t\in T})$ of width 1. We claim that we may w.l.o.g. assume that the decomposition is such that for each edge $e \in E$ there is a $t_e \in T$ with $\lambda_{t_e} = \{e\}$ and $\chi_{t_e} = e$. Assume this is not the case. By definition there is a $t \in T$ such that $e \subseteq \chi_t$. We add a new leaf t_e to $\mathcal{T}$ that we connect to t by an edge. Moreover, we set $\lambda_{t_e} := \{e\}$ and $\chi_{t_e} := e$. It is easy to see that the result is still a generalized hypertree decomposition of width 1. Doing this for all edges $e \in E$, for which it is necessary, yields a decomposition of the desired form.

We now transform $(\mathcal{T}, (\lambda_t)_{t\in T}, (\chi_t)_{t\in T})$ into into another decomposition. For each edge e such that there are vertices $t_1, \ldots, t_k$ distinct from t_e with $\lambda_{t_i} = \{e\}$, $i \in [k]$ we do the following: For $i = 1, \ldots, k$ we delete t_i and connect all children of t_i in $\mathcal{T}$ to t_e. Observe that this operation preserves the connectivity condition, because $\lambda_{t_i} = \{e\}$ and thus $\chi_{t_i} \subseteq e$.

The result of the above construction is a generalized hypertree decomposition $(\mathcal{T}', (\lambda'_t)_{t\in T'}, (\chi'_t)_{t\in T'})$ of width 1 with $\mathcal{T}' = (T', F')$. Each edge $e \in E$ appears in exactly one guard λ'_{t_e} and for the corresponding block we have $\chi'_{t_e} = e$. It is easy to see that $(\mathcal{T}', (\lambda''_t)_{t\in T'})$ with $\lambda''_t := \chi'_t$ is a join tree of $\mathcal{H}$. Thus $\mathcal{H}$ is acyclic. ■

Using Lemma 2.3.26 it is very easy to prove the following lemma.

Lemma 2.3.27. *Let $\mathcal{H}$ be a hypergraph with a generalized hypertree decomposition $(\mathcal{T}, (\lambda_t)_{t\in T}, (\chi_t)_{t\in T})$. Let $\mathcal{H}' = (V, E')$ where $E' := \{\chi_t \mid t \in T\}$. Then $\mathcal{H}'$ is acyclic and $(\mathcal{T}, (\chi_t)_{t\in T})$ is a join tree of $\mathcal{H}$.*

Proof. We have that $(\mathcal{T}, (\lambda'_t)_{t\in T}, (\chi'_t)_{t\in T})$ with $\lambda'_t := \{\chi_t\}$ and $\chi'_t := \chi_t$ is a generalized hypertree decomposition of $\mathcal{H}'$ of width 1 and thus $\mathcal{H}'$ is acyclic with Lemma 2.3.26. ■

Next we state a lemma that in different forms is (often implicitly) used in most papers that deal with the application of hypergraph decomposition techniques to CQ.

Lemma 2.3.28. *Given a* CQ*-instance* Φ *with associated hypergraph* $\mathcal{H}$ *and a generalized hypertree decomposition of* $\mathcal{H}$ *of width* k*, one can compute an* ACQ*-instance* Ψ *in time* $\|\Phi\|^{O(k)}$ *such that* Φ *and* Ψ *are solution equivalent.*

Proof. Let $\Phi = (\mathcal{A}, \phi)$ and let the given generalized hypertree decomposition be $(\mathcal{T}', (\lambda_t)_{t\in T}, (\chi_t)_{t\in T})$. We construct $\Psi = (\mathcal{B}, \psi)$ with $\mathsf{var}(\psi) = \mathsf{var}(\phi)$ as follows: For every $t \in T$ the query ψ has an atom ψ_t with relation symbol $\mathcal{R}_t$ and $\mathsf{var}(\psi_t) := \chi_t$. The quantifiers are the same as for ϕ. For every $t \in T$ let $\phi_1, \ldots, \phi_s$ be the atoms associated to the edges in λ_t. We have $s \leq k$. Let $\phi'_1, \ldots, \phi'_\ell$ be the atoms associated to the edges e with $e \subseteq \chi_t$. Then we define the relation $\mathcal{R}_t^{\mathcal{B}}$ as

$$\mathcal{R}_t^{\mathcal{B}} := \pi_{\chi_t}(\phi_1(\mathcal{A}) \bowtie \ldots \bowtie \phi_s(\mathcal{A})) \bowtie \phi'_1(\mathcal{A}) \bowtie \ldots \bowtie \phi'_\ell(\mathcal{A}).$$

We claim that ϕ and ψ are solution equivalent. First consider an assignment a that satisfies all atoms of ϕ. Then we have for every subset $\phi'_1, \ldots, \phi'_r$ of $\mathsf{atom}(\phi)$ that the assignment b is compatible to an assignment in $\phi'_1 \bowtie \ldots \bowtie \phi'_r$. If follows that for each t the new atom ψ_t is satisfied by a. Consequently, $\phi(\mathcal{A}) \subseteq \psi(\mathcal{B})$.

Let now a be an assignment that satisfies all atoms of ψ. Then a must for each t satisfy the atoms ϕ'_i from the construction of $\mathcal{R}^{\mathcal{B}}$. But since each edge e of $\mathcal{H}$ is covered by a set $\chi_{t'}$, every $\phi' \in \mathsf{atom}(\phi)$ contributes as a ϕ'_i in the construction of a ϕ_t. Consequently, a satisfies all atoms of ϕ and thus $\psi(\mathcal{B}) \subseteq \phi(\mathcal{A})$.

We claim that this construction can be done in time $\|\Phi\|^{O(k)}$. To see this, observe that the relation $A_{\lambda_t} := \pi_{\chi_t}(\phi_1(\mathcal{A}) \bowtie \ldots \bowtie \phi_s(\mathcal{A}))$ has size at most $\|\mathcal{A}\|^s \leq \|\mathcal{A}\|^k$. Since for the ϕ'_i we have $\mathsf{var}(\phi'_i) \subseteq \chi_t$, it follows that $\mathcal{R}_t^{\mathcal{B}} \subseteq A_{\lambda_t}$ and thus consequently $\|\mathcal{R}_t^{\mathcal{B}}\| \leq \|\mathcal{A}\|^k$. With Lemma 2.1.6 it follows that we can compute $\mathcal{R}_t^{\mathcal{B}}$ in time $|\phi|\|\mathcal{A}\|^{O(k)}$. Thus computing the instance Ψ takes time $\|\Phi\|^{O(k)}$.

Finally, by Lemma 2.3.27 we have that Ψ is acyclic. ∎

The combination of Theorem 2.3.19 and Lemma 2.3.28 allows to solve CQ-instances in time $\|\Phi\|^{O(k)}$ provided that a generalized hypertree decomposition of width k is given. Thus the bottleneck for solving CQ-instances for many proposed decomposition techniques is the efficient computation of a good decomposition of the instance.

Let us fix some notation: For an edge set $\lambda \subseteq E$ we use the shorthand $\bigcup\lambda := \bigcup_{e\in\lambda} e$. For a decomposition $(\mathcal{T}, (\lambda_t)_{t\in T}, (\chi_t)_{t\in T})$ we write $\mathcal{T}_t$ for the subtree of $\mathcal{T}$ that has t as its root. We also write $\chi(\mathcal{T}_t) := \bigcup_{t'\in V(\mathcal{T}_t)} \chi_{t'}$. We use these notations for tree decompositions as well.

It is sometimes helpful to consider restrictions of generalized hypertree decompositions, because those might have better structural

or algorithmic properties. One such restriction are hingetree decompositions.

Definition 2.3.29. *A generalized hypertree decomposition is called* hingetree decomposition *if it satisfies the following conditions:*

- *For each pair $t_1, t_2 \in T$ with $t_1 \neq t_2$ there are edges $e_1 \in \lambda_{t_1}$ and $e_2 \in \lambda_{t_2}$ such that $\chi_{t_1} \cap \chi_{t_2} \subseteq e_1 \cap e_2$.*
- *For each $t \in T$ we have $\bigcup \lambda_t = \chi_t$.*
- *For each $e \in E$ there is a $t \in T$ such that $e \in \lambda_t$.*

The hingetree width *(also called* degree of cyclicity*) of $\mathcal{H}$ is the minimum width over all hingetree decompositions of $\mathcal{H}$.* □

Note that this is not the original definition from [GJC94] but an alternative, equivalent definition from [CJG08].

Example 2.3.30. The decomposition from Figure 10 is also a hingetree decomposition. □

Like treewidth, hingetree width is strictly less general than generalized hypertree width in the following sense.

Lemma 2.3.31 ([GLS00]). *For every hypergraph the generalized hypertree width is less than or equal to the hingetree width. Moreover, there are hypergraphs for which the generalized hypertree width is strictly less than the hingetree width.*

Hingetree width makes up for this lack of generality by the fact that optimal decompositions can be computed very efficiently.

Lemma 2.3.32 ([GJC94]). *There is an algorithm that, given a hypergraph $\mathcal{H} = (V, E)$, computes a minimum width hingetree decomposition of $\mathcal{H}$ in time $|V|^{O(1)}$.*

As a consequence of the results presented above we get the following lemma.

Lemma 2.3.33 (see e.g. [CJG08]). *For all of the width measures defined above* CQ *restricted to instances Φ of width k can be solved in time $\|\Phi\|^{p(k)}$ for a polynomial p.*

3

THE COMPLEXITY OF #CQ AND QUANTIFIED STAR SIZE

3.1 THE COMPLEXITY OF #CQ

In this section we quickly survey some known results on the complexity of #CQ and prove a parameterized hardness result that we will use later.

From Example 2.1.2 it is easily seen that CQ is NP-complete in general [CM77], but as discussed in Chapter 2 there are structural restrictions of the associated hypergraphs like acyclicity or bounded generalized hypertree width that make CQ tractable. The situation for #CQ is more complicated.

Theorem 3.1.1 ([BCC+05]). #CQ *is* #P*-complete for quantier free queries and* # · NP*-complete in general.*

Thus, for quantifier free queries the situation is as expected, but adding quantifiers makes the problem complete for the somewhat obscure class # · NP. It follows that #CQ is likely not in #P. Pichler and Skritek [PS13] considered the acyclic case and found that the quantifier free case is tractable.

Theorem 3.1.2 ([PS13]). #ACQ *restricted to quantifier free instances can be solved in polynomial time.*

We will get a proof of Theorem 3.1.2 as a corollary in Chapter 10.

As in Theorem 3.1.1, unlike for decision the counting problem gets harder when allowing quantification.

Theorem 3.1.3 ([PS13]). #ACQ *is* #P*-complete. It is* #P*-hard even for instances that have only one single existential quantifier.*

To keep this thesis self-contained we give an alternative proof of Theorem 3.1.3. Also we give a parameterized version of it which we will use later. To this end, we state a more detailed lemma.

Lemma 3.1.4. *Let* $\phi_{star,n} := \exists z \bigwedge_{i\in[n]} \mathcal{R}_i(z, y_i)$ *and let* $\mathcal{C}_{star} := \{\phi_{star,n} \mid n \in \mathbb{N}\}$.

a) #CQ *is* #P*-hard for instances restricted to queries in* $\mathcal{C}_{star}$.

b) p-#CQ *is* #W[1]*-hard for instances restricted to queries in* $\mathcal{C}_{star}$.

Proof. We show hardness by a polynomial time, parameterized T-reduction from p-#Clique. It is well-known that #Clique, the problem of counting all clicques in a graphs is #P-complete [PB83]. Technically this paper only proves #P-completeness for counting independent sets of a graph but #P-completeness for counting cliques follows easily by considering complement graphs. It follows easily that the following problem is #P-complete as well.

$\#Clique_k$ **Input:** Graph G, $k \in \mathbb{N}$. **Problem:** Compute the number of k-cliques in G.

Thus a polynomial-time reduction from p-#Clique suffices to show both hardness results (cf. Section 2.2).

The basic idea of the reduction is that instead of counting k-cliques in a graph, we can also count the k-tuples of vertices that are *not* a clique.

So let $G = (V, E)$ be a simple, undirected graph and $k \in \mathbb{N}$. A tuple $(v_1, \dots, v_k) \in V^k$ is not a clique if and only if it there are $i, j \in [k], i \neq j$ such that $v_i v_j$ is not an edge. Observe that because G is loopless this is necessarily true if $(v_1, \dots, v_k)$ contains a double vertex. We will show how to check if a tuple $(v_1, \dots v_k)$ is not a clique with a CQ-instance of the prescribed form.

More concretely, we construct a #CQ-instance $\Phi = (\mathcal{A}, \phi)$ with $\phi := \exists z \bigwedge_{i \in [k]} \mathcal{R}_i(z, y_i)$. Clearly the query is of the right form. The domain of $\mathcal{A}$ is $A := V \cup (V \times V \times [k] \times [k])$. For each $i \in [k]$ the structure $\mathcal{A}$ has the following relation associated with $\mathcal{R}_i$

$$\begin{aligned}\mathcal{R}_i^{\mathcal{A}} :=& \big\{((v, w, i, j), v), ((v, w, j, i), w) \mid v, w \in V \\ & \qquad v \neq w, vw \notin E, i, j \in [k], j \neq i\big\} \\ & \cup \big(V \times V \times ([k] \setminus \{i\}) \times ([k] \setminus \{i\})\big) \times V.\end{aligned}$$

This completes the construction of Φ.

First, observe that Φ can be constructed in time polynomial in $|G|$ and k, so if we can compute the number of k-cliques of G from $|\phi(\mathcal{A})|$ sufficiently quickly, the construction is indeed a polynomial-time parameterized T-reduction.

Furthermore, observe that for each satisfying assignment of Φ the variables $y_1, \dots, y_k$ take only values in V. We claim that an assignment $a : \{y_1, \dots, y_k\} \to A$ satisfies ϕ if and only if $a(y_1), \dots, a(y_k)$ is not a clique of size k in G. Essentially, the quantified variable z here guesses the edge that is missing between $a(y_i)$ and $a(y_j)$.

Indeed, if $a(y_1), \dots, a(y_k)$ is a tuple of vertices such that two vertices in it are not adjacent, say $a(y_i) = v_i$, $a(y_j) = v_j$, $v_i v_j \notin E$, then assigning (v_i, v_j, i, j) to z satisfies all atoms.

Let on the other hand $a(y_1), \dots, a(y_k)$ be a clique of size k in G. We claim that there is no assignment to z that satisfies all atoms. Clearly in a satisfying assignment z can take no value in V. So z must take a

value in $V \times V \times [k] \times [k]$, say (v, w, i, j). But then in particular $\mathcal{R}_i(z, y_i)$ and $\mathcal{R}_j(z, y_j)$ are satisfied. It follows that $a(y_i) = v$, $a(y_j) = w$, $vw \notin E$, which is a contradiction. So indeed, $a(y_1), \ldots, a(y_k)$ is a clique of size k in G if and only if a is a satisfying assignment.

It follows that the number of cliques in G is $\frac{1}{k!}(|V|^k \setminus |\phi(\mathcal{A})|)$. But $|V|^k$ and $k!$ can be easily computed in time $(k|V|)^{O(1)}$ and thus one can compute the number of k-cliques of G from $|\phi(\mathcal{A})|$, G and k in time $(k|V||\phi|)^{O(1)}$. Observing that we may safely assume that $k \leq |V|$—otherwise G does not contain any k-Cliques trivially—completes the reduction. ■

Remark 3.1.5. Observe that the instance Φ constructed in the proof of Lemma 3.1.4 has the following property: If we substitute in ϕ one $\mathcal{R}_j(z, x_j)$ by $\mathcal{R}_i(z, x_j)$ for a pair $i, j \in [k]$ with $i \neq j$, then every assignment $a : \mathsf{free}(\phi) \to V$ is in $\phi(\mathcal{A})$. This is because setting z to (v, v, j, j) satisfies all atoms. We will use this observation later in Chapter 6. □

We now get Theorem 3.1.3 as a corollary.

Proof of Theorem 3.1.3. Containment in #P is straightforward: Given an instance $(\mathcal{A}, \phi)$, nondeterministically guess an assignment to the variables in $\mathsf{free}(\phi)$ and solve the resulting CQ-instance with Theorem 2.3.19 (observe that by Observation 3.2.2 this instance is still acyclic).

#P-hardness follows directly from Lemma 3.1.4 ■

The remainder of this and the following chapters will mostly be devoted to a better understanding of the hardness result of Lemma 3.1.4. We will analyse exactly what makes the instances in Theorem 3.1.3 hard and use this to define a parameter that makes #CQ tractable.

3.2 QUANTIFIED STAR SIZE

As proved in Lemma 3.1.4, even introducing one single existential quantifier in acyclic conjunctive queries leads to #P-complete counting problems. It follows that bounding the number of quantified variables does not yield tractable instances. In this section, we will introduce a new parameter for #CQ-instances, called *quantified star size*, that makes #CQ tractable when combined with known decomposition techniques. This shows that not the number of quantified variables is crucial but how they are distributed in the associated hypergraph. A basic observation on the hard instances in the proof of Lemma 3.1.4 is that their associated hypergraphs are stars whose center is the single quantified variable. Abstracting this observation, we shall introduce the parameter *quantified star size* that, when bounded, leads to tractable #CQ instances.

Before we introduce quantified star size, we make several other definitions.

Let $\mathcal{H} = (V, E)$ be a hypergraph and $V' \subseteq V$. The *induced subhypergraph* $\mathcal{H}[V']$ of $\mathcal{H}$ is the hypergraph $\mathcal{H}[V'] = (V', E')$ where $E' := \{e \cap V' \mid e \in E, e \cap V' \neq \emptyset\}$. The induced subhypergraph of an edge set $E' \subseteq E$ is $\mathcal{H}[E'] = (\bigcup_{e \in E'} e, E')$. Let $x, y \in V$, a *path* between x and y is a sequence of vertices $x = v_1, ..., v_k = y$ such that for each $i \in [k-1]$ there is an edge $e_i \in E$ with $v_i, v_{i+1} \in e_i$.

A (connected) *component* of $\mathcal{H}$ is an induced subhypergraph $\mathcal{H}[V']$ where V' is a maximal vertex set such that for each pair $x, y \in V'$ there is a path between x and y in $\mathcal{H}$. These definitions apply to graphs as well.

An edge-path between two vertices $x, y \in V$ is a sequence of edges $e_1, ..., e_k \in E$ such that $x \in e_1$, $y \in e_k$, and for all $i \leq k-1$, $e_i \cap e_{i+1} \neq \emptyset$. One could define components of a hypergraph based on edge paths instead of paths in the obvious way. Fortunately, it is easy to see that both definitions coincide.

We will use the following observation on induced subgraphs of acyclic hypergraphs.

Observation 3.2.1. *If $\mathcal{H} = (V, E)$ is an acyclic hypergraph and $V' \subseteq V$, then $\mathcal{H}[V']$ is acyclic. More specifically, let $(\mathcal{T}, (\lambda)_{t \in T})$ with $\mathcal{T} = (T, F)$ be a join tree of $\mathcal{H}$. Then $(\mathcal{T}[T'], (\lambda_t \cap V')_{t \in T'})$ where $T' := \{t \in T \mid \lambda_t \cap V' \neq \emptyset\}$ can be made into a join tree of $\mathcal{H}[V']$ by connecting the components of $\mathcal{T}[T']$ arbitrarily.*

Proof. Immediate. $\mathcal{T}[T']$ is a subforest of $\mathcal{T}$. The connectedness condition of the set $\{t \in T' \mid v \in \lambda(t)\}$, for all $v \in V'$ follows directly from the connectedness condition of $(\mathcal{T}, (\lambda_t)_{t \in T})$. ■

An analogous statement is also true for the more general classes of hypergraphs we consider.

Observation 3.2.2. *Let β be any decomposition technique defined in this section. Let $\mathcal{H} = (V, E)$ be a hypergraph of β-width k. Then for every $V' \subseteq V$ the induced subhypergraph $\mathcal{H}[V']$ has β-width at most k.*

Proof. Let $(\mathcal{T}, (\lambda_t)_{t \in T}, (\chi_t)_{t_i n T})$ be a β-decomposition of $\mathcal{H}$ of width k. For each guarded block (λ_t, χ_t) compute a guarded block (λ'_t, χ'_t) with $\chi_t := \chi_t \cap V'$ and $\lambda_t := \{e \cap V' \mid e \in \lambda\}$. It is easy to check that $(\mathcal{T}, (\lambda'_t)_{t \in T}, (\chi'_t)_{t_i n T})$ is a β-decomposition of width at most k. ■

A *subhypergraph* $\mathcal{H}' = (V', E')$ of $\mathcal{H} = (V, E)$ is a hypergraph with $V' \subseteq V$ and $E' \subseteq \{e \cap V' \mid e \in E, e \cap V' \neq \emptyset\}$. Observe that there is no version of Observation 3.2.1 or Observation 3.2.2 for subhypergraphs instead of induced subhypergraphs. To see this consider an arbitrary hypergraph $\mathcal{H} = (V, E)$. Adding the edge V to E yields an acyclic hypergraph, independent of the generalized hypertree width of $\mathcal{H}$.

Definition 3.2.3. *An* S-hypergraph *is a pair $(\mathcal{H}, S)$ where $\mathcal{H} = (V, E)$ is a hypergraph and $S \subseteq V$. If $\mathcal{H}$ is a graph, we also call $(\mathcal{H}, S)$ an* S-graph.

The S-hypergraph associated to a CQ*-instance $\Phi = (\mathcal{A}, \phi)$ consists of the hypergraph associated to ϕ and $S := \mathsf{free}(\phi)$. The* primal S-graph *of $\mathcal{H}$ is defined as $(\mathcal{H}_P, S)$.* □

Definition 3.2.4. *Let $\mathcal{G}$ be a class of S-hypergraphs. By* #CQ on $\mathcal{G}$ *we denote the restriction of* #CQ *to instances whose associated S-hypergraph is in $\mathcal{G}$. Analogously, by* p-#CQ on $\mathcal{G}$ *we denote the restriction of* p-#CQ *to instances whose associated S-hypergraph is in $\mathcal{G}$.* □

Definition 3.2.5. *We call an S-hypergraph* S-connected *if for every pair of vertices x, y there is a path $x = v_1, v_2, \ldots, v_{k-1}, v_k = y$ such that $v_i \notin S$ for $i \notin \{1, k\}$.* □

Let us consider some examples of queries that have S-connected S-hypergraphs.

Example 3.2.6. Path queries (of arbitrary length), for example

$$\phi(x, y) := \exists t_1 \exists t_2 \exists t_3 \mathcal{R}(x, t_1) \wedge \mathcal{R}(t_1, t_2) \wedge \mathcal{R}(t_2, t_3) \wedge \mathcal{R}(t_3, y)$$

have as their associated S-hypergraph a path in which only the end vertices are in S. Thus their S-hypergraph is S-connected. □

Example 3.2.7. The associated graph of the query $\phi_{\text{star},n}$ of Lemma 3.1.4 is the star $G_n = (V_n, E_n)$ where $V_n = \{z, y_1, \ldots, y_n\}$ and $E_n = \{zy_1, \ldots, zy_n\}$. Furthermore, the free variables are $S_n = \{y_1, \ldots, y_n\}$. Every vertex in V_n is connected to every other vertex via $z \notin S_n$. Thus (G_n, S_n) is S_n-connected. □

Definition 3.2.8. *An* independent set *I in a hypergraph $\mathcal{H} = (V, E)$ is a set $I \subseteq V$ such that there are no distinct vertices $x, y \in I$ that lie in a common edge $e \in E$.* □

Definition 3.2.9. *The S-*star size *of an S-connected S-hypergraph is the maximum size of an independent set consisting of vertices in S only. We say that such an independent set* forms *an S-star.* □

We remark that the S-star size of an S-connected S-hypergraph can equivalently be expressed as the the size of a maximum independent set in $\mathcal{H}[S]$.

Example 3.2.10. The S-hypergraphs associated to the path queries of Example 3.2.6 have S-star size 2, because the two end vertices of the paths are independent.

Now consider the S_n-hypergraph (G_n, S_n) from Example 3.2.7. The vertices of S_n are all independent. Consequently, the S_n-star size of (G_n, S_n) is n.

Note that while the quantified star size of instances whose associated hypergraph is a path is bounded by 2, the S-star size of bounded

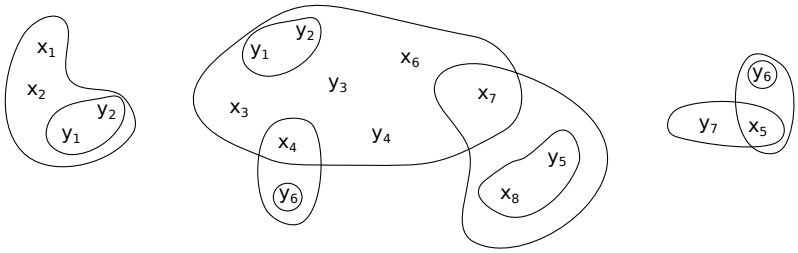

Figure 11: The S-components of the query from Example 2.3.12.

pathwidth instances is unbounded. To see this, observe that the graph S_n-hypergraph (G_n, S_n) from above has pathwidth 1: The decomposition $(\mathcal{P}, (\chi_t)_{t\in T})$ where $\mathcal{P}$ is the path with vertex set $[n]$ and $\chi_i := \{z, y_i\}$, is a path decomposition of G_n of width 1. □

We want to extend the notion of S-star size to S-hypergraphs that are not S-connected. To this end, we consider certain S-connected subhypergraphs that we call S-components. We make the following definition which will be crucial during the remainder of Part i.

Definition 3.2.11. *Let $\mathcal{H} = (V, E)$ be a hypergraph and $S \subseteq V$. Let C be the vertex set of a connected component of $\mathcal{H}[V - S]$. Let E_C be the set of hyperedges $\{e \in E \mid e \cap C \neq \emptyset\}$ and $V'_C := \bigcup_{e\in E_C} e$. Then $\mathcal{H}[V'_C]$ is called an* S-component *of $\mathcal{H}$.* □

Since the definition of S-components is somewhat involved, let us consider several examples.

Example 3.2.12. We claim that the S-hypergraph $(\mathcal{H}, S)$ associated to the query $\phi(y_1, \ldots, y_7)$ of Example 2.3.12 (see also Figure 7) has the three S-components that are depicted in Figure 11. To see this, observe first that $S = \{y_1, \ldots, y_7\}$. It follows that the vertex sets of the components of $\mathcal{H}[V \setminus S]$ are $C_1 := \{x_1, x_2\}$, $C_2 := \{x_3, x_4, x_6, x_7, x_8\}$ and $C_3 := \{x_5\}$. Consequently,

- $E_{C_1} = \big\{\{x_1, x_2, y_1, y_2\}\big\}$,
- $E_{C_2} = \big\{\{x_3, x_4, x_6, x_7, y_1, y_2, y_3, y_4\}, \{x_4, y_6\}, \{x_7, x_8, y_5\}, \{x_8, y_5\}\big\}$,
- $E_{C_3} = \big\{\{x_5, y_7\}, \{x_5, y_6\}\big\}$.

Thus $V'_{C_1} = \{x_1, x_2, y_1, y_2\}$, $V'_{C_2} = \{x_3, x_4, x_6, x_7, x_8, y_1, y_2, y_3, y_4, y_5, y_6\}$ and $V'_{C_3} = \{x_5, y_6, y_7\}$. The resulting S-components are depicted in Figure 11. □

Example 3.2.13. Let us consider the S-hypergraph of the query from Example 2.3.13. The associated hypergraph $\mathcal{H}$ was already illustrated in Figure 8. We have $S = \{v_1, ..., v_9\}$. Then $\mathcal{H}[V \setminus S]$ has three components with the vertex sets $C_1 := \{u_7\}$, $C_2 := \{u_8\}$ and $C_3 := \{u_1, \ldots, u_6\}$. Thus

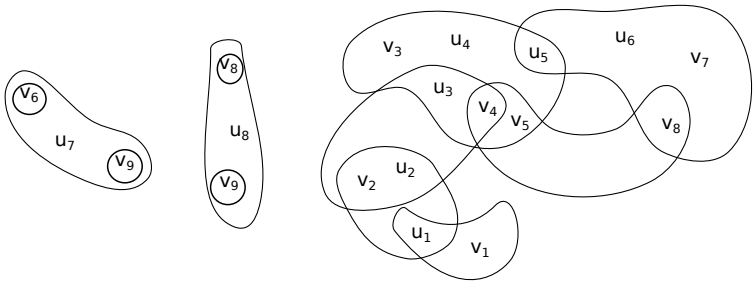

Figure 12: The S-components of the S-hypergraph discussed in Example 3.2.13.

- $E_{C_1} = \{\{u_7, v_6, v_9\}\}$,
- $E_{C_2} = \{\{u_8, v_8, v_9\}\}$, and
- $E_{C_3} = \{\{u_1, v_1\}, \{u_1, u_2, v_2\}, \{u_2, u_3, v_2, v_4\}, \{u_3, u_4, u_5, v_3, v_4, v_5\}, \{u_5, u_6, v_7, v_8\}\}$.

Hence, $V'_{C_1} = \{u_7, v_6, v_9\}$, $V'_{C_2} = \{u_8, v_8, v_9\}$ and $V'_{C_3} = \{u_1, u_2, u_3, u_4, u_5, u_6, v_1, v_2, v_3, v_4, v_5, v_7, v_8\}$. The three resulting S-components are depicted in Figure 12. □

The following observation is evident from the definition.

Observation 3.2.14. *To an S-hypergraph $(\mathcal{H}, S)$ one can in polynomial time compute all its S-components.*

Observation 3.2.15. *Let $\mathcal{H} = (V, E)$ be a hypergraph, $S \subseteq V$ and $\mathcal{H}'$ be an S-component of $\mathcal{H}$. Then, if $\mathcal{H}$ is acyclic, $\mathcal{H}'$ is acyclic.*

Proof. By definition $\mathcal{H}'$ is an induced subgraph. By Observation 3.2.1 it follows that it is acyclic. ■

The definitions directly yields the following observation.

Observation 3.2.16. *The only S-component of an S-connected S-hypergraph $(\mathcal{H}, S)$ is $\mathcal{H}$. Moreover, the S-components of S-hypergraphs are S-connected.*

Observation 3.2.16 allows to extend the definition of S-star size to not necessarily S-connected hypergraphs.

Definition 3.2.17. *For an S-hypergraph $(\mathcal{H}, S)$ we define S-*star size *as the maximum S-star size of its S-components.* □

Example 3.2.18. Let us continue the discussion from Example 3.2.12 (see also Figures 7 and 11). The S-component induced by V'_{C_1} has no

independent set of size greater than 1, because the only two vertices y_1, y_2 from V'_{C_1} appear in a common edge. Thus the S-star size of this component is 1.

We have $V_{C_2} \cap S = \{y_1, y_2, y_3, y_4, y_5, y_6\}$. A maximum independent set of these vertices in the S-component induced by V'_{C_2} is for example $\{y_1, y_5, y_6\}$. Thus the S-star size of this S-component is 3.

Finally, $V'_{C_3} \cap S = \{y_6, y_7\}$ and these vertices are independent. Thus the S-component induced by V'_{C_3} has S-star size 2

It follows that the S-star size of $(\mathcal{H}, S)$ is 3. □

Example 3.2.19. Let us compute the S-star size of the S-hypergraph of Example 3.2.13. The S-components induced by V'_{C_1} and V'_{C_2} are completely covered by the edges $\{u_7, v_6, v_9\}$, resp., $\{u_8, v_8, v_9\}$. It follows that the S-star size of these S-components is 1. We have $V_{C_3} \cap S = \{v_1, v_2, v_3, v_4, v_5, v_7, v_8\}$. There are several maximum independent sets of vertices in $V_{C_3} \cap S$ in the S-component induced by V_{C_3}, all of size 4. An example is $\{v_1, v_2, v_3, v_7\}$. It follos that the S-star size of $(\mathcal{H}, S)$ is 4 □

Now we can finally come back to CQ-instances and define the promised parameter quantified star size.

Definition 3.2.20. *The* quantified star size *of a conjunctive query is defined as the S-star size of the associated S-hypergraph. The quantified star size of a* CQ*-instance is that of its query.* □

Example 3.2.21. The query from Example 2.3.12 has quantified star-size 3 as we have seen in Example 3.2.18.

The query from Example 2.3.13 has quantified starsize 4 as we have seen in Example 3.2.19.

From Example 3.2.10 we get that the queries $\phi_{\text{star},n}$ of Lemma 3.1.4 are of quantified star size n, which is nearly the size of the query. □

3.3 FORMULATION OF MAIN RESULTS

We have now introduced all necessary preliminaries to formulate the main results of Part i of this thesis.

COUNTING SOLUTIONS TO QUERIES In Chapter 5 we justify our definition of quantified star size. We show that every class $\mathcal{G}$ of S-hypergraphs such that #CQ on $\mathcal{G}$ is tractable must be of bounded quantified star size—under the assumption $\mathsf{FPT} \neq \#\mathsf{W}[1]$ from parameterized complexity. We then go on showing that for all decomposition techniques for CQ-instances commonly considered in the literature combining them with bounded quantified star size leads to tractable counting problems. Combining the results we get an exact characterization of the subclasses of CQ-instances that allow tractable

counting for commonly considered classes defined by decomposition techniques.

Let us illustrate this for the example of generalized hypertree decompositions. We will show that, under the assumption that FPT $\neq$ #W[1], for any (recursively enumerable) class $\mathcal{G}$ of S-hypergraphs of bounded generalized hypertreewidth the following statements are equivalent:

- #CQ on $\mathcal{G}$ is tractable.
- p-#CQ on $\mathcal{G}$ is fixed-parameter tractable.
- $\mathcal{G}$ is of bounded quantified star size.

In our considerations, the arity of atoms of queries is not a priori bounded. For this setting, there is no known characterization of classes $\mathcal{G}$ of graphs that yield tractable instances for the decision problem CQ. This explains why our characterizations are stated for *each* decomposition method individually.

For bounded arity however, the situation is different as we will discuss in Chapter 6. It is known that the classes of hypergraphs of bounded arity that lead to tractable CQ are exactly the classes of bounded treewidth [GSS01, Gro07]. The same result is true for counting of quantifier free instances [DJ04], i.e., #CSP. In Chapter 6 we shall combine [GSS01, Gro07] and our results from Chapter 5 to derive a complete characterization of the classes $\mathcal{G}$ of S-hypergraphs such that #CQ is tractable on $\mathcal{G}$. These classes are exactly the classes of bounded treewidth and bounded quantified star size.

In the bounded arity case we arrive at a more finegrained analysis of tractable #CQ: In Chapter 7 we give a characterization of tractable classes of *queries* instead of associated S-hypergraphs, using the notion of the *core* of a conjunctive quer ϕ [CM77] which is a certain "minimal" equivalent subquery of ϕ. We show that #CQ restricted to a class $\mathcal{C}$ of queries is tractable if and only if the treewidth and the quantified star size of the cores of $\mathcal{C}$ is bounded. This result allows to isolate larger classes of tractable #CQ-instances which we do not get from the S-hypergraph perspective alone.

DISCOVERING QUANTIFIED STAR SIZE To exploit tractability results of the above kind it is helpful if the membership in a tractable class can be decided efficiently, i.e., in our case if computing the quantified star size of an instance is tractable. Therefore, we consider this "discovery problem" of determining the quantified star size of queries in Chapter 4.

We first show that the quantified star size of ACQ-instances can be determined in polynomial time. This result will also be used in the counting algorithms of Chapter 5 which explains the order of the presentation.

We show that computing the S-star size of S-hypergraphs is equivalent to computing maximum independent sets in hypergraphs. Consequently, we cannot expect a polynomial time algorithm for computing the quantified star size of general CQ-instances. Fortunately, it turns out that for queries ϕ of generalized hypertree width k and thus for all more restrictive decomposition techniques like hingetree width or treewidth, there is an algorithm that computes in time $|\phi|^{O(k)}$ the quantified star size of ϕ. We show that this is in a sense optimal, because computing the quantified star size of a given query ϕ is $\mathsf{W}[1]$-hard parameterized by the generalized hypertree width of ϕ. Thus, under the standard assumption $\mathsf{FPT} \neq \mathsf{W}[1]$, there is no fixed-parameter algorithm for this problem.

Still, if we parameterize the computation of quantified star size by more restrictive width measures, computing the quantified star size of conjunctive queries in some cases becomes fixed-parameter tractable. We prove that this is the case queries if we parameterize by hingetree width. Because of the connection between S-star size and maximum independent set, this result provides a new parameter of hypergraphs for which computing maximum independent sets is fixed-parameter tractable. Note that the $\mathsf{W}[1]$-hardness result from above shows that fixed-parameter tractability of computing maximum independent sets is unlikely to hold for other hypergraph decomposition techniques.

We then turn our attention to the approximation of quantified star size. We show that there is a polynomial time algorithm that, given a query ϕ and a decomposition of ϕ of width k, computes in time independent of k a k-approximation of the quantified star size of ϕ.

Summing these results up, quantified star size does not only imply tractable counting if combined with well known decomposition techniques, but in case the decomposition is given or can be efficiently computed (treewidth, hingetree width) or approximated (generalized hypertreewidth), then computing quantified star size is itself tractable.

3.4 DIGRESSION: UNIONS OF ACYCLIC QUERIES

For two CQ-instances $\Phi_1 = (\mathcal{A}_1, \phi_1)$ and $\Phi_2 = (\mathcal{A}_2, \phi_2)$ define the *union* as $(\Phi_1 \vee \Phi_2) := (\mathcal{A}_1 \cup \mathcal{A}_2, \phi_1 \vee \phi_2)$. The *intersection* of Φ_1 and Φ_2 is $(\Phi_1 \wedge \Phi_2) := (\mathcal{A}_1 \cup \mathcal{A}_2, \phi_1 \wedge \phi_2)$. Both notions naturally extend to more than two instances. Observe that the queries of unions of CQ-instances are not CQ-instances anymore but intersections are.

For deciding if the query result is empty, it is clear that taking unions of acyclic queries does not change the complexity—one can simply solve one instance after the other. Thus unions of ACQ-instances stay tractable.

For counting the situation is different: From Theorem 3.1.3, Pichler and Skritek directly get the following corollary.

Corollary 3.4.1 ([PS13]). *Counting solutions to unions of quantifier free* ACQ*-instances is* #P*-complete.*

Proof. Containment in #P is clear, because deciding emptyness of the query result of unions of ACQ-instances is in P as discussed above.

For hardness we reduce from CQ on $\mathcal{C}_{\text{star}}$ from Lemma 3.1.4. So let $\Phi = (\mathcal{A}, \phi)$ be such an with $\phi \in \mathcal{C}_{\text{star}}$. Remember that ϕ has only one free variable z. For each $d \in A$ let Φ_d be the CQ-instance that we get from Φ by fixing z to d. Clearly, the union of all Φ_d has the same satisfying assignments as Φ. ■

In this Chapter we will strengthen the hardness result of Corollary 3.4.1 in several directions: We show that already counting solutions for the union of *two* ACQ-instances is hard even when those are assumed to be of bounded domain size.

Proposition 3.4.2. *Counting solutions to the union and the intersection of two quantifier free* #ACQ*-instances are both* #P*-complete. This remains true for* #ACQ *on Boolean domain and arity at most* 3*.*

Remark 3.4.3. In [GSS01], it is proved that the (bi-)colored grid homomorphism problem is NP-complete. This result implies part of Proposition 3.4.2, i.e., that counting the solutions of the conjunction of two ACQ-instances is #P-complete (the fact that this hardness result is still true on Boolean domain does not follow, however). □

An $(n \times m)$*-grid* is a graph isomorphic to $G_{n,m} = (V_{n,m}, E_{n,m})$ where $V_{n,m} := [n] \times [m]$ and $(i,j)(i',j') \in E_{n,m}$ if and only if $|i - i'| + |j - j'| = 1$. A graph is a *grid* if and only if it is an $(n \times m)$-grid for a pair $n, m \in \mathbb{N}$.

For the proof of Proposition 3.4.2 we will use the following lemma.

Lemma 3.4.4. *Counting solutions to quantifier free* CQ*-instances whose primal graph is a grid is* #P*-complete even for domains of size* 4*.*

We will show the hardness part of Lemma 3.4.4 by reducing a restricted version of #CIRCUITSAT to #CQ with the desired grid structure. From the #P-completeness of our #CIRCUITSAT version we will get #P-hardness for counting solutions of conjunctive queries with grid structure.

We now define the version of #CIRCUITSAT that we call #(∧-¬-GRID)-CIRCUITSAT: An instance of #(∧-¬-GRID)-CIRCUITSAT is a Boolean circuit which only contains ∧- and ¬-gates and in which all gates are vertices of a 2-dimensional grid. Furthermore, the edges of the circuit are non-intersecting paths along the edges of the grid.

Lemma 3.4.5. *#(∧-¬-GRID)-CIRCUITSAT is #P-complete under parsimonious reductions.*

Proof. We make a parsimonious reduction from #CIRCUITSAT. Let C be a #CIRCUITSAT instance, i.e., a Boolean circuit. In a first step we substitute all $\vee$-gates $x \vee y$ by $\neg(\neg x \wedge \neg y)$. We then make sure that every gate has at most degree 3 and that all input gates and the output gate has at most degree 2 by adding double negations. Call the resulting circuit C'.

We now embed C' into a grid. To do so we take a three step approach that starts with a coarse grid that is then refined. Let n be the size of C'. We first distribute the gates of C' into a $(n \times n)$-grid G_1 such that each gate of C' of depth i has the coordinates (i, j) for some j. Furthermore, each edge of the circuit is a sequence of straight lines where each straight line goes from a vertex in one row to another vertex in the next row. Note that these stright lines are *not* edges of the grid G_1. we require that in each vertex of G_1 at most two lines start and end. For vertices on which no gate of C' lies, we assume that at most one edge starts and ends. It is clear that such an embedding can be constructed easily.

In a second step we make sure that the edges of the circuit follow the edges of a grid without congestion. We do this for each row of the coarse grid G_1 individually. We construct a new grid G_2 by adding $2n - 1$ new rows before each row in G_1 and one new column before each column. Observe that each vertex (i, j) in G_1 has the coordinates $(2ni, 2j)$ in G_2. Each vertex v of G_1 in row i has a most 2 outgoing straight lines l_1, l_2 representing edges of the circuit C' which both end in a vertex of row $i + 1$. Let l_1 end in $(i + 1, j)$ and l_2 end in $(i + 1, j')$ with $j < j'$, then we call l_1 be the low output and l_2 the high output. If there is only one output, we define it to be high. We also make the equivalent definition for high and low inputs.

Now we substitute the lines representing edges of C' by paths in G_2. Let l be a line that starts in G_1 in (i, j') and ends in $(i + 1, j)$. We construct a path P_l from $(2ni, 2j)$ to $(2n(i + 1), 2j')$:

- If l is a low output and a low input the path is the piecewise linear curve through the vertices $(2ni, 2j)$, $(2ni, 2j - 1)$, $(2ni + 2j, 2j - 1)$, $(2ni + 2j, 2j')$, $(2n(i + 1), 2j')$.
- If l is a high output and a low input the path is through $(2ni, 2j)$, $(2ni + 2j + 1, 2j)$, $(2ni + 2j + 1, 2j')$, $(2n(i + 1), 2j')$.
- If l is a low output and a high input the path is through $(2ni, 2j)$, $(2ni, 2j - 1)$, $(2ni + 2j, 2j - 1)$, $(2ni + 2j, 2j' + 1)$, $(2n(i + 1), 2j' + 1)$, $(2n(i + 1), 2j')$.
- If l is a high output and a high input the path is through $(2ni, 2j)$, $(2ni + 2j + 1, 2j)$, $(2ni + 2j + 1, 2j' + 1)$, $(2n(i + 1), 2j' + 1)$, $(2n(i + 1), 2j')$.

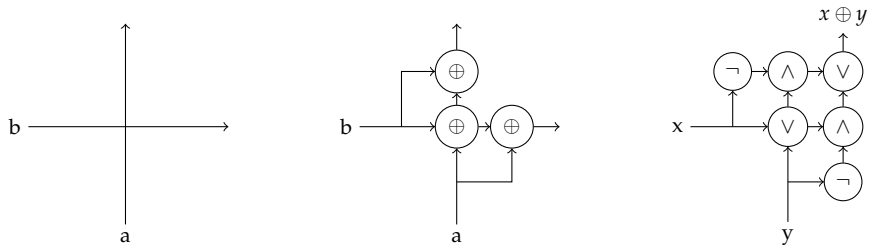

Figure 13: The crossing paths in the left are substituted by a gadget without crossings in the middle that uses $\oplus$-gates which compute xor of its inputs. It is easily checked that the outputs compute $(a \oplus b) \oplus a$ and $(a \oplus b) \oplus b$ which simplify to b and a respectively. On the right we show how the $\oplus$-gates can be simulated over the basis $\wedge, \vee, \neg$ without losing planarity. Degree 4 gates, splitting of edges and $\vee$-gates can be avoided by introducing some more $\neg$-gates and using De Morgan's law.

The result is an embedding of C' into a grid such that the gates are on vertices of G_2 and the edges of C' are paths in the grid. Observe that the paths were constructed in such a way that two paths between gates never share edges, so they only intersect in single vertices.

In the final step of the reduction we get rid of these intersections on non-gate vertices by adding additional gates. Each crossing in G_2 is substituted by the gadget illustrated in Figure 13. To do so we make the grid finer again by a constant factor. The result is a circuit C'' that is embedded into a grid. Furthermore C'' has the same satisfying assignments as C'. This completes the proof of Lemma 3.4.5. ■

Remark 3.4.6. We could also have given a proof of Lemma 3.4.5 with results on embedding general planar graphs into grids in the way we need it (see e.g. [Val81]). We have chosen to present an ad-hoc proof instead to keep the presentation self-contained. □

Proof of Lemma 3.4.4. By Theorem 3.1.1, counting solutions to general quantifier free CQ-instances is in #P, so we only need to show hardness.

To this end, we will now reduce #($\wedge$-$\neg$-GRID)-CIRCUITSAT to #CQ instances of grid structure. So let (C, G) be an instance of #($\wedge$-$\neg$-GRID)-CIRCUITSAT, i.e., a circuit C that is embedded into a grid G. Let G be of size $n \times n$. W.l.o.g. we may assume that no gates are on neighboring vertices in G and that the output gate is not a $\wedge$-gate. For each $\wedge$-gate of C we arbitrarily fix one input as the first input while the other one is the second one. We construct a binary CQ-instance Φ whose associated hypergraph is G. The domain is $A := \{0, 1, 2, 3\}$ where 0 and 1 represent the usual boolean values while 2 and 3 are used in a gadget construction for $\wedge$-gates. For each edge $e = uv$ in G we add an atom $\phi_e(u, v)$ with the following satisfying assignments:

- If e is not an edge of C, ϕ_e has the satisfying assignments $\{ab \mid a, b \in \{0, 1, 2, 3\}\}$.
- If e is an edge of C directed from u to v and v is not a gate and u is not a $\wedge$-gate, ϕ_e has the satisfying assignments $\{00, 11\}$.
- If e is an edge of C directed from u to v and v is a $\neg$-gate, ϕ_e has the satisfying assignments $\{01, 10\}$.
- If e is an edge of C directed from u to v and v is a $\wedge$-gate and the path to v over u is from the first input of v, ϕ_e has the satisfying assignment $\{00, 02, 11, 13\}$.
- If e is an edge of C directed from u to v and v is a $\wedge$-gate and the path to v over u is from the second input of v, ϕ_e has the satisfying assignment $\{00, 01, 12, 13\}$.
- If e is an edge of C directed from u to v and v is not a gate and u is a $\wedge$-gate, ϕ_e has the satisfying assignments $\{00, 10, 20, 31\}$.

Observe that the construction near the $\wedge$-gates is possible, because no two gates are neighbors. So the relations of the ϕ_e are all well defined. Now each vertex that is not part of C gets a unary atom that has only the single satisfying assignment 1. Also the output gate of C gets such a unary atom.

We claim that if we fix an assignment a to the variables representing the inputs of C, there is an satisfying extension to the other variables if and only if a satisfies C. Furthermore, this extension is unique. It is clear that the constraints along the paths and on the $\neg$-gates propagate the correct values along the grid. In a satisfying assignment, the variable representing an $\wedge$-gate has to take the value representing the values of its inputs in binary. The gates after the $\wedge$-gates then calculate the conjunction value for these inputs. This completes the proof of Lemma 3.4.4 ■

Proof of Proposition 3.4.2. Again, we only need to show hardness. By the inclusion-exclusion principle counting for unions and intersections is equally hard, so it suffices to show hardness for intersections. The reduction is straightforward with Lemma 3.4.4. Let Φ be a conjunctive query whose primal graph is a grid. We separate the atoms into two new CQ-instances: Φ_1 gets all the atoms that correspond to edges on rows of the grid, Φ_2 gets those that correspond to the columns. Clearly we have $\Phi = \Phi_1 \wedge \Phi_2$ and the Φ_i are acyclic. Thus the first part of the lemma follows.

To show that the result is true for queries on boolean domain, we sketch a different encoding of ($\wedge$-$\neg$-GRID)-#CIRCUITSAT into conjunctive queries. Roughly speaking, the structure of the encoding is basically the same but non-Boolean elements are mapped to sequences of boolean variables (that represent their binary encodings). To do so we need ternary relations. For completeness, details are given below.

Again, let G be the $(n \times n)$-grid and suppose no gates are on neighboring vertices in G and that the output gate is not a $\wedge$-gate. For each $\wedge$-gate v, we introduce a second vertex/variable v_1. We construct a ternary CQ instance Φ as follows. For each edge $e = uv$ in G we add an atom ϕ_e in the following way:

- If e is not edge of C , ϕ_e has the satisfying assignments $\{00, 01, 10, 11\}$.
- If e is an edge of C directed from u to v and v is not a gate and u is not a $\wedge$-gate, ϕ_e has the satisfying assignments $\{00, 11\}$.
- If e is an edge of C directed from u to v and v is a $\neg$-gate, ϕ_e has the satisfying assignments $\{01, 10\}$.
- If e is an edge of C directed from u to v and v is a $\wedge$-gate and the path to v over u is from the first input of v, ϕ_e is the ternary atom on variables u, v, v_1 with the satisfying assignment set $\{000, 010, 101, 111\}$.
- If e is an edge of C directed from u to v and v is a $\wedge$-gate and the path to v over u is from the second input of v, ϕ_e is the ternary atom on variables u, v, v_1 which has the satisfying assignment $\{000, 001, 110, 111\}$.
- If e is an edge of C directed from u to v and v is not a gate and u is a $\wedge$-gate, ϕ_e is the atom on variables u_1, u, v which has the satisfying assignments $\{000, 010, 100, 111\}$.

The atoms are then distributed into two CQ-instances Φ_1 and Φ_2 as above, grouping atoms with horizontal and vertical edges separately. Note that, connection at $\wedge$ gates v between hyperedges is now on two vertices v and v_1. But the resulting hypergraphs for Φ_1 and Φ_2 are still easily seen to be acyclic. ■

The reductions of this section are all parsimonious, so we directly get the following corollary [1]:

Corollary 3.4.7. *Deciding if the query result of the intersection of two quantifier free acyclic conjunctive queries is nonempty is* NP*-complete even for bounded arity and domain.*

Observe that as discussed above it is well-known that deciding the union of ACQ-instances can be done in time polynomial time in $\|\Phi\|$.

1 We state this corollary for completeness. Although we found no references, it is certainly already known

4

COMPUTING S-STAR SIZE

In this chapter we show how S-star size can be computed for different classes of S-hypergraphs. We start with acyclic S-hypergraphs before turning to more general classes.

4.1 ACYCLIC HYPERGRAPHS

In this section we show that the S-star size of acyclic S-hypergraphs can be computed in polynomial time.

Let $\mathcal{H} = (V, E)$ be a hypergraph and $S \subseteq V$. We say that $E^* \subseteq E$ *covers* S if $S \subseteq \bigcup_{e \in E^*} e$. We call E^* an *edge cover* of S. When $S = V$ we call E^* an edge cover of $\mathcal{H}$.

Lemma 4.1.1. *For acyclic hypergraphs the size of a maximum independent set and a minimum edge cover coincide. Moreover, there is a polynomial-time algorithm that, given an acyclic hypergraph $\mathcal{H}$, computes a maximum independent set I and a minimum edge cover E^* of $\mathcal{H}$.*

The first sentence in Lemma 4.1.1 can be seen as an adaptation of Kőnig's theorem for bipartite graphs (see e.g. [Bol98]) to acyclic hypergraphs. The proof uses a minimally modified version of an algorithm that Guo and Niedermeier [GN06] describe to compute minimum (unweighted) edge covers of acyclic hypergraphs. We show here that their techniques cannot only be used to compute minimum edge covers but also maximum independent sets of acyclic hypergraphs.

Proof of Lemma 4.1.1. Clearly the size of any independent set is not greater than that of any edge cover, simply because no edge can cover two vertices in an independent set. So if we present an algorithm that computes an independent set I and an edge cover E^* of a given acyclic hypergraph $\mathcal{H} = (V, E)$ such that $|I| = |E^*|$, we are done.

So let us now describe an algorithm that computes I and E^* of the same size: Let $\mathcal{T} = (T, F)$ be a join tree of $\mathcal{H}$ with root r. We start with initially empty sets I and E^* and iteratively delete leaves of $\mathcal{T}$ in a bottom-up manner from the leaves to the root. For each leaf $t \in T$, either $\lambda_t \subseteq \lambda_{t'}$, where t' is the parent of t, or there exists $y \in \lambda_t$ such that $y \notin \lambda_{t'}$. In the latter case, we will say that y is *unique* for t. If $t = r$ is a leaf of $\mathcal{T}$, i.e., r is the only vertex in $\mathcal{T}$, we say by convention that if λ_t contains any vertices, they are all unique for t.

We do the following until T is empty. First, choose a leaf t of $\mathcal{T}$. If there is no vertex unique for t, we simply delete t from T. If there are vertices that are unique for t, choose one vertex y among them and add it to I. Furthermore, add λ_t to E^*, delete all vertices in λ_t from $\mathcal{H}$

and delete t from T. When T is empty, I and E^* are the result of the algorithm.

It is instructive to execute this algorithm for the join tree in Figure 9.

Remember that for a vertex $t \in T$ we denote by $\mathcal{T}_t$ the subtree of $\mathcal{T}$ with the root t. Let furthermore V_t be defined as the set of vertices in V that appear only in $\{\lambda_{t^*} \mid t^* \in V(\mathcal{T}_t)\} \subseteq E$ and in no other edge in E.

Claim 4.1.2. *Whenever the algorithm deletes $t \in T$, the edge set E^* covers the vertices V_t.*

Proof. Assume that the claim is false, then there is a first vertex $t \in T$ met during the execution of the algorithm for which after t is deleted some vertex $y \in V_t$ is not covered by E^*. For all children t^* of t the vertices in V_{t^*} are covered by E^*, so y must lie in λ_t. But then y is unique for t before t is deleted. Thus λ_t is added to E^* and y is covered by $\lambda_t \in E^*$ after t is deleted which is a contradiction. ■

From Claim 4.1.2 it follows directly that E^* is an edge cover of $\mathcal{H}$ at the end of the algorithm.

Claim 4.1.3. *At each point in time during the algorithm, I is an independent set in $\mathcal{H}$.*

Proof. Assume the claim is wrong. Then, there is a first vertex y that is added to I such that y is adjacent to x already in I. The vertex x was added to I, so there was $t \in T$ such that x was unique for t when t was considered by the algorithm. Thus x is in V_t, and consequently $x \notin \lambda_{t'}$ for any vertex $t' \in T \setminus V(\mathcal{T}_t)$. As x and y are adjacent, there must be a vertex $t^* \in V(\mathcal{T}_t)$ such that $\{x, y\} \subseteq \lambda_{t^*}$. But y is added to I after x and thus it must appear in $\lambda_{t'}$ for a vertex $t' \in T \setminus V(\mathcal{T}_t)$. Then because of the connectedness condition and the fact that $\mathcal{T}$ is a tree, y must also be in λ_t and thus is deleted from $\mathcal{H}$ when t is deleted. But then y cannot be added later which is a contradiction. ■

With Claim 4.1.2 and Claim 4.1.3 we have that at the end of the algorithm E^* is an edge cover of G and I is an independent set in G. From the algorithm it is obvious that $|E^*| = |I|$. This completes the proof. ■

Corollary 4.1.4. *Let $\mathcal{H} = (V, E)$ be an acyclic hypergraph and $S \subseteq V$. Then the following statements are true:*

a) The S-star size of $\mathcal{H}$ can be computed in polynomial time.

b) Let $\mathcal{H}' = (V', E')$ be an S-component of $\mathcal{H}$ and let k be the S-star size of $\mathcal{H}'$. There is a polynomial time algorithm that computes an edge set $E^* \subseteq E'$ that covers $S \cap V'$ and $|E^*| = k$.

Proof. a) Let $\mathcal{H}_1, \dots, \mathcal{H}_m$ be the S-components of $\mathcal{H}$ which can be compute in polynomial time by Observation 3.2.14. By Observation 3.2.15, each $\mathcal{H}_i = (V_i, E_i)$, $i \in \{1, \dots, m\}$, is acyclic and hence by Observation 3.2.1, $\mathcal{H}_i[S]$ is acyclic too. By Lemma 4.1.1, for each $i \in \{1, \dots, m\}$, one can determine the size of a maximum independent set I_i of $\mathcal{H}_i[S]$. The I_i of maximum size gives us the S-star size of $\mathcal{H}$.

b) We compute an edge cover $\tilde{E}$ of size k for $\mathcal{H}'[S]$ with Lemma 4.1.1. Then for each edge $\tilde{e} \in \tilde{E}$ one can easily find an edge $e \in E'$ with $\tilde{e} \subseteq e$. ■

4.2 GENERAL HYPERGRAPHS

In this section we consider the problem of computing the quantified star size of hypergraphs that have small width for the decomposition techniques defined in Section 2.3. Note that the computation of quantified star size is not strictly necessary for tractable counting. The algorithm of Chapter 5 does not need to compute the S-star size for graphs of width k but only for acyclic hypergraphs which we considered in the previous section. Still it is of course desirable to know the quantified star size of an instance before applying the counting algorithm, because quantified star size has an exponential influence on the runtime.

We show that for all decomposition techniques considered in this thesis the quantified star size can be computed rather efficiently, in time roughly $|V|^{O(k)}$ where k is the width of the input. For small values of k, this bound is reasonable. We then proceed by showing that, on the one hand, for some decomposition measures such as treewidth or hingetree width, the computation of quantified star size is even fixed parameter tractable parameterized by the width. On the other hand, we show that for decomposition measures above hypertree width it is unlikely that fixed parameter tractability can be obtained (under standard assumptions).

Instead of tackling quantified star size directly, we consider the combinatorially less complicated notion of independent sets as we did in Section 4.1. This is justified by the following observation:

Observation 4.2.1. *Let β be any decomposition technique considered in this thesis. Then, for every $k \in \mathbb{N}$, computing the S-star size of S-hypergraphs of β-width at most k polynomial time Turing-reduces to computing the size of a maximum independent set for hypergraphs of β-width at most k. Furthermore, there is a polynomial time many one reduction from computing the size of a maximum independent set in hypergraphs of β-width at most k to computing the S-star size of hypergraphs of β-width at most $k+1$.*

Proof. By definition computing S-starsize reduces to the computation of independent sets of S-components. S-components are induced subhypergraphs, so we get the first direction from Observation 3.2.2.

For the other direction let $\mathcal{H} = (V, E)$ be a hypergraph for which we want to compute the size of a maximum independent set. Let $x \notin V$. We construct the hypergraph $\mathcal{H}'$ of vertex set $V' = V \cup \{x\}$ and edge set $E' = \{e \cup \{x\} \mid e \in E\}$ and set $S := V$. The hypergraph is one single S-component, because x is in every edge. Furthermore, the S-starsize of $\mathcal{H}'$ is obviously the size of a maximum independent set in $\mathcal{H}$. It is easy to see that the construction increases the treewidth of the hypergraph by at most 1 and does not increase the β-width for all other decomposition considered here at all. ■

Because of Observation 4.2.1 we will not talk about S-star size in this section anymore but instead formulate everything with independent sets.

4.2.1 *Exact computation*

Proposition 4.2.2. *There is an algorithm that, given a hypergraph $\mathcal{H} = (V, E)$ and a generalized hypertree decomposition $(\mathcal{T}, (\lambda_t)_{t\in T}, (\chi_t)_{t\in T})$ of width k of $\mathcal{H}$, computes a maximum independent set of $\mathcal{H}$ in time $k|V|^{O(k)}$.*

Proof. We apply dynamic programming along the decomposition. Let $b = (\lambda, \chi)$ be a guarded block of $\mathcal{T}$. Let $\mathcal{T}_b$ be the subtree of $\mathcal{T}$ with b as its root. We set $V_b := \chi(\mathcal{T}_b)$. Observe that $I \subseteq V_b$ is independent in $\mathcal{H}$ if and only if it is independent in $\mathcal{H}[V_b]$ so we do not differentiate between the two notions. For each independent set $\sigma \subseteq \chi$ we will compute an independent set $I_{b,\sigma} \subseteq V_b$ that is maximum under the independent sets containing exactly the vertices σ from χ. Observe that because λ contains at most k edges that cover χ we have to compute at most kn^k independent sets $I_{b,\sigma}$ for each b.

If b is a leaf of $\mathcal{T}$, the construction of the $I_{b,\sigma}$ is straightforward and can certainly be done in time $k|V|^{O(k)}$.

Let now $b = (\lambda, \chi)$ be an inner vertex of $\mathcal{T}$ with children $b_1, \ldots, b_r$ and let $b_i = (\lambda_i, \chi_i)$. For each independent set $\sigma \subseteq \chi$ we do the following: For each i, let σ_i be an independent set of χ_i such that $\sigma \cap \chi \cap \chi_i = \sigma_i \cap \chi \cap \chi_i$ and $|I_{b_i,\sigma_i}|$ is maximal. We claim that we can set $I_{b,\sigma} := \sigma \cup I_{b_1,\sigma_1} \cup \ldots \cup I_{b_r,\sigma_r}$.

We first show that $I_{b,\sigma}$ defined this way is independent. Assume this is not true, then $I_{b,\sigma}$ contains x, y that are in one common edge e in $\mathcal{H}[V_b]$. But then x, y do not lie both in χ, because $I_{b,\sigma} \cap \chi = \sigma$ and σ is independent. By induction x, y do not lie in one V_{b_i} either. Assume that $x \in \chi$ and $y \in V_{b_i}$ for some i. Then certainly $x \notin V_{b_i}$ and $y \notin \chi$. But the edge e must lie in one block χ'. Because of the connectivity condition for y, the guarded block (λ', χ') must lie in the subtree with root b_i, which contradicts $x \in e$. Finally, assume that $x \in V_{b_i}$ and $y \in V_{b_j}$ for $i \neq j$ and $x, y \notin \chi$. Then x and y cannot be adjacent because of the connectivity condition. This shows that $I_{b,\sigma}$ is indeed independent.

Now assume that $I_{b,\sigma}$ is not of maximum size and let $J \subseteq V_b$ be an independent set with $|J| > |I_{b,\sigma}|$ and $J \cap \chi = \sigma$. Because J and $I_{b,\sigma}$ are fixed to σ on χ there must be a b_i such that $|J \cap V_{b_i}| > |I_{b_i,\sigma_i}|$. This contradicts the choice of σ_i. So $I_{b,\sigma}$ is indeed of maximum size.

Because each block has at most $k|V|^k$ independent sets, all computations can be done in time $k|V|^{O(k)}$. ■

4.2.2 *Parameterized complexity*

While the algorithm in the last section is nice in that it is a polynomial time algorithm for fixed k, it is somewhat unsatisfying for some decomposition techniques: If we can compute the decomposition quickly, we would ideally want to be able to compute the S-star size efficiently, too. Naturally we cannot expect a polynomial time algorithm independent of the width k for any of the decomposition techniques we consider, because computing maximum independent sets is NP-hard. Instead, we can hope for independent set to be at least fixed-parameter tractable with respect to k. We will show that for general hypertree width even this is unlikely, because independent set parameterized by generalized hypertree width is W[1]-hard. More positively, we will show that computing maximum independent sets is fixed-parameter tractable for some other decomposition techniques, in particular tree decompositions and hingetree decompositions.

Lemma 4.2.3. *Computing maximum independent sets on hypergraphs is* W[1]*-hard parameterized by generalized hypertree width.*

Proof. We will show a parameterized many-one reduction from the problem p-IndependentSet defined as follows:

p-IndependentSet
Input: a graph G, $k \in \mathbb{N}$.
Parameter: k.
Problem: Decide if G has an independent set of size k.

Because p-IndependentSet is well known to be W[1]-hard, this suffices to establish W[1]-hardness of independent sets on hypergraphs parameterized by generalized hypertree width.

So let $G = (V, E)$ be a graph and let k be a positive integer. We construct a hypergraph $\mathcal{H} = (V', E')$ in the following way: For each vertex v the hypergraph $\mathcal{H}$ has k vertices $v_1, \ldots, v_k$. For $i = 1, \ldots, k$ we have an edge $V_i := \{v_i \mid v \in V\}$ in E'. Furthermore, for each $v \in V$ we add an edge $H_v := \{v_i \mid i \in [k]\}$. Finally we add the edge sets $E_{ij} := \{v_i u_j \mid uv \in E\}$ for $i, j \in [k]$. $\mathcal{H}$ has no other vertices or edges. The construction is illustrated in Figure 14.

We claim that G has an independent set of size k if and only if $\mathcal{H}$ has an independent set of size k. Indeed, if G has an independent set $v^1, \ldots, v^k$, then $v^1_1, \ldots v^k_k$ is easily seen to be an independent

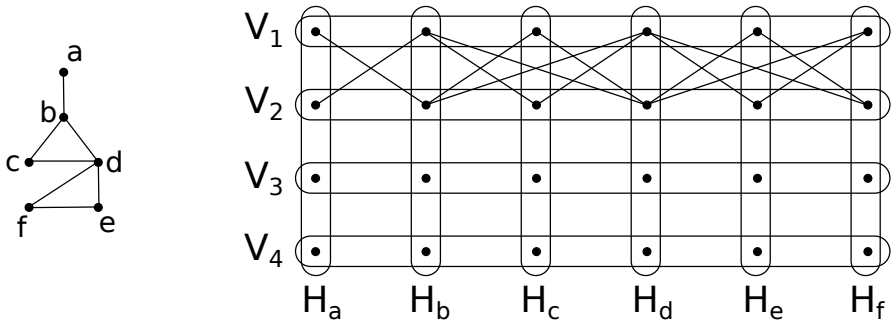

Figure 14: We illustrate the construction for Lemma 4.2.3 by an example. A graph G on the left with the associated hypergraph $\mathcal{H}$ for $k = 4$ on the right. To keep the illustration more transparent the edge sets E_{ij} are not shown except for $E_{1,2}$ and $E_{2,1}$.

set of size k in $\mathcal{H}$. Now assume that $\mathcal{H}$ has an independent set I of size k. Then for each $v \in I$ we can choose a vertex $\pi(v) \in V$ such that $v \in H_{\pi(v)}$. Furthermore for distinct $v, u \in I$ the corresponding vertices $\pi(v), \pi(u)$ have to be distinct, too, so $\pi(I) \subseteq V$ has size k. Finally, we claim that $\pi(I)$ is independent in G. Assume this is not true, then there are vertices $\pi(v), \pi(u)$ such that $\pi(v)\pi(u) \in E$. But then $vu \in E'$ by construction which is a contradiction. So, indeed G has an independent set of size k if and only if $\mathcal{H}$ has one.

From Observation 2.3.23 we get that $\mathcal{H}$ has generalized hypertree-width at most k, because $V_1, \ldots, V_k$ cover V'.

Observing that the construction of $\mathcal{H}$ from G can be done in time polynomial in $|V|$ and k completes the proof. ■

We start our fixed-parameter tractability results with an easy observation.

Proposition 4.2.4. *Given a hypergraph $\mathcal{H}$ computing a maximum independent set in $\mathcal{H}$ is fixed parameter tractable parameterized by the treewidth of $\mathcal{H}$.*

This can be seen either by applying an optimization version of Courcelle's Theorem (see Section 11.2) or by straightforward dynamic programming. Interestingly, one can show the same result also for bounded hingetree width, which is a decomposition technique in which the blocks are of unbounded size. This unbounded size makes the dynamic programming in the proof far more involved than for treewidth.

Proposition 4.2.5. *Given a hypergraph $\mathcal{H} = (V, E)$ of hingetree width k, a maximum independent set in $\mathcal{H}$ can be computed in time $k2^{k^2}|V|^{O(1)}$. It follows that independent set is fixed parameter tractable parameterized by hingetree width.*

Proof. First observe that minimum width hingetree decompositions can be computed in polynomial time by Lemma 2.3.32, so we simply assume that a decomposition is given in the rest of the proof.

The proof has some similarity with that of Proposition 4.2.2, so we use some notation from there. For guarded block (λ, χ) we will again compute maximum independent sets containing prescribed vertices. The difference is, that we can take these prescribed sets to be of size 1: because of the hingetree condition, only one vertex of a block may be reused in any independent set in the parent. The second idea is that we can use equivalence classes of vertices in the computation of independent sets in the considered guarded blocks, which limits the number of independent sets we have to consider. We now describe the computation in detail.

Let $\Xi = (\mathcal{T}, (\lambda_t)_{t\in T}, (\chi_t)_{t\in T})$ be a hingetree decomposition of $\mathcal{H}$ of width k. Let $b = (\lambda, \chi)$ be a guarded block of Ξ and let $b' = (\lambda', \chi')$ be its parent. As before, let $\mathcal{T}_b$ be the subtree of $\mathcal{T}$ with b as its root and $V_b := \chi(\mathcal{T}_b)$. Set $\mathcal{H}_b := (V_b, E_b)$ with $E_b := \bigcup \lambda^*$ with the union being over all guarded blocks in $\mathcal{T}_b$. The main idea is to iteratively compute, for all vertices $v \in \chi' \cap \chi$, a maximum independent set $J_{v,b}$ in $\mathcal{H}_b = (V_b, E_b)$ containing v. Furthermore, we also compute an independent set $J_{\emptyset,b}$ that contains no vertices of $\chi' \cap \chi$. Note that, since $\chi \subseteq \bigcup_{e\in\lambda} e$, there are no isolated vertices in χ and the size of a maximum independent set is bounded by k in each block.

For a node $b = (\lambda, \chi)$, we organize the vertices in χ into at most 2^k equivalence classes by defining v and u to be equivalent if they lie in the same subset of edges of λ. The equivalence class of v is denoted by $\mathbf{c}(v)$. For each class, a representant is fixed. We denote by $\bar{v}$, the representant of the equivalence class of v and by $\bar{\chi} \subseteq \chi$, the restriction of χ on these at most 2^k representants.

Let first b be a leaf. We first compute independent sets on $\bar{\chi}$. Observe that the independent sets are invariant under the choice of representants. For each equivalence class $\mathbf{c}(v)$, we compute $J_{\bar{v},b} \subseteq \bar{\chi}$ as a maximum independent set containing $\bar{v}$. Computing the classes and a choice of maximum independent sets containing each $\bar{v}$ can be done in time $k2^{k^2}$ because independent sets cannot be bigger than k. Clearly, $J_{v,b}$, a maximum independent set containing v, can be easily computed from the set $J_{\bar{v},b}$. Thus, one can compute all the $J_{v,b}$ in time $k2^{k^2}n$. The computation of $J_{\emptyset,b}$ can be done on representants, too, by simply excluding the vertices from $\chi' \cap \chi$.

Let b now be an inner vertex and $b_1, b_2, ..., b_m$ be its children with $b_i = (\lambda_i, \chi_i)$, $i \in [m]$. We again consider equivalence classes on χ. Fix $v \in \chi$ and compute the list $L_{\bar{v},b}$ of all independent sets $\sigma \subseteq \bar{\chi}$ containing $\bar{v}$. Fix now $\sigma \in L_{\bar{v},b}$. We first compute a set $J^{\sigma}_{v,b}$ as a maximum independent set of $\mathcal{H}_b$ containing v and whose vertices in χ have the representants σ. We will distinguish for a given vertex $\bar{u} \in \sigma$ if it is the representant of a vertex belonging to the block of some (or several)

children of b or if it represents vertices of $\chi \setminus (\bigcup_{i=1}^{m} \chi_i)$ only. Therefore we partition σ into σ', σ'' accordingly:

- $\sigma := \sigma' \cup \sigma''$
- $\sigma' := \bar{\chi} \cap \{\bar{u} \mid u \in \bigcup_{i=1}^{m} \chi_i\}$.
- $\sigma'' := \bar{\chi} \setminus \{\bar{u} \mid u \in \bigcup_{i=1}^{m} \chi_i\}$

Set $\sigma' := \{\bar{u}_1, ..., \bar{u}_h\}$ with $h \leq m$. Let us examine the consequences of $\mathcal{T}$ being a hingetree decomposition. We have that, for all $i \in [m]$, there exists $e_i \in \lambda$, such that $\chi \cap \chi_i \subseteq e_i$. Thus, since σ is an independent set in $\bar{\chi} \subseteq \chi$, at most one vertex in σ' is a representant of a vertex in χ_i. Thus

$$\forall u \neq u' \in \sigma \colon \chi_i \cap \mathbf{c}(u) = \emptyset \vee \chi_i \cap \mathbf{c}(u') = \emptyset. \tag{1}$$

We denote by $S_i = \{j \mid \mathbf{c}(u_i) \cap \chi_j \neq \emptyset\}$ and by $S = [m] \setminus \bigcup S_i$. By (1) the sets $S_1, ..., S_h, S$ form a partition of $[m]$. To construct $J^{\sigma}_{v,b}$, we now determine for each $i \leq h$, which vertex u of $\mathbf{c}(u_i)$ can contribute the most, by taking the union of all the maximum independent sets J_{u,b_j}, $j \in S_i$, it induces at the level of the children of b.

For each fixed $u \in \mathbf{c}(u_i)$, let

$$I_{i,u} = \{u\} \cup \bigcup_{j \in S_i} J_{u,b_j},$$

where we set $J_{u,b_j} := J_{\emptyset,b_j}$ if $u \notin \chi_j$. Let then $I_i = I_{i,u}$ for some $u \in \mathbf{c}(u_i)$ for which the size of $I_{i,u}$ is maximal.

The set $J^{\sigma}_{v,b}$ is now obtained as follows depending on whether $\bar{v} \in \sigma''$ or $\bar{v} \in \sigma'$. If $\bar{v} \in \sigma''$, we claim that $J^{\sigma}_{v,b}$ can be chosen as

$$J^{\sigma}_{v,b} := \{v\} \cup (\sigma'' \setminus \{\bar{v}\}) \cup \bigcup_{i=1}^{h} I_i \cup \bigcup_{i \in S} J_{\emptyset,b_i}.$$

If $\bar{v} \in \sigma'$, say $\bar{v} = u_1$, we claim that $J^{\sigma}_{v,b}$ can be chosen as

$$J^{\sigma}_{v,b} := \sigma'' \cup \bigcup_{j \in S_1 : v \in \chi_j} J_{v,b_j} \cup \bigcup_{j \in S_1 : v \notin \chi_j} J_{\emptyset,b_j} \cup \bigcup_{i=2}^{h} I_i \cup \bigcup_{i \in S} J_{\emptyset,b_i}.$$

The set $J_{v,b}$ is taken as one of the sets $J^{\sigma}_{v,b}$ of maximal size for a $\sigma \in L_{v,b}$. To compute $J_{\emptyset,b}$, the arguments are similar.

We first show that all $J_{v,b}$ are indeed independent sets in $\mathcal{H}_b$. Clearly, it is enough to prove this for any $J^{\sigma}_{v,b}$. There will be no reason to distinguish whether $\bar{v} \in \sigma''$ or $\bar{v} \in \sigma'$, because our arguments will apply to all $J^{\sigma}_{v,b}$ independent of the choice of a distinguished element v. We will make extensive use of the two following facts.

- Let $j, j' \in [m]$ and $I \subseteq V_{b_j}, I' \subseteq V_{b_{j'}}$ independent sets of $\mathcal{H}_{b_j}$ and $\mathcal{H}_{b'_j}$ respectively. By the connectivity condition for tree decomposition we have
$$I \cap I' \subseteq \chi_j \cap \chi_{j'} \cap \chi.$$
This permits to investigate the intersection of two independent sets I, I' by looking at their restriction on χ.

- Let now $I \subseteq V_{b_j}$ be an independent set of $\mathcal{H}_{b_j}$. Then, I remains an independent set in $\mathcal{H}_b$. Indeed, suppose there is a $e \in E_b \setminus E_{b_j}$ containing two vertices $y_1, y_2 \in I$. Since all edges must belong to a guard, there exists a node $b^* = (\lambda^*, \chi^*)$ such that $e \in \lambda^*$. Then, since in a hingetree decomposition we have $\chi^* = \bigcup \lambda^*$, then $\{y_1, y_2\} \subseteq e \subseteq \chi^*$. But then, by the connectivity condition it follows that $\{y_1, y_2\} \subseteq \chi$. Hence, by the intersection property of hingetree decomposition, there exists $e_j \in \chi_j$ such that
$$\{y_1, y_2\} \subseteq \chi \cap \chi_j \cap e_j$$
which implies that y_1 and y_2 are adjacent in $\mathcal{H}_{b_j}$. Contradiction.

We now start the proof that $J^{\sigma}_{v,b}$ is independent incrementally. Let $i \in [h]$, $u \in \mathbf{c}(u_i)$ and $j \in S_i$ and consider the set $I := J_{u,b_j}$. By induction, the set I is independent in $\mathcal{H}_{b_j}$. By the hingetree condition, there exists $e_j \in \lambda_j$ such that $\chi \cap \chi_j \subseteq e_j$. By the connectivity condition, this implies $\chi \cap I \subseteq e_j$. Then, since I is an independent set, no two vertices of χ can belong to I i.e., $|\chi \cap I| \leq 1$. The connectivity condition also implies that, for $j' \neq j$, $V_{b_{j'}} \cap I \subseteq \chi \cap \chi_j$, hence $|V_{b_{j'}} \cap I| \leq 1$ and I is an independent set of $\mathcal{H}_b$. Finally, the set $I_i = \bigcup_{j \in S_i} J_{u,b_j}$ is also an independent set of $\mathcal{H}_b$, since for any distinct $j, j' \in S_i$:
$$J_{u,b_j} \cap J_{u,b_{j'}} \subseteq \chi_j \cap \chi_{j'} \cap \chi \subseteq e_j.$$
Hence $J_{u,b_j} \cap J_{u,b_{j'}}$ contains at most one vertex (which is in χ and could then only be u).

Let now $i, i' \in [m]$ be distinct. By the arguments above, I_i (resp. $I_{i'}$) contains at most one element u (resp. u') such that $u \in \mathbf{c}(u_i)$ (resp. $u' \in \mathbf{c}(u_{i'})$). By Equation 1, we have that the two classes are distinct and that $u_i \neq u_{i'}$. But $u_i, u_{i'} \in \sigma$ and σ is independent in χ. Hence, $u_i, u_{i'}$ cannot be adjacent in $\mathcal{H}_b$. Consequently,
$$\bigcup_{i=1}^{h} I_i$$
is an independent set in $\mathcal{H}_b$.

Let $j \in S$. $J_{\emptyset,b_j}$ is independent in $\mathcal{H}_{b_j}$ and $J_{\emptyset,b_j} \subseteq V_{b_j} \setminus \chi$. Hence, $J_{\emptyset,b_j}$ is independent in $\mathcal{H}_b$. This also implies that, given $j' \in [m]$ distinct from j, $J_{\emptyset,b_j} \cap V_{b_{j'}} = \emptyset$. Thus,
$$\bigcup_{i=1}^{h} I_i \cup \bigcup_{i \in S} J_{\emptyset,b_i}.$$

is independent in $\mathcal{H}_b$.

Finally, by construction, for all $i \in [h]$, $I_i \cap \chi = \{u\}$ with $\bar{u} = \bar{u}_i \in \sigma'$. Also $\sigma = \sigma' \cup \sigma''$ is independent in χ hence in $\mathcal{H}_b$. No vertices $y_1 \in I_i$ and $y_2 \in \sigma''$ can be adjacent because, again, this would imply that $\{y_1, y_2\} \subseteq \chi$ and contradict the fact that $\bar{y}_1, \bar{y}_2$ are independent in σ. Thus $J^{\sigma}_{v,b}$ is independent.

We now prove that $J_{v,b}$ is of maximum size. Observe that it suffices to show this again for each $J^{\sigma}_{v,b}$. Each maximum independent set J of $\mathcal{H}_b$ that contains v and whose vertices in χ have exactly the representants σ can be expressed as $\tau \cup J_1 \cup J_2 \cup ... \cup J_m$. Here $\tau \subseteq \chi$ is an independent set of b containing v and whose representants are σ. Furthermore, J_i is an independent set of $\mathcal{H}_b$ that contains only vertices of V_{b_i}. The set J_i may only contain one vertex u_i from $\chi \cap \chi_i$. But then exchanging J_i for J_{u_i,b_i} may only increase the size of the independent set, so we can assume that I has the form $\tau \cup J_{u_1,b_i} \cup J_{u_2,b_2} \cup \ldots \cup J_{u_m,b_m}$ where u_i may also stand for $\emptyset$.

Assume now that $J^{\sigma}_{v,b}$ is not maximum, i.e., there is an independent set J containing v whose vertices in χ have the representants σ and J is bigger than $J^{\sigma}_{v,b}$. Then one of four following things must happen:

- There is an i such that $v \in \chi_i$ and $J \cap V_{b_i}$ is bigger than J_{v,b_i}. But this case cannot occur by induction.
- $v = u_1$ and there is a $j \in S_1$ such that $v \notin \chi_j$ and $|J \cap V_{b_j}| > |J_{\emptyset,b_j}|$. By induction we know that $J_{\emptyset,b_j}$ is optimal under all independent sets of $\mathcal{H}_{b_j}$ not containing any vertex of $\chi_j \cap \chi$, so there must be a vertex $u \in J \cap \chi \cap \chi_j$. Since J is independent, v and u share no edge in λ and then $\bar{v} \neq \bar{u}$. Since $j \in S_1$, it holds that $\mathbf{c}(v) \cap \chi_j \neq \emptyset$ and by Equation 1, $\mathbf{c}(u) \cap \chi_j = \emptyset$. Contradiction.
- There is an $i \in S$ such that $J \cap V_{b_i}$ is bigger than $J_{\emptyset,b_i}$. But from $i \in S$ it follows by definition that $\chi \cap \chi_i \cap J = \emptyset$, so this case can not occur by induction, either.
- There is an $i \in [h]$ such that $|J \cap (\bigcup_{j \in S_i} V_j)| > |I_i|$. We claim that $(\bigcup_{j \in S_i} \chi_j) \cap \chi \cap J$ contains only one vertex. Assume there are two such vertices x and y. By definition, $\bar{x}, \bar{y} \in \bar{\tau}$. Since J is independent, $\bar{x}$ and $\bar{y}$ are not adjacent in $\bar{\chi}$ and $\bar{x} \neq \bar{y}$. At least one of these, say y, must be in $\mathbf{c}(u_i)$, because $\bar{u}_i \in \bar{\tau}$ by definition. Let $x \in V_{j'}$ with $j' \in S_i$, then there is a vertex $w \in \mathbf{c}(u_i) = \mathbf{c}(y)$ in $\chi_{j'} \cap \chi \subseteq e_j$ by definition of S_i. But then $\bar{x}$ and $\bar{y}$ are adjacent in $\bar{\chi}$ which is a contradiction.

  So there is exactly one vertex u in $(\bigcup_{j \in S_i} \chi_j) \cap \chi \cap J$. But then $|J \cap (\bigcup_{j \in S_i} V_j)| > I_{i,u}$. Thus either there must be a $j \in S_i$ with $u \in V_j$ such that $|J \cap V_j| > |J_{u,b_j}|$ or there must be a $j \in S_i$ with $u \notin V_j$ such that $|J \cap V_j| > |J_{\emptyset,b_j}|$. The former clearly contradicts the optimality of J_{u,b_j}, while the latter leads to a contradiction completely analogously to the second item above.

Because only $k2^{k^2}n^2$ sets have to be considered for each guarded block, this results in an algorithm with runtime $k2^{k^2}|V|^{O(1)}$. ■

4.2.3 *Approximation*

We have seen that computing maximum independent sets of hypergraphs with decompositions of width k can be done in polynomial time for fixed width k and that for some decompositions it is even fixed parameter tractable with respect to k. Still, the exponential influence of k is troubling. In this section we will show that we can get rid of it if we are willing to sacrifice the optimality of the solution. We give a polynomial time k-approximation algorithm for computing maximum independent sets of graphs with generalized hypertree width k assuming that a decomposition is given. We start by formulating a lemma.

Lemma 4.2.6. *Let $\mathcal{H} = (V, E)$ be a hypergraph with a generalized hypertree decomposition $\Xi = (\mathcal{T}, (\lambda_t)_{t\in T}, (\chi_t)_{t\in T})$ of width k. Let $\mathcal{H}' = (V, E')$ where $E' := \{\chi_t \mid t \in T\}$. Let ℓ be the size of a maximum independent set in $\mathcal{H}$ and let ℓ' be the size of a maximum independent set in $\mathcal{H}'$. Then*

$$\frac{\ell}{k} \leq \ell' \leq \ell.$$

Before we prove Lemma 4.2.6 we will show how to get the approximation algorithm from it.

Observation 4.2.7. *Every independent set of $\mathcal{H}'$ is also an independent set of $\mathcal{H}$.*

Proof. Each pair of independent vertices x, y in $\mathcal{H}'$ is by definition in different blocks χ_t in $\mathcal{H}$. For each edge $e \in E$ there must (by definition of generalized hypertree decompositions) be a block χ such than $e \subseteq \chi$. Thus no edge $e \in E$ can contain both x and y, so x and y are independent in $\mathcal{H}$ as well. ■

Corollary 4.2.8. *There is a polynomial time algorithm that given a hypergraph $\mathcal{H}$ and a generalized hypertree decomposition of width k computes an independent set of size ℓ of $\mathcal{H}$ such that $|I| \geq \frac{\ell}{k}$ where ℓ is the size of a maximum independent set of $\mathcal{H}$.*

Proof. Observe that $\mathcal{H}'$ is acyclic by Lemma 2.3.27. By Lemma 4.1.1, we compute in polynomial time a maximum independent set I of $\mathcal{H}'$ whose size by Lemma 4.2.6 only differs by a factor $\frac{1}{k}$ from ℓ. By Observation 4.2.7, we know that I is also an independent set of $\mathcal{H}$. ■

Proof of Lemma 4.2.6. The second inequality follows directly from Observation 4.2.7.

For the first inequality consider a maximum independent set I of $\mathcal{H}$. Observe that a set I' is an independent set of $\mathcal{H}'$ if and only if it is an independent set of its primal graph $\mathcal{H}'_p$, so it suffices to show the same result for $\mathcal{H}'_p$.

Claim 4.2.9. *The graph $\mathcal{H}'_p[I]$ has treewidth at most $k-1$.*

Proof. First observe that vertices v that appear in no edge $e \in E$ change neither the treewidth nor the generalized hypertree width of a graph or hypergraph. Thus we assume that every vertex $v \in V$ is in at least one edge $e \in E$.

We construct a tree decomposition $(\mathcal{T}', (\chi'_t)_{t\in T'})$ of $\mathcal{H}'_p[I]$ from Ξ as follows: We set $\mathcal{T}' := \mathcal{T}[T']$ where $T' := \{t \in T \mid \chi_t \cap I \neq \emptyset\}$. Furthermore, $\chi'_t := \chi_t \cap I$ for $t \in T'$. For every $v \in I$ there is an edge $e \in E$ and $t \in T$ such that $v \in e \subseteq \chi_t$ and thus $v \in \chi'_t$. It follows that the bags χ'_t cover I. Moreover, the connectivity condition for I is satisfied, because it is satisfied for Ξ. Finally, for each edge uv in $\mathcal{H}'_p[I]$ there is a guarded block (λ_t, χ_t) such that $u, v \in \chi_t$ and thus $u, v \in \chi'_t$. Hence, $(\mathcal{T}', (\chi'_t)_{t\in T'})$ is indeed a tree decomposition.

Thus we only have to show $|\chi'_t| \leq k$. To see this, observe that for each t the bag $\chi'_t \subseteq \chi_t$ is covered by λ_t. But the vertices in $\chi'_t \subseteq I$ are independent in $\mathcal{H}$ and thus each $e \in \lambda_t$ can contain only a single vertex from χ'_t. Thus $|\chi'_t| \leq |\lambda_t| \leq k$. ■

Claim 4.2.10. *The graph $\mathcal{H}'_p[I]$ has an independent set I' of size at least $\frac{|I|}{k}$.*

Proof. From Claim 4.2.9 it follows with Lemma 2.3.6 that $\mathcal{H}'[I]$ and all of its induced subgraphs have a vertex of degree at most k. We construct I' iteratively by choosing a vertex of minimum degree and deleting it and its neighbors from the graph. In each round we delete at most k vertices, so we can choose a vertex in at least $\frac{|I|}{k}$ rounds. Obviously the chosen vertices are independent. ■

Every independent set of $\mathcal{H}'_p[I]$ is also an independent set of $\mathcal{H}'_p$ which completes the proof of Lemma 4.2.6. ■

5

QUANTIFIED STAR SIZE IS SUFFICIENT AND NECESSARY FOR EFFICIENT COUNTING

5.1 BOUNDED QUANTIFIED STAR SIZE IS NECESSARY

In this section we will show that bounded quantified star size is a necessary restriction for tractable #CQ: Under the assumption FPT $\neq$ #W[1], all classes $\mathcal{G}$ of S-hypergraphs for which p-#CQ is fixed-parameter tractable must have bounded quantified star size. As polynomial time tractability trivially implies fixed-parameter tractability, it follows that bounded quantified star size must also be necessary for classes $\mathcal{G}$ of S-hypergraphs that allow polynomial time algorithms.

Let $\mathcal{G}$ be a class of S-hypergraphs. Remember that by #CQ *on* $\mathcal{G}$ we denote the restriction of #CQ to instances whose associated S-hypergraph is in $\mathcal{G}$. Analogously, by p-#CQ *on* $\mathcal{G}$ we denote the restriction of p-#CQ to instances whose associated S-hypergraph is in $\mathcal{G}$.

We will use the fact that #CQ is already hard for very restricted S-hypergraphs, namely those of the queries from the class $\mathcal{C}_{\text{star}}$ from Lemma 3.1.4.

Theorem 5.1.1. *Assume* FPT $\neq$ #W[1]*, and let* $\mathcal{G}$ *be a recursively enumerable class of* S*-hypergraphs. If* p-#CQ *is fixed-parameter tractable for* $\mathcal{G}$*, then* $\mathcal{G}$ *is of bounded* S*-star size.*

Before proving Theorem 5.1.1, let us take some time to discuss its assumptions, because we will see these and similar assumptions throughout the rest of this thesis. First of all, the reader might feel that it would be more satisfying to prove a version of this theorem not under the assumption FPT $\neq$ #W[1] from parameterized complexity but instead to prove it based on a more standard assumption like FP $\neq$ #P. Clearly, the statement would then have to change from "fixed-parameter tractable" to "polynomial time tractable", but this could still be preferable. Unfortunately, it is unlikely that such a version of Theorem 5.1.1 can be proved. We will see in Section 5.3 that assuming FP $\neq$ #P there are classes of S-graphs on which #CQ is neither in FP nor #P-complete. Thus is seems unlikely that the theory of #P-completeness suffices to identify the classes of S-hypergraphs on which #CQ is tractable.

Furthermore, let us remark that if the reader feels uncomfortable with parameterized complexity, he can safely exchange the assumption FPT $\neq$ #W[1] against the so-called *exponential time hypothesis* which is the following conjecture.

Conjecture 1 (Exponential time hypothesis). 3-SAT *cannot be solved in time* $2^{o(n)}$ *where* n *is the number of variables of the input.*

The exponential time hypothesis implies FPT $\neq$ W[1] [DF99, Chapter 17] and thus also FPT $\neq$ #W[1]. Hence Theorem 5.1.1 and several other results of this thesis could also be formulated with the assumption that the exponential time hypothesis is true if the reader prefers an assumption from more classical complexity theory.

The other assumption that the reader might feel uncomfortable with is the recursive enumerability of $\mathcal{G}$. We will see that in the proof it will play an important role, but still in the formulation of the theorem it looks slightly out of place. One way of getting rid of this condition is assuming a non-uniform version of FPT $\neq$ #W[1] in the formulation of the theorem. Also we argue that recursive enumerability is not a strong restriction of $\mathcal{G}$. After all, from a practical perspective non-recursively enumerable classes of hypergraphs (and thus queries) are not interesting at all. Assume that there is such a class $\mathcal{G}$ that allows fixed-parameter tractable counting but has unbounded S-star size. Then each recursively enumerable class $\mathcal{G}'$ of $\mathcal{G}$ would by Theorem 5.1.1 be of bounded S-star size. Thus, the unbounded S-star size of $\mathcal{G}$ would crucially depend on the uncomputability, so it can safely be considered as a degenerate case of no practical interest.

We will use the following lemma to prove Theorem 5.1.1.

Lemma 5.1.2. *Let* $\mathcal{G}$ *be a recursively enumerable class of* S*-hypergraphs of unbounded* S*-star size. Then p-#CQ on* $\mathcal{G}$ *is* #W[1]*-hard.*

Let $\mathcal{G}_{\text{star}}$ be the class of S-graphs (G_n, S_n) where G_n is the star with n leaves and S_n consists of all vertices but the center of G_n. Note that the S-hypergraphs of the queries $\mathcal{C}_{\text{star}}$ from Lemma 3.1.4 are the S-graphs in $\mathcal{G}_{\text{star}}$ (see Example 3.2.7). Since by Lemma 3.1.4 #CQ restricted to instances with queries in $\mathcal{C}_{\text{star}}$ is #W[1]-hard, it follows directly that #CQ on $\mathcal{G}_{\text{star}}$ is #W[1]-hard. The idea of the proof of Lemma 5.1.2 is to show that $\mathcal{G}_{\text{star}}$ can be embedded in an appropriate way into *any* class $\mathcal{G}$ of S-hypergraphs of unbounded S-star size to show that #CQ on $\mathcal{G}$ is #W[1]-hard.

We feel that it is more transparent to show Lemma 5.1.2 for the restricted case of S-connected S-hypergraphs and to sketch afterwards how to generalize the proof to the general case. Remember that an S-hypergraph $(\mathcal{H}, S)$ is called S-connected if for every pair of vertices x, y there is a path $x = v_1, v_2, \ldots, v_{k-1}, v_k = y$ such that $v_i \notin S$ for $i \notin \{1, k\}$.

Lemma 5.1.3. *Let* $\mathcal{G}$ *be a recursively enumerable class of* S*-hypergraphs of unbounded* S*-star size. Then p-#CQ on* $\mathcal{G}$ *is* #W[1]*-hard.*

Proof of Lemma 5.1.3. Let $\mathcal{G}$ be a class of S-connected S-hypergraphs of unbounded S-star size.

Remember that we defined $\mathcal{C}_{\text{star}} := \{\phi_{\text{star},n} \mid n \in \mathbb{N}\}$ with $\phi_{\text{star},n} = \exists z \bigwedge_{i\in[n]} \mathcal{R}_i(z, y_i)$ (see Lemma 3.1.4). We will show a parameterized parsimonious reduction from p-#CQ, restricted to instances that have queries in $\mathcal{C}_{\text{star}}$, to p-#CQ on $\mathcal{G}$. As p-#CQ on the former class of instances is #W[1]-hard by Lemma 3.1.4, the claim will follow.

Let $\Phi = (\mathcal{A}, \phi)$ be an instance of #CQ restricted to queries in $\mathcal{C}_{\text{star}}$, i.e., ϕ has the form $\phi = \exists z \bigwedge_{i=1}^{k} \mathcal{R}_i(z, y_i)$. Because $\mathcal{G}$ is recursively enumerable and of unbounded S-star size, there is a computable function $g : \mathbb{N} \to \mathbb{N}$ such that for given $k \in \mathbb{N}$ one can in time $g(k)$ compute an S-connected S-hypergraph $(\mathcal{H}, S) \in \mathcal{G}$ of S-star size at least k. We will embed Φ into $\mathcal{H} = (V, E)$ to construct a #CQ-instance $\Psi := (\mathcal{B}, \psi)$ of size at most $g(k)\|\Phi\|^2$. The instance Ψ will have the S-hypergraph $(\mathcal{H}, S)$ and the same domain $B := A$ as Φ.

For each $e \in E$ let ψ_e be an atom with the relation symbol $\mathcal{E}_e$ and the set of variables $\mathsf{var}(\psi_e) = e$. Let

$$\psi' := \bigwedge_{e\in E} \psi_e,$$

then ψ is the query we get from ψ' by existentially quantifying all variables in $V \setminus S$. This completes the construction of the query ψ.

We now construct the structure $\mathcal{B}$. Let $Y = \{y_1, \ldots, y_k\} \subseteq S$ be a set of independent vertices. Such a set Y must exist, because $(\mathcal{H}, S)$ has S-star size at least k. Let d be an arbitrary but fixed element of A. We define $\mathcal{E}_e^{\mathcal{B}}$ depending on the vertices in e as follows:

Case 1: Let first $e \in E$ be an edge that contains y_i for some $i \in [k]$. Observe that y_i is uniquely determined, because no two of the vertices y_i share an edge. The atom ψ_e has the relation symbol $\mathcal{E}_e$ and as variables the vertices of e. We assume that the order of the variables in ψ_e is as follows: y_i is the first variable, followed by the other variables in $e \cap S$ and after those the variables in $e \setminus S$. We define $\mathcal{E}_e^{\mathcal{B}} := \{(v_2, d, \ldots, d, v_1, \ldots, v_1) \mid (v_1, v_2) \in \mathcal{R}_i^{\mathcal{A}}\}$, where $\mathcal{R}_i^{\mathcal{A}}$ is the relation of $\mathcal{R}_i$ in $\mathcal{A}$. Observe that this forces all variables in $(e \cap S) \setminus \{y_i\}$ to be equal to the value d in satisfying assignments, while the variables in $e \setminus S$ must all have a common value v_1. Furthermore, because y_i is uniquely determined, the relation $\mathcal{E}_e^{\mathcal{B}}$ is well defined.

Case 2: Let now $e \in E$ with $e \cap Y = \emptyset$. We assume that the variables in ψ_e are ordered such that all variables in $S \cap e$ appear before those in $e \setminus S$. Then we define $\mathcal{E}_e^{\mathcal{B}} := \{(d, \ldots, d, v_1, \ldots, v_1) \mid v_1 \in A\}$. Again in the satisfying assignments all variables in $e \cap S$ are forced to be equal to d, while the variables in $e \setminus S$ can take an arbitrary but equal value.

This completes the construction of $\mathcal{B}$ and thus that of $\Psi = (\mathcal{B}, \psi)$.

Claim 5.1.4. $|\phi(\mathcal{A})| = |\psi(\mathcal{B})|$.

Proof. Let ϕ', resp., ψ' be the quantifier free queries we get from ϕ, resp., ψ by deleting all quantifiers.

We construct a function B that to an assignment from $\phi'(\mathcal{A})$ constructs an assignment $B(a) := a'$ with $a' : V \to B$. We define

$$a'(x) := \begin{cases} a(x), & x \in Y, \\ d, & x \in S \setminus Y, \\ a(z), & x \in V \setminus Y \end{cases}.$$

We claim that B is a bijection from $\phi(\mathcal{A})$ to $\psi(\mathcal{B})$. It is easily seen from the construction of ψ that a' satisfies all atoms of ψ and thus $a' \in \psi'(\mathcal{B})$. Furthermore, B is obviously injective. Thus it only remains to prove that B is surjective. To see this, consider $b' \in \psi(\mathcal{B})$. By construction of Ψ, we have $b'(x) = d$ for all $x \in S \setminus Y$. Because $\mathcal{H}$ is S-connected, we have that $\mathcal{H}[V \setminus S]$ is connected. From the construction of Ψ it follows by an easy induction that there is a $v_1 \in A$ such that $b'(x) = v_1$ for all $x \in V \setminus S$. We construct an assignment $b : \mathsf{var}(\phi) \to A$ by $b(x) := b'(x)$ for $x \in \{y_1, \dots, y_k\}$ and $b(z) := v_1$. Obviously, $B(b) = b'$. Moreover, from the construction of Ψ is follows that $b \in \phi'(\mathcal{A})$. Thus B is a bijection from $\phi'(\mathcal{A})$ to $\psi'(\mathcal{B})$.

We now construct a mapping B' from $\phi(\mathcal{A})$ to $\psi(\mathcal{B})$ as follows: For $a \in \phi'(\mathcal{A})$ we map $a|_{\mathsf{free}(\phi)}$ to $B(a)|_{\mathsf{free}(\psi)}$. Since B is a bijection, it follows that B' is a bijection as well. This proves the claim. ■

Obviously, the S-hypergraph associated to ψ is $(\mathcal{H}, S)$. Moreover, by construction we have $|\psi| \leq g(k)$ and Ψ can be constructed in time at most $g(k)\|\Phi\|^2$, because $\mathcal{H}$ has size at most $g(k)$ and the size of the relations is bounded by $|A|^2$. Thus, with Claim 5.1.4, the construction of Ψ form Φ is a parameterized parsimonious reduction. This completes the proof of Lemma 5.1.3 ■

We now sketch how to extend Lemma 5.1.3 from S-connected S-hypergraphs to general S-hypergraphs in a straightforward way.

Proof of Lemma 5.1.2 (Sketch). The proof follows the same ideas as that of Lemma 5.1.3: We first compute an S-hypergraph $\mathcal{H}$ in $\mathcal{G}$ of S-star size at least k. Then we choose an S-component $\mathcal{H}'$ of S-star size at least k in $\mathcal{G}$. We construct the relations $\mathcal{E}_e^{\mathcal{B}}$ in such a way that in every satisfying assignment every variable not in $\mathcal{H}$ is forced to the value d. For all other variables we construct the relations as in the proof of Lemma 5.1.3. Since $\mathcal{H}'$ is S-connected by Observation 3.2.16, the same arguments as in the proof of Lemma 5.1.3 show that the construction is a parsimonious parameterized reduction. ■

Proof of Theorem 5.1.1. Assume that p-#CQ on $\mathcal{G}$ is fixed-parameter tractable. By Lemma 5.1.2 we directly get $\mathsf{FPT} = \#\mathsf{W}[1]$ which contradicts the assumption. ■

5.2 THE COMPLEXITY OF COUNTING

In this section we show that the decomposition techniques introduced in Section 2.3 lead to efficient counting when combined with bounded quantified star size. We proceed with the following rather technical lemma.

Lemma 5.2.1. *There is an algorithm that, given a* CQ*-instance* $\Phi = (\mathcal{A}, \phi)$ *of quantified starsize* ℓ *and a generalized hypertree decomposition* $\Xi = (\mathcal{T}, (\lambda_t)_{t\in T}, (\chi_t)_{t\in T})$ *of* Φ *of width* k*, constructs a* CQ*-instance* $\Psi = (\mathcal{B}, \psi)$ *in time* $\|\Phi\|^{p(k,\ell)}$ *for a fixed polynomial* p *such that*

- Φ *and* Ψ *are solution equivalent,*
- Ψ *is acyclic, and*
- ψ *is quantifier free.*

Proof. Given $\Phi = (\mathcal{A}, \phi)$, we construct Ψ in several steps.

Let $\mathcal{H} = (V, E)$ be the hypergraph of ϕ. Let $V_1, \dots, V_m$ be the vertex sets of the connected components of $\mathcal{H}[V \setminus S]$ and let $V'_1, \dots, V'_m$ be the vertex sets of the corresponding S-components of $\mathcal{H}$. Clearly, we have $V_i \subseteq V'_i$ and $V'_i \setminus V_i = V'_i \cap S =: S_i$. Let Φ_i be the CQ-instance whose query ϕ_i is obtained by restricting all atoms of ϕ to the variables in V'_i and whose structure $\mathcal{A}_i$ is obtained by projecting all relations of $\mathcal{A}$ accordingly. The associated hypergraph of ϕ_i is $\mathcal{H}[V'_i]$. Moreover, $\mathcal{H}[V'_i]$ has a generalized hypertree decomposition Ξ_i of width at most k with tree a $\mathcal{T}_i$ that is a subtree of $\mathcal{T}$ (see Observation 3.2.2).

Now fix i. To Φ_i we construct a solution equivalent ACQ-instance $\Phi'_i = (\mathcal{A}'_i, \phi'_i)$ as in the proof of Lemma 2.3.28: For each $t \in T$ we construct an atom ϕ_t in the variables χ_t. The associated relation is given by

$$\pi_{\chi_t}\left(\bowtie_{\substack{\phi'\in\mathsf{atom}(\phi):\\ \mathsf{var}(\phi')\in\lambda_t}} \phi'(\mathcal{A}_i)\right) \bowtie \left(\bowtie_{\substack{\phi'\in\mathsf{atom}(\phi_i):\\ \mathsf{var}(\phi)\subseteq\chi_t}} \phi'(\mathcal{A}_i)\right),$$

i.e., by taking the natural join of the relations belonging to the atoms of the guard λ_t projected to χ_t and all relations of the atoms in ϕ_i whose variables lie in χ_t. The decomposition Ξ_i has width at most k so this construction can be done in time $\|\Phi\|^{O(k)}$ as seen in the proof of Lemma 2.3.28. The query ϕ'_i of Φ'_i is defined as the conjunction of the ϕ_t over all $t \in T$ and with the same quantified variables as ϕ. By Lemma 2.3.28, Φ_i and Φ'_i are solution equivalent, we have $\|\Phi'_i\| \leq \|\Phi_i\|^{O(k)}$ and ϕ'_i is acyclic.

Let $\mathcal{H}_i$ be the associated hypergraph of ϕ'_i, then $(\mathcal{H}_i, S_i)$ has only one single S_i-component, because all the vertices in V_i are connected in $\mathcal{H}$ and thus also in $\mathcal{H}_i$.

We claim that the S_i-star size of $\mathcal{H}_i$ is at most the S_i-star size of $\mathcal{H}[V_i']$. To see this, consider two independent vertices u, v in $\mathcal{H}_i$. The edges e of $\mathcal{H}_i$ are equal to the blocks χ_t of Ξ_i. Because u and v are independent in $\mathcal{H}_i$, they do not appear in a common block χ_t in Ξ_i. But then u and v cannot lie in one common edge in $\mathcal{H}[V_i]$, because every edge in $\mathcal{H}[V_i']$ is contained in a block χ_t by definition of generalized hypertree decompositions. So u and v are independent in $\mathcal{H}[V_i']$ as well. Thus every independent set in $\mathcal{H}_i$ is also independent in $\mathcal{H}[V_i']$. So the S_i-star size of $\mathcal{H}_i$ indeed is at most the S_i-star size of $\mathcal{H}[V_i']$ which is at most ℓ by assumption.

Thus by Lemma 4.1.1 the vertices in S_i can be covered by at most ℓ edges $e_1, \dots, e_\ell$ in $\mathcal{H}_i$ which we can compute in polynomial time. Let $\alpha_1, \dots, \alpha_\ell$ be the atoms corresponding to the edges $e_1, \dots, e_\ell$.

We construct a new atomic formula ϕ_i'' in the variables S_i and an associated relation $\mathcal{R}_i''$ as follows: For each combination $t_1, \dots, t_\ell$ of compatible tuples in $\alpha_1(\mathcal{A}_i'), \dots, \alpha_\ell(\mathcal{A}_i')$ let t be the single tuple in $\pi_{S_i}(\{t_1\} \bowtie \dots \bowtie \{t_\ell\})$. We fix the free variables in ϕ_i' to the constants prescribed by t. The result is a CQ-instance Φ_t with the associated hypergraph $\mathcal{H}[V_i]$. By Observation 3.2.1 Φ_t is acyclic and can thus be solved in polynomial time with Theorem 2.3.19. If Φ_t has a solution, add t to the relation $\mathcal{R}_i''$. This completes the construction of $\mathcal{R}_i''$.

Let $\mathcal{A}_i''$ be the structure containing only the relation $\mathcal{R}_i''$. By construction, $(\mathcal{A}_i'', \varphi_i'')$ is solution equivalent to Φ_i' and and thus also to Φ_i. Observe that the instances Φ_t can be solved in polynomial time by Theorem 2.3.19. Moreover, since $\|\Phi_i'\| \leq \|\Phi\|^{O(k)}$, only $\|\Phi\|^{O(k\ell)}$ tuples t need to be considered. Thus Φ_i'' can be constructed in time $\|\Phi_i\|^{p(k,\ell)}$ for a polynomial p.

We now return to the original instance Φ and eliminate the quantified variables in the query ϕ. To do so, we add the atom ϕ_i'' for $i \in [m]$ and delete all atoms that contain any quantified variable. Moreover, we add the relation $\mathcal{R}_i''$ to the structure $\mathcal{A}$. We call the resulting #CQ instance $\Phi'' = (\mathcal{A}'', \phi'')$. The overall runtime of the construction is at most $\|\Phi\|^{p(k,\ell)}$. Also Φ'' is solution equivalent to Φ, because $(\mathcal{A}_i'', \phi_i'')$ is solution equivalent to Φ_i'.

We claim that Φ'' has generalized hypertree width at most k. To show this we construct a generalized hypertree decomposition Ξ'' of ϕ'' by doing the following: For each $t \in T$ with $\chi_t \cap V_i \neq \emptyset$ we construct a guarded block (λ_t', χ_t') by deleting all edges e with $e \cap V_i \neq \emptyset$ from λ_t and adding the edge S_i for ϕ_i''. Furthermore we set $\chi_t' = (\chi_t \setminus V_i) \cup S_i$. It is easy to see that the result is indeed a generalized hypertree decomposition of ϕ'' of width at most k.

Finally, we construct an ACQ-instance $\Psi := (\mathcal{B}, \psi)$ equivalent to Φ'' with Lemma 2.3.28. ■

We now directly get the desired counting result:

Corollary 5.2.2. #CQ *on instances* Φ *of generalized hypertree width* k *and quantified star size* ℓ *can be solved in time* $\|\Phi\|^{p(k,l)}$ *for a polynomial* p.

Proof. Use Theorem 2.3.25 to construct a generalized hypertree decomposition of width $O(\ell)$, then apply Lemma 5.2.1 and count with Theorem 3.1.2. ■

We state also a path version of Lemma 5.2.1 which we will use in Chapter 10.

Lemma 5.2.3. *There is an algorithm that, given a* CQ*-instance* $\Phi = (\mathcal{A}, \phi)$ *of quantified starsize* ℓ *and a path decomposition* $\Xi = (\mathcal{T}, (\chi_t)_{t \in T})$ *of* Φ *of width* k*, constructs a* CQ*-instance* $\Psi = (\mathcal{B}, \psi)$ *in time* $\|\Phi\|^{p(k,\ell)}$ *for a fixed polynomial* p *such that*

- Φ *and* Ψ *are solution equivalent,*
- Ψ *is acyclic,*
- ψ *is quantifier free.*

Furthermore, the algorithm constructs a join tree $(\mathcal{T}, (\lambda_t)_{t \in T})$ *of* Ψ *such that* $\mathcal{T}$ *is a path.*

Proof (sketch). The approach is the same as for Lemma 5.2.1: We construct instances $(\mathcal{A}_i'', \phi_i'')$ where ϕ_i'' is a single atom as before. Then we substitute all atoms that contain a quantified variable by the corresponding ϕ_i''. This yields a solution equivalent instance whose pathwidth is at most $k\ell$. Finally, we turn this instance into an acyclic instance whose join tree is a path as in Lemma 2.3.28. ■

Combining Corollary 5.2.2 with Theorem 5.1.1 yields a characterization of classes of S-hypergraphs of bounded generalized hypertree width that allow efficient #CQ.

Corollary 5.2.4. *Let* $\mathcal{G}$ *be a recursively enumerable class of* S*-hypergraphs of bounded generalized hypertree width. Then (assuming* $\mathsf{FPT} \neq \#\mathsf{W}[1]$*) the following statements are equivalent:*

1. #CQ *on* $\mathcal{G}$ *is polynomial time tractable.*
2. p-#CQ *on* $\mathcal{G}$ *is fixed-parameter tractable.*
3. $\mathcal{G}$ *is of bounded* S*-star size.*

Proof. $1 \to 2$ is trivial. $2 \to 3$ is Theorem 5.1.1. Finally, $3 \to 1$ is Corollary 5.2.2. ■

As another corollary we get that for a wide range of decomposition techniques commonly considered in the database and artificial intelligence literature, we can characterize the tractable classes of S-graphs by bounded quantified star size. For the decomposition techniques not defined here see [GLS00].

Corollary 5.2.5. *Let β be one of the following decomposition techniques: biconnected component, cycle-cutset, cycle-hypercutset, hingetree, hypertree, or generalized hypertree decomposition. Let furthermore $\mathcal{G}$ be a recursively enumerable class of S-hypergraphs of bounded β-width. Then (assuming* FPT $\neq$ #W[1]*), the following statements are equivalent:*

1. #CQ *on $\mathcal{G}$ is polynomial time tractable.*
2. *p-*#CQ *on $\mathcal{G}$ is fixed-parameter tractable.*
3. *$\mathcal{G}$ is of bounded S-star size.*

Proof. $1 \to 2$ is trivial. $2 \to 3$ follows from Theorem 5.1.1. For $3 \to 1$ observe that for every β of the claim we have that for every hypergraph $\mathcal{H}$ the β-width of $\mathcal{H}$ bounded from below by a function in the generalized hypertree width of $\mathcal{H}$. Thus $\mathcal{G}$ has bounded generalized hypertree width and the claim follows with Corollary 5.2.2. ■

5.3 A #P-INTERMEDIATE CLASS OF COUNTING PROBLEMS

Let us reformulate the results for *p*-#CQ of the last sections as a corollary.

Corollary 5.3.1. *Let $\mathcal{G}$ be a recursively enumerable class of S-hypergraphs of bounded generalized hypertree width. Then p-*#CQ *on $\mathcal{G}$ is either in* FPT *or* #W[1]*-hard.*

Corollary 5.3.1 is an example of what is commonly called a dichotomy result. These are results in which a class of problems is either tractable in the considered sense or hard for a class commonly considered intractable. It is well-known that the class of problems in NP does not allow a P-NP-dichotomy in general. Ladner's Theorem [Lad75] states that if P $\neq$ NP, then there are problems that are neither in P nor NP-complete. Still, restricted classes of problems in NP do allow a dichotomy. An easy example are the k-coloring problems: k-coloring is in P if $k \leq 2$, otherwise it is NP-hard.

Right-hand-side restriction of CSP, i.e., such in which the relations of the instances are restricted but the structure of the queries is unrestricted, are a source for many dichotomy results. The starting point for this is the seminal result of Schaefer [Sch78] that gives a P-NP-dichotomy for right-hand-side restrictions of CSP restricted to the domain $\{0, 1\}$. Feder and Vardi [FV98] conjectured that there is a dichotomy for CSP in general. This conjecture is still open, but it has spurred a huge amount of research in the area with many partial results (see [Bul11] for a survey). For #CSP a general FP-#P-dichotomy has been proven recently even for weighted counting [CC12].

For restrictions on the queries of the type we are considering in this thesis, the situation is less clear. For general CQ it is unknown if

there is any type of dichotomy (there is however a characterization of the classes of hypergraphs allowing fixed-parameter tractable p-CQ assuming the exponential time hypothesis [Mar10]). For bounded arity, Grohe [Gro07] showed an FPT-W[1]-dichotomy of p-CQ; Dalmau and Jonsson [DJ04] modified Grohe's techniques to give a FPT-#W[1]-dichotomy of p-#CQ restricted to quantifier free instances (see also Chapter 7). However, there is no P-NP-dichotomy for CQ [BG08]. We will show that similarly, there is no FP-#P-dichotomy version of Corollary 5.3.1 for #CQ.

Let $\mathcal{G}_{\text{star}}$ be the class of S-graphs (G_n, S_n) where G_n is the star with n leaves and S_n consists of all vertices but the center of G_n. Note that the S-hypergraphs of the queries $\mathcal{C}_{\text{star}}$ from Lemma 3.1.4 are in $\mathcal{G}_{\text{star}}$ (see Example 3.2.7).

Theorem 5.3.2. *There is a subclass $\mathcal{G}_0$ of $\mathcal{G}_{star}$ such that* #CQ *on $\mathcal{G}_0$ is neither in* FP *nor* #P*-complete unless* FP $\neq$ #P.

The proof is similar to the diagonalization proof of Ladner's Theorem [Lad75] as presented in [Pap94]. We follow an adaption of these techniques by Bodirsky and Grohe [BG08].

The class $\mathcal{G}_0$ is defined by a function $f : \mathbb{N} \to \mathbb{N}$ which we define below. $\mathcal{G}_0$ contains exactly those S-hypergraphs from $\mathcal{G}_{\text{star}}$ for which $f(|G|)$ is even.

Let $M_1, M_2, \ldots$ be an enumeration of all polynomial-time bounded Turing-machines computing functions $h : \mathbb{N} \to \mathbb{N}$ and let $R_1, R_2, \ldots$ be an enumeration of all polynomial-time bounded oracle Turing-machines computing a function $g : \mathbb{N} \to \mathbb{N}$ given an oracle $h' : \mathbb{N} \to \mathbb{N}$. Furthermore, we assume that we have an enumeration $\Phi_1, \Phi_2, \ldots$ of all CQ-instances $\Phi_i = (\mathcal{A}_i, \phi_i)$ where the S-hypergraph of ϕ_i is in $\mathcal{G}_{\text{star}}$.

The function f is defined by a polynomial time Turing machine F that computes f. The machine F is given its input n in unary and works in two phases. In the first phase F simulates itself on inputs $1, 2, \ldots$ and thus computes $f(1), f(2), \ldots$ until the total number of steps done in this phase exceeds n. Let ℓ be the last value for which this simulation was finished. We set $k := f(\ell)$. If F did not finish the simulation on the input 1, we set $k := 0$. The value computed by f on the input n will either be k or $k+1$, depending on the outcome of the second phase.

This second phase of F depends on whether k is even or odd. If $k = 2i$ is even then F tries to find an instance Φ_j for which $M_i(\Phi_j) \neq |\phi_j(\mathcal{A}_j)|$. To do so it enumerates $\Phi_1, \Phi_2, \ldots$ and simulates M_i on each instance Φ_j and computes $|\phi_j(\mathcal{A}_j)|$ and $f(|G_j|)$ where G_j is the associated graph of ϕ. If $M_i(\Phi_j) \neq |\phi_j(\mathcal{A}_j)|$ and $f(|G|)$ is even, i.e., the S-hypergraph of ϕ is in $\mathcal{G}_0$, then F stops and outputs $k+1$. Otherwise it proceeds with the next instance Φ_{j+1}. When the overall number of steps in this phase exceeds n and F has not stopped yet, it stops and outputs k.

If $k = 2i - 1$ is odd, then F tries to find an instance Φ_j such that R_i, given a correct oracle for #CQ, does not compute $|\phi_j(\mathcal{A}_j)|$. To do so F again enumerates instances $\Phi_1, \Phi_2, \ldots$, simulates R_i on each instance Φ_j where the oracle calls are answered correctly by brute force computations and computes $|\phi_j(\mathcal{A}_j)|$. During these simulations F also computes $f(|G|)$ for each oracle question $(\mathcal{A}, \phi)$ where G is the associated hypergraph of ϕ. If $R_i(\Phi_j) \neq |\phi_i(\mathcal{A})|$ or for one of the oracle questions $(\mathcal{A}, \phi)$ we have that $f(|G|)$ is odd, then F stops and outputs $k + 1$. Otherwise it proceeds with the next instance. When the overall number of steps in this phase exceeds n and F has not stopped yet, it stops and outputs k.

Proposition 5.3.3. *For every n_0 and every k^* with $0 \leq k^* \leq f(n_0)$ there is an n' such that $f(n') = k^*$.*

Proof. On input $n = 0$ the machine F does not finish any simulations in either of its phases, so $f(0) = 0$.

For $n > 0$, the value k after the first phase is either $k = f(n)$ or $k = f(n) - 1$. On input n the machine F only simulates itself for overall n steps in the first phase, so F only simulates itself on inputs $\ell < n$. Thus when n' is the smallest integer for which F outputs $f(n)$, the the value of k after the first phase on input n' must have been $f(n) - 1$. It follows that there must be a n'' with $f(n'') = k - 1$ by construction. Now the result follows by induction. ■

Proposition 5.3.4. *The function f is non-decreasing.*

Proof. We make an induction on n. Assume that $f(n' - 1) \leq f(n')$ for all $n' \leq n$. In the first phase F cannot simulate itself on more than n values on input n. Furthermore, on input $n + 1$ the machine can simulate itself on no less inputs than on input n, so if F computes the value k on input n in the first phase, then F computes at least the value k in the first phase on input $n + 1$.

After the second phase the output on n can only be k or $k + 1$. If it is k, then $f(n) = k \leq f(n + 1)$, because the second phase does not decrease output value and the output of the first phase of F on $n + 1$ is at least k. If $f(n) = k + 1$ and F computed $k + 1$ in the first phase on $n + 1$, then $f(n) = k + 1 \leq f(n + 1)$. This is again because the second phase never decreases the output. If $f(n) = k + 1$ and F computed k in the first phase on $n + 1$, then F simulates the same machine M_i, resp. R_i on input n and $n + 1$. Furthermore, there must be a CQ-instance Φ_j that lead to increasing the output $f(n)$ to $k + 1$. But as F on input $n + 1$ has more time for simulations, it finds Φ_j then, too. Thus $f(n + 1) = k + 1 = f(n)$. ■

Proposition 5.3.5. *The function f is unbounded unless* FP $\neq$ #P.

Proof. Assume that f is bounded. With Proposition 5.3.4 we know that f actually becomes constant starting from an integer n_0. Let $k :=$

$f(n_0)$. Combining Proposition 5.3.3 and Proposition 5.3.4 we get that there is an integer n_1 such that for every input $n > n_1$ the first phase of F computes k.

If $k = 2i$ is even, then only finitely many of the S-graphs in $\mathcal{G}_{\text{star}}$ are not in $\mathcal{G}_0$ by definition. Furthermore, for every input $n > n_1$ the machine F in the second phase simulates the machine M_i. Since f never increases to any value $k' > k$ all computations of M_i must give the correct value on the inputs Φ_j. But as M_i is eventually run on every input Φ_j, the machine M_i solves #CQ in $\mathcal{G}_0$ in polynomial time and it follows that #CQ on $\mathcal{G}_0$ is in FP. We give a polynomial time algorithm for #CQ on $\mathcal{G}_{\text{star}}$: On input $(\mathcal{A}, \phi)$ check if the associated hypergraph of ϕ is in $\mathcal{G}_0$. If yes, use M_i to compute $|\phi(\mathcal{A})|$. Otherwise, compute $|\phi(\mathcal{A})|$ by brute force in time $\|\mathcal{A}\|^{O(|\mathsf{free}(\phi)|)}$. This is polynomial time as $|\mathsf{free}(\phi)|$ is bounded because $\mathcal{G}_{\text{star}} \setminus \mathcal{G}_0$ is finite. If follows that #CQ on $\mathcal{G}_{\text{star}}$ is in FP and with Lemma 3.1.4 we get FP = #P.

If $k = 2i - 1$ is odd, then $\mathcal{G}_0$ is finite. Furthermore, for each $n > n_1$ the machine F simulates the oracle machine R_i. Since F does never increase, the computations of R_i with a correct oracle produce the correct result. Furthermore, all oracle questions $(\mathcal{A}, \phi)$ have their associated S-hypergraph in $\mathcal{G}_0$ because $f(|G|)$ is even for all oracle questions. It follows that R_i computes a Turing-reduction from #CQ on $\mathcal{G}_{\text{star}}$ to #CQ on $\mathcal{G}_0$. Hence, by Lemma 3.1.4 #CQ on $\mathcal{G}_0$ is #P-hard. But the brute force algorithm for this problem takes again time $\|\mathcal{A}\|^{O(|\mathsf{free}(\phi)|)}$ which is polynomial, because $|\mathsf{free}(\phi)|$ is bounded. It follows that FP = #P. ■

Proof of Theorem 5.3.2. Obviously, #CQ on $\mathcal{G}_0$ is in #P, because #CQ on the class $\mathcal{G}_{\text{star}}$ is in #P.

Assume first that #CQ on $\mathcal{G}_0$ is in FP and FP $\neq$ #P. Then there is a machine M_i that solves #CQ on $\mathcal{G}$. Combining the propositions from above, there is an integer n_0 such that $f(n_0) = 2i$. Again combining the three propositions from above there is then also an $n_1 > n_0$ such that the first phase of F computes $k = 2i$. Since M_i always computes the correct value $|\phi_j(\mathcal{A})|$ on instances that have their associated S-hypergraph in $\mathcal{G}_0$, the output of F on input n_1 is also k. Thus by Proposition 5.3.4 $f(n') = k$ for all n' between n_0 and n_1. Now a straighforward induction shows that F never computes a value bigger than k which contradicts FP $\neq$ #P by Proposition 5.3.5.

Now assume that #CQ on $\mathcal{G}_0$ is #P-complete. Then there is an oracle machine R_i reducing #CQ on $\mathcal{G}_{\text{star}}$ to #CQ on $\mathcal{G}_0$. We again combine the three propositions and get that there is an integer n_0 such that $f(n_0) = 2i - 1$. Furthermore there is an n_1 such that the first phase of F computes $k = 2i - 1$. Thus F simulates the oracle machine R_i in the second phase. But R_i reduces correctly and the oracle questions are answered correctly as well. Thus all simulations compute $|\phi_j(\mathcal{A}_j)|$ for each instance Φ_j. Furthermore, all oracle questions $(\mathcal{A}, \phi)$ have their associated S-hypergraph in $\mathcal{G}_0$ and thus $f(|G|)$ is even where G is the

associated hypergraph of ϕ. It follows that the output $f(n_1)$ is k. As before, using Proposition 5.3.4 shows $f(n') = k$ for all n' between n_0 and n_1 and a straightforward induction shows that f is bounded by k. This again implies FP = #P with Proposition 5.3.5. ■

5.4 FRACTIONAL HYPERTREE WIDTH

In this section we extend the main results of this chapter to *fractional hypertree width*, which is the most general notion known that leads to tractable CQ [GM06]. In particular it is strictly more general than generalized hypertree width. The proofs can be found in the appendix.

Definition 5.4.1. *Let $\mathcal{H} = (V, E)$ be a hypergraph. A* fractional edge cover *of a vertex set $S \subseteq V$ is a mapping $\psi : E \to [0,1]$ such that for every $v \in V$ we have $\sum_{e \in E : v \in e} \psi(e) \geq 1$. The weight of ψ is $\sum_{e \in E} \psi(e)$. The* fractional edge cover number *of S, denoted by $\rho^*_{\mathcal{H}}(S)$, is the minimum weight taken over all fractional edge covers of S.*

We define a fractional hypertree decomposition *of $\mathcal{H}$ to be a triple $(\mathcal{T}, (\chi_t)_{t \in T}, (\psi_t)_{t \in T})$ where $\mathcal{T} = (T, F)$ is a tree, $\chi_t \subseteq V$ and ψ_t is a fractional edge cover of χ_t for every $t \in T$, satisfying the following properties:*

1. *For every $v \in V$ the set $\{t \in T \mid v \in \chi_t\}$ induces a subtree of $\mathcal{T}$.*
2. *For every $e \in E$ there is a $t \in T$ such that $e \subseteq \chi_t$.*

The width *of a fractional hypertree decomposition $(\mathcal{T}, (\chi_t)_{t \in T}, (\psi_t)_{t \in T})$ is $\max_{t \in T}(\rho^*_{\mathcal{H}}(\chi_t))$. The* fractional hypertree width $\mathrm{fhw}(\mathcal{H})$ *of $\mathcal{H}$ is the minimum width over all fractional hypertree decompositions of $\mathcal{H}$.* □

Fractional hypertree width is more general that generalized hypertree width in the following sense:

Lemma 5.4.2 ([GM06]). *The fractional hypertree width of a hypergraph $\mathcal{H}$ is at most the generalized hypertree width of $\mathcal{H}$. Moreover, there are families of hypergraphs of bounded fractional hypertree width but unbounded generalized hypertree width.*

We first formulate a version of Corollary 5.2.2 for fractional hypertree width.

Theorem 5.4.3. *There is an algorithm that, given a #CQ-instance Φ of quantified starsize ℓ and fractional hypertree width k, counts the solutions of Φ in time $\|\Phi\|^{p(k,\ell)}$ for a polynomial p.*

Furthermore, we will show that the S-star size, or equivalently independent sets, of bounded fractional hypertree width hypergraphs can be computed efficiently.

Lemma 5.4.4. *There is an algorithm that given a hypergraph $\mathcal{H} = (V, E)$ of fractional hypertree width at most k computes a maximum independent set of $\mathcal{H}$ in time $|\mathcal{H}|^{k^{O(1)}}$.*

As a corollary we get a version of Corollary 5.2.5.

Corollary 5.4.5. *Let $\mathcal{G}$ be a recursively enumerable class of S-hypergraphs of bounded fractional hypertree width. Then (assuming $\mathsf{FPT} \neq \#\mathsf{W}[1]$) the following statements are equivalent:*

- #CQ *on $\mathcal{G}$ is polynomial time tractable.*
- *p*-#CQ *on $\mathcal{G}$ is fixed-parameter tractable.*
- *$\mathcal{G}$ is of bounded S-star size.*

6 QUERIES OF BOUNDED ARITY

In this chapter we show that for bounded arity #CQ we can exactly characterize the classes of S-hypergraphs that allow polynomial time counting. In this chapter all CQ-instances and all S-hypergraphs are always assumed to be of bounded arity.

We will give two different characterizations of S-hypergraphs of bounded arity that allow tractable #CQ: The first characterization is presented in Section 6.1 and uses treewidth and S-star size, following the ideas of Chapter 5. In Section 6.2 we introduce a notion of elimination width for conjunctive queries. It will allow us to characterize the S-hypergraphs of bounded arity that allow tractable #CQ with a single parameter.

6.1 A CHARACTERIZATION BY TREEWIDTH AND S-STAR SIZE

In this section we characterize the S-hypergraphs of bounded arity that allow tractable #CQ by treewidth and S-star size. The result of this section is based on a combination of the results of Chapter 5 and a result by Grohe from [Gro07] which is a followup of results by Grohe, Schwentick and Segoufin [GSS01]. We state the theorem in our slightly different wording.

For a class $\mathcal{G}$ of hypergraphs we denote by CQ on $\mathcal{G}$ the decision problem CQ restricted to instances whose associated hypergraph is in $\mathcal{G}$.

Theorem 6.1.1 ([Gro07]). *Let $\mathcal{G}$ be a recursively enumerable class of hypergraphs of bounded arity. Assume $\mathsf{FPT} \neq \mathsf{W}[1]$. Then the following three statements are equivalent:*

1. *CQ on $\mathcal{G}$ can be decided in polynomial time.*
2. *p-CQ on $\mathcal{G}$ is fixed parameter tractable.*
3. *There is a constant c such that the hypergraphs in $\mathcal{G}$ have treewidth at most c.*

Theorem 6.1.1 is originally stated even for every fixed vocabulary. In Section 7 we will see a refinement of it.

Our goal is to provide a complete characterization of classes of S-hypergraphs of bounded arity that yield tractability for #CQ. Not too surprisingly, tractability depends on both treewidth and star size of the underlying S-hypergraph.

Theorem 6.1.2. *Let $\mathcal{G}$ be a recursively enumerable class of S-hypergraphs of bounded arity. Assume that $\mathsf{W}[1] \neq \mathsf{FPT}$. Then the following statements are equivalent:*

1. *#CQ on $\mathcal{G}$ is solvable in polynomial time.*
2. *p-#CQ on $\mathcal{G}$ is fixed-parameter tractable.*
3. *There is a constant c such that for each S-hypergraph $(\mathcal{H}, S)$ in $\mathcal{G}$ the treewidth of $\mathcal{H}$ and the S-star size of $\mathcal{H}$ are at most c.*

Let us discuss how Theorem 6.1.2 and Corollary 5.2.4 relate. First, it is not hard to see that for bounded arity hypergraphs treewidth and generalized hypertree width differ only by a constant factor. So we could have formulated Theorem 6.1.2 with generalized hypertree width instead of treewidth as well.

The key difference between Theorem 6.1.2 and Corollary 5.2.4 is that we can show here that bounded treewidth is not only sufficient for tractable counting but also necessary. As we already directly get from Theorem 3.1.2, there are by Lemma 2.3.4 families of S-hypergraphs of unbounded arity, and thus also unbounded treewidth, on which #CQ is tractable, so treewidth is not the right notion for this case. It is an intriguing question if there is a width measure that completely characterizes tractable CQ or tractable #CQ on graphs of unbounded arity, similarly to Theorem 6.1.1 and Theorem 6.1.2 in the bounded arity case.

Before giving the proof of Theorem 6.1.2 we make an observation.

Observation 6.1.3. *If there is a recursively enumerable class $\mathcal{G}$ of S-hypergraphs of unbounded treewidth such that p-#CQ on $\mathcal{G}$ is fixed-parameter tractable, then there is such a class $\mathcal{G}'$ that is recursive.*

Proof. Fix a Turing machine M that enumerates $\mathcal{G}$. Let the order in which the S-hypergraphs of $\mathcal{G}$ are enumerated by M be $(\mathcal{H}_1, S_1)$, $(\mathcal{H}_2, S_2), \ldots$. Then define $\mathcal{G}'$ as containing the S-hypergraphs $(\mathcal{H}'_i, S'_i)$ where $\mathcal{H}'_i$ is the disjoint union of the hypergraphs $\mathcal{H}_1, \ldots, \mathcal{H}_i$ and $S'_i := \bigcup_{j \in [i]} S_i$.

We claim that $\mathcal{G}'$ is recursive. Indeed the definition of $\mathcal{G}'$ directly gives an algorithm that enumerates the elements of $\mathcal{G}'$ ordered by size. This yields an algorithm to decide membership in $\mathcal{G}'$: Given an input $(\mathcal{H}, S)$, enumerate the elements of $\mathcal{G}'$ until $(\mathcal{H}, S)$ is found or an element that has more vertices than $(\mathcal{H}, S)$ is enumerated.

The treewidth of $\mathcal{G}$ is trivially unbounded.

Finally, we claim that #CQ on $\mathcal{G}'$ is fixed-parameter tractable. Given an input $\Phi := (\mathcal{A}, \phi)$ first check if the associated S-hypergraph $(\mathcal{H}, S)$ is in $\mathcal{G}'$. If not, stop. It yes, the query ϕ must decompose into subqueries $\phi_1, \ldots, \phi_i$ such that for each $j \in [i]$ the query ϕ_j has the S-hypergraph $(\mathcal{H}_j, S_j)$ and the ϕ_j have disjoint variable sets. Using the enumerating machine M we can compute such a decomposition. Now

since #CQ on $\mathcal{G}$ is fixed-parameter tractable we can solve the instances $\Phi_j := (\mathcal{A}, \phi_j)$ in time $g(|\phi_j|)\|\Phi_j\|^c$ for a computable function g and a constant c. If follows that $|\phi(\mathcal{A})| = \prod_{j\in[i]} |\phi_j(\mathcal{A})|$ can be computed in time $\sum_{j\in[i]} g(|\phi_j|)\|\Phi_j\|^c \leq |\phi| g(|\phi|)\|\Phi\|^c$ and thus #CQ on $\mathcal{G}'$ is fixed-parameter tractable. ■

Proof of Theorem 6.1.2. The direction 1 → 2 is trivial. Furthermore, 3 → 1 follows directly from Corollary 5.2.5. So it remains only to show 2 → 3.

By way of contradiction, we assume that there is a recursively enumerable class $\mathcal{G}$ of S-hypergraphs such that counting solutions to #CQ-instances, whose S-hypergraph are in $\mathcal{G}$, is fixed parameter tractable, but 3 is not satisfied by $\mathcal{G}$. From Theorem 5.1.1 we know that the S-starsize of $\mathcal{G}$ must be bounded, so it follows that the treewidth of $\mathcal{G}$ is unbounded. With Observation 6.1.3 we may assume that $\mathcal{G}$ is recursive.

We construct a class $\mathcal{G}'$ of hypergraphs as

$$\mathcal{G}' := \{\mathcal{H} \mid (\mathcal{H}, S) \in \mathcal{G}\}.$$

Clearly $\mathcal{G}'$ is recursive and of unbounded treewidth. We will show that p-CQ on $\mathcal{G}'$ is fixed-parameter tractable. This is a contradiction with Theorem 6.1.1.

Because $\mathcal{G}$ is recursive, there is an algorithm that for each $\mathcal{H}$ in $\mathcal{G}'$ constructs an S-hypergraph $(\mathcal{H}, S)$ in $\mathcal{G}$. For example, one can simply try all vertex sets S and check if $(\mathcal{H}, S)$ is in $\mathcal{G}$. Let $f(\mathcal{H})$ be the number of steps the algorithm needs on input $\mathcal{H}$. The function $f(\mathcal{H})$ is well defined and computable. We then define $g : \mathbb{N} \to \mathbb{N}$ by setting $g(k) := \max_{\mathcal{H}}(f(\mathcal{H}))$, where the maximum is over all hypergraphs $\mathcal{H}$ of size k in $\mathcal{G}'$. The function g is well defined and computable, because $\mathcal{G}'$ is recursive. Thus for each $\mathcal{H}$ in $\mathcal{G}'$ we can compute in time $g(|\mathcal{H}|)$ an S-hypergraph $(\mathcal{H}, S)$ in $\mathcal{G}$.

Now let $\Phi = (\mathcal{A}, \phi)$ be a CQ-instance with hypergraph $\mathcal{H}$ in $\mathcal{G}'$. To solve it we first compute $(\mathcal{H}, S)$ as above and construct a CQ-instance $\Psi = (\mathcal{A}, \psi)$ with $(\mathcal{H}, S)$ as associated S-hypergraph for ψ by adding existential quantifiers for all variables not in S. Obviously Φ has solutions if and only if Ψ has one. But by assumption the solutions of Ψ can be counted in time $h(|\psi|)\|\Psi\|^{O(1)}$ for some computable function h, so Φ can be decided in time $(g(|\phi|) + h(|\phi|))\|\Phi\|^{O(1)}$. Thus p-CQ on $\mathcal{G}'$ is fixed-parameter tractable. This is the desired contradiction to Theorem 6.1.1. ■

6.2 A CHARACTERIZATION BY ELIMINATION ORDERS

While the characterization of Theorem 6.1.2 is great because it completely characterizes the tractable classes of S-hypergraphs for #CQ, it has the somewhat unpleasant property that we have to bound two

different parameters of the hypergraphs instead of just one. Also, it is not clear how robust and natural the defined classes of hypergraphs are. In contrast to this, treewidth is a very robust notion that has many equivalent definitions.

In this section we improve the situation by showing that there is a notion of elimination width for S-hypergraphs that is equivalent to the combination of treewidth and S-star size.

Recall the notion of elimination orders from Section 2.3.1.

Definition 6.2.1. *Let (G, S) be an S-graph. We define an* elimination order *π of an S-graph (G, S) as an elimination order of $G = (V, E)$ such that for each pair $v \in S$, $u \in V \setminus S$ such that uv is an edge in the fill-in graph G_π we have $\pi(u) < \pi(v)$.*

The elimination width elim-width(G, S) *of (G, S) is defined as the minimum width taken over all elimination orders of (G, S).*

The elimination width elim-width$(\mathcal{H}, S)$ *of an S-hypergraph $(\mathcal{H}, S)$ is defined that the elimination width of its primal S-graph. By $\mathcal{H}_{P,\pi}$ we denote the fill-in graph of the primal graph $\mathcal{H}_P$ of $\mathcal{H}$ with respect to π.* □

Remark 6.2.2. Observe that for every S-graph (G, S) we have

$$\text{elim-width}(G, S) \geq \text{elim-width}(G),$$

because every elimination order of (G, S) is an elimination order of G. □

Proposition 6.2.3. *Let $\mathcal{G}$ be a class of S-hypergraphs. Then the following statements are equivalent:*

- *The treewidth and the S-star size of the S-hypergraphs in $\mathcal{G}$ are bounded by a constant c.*
- *The elimination width of the S-hypergraphs in $\mathcal{G}$ is bounded by a constant c'.*

The proof of Proposition 6.2.3 is somewhat lengthy, so we prove it in two individual lemmas.

Lemma 6.2.4. *Let $(\mathcal{H}, S)$ be an S-hypergraph of elimination width k. Then the treewidth of $\mathcal{H}$ is at most k and the S-star size of $(\mathcal{H}, S)$ is at most $k + 1$.*

Proof. From Remark 6.2.2 and Lemma 2.3.9 it follows directly that the treewidth of $\mathcal{H}$ is at most k. Thus we only have to show the bound on the S-star size of $(\mathcal{H}, S)$. To this end, we define an *S-path* (P, S) as a path whose end vertices are in S but all other vertices are not in S.

Claim 6.2.5. *Let u, v be the end vertices of an S-path (P, S) with $P = ux_1 \dots x_\ell v$. Then for every elimination order π of (P, S) we have $\pi(v) > \pi(x_i)$ and $\pi(u) > \pi(x_i)$ for all $i \in [\ell]$. Furthermore, uv is an edge of the fill-in graph P_π.*

Proof. We prove this by induction on ℓ. For $\ell = 0$ there is nothing to show.

Now let $\ell \geq 1$. Let $x_j = \operatorname{argmin}_{i \in \ell}(\pi(x_i))$. By the definition of elimination orders we have $\pi(x_1) < \pi(u)$ and $\pi(x_\ell) < \pi(v)$, so $\pi(x_j) < \min(\pi(v), \pi(u))$. Let P' be the path that we get from P when deleting x_j and connecting x_{j-1} and x_{j+1} by an edge. P' is a subgraph of the fill-in graph P_π and π induces an elimination order on P' by $\pi'(w) := \pi(w) - 1$. It follows that $P'_{\pi'}$ is a subgraph of P_π. By induction $\pi'(v) > \pi'(x_i)$ and $\pi'(u) > \pi'(x_i)$ for all $i \in [\ell] \setminus \{j\}$ and thus $\pi(v) > \pi(x_i)$ and $\pi(u) > \pi(x_i)$ for all $i \in [\ell]$. Furthermore, by induction uv is an edge of $P'_{\pi'}$ and thus also of P_π. This completes the proof of the claim. ■

By definition of S-components, in every S-graph (G, S) every pair $u, v \in S$ must be connected by an S-path.

Let $\mathcal{H}_P = (V, E_P)$ be the primal graph of $\mathcal{H}$. Let $\mathcal{H}'$ be an S-component of $\mathcal{H}$ with primal S-graph $\mathcal{H}'_P = (V', E'_P)$ and let $S' := S \cap V'$.

Let π be an optimal elimination order of $(\mathcal{H}, S)$ of width k. Then π induces for every subgraph $\mathcal{H}''$ an elimination order of $\mathcal{H}''$ of width at most k. To ease notation we will not differentiate between π and these induced elimination orders and simply call π an elimination order of all subgraphs, too.

As already remarked, all pairs $u, v \in S'$ are connected by S-paths in $\mathcal{H}'_P$. The fill-in graph of every subgraph of $\mathcal{H}'_P$ is a subgraph of the fill-in graph $\mathcal{H}'_{P,\pi}$ of $\mathcal{H}'_P$. Thus by Claim 6.2.5 we have that the vertices in S' form a clique in $\mathcal{H}'_{P,\pi}$. Because π has width k, it follows that $|S'| \leq k + 1$. Hence the S-star size of $\mathcal{H}'$ is at most $k + 1$. This completes the proof of Lemma 6.2.4.

For the other direction of Proposition 6.2.3 we will use the following lemma.

Lemma 6.2.6. *Let $(\mathcal{H}, S)$ be an S-hypergraph of treewidth at most c and S-star size at most k. Then every S-component of $\mathcal{H}$ contains at most $k(c + 1)$ vertices from S.*

Proof. We prove this by induction on the S-star size k while keeping the treewidth fixed to c. If the S-star size is $k = 1$, then in every S-component all vertices from S are adjacent. But then they induce a clique in the primal graph of $\mathcal{H}$ and thus by Lemma 2.3.4 there may be at most $c + 1$ of them.

Let now $k > 1$. Consider an S-component $\mathcal{H}'$ of $\mathcal{H}$. The graph $\mathcal{H}'_P[S]$ has at most treewidth c because it is an induced subgraph of $\mathcal{H}'_P$ which has by assumption treewidth at most c. By Lemma 2.3.6, there is a vertex v in $\mathcal{H}'_P[S]$ of degree at most c. If follows that v has at most c neighbors in S in $\mathcal{H}'$. Let $\mathcal{H}''$ be the hypergraph we get from

$\mathcal{H}' = (V', E')$ by deleting v and all of its neighbors in S. We claim that $(\mathcal{H}'', S \cap V')$ has S-star size at most $k-1$.

Assuming this is false, there are k independent vertices $v_1, \dots, v_k \in S$ in $\mathcal{H}''$. But then $v_1, \dots, v_k, v$ are $k+1$ independent vertices from S in $\mathcal{H}'$, so the S-star size of $\mathcal{H}$ is at most $k+1$ which contradicts the assumption.

So the S-star size of $\mathcal{H}''$ is indeed bounded by $k-1$. By induction $\mathcal{H}''$ contains at most $(k-1)(c+1)$ vertices from S, and since we deleted at most $c+1$ vertices during the construction of $\mathcal{H}''$ we get that $\mathcal{H}'$ contains at most $k(c+1)$ vertices from S. ■

We now prove the second direction of Proposition 6.2.3

Lemma 6.2.7. *Let $(\mathcal{H}, S)$ be an S-hypergraph such that the treewidth and the S-star size of $(\mathcal{H}, S)$ are bounded by $c \in \mathbb{N}$. Then the elimination width of $(\mathcal{H}, S)$ is at most $(c+1)^3 + (c+1)^2$.*

Proof. Let $(\mathcal{T}, (\chi_t)_{t \in T})$ be a tree decomposition of $\mathcal{H} = (V, E)$ of minimal width ℓ. Let $S(v)$ for every $v \in V \setminus S$ be the set of vertices from S in the S-component of v. For every $t \in T$ we construct a new bag χ'_t as

$$\chi'_t := \chi_t \cup \bigcup_{v \in (V \setminus S) \cap \chi_t} S(v).$$

Because the S-star size of $\mathcal{H}$ is at most c we get by Lemma 6.2.6 that $|S(v)| \leq c(\ell+1)$. It follows with $\ell \leq c$ that

$$\begin{aligned} |\chi'_t| &\leq |\chi_t| + \sum_{v \in (V \setminus S) \cap \chi_t} |S(v)| \\ &\leq (\ell+1) + (\ell+1)c(\ell+1) \\ &\leq (c+1)^3 \end{aligned}$$

It is easy to see that $(\mathcal{T}, (\chi'_t)_{t \in T})$ is a tree decomposition. Remember that for each $t \in T$ the tree $\mathcal{T}_t$ is the subtree of $\mathcal{T}$ with t as its root. Let V_t be the set of vertices appearing in the bags $\chi'_{t'}$ of $\mathcal{T}_t$. For each $y \in V$ let $r(y)$ be the $t \in T$ with $y \in \chi'_t$ that is nearest to the root of $\mathcal{T}$.

Claim 6.2.8. *There exists $t \in T$ such that $\emptyset \neq V_t \cap (V \setminus S) \subseteq \chi'_t$ with a vertex $y \in V_t \cap (V \setminus S)$ with $t = r(y)$.*

Proof. We find t and y by descending in $\mathcal{T}$. Let r be the root of $\mathcal{T}$. If $V_r \cap (V \setminus S) \subseteq \chi'_r$ we are done. Otherwise let t_i be a child of r such that $V_{t_i} \cap (V \setminus S) \not\subseteq \chi'_r$. Now check if $V_{t_i} \cap (V \setminus S) \subseteq \chi'_{t_i}$ and if not go deeper in $\mathcal{T}$. Let t be the first vertex on this descent with $V_t \cap (V \setminus S) \subseteq \chi'_t$. Then χ'_t must contain a vertex y that is not in $\chi'_{t'}$ where t' is the parent of t in $\mathcal{T}$. But then $r(y) = t$ as desired. ■

We construct an elimination order π of $\mathcal{H}$ inductively as follows, starting from the empty elimination order: While any bag of the tree

decomposition $(\mathcal{T}, (\chi'_t)_{t\in T})$ contains a vertex from $V \setminus S$, do the following: Choose by Claim 6.2.8 $t \in T$ such that $\emptyset \neq V_t \cap (V \setminus S) \subseteq \chi'_t$ with a vertex $y \in V_t \cap (V \setminus S)$ with $t = r(y)$, delete y from $\mathcal{H}$ and all bags and add y as the next vertex to the elimination order π.

When the vertices from $V \setminus S$ have all been deleted, we proceed with the vertices in S in a similar fashion: While there is a non-empty bag, choose one $t \in T$ with $\emptyset \neq S \cap V_t \subseteq \chi'_t$ and $y \in S \cap V_t$ with $r(y) = t$, delete y and add y as the next vertex in π. Again, such t and y can always be found.

All vertices in $V \setminus S$ appear before all vertices in S in π, so π is an elimination order of $(\mathcal{H}, S)$. We will now bound the width of π.

Claim 6.2.9. *Let $x, y \in V$ with $x, y \in V \setminus S$ or $x, y \in S$ such that there exists $t \in T$ with $x, y \in \chi'_t$. If x is higher-numbered than y with respect to π, then $x \in \chi_{r(y)}$.*

Proof. x and y appear in a common bag χ'_t and thus $x \in V_{r(y)}$. But x is higher-numbered, so y was deleted before x. Hence, when y was chosen to be deleted the vertex x was still in $V_{r(y)}$. But then $x \in \chi_{r(y)}$ because otherwise y would not have been chosen for deletion. ■

Claim 6.2.10.
a) For every vertex $y \in V \setminus S$ the neighbors of y in $\mathcal{H}_{P,\pi}$ are vertices of the same S-component as y.

b) When a vertex $y \in V \setminus S$ is deleted, the bag $\chi'_{r(y)}$ contains all higher-numbered neighbors of y in the fill-in graph $\mathcal{H}_{P,\pi}$ that are in $V \setminus S$.

c) When a vertex $y \in S$ is deleted, the bag $\chi'_{r(y)}$ contains all higher-numbered neighbors of y in the fill-in graph $\mathcal{H}_{P,\pi}$.

Proof. We first prove a) and b) by induction along the elimination order π. So let first y be the vertex with $\pi(y) = 1$. We claim that the higher-numbered neighbors of y in $\mathcal{H}_{P,\pi}$ are simply the neighbors of y in $\mathcal{H}_P$. Certainly, these are all higher-numbered. Also, in the construction of $\mathcal{H}_{P,\pi}$ from $\mathcal{H}_P$ edges incident to y may only be added by lower-numbered vertices. As there are none for y, all neighbors of y in $\mathcal{H}_{P,\pi}$ are already neighbors in $\mathcal{H}_P$. This proves the induction start for a). For b) consider a neighbor $x \in V \setminus S$ of y in $\mathcal{H}$. By the definition of tree decompositions x and y must be in one common bag χ'_t. With Claim 6.2.9 it follows that $x \in \chi_{r(y)}$.

Consider now $y \in V \setminus S$ with $\pi(t) > 1$. All neighbors $x \in V \setminus S$ of y in $\mathcal{H}_{P,\pi}$ are either already neighbors of y in $\mathcal{H}$ and thus in the same S-component as y or they are neighbors that originate from edges added in the construction of $\mathcal{H}_{P,\pi}$ from $\mathcal{H}_P$. In the latter case the edge xy must have been added because of a common lower-numbered neighbor v of y and x. Because v is lower-numbered than y it follows that $v \in V \setminus S$. By induction all higher-numbered neighbors of v in $\mathcal{H}_{P,\pi}$ are in the same S-component as v in $\mathcal{H}$, so y, x and v are all vertices of the same S-component which completes the proof of a).

Now let $x \in V \setminus S$ be a higher-numbered neighbor of $y \in V \setminus S$ in $\mathcal{H}_{P,\pi}$. Consider first the case that xy is already an edge in $\mathcal{H}_P$. Then there is a bag χ'_t such that $x, y \in \chi'_t$. With Claim 6.2.9 we get $x \in \chi_{r(y)}$ as desired. If x and y are not neighbors in $\mathcal{H}_P$, then there is a lower-numbered vertex v of x and y in $\mathcal{H}_{P,\pi}$ that led to the introduction of the edge xy. By induction $x, y \in \chi_{r(v)}$. We conclude with Claim 6.2.9 that $x \in \chi_{r(y)}$. This completes the proof of b).

To prove c) consider a vertex $y \in S$. Let x be a higher-numbered neighbor of y in $\mathcal{H}_{P,\pi}$. By construction $x \in S$. Assume first that x and y are in a common S-component. Let $v \in V \setminus S$ be a vertex of this S-component, then, by construction of $(\mathcal{T}, (\chi'_t)_{t \in T})$, the vertices x and y both appear in any bag χ'_t that contains v. We conclude with Claim 6.2.9 that $x \in \chi_{r(y)}$. If x and y are not in a common S-component, then the vertex v that leads to the introduction of the edge xy must be in S by a). Because v is a lower-numbered neighbor of x and y, we have by induction that $x, y \in \chi_{r(v)}$. By Claim 6.2.9 we get $x \in \chi_{r(y)}$ which completes the proof of Claim 6.2.10. ■

We claim that the width of π is at most $(c+1)^3 + (c+1)^2$. As the bags χ'_t have size at most $(c+1)^3 + 1$, every vertex $y \in V \setminus S$ has at most $(c+1)^3$ higher-numbered neighbors in $V \setminus S$ in $\mathcal{H}_{P,\pi}$ by Claim 6.2.10. Furthermore, $y \in V \setminus S$ has by Claim 6.2.10 and Lemma 6.2.6 at most $(c+1)^2$ neighbors in S in $\mathcal{H}_{P,\pi}$. Finally, $y \in S$ has at most $(c+1)^3$ higher-numbered neighbors by Claim 6.2.10 and the bound on the size of the bags. This completes the proof of Lemma 6.2.7. ■

From Proposition 6.2.3 and Theorem 6.1.2 we get the following alternative characterization of S-hypergraphs of bounded arity that allow tractable #CQ.

Theorem 6.2.11. *Let $\mathcal{G}$ be a recursively enumerable class of S-hypergraphs of bounded arity. Assume that $\mathsf{W}[1] \neq \mathsf{FPT}$. Then the following statements are equivalent:*

1. *#CQ on $\mathcal{G}$ is solvable in polynomial time.*
2. *p-#CQ on $\mathcal{G}$ is fixed-parameter tractable.*
3. *There is a constant c such that all S-hypergraphs in $\mathcal{G}$ have elimination width at most c.*

Let us remark that there is a similar notion of elimination width for quantified constraint satisfaction (QCSP) which is a version of CQ in which also universal quantification is allowed. Chen and Dalmau [CD12] introduced this measure and showed that it characterizes the tractable classes of graphs for QCSP. We consider it as likely that an equivalent characterization of the same classes of graphs could be given by treewidth and an adapted notion of S-star size. This would probably also make it possible to get a better understanding

of tractable classes of hypergraphs of unbounded arity for QCSP by exchanging treewidth for e.g. generalized hypertree width.

7 TRACTABLE CONJUNCTIVE QUERIES AND CORES

In this section we will give a more fine-grained analysis of tractable bounded arity #CQ by not considering the underlying hypergraphs as in Chapter 6 but analyzing the queries themselves. It turns out that using certain subqueries called *cores* of conjunctive queries that have essentially already been considered in [CM77], lets us find tractable classes of queries that we do not get from Theorem 6.1.2. Furthermore, we can completely characterize the classes of queries of bounded arity that allow tractable #CQ.

It will be convenient to present the results of this chapter not from the logical perspective used before. Instead we will give them in the homomorphism perspective often used in constraint satisfaction literature (see e.g. [Gro07]).

We will start off this chapter by showing an improved version of Theorem 3.1.3 to showcase some of the techniques used later in this chapter. In Section 7.2 we will introduce the homomophism perspective on #CQ. In Section 7.3 we will show how to use cores of conjunctive queries to completely characterize classes of queries of bounded arity that lead to tractable #CQ.

7.1 WARMUP: AN IMPROVED HARDNESS RESULT FOR #CQ ON STAR-SHAPED QUERIES

Before going into the more technical proofs of this chapter, let us do a small warmup by giving an improved version of Lemma 3.1.4. The proof will use similar ideas as the later proofs of this chapter but will be simpler.

Remember that $\mathcal{G}_{\text{star}}$ is the class of S-graphs (G_n, S) where G_n is the star with n leaves and S consists of all vertices but the the center of G_n (see Example 3.2.7).

Lemma 7.1.1. *Let τ be a vocabulary consisting of a single binary relation symbol $\mathcal{R}$. Then* #CQ *on $\mathcal{G}_{star}$ is* #P*-hard for queries that use only the relation symbol $\mathcal{R}$.*

Let us quickly discuss the difference between Lemma 3.1.4 and Lemma 7.1.1. The former told us that we could construct queries with S-hypergraphs in $\mathcal{G}_{\text{star}}$ such that counting their solutions was #P-hard. In the proof of Lemma 3.1.4 part of the hardness came from the design of the queries and we used in the hardness proof that we could use an unbounded number of relation symbols.

Now Lemma 7.1.1 tells us that we actually do not need an unbounded number of relation symbols for constructing hard instances.

In fact *every* class of queries that has $\mathcal{G}_{\text{star}}$ as its class of associated S-hypergraph yields a hard counting problem. Thus for $\mathcal{G}_{\text{star}}$ an analysis of specific classes of queries does not give us any tractable cases.

We will see that for other classes of of S-hypergraphs the situation will be different. There will be additional tractable classes of queries that we do not get directly by Corollary 5.2.5. We will also see in the remainder of this chapter that in the case of bounded arity we will be able to exactly characterize these new tractable classes of queries.

Proof of Lemma 7.1.1. We reduce from the instances constructed in the proof of Lemma 3.1.4. So let $(\mathcal{A}, \phi)$ be one of those instances. Remember that the query is $\phi := \exists z \bigwedge_{i\in[n]} \mathcal{R}_i(z, y_i)$.

Let $U := \{z, y_1, \ldots, y_n\}$ and $S := \mathsf{free}(\phi) = \{y_1, \ldots, y_n\}$. We construct a new instance $\Psi = (\mathcal{D}, \psi)$ as follows: The query ψ is

$$\psi := \exists z \bigwedge_{i\in[n]} \mathcal{R}(z, y_i).$$

The domain D of $\mathcal{D}$ is $D := U \times A$. We set

$$\mathcal{R}^{\mathcal{D}} := \{((z, v_1), (y_i, v_2)) \mid i \in [n], (v_1, v_2) \in \mathcal{R}_i^{\mathcal{A}}\}.$$

This completes the construction of $\Psi = (\mathcal{D}, \psi)$.

We will show that $|\phi(\mathcal{A})|$ can be computed from $|\psi(\mathcal{D})|$.

To this end, let $\Pi : D \to U$ with $(u, v) \mapsto u$ be the projection onto the first component of D. Let

$$\mathcal{N} := \{a \in \psi(\mathcal{D}) \mid \Pi \circ a = \mathrm{id}\}.$$

Obviously, $\mathcal{N}$ and $\phi(\mathcal{A})$ have the same size, so it suffices to determine $|\mathcal{N}|$.

Let now $\mathcal{N}' := \{a \in \psi(\mathcal{D}) \mid (\Pi \circ a)(S) = S\}$. We claim that $|\mathcal{N}'| = |\mathcal{N}|n!$. To see this observe that $r : \mathcal{N} \times S_n \to \mathcal{N}'$ with $(a, \sigma) \to (a \circ \sigma)$ is a bijection. Thus we have reduced computing $|\phi(\mathcal{A})|$ to computing $|\mathcal{N}'|$.

Now consider assignments $a : S \to D$. First observe that if for any $y_i \in S$ we have $(\Pi \circ a)(y_i) = z$, then a cannot be in $\psi(\mathcal{D})$ by construction of the relation $\mathcal{R}^{\mathcal{D}}$. So we can in the following assume that $(\Phi \circ h)(S) \subseteq S$ for all h we consider.

It follows that only two cases of satisfying assignments in $a \in \psi(\mathcal{D})$ occur: Either $(\Phi \circ a)(S) = S$ or $(\Phi \circ a)(S) \subset S$. The mappings a from the first case form the set $\mathcal{N}'$. We claim that the satisfying a from the second case can be counted easily. Indeed, similarly to Remark 3.1.5 we claim that *all* such mappings $a : S \to D$ satisfy ψ. To see this, recall how the relations of $\mathcal{R}_i$ were constructed in the proof of Lemma 3.1.4. Consider an a with $(\Pi \circ a)(S) \subset S$ and let $y_j \in S$ be such that $y_j \notin (\Pi \circ h)(S)$. Then assigning z the value $(z, (v, v, j, j))$ for arbitrary $v \in V$, where V is the vertex set from the graph in the reduction of Lemma 3.1.4, satisfies all atoms.

Since there are $(|S|^{|S|} - |S|!)|A|^{|S|}$ mappings $a : S \to D$ with $(\Pi \circ a)(S) \subset S$, it follows that

$$|\psi(\mathcal{D})| = |\mathcal{N}'| + (|S|^{|S|} - |S|!)|A|^{|S|}.$$

Thus computing $|\phi(\mathcal{A})|$ reduces to computing $|\psi(\mathcal{D})|$ and thus #CQ ion $\mathcal{G}_{\text{star}}$ over the vocabulary τ is #P-hard. ■

7.2 HOMOMORPHISMS BETWEEN STRUCTURES AND CORES

In this section we introduce a more symmetric view on CQ where we see the queries of instances as structures themselves. The definitions and results in this subsection are mostly taken from [FG06] where the reader can also find more background.

Definition 7.2.1. *To a conjunctive query ϕ over the vocabulary τ we assign a structure $\mathcal{A} = \mathcal{A}_\phi$ called the* natural model *as follows:*

- *the domain of $\mathcal{A}$ is $\mathsf{var}(\phi)$,*
- *$\mathcal{A}$ is over the vocabulary τ, and*
- *for each relation symbol $\mathcal{R} \in \tau$ we set*

$$\mathcal{R}^{\mathcal{A}} := \{\mathsf{var}(\phi') \mid \phi' \in \mathsf{atom}(\phi), \mathcal{R} \text{ relation symbol of } \phi'\}. \qquad \square$$

Note that the definition of the natural model is very similar to that of the hypergraph associated to a query. The main difference that we do not only remember the scope of the atoms but also the vocabulary symbol. This gives us additional information that we can use to isolate larger tractable classes of queries than those guaranteed by Theorem 6.1.2. The general idea is illustrated in the following example.

Example 7.2.2. Consider the query

$$\phi_n := \exists x_1 \dots \exists x_n \exists y_1 \dots \exists y_n \bigwedge_{i,j \in [n]} E(x_i, y_j).$$

When trying to solve CQ-instances $\Phi_n := (\mathcal{B}, \phi_n)$, Theorem 6.1.1 does not help: The hypergraph of ϕ_n is the complete bipartite graph $K_{n,n}$ which has treewidth n, so the runtime of the algorithm of Theorem 6.1.1 will be exponential.

But it is easy to see that Φ_n has a solution if and only if $E^{\mathcal{A}}$ is non-empty, i.e., ϕ_n is equivalent to the atomic subformula $E(x_1, y_1)$. Thus in contrast to what one could get by applying Theorem 6.1.2, the decision problem CQ on instances with the query ϕ_n is actually extremely easy. □

In the remainder of this section we will generalize and formalize the observation of Example 7.2.2.

Definition 7.2.3. *Let $\mathcal{A}$ and $\mathcal{B}$ be two structures over the same vocabulary τ. A* homomorphism *from $\mathcal{A}$ to $\mathcal{B}$ is a function $h : A \to B$ such that for each relation symbol $\mathcal{R} \in \tau$ and each $t = (t_1, \ldots, t_\ell) \in \mathcal{R}^{\mathcal{A}}$ we have $(h(t_1), \ldots, h(t_\ell)) \in \mathcal{R}^{\mathcal{B}}$. We denote the set of homomorphisms from $\mathcal{A}$ to $\mathcal{B}$ by* $\hom(\mathcal{A}, \mathcal{B})$.

A homomorphism h from $\mathcal{A}$ to $\mathcal{B}$ is called an isomorphism *if h is bijective and h^{-1} is a homomorphism from $\mathcal{B}$ to $\mathcal{A}$. We call $\mathcal{A}$ and $\mathcal{B}$* isomorphic *if there is an isomorphism from $\mathcal{A}$ to $\mathcal{B}$. An isomorphism from $\mathcal{A}$ to $\mathcal{A}$ is called* automorphism. □

In the definition above we always assume the structures $\mathcal{A}$ and $\mathcal{B}$ to be over the same vocabulary. If this is not the case, we make the convention that there are no homomorphisms from $\mathcal{A}$ to $\mathcal{B}$ and thus $\hom(\mathcal{A}, \mathcal{B}) = \emptyset$.

We fix a vocabulary τ. Unless stated otherwise, all structures in the remainder of this chapter will be over this vocabulary τ.

Let $\Phi = (\mathcal{B}, \phi)$ be a CQ-instance with domain B. Let moreover $\mathcal{A}_\phi$ be the natural model of ϕ. It is important to note that (by definition) a function $h : \mathsf{var}(\phi) \to B$ is a satisfying assignment of Φ if and only if it is a homomorphism from $\mathcal{A}_\phi$ to $\mathcal{B}$. For this reason, research in constraint satisfaction and Boolean CQ is often stated in this homomorphism perspective in the following way.

CQ **Input:** Two structures $\mathcal{A}$ and $\mathcal{B}$. **Problem:** Decide if $\hom(\mathcal{A}, \mathcal{B})$ is nonempty.

p-CQ **Input:** Two structures $\mathcal{A}$ and $\mathcal{B}$. **Parameter:** $\|\mathcal{A}\|$ **Problem:** Decide if $\hom(\mathcal{A}, \mathcal{B})$ is nonempty.

The *primal graph of a structure* $\mathcal{A}$ is the graph $G_{\mathcal{A}} := (A, E)$ where $uv \in E$ if and only if there is a relation $\mathcal{R}^{\mathcal{A}}$ in $\mathcal{A}$ such that u and v appear together in a tuple of $\mathcal{R}^{\mathcal{A}}$. The *treewidth* of a structure is defined as the treewidth of its primal graph.

As before, we will consider #CQ and p-#CQ on restricted classes of instances. The restriction of the queries in the previous chapters directly corresponds to a restriction of the structure $\mathcal{A}$. For example, one could consider classes of queries of bounded treewidth that directly correspond to classes of structures $\mathcal{A}$ of bounded treewidth. Because $\mathcal{A}$ is on the left side of the homomorphism, these restrictions are also called "left-hand side restrictions". Let $\mathcal{C}$ be a class of structures, then we define the restricted version of the decision problems as follows.

CQ($\mathcal{C}$) **Input:** $\mathcal{A} \in \mathcal{C}$ and a structure $\mathcal{B}$. **Problem:** Decide if $\hom(\mathcal{A}, \mathcal{B})$ is nonempty.

p-CQ($\mathcal{C}$)
Input: $\mathcal{A} \in \mathcal{C}$ and a structure $\mathcal{B}$.
Parameter: $\|A\|$.
Problem: Decide if $\hom(\mathcal{A}, \mathcal{B})$ is nonempty.

The counting versions in the quantifier-free setting are

#CSP($\mathcal{C}$)
Input: $\mathcal{A} \in \mathcal{C}$ and a structure $\mathcal{B}$.
Problem: Compute $\lvert\hom(\mathcal{A}, \mathcal{B})\rvert$.

p-#CSP($\mathcal{C}$)
Input: $\mathcal{A} \in \mathcal{C}$ and a structure $\mathcal{B}$.
Parameter: $\|A\|$.
Problem: Compute $\lvert\hom(\mathcal{A}, \mathcal{B})\rvert$.

In Example 7.2.2 we have seen that there are classes of queries of unbounded treewidth that are tractable. This is because we may have equivalent subqueries that are easier to handle algorithmically than the original queries. This is formalized by the notions of *homomorphic equivalence* and *cores*.

Definition 7.2.4. *Two structures $\mathcal{A}$ and $\mathcal{B}$ are* homomorphically equivalent *if there are homomorphisms from $\mathcal{A}$ to $\mathcal{B}$ and from $\mathcal{B}$ to $\mathcal{A}$.* □

Example 7.2.5. To get a feeling for the nature of homomorphisms and homomorphic equivalence, let us consider the case of graphs. Each graph $G = (V, E)$ naturally gives a structure $\mathcal{A}$ where the domain is $A := V$ and E is the only relation of $\mathcal{A}$. Observe that $G = G_{\mathcal{A}}$, i.e., G is the primal graph of its structure.

Now consider two graphs G, G' with structures $\mathcal{A}$ and $\mathcal{B}$. Then there is a homomorphism from $\mathcal{A}$ to $\mathcal{B}$ if and only if there is a homomorphism from G to G'. Thus, in this setting a homomorphism of structures is simply a homomorphism between graphs (see e.g. [GR01] for more on graph homomorphisms).

Let now G be a bipartite graph and G' any graph that contains at least one edge. Let furthermore $\mathcal{A}$ and $\mathcal{B}$ be the structures of G and G', respectively. Then there is a homomorphism from $\mathcal{A}$ to $\mathcal{B}$. It follows that if G and G' are both bipartite and have an edge then the corresponding structures $\mathcal{A}$ and $\mathcal{B}$ are homomorphically equivalent. □

Note that, as we have seen in Example 7.2.5, homomorphically equivalent structures $\mathcal{A}$ and $\mathcal{B}$ need in general *not* be isomorphic and need not even have the same domain size.

Definition 7.2.6. *A structure is a called* core *if it is not homomorphically equivalent to a proper substructure of itself.*

A structure $\mathcal{B}$ is a core of *a structure $\mathcal{A}$ if*

- $\mathcal{B}$ *is a substructure of* $\mathcal{A}$,
- $\mathcal{B}$ *is homomorphically equivalent to* $\mathcal{A}$, *and*
- $\mathcal{B}$ *is a core.* □

We state a basic property of cores of structures.

Lemma 7.2.7. *Every (finite) structure* $\mathcal{A}$ *has at least one core. Furthermore, every two cores* $\mathcal{B}_1$ *and* $\mathcal{B}_2$ *of* $\mathcal{A}$ *are isomorphic.*

Because of Lemma 7.2.7 we will speak of *the* core of a structure instead of *a* core.

Example 7.2.8. It is easy to see that the structure $\mathcal{A}_n$ of the graph K_n is a core for every $n \in \mathbb{N}$.

Let G be any bipartite graph with at least one edge and the structure $\mathcal{A}$. Then the core of $\mathcal{A}$ is isomorphic to the structure of $K_{1,1}$, the graph consisting of two vertices connected by a single edge. □

Dalmau, Kolaitis and Vardi [DKV02] proved that for the decision problem CQ not the treewidth of instances is important, but it suffices to have bounded treewidth of the cores of the instances.

Theorem 7.2.9 ([DKV02]). *Let* $k \in \mathbb{N}$ *be a fixed constant. Let* $\mathcal{C}_k$ *be the class of all structures with cores of treewidth at most* k. *Then* $\mathrm{CQ}(\mathcal{C}_k)$ *can be decided in polynomial time.*

Grohe [Gro07] showed that this result is optimal.

Theorem 7.2.10 ([Gro07]). *Let* $\mathcal{C}$ *be a recursively enumerable class of structures of bounded arity. Assume* $\mathsf{FPT} \neq \mathsf{W}[1]$. *Then the following statements are equivalent:*

1. $\mathrm{CQ}(\mathcal{C}) \in \mathsf{P}$.
2. $p\text{-}\mathrm{CQ}(\mathcal{C}) \in \mathsf{FPT}$.
3. *There is a constant* c *such that the cores of the structures in* $\mathcal{C}$ *have treewidth at most* c.

Dalmau and Jonsson [DJ04] considered the analogous question for #CSP and found that cores do not help in this setting.

Theorem 7.2.11 ([DJ04]). *Let* $\mathcal{C}$ *be a recursively enumerable class of structures of bounded arity. Assume* $\mathsf{FPT} \neq \#\mathsf{W}[1]$. *Then the following statements are equivalent:*

1. $\#\mathrm{CSP}(\mathcal{C}) \in \mathsf{FP}$.
2. $p\text{-}\#\mathrm{CSP}(\mathcal{C}) \in \mathsf{FPT}$.
3. *There is a constant* c *such that the structures in* $\mathcal{C}$ *have treewidth at most* c.

7.3 TRACTABLE CONJUNCTIVE QUERIES AND CORES

In this section we will give a refinement of Theorem 6.1.2 to capture tractable classes of queries directly instead of characterizing them by their S-hypergraphs. This result will generalize both Theorem 7.2.10 and Theorem 7.2.11. To allow instances with quantification we make the following definition analogous to S-hypergraphs: To each conjunctive query ϕ we assign the pair $(\mathcal{A}, S)$ where $\mathcal{A}$ is the natural model of ϕ and S the set of its free variables. To each class of conjunctive queries $\mathcal{C}$ we assign the corresponding class of pairs.

From a structure $\mathcal{A}$ and a set S of variables it is easy to reconstruct the corresponding query ϕ: $\mathcal{A}$ corresponds to a quantifier free query ϕ' as discussed in the previous section. From ϕ' we get the query ϕ corresponding to $(\mathcal{A}, S)$ by existentially quantifying the variables not in S.

Because of this easy correspondence between queries and pairs $(\mathcal{A}, S)$ with $S \subseteq A$, in a slight abuse of notation, we do not differentiate between pairs $(\mathcal{A}, S)$ and queries in this section. In particular, we will call a pair $\mathcal{A}, S)$ a query, and we will use $\mathcal{C}$ interchangably for classes of queries and classes of pairs $(\mathcal{A}, S)$.

To formulate #CQ in the homomorphism perspective, we make the following definition.

Definition 7.3.1. *For a pair $(\mathcal{A}, S)$, where $\mathcal{A}$ is a structure and $S \subseteq A$, and a structure $\mathcal{B}$ we define* $\hom(\mathcal{A}, \mathcal{B}, S)$ *to be the set of functions $h : S \to B$ that can be extended to homomorphisms $h' : \mathcal{A} \to \mathcal{B}$.* □

Let ϕ be a query with associated pair $(\mathcal{A}, S)$. Then we have $\phi(\mathcal{B}) = \hom(\mathcal{A}, \mathcal{B}, S)$ for every structure $\mathcal{B}$. This allows us to give an alternative formulation of our conjunctive query problems, which we give directly for the restricted problems. So let $\mathcal{C}$ in the following be a class of conjunctive queries.

#CQ($\mathcal{C}$) **Input:** $(\mathcal{A}, S) \in \mathcal{C}$ and a structure $\mathcal{B}$. **Problem:** Compute $\lvert\hom(\mathcal{A}, \mathcal{B}, S)\rvert$.

p-#CQ($\mathcal{C}$) **Input:** $(\mathcal{A}, S) \in \mathcal{C}$ and a structure $\mathcal{B}$. **Parameter:** $\lVert\mathcal{A}\rVert$ **Problem:** Compute $\lvert\hom(\mathcal{A}, \mathcal{B}, S)\rvert$.

Remark 7.3.2. Observe that #CQ contains both CQ and #CSP as special cases: A pair $(\mathcal{A}, S)$ is associated to a quantifier free query if $S = A$ (this corresponds to the special case #CSP). If $S = \emptyset$ then $(\mathcal{A}, S)$ is associated to a fully quantified query (this corresponds to the special case CQ).

We have seen that for CQ cores of instances are crucial while for #CSP they do not matter at all. Thus we introduce a notion of cores for conjunctive queries that interpolates between these two extreme cases. The idea behind the definition is that we require the homomorphisms between $(\mathcal{A}, S)$ and its core to be the identity on the free variables, while they may map the quantified variables in any way that leads to a homomorphism. This is formalized as follows. □

Definition 7.3.3. *To a conjunctive query $(\mathcal{A}, S)$ we assign the* augmented structure *$\mathcal{A}'$ over the augmented vocabulary $\tau \cup \{\mathcal{R}_a \mid a \in S\}$ defined as $\mathcal{A}' := \mathcal{A} \cup \bigcup_{a \in S} \mathcal{R}_a^{\mathcal{A}'}$ where $\mathcal{R}_a^{\mathcal{A}'} := \{a\}$. We call $(\mathcal{A}', S)$ the augmented query of $(\mathcal{A}, S)$. The* core *of $(\mathcal{A}, S)$ is defined as the core of $\mathcal{A}'$.* □

Example 7.3.4. Let $(\mathcal{A}, S)$ be a fully quantified query, i.e., $S = \emptyset$. Then the core of $(\mathcal{A}, S)$ is the core of $\mathcal{A}$.

If $(\mathcal{A}, S)$ is quantifier free, i.e., $S = A$, then the core of $(\mathcal{A}, S)$ equals $\mathcal{A}$.

Thus our notion of cores of conjunctive queries really does what we asked for in Remark 7.3.2. □

Lemma 7.3.5. *Let $(\mathcal{A}, S)$ be a conjunctive query such that the augmented structure $\mathcal{A}'$ is a core. Then every homomorphism $h : \mathcal{A} \to \mathcal{A}$ with $h|_S = \mathrm{id}$ is a bijection.*

Proof. Clearly, h is also a homomorphism $h : \mathcal{A}' \to \mathcal{A}'$, because $h(a) = a \in \mathcal{R}_a^{\mathcal{A}'}$ for every $a \in S$. But by assumption $\mathcal{A}'$ is a core, so there is no homomorphism from $\mathcal{A}'$ to a proper substructure and thus h must be a bijection on $\mathcal{A}'$ and consequently also on $\mathcal{A}$. ■

The cores of conjunctive queries were essentially already studied by Chandra and Merlin in a seminal paper [CM77] although the notation used there is different. We formulate some fundamental results on conjunctive queries.

Definition 7.3.6. *We call two queries $(\mathcal{A}_1, S)$ and $(\mathcal{A}_2, S)$* equivalent *if for each structure $\mathcal{B}$ we have*

$$\hom(\mathcal{A}_1, \mathcal{B}, S) = \hom(\mathcal{A}_2, \mathcal{B}, S).$$ □

Theorem 7.3.7 ([CM77]). *If two conjunctive queries $(\mathcal{A}_1, S)$ and $(\mathcal{A}_2, S)$ have the same core, then they are equivalent.*

The following lemma seems to be folklore. A proof can be found e.g. in [FG06].

Lemma 7.3.8. *Let $\mathcal{A}$ and $\mathcal{B}$ be two homomorphically equivalent structures, and let $\mathcal{A}'$ and $\mathcal{B}'$ be cores of $\mathcal{A}$ and $\mathcal{B}$, respectively. Then $\mathcal{A}'$ and $\mathcal{B}'$ are isomorphic.*

We now assign another structure $\mathcal{A}^*$ to a query $(\mathcal{A}, S)$:

Definition 7.3.9. *To a conjunctive query $(\mathcal{A}, S)$ we assign the structure $\mathcal{A}^*$ over the vocabulary $\tau \cup \{\mathcal{R}_a \mid a \in A\}$ defined as $\mathcal{A}^* := \mathcal{A} \cup \bigcup_{a \in A} \mathcal{R}_a^{\mathcal{A}^*}$ where $\mathcal{R}_a^{\mathcal{A}^*} := \{a\}$.* □

Note that $\mathcal{A}'$ and $\mathcal{A}^*$ differ in which relations we add: For $\mathcal{A}'$ we add $\mathcal{R}_a^{\mathcal{A}'}$ for variables $a \in S$ while for $\mathcal{A}^*$ we add $\mathcal{R}_a^{\mathcal{A}^*}$ for all $a \in A$. Thus, $\mathcal{A}^*$ in general has more relations than $\mathcal{A}'$.

We now formulate the main technical lemma of this section whose proof uses ideas from [DJ04].

Lemma 7.3.10. *Let $\mathcal{C}$ be a class of conjunctive queries such that for each $(\mathcal{A}, S) \in \mathcal{C}$ the augmented structure $\mathcal{A}'$ is a core. Let $\mathcal{C}^* := \{(\mathcal{A}^*, S) \mid (\mathcal{A}, S) \in \mathcal{C}\}$. Then there is a parameterized T-reduction from p-#CQ($\mathcal{C}^*$) to p-#CQ($\mathcal{C}$).*

Proof. Let $((\mathcal{A}^*, S), \mathcal{B})$ be an input for #CQ($\mathcal{C}^*$). Remember that $\mathcal{A}^*$ and $\mathcal{B}$ are structures over the vocabulary $\tau \cup \{\mathcal{R}_a \mid a \in A\}$. By the definition of $\mathcal{A}^*$ we get the corresponding structure $\mathcal{A}$ with $(\mathcal{A}, S) \in \mathcal{C}$ by deleting the the relations $\mathcal{R}_a^{\mathcal{A}^*}$ for $a \in A$. We will reduce the computation of $|\hom(\mathcal{A}^*, \mathcal{B}, S)|$ to the computation of $|\hom(\mathcal{A}, \mathcal{B}', S)|$ for different structures $\mathcal{B}'$.

Let $D := \{(a, b) \in A \times B \mid b \in \mathcal{R}_a^{\mathcal{B}}\}$ and define a structure $\mathcal{D}$ over the vocabulary τ with the domain D that contains for each relation symbol $\mathcal{R} \in \tau$ the relation

$$\begin{aligned}\mathcal{R}^{\mathcal{D}} := \{((a_1, b_1), \ldots, (a_r, b_r)) \mid \\ (a_1, \ldots, a_r) \in \mathcal{R}^{\mathcal{A}}, (b_1, \ldots, b_r) \in \mathcal{R}^{\mathcal{B}}, \\ \forall i \in [r] : (a_i, b_i) \in D\}.\end{aligned}$$

Let furthermore $\Pi : D \to A$ be the projection onto the first coordinate, i.e., $\Pi(a, b) := a$. Observe that Π is by construction of $\mathcal{D}$ a homomorphism from $\mathcal{D}$ to $\mathcal{A}$.

We will several times use the following claim:

Claim 7.3.11. *Let $h \in \hom(\mathcal{A}, \mathcal{D})$ with $h(S) = S$. Then $\Pi \circ h$ is an automorphism of $\mathcal{A}$.*

Proof. Let $g := \Pi \circ h$. As the composition of two homomorphisms, g is a homomorphism from $\mathcal{A}$ to $\mathcal{A}$. Furthermore, by assumption $g|_S$ is a bijection from S to S. Since S is finite, there is $i \in \mathbb{N}$ such that $g^i|_S = \mathrm{id}$. But g^i is a homomorphism and thus, by Lemma 7.3.5, g^i is a bijection. It follows that g is a bijection.

Since A is finite, there is $j \in \mathbb{N}$ such that $g^{-1} = g^l$. It follows that g^{-1} is a homomorphism and thus g is an automporphism. ■

Let $\mathcal{N}$ be the set of mappings $h : S \to D$ with $\Pi \circ h = \mathrm{id}$ that can be extended to a homomorphism $h' : \mathcal{A} \to \mathcal{D}$.

Claim 7.3.12. *There is a bijection between* $\hom(\mathcal{A}^*, \mathcal{B}, S)$ *and* $\mathcal{N}$.

Proof. For each $h^* \in \hom(\mathcal{A}^*, \mathcal{B}, S)$ we define $P(h^*) := h$ by $h(a) := (a, h^*(a))$ for $a \in S$. From the extension of h^* to A we get an extension of h that is a homomorphism and thus $h \in \mathcal{N}$. Thus P is a mapping $P : \hom(\mathcal{A}^*, \mathcal{B}, S) \to \mathcal{N}$.

We claim that P is a bijection. Clearly, P is injective. We we will show that it is surjective as well. To this end, let $h : S \to D$ be a mapping in $\mathcal{N}$ and let $h_e \in \hom(\mathcal{A}, \mathcal{D})$ be an extension of h which exists by definition of $\mathcal{N}$. By Claim 7.3.11 we have that $\Pi \circ h_e$ is an automorphism, and thus $(\Pi \circ h_e)^{-1}$ is a homomorphism. We set $h'_e := h_e \circ (\Pi \circ h_e)^{-1}$. Obviously, h'_e is a homomorphism in $\hom(\mathcal{A}, \mathcal{D})$, because h'_e is the composition of two homomorphisms. Furthermore, for all $a \in S$ we have $h'_e(a) = (h_e \circ (\Pi \circ h_e))(a) = (h_e \circ (\Pi \circ h))(a) = h_e(a) = h(a)$, so h'_e is an extension of h. Moreover $\Pi \circ h'_e = (\Pi \circ h_e) \circ (\Pi \circ h_e)^{-1} = \mathrm{id}$. Hence, we have $h'_e = \mathrm{id} \times \bar{h}$ for a homomorphism $\bar{h} : \mathcal{A} \to \overline{\mathcal{B}}$, where $\overline{\mathcal{B}}$ is the structure we get from $\mathcal{B}$ by deleting the relations $\mathcal{R}_a^{\mathcal{B}}$ for $a \in A$. But by definition $h'_e(a) \in D$ for all $a \in A$ and thus $\bar{h}(a) \in \mathcal{R}_a^{\mathcal{B}}$. It follows that $\bar{h} \in \hom(\mathcal{A}^*, \mathcal{B})$. We set $h^* := \bar{h}|_S$. Clearly, $h^* \in \hom(\mathcal{A}^*, \mathcal{B}, S)$ and $P(h^*) = h$. It follows that P is surjective. This proves the claim. ■

Let I be the set of mappings $g : S \to S$ that can be extended to an automorphism of $\mathcal{A}$. Let $\mathcal{N}'$ be the set of mappings $h : S \to D$ with $(\Pi \circ h)(S) = S$ that can be extended to homomorphisms $h' : \mathcal{A} \to \mathcal{D}$.

Claim 7.3.13.

$$|\hom(\mathcal{A}^*, \mathcal{B}, S)| = \frac{|\mathcal{N}'|}{|I|}.$$

Proof. Because of Claim 7.3.12 it is sufficient to show that

$$|\mathcal{N}'| = |\mathcal{N}||I|. \tag{2}$$

We first prove that

$$\mathcal{N}' = \{f \circ g \mid f \in \mathcal{N}, g \in I\}. \tag{3}$$

The $\supseteq$ direction is obvious. For the other direction let $h \in \mathcal{N}'$. Let h' be the extension of h that is a homomorphism $h' : \mathcal{A} \to \mathcal{D}$. By Claim 7.3.11, we have that $g := \Pi \circ h'$ is an automorphism of $\mathcal{A}$. It follows that $g^{-1}|_S \in I$. Furthermore, $h \circ g^{-1}|_S$ is a mapping from S to D and $h' \circ g^{-1}$ is an extension that is a homomorphism from $\mathcal{A}$ to $\mathcal{D}$. Furthermore $(\Pi \circ h' \circ g^{-1}|_S)(a) = (g|_S \circ g^{-1}|_S)(a) = a$ for every $a \in S$ and hence $h' \circ g^{-1}|_S \in \mathcal{N}$ and $h = h \circ g^{-1}|_S \circ g|_S$ which proves the claim (3).

To show (2), we claim that for every $h, h' \in \mathcal{N}$ and every $g, g' \in I$, if $f \neq f'$ or $g \neq g'$, then $f \circ g \neq f' \circ g'$. To see this, observe that f can always be written as $f = \mathrm{id} \times f_2$ and thus $(f \circ g)(a) = (g(a), f_2(g(a)))$.

Thus, if g and g' differ, $\Pi \circ f \circ g \neq \Pi \circ f' \circ g'$ and thus $f \circ g \neq f' \circ g'$. Also, if $g = g'$ and $f \neq f'$, then clearly $f \circ g \neq f' \circ g'$. This completes the proof of (2) and the claim. ■

Clearly, the set I depends only on $(\mathcal{A}, S)$ and thus it can be computed by an FPT-algorithm. Thus it suffices to show how to compute $|\mathcal{N}'|$ in the remainder of the proof.

For each set $T \subseteq S$ we define $\mathcal{N}_T := \{h \in \hom(\mathcal{A}, \mathcal{D}, S) \mid (\Pi \circ h)(S) \subseteq T\}$. We have by inclusion-exclusion

$$|\mathcal{N}'| = \sum_{T \subseteq S} (-1)^{|S \setminus T|} |\mathcal{N}_T|. \tag{4}$$

Observe that there are only $2^{|S|}$ summands in (4) and thus if we can reduce all of them to #CQ with the query $(\mathcal{A}, S)$ this will give us the desired parameterized T-reduction.

We will now show how to compute the $\mathcal{N}_T$ by interpolation. So fix a $T \subseteq S$. Let $\mathcal{N}_{T,i}$ for $i = 0, \ldots, |S|$ be

$$\mathcal{N}_{T,i} := \{h \in \hom(\mathcal{A}, \mathcal{D}, S) \mid |(\Pi \circ h)(S) \cap T| = i\},$$

i.e., $\mathcal{N}_{T,i}$ consists of the mappings $h \in \hom(\mathcal{A}, \mathcal{D}, S)$ such that there are exactly i elements $a \in S$ that are mapped to $h(a) = (a', b)$ such that $a' \in T$. Obviously, $\mathcal{N}_T = \mathcal{N}_{T,|S|}$ with this notation.

Now for each $j = 1, \ldots, |S|$ we construct a new structure $\mathcal{D}_{j,T}$ over the domain $D_{j,T}$. To this end, let $a^{(1)}, \ldots, a^{(j)}$ be copies of $a \in T$ that are not in D. Then we set

$$\begin{aligned} D_{j,T} := \{(a^{(k)}, b) \mid (a, b) \in D, a \in T, k \in [j]\} \\ \cup \{(a, b) \mid (a, b) \in D, a \notin T\}. \end{aligned}$$

We define a mapping $B : D \to \mathcal{P}(D_{j,T})$, where $\mathcal{P}(D_{j,T})$ is the power set of $D_{j,T}$, by

$$B(a, b) := \begin{cases} \{(a^{(k)}, b) \mid k \in [j]\}\}, & \text{if } a \in T \\ \{(a, b)\}, & \text{otherwise.} \end{cases}$$

For every relation symbol $\mathcal{R} \in \tau$ we define

$$\mathcal{R}^{\mathcal{D}_{T,j}} := \bigcup_{(d_1, \ldots, d_s) \in \mathcal{R}^D} B(d_1) \times \ldots \times B(d_2).$$

Then every $h \in \mathcal{N}_{T,i}$ corresponds to i^j mappings in $\hom(\mathcal{A}, \mathcal{D}_{j,T}, S)$. Thus for each j we get

$$\sum_{i=1}^{|S|} i^j |\mathcal{N}_{T,i}| = |\hom(\mathcal{A}, \mathcal{D}_{j,T}, S)|.$$

This is a linear system of equations and the corresponding matrix is a Vandermonde matrix, so $\mathcal{N}_T = \mathcal{N}_{T,|S|}$ can be computed with an oracle

for #CQ on the instances $((\mathcal{A}, S), \mathcal{D}_{j,T})$. The size of the linear system depends only on $|S|$. Furthermore, $\|D^j\| \leq \|D\| j^s \leq \|D\|^{s+1}$ where s is the bound on the arity of the relations symbols in τ and thus a constant. If follows that the algorithm described above is the desired parameterized T-reduction. This completes the proof of Lemma 7.3.10.■

We will use a lemma that is probably well known, but as we could not find a reference, we give a proof for it.

Lemma 7.3.14. *Let $k \in \mathbb{N}$ be a fixed constant. Let $\mathcal{A}$ be a structure having a core with treewidth at most k. Then we can compute a core of $\mathcal{A}$ in time polynomial in $\|\mathcal{A}\|$.*

Proof. The proof is based on well-known query minimization techniques already pioneered in [CM77]. The basic observation is that if $\mathcal{A}$ is not a core, then there is a substructure $\mathcal{A}_s$, that we get by deleting a tuple from a relation of $\mathcal{A}$, that contains a core of $\mathcal{A}$. Trivially, $\mathcal{A}_s$ is homomorphically equivalent to $\mathcal{A}$. Thus by Lemma 7.3.8, for every substructure $\mathcal{A}_s$ of $\mathcal{A}$ that is homomorphically equivalent to $\mathcal{A}$, the core of $\mathcal{A}_s$ is also a core of $\mathcal{A}$.

The construction of a core $\mathcal{A}_c$ goes as follows: For every tuple t in every relation check, using the algorithm of Theorem 7.2.9, if the structure we get from $\mathcal{A}$ by deleting t is homomorphically equivalent to $\mathcal{A}$. If there is such a tuple t, delete it from $\mathcal{A}$ and iterate the process, until no tuple can be deleted anymore.

By the discussion above the end result $\mathcal{A}_c$ of this procedure must be a core. Furthermore, $\mathcal{A}_c$ is homomorphically equivalent to $\mathcal{A}$ and a substructure of $\mathcal{A}$, so $\mathcal{A}_c$ is a core of $\mathcal{A}$. Finally, at most $\|\mathcal{A}\|$ tuples get deleted and for every deleted tuple the algorithm has to perform at most $\|\mathcal{A}\|$ homomorphism test. The left hand sides of these test all have the same core of treewidth at most k and the right hand sides have size at most $\|\mathcal{A}\|$. Using Theorem 7.2.9 then gives a runtime polynomial in $\|\mathcal{A}\|$. ■

We make an observation similar to Observation 6.1.3 For a conjunctive query $(\mathcal{A}, S)$ with core $\mathcal{A}_c$ we define $w(\mathcal{A}, S)$ as the sum of the treewidth of $\mathcal{A}_c$ and the quantified star size of of the core of $(\mathcal{A}_c, S)$.

Observation 7.3.15. *Let $\mathcal{C}$ be a recursively enumerable class of conjunctive queries such that p-#CQ($\mathcal{C}$) is fixed-parameter tractable and the values of w on the instances of $\mathcal{C}$ is unbounded. Then there is a* recursive *class $\mathcal{C}'$ of conjunctive queries such that p-#CQ($\mathcal{C}'$) is fixed-parameter tractable, the value of w on the instances of $\mathcal{C}'$ is unbounded and there is a Turing machine M' that enumerates $\mathcal{C}'$ in an order $(\mathcal{H}'_1, S'_1), (\mathcal{H}'_2, S'_2), \ldots$ such that the function $i \mapsto w(\mathcal{H}_i, S_i)$ is increasing.*

Proof. Let M be a Turing-machine that enumerates the queries in $\mathcal{C}$. Let $(\mathcal{A}_1, S_1), (\mathcal{A}_2, S_2), \ldots$ be the order in which M enumerates the

queries in $\mathcal{C}$. We define $\mathcal{C}' := \{(\mathcal{A}'_1, S'_1), (\mathcal{A}'_2, S'_2), \ldots\}$ by $(\mathcal{A}'_1, S'_1) := (\mathcal{A}_1, S_1)$ and $(\mathcal{A}'_i, S'_i) := (\mathcal{A}_{n(i)}, S_{n(i)})$ where $n(i) := \min\{j \in \mathbb{N} \mid j > n(i-1), w(\mathcal{A}_j, S_j) > w(\mathcal{A}'_{i-1}, S'_{i-1})\}$. Because w is unbounded on the instances in $\mathcal{C}$, we have that $\mathcal{C}'$ is infinite and w is unbounded on the instances of $\mathcal{C}'$.

From M it is easy to construct a Turing-machine M' that enumerates $\mathcal{C}'$ in the order $(\mathcal{A}'_1, S'_1), (\mathcal{A}'_2, S'_2), \ldots$. Furthermore, there is an algorithm that decides membership in $\mathcal{C}'$: On an input $(\mathcal{A}, S)$, enumerate $\mathcal{C}'$ with M' until $(\mathcal{A}, S)$ is found or M' enumerates a query $(\mathcal{A}', S')$ such that $w(\mathcal{A}', S') > w(\mathcal{A}, S)$. In the first case the algorithm accepts, in the second case the algorithm rejects. Thus $\mathcal{C}'$ is recursive.

Finally, all queries in $\mathcal{C}'$ are by construction also in $\mathcal{C}$, so #CQ($\mathcal{C}'$) is fixed-parameter tractable. ■

We can now finally fully characterize the tractable classes queries for #CQ of bounded arity.

Theorem 7.3.16. *Let $\mathcal{C}$ be a recursively enumerable class of conjunctive queries of bounded arity. Assume* FPT $\neq$ W[1]. *Then the following statements are equivalent:*

1. #CQ($\mathcal{C}$) $\in$ FP.
2. *p*-#CQ($\mathcal{C}$) $\in$ FPT.
3. *There is a constant c such that the cores of the queries in $\mathcal{C}$ have quantified star-size and treewidth at most c.*

Proof. 1 $\to$ 2 is trivial.

2 $\to$ 3: By way of contradiction, assume that there is a class $\mathcal{C}$ of conjunctive queries of unbounded treewidth or unbounded quantified star size such that *p*-#CQ($\mathcal{C}$) is fixed-parameter tractable. We will show that there is then also a class $\mathcal{G}$ of S-hypergraphs of unbounded treewidth or unbounded S-star size such that #CQ on $\mathcal{G}$ is fixed-parameter tractable. This is a contradiction with Theorem 6.1.2.

Obviously, the function w is unbounded on $\mathcal{C}$ and thus we assume w.l.o.g. that $\mathcal{C}$ has the properties of the class known to exist from Observation 7.3.15. In particular, there is a machine M' that enumerates $\mathcal{C}$ by increasing value of w.

For each $(\mathcal{A}, S) \in \mathcal{C}$ we construct a structure $\overline{\mathcal{A}}$ as follows: Construct the augmented structure $\mathcal{A}'$ of $\mathcal{A}$ and compute its core $\overline{\mathcal{A}}'$. Then we define $\overline{\mathcal{A}}$ to be the structure that we get by deleting the relations $\mathcal{R}_a$ for $a \in S$ that we added in the construction of $\mathcal{A}'$. We set

$$\overline{\mathcal{C}} := \{(\overline{\mathcal{A}}, S) \mid (\mathcal{A}, S) \in \mathcal{C}\}.$$

Obviously, w is unbounded on $\overline{\mathcal{C}}$. Furthermore, observe that $\overline{\mathcal{C}}$ is recursive: For a query $(\overline{\mathcal{A}}, S)$ simply enumerate $\mathcal{C}$ until a corresponding query $(\mathcal{A}, S) \in \mathcal{C}$ is found or M' enumerates a query $(\mathcal{A}', S')$ with $w(\mathcal{A}', S') > w(\overline{\mathcal{A}}, S)$. Observe that this also gives an algorithm that, given a query $(\overline{\mathcal{A}}, S) \in \overline{\mathcal{C}}$, computes a corresponding $(\mathcal{A}, S) \in \mathcal{C}$.

Claim 7.3.17. *#CQ($\overline{\mathcal{C}}$) is fixed-parameter tractable.*

Proof. We will show a parameterized parsimonious reduction from p-#CQ($\overline{\mathcal{C}}$) to p-#CQ($\mathcal{C}$).

First, we claim that there is a computable function g such that to an instance $(\overline{\mathcal{A}}, S) \in \overline{\mathcal{C}}$ we can in time $g(|\overline{\mathcal{A}}|)$ find a corresponding $(\mathcal{A}, S) \in \mathcal{C}$. We have already seen that there is an algorithm that to any individual $(\overline{\mathcal{A}}, S) \in \overline{\mathcal{C}}$ computes $(\mathcal{A}, S) \in \mathcal{C}$. Let $f(\overline{\mathcal{A}}, S)$ be the time that the algorithm needs for this computation. We set $g(k) := \max_{(\overline{\mathcal{A}},S)}(f(\overline{\mathcal{A}}, S))$ where the maximum is over all $(\overline{\mathcal{A}}, S) \in \overline{\mathcal{C}}$ with $\|\mathcal{A}\| = k$. Since $\overline{\mathcal{C}}$ is recursive, it follows that g is computable.

We have that $\overline{\mathcal{A}}$ is a substructure of $\mathcal{A}$ and there is a homomorphism from $\mathcal{A}$ to $\overline{\mathcal{A}}$, because there is a homomorphism from $\mathcal{A}'$ to $\overline{\mathcal{A}}'$. Hence, $\mathcal{A}$ and $\overline{\mathcal{A}}$ are homomorphically equivalent and by Theorem 7.3.7 we have that $(\mathcal{A}, S)$ and $(\overline{\mathcal{A}}, S)$ are equivalent.

Thus there is indeed a parameterized parsimonious reduction from p-#CQ($\overline{\mathcal{C}}$) to p-#CQ($\mathcal{C}$) and it follows that p-#CQ($\overline{\mathcal{C}}$) is fixed-parameter tractable. ■

Let $\mathcal{C}^* := \{(\overline{\mathcal{A}}^*, S) \mid (\overline{\mathcal{A}}, S) \in \overline{\mathcal{C}}\}$. By Lemma 7.3.10 there is a parameterized T-reduction from p-#CQ($\mathcal{C}^*$) to p-#CQ($\overline{\mathcal{C}}$) and thus p-#CQ($\mathcal{C}^*$) is fixed-parameter tractable.

Let now $\mathcal{G}$ be the class of S-hypergraphs of the instances in $\overline{\mathcal{C}}$. By definition the S-star size or the treewidth of $\mathcal{G}$ is unbounded.

Claim 7.3.18. *#CQ on $\mathcal{G}$ is fixed-parameter tractable.*

Note that Claim 7.3.18 is a contradition to Theorem 6.1.2, because, as we have already observed, the treewidth of the S-star size of $\mathcal{G}$ is unbounded. Thus it only remains to prove Claim 7.3.18.

Proof of Claim 7.3.18. We will show that there is a parameterized parsimonious reduction from #CQ on $\mathcal{G}$ to #CQ($\mathcal{C}^*$). To this end, let $(\mathcal{B}, \phi)$ be a #CQ-instance such that the S-hypergraph $(\mathcal{H}, S)$ of ϕ is in $\mathcal{G}$. We assume w.l.o.g. that every scope appears only once in ϕ. If this is not the case, we can substitute atoms with the same scope by one atom whose relation is the intersection of the relation of the original atoms. This does not change the S-hypergraph associated to ϕ, so this new instance still has the S-hypergraph $(\mathcal{H}, S)$.

By similar arguments as before, there is a computable function g' such that we can compute in time $g'(|\phi|)$ a query $(\overline{\mathcal{A}}, S) \in \overline{\mathcal{C}}$ that has the S-hypergraph $(\mathcal{H}, S)$.

Let the vocabulary of $(\overline{\mathcal{A}}, S)$ be τ.

We now construct a structure $\overline{\mathcal{B}}$ over the same relation symbols as $\overline{\mathcal{A}}^*$, i.e., over the vocabulary $\tau \cup \{\mathcal{R}_x \mid x \in \overline{A}\}$. The structure $\overline{\mathcal{B}}$ has

the domain $\overline{B} := A \times B$ where $A := \mathsf{var}(\phi)$ is the domain of $\mathcal{A}$ and B is the domain of $\mathcal{B}$. For $\overline{\mathcal{R}} \in \tau$ we set

$$\overline{\mathcal{R}}^{\overline{\mathcal{B}}} := \{((x_{i_1}, a_1), \ldots, (x_{i_k}, a_k)) \mid (x_{i_1}, \ldots, x_{i_k}) \in \overline{\mathcal{R}}^{\overline{\mathcal{A}}}, \\ \mathcal{R}(x_{i_1}, \ldots, x_{i_k}) \in \mathsf{atom}(\phi), \\ (a_1, \ldots, a_k) \in \mathcal{R}^{\mathcal{B}}\}.$$

Furthermore, for the relations symbols $\overline{\mathcal{R}}_x$ that are added in the construction of $\overline{\mathcal{A}}^*$ from $\overline{\mathcal{A}}$ we set

$$\overline{\mathcal{R}}_x^{\overline{\mathcal{B}}} := \{(x, a) \mid a \in B\}$$

where B is the domain of $\mathcal{B}$.

Let ϕ' be the query we get from ϕ by deleting all quantifiers. It is easy to see that from a satisfying assignment $h \in \phi'(\mathcal{B})$ we get a homomorphism $h' : \overline{\mathcal{A}}^* \to \overline{\mathcal{B}}$ by setting $h'(x) := (x, h(x))$. Furthermore, this construction is obviously bijective. Thus we get $|\phi(\mathcal{B})| = |\hom(\overline{\mathcal{A}}^*, \overline{\mathcal{B}}, S)|$. Since $\|\overline{\mathcal{B}}\|$ is polynomial in $|\phi|$ and $\|\mathcal{B}\|$, the construction is a parameterized parsimonious reduction from #CQ on $\mathcal{G}$ to #CQ($\mathcal{C}^*$). This completes the proof of Claim 7.3.18. ■

3 → 1: Let $((\mathcal{A}, S), \mathcal{B})$ be an instance of #CQ($\mathcal{C}$) with domain A. By assumption the treewidth and the quantified star size of the core of $(\mathcal{A}, S)$ are at most c. We simply compute the core of $(\mathcal{A}, S)$ with Lemma 7.3.14 and then use Theorem 7.3.7 and Theorem 6.1.2 to solve the instance in polynomial time. ■

8

CONCLUSION

In this part of this thesis we have achieved a very fine understanding of tractability for #CQ. Starting off from the general hardness results in [BCC+05] and [PS13] we have analysed the hard cases and used our understanding of them to distill the parameter quantified star size that characterizes the tractable cases for large classes of fragments defined by well-known decomposition techniques. Of course there remain numerous open questions and we will present several of them in this final chapter of this part.

First, we did not completely characterize the tractable cases for #CQ of unbounded arity. This is not surprising, because the analogous question for CQ is not understood so far, either. There is however a characterization of the fixed parameter tractable classes of CQ by Dániel Marx [Mar10]: The parameter "submodular width" introduced by Marx makes CQ fixed-parameter tractable and it is the most general parameter to do so—assuming the exponential time hypothesis (see 5.1). It would be interesting to see if #CQ-instances of bounded submodular width and bounded quantified star size are tractable. This would then solve the complexity of p-#CQ completely. Since the techniques of Marx are quite involved it is not apparent if they transfer to counting.

Another direction of future research is trying to adapt our star size techniques to different problems. For example, one could try to generalize the results of this part of the thesis by extending the logical fragment that the queries can be formulated in. Just recently, Chen and Dalmau [CD12] have characterized the tractable classes of bounded arity QCSP which is a version of CQ in which also universal quantifiers are allowed. They do this by introducing a new width measure based on elimination orders for first order $\{\forall, \exists, \wedge\}$-formulas. As already discussed in Section 6.2 it appears likely that the tractable fragment found by Chen and Dalmau can also be characterized by an adapted version of quantified star size. This alternative characterization would then maybe make it easier to understand tractable fragments of unbounded arity.

Another extension of conjunctive queries appears in a recent paper by Chen [Che12] where he considers existential formulas that may use conjunction and disjunction. This is particularly interesting, because it corresponds to the classical select-project-join queries with union that play an important role in database theory (see e.g. the textbook [AHV95]). One may wonder if techniques used in this thesis may help to understand the complexity of this class of queries better.

Another basic question is the role of negation in queries. If we allow negation in front of the atoms of a conjunctive query the decomposition techniques of Chapter 2 do not yield tractable instances anymore. Indeed, it is easy to see that if we allow such negation even quantifier free, acyclic instances become NP-hard and #P-hard [SS10]. Ordyniak et al. [OPS10] have shown that a restriction of acyclicity called β-acyclicity leads to tractable SAT. Brault-Baron generalized this to NCQ—a version of CQ in which all atoms are negated—and showed that β-acyclic instances cannot only be solved in polynomial time but even in quasi-linear time. He also formulated a partial converse by proving that if there is any non-β-acyclic query that allows quasi-polynomial decision, then there are very fast algorithms for detecting triangles in graphs. So far there is no known way to extend β-acyclicity ot a width measure that yields tractable NCQ, but there are several incomparable width measures based on treewidth or cliquewidth that make NCQ tractable (see e.g. [PSS13]). For counting it is not known if β-acyclicity is helpful; even the question if counting for quantifier free NCQ-instances is tractable is open.

Part II

UNDERSTANDING ARITHMETIC CIRCUIT CLASSES

9 INTRODUCTION AND PRELIMINARIES

9.1 INTRODUCTION

In this part of the thesis we change the underlying model of computation from Turing machinges to *arithmetic circuits*. Arithmetic circuit complexity is a classical area proposed by Valiant [Val79, Val82]. Over that last decades it has attracted considerable attention and is still a very active field.

Our main interest in this part will be to understand efficient computation in the arithmetic circuit setting which is formalized by the class VP. This class consists of families of polynomials of polynomial degree that are computed by arithmetic circuits of polynomial size. Consider for example the family $(\det_n)_{n\in\mathbb{N}}$ where $\det_n$ is the determinant of the $(n \times n)$-matrix $(X_{ij})_{1\leq i,j\leq n}$ where the X_{ij} are variables. Then $(\det_n)$ lies in VP.

Despite its apparently natural definition there is one irritating aspect in which VP differs from other arithmetic circuit classes: There are no known natural complete problems for VP—artificial ones can be constructed (see e.g. [Mal07])—and prior to the work presented in this part of this thesis there had been no natural characterizations of VP that did not in one form or another depend on circuits. This puzzling feature of VP raises the question whether VP is indeed the right class for measuring natural efficient computability. This scepticism is further strengthened by the fact that Malod and Portier [MP08] have shown that many natural problems from linear algebra are complete for VP_{ws}, a subclass of VP. Thus the search for complete problems or natural characterizations of VP is an interesting and meaningful problem in algebraic complexity. In this part of this thesis we give several such natural characterization of VP and other classes, none of them depending on arithmetic circuits directly.

The first set of results will connect arithmetic circuit classes to the conjunctive queries of Part i. We will see in Chapter 10 that conjunctive queries can be used to give circuit-free characterizations of all arithmetic circuit classes commonly considered in the literature. It follows that all counting results of Part i can be turned into results for weighted counting which is often considered in the counting complexity literature (see e.g. [CC12]).

In Chapter 11 we consider polynomials defined as generating functions of graph properties. For most interesting graph properties these polynomials are known to be VNP-complete [Büo0, BK09] and thus conjectured to be intractable. We will follow the idea by Flarup et

al. in [FKL07] and consider graph polynomials for graphs of bounded treewidth. This restriction will for several graph polynomials allow us to characterize VP_e, the class of families of polynomials computed by polynomial size *arithmetic formulas*. We will also disprove a conjecture of Lyaudet [Lya07] by showing that some polynomial familes that are VNP-complete for general graphs fail to capture VP_e on bounded treewidth graphs.

Finally, in Chapter 12 we extend the well known characterization of VP_{ws} as the class of polynomials computed by polynomial size *arithmetic branching programs* [MP08] to other complexity classes. In order to do so we add additional memory to the computation of branching programs to make them more expressive. We show that allowing different types of memory in branching programs increases the computational power even for constant width programs. In particular, this leads to very natural and robust characterizations of VP and VNP by branching programs with memory.

We will only consider some small parts of the rich field of arithmetic circuit complexity in this and the next part of this thesis. For a larger overview of the field see the textbooks [BCS97, Bü00] and the recent surveys [SY10, Mah12].

9.2 SOME BACKGROUND ON ARITHMETIC CIRCUIT COMPLEXITY

We present a short introduction into the setting of arithmetic circuit complexity as proposed by Valiant [Val79, Val82].

An *arithmetic circuit* over a field $\mathbb{F}$ is a labeled directed acyclic graph (DAG) consisting of vertices or gates with indegree or fanin 0 or 2. The gates with fanin 0 are called input gates and are labeled with constants from $\mathbb{F}$ or variables $X_1, X_2, \ldots, X_n$. The gates with fanin 2 are called computation gates and are labeled with $\times$ or $+$.

In most parts of this thesis we will not specify which field $\mathbb{F}$ we are computing in. All proofs will be valid for any field of characteristic different from 2. Most results could also be proved for characteristic 2, but as this requires some cumbersome work and does not lead to many new insights, we will not carry this out.

We sometimes also consider circuits in which gates may receive more than two edges. In this case we say that they have *unbounded fanin*. Circuits in which only the $+$-gates may have unbounded fanin are called *semi-unbounded circuits*. Observe that in semi-unbounded circuits $\times$-gates still have fanin 2. A circuit is called *multiplicatively disjoint* if for each $\times$-gate v the subcircuits that have the children of v as output-gates are disjoint. A circuit is called *skew*, if for all of its $\times$-gates one of the children is an input gate. An *arithmetic formula* is an arithmetic circuit whose underlying graph is a tree.

The polynomial computed by an arithmetic circuit is defined in the obvious way: an input gate computes the value of its label, a

computation gate computes the product or the sum of its children's values, respectively. We assume that a circuit has only one sink which we call the output gate. We say that the polynomial computed by the circuit is the polynomial computed by the output gate. The *size* of an arithmetic circuit is the number of gates. The *depth* of a circuit is the length of the longest path from an input gate to the output gate in the circuit.

An arithmetic circuit is called *constant free*, if the only constants at input gates are -1, 0 and 1. Finally, a circuit or formula is called *monotone* if only the constants 0 and 1 are allowed. When we consider constant free arithmetic circuits we always assume them to compute over a field of characteristic 0, such that the circuits compute polynomials in $\mathbb{Z}[X_1, \ldots, X_n]$. When a constant free arithmetic circuit is the input of a problem, we assume that it is given as a graph with labels on the vertices, for instance as an adjacency list.

We call a sequence (f_n) of multivariate polynomials a family of polynomials or *polynomial family*. We say that a polynomial family is of polynomial degree, if there is a univariate polynomial p such that $\deg(f_n) \leq p(n)$ for each n. VP is defined as the class of polynomial families of polynomial degree computed by families of polynomial size arithmetic circuits. Throughout the thesis we will use the following characterizations of VP, sometimes without explicitly referencing this theorem.

Theorem 9.2.1. *([VSBR83, MP08]) Let (f_n) be a family of polynomials. The following statements are equivalent:*

1. $(f_n) \in \mathsf{VP}$
2. *(f_n) is computed by a family of multiplicatively disjoint polynomial size circuits.*
3. *(f_n) is computed by a family of semi-unbounded circuits of logarithmic depth and polynomial size.*
4. *(f_n) is computed by a family of semi-unbounded, multiplicatively disjoint circuits of logarithmic depth and polynomial size.*

The last item of Theorem 9.2.1 is not stated explicitly in the referenced papers, but it follows easily by applying the techniques of Malod and Portier [MP08, Lemma 2] on the characterization of VP by logarithmic depth semi-unbounded arithmetic circuits.

VP_e is defined as the class of polynomial families computed by arithmetic formulas of polynomial size. By a classical result of Brent [Bre76], VP_e can equivalently be defined as the class of polynomial families computed by arithmetic circuits of depth $O(\log(n))$. VP_{ws} is defined as the class of families of polynomials computed by families of skew circuits of polynomial size. Finally, a family (f_n) of polynomials is defined to be in VNP, if there is a family $(g_n) \in \mathsf{VP}$ and a

polynomial p such that $f_n(X) = \sum_{e\in\{0,1\}^{p(n)}} g_n(e, X)$ for all n where X denotes the vector $(X_1, \ldots, X_{q(n)})$ for some polynomial q.

The most prominent polynomial in arithmetic circuit complexity besides the determinant is certainly the *permanent*. For an $(n \times n)$-matrix $(X_{ij})_{i,j\in[n]}$ we define the permanent as

$$\mathrm{PER}_n := \sum_{\sigma\in S_n} \prod_{i=1}^{n} X_{i\sigma(i)}.$$

The permanent has roughly the same importance in arithmetic circuit complexity as SAT has in Boolean complexity because of the following theorem.

Theorem 9.2.2 ([Val79]). *The family (PER_n) is* VNP*-complete.*

The following criterion by Valiant [Val79] (see also [Büo0, Proposition 2.20]) for containment in VNP is often helpful:

Lemma 9.2.3 (Valiant's criterion). *Let $\phi : \{0,1\}^* \to \mathbb{N}$ be a function in* #P/poly*, Then the family (f_n) of polynomials defined by*

$$f_n = \sum_{e\in\{0,1\}^n} \phi(e) \prod_{i=1}^{n} X_i^{e_i}$$

is in VNP*.*

A polynomial f is called a *projection* of g (symbol: $f \leq g$), if there are values $a_i \in \mathbb{F} \cup \{X_1, X_2, \ldots\}$ such that $f(X) = g(a_1, \ldots, a_q)$. A family (f_n) of polynomials is called a *p-projection* of (g_n) (symbol: $(f_n) \leq_p (g_n)$), if there is a polynomial r such that $f_n \leq g_{r(n)}$ for all n. As usual we say that (g_n) is hard for an arithmetic circuit class $\mathcal{C}$ if for every $(f_n) \in \mathcal{C}$ we have $(f_n) \leq_p (g_n)$. If further $(g_n) \in \mathcal{C}$ we say that (g_n) is $\mathcal{C}$-complete.

When considering arithmetic circuits as the input of computational problems as we will do in Part iii, it is common to consider so-called *degree-bounded* arithmetic circuits, i.e., one assumes that there is a polynomial p such that for all polynomial f computed as circuits C that are inputs to the computational problem we have $\deg(f) \leq p(|C|)$. This kind of degree bound has two problems: One is that computing the degree of a polynomial represented by a circuit is suspected to be hard (see also Section 14.4 and [ABKPM09, KP07, KS11]), so problems defined with this degree bound must often be promise problems. The other problem is that the bound on the degree does not bound the size of the coefficients of the computed polynomial, which by iterative squaring can have exponential bitsize. Thus even evaluating circuits on a Turing machine in polynomial time becomes intractable. Problems resulting from this are discused in a paper by Allender et al. [ABKPM09].

To avoid all these complications, instead of bounding the degree of the computed polynomial, we will bound the *formal degree* of the circuit. The formal degree of a circuit C is defined inductively:

- The formal degree of an input gate v is 1.
- The formal degree of an addition gate v with with children u, w is the maximum of the formal degree of u and the formal degree of w.
- The formal degree of a multiplication gate v with children u, w is the sum of the the formal degree of u and the formal degree of w.

The formal degree of C is defined to be the formal degree of its output gate.

Clearly, the formal degree of a circuit C is an upper bound for the degree of the polynomial f computed by C. Also it is easy to see that the formal degree of C can be computed efficiently. Finally, a circuit C can be evaluated in time polynomial in $|C|$ and the formal degree. Thus we avoid the complications dicussed above for the degree.

Instead of considering circuits of bounded formal degree, it is often more convenient to consider multiplicatively disjoint circuits. This is justified by the following Lemma of Portier and Malod [MP08] which states that the two notions are essentially equivalent.

Lemma 9.2.4. *a) Let C be a constant free arithmetic circuit of formal degree d. Then there is a constant free arithmetic circuit of size at most $d|C|$ computing the same polynomial as C.*

b) Every multiplicatively disjoint circuit C has formal degree at most $|C|$.

We will several times consider parse trees of multiplicatively disjoint ciruits, which can be seen as objects tracking the formation of monomials during the computation [MP08, Section 4] and which are the algebraic analog of proof trees [VT89].

A *parse tree* T of a multiplicatively disjoint circuit C is a subgraph of C that is constructed in the following way:

- Add the output gate of C to T.
- For every gate v added to T do the following;
  - If v is a $+$-gate, add exactly one of its children u and the edge vu to T.
  - If v is a $\times$-gate with inputs u and w, add both u and w and the edges vu and vw to T.

Observe that parse trees are binary trees. The weight $w(T)$ of a parse tree T is defined as the product of the labels of its leaves. It is straightforward to check that the polynomial computed by C is the sum of the weights of all of C's parse trees.

We remark that parse trees can also be defined for arithmetic circuits that are not multiplicatively disjoint. In this case the definition

is a little more tricky, because the parse tree of a circuit C is in general not a subtree of C and can be of exponential size in $|C|$. Since we will only need parse trees of multiplicatively disjoint circuits, we refer the reader to [MP08] for more details.

9.3 DIGRESSION: REDUCTION NOTIONS IN ARITHMETIC CIRCUIT COMPLEXITY

The usual reduction notion in arithmetic circuit complexity is that of p-projections introduced above. There is also a different notion of reductions more similar to Turing- or oracle-reductions that was defined in [Büo00] and is used in some recent papers (see e.g. [BK09, dRA12]).

Definition 9.3.1. *The oracle complexity $L^{(g)}(f)$ of a polynomial f with oracle g is the minimum number of arithmetic operations $+, -, \times$ and evaluations of g (at previously computed values) that are sufficient to compute f from the variables $X_1, X_2, \ldots$ and constants in $\mathbb{F}$.* □

Definition 9.3.2. *Let (f_n) and (g_n) be families of polynomials. We call (f_n) a c-*reduction *of (g_n), symbol $(f_n) \leq_c (g_n)$, if and only if there is a polynomially bounded function $t : \mathbb{N} \to \mathbb{N}$ such that the map $n \mapsto L^{g_{t(n)}}(f_n)$ is polynomially bounded.* □

Intuitively, if (f_n) c-reduces to (g_n), then there is a family of arithmetic circuit of polynomial size that may use polynomials from (g_n) to compute (f_n).

It appears as though c-reductions should be more powerful than p-projections. The main observation of this section is that we can actually prove this unconditionally over the field $\mathbb{R}$. This is in contrast to the Boolean setting where results of this kind have only been proven under some complexity assumptions (see e.g. [HP06]).

We consider a polynomial that we model after the permanent polynomial PER_n:

$$P_n := \sum_{\sigma \in S_n} \prod_{i=1}^{n} X_{i,\sigma(i)} + \sum_{\sigma} \prod_{i=1}^{n} X_{i,\sigma(i)}^2,$$

where the sum is over all permutations.

We remark that the same construction would also work for an arbitrary homogeneous polynomial known to be VNP-complete.

Lemma 9.3.3. *(P_n) is* VNP*-complete under c-reductions over $\mathbb{R}$.*

Proof. This is a simple interpolation argument using that the homogeneous part of degree n of P is obviously the permanent. We interpolate $P(\lambda X)$ at the arguments 1 and 2 and get that

$$\mathrm{PER}_n(X) = \frac{2^n}{2^n - 1} P_n(X) - \frac{1}{2^n(2^n - 1)} P_n(2X).$$

It follows that P_n is even VNP-complete under linear p-projections (see [Büoo, p. 54]). ■

Remark 9.3.4. This construction works for every field with at least 2 elements, i.e., for every field except $GF(2)$. We can interpolate at arguments 1 and a with $a \neq 1$ and $a \neq 0$. The only problem in this case is that a^n might be 1 so that the interpolation will not work. We can work around this by simply evaluating P_{n+1} in this case. □

Lemma 9.3.5. *(P_n) is not* VNP*-complete under p-projections over* $\mathbb{R}$.

The short proof of Lemma 9.3.5 emerged from discussions with Dennis Amelunxen and Christian Ikenmeyer.

Proof. We show that the simple polynomial X is not a projection of P_n for any n (X is just a single variable here). Assume this were not the case. Then there is a $(n \times n)$-matrix $A = (a_{ij})_{i,j\in[n]}$ with entries from $\{X\} \cup \mathbb{R}$ such that $P_n(A) = X$. Let σ be a permutation such that $\prod_{i=1}^n a_{i\sigma(i)}$ has maximal degree. Obviously this degree is at least 1. Then the monomial $\prod_{i=1}^n a_{i\sigma(i)}^2$ has at least degree 2 and it cannot cancel out because

- it cannot cancel with any $\prod_{i=1}^n a_{i\mu(i)}$ for a permutation μ, because those all have smaller degrees, and
- it cannot cancel out with any $\prod_{i=1}^n a_{i\mu(i)}^2$, because those all have positive coefficients in $P_n(A)$.

Thus $P_n(A)$ has degree at least 2, which implies that it cannot compute X. ■

As a corollary we get that over $\mathbb{R}$, c-reductions yield strictly more complete problems that p-projections.

Theorem 9.3.6. *There is a family of polynomials that is* VNP*-complete over* $\mathbb{R}$ *under c-reductions but not under p-projections.*

Lemma 9.3.5 unfortunately does not generalize to arbitrary fields.

Lemma 9.3.7. *Let F be a field such that there are elements $a_1, \ldots, a_s$ with $\sum_{i=1}^s a_i \neq 0$ and $\sum_{i=1}^s a_i^2 = 0$. Then (P_n) is* VNP*-complete over F under p-projections.*

Proof. For an $(n \times n)$-matrix A let $\mathrm{PER}(A)$ be the permanent of A and set $\mathrm{PER}(A^2) := \sum_\sigma \prod_{i=1}^n a_{i\sigma(i)}^2$. With this notation clearly $P_n(A) = \mathrm{PER}(A) + \mathrm{PER}(A^2)$. For block matrices we have the following decomposition formula

$$P_{s+t}\begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix} = \mathrm{PER}(A)\mathrm{PER}(B) + \mathrm{PER}(A^2)\mathrm{PER}(B^2), \tag{5}$$

where A is an $(s \times s)$-matrix and B is a $(t \times t)$-matrix.

Let $a := \sum_{i=1}^{s} a_i$ and

$$A := \begin{pmatrix} a_1 & a_2 & a_3 & \dots & a_s & 0 \\ 1 & 0 & 0 & \dots & 0 & a^{-1} \\ 0 & 1 & 0 & \dots & 0 & a^{-1} \\ 0 & 0 & 1 & \dots & 0 & a^{-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & a^{-1} \end{pmatrix}$$

It is easy to verify that $\mathrm{PER}(A) = \sum_{i=1}^{s} a_i a^{-1} = 1$ and $\mathrm{PER}(A^2) = \sum_{i=1}^{s} a_i^2 (a^{-1})^2 = \left(\sum_{i=1}^{s} a_i^2\right) a^{-2} = 0$.

Thus we get with (5)

$$P_{s+t+1} \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix} = \mathrm{PER}(A)\mathrm{PER}(B) + \mathrm{PER}(A^2)\mathrm{PER}(B^2) = \mathrm{PER}(B)$$

for every $(t \times t)$-matrix B.

Thus the permanent family is a p-projection of (P_n) and with the VNP-completeness of the permanent under p-projections the claim follows. ■

Corollary 9.3.8. *a)* (P_n) *is* VNP*-complete under p-projections over* $\mathbb{C}$.

b) (P_n) *is* VNP*-complete under p-projections over any field of characteristic greater than* 2.

Proof. a) Set $a_1 = 1$ and $a_2 = i$. We have $a_1 + a_2 = 1 + i$ and $a_1^2 + a_2^2 = 0$ and thus the claim follows by Lemma 9.3.7.

b) Let $p > 2$ be the characteristic of the field. We have

$$\sum_{i=1}^{\frac{p-1}{2}} 1 + \sum_{i=1}^{\frac{p+1}{2}} (-1) = -1 \neq 0$$

and

$$\sum_{i=1}^{\frac{p-1}{2}} 1^2 + \sum_{i=1}^{\frac{p+1}{2}} (-1)^2 = p \cdot 1 = 0.$$

With Lemma 9.3.7 the claim follows. ■

INTERPRETATION We have shown that there are fields in which c-reductions and p-projections differ. This result is a little irritating because it depends crucially on the field of real numbers. It is not clear what happens over other fields, especially in the light of Corollary 9.3.8, so we close this digression with a question:

Question 1. *For which fields* $\mathbb{F}$ *is there a family of polynomials that is* VNP*-complete over* $\mathbb{F}$ *under c-reductions but not under p-projections?*

10

CONSTRAINT SATISFACTION PROBLEMS, CONJUNCTIVE QUERIES AND ARITHMETIC CIRCUIT CLASSES

In this chapter we will connect the considerations on conjunctive queries and constraint satisfaction problems of Part i to the arithmetic circuit setting. While conjunctive queries so far have not been considered in the arithmetic circuit literature, constraint satisfaction has been considered before by Briquel and Koiran [BK09]: They gave a dichotomy result for arithmetic circuits similar to those of Schaefer [Sch78] for decision and Hermann and Creignou [CH96] for counting. To a family (Φ_n) of Boolean CSP-instances they assign a polynomial family $(P(\Phi_n))$ and show that there is a small set S of Boolean relations with the following property: If a family (Φ_n) of CSP-instances is constructed of relations in S only, then $(P(\Phi_n)) \in \mathsf{VP}$. On the other hand if instances may be constructed with any Boolean relation not in S, one can construct a family of CSP-instances (Φ_n) such that $(P(\Phi_n))$ is VNP-complete.

Because constraint satisfaction and conjunctive queries are important for practical purposes, researchers in database theory and AI have tried to pinpoint the exact complexity of both problems when restricted to the classes of Part i (see Section 2). The key class of problems is ACQ, i.e., the class of CQ-instances that have acyclic associated hypergraphs, because more general instances can be reduced to them (see Lemma 2.3.28). It was shown that ACQ can be solved in parallel, i.e., in the NC-hierarchy, but the exact complexity of the problem was open for some time. Gottlob et al. [GLS01] solved this question by proving that ACQ is complete for the class LOGCFL, the Boolean sibling of VP. This result also extends to CSP-instances of bounded width for many of the width measures considered in Part i.

During the last years treewidth has found its way into arithmetic circuit complexity. This was started by Courcelle et al. [CMR01] who showed that generating functions of graph problems expressible in monadic second order logic have small arithmetic circuits for graphs of bounded treewidth. This line of research was continued by Flarup et al. [FKL07] who improved these upper bounds and showed matching lower bounds for some families of polynomials (see also Chapter 11). Briquel, Koiran and Meer [BKM11]—building on a paper by Fischer et al. [FMR08] which deals with counting problems—considered polynomials defined by CNF-formulas of bounded treewidth. The more general width measures of Part i have so far not appeared in arithmetic circuit complexity.

In this chapter we unify these different lines of work sketched above: We complement the general intractability results of Briquel and Koiran [BK09] by identifying tractable subclasses of polynomials assigned to CQ-instances. In this respect the results in this paper correspond to the results of Gottlob et al. [GLS01] for CQ and the results presented in Part i. Also, this chapter can be seen as an extension of the work of Briquel, Koiran and Meer [KM08, BKM11] by considering CQ-instances instead of CNF-formulas. We consider two kinds of polynomials for CQ-instances and show that they characterize the hierarchy $\mathsf{VP}_e \subseteq \mathsf{VP}_{ws} \subseteq \mathsf{VP} \subseteq \mathsf{VNP}$ of arithmetic circuit classes commonly considered, respectively, for different classes of CQ-instances. CQ-instances of bounded relation size, i.e., with relations whose size is bounded by a constant, capture VP_e when restricted by the width measures of Part i, while in the case of unbounded relation size we get VP_{ws} for bounded pathwidth and VP for the width measures of Part i, e.g. generalized hypertree width, when combining them with bounded quantified star size.

It will be convenient to prove most of our results in this chapter in the quantifier free setting, i.e., for CSP-instances instead of the more general CQ-instances. The generalization to CQ-instances will follow by Lemma 5.2.1.

10.1 POLYNOMIALS DEFINED BY CONJUNCTIVE QUERIES

We call a CQ-instance Φ Boolean if it has domain $\{0,1\}$. To avoid confusion, we warn the reader that word "Boolean" is used in two different ways in the context of conjunctive queries: On the one hand, the decision problem CQ is called the Boolean conjunctive query problem. On the other hand, CQ-instances are called Boolean if they have domain $\{0,1\}$. Fortunately, we will not consider the Boolean conjunctive query problem CQ in this chapter at all, so the word "Boolean" stands for the domain $\{0,1\}$ in this chapter.

To a CQ-instance $\Phi = (\mathcal{A}, \phi)$ we will assign two polynomials $P(\Phi)$ and $Q(\Phi)$. However, $P(\Phi)$ is only defined for Boolean instances, i.e., for such with domain $\{0,1\}$. So let Φ first be a Boolean CQ-instance with the free variables $\mathsf{free}(\Phi) = \{x_1, \dots, x_n\}$. We assign to Φ a polynomial $P(\Phi)$ in the (position) variables $Y_1, \dots, Y_n$ in the following way:

$$P(\Phi) := \sum_{e \in \phi(\mathcal{A})} Y^e.$$

Here Y^e stands for $Y_1^{e(x_1)} Y_2^{e(x_2)} \dots Y_n^{e(x_n)}$.

Example 10.1.1. Let the atoms in Φ be defined by the propositional formulas $\{x_1 \vee x_2, x_3 \neq x_2, \neg x_4 \vee x_2\}$. The satisfying assignments are then 0100, 0101, 1010, 1100 and 1101. This results in $P(\Phi) = X_2 + X_2X_4 + X_1X_3 + X_1X_2 + X_1X_2X_4$. □

In contrast to $P(\Phi)$ the second polynomial $Q(\Phi)$ is also defined for non-boolean CQ-instances. So let $\Phi = (\mathcal{A}, \phi)$ be a CQ-instance with domain A. We assign to Φ the following polynomial $Q(\Phi)$ in the variables $\{X_d \mid d \in A\}$

$$Q(\Phi) := \sum_{a \in \phi(\mathcal{A})} \prod_{x \in \mathsf{free}(\Phi)} X_{a(x)} = \sum_{a \in \phi(\mathcal{A})} \prod_{d \in A} X_d^{\mu_d(a)},$$

where $\mu_d(a) = |\{x \in \mathsf{free}(\Phi) \mid a(x) = d\}|$ is the number of variables mapped to d by a. Note that the number of variables in $Q(\Phi)$ is $|A|$, the size of the domain, and that $Q(\Phi)$ is homogeneous of degree $|\mathsf{free}(\Phi)|$.

Example 10.1.2. Let $A = \{1,2,3,4\}$ and let the atoms in Φ have the relations$\{x_1 + x_2 \geq 4, x_3 = 5 - x_2, x_1 < x_2\}$. The satisfying assignments are then $(1,3,2)$, $(2,3,2)$, $(1,4,1)$, $(2,4,1)$ and $(3,4,1)$. This results in $Q(\Phi) = X_1X_2X_3 + X_2^2X_3 + X_1^2X_4 + X_1X_2X_4 + X_1X_3X_4$. □

Remark 10.1.3. The polynomial Q has a very natural algebraic interpretation: Consider the free monoid A^* of words over A. Furthermore consider the free commutative monoid X_A^c over $X_A := \{X_d \mid d \in A\}$, which is essentially the set of monomials in the variables in X_A. There is a natural monoid morphism $q : A^* \to X_A^c$ with $q(a_1 \dots a_s) = \prod_{i=1}^s X_{a_i}$. The morphism q ignores the order of the symbols in a word and thus computes a commutative version of it.

Now we consider two rings: The first one is $\mathbb{Z}[A^*]$ consisting of formal integer linear combinations of words in A^*. Observe that we can think of any finite set $S \subseteq A^*$ as an element of $\mathbb{Z}[A^*]$ by encoding it as $\sum_{a \in S} a$. The second ring we consider is $\mathbb{Z}[X_A]$ which is simply the polynomial ring over $\mathbb{Z}$ in the variables X_A. The monoid morphism q induces the ring morphism $Q : \mathbb{Z}[A^*] \to \mathbb{Z}[X_A]$ by $Q(\sum_a c_a a) = \sum_a c_a q(a)$. Given the encoding $\sum_{a \in S} a$ of a set S, Q computes a commutative version of it. This is exactly what the polynomial $Q(\Phi)$ defined above does: To a CQ-instance Φ it computes a commutative version of the query result of Φ. □

Remark 10.1.4. If Φ is a Boolean CQ-instance, i.e., it has domain $A = \{0,1\}$, we can get $Q(\Phi)$ from $P(\Phi)$ easily. $Q(\Phi)$ has only two variables X_0 and X_1 and it is homogeneous of degree $|\mathsf{free}(\Phi)|$. Substituting Y_i of $P(\Phi)$ by $\frac{X_1}{X_0}$ we get

$$X_0^{|\mathsf{free}(\Phi)|} P(\Phi)\left(\frac{X_1}{X_0}, \dots, \frac{X_1}{X_0}\right) = Q(\Phi)(X_1, X_0).$$

□

In the following two lemmas we will see that the two polynomials we defined for CQ-instances are even closer related than observed in Remark 10.1.4. Recall the definition of the S-hypergraph associated to a query ϕ from Definition 3.2.3.

Lemma 10.1.5. *For every Boolean* CQ*-instance* $\Phi = (\mathcal{A}, \phi)$ *there is a* CQ*-instance* $\Psi = (\mathcal{B}, \psi)$ *of size polynomial in* $\|\Phi\|$ *such that* $P(\Phi) \leq Q(\Psi)$ *and* ϕ *and* ψ *have the same* S*-hypergraph. Furthermore, the size of the biggest relation in* $\mathcal{A}$ *and* $\mathcal{B}$ *coincide.*

Proof. We show this for quantifier free instances. The extension to general CQ-instances will be clear.

Let $\mathsf{var}(\phi) = \{x_1 \dots, x_n\}$. We assume w.l.o.g. that each relation symbol in ϕ appears only once. This can always be achieved by renaming relation symbols of atoms and copying relations. This only increases the size of Φ polynomially and does not change the S-hypergraph. We set $\psi := \phi$ and construct $\mathcal{B}$ with the domain B that contains two elements $(i, 0)$ and $(i, 1)$ for each $i \in [n]$.

For each atom $\mathcal{R}(x_{i_1}, \dots, x_{i_k})$ of ϕ and ψ we construct the relation $\mathcal{R}^{\mathcal{B}}$ as

$$\mathcal{R}^{\mathcal{B}} := \{((i_1, t_1), \dots, (i_k, t_k)) \mid (t_1, \dots, t_k) \in \mathcal{R}^{\mathcal{A}}\}.$$

Observe that this construction is well-defined because each relation symbol only appears in one atom. This completes the construction of Ψ.

Let H_Φ be the set of assignments $a : \mathsf{var}(\phi) \to \{0, 1\}$ and H_Ψ the set of assignments $a' : \mathsf{var}(\psi) \to B$. Then the mapping $m : H_\Phi \to H_\Psi$ with $m(x_i \mapsto b_i) = (x_i \mapsto (i, b_i))$ is an isomorphisms between $\phi(\mathcal{A})$ and $\psi(\mathcal{B})$. Furthermore, each assignment $a \in \phi(\mathcal{A})$ yields a monomial $\prod_{i=1}^{s} X_i^{a(i)}$ in $P(\Phi)$ while $m(a)$ yields the monomial $\prod_{i=1}^{s} X_{i,a(i)}$ in $Q(\Psi)$. Thus substituting for each $i \in [s]$ the variable $X_{(i,0)}$ by 1 and $X_{(i,1)}$ by X_i gives $P(\Phi) \leq Q(\Psi)$.

The rest of the claims is clear by construction. ■

Next we formulate a lemma that can be seen as a partial converse of Lemma 10.1.5: Q-polynomials can always be expressed as P-polynomials. Unfortunately, the construction leads to an increase of the arity of the atoms.

Let $\mathcal{H} = (V, E)$ be a hypergraph. A *blow-up hypergraph* $\mathcal{H}' = (V', E')$ of $\mathcal{H}$ is a hypergraph that has for each $v \in V$ a set V_v such $V' = \dot{\bigcup}_{v \in V} V_v$ and $E' = \{\bigcup_{v \in e} V_v \mid e \in E\}$. If for every $v \in V$ we have $|V_v| \leq \ell$ then we call $\mathcal{H}'$ an ℓ-bounded blow-up graph.

Lemma 10.1.6. *For every* CQ*-instance* Φ *one can construct in polynomial time a Boolean* CQ*-instance* Ψ *with* $Q(\Phi) \leq P(\Psi)$*. The sizes of the biggest relations of* Φ *and* Ψ *coincide. Furthermore, the associated hypergraph of* Ψ *is an* ℓ*-bounded blow-up hypergraph of the associated hypergraph of* Φ *where* ℓ *is the size of the biggest relation in* Φ*. Moreover, the quantified star sizes of* Φ *and* Ψ *coincide.*

Proof. Let $\Phi = (\mathcal{A}, \phi)$. We again assume that each relation symbol appears only once in ϕ. Moreover, we assume that Φ satisfies the following consistency condition: Let x be a variable and ϕ' an atom in

which x appears. Let $a \in \phi'(\mathcal{A})$, then for every atom ϕ'' in which x appears there is a $a' \in \phi''(\mathcal{A})$ such that $a'(x) = a(x)$. Clearly, every instance Φ can be transformed into an instance that satisfies this consistency condition by iteratively deleting tuples from the relations of $\mathcal{A}$. Moreover, this procedure does not change the query result, so we can assume that Φ satisfies this consistency condition. It follows that there is a set $D_x \subseteq A$ for every variable x such that in every satisfying assignment $a \in \phi(\mathcal{A})$ we have $a(x) \in D_x$ and, since all relations have size at most ℓ, we have $d(x) := |D_x| \leq \ell$.

Let $D_x := \{s_1, \dots, s_{d(x)}\} \subseteq A$. We encode $s \in D_x$ by a $\{0,1\}$-string $e_x(s)$ of length $d(x)$ by setting $e_x(s)_i := 1$ if $s = s_i$ and $e_x(s)_i := 0$ otherwise. This encoding induces an encoding of tuples and relations which we call $e(t)$ for a tuple t and $e(\mathcal{R}^{\mathcal{A}})$ for a relation $\mathcal{R}^{\mathcal{A}}$, respectively.

We now construct $\Psi = (\mathcal{B}, \psi)$. For every atom $\mathcal{R}(x_{i_1}, \dots, x_{i_k})$ of ϕ the query ψ has an atom $\mathcal{R}'(x_{i_1,1}, \dots, x_{i_1,d(x_{i_1})}, \dots, x_{i_k,1}, \dots, x_{i_k,d(x_{i_k})})$. The associated relation is $\mathcal{R}'^{\mathcal{B}} := e(\mathcal{R}^{\mathcal{B}})$. Finally, $x_{i,j} \in \mathsf{var}(\psi)$ is quantified in ψ if and only if $x_i \in \mathsf{var}(\phi)$ is quantified. This completes the construction of Ψ.

By construction, $a \in \phi(\mathcal{A})$ if and only if $e(a) \in \phi(\mathcal{B})$. Thus we get $Q(\Phi) \leq P(\Psi)$ by substituting all $X_{i,j}$ by X_j. The other desired properties of Ψ are easily checked. ■

Since we want to compute families of polynomials we also consider families of CQ-instances in this chapter.

Definition 10.1.7. *We call a family (Φ_n) of* CQ*-instances* p-bounded *if the size $\|\Phi_n\|$ is polynomially bounded. We call (Φ_n)* relation bounded *if there is a constant ℓ such that all relations of all instances Φ_n have size at most ℓ.* □

Example 10.1.8. We consider a straightforward encoding of 3-CNF-formulas into CQ-instances: Each clause C with three variables is encoded by an atom ϕ_c in these variables. The associated relation contains the 7 assignments that make C true.

It follows that every family (ψ_n) of 3-CNF-formulas of polynomial size can be encoded by a family (Φ_n) of CQ-instances. Since every relation of every Φ_n contains exactly 7 tuples, the family (Φ_n) is p-bounded and relation bounded. □

Example 10.1.9. Of course, one can also encode general CNF-formulas into CQ-instances as in Example 10.1.8. Observe though that a clause with n variables has $2^n - 1$ satisfying assignments and thus the size of the relations encoding the clauses grows exponentially in the arity of the clauses. It follows that families of CNF-formulas in general cannot be encoded by p-bounded families of CQ-instances.□

10.2 MAIN RESULTS

In this section we give a short overview over the characterizations of different arithmetic circuit classes by p-bounded families of CQ-instances that we prove in this chapter.

In Section 10.3 we show that all #P-complete cases of #CQ discussed in Part i yield characterizations of VNP:

- p-bounded families of CSP-instances of unrestricted structure,
- p-bounded families of ACQ-instances without restriction of the quantified star size, and
- p-bounded families of unions or intersections of quantifier free ACQ-instances

all characterize VNP.

It follows that families of CQ-instances for which we can hope to get tractable polynomials should have restricted hypergraphs, e.g. bounded generalized hypertree width, and bounded quantified star size. We will see that these restrictions indeed yield tractable polynomials and that all families in VP can be expressed that way.

Theorem 10.2.1. *For every p-bounded family (Φ_n) of CQ-instances with bounded generalized hypertree width and bounded quantified star size, the family $(Q(\Phi_n))$ is in VP. Moreover, every family in VP is a p-projection of such $(Q(\Phi_n))$.*

It is not hard to see that one gets another characterization of VP by choosing any other width measure from Part i, except pathwidth.

The exception of pathwidth is because the expressivity of the CQ-instances drops if the decompositions are too path-like.

Theorem 10.2.2. *For every p-bounded family (Φ_n) of CQ-instances with bounded pathwidth and bounded quantified star size, the family $(Q(\Phi_n))$ is in VP_{ws}. Moreover, every family in VP_{ws} is a p-projection of such $(Q(\Phi_n))$.*

Finally, considering relation bounded families gives very robust characterizations of VP_e.

Theorem 10.2.3. *Let (Φ_n) be a relation bounded and p-bounded family of CQ-instances of bounded generalized hypertree width. Then $(Q(\Phi)) \in \mathsf{VP}_e$. Moreover, every family in VP_e is a p-projection of such $(Q(\Phi_n))$, where (Φ_n) may even be assumed to be of bounded pathwidth and quantifier free.*

Note that in the case of relation bounded instances existential quantification does not increase the complexity of the computed polynomials. This is because quantified variables in CQ-instances can be handled by dynamic programming which yields circuits of size exponential in the size of the relations. Since the size of the relations is

assumed to be bounded by a constant, this exponential size is not a problem in this setting. Furthermore, note that there is no difference in expressivity between patwidth and more general width measures: All considered width measures characterize VP_e.

We remark that while relation bounded instances appear quite arbitrary at first sight, they are actually considered very often in the database and constraint satisfaction literature. In the database literature they appear when one considers the so-called *query complexity* (sometimes also called expression complexity) of query problems. Here the database (or in our wording the structure) is considered as fixed and one asks what the complexity of queries against this fixed database is. Thus only the size of the query counts as input size of the computational problem while the size of the database is considered as constant.

In the constraint satisfaction literature one often considers instances built with a fixed set of relations called the constraint language. Here one often tries to find dichotomy results for the complexity depending on the constraint language.

Pichler and Skritek [PS13] have shown that #ACQ is tractable in the sense of query complexity. Our results can be seen as a version of this result for the arithmetic circuit setting.

We remark that for nearly all of the theorems presented above one can also prove versions for the P-polynomial with the help of Lemma 10.1.5 and Lemma 10.1.6. We leave the details to the reader.

10.3 CHARACTERIZATIONS OF VNP

To start off our explorations of the expressivity of polynomials defined by CQ-instances we first examine the hard cases, i.e., those that are at least as hard as VNP.

10.3.1 *Instances of unrestricted structure*

We first show that that if we do not restrict the structure of quantifier free instances, this gives us a characterization of VNP in both the Boolean and non-Boolean case. Both the upper and lower bound are more or less standard arguments. Remember that we call a CSP-instance binary if all its atoms have arity at most 2. Recall also that a Boolean CSP-instance is an instance with domain $\{0,1\}$.

Lemma 10.3.1. *a) Let (Φ_n) be a p-bounded family of Boolean* CSP*-instances. Then the family $(P(\Phi_n))$ of polynomials is in* VNP*. Moreover, every family in* VNP *is a p-projection of such $(P(\Phi_n))$ where Φ_n can even be assumed to be binary.*

b) Let (Φ_n) be a p-bounded family of CSP*-instances. Then $(Q(\Phi_n)) \in$ VNP. Moreover, every family in* VNP *is a p-projection of such a family $(Q(\Phi_n))$ where Φ_n can even be assumed to be binary.*

Proof. Note that with Lemma 10.1.5 we only need to show the lower bound for a) and the upper bound for b). The lower bound for a) is proved in [BK09, Section 3], so it only remains to prove that $Q(\Phi_n) \in$ VNP for a p-bounded family (Φ_n) of CSP-instances.

So let (Φ_n) be a p-bounded family of CSP-instances where $\Phi_n = (\mathcal{A}_n, \phi_n)$. Let $d_n := |A_n|$ be the size of the domain of Φ_n and set $a_n := |\mathsf{var}(\Phi_n)|$. We encode assignments $a\colon \mathsf{var}(\phi_n) \to A_n$ by $(d_n \times a_n)$-matrices $M = (m_{d,x})_{d\in A_n, x\in\mathsf{var}(\Phi_n)}$ with entries 0 and 1. The entry $m_{d,x}$ is 1 if and only if $a(x) = d$. Note that a matrix $M \in \{0,1\}^{d_n\times a_n}$ is an encoding of an assignment $a\colon \mathsf{var}(\phi_n) \to A_n$ if and only if there is exactly one 1 in each column of M.

For Φ_n we will construct an arithmetic formula Ψ_n of size polynomial in $\|\Phi_n\|$ such that $Q(\Phi)(X) = \sum_{M\in\{0,1\}^{d_n\times a_n}} \Psi(M, X)$. We subdivide Ψ_n into three factors $\Psi_{n,1}$, $\Psi_{n,2}$ and $\Psi_{n,3}$.

We set

$$\Psi_{n,1} = \prod_{x\in\mathsf{var}(\Phi_n)} \sum_{d\in A_n} m_{d_x} \prod_{d'\in A_n, d'\neq d} (1 - m_{d'x}).$$

It is easy to see that for $M \in \{0,1\}^{d_n\times a_n}$ we have $\Psi_{n,1}(M) \in \{0,1\}$ and $\Psi_{n,1}(M) = 1$ if and only if M is an encoding of an assignment $a : \mathsf{var}(\Phi_n) \to A_n$, i.e., in every column there is exacly one 1-entry.

We set

$$\Psi_{n,2} = \prod_{\phi \text{ atom of } \phi_n} \psi_\phi,$$

where ψ_ϕ is the following: Assuming that the matrix M encodes an assignment $a : \mathsf{var}(\phi_n) \to A_n$ we have $\psi_\phi(M) = 1$ if $a|_{\mathsf{var}(\phi)}$ satisfies ϕ, otherwise $\psi_\phi = 0$. The relation of ϕ is bounded in size by $\|\Phi_n\|$, so there are at most $\|\Phi_n\|$ satisfying assignments of ϕ. Each of these can be checked individually by an arithmetic formula of size $O(\mathsf{arity}(\phi)) = O(\Phi_n)$, so ψ_ϕ can be realized as a formula of size $O(\|\Phi\|^2)$. Since the number of atoms ϕ is bounded by $\|\Phi_n\|$ if follows $\Psi_{n,2}$ has a formula of polynomial size.

Finally,

$$\Psi_{n,3} = \prod_{x\in\mathsf{var}(\Phi)} (\sum_{d\in D} m_{d,x} X_d).$$

It is clear that indeed $\Psi_n = \Psi_{n,1}\Psi_{n,2}\Psi_{n,3}$ can be computed by a polynomial size formula. Also we have $Q(\Phi)(X) = \sum_{M\in\{0,1\}^{d_n\times a_n}} \Psi(M, X)$ and thus $(Q(\Phi_n)) \in$ VNP. ■

10.3.2 *Acyclic instances with quantification*

In this section we show a version of Theorem 3.1.3—Pichler and Skritek's hardness result for #ACQ—for the arithmetic circuit model.

We will see later in Section 10.5 that, for quantifier free ACQ-instances (Φ_n), computing $(Q(\Phi_n))$ is in VP. So again it is quantification that makes the considered problems hard.

To show VNP-hardness for polynomials defined by ACQ-instances it is convenient to show hardness of a clique-polynomial. This allows to keep the reduction very parallel to that in the proof of Lemma 3.1.4.

So let G be a graph. We give a vertex weight X_v to each vertex v of G and consider the following clique-polynomial

$$CP_{G,c} := \sum_C \prod_{v \in C} X_v,$$

where the sum is over the vertex sets C of all cliques of size c in G.

Lemma 10.3.2. *There is a family G_n of graphs of polynomial size such that $(CP_{G_n,n+1})$ is* VNP*-complete.*

The proof is a minor modification of the proof of Theorem 3.10 of [Büoo]. We give it here for the sake of completeness.

Proof. We reduce from the permanent PER_n. Let $[n]^2 \cup (0,0)$ be the vertex set of G_n. We construct G_n as follows: For each $(i,j) \in [n]^2$ the graph G contains the edge $(0,0)(i,j)$. For each pair $(i,j),(k,\ell) \in [n]^2$ the edge $(i,j)(k,\ell)$ is in G_n if and only if $i \neq k$ and $j \neq \ell$.

We now show that $\mathrm{PER}_n \leq CP_{G_n,n+1}$. To do so we give the vertex $(0,0)$ the weight 1. Each other vertex (i,j) gets weight X_{ij}.

Let C be the vertex set of a clique of size $n+1$ in G_n. By construction of G_n, each pair $(i,j),(k,\ell) \in C$ with $i,j,k,l > 0$ must differ in the first and in the second coordinate. Thus, $(0,0)$ must be part of any such clique C and furthermore each $i \in [n]$ must appear exactly once in the first and the second coordinate of the vertices in C, respectively. This gives a bijection between perfect matchings in $K_{n,n}$ and the cliques of size $n+1$ in G_n. Furthermore, this reduction obviously preserves the weights and thus $\mathrm{PER}_n \leq CP_{G_n,n+1}$. ∎

We now formulate a version of Lemma 3.1.4 in the arithmetic circuit setting. Remember that $\phi_{\mathrm{star},n} = \exists z \bigwedge_{i \in [n]} \mathcal{R}_i(z, y_i)$.

Proposition 10.3.3. *If (Φ_n) is a family of p-bounded* ACQ*-instances, then $(Q(\Phi_n)) \in$* VNP*. Moreover, there is a family (Φ_n) of p-bounded* ACQ*-instances with $\Phi_n = (\mathcal{A}_n, \phi_{star,n})$ such that $(Q(\Phi_n))$ is* VNP*-complete.*

Proof. For the first part of the statement we will use Valiant's criterion. To do so we show that the coefficient function of $Q(\Phi_n)$ is in $\#\mathsf{P}/\mathsf{poly}$. Consider an instance $\Phi = (\mathcal{A}, \phi)$. Let the number of free variables of ϕ be m. Remember that $Q(\Phi)$ is homogeneous of degree m. Consider a monomial $X_{a_1} \dots X_{a_m}$, $a \in A$ with possibly several occurences of the same variable. Let $a := (a_1, \dots, a_m)$ and let M_a be the set of tuples of length m in which each entry a_i appears as often as in a. Then the

coefficient of $X_{a_1} \dots X_{a_m}$ in $Q(\Phi)$ equals the number of $b \in M_a$ such that $b \in \phi(\mathcal{A})$.

Let now T_Φ be a nondeterministic Turing-machine that does the following: On input $(a_1, \dots, a_m)$ it nondeterministically guesses a a tuple $b \in M_a$ and accepts if and only if b satisfies Φ. By Theorem 2.3.19 the machine T_Φ runs in polynomial time. Furthermore, the number of accepting computations of T_Φ is the coefficient of $X_{a_1} \dots X_{a_m}$ in $Q(\Phi)$. It follows that for every family Φ_n of ACQ-instances the coefficient function of $Q(\Phi_n)$ is in #P/poly (the polynomial size advice is the encoding of Φ_n). With Theorem 9.2.3 we get that $Q(\Phi_n) \in \mathsf{VNP}$.

For the second part of the claim it remains to show hardness. To this end, we will show that for every family (G_n) of graphs of polynomial size the family $(CP_{G_n,n+1})$ is a p-projection of a p-bounded family (Φ_n) of ACQ-instances with $\Phi_n = (\mathcal{A}_n, \phi_{\text{star},n})$. With Lemma 10.3.2 the claim will follow.

So let (G_n) be a family of polynomial size graphs. Fix n and set $G := G_{n-1} = (V, E)$. We will reduce $CP_{G,n}$ to a CQ-instance Φ with the query $\phi_{\text{star},n+1}$. Let $\Psi := (\mathcal{A}, \phi_{\text{star},n})$ be the CQ-instance that we constructed in the proof of Lemma 3.1.4. Remember that the relations $\mathcal{R}_i^{\mathcal{A}}$ of $\mathcal{A}$ were constructed in such a way that an assignment $a : \{y_1, \dots, y_n\} \to A$ satisfies Ψ if and only if the set $\{a(y_1), \dots, a(y_n)\} \subseteq V$ does *not* induce a clique of size n in G.

We extend $\mathcal{A}$ to a structure $\mathcal{B}$ of $\phi_{\text{star},n+1} = \exists z \bigwedge_{i \in [n+1]} \mathcal{R}_i(z, y_i)$. For the new relation symbol $\mathcal{R}_{n+1}$ we define

$$\mathcal{R}_{n+1}^{\mathcal{B}} := \{(d,d)\} \cup \{(a,c) \mid a \in A\},$$

where c and d are new domain elements. Furthermore, for each $i \in [n]$ we set

$$\mathcal{R}_i^{\mathcal{B}} := \mathcal{R}_i^{\mathcal{A}} \cup \{(d,a) \mid a \in A\}.$$

We set $\Phi := (\mathcal{B}, \phi_{\text{star},n+1})$.

Depending on the value that is assigned to y_{n+1}, there are two types of assignments $a : \{y_1, \dots, y_n\} \to A \cup \{c,d\}$ that satisfy the query

$$\phi_{\text{star},n+1} = \exists z \bigwedge_{i \in [n+1]} \mathcal{R}_i(z, y_i)$$

with respect to $\mathcal{B}$: If $a(y_{n+1}) = d$, then any assignment to the other variables is satisfying. If $a(y_{n+1}) = c$, then a is satisfying if and only if $a|_{\{v_1,\dots,v_n\}} \in \phi_{\text{star},n}(\mathcal{A})$. Let $\text{noclique}_n(G)$ be the sets of vertices of G that do *not* induce a clique of size n in G. Then

$$Q(\Phi) = X_d \cdot \sum_{(v_1,\dots,v_n) \in V^n} \prod_{i \in [n]} X_{v_i} \quad + \quad X_c \cdot \sum_{\substack{(v_1,\dots,v_n) \in V^n \\ \{v_1,\dots,v_n\} \in \text{noclique}_n(G)}} \prod_{i \in [n]} X_{v_i}.$$

Setting $X_c := -X_d$ simplifies the expression to

$$Q(\Phi)|_{X_c := -X_d} = X_d \cdot \sum_{\substack{(v_1,\dots,v_n) \in V^n, \\ \{v_1,\dots,v_n\} \text{ induces} \\ \text{a clique of size } n \text{ in } G}} \prod_{i \in [n]} X_{v_i}.$$

Observe that we sum each monomial $n!$ times, because we sum once for every permutations of the set $\{v_1, \ldots, v_n\}$. Thus setting $X_d := \frac{1}{n!}$ yields $CP_{G,n+1} \leq Q(\Phi_{n+1})$ as desired. ■

We also get a version of Proposition 10.3.3 for the P-polynomial.

Corollary 10.3.4. *Let (Φ_n) be a p-bounded family of Boolean* ACQ*-instances, then $(P(\Phi_n)) \in$ VNP. Moreover, there is a family (Φ_n) of p-bounded Boolean* ACQ*-instances such that $(P(\Phi_n))$ is* VNP*-complete.*

Proof. The upper bound follows directly by applying Lemma 10.1.5 and the upper bound of Proposition 10.3.3.

For the lower bounds we reduce from the hard polynomials of Proposition 10.3.3 by using Lemma 10.1.6. This is possible, because blow-up graphs of acyclic graphs are obviously acyclic. ■

Observe that the instances Φ_n of Corollary 10.3.4 are acyclic but not of bounded treewidth because their atoms have unbounded arity. We will see that this is not a conincidence in Section 10.5.1: For Boolean CQ-instances of bounded treewidth the resulting P-polynomials are tractable and even in VP_e.

10.3.3 *Unions and intersections of ACQ-instances*

We now show that a version of Proposition 3.4.2 is also true for the arithmetic circuit setting.

Proposition 10.3.5. *Let (Φ_n) be a family of p-bounded query instances that are conjunction (resp. disjunction) of two acyclic* CSP*-instances, then the family $(Q(\Phi_n))$ of polynomials is in* VNP*. Moreover, any family in* VNP *is a p-projection of such a $(Q(\Phi_n))$. An analogous result holds for $(P(\Phi_n))$.*

Proof (Sketch). The upper bound is shown as in the proof of Lemma 10.3.1.

The proof of the lower bound for conjunction of acyclic queries follows directly like Proposition 3.4.2. The case of disjunction is obtained by reduction from the case of conjunction. Let $\Phi = (\mathcal{A}, \phi(x))$ and $\Psi = (\mathcal{A}, \psi(x))$ be two acyclic CSP-instances. W.l.o.g. assume that they both are on the same structure $\mathcal{A}$ of signature τ and domain A. We denote by $\Phi \wedge \Psi$ the instance $(\mathcal{A}, \phi(x) \wedge \psi(x))$ and by $\Phi \vee \Psi$ the instance $(\mathcal{A}, \phi(x) \vee \psi(x))$. Let $\mathcal{A}'$ be a new structure of domain $A' = A \cup \{\alpha_1, \alpha_2, \alpha_3\}$ where $\alpha_1, \alpha_2, \alpha_3$ are not in D. The structure $\mathcal{A}'$ includes $\mathcal{A}$ and has two new unary relation symbols $\mathcal{R}$ and $\mathcal{S}$ which are interpreted by

$$\mathcal{R}^{\mathcal{A}'} = \{\alpha_1, \alpha_2\}, \mathcal{S}^{\mathcal{A}'} = \{\alpha_2, \alpha_3\}.$$

Let us now consider the following disjunction of two acyclic formulas:

$$\lambda(x, y) = (\phi(x) \wedge \mathcal{R}(y)) \vee (\psi(x) \wedge \mathcal{S}(y)).$$

The instance $\Lambda = (\mathcal{A}', \lambda(x,y))$ has the following tuples as solutions:

- (a, α_1) for $a \in \phi(\mathcal{A})$.
- (a, α_3) for $a \in \psi(\mathcal{A})$.
- (a, α_2) for $a \in \phi(\mathcal{A}) \cup \psi(\mathcal{A})$.

We associate each value α_i with variable Y_i and get

$$Q(\Lambda) = Y_1 Q(\Phi) + Y_3 Q(\Psi) + Y_2 Q(\Phi \vee \Psi).$$

By projection, we get

$$Q(\Phi \wedge \Psi) = Q(Y)(X, Y_1, Y_2, Y_3)|_{Y_1=1, Y_2=-1, Y_3=1}.$$

This shows that Q-polynomials obtained as the disjunction of two acyclic CSP-instances can be represented as projections of polynomials obtained by conjunction and hence this is true for all polynomial families in VNP. The case of P-polynomials follows easily with the Lemmas 10.1.5 and 10.1.6 ■

10.4 LOWER BOUNDS FOR INSTANCES OF BOUNDED WIDTH

In this section we show the lower bounds of the Theorems 10.2.1, 10.2.2 and 10.2.3. In Section 10.5 we will see the matching upper bounds for the different cases to get the upper bounds of the theorems.

We start off with the lower bound for Theorem 10.2.3.

Remember that a Boolean CSP-instance is an instance with domain $\{0,1\}$.

Lemma 10.4.1. *There is a constant $c \leq 26$ such that the following holds: For every $(f_n) \in \mathsf{VP}_e$ there is a p-bounded family (Φ_n) of Boolean* CSP*-instances with pathwidth at most c such that $(f_n) \leq_p (P(\Phi_n))$.*

Proof. Let $A_1, \ldots, A_n$ be (3×3)-matrices. We denote the entries in matrix A_i by $(X^i_{jk})_{j,k \in [3]}$. Let f_n be the $(1,1)$-entry of the product $A_1 A_2 \ldots A_n$. We will show that there is a family Φ_n of boolean CSP-instances with pathwidth 26 such that $(f_n) \leq (P(\Phi_n))$. With the well-known VP_e-completeness of (3×3)-matrix product (see [BOC92]) the claim of Lemma 10.4.1 will follow.

Let f^i_{jk} be the polynomial computed in the (j,k)-entry of the product $A_1 A_2 \ldots A_i$. We will simulate the computation $\sum_{l=1}^{3} X^{i+1}_{l,k} f^i_{j,l} = f^{i+1}_{jk}$ by CSP-instances Φ^i_n in the Boolean variables x^l_{jk} and y^l_{jk} with $l \leq i$ and $j,k \in [3]$ and we construct the Φ^i_n iteratively for each i.

While the x^l_{jk} correspond to the variables X^l_{jk} of the polynomial we want to compute, the y^l_{jk} are "selector" variables that will allow

us to choose individual entries f^i_{jk} from $P(\Phi^i_n)$. The corresponding variables Y^l_{jk} do not appear in the iterated matrix product and we get rid of them by projecting them all to 1. In order not to clutter the construction too much with these variables, we already substitute them by 1 in the polynomials $P(\Phi^i_n)$, so they never appear in our computations.

In a slight abuse of notation we write the Φ^i_n as a conjunction of atoms ϕ_l and define the relations of the atoms implicitly by propositional formulas.

Let $a_y(i,j,k) := y^i_{jk} \wedge \bigwedge_{(j'k')\neq(j,k)} \neg y^i_{j'k'}$. Note that $a_y(i,j,k)$ defines an atom in the variables y^i_{jk} that is satisfied only by the assignment $y^i_{jk} \mapsto 1$ and $y^i_{j'k'} \mapsto 0$ for $(j,k) \neq (j',k')$. Let $a_x(i,j,k)$ be the same for the variables x^i_{jk}.

We now construct the Φ^i_n iteratively. During the construction we will make sure that the following holds:

$$P\left(\Phi^i_n \wedge a_y(i,j,k)\right) = f^i_{jk}.$$

Intuitively, by fixing the y^i_{jk}, we can compute individual entries of the product $A_1 A_2 \dots A_i$.

The CSP-instance Φ^1_n has the single atom

$$\phi_1 = \bigvee_{j',k'} (y^1_{j'k'} \wedge a_x(1,j',k')).$$

We have $P(\Phi^1_n \wedge a_y(1,j,k)) = P(a_x(1,j,k)) = X^1_{jk}$ as desired.

For the construction of Φ^{i+1}_n assume that we have already constructed Φ^i_n with the desired properties. We construct Φ^{i+1}_n from Φ^i_n by adding one atom ϕ_{i+1}. We set

$$\Phi^{i+1}_n = \Phi^i_n \wedge \underbrace{\left(\bigvee_{(j',k')} \left(y^{i+1}_{j'k'} \wedge \bigvee_l (a_x(i+1,l,k') \wedge a_y(i,j',l))\right)\right)}_{:=\phi_{i+1}}.$$

We get

$$\begin{aligned}
&P(\Phi^{i+1}_n \wedge a_y(i+1,j,k)) \\
=&P\left(\Phi^i_n \wedge \left(\bigvee_l (a_x(i+1,l,k) \wedge a_y(i,j,l))\right)\right) \\
=&\sum_{l=1}^{3} P(\Phi^i_n \wedge a_x(i+1,l,k) \wedge a_y(i,j,l)) \\
=&\sum_{l=1}^{3} X^{i+1}_{l,k} P(\Phi^i_n \wedge a_y(i,j,l)) \\
=&\sum_{l=1}^{3} X^{i+1}_{l,k} f^i_{j,l} \\
=&f^{i+1}_{jk}
\end{aligned}$$

Having constructed Φ_n^n we easily get f_n as

$$f_n = P(\underbrace{\Phi_n^n \wedge a(n,1,1)}_{:=\Phi_n}).$$

Since each atom ϕ_i has at most 27 Boolean variables, (Φ_n) is p-bounded and relation bounded. So only the bound on the pathwidth remains to be shown. The primal graph $\mathcal{H}_P$ of Φ_n has the vertices x_{jk}^i and y_{jk}^i for $i \in [n]$ and $j,k \in [3]$. Each atom ϕ_i yields a clique in $\mathcal{H}_P$ with the vertices $\mathsf{var}(\phi_i)$, and there are no other edges in $\mathcal{H}_P$. We have $\mathsf{var}(\phi_1) = \{x_{j,k}^1, y_{j,k}^1 \mid j,k \in [3]\}$ and $\mathsf{var}(\phi_i) = \{x_{j,k}^i, y_{j,k}^i, y_{j,k}^{i-1} \mid j,k \in [3]\}$ for $1 < i \leq n$.

We give a path decomposition by a path $\mathcal{P}$ of the vertices $t_1, \dots t_n$ and the bags $\chi_{t_i} = \mathsf{var}(\phi_i)$. We have $\mathsf{var}(\phi_i) \cap \mathsf{var}(\phi_j) = \emptyset$ for $i \in [n]$ and $j > i+1$. Furthermore, $\mathsf{var}(\phi_i) \cap \mathsf{var}(\phi_{i+1}) = \{y_{j,k}^i \mid j,k \in [3]\}$. Thus it is easy to check that $(\mathcal{P}, (\chi_t)_{t \in T})$ is indeed a path decomposition of $\mathcal{H}_P$ and it has width 26. ■

One could show a version of Lemma 10.4.1 for bounded tree-width with a more standard parse tree argument. We instead presented this version, not only because it is stronger due to its path-width formulation but mainly because we deem the proof to be more interesting. We will see parse tree arguments in the proofs of Lemma 10.4.3 and Lemma 10.4.4.

Combining Lemma 10.4.1 and Lemma 10.1.5 directly yields the following corollary.

Corollary 10.4.2. *There is a constant $c \leq 26$ such that the following holds: For every $(f_n) \in \mathsf{VP}_e$ there is a p-bounded family (Ψ_n) of relation bounded* CSP*-instances with pathwidth at most c such that $(f_n) \leq_p (Q(\Psi_n))$.*

Next we will show the lower bound for the characterization of VP of Theorem 10.2.1. Remember that we call a CSP-instance *binary* if all its atoms have arity at most 2.

Lemma 10.4.3. *Let $(f_n) \in \mathsf{VP}$, then there is a p-bounded family (Φ_n) of binary* CSP*-instances such that $(f_n) \leq_p (Q(\Phi_n))$. Furthermore, the associated primal graph of Φ_n can be assumed to be a* tree *for every n.*

Proof. The idea of the proof is the following: We use the characterization of VP in Theorem 9.2.1 by semi-unbounded, multiplicatively disjoint circuits of logarithmic depth. It follows that there are circuits computing f_n that have logarithmic depth parse trees. We encode these parse trees into polynomial size CSP-instances whose primal graphs are trees isomorphic to the parse trees of the f_n. Summing up over all possible encodings of parse trees we get polynomials whose projection are the f_n. We now describe the construction in more detail.

Consider a polynomial $f = f_n$ from our family. By Theorem 9.2.1 we know that f_n is computed by a logarithmic depth semi-unbounded

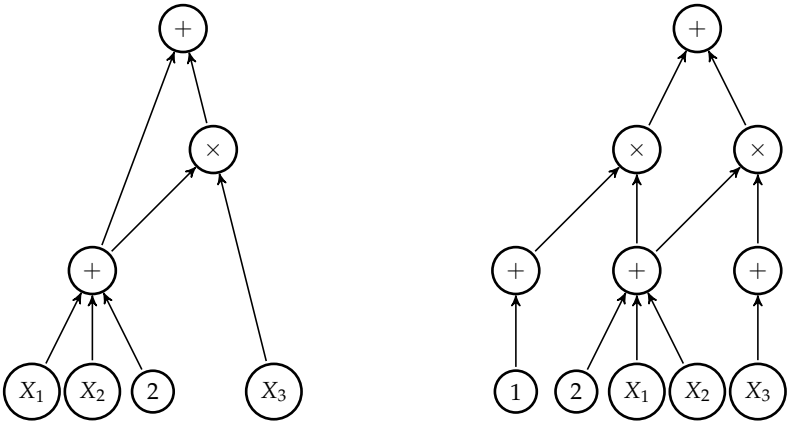

Figure 15: The original circuit on the left is changed into one in which the gates on each level have the same operation label.

circuit C of polynomial size. By adding dummies we can make sure that C has the following "layered" form (see Figure 15 for an illustration):

- All operation gates at the same depth have the same operation.
- All leaves are at the same depth level.

This layered form implies that all parse trees of C are isomorphic binary trees. Let the children of the $\times$-gates in Φ be ordered, i.e., we call one of them the left child and the other one the right child. Let T be a binary tree isomorphic to the parse trees of C. The children of vertices in T that correspond to $\times$-gates in C are also ordered.

We now construct a CSP-instance $\Phi = (\mathcal{A}, \phi)$ with $\mathsf{var}(\Phi) = V(T)$ and associated hypergraph T. The domain A of $\mathcal{A}$ is the vertex set $V(C)$ of C. To distinguish the vertices of T from the gates of C we write the vertices of T with a hat, e.g. $\hat{v} \in V(T)$. For each edge $\hat{u}\hat{v}$ in T the query ϕ has an atom $\mathcal{R}_{\hat{u}\hat{v}}(\hat{u}, \hat{v})$. The corresponding relation $\mathcal{R}^{\mathcal{A}}_{\hat{u}\hat{v}}$ is defined as follows: If $\hat{u}$ corresponds to a $+$-gate in C, then

$$\mathcal{R}^{\mathcal{A}}_{\hat{u}\hat{v}} := \{(u, v) \mid u, v \in V(C), u \text{ is a } +\text{-gate}, v \text{ is a child of } u\}.$$

If $\hat{u}$ corresponds to a $\times$-gate and $\hat{v}$ is the left child of $\hat{u}$, then

$$\mathcal{R}^{\mathcal{A}}_{\hat{u}\hat{v}} := \{(u, v) \mid u, v \in V(C), u \text{ is a } \times\text{-gate}, v \text{ is the left child of } u\}.$$

For right children $\mathcal{R}^{\mathcal{A}}_{\hat{u}\hat{v}}$ is defined in an analog fashion.

It is easy to see that Φ is satisfied by an assignment $a : V(T) \to V(C)$ if and only if a maps T onto a parse tree T_a of C. Also for each satisfying assignment a the resulting monomial $\prod_{\hat{u} \in V(T)} X_{a(\hat{u})}$ can be projected to $w(T_a)$ by doing the following: If v is an operation gate of C, then substitute X_v by 1. If v is an input gate of C with label l, then substitute X_v by l. Because each v is either an operation gate or an input gate but never both, these settings do not contradict for

different satisfying assignments of Φ. It follows that $f \leq Q(\Phi)$. The associated hypergraph of Φ is by construction the tree T. The observation that the size of ϕ and the domain $A := V(C)$ are polynomial completes the proof. ■

We use a similar parse tree argument to show the lower bound for Theorem 10.2.2.

Lemma 10.4.4. *Let $(f_n) \in \mathsf{VP}_{ws}$, then there is a p-bounded family (Φ_n) of binary* CSP*-instances such that $(f_n) \leq_p (Q(\Phi_n))$. Furthermore, the associated primal graph of Φ_n can be assumed to be a* path *for every n.*

Proof. The main difference to the proof of Lemma 10.4.3 is that for skew circuits not all parse trees are isomorphic and that we know of no way to make them isomorphic without losing skewness. This problem is remedied by the insight that parse trees of skew circuits have a very restricted form that allows encoding them into CSP-instances with paths as associated hypergraphs.

So let $(f_n) \in \mathsf{VP}_{ws}$. Then there is a family (C_n) of polynomial size skew circuits such that C_n computes f_n for every n. Let $f = f_n$ and $C = C_n$ and $s = |C|$. Each multiplication gate v of C has at least one child that is an input gate. We call this child the *leaf child* of v and the other child the *inner child* of v. If both children of v are input gates we arbitrarily choose one of them to be the leaf child and the other one to be the inner child.

Each parse tree T of C has a very special form: T consists of a path P with some dangling leaf children. An illustration is shown in Figure 16. We define the *essential path* P of a parse tree as the path we get from T after deleting all leaf children. Observe that from the essential path P one can uniquely recover the parse tree T, because each multiplication gate only has one leaf child and this child must be part of T. Note that in general not all parse trees have the same depth and that the order of the $+$- and $\times$-gates may differ in them. In particular, this results in not all essential paths having the same length.

We construct a CSP-instance $\Phi = (\mathcal{A}, \phi)$ that has the associated graph $P = (V, E)$ with $V = \{\hat{v}_i \mid i \in \{0, \dots s\}\}$ and $E = \{\hat{v}_i\hat{v}_{i+1} \mid i \in \{0, \dots s-1\}\}$. We have $\mathsf{var}(\phi) := V$ and the domain $A := V(C) \cup \{d\}$ for a dummy value d. Note that ϕ has more variables than any parse tree of C has vertices, so that we cannot simply map $\mathsf{var}(\phi)$ onto the parse trees like in the proof of Lemma 10.4.3. Instead we will map onto essential paths of parse trees and map redundant vertices in $\mathsf{var}(\Phi)$ onto the dummy d. By doing the latter we will deal with the fact that the parse trees are not isomorphic.

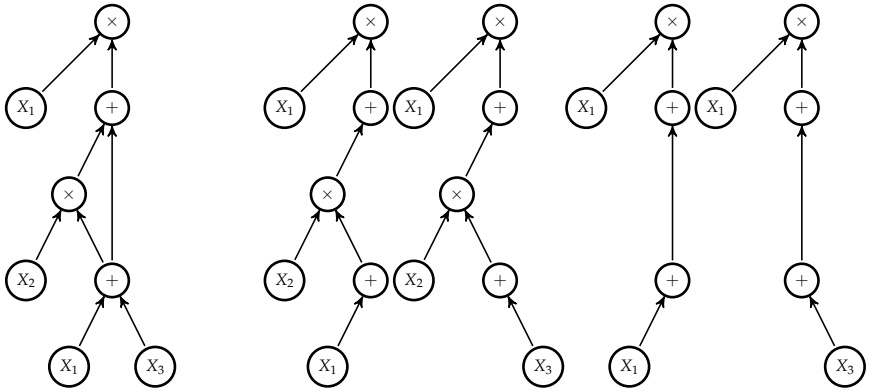

Figure 16: A skew cicuit and all of its parse trees. The polynomial computed equals $X_1^2X_2 + X_1X_2X_3 + X_1^2 + X_1X_3$.

For each edge $e = \hat{v}_i\hat{v}_{i+1}$ the query ϕ has an atom $\mathcal{R}_{\hat{v}_i\hat{v}_{i+1}}(\hat{v}_i, \hat{v}_{i+1})$ which is interpreted by the relation

$$\begin{aligned}\mathcal{R}^{\mathcal{A}}_{\hat{v}_i\hat{v}_{i+1}} :=& \{(v, v') \mid v \times\text{-gate}, v' \text{ its inner child}\}\\ &\cup \{(v, v') \mid v +\text{-gate}, v' \text{ child of } v\ \}\\ &\cup \{(v, d) \mid v \text{ input gate}\ \}\\ &\cup \{(d, d)\}.\end{aligned}$$

Furthermore ϕ has one unary atom $\mathcal{R}_{\hat{v}_0}(v_0)$ which is interpreted by the relation $\mathcal{R}^{\mathcal{A}}_{\hat{v}_0} := \{v^*\}$ where v^* is the output gate of C.

It is easy to see that the satisfying assignments $a\colon V \to V(C)$ of Φ are exactly the encodings of essential paths of C: The atom $\phi_{\hat{v}_0}$ forces the satisfying assignments to map $\hat{v}_0$ to the output gate v^*; the other atoms force to assign the other variables along an essential path. As discussed before, the satisfying assignments are "filled up" with dummies to deal with the different size of the parse trees. We now project from $Q(\Phi)$ as follows: For each input gate $\hat{v}$ of C we substitute X_v by the label of v in C. For each $+$-gate v of C we substitute X_v by 1. For each $\times$-gate v with leaf child u we substitute X_v by the label of u in C. Finally, we substitute X_d by 1. Obviously the computed polynomial is f and thus $f \leq Q(\Phi)$. ■

10.5 CONSTRUCTING CIRCUITS FOR CONJUNCTIVE QUERIES

In this section we will prove the upper bounds of the Theorems 10.2.1, 10.2.2 and 10.2.3, i.e., we will show that the structural restrictions of CQ-instances we considered in Part i yield tractable polynomials. Using Lemma 5.2.1 it will be enough to show this for acyclic, quantifier free instances. We start off with a lemma that lets us balance join trees.

Lemma 10.5.1. *For every* ACQ*-instance* $\Phi = (\mathcal{A}, \phi)$ *one can in polynomial time construct a solution equivalent instance* $\Psi = (\mathcal{B}, \psi)$ *and a join*

tree $(\mathcal{T}, (\lambda_t)_{t\in T})$ *of* Ψ *such that the tree* $\mathcal{T} = (T, F)$ *is binary and and of depth* $O(\log(|T|))$. *Furthermore, if all relations in* $\mathcal{A}$ *have size at most* ℓ, *then all relations in* $\mathcal{B}$ *have size at most* ℓ^4.

Proof. Given $\Phi = (\mathcal{A}, \phi)$, first use the algorithm of Lemma 2.3.17 to compute in polynomial time a join tree $(\mathcal{T}, (\lambda_t)_{t\in T})$ with $\mathcal{T} = (T, F)$. Until $\mathcal{T}$ is binary, do the following: Take a vertex $t \in T$ with children $t_1, t_2, \ldots, t_\ell$ for $\ell \geq 3$. Add a new vertex t', delete the edges tt_1 and tt_2 and add the edges tt', $t't_1$ and $t't_2$. We set $\lambda_{t'} = \lambda_t$. Clearly, after a linear number of iterations the tree is binary and still a join tree of Φ. To keep the notation simple we also denote this tree by $\mathcal{T}$.

Now apply Lemma 2.3.10 to $\mathcal{T}$. Let $(\mathcal{T}', (\chi_{t'})_{t'\in T'})$ with $\mathcal{T}' = (T', F')$ be the resulting tree decomposition of width 3 of $\mathcal{T}$. We construct a new ACQ-instance $\Psi = (\mathcal{B}, \psi)$ as follows: For each $t' \in T'$ let $\phi_{t',1}$, $\phi_{t',2}$, $\phi_{t',3}$ and $\phi_{t',4}$ be the atoms of ϕ that belong to the edges λ_{t_1}, λ_{t_2}, λ_{t_3} and λ_{t_4} where $\{t_1, t_2, t_3, t_4\} = \chi_{t'}$ (if $\chi_{t'}$ contains fewer than 4 vertices, we act accordingly on less atoms). For each $t' \in T'$ the query ψ has an atom $\psi_{t'}$ in the variables $\bigcup_{i=1}^{4} \mathsf{var}(\phi_{t_i})$. We define the corresponding relation as $\mathcal{R}^{\mathcal{B}}_{t'} := \phi_{t_1}(\mathcal{A}) \bowtie \phi_{t_2}(\mathcal{A}) \bowtie \phi_{t_3}(\mathcal{A}) \bowtie \phi_{t_4}(\mathcal{A})$. The query ψ has the same quantifiers as ϕ. This completes the construction of Ψ.

Obviously, ϕ and ψ are solution equivalent. Moreover, the bound on the size of the relations of $\mathcal{B}$ is clear from the construction. Set $\lambda'_{t'} := e_{t'}$ where $e_{t'}$ is the edge induced by $\psi_{t'}$.

We claim that $(\mathcal{T}', (\lambda'_{t'})_{t'\in T'})$ is a join tree of ψ which will complete the proof. We only have to check the connectivity condition. So let v be any variable of ψ. In $\mathcal{T}$ the set $C := \{t \in T \mid v \in \lambda_t\}$ is connected, because $\mathcal{T}$ is a join tree. Also for each $t \in T$ the set $C_t := \{t' \in T' \mid t \in \chi_{t'}\}$ is connected in $(\mathcal{T}', (\chi_{t'})_{t'\in T})$. Furthermore, for each edge $t_1t_2 \in F$ we have that t_1 and t_2 are in one common bag $\chi_{t'}$. Consequently, the set $C_{t_1} \cup C_{t_2}$ is connected in $(\mathcal{T}', (\chi_{t'})_{t'\in T})$. It follows that for every set $C' \subseteq T$ that is connected in $\mathcal{T}$ the set $\{t' \in T' \mid \exists t \in C' : t \in \chi_{t'}\}$ is connected in $(\mathcal{T}', (\chi_{t'})_{t'\in T})$. Thus we get that $\{t' \in T' \mid \exists t \in T\colon v \in \lambda_t, t \in \chi_{t'}\} = \{t' \in T' \mid \exists t \in C\colon t \in \chi_{t'}\} = \{t' \in T' \mid v \in \lambda'_{t'}\}$ is connected which completes the proof. ∎

Proposition 10.5.2. *There is an algorithm that, given an* acyclic CSP*-instance* $\Phi = (\mathcal{A}, \phi)$, *constructs in time polynomial in* $\|\Phi\|$ *an arithmetic* circuit C *that computes* $Q(\Phi)$.

Proof. We construct C by dynamic programming on the join tree of Φ. So let $\Phi = (\mathcal{A}, \phi)$ be an acyclic CSP-instance. Let $(\mathcal{T}, (\lambda_t)_{t\in T})$ be the join tree associated to ϕ. By Lemma 2.3.17 $\mathcal{T}$ can be constructed from ϕ in polynomial time and thus we do not consider the construction of $\mathcal{T}$ but take it as given. By Lemma 10.5.1 we may w.l.o.g. assume that $\mathcal{T}$ is binary. By possibly adding some additional leaves to $\mathcal{T}$ we may also assume that $\mathcal{T}$ is a full binary tree, i.e., every vertex

in T that is not a leaf has exactly two children. By definition, the tree $\mathcal{T}$ has m vertices $t_1, \ldots, t_m$ associated to the edges $\lambda_{t_1}, \ldots, \lambda_{t_m}$ in the hypergraph $\mathcal{H}$ associated to ϕ. To ease notation we do not differentiate between the atoms of ϕ and the edges λ_t induced by them in $\mathcal{H}$ but denote the atoms as λ_t.

For $t \in T$ we call ϕ_t the conjunction of atoms corresponding to the subtree $\mathcal{T}_t$ with t as root. The set $\mathsf{var}(\phi_t) = \bigcup_{t' \in \mathcal{T}_t} \mathsf{var}(\lambda_{t'})$ is denoted by e_t.

To every $t \in T$ we assign a set $c(t) \subseteq e_t$ as follows: To the root r of $\mathcal{T}$ we assign $c(r) := \mathsf{var}(\phi)$. To a vertex t with parent $\tilde{t}$ we assign $c(t) := e_t \setminus \lambda_{\tilde{t}}$. Furthermore, we assign to every $t \in T$ the set $c_0(t) := \mathsf{var}(\lambda_t) \cap e_t$. Observe that for a vertex t with children t_1, t_2 the sets $c_0(t)$, $c(t_1)$ and $c(t_2)$ form a partition of $c(t)$ into disjoint sets. This partition will make sure that, in the dynamic programming below, each variable x appearing in several λ_{t_i} will be taken into account at most once and thus the exponent of $X_{a(x)}$ will not be overcounted in $Q(\Phi)$.

We define for every $t \in T$ the set $v_t \subseteq \mathsf{var}(\lambda_t)$. For the root r of $\mathcal{T}$ we set $v_t := \mathsf{var}(\lambda_r)$. For all other $t \in T$ with parent $\tilde{t}$ we set $v_t := \mathsf{var}(\tilde{t}) \cap \mathsf{var}(t)$.

Let $t \in T$ and let a be an assignment to v_t. Remember that for an assignment α we say that a and α are compatible, in symbols $a \sim \alpha$, if they agree on their common variables. We consider the polynomial in the variables $\{X_d \mid d \in A\}$

$$f_{t,a} = \sum_{\substack{\alpha \in \phi_t(\mathcal{A}) \\ \alpha \sim a}} \prod_{x \in c(t)} X_{\alpha(x)}.$$

We will show by induction on $\mathcal{T}$ that $f_{t,a}$ can for every fixed combination of t and a be computed by an arithmetic circuit of polynomial size.

For every $t \in T$ let A_t be the set of assignments $a : \mathsf{var}(\lambda_t) \to A$ in $\lambda_t(\mathcal{A})$ that can be extended to a satisfying assignment in $\phi_t(\mathcal{A})$. Obviously, if t is a leaf, then $A_t := \lambda_t(\mathcal{A})$. If t has two children t_1, t_2, then

$$A_t := \pi_{\mathsf{var}(\lambda_t)}(\phi_t(\mathcal{A})) = (\lambda_t(\mathcal{A}) \ltimes \phi_{t_1}(\mathcal{A})) \ltimes \phi_{t_2}(\mathcal{A}).$$

Claim 10.5.3. *For every $t \in T$ with children t_1, t_2 and every assignment $a : v_t \to A$ we have*

$$f_{t,a} = \sum_{\substack{\beta \in A_t \\ \beta \sim a}} \prod_{x \in c_0(t)} X_{\beta(x)} \cdot f_{t_1, \beta|_{v_{t_1}}} \cdot f_{t_2, \beta|_{v_{t_2}}}.$$

Proof. By definition of $f_{t,a}$ we have

$$\begin{aligned} f_{t,a} &= \sum_{\substack{\alpha \in \phi_t(\mathcal{A}) \\ \alpha \sim a}} \prod_{x \in c(t)} X_{\alpha(x)} \\ &= \sum_{\substack{\alpha \in \phi_t(\mathcal{A}) \\ \alpha \sim a}} \prod_{x \in c_0(t)} X_{\alpha(x)} \prod_{x \in c(t_1)} X_{\alpha(x)} \prod_{x \in c(t_2)} X_{\alpha(x)}. \end{aligned}$$

Each tuple $\alpha \in \phi_t(\mathcal{A})$ can be uniquely expressed as the natural join of a tuple $\beta \in A_t$ with two tuples $\alpha_1 \in \phi_{t_1}(\mathcal{A})$ and $\alpha_2 \in \phi_{t_2}(\mathcal{A})$ compatible with β (more formally by the natural join of singleton relations containing these tuples), i.e., given $\alpha \in \phi_t(\mathcal{A})$, there exist $\beta \in A_t$ and $\alpha_1 \in \phi_{t_1}(\mathcal{A})$ and $\alpha_2 \in \phi_{t_2}(\mathcal{A})$ such that

$$\{\alpha\} = \{\beta\} \bowtie \{\alpha_1\} \bowtie \{\alpha_2\}.$$

Conversely, given $\beta \in A_t$ and a pair $\alpha_1 \in \phi_{t_1}(\mathcal{A})$ and $\alpha_2 \in \phi_{t_2}(\mathcal{A})$ compatible with β, the natural join α of β, α_1 and α_2 is contained in $\alpha \in \phi_t(\mathcal{A})$. This follows from the connectedness condition in the join tree, i.e., from the fact that $\mathsf{var}(\phi_{t_1}) \cap \mathsf{var}(\phi_{t_2}) \subseteq \mathsf{var}(\lambda_t)$ and thus α_1 and α_2 agree on their common variables, because they both agree with β.

These reasonings impliy the following equalities:

$$\begin{aligned} f_{t,a} &= \sum_{\substack{\alpha \in \phi_t(\mathcal{A}) \\ \alpha \sim a}} \prod_{x \in c_0(t)} X_{\alpha(x)} \prod_{x \in c(t_1)} X_{\alpha(x)} \prod_{x \in c(t_2)} X_{\alpha(x)} \\ &= \sum_{\substack{\beta \in A_t \\ \beta \sim a}} \sum_{\substack{\alpha_1 \in \phi_{t_1}(\mathcal{A}) \\ \alpha_1 \sim \beta}} \sum_{\substack{\alpha_2 \in \phi_{t_2}(\mathcal{A}) \\ \alpha_2 \sim \beta}} \prod_{x \in c_0(t)} X_{\beta(x)} \prod_{x \in c(t_1)} X_{\alpha_1(x)} \prod_{x \in c(t_2)} X_{\alpha_2(x)} \\ &= \sum_{\substack{\beta \in A_t \\ \beta \sim a}} \prod_{x \in c_0} X_{\beta(x)} \cdot f_{t_1,\beta|_{v_{t_1}}} \cdot f_{t_2,\beta|_{v_{t_2}}} \end{aligned}$$

This proves the claim. ■

Note that the sum is now over A_t and not over $\phi_t(\mathcal{A})$ anymore.

Claim 10.5.4. *The relation A_t can be computed in time*

$$O(\|\mathcal{A}\| \log(\|\mathcal{A}\|)|\phi_t|).$$

Proof. The proof is an adaptation of Yannakakis' [Yan81] algorithm for ACQ (compare Theorem 2.3.19). We proceed by induction on $\mathcal{T}_t$. If t is a leaf, the result is obvious.

If t is not a leaf, let $t_1, t_2 \in T$ be its children. Note that

$$A_t = (\lambda_t(\mathcal{A}) \ltimes A_{t_1}) \ltimes A_{t_2},$$

since each A_{t_i} is the projection of $\phi_{t_i}(\mathcal{A})$ onto $\mathsf{var}(\lambda_{t_i})$. By Lemma 2.1.6 we can thus compute A_t from A_{t_1} and A_{t_2} in time $O(\|\mathcal{A}\| \log(\|\mathcal{A}\|))$. By induction each A_{t_i} is computable in time $O(\|\mathcal{A}\| \log(\|\mathcal{A}\|)|\phi_{t_i}|)$. Thus, A_t is computable in time $O(\|\mathcal{A}\| \log(\|\mathcal{A}\|)(1 + |\phi_{t_1}| + |\phi_{t_2}|)) \leq O(\|\mathcal{A}\| \log(\|\mathcal{A}\|)|\phi_t|)$. ■

Claim 10.5.5. *For evert $t \in T$ and every assignment $a : v_t \to A$ the polynomial $f_{t,a}$ can be computed by a circuit C of size $O(|\phi_t|\|\mathcal{A}\|^2)$. Furthermore, the circuit C can be computed in polynomial time.*

Proof. First observe, that if a cannot be extended to a tuple $a' \in A_t$, then $f_{t,a} = 0$ and the claims are trivial. For the other assignments a we make a slightly stronger claim: For every $t \in T$ we can compute a circuit of size $O(|\phi_t|\|\mathcal{A}\|^2)$ that computes $f_{t,a}$ for *all* $a : v_t \to A$ that can be extended to an assignment $a' \in A_t$.

We make an induction on the number of atoms in $|\phi_t|$.

Assume first that t is a leaf. Then ϕ_t is an atom λ_t. Hence every $f_{t,a}$ can be computed naively because it consists only of $|\lambda_t(\mathcal{A})|$ monomials whose degree is at most the arity of λ_t. For fixed a this circuit has size at most $|\phi_t| \cdot \|\mathcal{A}\|$, so we can compute all of the $f_{t,a}$ with a circuit of size $|\phi| \cdot \|\mathcal{A}\| \cdot |A_t| \leq |\phi| \cdot \|\mathcal{A}\|^2$. Certainly, the construction can be done in polynomial time.

Let t now be a vertex with children t_1, t_2. We only have to compute the $f_{t,a}$ for $|A_t| \leq \|\mathcal{A}\|$ assignments a. By Claim 10.5.3, we can compute $f_{t,a}$ for a fixed a from the $f_{t_i,\beta|_{v_{t_i}}}$ with $O(|A_t| \cdot |c_0(t)|) = O(\|\mathcal{A}\| \cdot |\phi|)$ arithmetic operations. By induction we can compute the necessary $f_{t_i,\beta|_{v_{t_i}}}$ by a circuit of size $O(|\phi_{t_i}| \cdot \|\mathcal{A}\|)$. Then we can compute all $f_{t,a}$ by a circuit of size $O((|\phi_1| + |\phi_2| + 1)\|\mathcal{A}\|^2) = O(|\phi_t| \cdot \|\mathcal{A}\|^2)$.

Now remark that each set A_t for $t \in T$ can be constructed in time $O(|\phi|\|\mathcal{A}\| \log(\|\mathcal{A}\|))$ by Claim 10.5.4. For a fixed a, filtering all elements β of A_t compatible with a can be done in linear time after sorting A_t. Hence, the index set of each sum is efficiently computable and the construction of the circuit can be done in polynomial time. ■

With the observation that $Q(\Phi) = \sum_{a \in A_r} f_{r,a}$ where r is the root of $\mathcal{T}$ we get by Claim 10.5.5 a polynomial size circuit C computing $Q(\Phi)$. Furthermore, C can be computed in polynomial time. ■

As a corollary of Proposition 10.5.2 we get Theorem 3.1.2 as we promised in Chapter 3.

Theorem 3.1.2 ([PS13]). #ACQ *restricted to quantifier-free instances can be solved in polynomial time.*

Proof. Given an instance Φ, compute a circuit C that computes $Q(\Phi)$ with Proposition 10.5.2 and then evaluate at the input $(1, \ldots, 1)$. ■

In the arithmetic circuit setting we get the following result.

Corollary 10.5.6. *Let (Φ_n) be a p-bounded family of acyclic* CSP*-instances, then $(Q(\Phi_n)) \in$ VP.*

Now we can use Lemma 5.2.1 to extend Corollary 10.5.6 to CQ-instances. We combine this also with the lower bound from Lemma 10.4.3 to give the promised characterization for VP.

Theorem 10.2.1. *For every p-bounded family (Φ_n) of* CQ*-instances with bounded generalized hypertree width and bounded quantified star size, the family $(Q(\Phi_n))$ is in* VP*. Moreover, every family in* VP *is a p-projection of such $(Q(\Phi_n))$.*

Obviously, on could easily generalize Theorem 10.2.1 to all decomposition techniques from Corollary 5.2.5.

There is also a path version of Proposition 10.5.2.

Proposition 10.5.7. *There is an algorithm that, given an acyclic* CSP*-instance* $\Phi = (\mathcal{A}, \phi)$ *that has a join tree* $(\mathcal{T}, (\lambda_t)_{t \in T})$ *such that* $\mathcal{T}$ *is a* path*, constructs in time polynomial in* $\|\Phi\|$ *a* skew arithmetic circuit C *that computes* $Q(\Phi)$*.*

Proof (sketch). The proof is a modification of that of Proposition 10.5.2. Since every vertex $t \in T$ has at most one child t^* the computation of $f_{t,a}$ is given as

$$f_{t,a} = \sum_{\substack{\beta \in A_t \\ \beta \sim a}} \prod_{x \in c_0(t)} X_{\beta(x)} \cdot f_{t^*,\beta}. \tag{6}$$

Observe that the product in (6) contains only one factor that is not an input. Thus the resulting circuit is skew. ■

One could now formulate a version of Theorem 10.2.1 for a path version of the generalized hypertree decompositions, but since this path version is not commonly considered in the literature, we only formulate the promised theorem for pathwidth.

Theorem 10.2.2. *For every p-bounded family* (Φ_n) *of* CQ*-instances with bounded pathwidth and bounded quantified star size, the family* $(Q(\Phi_n))$ *is in* VP_{ws}*. Moreover, every family in* VP_{ws} *is a p-projection of such* $(Q(\Phi_n))$*.*

Proof. The upper bound follows by the combination of Lemma 5.2.3 and Proposition 10.5.7.

The lower bound here follows from Lemma 10.4.4. ■

10.5.1 *The relation bounded case*

We now consider relation bounded families of CQ-instances, i.e., we allow existential quantification but assume that the families we consider have a bound ℓ on the size of all occuring relations in the instance. As discussed in Section 10.2, Pichler and Skritek [PS13] have shown that #CQ is tractable on such instances. The following theorem is a version of this result for the arithmetic circuit setting. The proof is a combination of techniques used by Pichler and Skritek and the proof of Proposition 10.5.2.

Proposition 10.5.8. *There is an algorithm that, given an* ACQ*-instance* $\Phi = (\mathcal{A}, \phi)$ *such that the size of each relation in* $\mathcal{A}$ *is bounded by* ℓ*, constructs in time polynomial in* $\|\Phi\|$ *and exponential in* ℓ *an arithmetic* formula *that computes* $Q(\Phi)$*.*

Proof. We use the same notation as in the proof of Proposition 10.5.2. Let $(\mathcal{T}, (\lambda_t)_{t\in T})$ be a join tree of Φ. With Lemma 10.5.1 we may assume that $\mathcal{T} = (T, F)$ has depth $O(\log(|\phi|))$ and is a binary tree. Furthermore, we assume that whenever $t \in T$ has two children t_1, t_2, then all three vertices have the same atom, i.e., $\lambda_t = \lambda_{t_1} = \lambda_{t_2}$. This form can always be easily achieved by introducing new vertices. To ease notation let $X := \mathsf{free}(\phi)$ and $Y := \mathsf{var}(\phi) \setminus \mathsf{free}(\phi)$. Remember that for an assignment a and a set of variables Z we denote by $a|_Z$ the restriction of a onto Z. We sometimes also use this notation if $Z \nsubseteq \mathsf{var}(a)$. In this case $a|_Z := a|_{Z\cap\mathsf{var}(a)}$. If a and a' are assignments with $a \sim a'$, we write in a slight abuse of notation $a \bowtie a'$ for the single assignment in the relation $\{a\} \bowtie \{a'\}$.

For $t \in T$ let ϕ'_t be the query that we get from ϕ_t by deleting all quantifiers. Let $X_t := X \cap \mathsf{var}(\lambda_t)$ and $Y_t := Y \cap \mathsf{var}(\lambda_t)$.

For every $t \in T$ we define the set $c(t)$ containing free variables as follows: For the root r of $\mathcal{T}$ we set $c(r) := \mathsf{free}(\phi_r)$. For a vertex t with parent $\tilde{t}$ we set $c(t) := \mathsf{free}(\phi_t) \setminus X_{\tilde{t}}$. Moreover, we set $c_0(t) := X_t \cap c(t)$. Observe that for a vertex $t \in T$ with children t_1, t_2 the sets $c_0(t)$, $c(t_1)$ and $c(t_2)$ form a disjoint partition of $c(t)$.

We will do dynamic programming similar to the proof of Proposition 10.5.2. The difference here is that we have to take the quantified variables into account when doing the dynamic programming step. It will be necessary to have knowledge on how an assignment $b \in \phi_t(\mathcal{A})$ can be extended to the quantified variables. To this end, we now define a decomposition of the set of satisfying assignments $\phi_t(\mathcal{A})$.

For each $t \in T$ we define a mapping $I_t : \phi_t(\mathcal{A}) \mapsto \mathcal{P}(\pi_{Y_t}(\lambda_t(\mathcal{A})))$ where $\mathcal{P}(\pi_{Y_t}(\lambda_t(\mathcal{A})))$ is the power set of $\pi_{Y_t}(\lambda_t(\mathcal{A}))$. For $b \in \phi_t(\mathcal{A})$ we define $I_t(b)$ as

$$I_t(b) := \{a' \in \pi_{Y_t}(\lambda_t(\mathcal{A})) \mid \exists b' \in \phi'_t(\mathcal{A}) : b' \sim a' \wedge b' \sim b)\}.$$

For a tuple $a \in \pi_{X_t}(\lambda_t(\mathcal{A}))$ and a nonempty set $I \subseteq \pi_{Y_t}(\lambda_t(\mathcal{A}))$ we define $N(t, a, I)$ as the fiber of (a, I) under the mapping $b \mapsto (b|_{X_t}, I_t(b))$, i.e., $N(t, a, I)$ is the set of assignments $b \in \phi_t(\mathcal{A})$ with $b \sim a$ and $I_t(b) = I$.

From the definition we directly get the decomposition

$$\phi_t(\mathcal{A}) = \dot{\bigcup_{\substack{a\in\pi_{X_t}(\lambda_t(\mathcal{A}))\\ I\subseteq\pi_{Y_t}(\lambda_t(\mathcal{A}))}}} N(t, a, I). \tag{7}$$

We will show how to compute the polynomials

$$f_{t,a,I} := \sum_{b\in N(t,a,I)} \prod_{x\in c(t)} X_{b(x)}$$

for every $t \in T$, $a \in \pi_{X_t}(\lambda_t)$ and $I \subseteq \pi_{Y_t}(\lambda_t(\mathcal{A}))$.

First observe that from the $f_{t,a,I}$ we can compute $Q(\Phi)$. Indeed, let r be the root of $\mathcal{T}$, then

$$Q(\Phi) := \sum_{\substack{a \in \pi_{X_r}(\lambda_r(\mathcal{A})),\\ I \subseteq \pi_{Y_r}(\lambda_r(\mathcal{A}))}} f_{r,a,I}, \tag{8}$$

because of the decomposition (7). Since $|\lambda_r(\mathcal{A})| \leq \ell$ we only have to show how to compute the $f_{t,a,I}$ for fixed t, a and I by formulas of size polynomial in $\|\Phi\|$ and the theorem will follow.

Let first t be a leaf. Note that in this case $\phi_t = \lambda_t$. Let $a \in \pi_{X_t}(\lambda_t(\mathcal{A}))$ and let I' be the set of possible extensions of a to Y_t, i.e., $I' := \{a' \in \pi_{Y_t}(\lambda_t(\mathcal{A})) \mid a \bowtie a' \in \lambda_t(\mathcal{A})\}$. Then $N(t,a,I) = \{a\}$ if $I = I'$, otherwise $N(t,A,I) = \emptyset$. It follows that

$$f_{t,a,I} = \begin{cases} \prod_{x \in c(t)} X_{a(x)} & \text{if } I = I', \\ 0 & \text{otherwise.} \end{cases} \tag{9}$$

Let t now be a vertex with a single child t'. Let

$$A_a := \{a' \in \pi_{X_{t'}}(\lambda_{t'}(\mathcal{A})) \mid a' \sim a\}.$$

For each $a' \in A_a$ let

$$I_{a'} := \{a'' \in \pi_{Y_{t'}}(\lambda_{t'}(\mathcal{A})) \mid a' \bowtie a'' \in \lambda_{t'}(\mathcal{A})\}.$$

Let similarly

$$I_a := \{a'' \in \pi_{Y_t}(\lambda_t(\mathcal{A})) \mid a \bowtie a'' \in \lambda_t(\mathcal{A})\}.$$

Claim 10.5.9. *$b \in N(t,a,I)$ if and only if $b|_{X_t} = a$ and $b|_{\mathsf{free}(\phi_{t'})} \in N(t',a',I')$ for some $a' \in A_a$ and $I' \subseteq I_{a'}$ with $I_a \ltimes I' = I$.*

We skip the proofs of Claim 10.5.9 and some later claims and present them at the end of the proof of Proposition 10.5.8.

We have by Claim 10.5.9

$$f_{t,a,I} = \prod_{c \in c_0} X_{a(x)} \cdot \sum_{a' \in A_a} \sum_{\substack{I' \subseteq I_{a'},\\ I_a \ltimes I' = I}} f_{t',a',I',c_1} \tag{10}$$

.

Let now t be a vertex with two children t_1 and t_2. By assumption we have that $\lambda_t = \lambda_{t_1} = \lambda_{t_2}$.

Claim 10.5.10. *$b \in N(t,a,I)$ if and only if $b|_{\mathsf{free}(\phi_{t_1})} \in N(t_1,a,I_1)$ and $b|_{\mathsf{free}(\phi_{t_2})} \in N(t_2,a,I_2)$ with $I_1 \cap I_2 = I$.*

We claim that

$$f_{t,a,I} = \sum_{\substack{I_1, I_2 \subseteq \pi_{Y_t}(\lambda_t(\mathcal{A}))\\ I_1 \cap I_2 = I}} f_{t_1,a,I_1} \cdot f_{t_2,a,I_2}. \tag{11}$$

Indeed, we have by Claim 10.5.10 and the assumption $\lambda_t = \lambda_{t_1} = \lambda_{t_2}$

$$\begin{aligned}
&f_{t,a,I} \\
&= \prod_{x \in c_0(t)} X_{a(x)} \cdot \sum_{\substack{I_1, I_2 \subseteq \pi_{Y_t}(\lambda_t(\mathcal{A})) \\ I_1 \cap I_2 = I}} \sum_{\substack{b_1 \in N(t,a,I_1), \\ b_2 \in N(t,a,I_2)}} \prod_{x \in c(t_1)} X_{b_1(x)} \prod_{x \in c(t_2)} X_{b_2(x)} \\
&= \prod_{x \in c_0(t)} X_{a(x)} \cdot \sum_{\substack{I_1, I_2 \subseteq \pi_{Y_t}(\lambda_t(\mathcal{A})) \\ I_1 \cap I_2 = I}} \left(\sum_{b_1 \in N(t_1,a,I_1)} \prod_{x \in c(t_1)} X_{b_1(x)} \right) \\
&\qquad \cdot \left(\sum_{b_2 \in N(t_2,a,I_2)} \prod_{x \in c(t_2)} X_{b_2(x)} \right) \\
&= \prod_{x \in c_0(t)} X_{a(x)} \cdot \sum_{\substack{I_1, I_2 \subseteq \pi_{Y_t}(\lambda_t(\mathcal{A})) \\ I_1 \cap I_2 = I}} f_{t_1,a,I_1} \cdot f_{t_2,a,I_2}.
\end{aligned}$$

The equations (9), (10) and (11) let us compute $f_{t,a,I}$ by dynamic programming. We will see that this computation is efficient.

Claim 10.5.11. *For every choice of t, a and I, the polynomial $f_{t,a,I}$ can be computed by a circuit of depth $(2\ell+2)d_t + \log(|c(t)|)$ where d_t is the depth of $\mathcal{T}_t$.*

Since $\mathcal{T}$ has depth $O(\log(|\phi|))$ it follows that $Q(\Phi)$ can be computed by a circuit C of depth $O(\log(|\phi|))$ by Claim 10.5.11 and (8). Unfolding this circuit we get a formula of size polynomial in $|\phi|$ computing $Q(\Phi)$. It is easy to see that the whole construction can be done in time polynomial in $|\phi|$. Thus it only remains to prove the claims.

Proof of Claim 10.5.9. For $b \in N(t,a,I)$ there must be clearly a tuple $a' \in A_a$ such that $b \sim a'$. But $\phi_{t'}$ decomposes into sets $N(t', a^*, I^*)$ by (7) and thus $b|_{\mathsf{free}(\phi_{t'})} \in N(t, a', I')$ for some $I' \subseteq I_{a'}$. Thus we only have to prove that $I_a \ltimes I' = I$. First assume that there is $b' \in (I_a \ltimes I') \setminus I$. Then there is a b'' such that $b \bowtie b'' \in \phi'_t(\mathcal{A})$ and $(b \bowtie b'')|_{Y_t} = b'$ and thus $b \notin N(t,a,I)$ which contradicts the assumption. Now assume that there is $b' \in I \setminus (I_a \ltimes I')$. Then there is no b'' such that $(b \bowtie b'') \in \phi'_t(\mathcal{A})$ and $(b \bowtie b'')|_{X_t} = b'$ and thus $b \notin N(t,a,I)$ which is again a contradiction.

Now now consider an assignment b of $\mathsf{free}(\phi_t)$ with $b|_{X_t} = a$ and $b|_{\mathsf{free}(\phi_{t'})} \in N(t', a', I')$ for some $a' \in A_a$ and some $I' \subseteq I_{a'}$ with $I_a \ltimes I' = I$. We show $b \in N(t,a,I)$.

Obviously $a \sim b$.

Consider now $a' \in I$. Because of $I = I_a \ltimes I'$ there is an $a'' \in I'$ such that $a'' \sim a'$. Because of $b|_{\mathsf{free}(\phi_{t'})} \in N(t', a', I')$ there is a $b'' \in \phi'_{t'}(\mathcal{A})$ with $b''|_{\mathsf{free}(\phi_t)} = b|_{\mathsf{free}(\phi_{t'})}$ and $b''|_{Y_{t'}} = a''$. It follows that $b'' \sim a'$ and $b'' \sim a$. Thus b'', a' and a can be combined to an assignment $b' := b'' \bowtie a' \bowtie a \in \phi'_t(\mathcal{A})$ with $b'|_{\mathsf{free}}(\phi_t) = b$ and $b|_{Y_t} = a'$ as desired.

Now let $b' \in \phi'_t(\mathcal{A})$ with $b'|_{\mathsf{free}}(\phi_t) = b$. Since $b|_{\mathsf{free}(\phi_{t'})} \in N(t', a', I')$, we have that $b'|_{Y_{t'}} \in I'$. Furthermore, obviously $b'|_{Y_t} \in I_a$. Thus $b'|_{Y_t} \in I_a \ltimes I' = I$ which completes the proof.

This proves Claim 10.5.9. ■

Proof of Claim 10.5.10. Let first $b \in N(t, a, I)$. Then $b \sim a$ and for each $a' \in I$ there is a $b' \in \phi'_t(\mathcal{A})$ such that $b'|_{\mathsf{free}(\phi_t)} = b$ and $b|_{Y_t} = a'$. It follows that $b|_{\mathsf{free}(t_1)} \sim a$, $b'|_{\mathsf{var}(t_1)} \in \phi'_{t_1}(\mathcal{A})$ and $b|_{Y_{t_1}} = a'$. Thus $b|_{\mathsf{free}(\phi_{t_1})} \in N(t_1, a, I_1)$ for a superset $I_1 = I_{t_1}(b|_{\mathsf{free}(\phi_{t_1})})$ of I. Analogously we get that $b|_{\mathsf{free}(\phi_{t_2})} \in N(t_2, a, I_2)$ for a superset $I_2 = I_{t_2}(b|_{\mathsf{free}(\phi_{t_2})})$ of I.

Now assume that there is an $a' \in (I_1 \cap I_2) \setminus I$. Then there is a $b'_1 \in \phi'_{t_1}(\mathcal{A})$ with $b'_1|_{\mathsf{free}(\phi_{t_1})} = b|_{\phi_{t_1}}$ and $b'_1|_{Y_{t_1}} = a'$. Also there is a $b'_2 \in \phi'_{t_2}(\mathcal{A})$ with $b'_2|_{\mathsf{free}(\phi_{t_2})} = b|_{\phi_{t_2}}$ and $b'_2|_{Y_{t_2}} = a'$. Then b'_1 and b'_2 coincide on $\mathsf{var}(\lambda_t)$ and thus by the connectivity condition $b'_1 \sim b'_2$. Thus, $b' := b'_1 \bowtie b'_2 \in \phi'_t(\mathcal{A})$, $b'|_{\mathsf{free}}(\phi_t) = b$ and $b'|_{Y_t} = a'$. But $a' \notin I$ and thus $b \notin N(t, a, I)$ which contradicts the assumption. Thus $I_1 \cap I_2 = I$ which completes the first direction.

Now consider sets $I_1, I_2 \subseteq \pi_{Y_t}(\lambda_t(\mathcal{A}))$ with $I_1 \cap I_2 = I$. Consider $b_1 \in N(t_1, a, I_1)$ and $b_2 \in N(t_1, a, I_2)$. Then we get $b_1 \sim b_2$ again by the connectivity condition. It follows that $b := b_1 \bowtie b_2$ is well-defined. We will show that $b \in N(t, a, I)$ which completes the proof of the claim.

First we observe that $b \in \phi_t(\mathcal{A})$. This is because $b_1 \in \phi_{t_1}(\mathcal{A})$ and $b_2 \in \phi_{t_2}(\mathcal{A})$ and thus there are $b'_1 \in \phi'_{t_1}(\mathcal{A})$ and $b'_2 \in \phi_{t_2}(\mathcal{A})$ with $b_1 \sim b'_1$ and $b_2 \sim b'_2$ and $b'_1 \sim a'$ and $b'_2 \sim a'$ for an $a' \in I$. Again by the connectivity condition we get $b'_1 \sim b'_2$, so they can be combined to $b' := b'_1 \bowtie b'_2 \in \phi_{t'}(\mathcal{A})$. But $b'|_{\mathsf{free}(\phi_t)} = b$ and thus $b \in \phi_t(\mathcal{A})$ as desired. With the same argument we also get that for each $a' \in I$ there is a $b' \in \phi'_t(\mathcal{A})$ with $b'|_{\mathsf{free}(\phi_t)} = b$.

It only remains to show that for every $b' \in \phi'_t(\mathcal{A})$ with $b'|_{\mathsf{free}(\phi_t)} = b$ we have that $b|_{Y_t} \in I$. So consider $b' \in \phi'_t(\mathcal{A})$. We have that $b|_{\mathsf{free}(\phi_{t_1})} \in N(t_1, a, I_1)$ and $b'|_{\mathsf{free}(\phi_{t_1})} = b|_{\mathsf{free}(\phi_{t_1})}$ and thus $b'|_{Y_t} \in I_1$. Similarly, $b'|_{Y_t} \in I_2$ and it follows $b'|_{Y_t} \in I_1 \cap I_2 = I$.

This proves Claim 10.5.10 ■

Proof of Claim 10.5.11. Proof by induction on $\mathcal{T}_t$. Let t first be a leaf, then by (9) we need at most $|c(t)| - 1$ multiplications and no additions to compute $f_{t,a,I}$. Doing the multiplications in a balanced fashion, leads to a circuit of depth $\log(|c(t)|)$.

Let now t be a vertex with two children t_1 and t_2. By induction $f_{t_1,a,I}$ and $f_{t_2,a,I}$ can be computed in depth $(2\ell+2)d_{t_1} + \log(|c(t_1)|)$ and $(2\ell+2)d_{t_2} + \log(|c(t_2)|)$, respectively. Since the computation of $f_{t,a,I}$ in (11) needs only at most $2 \cdot (2^\ell)^2 \leq 2^{2\ell+2} + |c_0(t)|$ addititions and multiplication we get that it can be computed in depth $\max((2\ell+2)d_{t_1} + \log(|c(t_1)|), (2\ell+2)d_{t_2} + \log(|c(t_2)|)) + 2\ell + 2 + \log(|c_0(t)|) \leq (2\ell+2)d_t + \log(|c(t)|)$.

For the case with one child observe that in (10) the first sum is over at most ℓ tuples a'. The second sum is over at most 2^ℓ subsets of $I_{a'}$. Since the $f_{t',a',I'}$ can by induction be computed in depth $(2\ell+2)d_{t'} + \log(|c(t')|)$ we get that $\frac{1}{\prod_{x\in c_0(t)} X_{a(x)}} f_{t,a,I}$ can be computed in depth $\log(|\ell| + 2^{|\ell|}) + (2\ell+2)(d_t - 1) + \log(|c(t')|)$. To compute $f_{t,a,I}$ we only have to multiply with $\prod_{x\in c_0(t)} X_{a(x)}$ and thus $f_{t,a,I}$ can be computed in depth

$$\begin{aligned} & 1 + \max(\log(|c_0(t)|), \\ & \qquad \log(|\ell| + 2^{|\ell|}) + (2\ell+2)(d_t - 1) + \log(|c(t')|)) \\ \leq\ & \log(\ell + 2^\ell) + (2\ell+2)(d_t - 1) + 1 + \log(|c(t)|) \\ =\ & (2\ell+2)d_t + \log(|c(t)|). \end{aligned}$$

This proves Claim 10.5.11. ■

This completes the proof of Proposition 10.5.8 ■

Now Theorem 10.2.3 follows directly from Proposition 10.5.8 and Lemma 10.4.1.

Since Boolean CQ-instances of bounded treewidth have bounded arity, they are always relation bounded. Thus we get the following Corollary that gives characterizations of VP_e.

Corollary 10.5.12. *a) Let (Φ_n) be a p-bounded family of Boolean CQ-instances of bounded treewidth. Then $P(\Phi) \in \mathsf{VP}_e$. Moreover, any family in VP_e is a p-projection of such $(P(\Phi_n))$.*

b) Let (Φ_n) be a p-bounded family of Boolean CQ-instances of bounded pathwidth. Then $P(\Phi) \in \mathsf{VP}_e$. Moreover, any family in VP_e is a p-projection of such $(P(\Phi_n))$.

11

GRAPH POLYNOMIALS ON BOUNDED TREEWIDTH GRAPHS

11.1 INTRODUCTION

In this chapter we explore a problem posed by Lyaudet in his PhD-thesis [Lya07]: Consider a graph property and the graph polynomial it defines as its generating function. If we restrict the graphs to be of bounded treewidth, then computing the considered graph polynomial is easy for most well known properties: If the property is definable in monadic second order logic then the graph polynomials can be computed efficiently [CMR01] (see Section 11.2.3 for more details).

Now assume that on general graphs the generating function of the considered graph property is VNP-complete. Does it follow then, that it is expressive enough to compute all families in VP_e on bounded treewidth graphs? Flarup, Koiran and Lyaudet showed that this is true for the permanent and the hamiltonian [FKL07]: Both polynomials give a characterization of VP_e when considered on bounded treewidth graphs. Lyaudet [Lya07] then conjectured that this might be extended to hold for all VNP-complete generating functions defined by monadic second order formulas.

In this chapter we settle this conjecture negatively: We show that there are some VNP-complete generating functions that, on graphs of bounded treewidth, are not expressive enough to capture VP_e. On the other hand we show results in the style of Flarup et al. hold for many known VNP-complete polynomial families.

11.2 MONADIC SECOND ORDER LOGIC, GENERATING FUNCTIONS AND UNIVERSALITY

11.2.1 *Monadic second order logic on graphs*

In this section we will give a very short introduction into monadic second order logic. For more background see e.g. [FG06, Lib04, Cou97]. We assume the reader to be to be familiar with the basics of second order logic which is the extension of first-order logic by relation variables and quantification over those. As it is common, we will write relation variables in uppercase letters, e.g. $P, Q, \ldots$, while the variables for domain elements are written in lowercase letters, e.g. $x, y, z, \ldots$.

A second order logic formula is monadic if it contains only unary relation variables. In this case quantification over relation variables is simply quantification over subsets of the domain. In the following we

will only consider formulas ϕ with one free relation variable X which we denote by $\phi(X)$.

So let $\phi(X)$ be a monadic second order formula with free relation variable X. Let $\mathcal{A}$ be a finite structure over the same vocabulary as ϕ with domain A. A solution of ϕ in the structure $\mathcal{A}$ is a subset $S \subseteq A$ such that $\mathcal{A}$ is a model of $\phi(S)$, the formula we get from $\phi(X)$ by substituting the free variable X by the set S. The *solution set* is defined as

$$\phi(\mathcal{A}) := \{S \subseteq A \mid \mathcal{A} \models \phi(S)\}.$$

Observe that in contrast to the query results considered before solution sets $\phi(\mathcal{A})$ are sets of subsets of A and *not* sets of tuples of elements of A.

We restrict our attention in this chapter to directed and undirected graphs which we encode as finite structures. There are two different ways to encode a directed graph $G = (V, E)$ as a structure $\mathcal{A}$. The first one is encoding G as a structure $\mathcal{A}$ over the vocabulary $\tau_1 = \{\mathcal{E}\}$, where the domain A of $\mathcal{A}$ is V and $\mathcal{E}^{\mathcal{A}} \subseteq V^2$. If G is undirected, then the edge relation $\mathcal{E}$ of G is symmetric. We denote the set of monadic second order formulas over τ_1 by MS_1.

Example 11.2.1. The formula

$$graph_1 := (\forall x \forall y\, (\mathcal{E}xy \to \mathcal{E}yz)) \wedge (\forall x \neg \mathcal{E}xx)$$

is true for a τ_1-structure $\mathcal{A}$ if and only if $\mathcal{A}$ is an encoding of an undirected graph as discussed above. □

In the remainder of this chapter, whenever we assume that a τ_1-structure encodes a graph, we could enforce this by considering instead of a formula ϕ always $\phi \wedge graph_1$. We will not do this here to in order not to clutter the notation unnecessarily. It will always be clear from the context if we talk about undirected or directed graphs.

Example 11.2.2. Consider the MS_1-formula

$$\phi_{clique1}(X) := \forall x \forall y\, ((x \in X \wedge y \in X) \to (\mathcal{E}xy \vee x = y))\,.$$

Given an encoding $\mathcal{A}$ of a graph G, the solution set $\phi(X)$ contains exactly the vertex sets of the cliques in G. □

The second encoding of a graph $G = (V, E)$ is by its incidence structure, a structure over the vocabulary $\tau_2 = \{\mathcal{V}, \mathcal{E}, \mathcal{I}\}$. Here the domain of $\mathcal{A}$ is $A = V \cup E$. Furthermore, $\mathcal{V}^{\mathcal{A}} := V$, $\mathcal{E}^{\mathcal{A}} := E$ and $\mathcal{I}^{\mathcal{A}} := \{(v, e) \mid v \in V, e \in E, v \text{ is a vertex of } e\}$.

For directed graphs the incidence structure $\mathcal{A}$ is over the vocabulary $\tau_{2,d} := \{\mathcal{V}, \mathcal{E}, \mathcal{I}, \mathcal{O}\}$ where again the domain of $\mathcal{A}$ is $A = V \cup E$, $\mathcal{V}^{\mathcal{A}} := V$ and $\mathcal{E}^{\mathcal{A}} := E$. Furthermore,

$$\mathcal{I}^{\mathcal{A}} := \{(v, e) \mid v \in V, e \in E, v \text{ is the target of } e\}$$

and

$$\mathcal{O}^{\mathcal{A}} := \{(v, e) \mid v \in V, e \in E, v \text{ is the source of } e\}.$$

Example 11.2.3. Again there is a formula that checks if a τ_2-structure encodes a graph, this time given by

$$\begin{aligned} graph_2 :=& \forall x\,((\mathcal{V}x \wedge \neg\mathcal{E}x) \vee (\neg\mathcal{V}x \wedge \mathcal{E}x)) \\ & \wedge \forall x\,((\exists y \mathcal{I}xy \to Vx) \wedge (\exists y \mathcal{I}yx \to \mathcal{E}x)) \\ & \wedge \forall x(\mathcal{E}x \to (\exists y \exists y' \forall z(y \neq y' \wedge \mathcal{I}yx \wedge \mathcal{I}y'x \wedge \\ & \qquad (\mathcal{I}zx \to y = z \vee y' = z)))). \end{aligned}$$

It is easy but somewhat tedious to show that there is a similar formula that checks if a $\tau_{2,d}$-structure is an encoding of a directed graph. □

We will assume that all τ_2- and $\tau_{2,d}$-structures are encodings of graphs. Furthermore, we will always assume that the solution set of every τ_2-formula contains either vertex sets or edge sets but never both. This can again be enforced by formulas easily. We will call the free variable a *vertex variable* or an *edge variable*, respectively. We denote the set of monadic second order formulas over τ_2 and $\tau_{2,d}$ by MS_2. When it is not important or clear from the context if we talk about MS_1- or MS_2-formulas, we sometimes drop the index and simply talk about MS-formulas.

Example 11.2.4. The edge set of cliques of a graph can be expressed as the solution set of an MS_2-formula as follows:

$$\begin{aligned} \phi_{clique2}(X) :=& \exists Y(\underbrace{(\forall y(Yy \leftrightarrow (\exists x(Xx \wedge \mathcal{I}yx)))}_{\substack{Y \text{ is the set of} \\ \text{end vertices of the} \\ \text{edges in } X}} \\ & \wedge \underbrace{(\forall y \forall y'((Yy \wedge Yy') \leftrightarrow (\exists x(\mathcal{I}xy \wedge \mathcal{I}xy'))))}_{\substack{X \text{ connects exactly} \\ \text{the vertices in } Y}}). \end{aligned}$$ □

From an algorithmic perspective it makes of course no difference if a graph is encoded by a τ_1- or a τ_2-structure since both representations can easily be transformed into each other. From a logical perspective there is a difference though. The possibility of MS_2-formulas to quantify over edge sets makes MS_2 strictly more expressive than MS_1.

Monadic second order logic on graphs is interesting because many computational problems can be expressed by it and because of the following theorem by Courcelle [Cou90] that is widely considered as one of the cornerstones of parameterized complexity theory.

Theorem 11.2.5 (Courcelle's Theorem). *For every monadic second order formula $\phi(X)$ there is a computable function f and an algorithm that on inputs $G = (V, E)$ decides in time $f(\mathrm{tw}(G))|V|$ if $\phi(\mathcal{A})$ is empty or not where $\mathcal{A}$ is an encoding of G.*

For a proof and several extensions of Courcelle's theorem see e.g. [Kre11, FG06].

11.2.2 *Generating functions*

We consider generating functions of monadic second order formulas on graphs. Since the free variables in the formulas may be either vertex or edge variables, we have to consider two different cases.

Definition 11.2.6. *The* generating function $GF(G, \phi)$ *of a graph G with edge weight function w and a monadic second order formula ϕ with free edge variable is defined as*

$$GF(G, \phi) := \sum_{E' \in \phi(\mathcal{A})} w(E'),$$

where $\mathcal{A}$ is the encoding of G and $w(E') = \prod_{e \in E'} w(e)$.

The generating function $GF(G, \phi)$ *of a graph G with a vertex weight function w and a monadic second order formula ϕ with a free vertex variable is defined as*

$$GF(G, \phi) := \sum_{V' \in \phi(\mathcal{A})} w(V'),$$

where $\mathcal{A}$ is the encoding of G and $w(V') = \prod_{v \in V'} w(v)$. □

Example 11.2.7. Let $\phi_{clique1}$ be the formula from Example 11.2.2. Then $GF(G, \phi_{clique1})$ sums over the weights of the vertices of all cliques of G.

Let $\phi_{clique2}$ be the formula from Example 11.2.4. Then $GF(G, \phi_{clique2})$ sums over the weights of the edges of all cliques of G. □

A *weighted graph* is defined to be a graph G with either an edge weight function w or a vertex weight function w. It will always be clear from the context if a weighted graph has edge or vertex weights. The size of a weighted graph is the size of the underlying graph. Since we want to consider families of polynomials computed by generating functions, we will consider families of weighted graphs. A family (G_n) of weighted graphs is called p-bounded if there is a polynomial p such that the size of G_n is at most $p(n)$.

Remark 11.2.8. Bürgisser in [Büoo, Chapter 3] considers generating functions of graph properties similar to the generating functions we defined above. A *graph property* is a class of graphs closed under isomorphisms. Given an edge weighted graph $G = (V, E)$ and a graph property $\mathcal{E}$, Bürgisser defines the generating function as

$$GF(G, \mathcal{E}) := \sum_{E' \subseteq E} w(E'),$$

where the sum is over all subsets E' of E such that the spanning subgraph $(V, E') \in \mathcal{E}$.

Clearly, for every MS-formula ϕ and every encoding $\mathcal{A}$ of a graph the solution set $\phi(\mathcal{A})$ is closed under isomorphisms and thus $\phi(\mathcal{A})$ is a graph property in Bürgisser's sense. This shows that our generating functions and that of Bürgisser are essentially the same. The key difference is that we restrict our graph properties to be definable by MS-formulas while for Bürgisser they can be arbitrary. Furthermore, to capture graph polynomials like the independent set polynomial or the dominating set polynomial we also consider generating properties defined by vertex subsets of G. □

Restricting our considerations to generating function defined by monadic second order formulas has the advantage that we get a very general upper bound on the complexity of these generating function on graphs of bounded treewidth. Courcelle, Makowski and Rotics [CMR01] gave the following version of Courcelle's theorem for the arithmetic circuit model.

Theorem 11.2.9. *Let ϕ be a monadic second order formula on graphs and let (G_n) be a p-bounded family of weighted graphs of bounded treewidth. Then the polynomial family $(GF(G_n, \phi))$ is in VP_e.*

Note that Courcelle, Makowski and Rotics in [CMR01] do not state Theorem 11.2.9 for VP_e. Instead they only state containment in VP. However, their methods can be seen to show containment in VP_e instead of VP rather easily [Dur13]. To not rely on an unpublished claim that we do not verify here, we will show all upper bounds in this chapter directly with the techniques of Chapter 10, even though we could get them directly from Theorem 11.2.9.

To an $(n \times n)$-matrix $M = (a_{ij})_{i,j \in [n]}$ we assign the directed graph $G_M = (V, E)$ by $V := [n]$ and $E := \{ij \mid i, j \in [n], a_{ij} \neq 0\}$. We give G_M the edge weights $w(ij) := a_{ij}$.

Example 11.2.10. Let $\mathcal{CC}$ be the class of directed graphs that consist of one or more directed cycles, where we also count loops as cycles. Then it is well-known that

$$\mathrm{PER}_n(M) = GF(G_M, \mathcal{CC}).$$

The corresponding MS_2-formula is

$$\begin{aligned}\phi_{CC}(X) :=& \forall x(\mathcal{V}x \to (\exists y \exists z(\mathcal{I}xy \wedge \mathcal{O}xz \wedge Xy \wedge Xz))) \wedge \\ & \forall x \forall y((Xx \wedge Xy \wedge \exists z\,((\mathcal{I}zx \wedge \mathcal{I}zy) \vee (\mathcal{O}zx \wedge \mathcal{O}zy)) \\ & \qquad \to x = y)).\end{aligned}$$

□

11.2.3 *Treewidth preserving reductions and universality*

One of our aims in this chapter is to show that VP_e can be characterized by generating functions for bounded treewidth graphs. To this end, we introduce the following universality notion:

Definition 11.2.11. *Let ϕ be a monadic second order formula. We say that ϕ is* VP_e-universal for bounded treewidth *if*

- *for every p-bounded family (G_n) weighted graphs of bounded treewidth we have $(GF(G_n, \phi)) \in \mathsf{VP}_e$, and*
- *for every family $(f_n) \in \mathsf{VP}_e$ there is a p-bounded family (G_n) of weighted graphs of bounded treewidth with $(f_n) \leq_p GF(G_n, \phi)$.* □

In a slight abuse of notation we sometimes call the corresponding graph polynomial $GF(., \phi)$ VP_e-universal.

Note that with Theorem 11.2.9 we could drop the first item in Definition 11.2.11. However, since we do not want to rely on Theorem 11.2.9 as discussed above, we keep it in the definition.

Let ϕ_{CC} be the formula of Example 11.2.10. Then we can formulate one of the main results proved by Flarup et al. [FKL07] as follows:

Theorem 11.2.12. *[FKL07] ϕ_{CC} is VP_e-universal.*

With these definitions we can now reformulate a conjecture by Lyaudet.

Conjecture 2 ([Lya07]). *Let ϕ be a monadic second order formula such that there is a family (G_n) of polynomial size graphs such that $(GF(G_n, \phi))$ is* VNP*-complete. Then ϕ is VP_e-universal on bounded treewidth graphs.*

To show universality results we use the usual technique of reductions. To do so we introduce a new kind of reduction between graph properties tailored exactly for our needs. Remember that for two polynomials f, g we denote by $f \leq g$ that f is a projection of g (see Section 9.2).

Definition 11.2.13. *Let ϕ and ψ be two monadic second order formulas. We say that there is a* treewidth preserving reduction *from $GF(., \phi)$ to $GF(., \psi)$ (in symbols: $GF(., \phi) \leq_{\mathcal{BW}} GF(., \psi)$) if there is a function f and a polynomial p such that for every weighted graph G there is a weighted graph G' such that $GF(G, \phi) \leq GF(G', \psi)$ and $|G'| \leq p(|G|)$ and $\mathrm{tw}(G') \leq f(\mathrm{tw}(G'))$.* □

The treewidth of an undirected graph is defined to be that of the underlying directed graph. The treewidth of a matrix is that of the associated weighted directed graph.

If we have $GF(., \phi) \leq_{\mathcal{BW}} GF(., \psi)$ then every family (f_n) of polynomials defined as a generating function of ϕ can be computed as

generating function of ψ. Furthermore, if the treewidth of the family of graphs (G_n) that allows computing (f_n) as the generating function of ψ is bounded, then the treewidth remains bounded for the computation with ϕ.

We will use treewidth preserving reductions to prove that specific generating functions on bounded treewidth graphs are expressive enough to capture VP_e. This allows us to give more graph polynomials that restricted to graphs of bounded treewidth characterize the class VP_e.

From Definition 11.2.11 and Definition 11.2.13 the following proposition is immediate.

Lemma 11.2.14. *Let ϕ be a monadic second order formula. The following statements are equivalent:*

1. *ϕ is VP_e-universal for bounded treewidth.*
2. *For every family (G_n) of weighted polynomial size graphs of bounded treewidth we have $(GF(G_n, \phi)) \in \mathsf{VP}_e$ and there is a monadic second order formula ψ that is VP_e-universal for bounded treewidth such that $GF(., \psi) \leq_{BW} GF(., \phi)$.*
3. *There are two monadic second order formulas ψ and ψ' which are VP_e-universal for bounded treewidth with $GF(., \psi) \leq_{BW} GF(., \phi) \leq_{BW} GF(., \psi')$.*

With Proposition 11.2.14 and Theorem 11.2.12 we can easily show universality for new graph properties by reductions without having to resort to direct simulations of formulas.

11.3 CLIQUES ARE NOT UNIVERSAL

Before showing more universality results, we prove that Lyaudet's conjecture is false. We will see that the formulas $\phi_{clique1}$ and $\phi_{clique2}$ from the Examples 11.2.2 and 11.2.4 are counterexamples.

Proposition 11.3.1. *$\phi_{clique1}$ and $\phi_{clique2}$ are not VP_e-universal for bounded treewidth.*

Proof. Let (G_n) be a family of bounded treewidth graphs. We have by Lemma 2.3.4 the size of cliques in any graph G_n is bounded by $\mathrm{tw}(G_n) + 1$ and hence by a constant. It follows that the of degree of $GF(G_n, \phi_{clique1})$ and $GF(G_n, \phi_{clique2})$ are bounded by a constant. p-projections can only decrease the degree of polynomials. Hence, the family (f_n) of polynomials with $f_n = X^n$, which certainly is in VP_e, is neither a p-projection of $(GF(G_n, \phi_{clique1}))$ nor of $(GF(G_n, \phi_{clique2}))$. ■

Corollary 11.3.2. *Conjecture 2 is false.*

Proof. For MS_2-formulas this follows directly from Proposition 11.3.1 and the VNP-completeness of $GF(K_n, \phi_{clique2})$ (see [Büoo]).

For the less expressive MS_1-formulas Conjecture 2 remains false. This follows from the non-universality of $\phi_{clique1}$ and the fact that there is a family (G_n) of polynomial size graphs such that the family $(GF(G_n, \phi_{clique2}))$ is VNP-complete. The latter follows easily from the VNP-completeness of the independent set polynomial (see [BK09]) by considering complement graphs. ■

11.4 vp_e-UNIVERSALITY FOR BOUNDED TREEWIDTH

In this section we show that while Lyaudet's conjecture is false in general, many well-known graph polynomials *are* VP_e-universal for bounded treewidth.

11.4.1 *Formulation of the results and outline*

In this subsection we introduce several graph polynomials and state a theorem that says that they are all VP_e-universal for bounded treewidth. Furthermore, we sketch an outline of the proof that we give in the next sections.

A *partial cycle cover* of a directed graph is an edge set such that every vertex has at most one incoming and one outgoing edge. Partial cycle covers are the solutions of the monadic second order formula

$$\phi_{PCC}(X) := \forall x \forall y((Xx \wedge Xy \wedge \exists z\,((\mathcal{I}zx \wedge \mathcal{I}zy) \vee (\mathcal{O}zx \wedge \mathcal{O}zy)) \rightarrow x = y)).$$

We assign to a matrix M a graph G_M as before and define the partial permanent $\mathrm{PER}^*(M)$ by

$$\mathrm{PER}^*(M) := GF(G_M, \phi_{PCC}).$$

Consider the MS_1-formula

$$\phi_{IS}(X) := \forall x \forall y((x \in X \wedge y \in X) \rightarrow \neq \mathcal{E}xy).$$

Let G be a graph with structure $\mathcal{A}$. Then it is easy to see that $\phi_{IS}(\mathcal{A})$ contains exactly the independent sets of G.

Remember that a vertex cover of G is a subset of V that contains at least one vertex of each edge in E. Let again G be a graph with structure $\mathcal{A}$. Then we have that $\phi_{VC}(\mathcal{A})$, where ϕ_{VC} is the MS_1-formulas

$$\phi_{VC}(X) := \forall y \forall y'(\mathcal{E}yy' \rightarrow (Xy \vee Xy')),$$

contains all vertex covers of G.

A dominating set of G is defined to be a vertex set D such that for every vertex $v \in V$ we have that v or a neighbor u of v is in D. It is easy to see that $\phi_{DS}(\mathcal{A})$, where

$$\phi_{DS}(X) := \forall x(Xx \vee \exists y(Xy \wedge \mathcal{E}xy)),$$

contains the contains the dominating sets of G.

Theorem 11.4.1. *ϕ_{PCC}, ϕ_{IS}, ϕ_{VC} and ϕ_{DS} are VP_e-universal for bounded treewidth.*

We will show the lower and upper bounds individually in the following subsections. We will start proving $\phi_{CC} \leq_{\mathcal{BW}} \phi_{BCC}$ which shows the lower bound for ϕ_{PCC}.

When trying to show lower bounds for the other properties, we face a small complication: We would like to use a known reduction from ϕ_{BCC} to ϕ_{IS} from [BK09]. This reduction constructs to a graph G the line graph $L(G)$. Unfortunately, in general, we cannot bound the treewidth of $L(G)$ by the treewidth of G. For example, the line-graph of a star is a clique, which by Lemma 2.3.4 has high treewidth. Thus we cannot use the reduction from [BK09] directly. We solve this small complication by bounding not only the treewidth but also the degree of the graph G. Fortunately, for this restriction ϕ_{CC} and ϕ_{PCC} are still expressive enough to capture all of VP_e. Now we can use the reduction from [BK09] to show the lower bound for ϕ_{IS}.

Lower bounds for ϕ_{VC} and ϕ_{DS} are then shown by the reductions $\phi_{IS} \leq_{\mathcal{BW}} \phi_{VC}$ and $\phi_{VC} \leq_{\mathcal{BW}} \phi_{DS}$.

Finally, we show the upper bounds which we only have to show for ϕ_{PCC} and ϕ_{DS} because ϕ_{IS} and ϕ_{VC} reduce to ϕ_{DS}.

11.4.2 *Reduction: $\phi_{CC} \leq_{\mathcal{BW}} \phi_{PCC}$*

Lemma 11.4.2. $\phi_{CC} \leq_{\mathcal{BW}} \phi_{PCC}$

Proof. The construction is an adaption of Jerrum's VNP-completeness proof for PER_n^* [Jer81] (see also [Büoo, Chapter 3]).

Let $G = (V, E)$ be an edge weighted, directed graph on the vertex set $[n]$. For each $ij \in E$ let X_{ij} be the weight of the edge ij. For $i \in V$ we add two new vertices i_{in} and i_{out} and the edges $i_{in}i$ and ii_{out} both with weight -1. Call the resulting directed graph G'.

Let $\pi \subseteq E$ be a partial cycle cover of G, i.e., an edge set E such the every vertex has at most one edge in E entering it and at most one leaving it. We define $w(\pi) := \prod_{ij \in \pi} X_{ij}$. Let $I(\pi)$ be the set of vertices that have no entering edge in π and $J(\pi)$ the set having no outgoing edge in π. We construct partial cycle covers of G' by choosing sets $M \subseteq I(\pi)$ and $N \subseteq J(\pi)$ and adding the edges $i_{in}i$ for $i \in M$ and ii_{out} for $i \in N$. Oberserve that the resulting partial cycle cover has the weight $(-1)^{|M|+|N|}w(\pi)$. Also every partial cycle cover of G' can be uniquely constructed by selecting a partial cycle cover of G and then selecting M and N. Thus we get

$$\mathrm{PER}^*(G') = \sum_{\pi : \pi \text{ partial cover of G}} \left(w(\pi) \sum_{M \subseteq I(\pi)} (-1)^{|M|} \sum_{N \subseteq J(\pi)} (-1)^{|N|} \right).$$

If $I(\pi)$ is not empty, then the term $\sum_{M \subseteq I(\pi)} (-1)^{|M|}$ is 0. The analogous statement is true for $J(\pi)$. Thus only partial cycle covers in which every vertex has an incoming and an outcoming edge contribute to the sum. But these are exactly the cycle covers of G and thus $\mathrm{PER}(G) = \mathrm{PER}^*(G')$.

Let $\mathcal{T}$ be a tree-decomposition of G. We construct a tree decomposition of G' by simply adding i_{in} and i_{out} to all the bags containing i in $\mathcal{T}$. It is easy to see, that the result is indeed a valid tree-decomposition and $\mathrm{tw}(G') \leq 3\mathrm{tw}(G)$. This means the construction is a treewidth preserving reduction. ■

11.4.3 ϕ_{CC} and ϕ_{PCC} on bounded degree graphs

Lemma 11.4.3. *For every family $(f_n) \in \mathsf{VP}_e$ there is a p-bounded family (G_n) of weighted graphs of bounded treewidth and bounded degree such that $(f_n) \leq_p GF(G_n, \phi_{CC})$.*

Proof (sketch). The proof is a minimal modification of the two-stage construction of Flarup et al. [FKL07] which follows Valiant's proof of the universality of the permanent [Val79] (see also [Büo0, Chapter 2]). In the first stage we simulate arithmetic formulas by arithmetic branching programs. Flarup et al. observed that the underlying graphs of the branching programs are so-called *series parallel* graphs which have treewidth at most 2. We modify the case $\phi_1 + \phi_2$: instead of identifying the sources s_1 and s_2 and the sinks t_1 and t_2 we add two new vertices s' and t' as new source and sink and add the edges $s's_1$, $s's_2$, t_1t' and t_2t' each with the weight 1. It is easy to see that this new graph computes the same polynomial, has size $O(|\phi|)$ and treewidth 2 and each of its vertices has in- and out-degree at most 2.

In the second step of the construction in [FKL07] the constructed ABP is reduced to the permanent. We add only self-loops and an edge from the sink of the ABP to its source. Thus we increases the in- and out-degree by at most 1 and the treewidth by at most 1. Consequently, every formula can be computed as the permanent of a degree 6 graph with treewidth at most 3. ■

Corollary 11.4.4. *For every family $(f_n) \in \mathsf{VP}_e$ there is a p-bounded family (G_n) of weighted graphs of bounded treewidth and bounded degree such that $(f_n) \leq_p GF(G_n, \phi_{PCC})$.*

Proof. The construction of Lemma 11.4.2 increases the degree only by 2. ■

11.4.4 The lower bound for ϕ_{IS}

Lemma 11.4.5. *For every family $(f_n) \in \mathsf{VP}_e$ there is a p-bounded family (G_n) of weighted graphs of bounded treewidth and bounded degree such that $(f_n) \leq_p GF(G_n, \phi_{IS})$.*

Proof. We show a treewidth preserving reduction from the family $GF(G_n, \phi_{PCC})$ of Corollary 11.4.4. This reduction is a slight modification of the VNP-hardness proof of $GF(., \phi_{IS})$ in [BK09] in which we also consider the treewidth. Let G be a directed graph with vertex set $[n]$, edge in $E \subseteq [n]^2$ and edge weight function w. We construct a graph $G' = (V', E')$ such that $\mathrm{PER}^*(G) = GF(G', \phi_{IS})$. We set $V' := E$. Two vertices $ij, i'j' \in V'$ are connected by an edge if and only if $i = i'$ or $j = j'$. We give G' the vertex weight w' with $w'(ij) := w(ij)$.

Let $\pi \subseteq E$ be a partial cycle cover of G. We claim that π is an independent set in G'. Indeed, assume that it is not, i.e., there are vertices ij and $i'j'$ that are adjacent. Then $i = i'$ or $j = j'$ which which contradicts π being a partial cycle cover.

Let now π be an independent set of G'. Then it is easy to see that π is an independent set of G.

Thus, partial covers of G and independent sets in G' coincide and, due to the definition of weights, $\mathrm{PER}^*(G) = GF(G', \phi_{IS})$.

For the bound on the treewidth of G' consider a tree decomposition $(\mathcal{T}, (\chi_t)_{t \in T})$ of G of width $k - 1$ and assume that G has maximum degree d. We construct a tree decomposition $(\mathcal{T}, (\chi'_t)_{t \in T})$ of G' with the same underlying tree $\mathcal{T}$. We define $\chi'_t := \{ij \in V' \mid i \in \chi_t \vee j \in \chi_t\}$.

We claim that the result is indeed a tree decomposition. Clearly, each $e \in V'$ is in at least one bag. Also two vertices $ij, i'j' \in V'$ are connected in G' if and only if $i = i'$ or $j = j'$. Thus ij and $i'j'$ share a bag χ'_t, if they are connected in G'. Also the bags containing a vertex $ij \in V'$ are connected, because the set of bags containing i and that containing j in the tree decomposition of G are connected and intersecting. Thus $(\mathcal{T}, (\chi'_t)_{t \in T})$ is a tree decomposition. It has width at most $dk - 1$, because for each bag χ_t and each $i \in \chi_t$ the bag χ'_t contains at most d vertices $ij \in V'$. Thus the construction is treewidth preserving if d is constant and with Corollary 11.4.4 the claim follows. ■

11.4.5 *Reduction: $\phi_{IS} \leq_{\mathcal{BW}} \phi_{VC}$*

Lemma 11.4.6. $\phi_{IS} \leq_{\mathcal{BW}} \phi_{VC}$

Proof. The reduction is taken from [BK09, p. 8]. We will show that it is treewidth preserving. Remember that the incidence graph $G' = (V', E')$ of a graph $G = (V, E)$ has the vertex set $V' := V \cup E$ and the edge set

$$E' := \{ve \mid v \in V, e \in E, v \text{ is a vertex of } e\}.$$

Let w be the vertex weight function of G, then we define the vertex weight function of G' as $w'(v) := w(v)$ for $v \in V$ and $w'(e) = -1$ for $e \in E$. Briquel and Koiran showed that

$$GF(G, \phi_{IS}) = (-1)^{|E'|} GF(G, \phi_{VC}). \tag{12}$$

We claim that G' always has an even number of edges. Indeed, every vertex $e \in E$ of G' has exactly two neighbors which are the end vertices of e in G. Thus $|E'| = 2|E|$ and (12) simplifies to

$$GF(G, \phi_{IS}) = GF(G, \phi_{VC})$$

which this is the desired reduction.

It remains to show that $\mathrm{tw}(G')$ is bounded in $\mathrm{tw}(G)$. To this end, let $(\mathcal{T}, (\chi_t)_{t\in T})$ be a tree decompositions of G of width $k-1$. We construct a tree decomposition $(\mathcal{T}, (\chi'_t)_{t\in T})$ by adding each $e \in E$ to any bag χ_t that contains both vertices of e. It is easy to see that the result is indeed a tree decomposition. To bound its width observe that to each bag χ_t at most $\binom{k}{2}$ vertices may have been added. It follows that $\mathrm{tw}(G') \leq \mathrm{tw}(G) + \binom{\mathrm{tw}(G)+1}{2}$. ■

11.4.6 *Reduction:* $\phi_{VC} \leq_{\mathcal{BW}} \phi_{DS}$

Lemma 11.4.7. $\phi_{VC} \leq_{\mathcal{BW}} \phi_{DS}$

Proof. To a graph $G = (V, E)$ we construct $G' = (V', E')$ where $V' := V \cup E$ and

$$E' := E \cup \{ve \mid v \in V, e \in E, v \text{ is a vertex of } e\}.$$

We set $w'(v) := w(v)$ for $v \in V$ and $w'(e) := 0$ for $e \in E$. Every dominating set D of G' has to dominate all vertices $e \in E$. But if $e \in D$, then the corresponding product contributes 0. So one of the end points v or u has to be in the dominating set D, if we want it to contribute. It follows that the dominating sets with non-zero contribution to $GF(G', \phi_{DS})$ are exactly the vertex covers of G. This implies $GF(G, \mathcal{IS}) = GF(G, \mathcal{DS})$.

We have $\mathrm{tw}(G') = O(\mathrm{tw}(G)^2)$ with the same proof as Lemma for 11.4.6. ■

Since treewidth preserving reductions are special p-projections, we get the following corollary which we will use in Chapter 12.

Corollary 11.4.8. *There is a family (G_n) of polynomial size graphs such that $(GF(G_n, \phi_{DS}))$ is* VNP*-complete.*

Proof. Containment in VNP is clear with Lemma 9.2.3, because for a vertex set D of a graph G one can in polynomial time decide if D is dominating.

VNP-hardness follows from the VNP-hardness of $(GF(G_n, \phi_{VC}))$ for a family (G_n) constructed in [BK09] and the proof of Lemma 11.4.7. ■

11.4.7 *The upper bounds*

As discussed in Subsection 11.4.1, it suffices to show containment in VP_e only for $GF(\mathcal{PCC}, G_n)$ and $GF(\mathcal{DS}, G_n)$ for bounded treewidth graphs (G_n). We will use Theorem 10.2.3 to establish these results and avoid dynamic programming here.

Lemma 11.4.9. *$GF(\mathcal{DS}, G_n) \in \mathsf{VP}_e$ for every p-bounded family (G_n) of weighted graphs of bounded treewidth.*

Proof. We construct Boolean, quantifier free ACQ-instances (Φ_n) of bounded arity such that $GF(G_n, \phi_{DS}) \leq P(\Phi_n)$. Obviously, such instances are relation bounded and thus the Claim will follow with Theorem 10.2.3.

So let $G = (V, E)$ be a vertex weighted graph and let $(\mathcal{T}, (\chi_t)_{t \in T})$ be a tree decomposition of G of width $k-1$. We assume w.l.o.g. that $\mathcal{T}$ is a full binary tree, i.e., every $t \in T$ is either a leaf or it has exactly 2 children. This form can alway be achieved by making copies of bags.

We construct an ACQ-instance $\Phi = (\mathcal{A}, \phi)$. Remember that $\mathcal{T}_t$ is the subtree of $\mathcal{T}$ with t as its root. For each $t \in T$ and each vertex $v \in \chi_t$ the query ϕ has the variables v_d and v_b^t. The intuitive interpretation of these variables is the following: v_d takes the value 1 if v is in a dominating set and 0 otherwise. The variable v_b^t takes the value 0 if v is dominated by a vertex u appearing in any bag $\chi_{t'}$ such that t' lies in $\mathcal{T}_t$. Otherwise, v_b^t takes the value 1.

The query ϕ is the conjunction of the following atoms: For each leaf $t \in T$ there is an atom ϕ_t with the relation symbol $\mathcal{R}_t$ and the variables v_d and v_b^t for each $v \in \chi_t$. For each non-leaf $t \in T$ with children t_1, t_2 we have an atom ϕ_t with the relation symbol $\mathcal{R}_t$ that has for $v \in \chi_t$ the variables v_d and v_b^t, for every $v \in \chi_{t_1}$ the variable $v_b^{t_1}$ and for every $v \in \chi_{t_2}$ the variable $v_b^{t_2}$. This completes the construction of ϕ.

We now describe the relations of the $\mathcal{R}_t^{\mathcal{A}}$ by their satisfying assignments. If t is a leaf of $\mathcal{T}$, then the satisfying assignments in $\mathcal{R}_t^{\mathcal{A}}$ are those that assign arbitrary values to the v_d and assign 1 to v_b^t if and only if v_d is assigned 1 or there is a neighbor $u \in \chi_t$ of v such that u_d is assigned 1.

For a non-leaf $t \in T$ with children t_1 and t_2 the satisfying assignments are those that for each $v \in \chi_t$ assign 0 to v_b^t if and only if one of the following conditions is satisfied:

- v_d is assigned 1,
- there is a neighbor $u \in \chi_t$ of v such that u_d is assigned 1, or
- at least one child t_i such that the bag χ_{t_i} contains v and $v_b^{t_i}$ is assigned 0.

This completes the construction of $\Phi = (\mathcal{A}, \phi)$.

Let us now show that $GF(G, \phi_{DS}) \leq Q(\Phi)$. To a vertex set $D \subseteq V$ we assign a partial assignment $a_D : \{v_d \mid v \in V\} \to \{0,1\}$ that assigns 1 to v_d if and only if $v \in D$.

We state the following observation which can be proved be an easy reduction:

Observation 11.4.10. *For every D the assignment a_D can be uniquely extended to a satisfying assignment a'_D of Φ. This assignment a'_D assigns the value 0 to v_b^t if and only if $a_D(v_d) = 1$ or there is a vertex u appearing in a bag in $\mathcal{T}_t$ such that u is a neighbor of v in G and $a_D(u_d) = 1$.*

Let $r(v)$ be the $t \in T$ that is nearest to the root of $\mathcal{T}$ with $v \in \chi_t$.

Claim 11.4.11. *A set $D \subseteq V$ is a dominating set if and only if $a'_D(v_b^{r(v)}) = 0$ for every $v \in V$.*

Proof. Let first D be a dominating set of G. Consider a vertex $v \in V$. If $v \in D$, then clearly $a'_D(v_b^t) = 0$ for every $t \in T$ with $v \in \chi_t$ and thus in particular $a'_D(v_b^{r(v)}) = 0$. If $v \notin D$, then there is a neighbor u of v in G such that $u \in D$. But then u is in a bag in $\mathcal{T}_t$ by the properties of tree decompositions. By definition of a_D if follows that $a_D(u_d) = 1$ and we get $a'_D(v_b^{r(v)}) = 0$ by the characterization of a'_D from above. Thus a'_D has the desired property.

Let now $a'_D(v_b^{r(v)}) = 0$ for every $v \in V$. Then either $v \in D$ or there is a neighbor u of v with $u \in D$ by the characterization of a'_D above. It follows that v is dominated by D. ■

We directly get

$$\begin{aligned} P(\Phi) &= \sum_{D \subseteq V} \prod_{v \in V} X_{v_d}^{a_D(v_d)} \prod_{t \in T, v \in V} X_{v_b^t}^{a'_D(v_b^t)} \\ &= \sum_{D \subseteq V} \prod_{v \in D} X_{v_d} \prod_{t \in T, v \in V, a'_D(v_b^t)=1} X_{v_b^t}. \end{aligned} \tag{13}$$

Now substitute for each $v \in V$ the variable $X_{v_b^{r(v)}}$ by 0 and all other $X_{v_b^t}$ by 1. Furthermore substitute X_{v_d} by X_v, then (13) simplifies to

$$\sum_{D \subseteq V} \prod_{v \in D} X_v \prod_{v \in V, a'_D(v_b^{r(v)})=1} 0 = \sum_{D \text{ dominating set of } G} \prod_{v \in D} X_v = GF(G, \phi_{DS}).$$

Thus we have $GF(G, \phi_{DS}) \leq P(\Phi)$.

Observe that ϕ is acyclic: It is easy to see that $(\mathcal{T}, (\lambda_t)_{t \in T})$ where λ_t is the edge associated to the atom ϕ_t is a join tree. Furthermore, each atom has at most $6k$ Boolean variables. Thus the relations of $\mathcal{A}$ have at most size 2^{6k}.

Hence, for every family (G_n) of graphs of bounded treewidth we can construct a relation bounded and p-bounded family (Φ_n) of quantifier free, Boolean ACQ-instances with $(GF(G_n, \phi_{DS})) \leq_p (P(\Phi_n))$. As the latter family of polynomials is in VP_e by Theorem 10.2.3 the lemma follows. ■

Lemma 11.4.12. *$GF(\mathcal{PCC}, G_n) \in \mathsf{VP}_e$ for every p-bounded family (G_n) of weighted graphs of bounded treewidth.*

Proof (sketch). The idea is the same as in the proof of Lemma 11.4.9. Instead of a vertex set D we select an edge set E' by setting variables x_e to 0 or 1. Again we have an atom ϕ_t for every bag in a tree decomposition of a graph G. The variable x_e appears in the atoms that correspond to the bags that contain both end vertices of e. We only have to check that every vertex v has at most one incoming and one outgoing edge in E'. This can again be done by variables v_b^t as in the proof of Lemma 11.4.9. ■

11.5 CONCLUSION

We have seen that the results of [FKL07] for the permanent and the hamiltonian can be extended to other graph polynomials by straightforward modification of known VNP-completeness proofs. It seems very likely that many more such results can be proved relatively easily. But we have seen that the general principle stated in Conjecture 2 is not true, because it fails for clique polynomials. It would be interesting to see if these clique polynomials can express all of VP_e for other width measures, say cliquewidth.

This chapter and [FKL07] give many alternative characterizations of VP_e by graph polynomials. In [FKL07] there is also a characterization of VP_{ws} by a matching polynomial on planar graphs. Finally, there are lots of VNP-complete graph polynomials (see e.g. [Büo0, Chapter 3]). Curiously, there is no known characterization of VP by graph polynomials, so we leave this as an open question.

Question 2. *Is there a way to define graph polynomials that capture* VP*?*

One idea would be to slightly generalize Question 2 to hypergraphs and then consider the hypergraph width measures of Part i. While it is clear that not all monadic second order formulas will give tractable polynomials [GP04], specific polynomials could still be interesting. For example, it is easy to define an independent set polynomial for hypergraphs. In the light of the results of Chapter 4 it seems plausible that it is tractable for, say, acyclic hypergraphs. But can the exact complexity be determined?

12

ARITHMETIC BRANCHING PROGRAMS WITH MEMORY

12.1 INTRODUCTION

Arithmetic Branching Programs (ABPs) are a well studied model of computation in algebraic complexity: They were already used in the VNP-completeness proof of the permanent by Valiant [Val79] and have since then contributed to the understanding of arithmetic circuit complexity (see e.g. [Nis91, Koi12]). The computational power of ABPs is well understood: They are equivalent to both skew and weakly skew arithmetic circuits and thus capture the determinant, matrix power and other natural problems from linear algebra [Tod92, MP08]. The complexity of bounded width ABPs is also well understood: In analogy to Barrington's Theorem [Bar89], Ben-Or and Cleve [BOC92] proved that polynomial size ABPs of bounded width are equivalent to arithmetic formulas.

We modify ABPs by giving them memory during their computations and ask how this changes their computational power. There are several different motivations for doing this: We define branching programs with stacks, that are an adaption of the nondeterministic auxiliary pushdown automaton (NAuxPDA) model to the arithmetic circuit model. The NAuxPDA-characterization of LOGCFL has been very successful in the study of this class and has contributed greatly to its understanding. In this section we give a characterization of VP and thus contribute to the general aim of this part of this thesis to get a better understanding of VP. Our characterization also has some similarity to results in the Boolean setting in which graph connectivity problems on edge-labeled graphs that are similar to our ABPs with stacks were shown to be complete for LOGCFL [SV85, WS07]. One motivation for adapting these results to the arithmetic circuit setting is the hope that one can apply techniques from the NAuxPDA setting to arithmetic circuits. We show that this is indeed applicable by sketching an adaption of a proof of Niedermeier and Rossmanith [NR95] to give a straightforward proof of the classical parallelization theorem for VP first proved by Valiant et al. [VSBR83].

Another motivation is that our modified branching programs in different settings give various very similar characterizations of different arithmetic circuit classes. This allows us to give a new perspective on problems like VP vs. VP_{ws}, VP vs. VNP that are classical question from arithmetic circuit complexity. This is similar to the motivation

that Kintali [Kin10] had for studying similar graph connectivity problems for the Boolean setting.

Finally, all modifications we make to ABPs are straightforward and natural. The basic question is the following: ABPs are in a certain sense a memoryless model of computation. At each point of time during the computation we do not have any information about the history of the computation so far apart from the state we are in. So what happens if we allow memory during the computation? Intuitively, the computational power should increase, and we will see that it indeed does (under standard complexity assumptions of course). How do different types of memory compare? What is the role of the width of the branching programs if we allow memory? In the remainder we will answer several of these questions.

The structure of this chapter is a follows: After some preliminaries we start off with ABPs that may use a stack during their computation. We show that they characterize VP, consider several restrictions and sketch a proof of the parallelization theorem for VP. Next we consider ABPs with random access memory, show that they characterize VNP and consider some restrictions of them as well.

12.2 ARITHMETIC BRANCHING PROGRAMS

We quickly present some basic facts on arithmetic branching programs, a classical model of arithmetic circuit complexity.

Definition 12.2.1. *An* arithmetic branching program *(ABP) G is a directed acyclic graph with two vertices s and t and an edge labeling $w : E \to \mathbb{F} \cup \{X_1, X_2, \ldots\}$. A path $P = v_1 v_2 \ldots v_r$ in G has the* weight $w(P) := \prod_{i=1}^{r-1} w(v_i v_{i+1})$. *Let v and u be two vertices in G, then we define*

$$f_{v,u} = \sum_P w(P),$$

where the sum is over all v-u-paths P. The ABP G computes the polynomial $f_G = f_{s,t}$. The size *of G is the number of vertices of G.* □

Toda and Malod and Portier proved the following theorem:

Theorem 12.2.2. *([Tod92, MP08]) We have $(f_n) \in \mathsf{VP}_{ws}$, iff (f_n) is computed by a family of polynomial size ABPs.*

Definition 12.2.3. *An ABP of width k is an ABP in which all vertices are organized into layers $L_i, i \in \mathbb{N}$, there are only edges from layer L_i to L_{i+1} and the number of vertices in each layer L_i is at most k.* □

The computational power of ABPs of constant width was settled by Ben-Or and Cleve:

Theorem 12.2.4. *([BOC92]) We have $(f_n) \in \mathsf{VP}_e$, iff (f_n) is computed by a family of polynomial size ABPs of constant width.*

12.3 STACK BRANCHING PROGRAMS

12.3.1 *Definition*

Let S be a set called *symbol set*. For a symbol $s \in S$ we define two *stack operations*: $push(s)$ and $pop(s)$. Additionally we define the stack operation nop without any arguments. A *sequence of stack operations* on S is a sequence $op_1\, op_2 \dots op_r$, where either $op_i = \overline{op}_i(s_i)$ for $\overline{op}_i \in \{push, pop\}$ and $s_i \in S$ or $op_i = nop$. *Realizable sequences* of stack operations are defined inductively:

- The empty sequence is realizable.
- If P is a realizable sequence of stack operations, then the sequence $push(s)P\, pop(s)$ is realizable for all $s \in S$. Also $nop\, P$ and $P\, nop$ are realizable sequences.
- If P and Q are realizable sequences of stack operations, then PQ is a realizable sequence.

Definition 12.3.1. *A* stack branching program *(SBP) G is an ABP with an additional edge labeling $\sigma : E \to \{op(s) \mid op \in \{push, pop\}, s \in S\} \cup \{nop\}$. A path $P = v_1 v_2 \dots v_r$ in G has the sequence of stack operations $\sigma(P) := \sigma(v_1v_2)\sigma(v_2v_3)\dots\sigma(v_{r-1}v_r)$. If $\sigma(P)$ is realizable we call P a* stack-realizable path*. The SBP G computes the polynomial*

$$f_G = \sum_P w(P),$$

where the sum is over all stack-realizable s-t-paths P. □

It is helpful to interpret the stack operations as operations on a real stack that happen along a path through G. On an edge uv with the stack operation $\sigma(uv) = push(s)$ we simply push s onto the stack. If uv has the stack operation $\sigma(uv) = pop(s)$ we pop the top symbol of the stack. If it is s we continue the path, but if it is different from s the path is not stack realizable and we abort it. nop stands for "no operation" and thus as this name suggests the stack is not changed on edges labelled with nop. Realizable paths are exactly the paths on which we can go from s to t in this way without aborting while starting and ending with an empty stack.

To ease notation we sometimes call edges e with $\sigma(e) = push(s)$ for an $s \in S$ simply $push$-edges. The pop-edges and nop-edges are defined in the obvious analogous way.

It will sometimes be convenient to consider only SBPs that have no nop-edges. The following easy proposition shows that this is not a restriction.

Proposition 12.3.2. *Let G be an SBP of size s. There is an SBP G' of size $O(s^2)$ such that $f_G = f_{G'}$ and G' does not contain any nop-edges. If G is layered with width k, then G' is layered as well and has width at most k^2.*

Proof. The idea of the construction is to subdivide every edge of G. So let G be an SBP with vertex set V and edge set E. Let σ and w be the stack symbol labeling and the weight function, respectively. G' will have the vertex set $V \cup \{v_e \mid e \in E\}$, stack symbol labeling σ' and weight function w'. The construction goes as follows: For each edge $e = uv \in E$ the SBP G' has the edges uv_e, v_ev. We set $w'(uv_e) := w(uv)$ and $w'(v_ev) := 1$. If e is a nop-edge we set $\sigma'(uv_e) := push(s)$ and $\sigma'(v_ev) = pop(s)$ for an arbitrary stack symbol s. Otherwise, both uv_e and v_ev get the stack operation $\sigma(uv)$.

It is easy to verify that G' has all the desired properties. ■

12.3.2 *Characterizing VP*

In this section we show that stack branching programs of polynomial size characterize VP.

Theorem 12.3.3. $(f_n) \in \mathsf{VP}$, *iff* (f_n) *is computed by a family of polynomial size SBPs.*

We prove the two directions of Theorem 12.3.3 in two steps.

Lemma 12.3.4. *If* (f_n) *is computed by a family of polynomial size SBPs, then* $(f_n) \in \mathsf{VP}$.

Proof. Let (G_n) be a family of SBPs computing (f_n), of size at most $p(n)$ for a polynomial p. Observe that $\deg(G_n) \leq p(n)$, so we only have to show that we can compute the f_n by polynomial size circuits C_n.

Let $G = G_n$ be an SBP with m vertices, source s and sink t. The construction of $C = C_n$ uses the following basic observation: Every stack-realizable path P of length i between two vertices v and u can be uniquely decomposed in the following way. There are vertices $a, b, c \in V(G)$ and a symbol $s \in S$ such that there are edges va and bc with $\sigma(va) = push(s)$ and $\sigma(bc) = pop(s)$. Furthermore there are stack-realizable paths P_{ab} from a to b and P_{cu} from c to u such that $length(P_{ab}) + length(P_{cu}) = i - 2$ and $P = vaP_{ab}bcP_{cu}$. The paths P_{ab} and P_{cu} may be empty.

We define $w(u, v, i) := \sum_P w(P)$ where the sum is over all stack-realizable s-t-paths of length i.

We claim that the values $w(v, u, i)$ can be computed efficiently by a straightforward dynamic programming approach. First observe that $w(v, u, i) = 0$ for odd i. For $i = 0$ we set $w(v, u, 0) = 0$ for $v \neq u$ and $w(v, v, 0) = 1$. For even $i > 0$ we get by the above observation

$$w(v, u, i) = \sum_{a,b,c,j,s} w(v, a)w(a, b, j)w(b, c)w(c, u, i - j - 2),$$

where the sum is over all $s \in S$, all $j \leq i - 2$ and all a, b, c such that $\sigma(va) = push(s)$ and $\sigma(bc) = pop(s)$. Since there are only polynomially many combinations v, u, i, this recursion formula allows us to

compute all $w(v,u,i)$ with a polynomial number of arithmetic operations. Having computed all $w(v,u,i)$ we get $f_G = \sum_{i\in[m]} w(s,t,i)$. ■

The more involved part of the proof of Theorem 12.3.3 will be the reverse direction. To prove it it will be convenient to slightly relax our model of computation. A *relaxed SBP* G is an SBP where the underlying directed graph is not necessarily acyclic. To make use of cyclicity, in a relaxed SBP G, we do not consider paths but *walks*, i.e., vertices and edges of G may be visited several times. *Realizable walks* are defined completely analogously to realizable paths. Also the weight $w(P)$ of a walk is defined in the obvious way. Clearly, we cannot define the polynomial computed by a relaxed ABP by summing over the weights of all realizable walks, because there may be infinitely many of them since they may be arbitrarily long. Hence, we define for each pair u,w of vertices and for each integer m the polynomial

$$f_{u,v,m} := \sum_P w(P),$$

where the sum is over all stack-realizable u-v-walks P in G that have length m. Furthermore, we say that, for each m, the relaxed SBP G computes the polynomial $f_{G,m} := f_{s,t,m}$.

The connection to SBPs is given by the following straight-forward lemma.

Lemma 12.3.5. *Let G be a relaxed SBP and $m \in \mathbb{N}$. Then for each m there is an SBP G'_m of size $m|G|$ that computes $f_{G,m}$.*

Proof. The idea is to unwind the computation of the relaxed SBP into m layers. Let $G = (V,E,w,\sigma)$, then for each $v \in V$ the SBP G' has m copies $\{v_1,\dots,v_m\}$. For each $uv \in E$ the SBP G' had the edges u_iv_{i+1} for $i \in [m-1]$ with weight $w(u_iv_{i+1}) := w(uv)$ and stack operation $\sigma(u_iv_{i+1}) := \sigma(uv)$. This completes the construction of G'.

Clearly, G' computes $f_{G,m}$ and has size $m|G|$. ■

To prove the characterization of VP we show the following rather technical proposition:

Proposition 12.3.6. *Let C be a multiplicatively disjoint arithmetic circuit. For each $v \in V$ we denote by C_v the subcircuit of C with output v, and we denote by f_v the polynomial computed by C_v. Then there is a relaxed SBP $G = (V,E,w,\sigma)$ of size at most $2|C|(|C|+1)+3(|C|)$ such that for each $v \in V$ there is a pair $v_-,v_+ \in V$ and an integer $m_v \leq 4|C_v|$ with*

- *$f_v = f_{v_-,v_+,m_v}$, and*
- *there is no stack-realizable walk from v_- to v_+ in G that is shorter than m_v.*

Proof. We construct G iteratively along a topological order of C by adding new vertices and edges, starting from the relaxed SBP with empty vertex set. We distinguish three cases:

Case 1: Let first v be an input of C with label X. We add two new vertices v_-, v_+ to G and the edge v_-v_+ with weigth $w(v_-v_+) := X$ and stack-operation $\sigma(v_-v_+) := nop$. Furthermore, $m_v := 1$. Clearly, none of the polynomials computed before change and the size of the relaxed SBP grows only by 2. Thus all statements of the proposition are fulfilled.

Case 2: Let now v be an addition gate with children u, w. By induction G contains vertices u_-, u_+, w_-, w_+ and there are m_u, m_v such that $f_{u_-,u_+,m_u} = f_u$ and $f_{w_-,w_+,m_w} = f_w$. Assume w.l.o.g. $m_u \geq m_w$. We add the new vertices v_-, v_+, v_s, v_t to G. We further add the edges v_-u_-, v_-v_s, v_tw_-, u_+v_+ and w_+v_+. Moreover, we connect v_s an v_t by a path of length $m_u - m_w$ whose inner vertices are also new. All edges we add get weight 1. Furthermore, we set the stack symbol operations $\sigma(v_-u_-) := push(vu)$, $\sigma(u_+v_+) := pop(vu)$, $\sigma(v_-v_s) := push(vw)$ and $\sigma(w_+v_+) := pop(vw)$ for new stack symbols vu and vw. All other edges we added are nop-edges. Finally, set $m_v := m_u + 2$.

Let us check that G computes the correct polynomials. First observe that the edges we added do not allow any new walks between old vertices, so we still compute all old polynomials by induction. Thus we only have to consider the realizable v_--v_+-walks of length m_v. Each of these either starts with the edge v_-u_- or the edge v_-v_s. In the first case, because of the stack symbols the walk must end with the edge u_+v_+. Thus the realizable v_-v_+-walks of length m_v that start with v_-u_- contribute exactly the same weight as the realizable u_--u_+-walks of length m_u. Hence, these weights add up to f_u by induction. Every v_-v_+-walks of length m_v that start with v_-v_s first makes $m_u - m_w$ unweighted steps to w_- and ends with the edge w_+v_+. Thus, these walks contribute exactly the same as the stackrealizable w_--w_+ walks of length $m_v - 2 - (m_u - m_w) = m_w$, so they contribute f_w. Combining all walks we get $f_{v_-,v_+,m_v} = f_u + f_w = f_v$ as desired.

We have that every realizable walk from u_+ to u_- has length at least m_u, and thus there is no realizable v_--v_+-walk starting with v_-u_- that is shorter than $m_u + 2 = m_v$. Moreover, since the realizable w_--w_+-walks have length at least m_w, the realizable paths starting with v_-w_- have length at least $m_w + (m_u - m_w) + 2 = m_u + 2 = m_w$. Thus there is no realizable v_--v_+-walk of length less that m_v.

We have $m_v = m_u + 2 \leq 4|C_u| + 2 \leq 4|C_v|$ where the first inequality is by induction and the second inequality follows from the fact that v is not contained in C_u and thus $|C_v| > |C_u|$. To see the bound on $|G|$ let s be the size of G before adding the new edges and vertices. By induction $s \leq 2(|C_v| - 1)(|C_v| - 1 + 1) + 3(|C_v| - 1)$. We have added $2 + m_u - m_v + 1$ vertices and thus G has now size $s + 3 + m_u - m_v \leq s + 3 + m_u$. But we have $m_u \leq 4|C_u| \leq 4|C_v|$ and thus the number

of vertices in G is at most $2(|C_v|-1)|C_v|+3(|C_v|-1)+3+4|C_v| \leq 2|C_v|(|C_v|+1)+3|C_v|$. This completes the case that v is an addition gate.

Case 3: Let now v be a multiplication gate with children u, w. As before, G already contains u_-, u_+, w_-, w_+ and there are m_u, m_v with the desired properties. We add three vertices v_-, v_+ and v_* and the edges v_-u_-, u_+v_*, v_*w_- and w_+v_+ all with weight 1. The new edges have the stack symbols $\sigma(v_-u_-) := push(vu)$, $\sigma(u_+v_*) := pop(vu)$, $\sigma(v_*w_-) := push(vw)$ and $\sigma(w_+v_+) := pop(vw)$ for new stack symbols vu and vw. Finally, set $m_v := m_u + m_w + 4$.

Clearly, no stack-realizable walk between any pair of old vertices can traverse v_-, v_+ or v_* and thus these walks still compute the same polynomials as before. Thus we only have to analyse the v_--v_+-walks of length m_v in G. Let P be such a walk. Because of the stack symbols vu and vw the walk P must have the structure $P = v_-u_-P_1u_+v_*w_-P_2w_+v_+$ where P_1 and P_2 are a stack-realizable u_--u_+-walk and a stack-realizable w_--w_+-walk, respectively. The walk P is of length m_v and thus P_1 and P_2 must have the combined length $m_u + m_w$. But by induction P_1 must at least have length m_u and P_2 must have at least length m_w, so it follows that P_1 has length exactly m_u and P_2 has length exactly m_w. The walks P_1 and P_2 are independent and thus we have $f_{v_-,v_+,m_v} = f_{u_-,u_+,m_u} f_{w_-,w_+,m_w} = f_u f_w$ as desired.

The circuit C is multiplicatively disjoint and thus we have $|C_v| = |C_u| + |C_w| + 1$. It follows that $m_v = m_u + m_w + 4 \leq 4|C_u| + 4|C_w| + 4 = 4|C_v|$ where we get the inequality by induction. The relaxed SBP has grown only by 3 vertices which gives the bound on the size of G. This completes the proof for the case that v is an addition gate and hence the proof of the lemma. ■

Now the second direction of Theorem 12.3.3 is straightforward.

Lemma 12.3.7. *Every family $(f_n) \in \mathsf{VP}$ can be computed by a family of SBPs of polynomial size.*

Proof. Given a family (C_n) of multiplicatively disjoint arithmetic circuits of polynomial size, first turn them into relaxed SBPs of polynomial size and polynomial m with Proposition 12.3.6 and then turn those relaxed SBPs into SBPs with Lemma 12.3.5. It is easy to check that the resulting SBPs have polynomial size. ■

Let us quickly discuss the proof of Proposition 12.3.6. The only place in which we use the multiplicative disjointness of the circuit C is when we give the upper bound for m_v in Case 3. Consequently, the following Lemma is not hard to prove.

Lemma 12.3.8. *For every (not necessarily multiplicatively disjoint) arithmetic circuit C there is a relaxed SBP G of size polynomial in $|C|$ and an integer $m = 2^{|C|^{O(1)}}$ such that $f_{G,m}$ is the polynomial computed by C.*

Proof (sketch). We make nearly the same construction as in the proof of Proposition 12.3.6. The only minor complication is that in Case 2 we cannot simply add a path of length $m_u - m_w$ to G, because this number may be exponential in $|G|$, since in Case 2 we may have $m_v = 2m_u$ and iterating Case 3 may cause an exponential blow-up. Fortunately, it is not hard to construct a gadget G_k of size $O(k)$ such that every stack-realizable walk through G_k must be of length 2^k. Using this gadget G_k for different values of k one can construct a gadget G' of polynomial size such that every stack-realizable walk through G' has length at least $m_u - m_v$. Using G' instead of a directed path of length $m_u - m_v$ gives a construction with all desired properties. ■

It is not apparent if the reverse of Lemma 12.3.8 is true, so we leave this as a small question.

Question 3. *Given a relaxed SBP G and an integer m of size at most $2^{O(|G|)}$, is there always an arithmetic circuit of size polynomial in $|G|$ that computes $f_{G,m}$?*

Let us also quickly point out the key difference between our construction in the proof of Proposition 12.3.6 and the corresponding construction of Malod and Portier [MP08] for weakly skew circuits. If a circuit C is weakly skew, we may assume that in Case 3 of the construction the pairs u_-, u_+ and w_-, w_+ are in different components of G. Thus one can simply append one of the components to the other to get an ABP such that the underlying graph is *acyclic*. When C is not weakly skew, u_-, u_+ and w_-, w_+ need not be in different components of G. Thus our construction introduces cycles in G and we use the stack to make sure that G is traversed by realizable walks in the desired way.

12.3.3 *Stack branching programs with few stack symbol*

We start off this section with an easy observation.

Lemma 12.3.9. *For every SBP G one can construct an SBP G' with stack symbol set $\{0,1\}$ and size $|G'| = O(|G| \log(|G|))$ such that G and G' compute the same polynomial.*

Proof. Let S be the set of stack symbols appearing along edges of G. Let $\ell := \lceil \log(|S|) \rceil$, then each stack symbol s can be encoded into a $\{0,1\}$-string $\mu(s)$ of length ℓ. We substitute each edge e of G by a path P_e of length ℓ. If $\sigma'(e) = push(s)$ the edges along P_e are *push*-edges as well, that push $\mu(e)$ onto the stack. If $\sigma'(e) = pop(s)$ we pop $\mu(s)$ in reverse order along P_e. If e is a *nop*-edge, all edges of P_e are also *nop*-edges. Finally, we give one of the edges in P_e the weight $w'(e)$, while all other edges get weight 1. Doing this for all

edges, it is easy to see that the resulting SBP G' computes the same polynomial as G. Moreover, G' has size $|G|\ell = O(|G|\log(|G|))$. ■

Considering Lemma 12.3.9, the only meaningful restriction of the size of the symbol set is the restriction to one single symbol. The following fairly straightforward lemma shows that doing so decreases the computational power. Note that Kintali [Kin10] proved a similar result for the Turing machine setting.

Lemma 12.3.10. $(f_n) \in \mathsf{VP}_{ws}$ *if and only if it can be computed by polynomial size SBPs with one stack symbol.*

Proof. The direction from left to right is easy: Simply interpret each edge e of an ABP G as a *nop*-edge.

For the other direction the key insight is that if one has only one stack symbol, one only has to keep track of the *size* of the stack at any point on the path. We will see that this size can be encoded by vertices of an ABP.

So let G be a SBP of size m. It is clear that the stack can never contain more than m symbols on any path through G. We construct an ABP G' that has for every vertex v in G the $m+1$ vertices $v_0, v_1, \dots v_m$. If vu is a *push*-edge in G, we connect v_i to u_{i+1} for $i = 0, \dots, m-1$ in G'. If vu is a *pop*-edge in G, we add $v_i u_{i-1}$ for $i = 1, \dots, m$ to G'. All these edges get the same weight as vu in G. It is easy to see that every stack-realizable s-t-path P in the SBP G corresponds directly to an s_0-t_0-path P' in the ABP G' and P and P' have the same weight. Thus G and G' compute the same polynomial. Moreover, $|G'| = (m+1)|G|$ which completes the proof. ■

12.3.4 *Width reduction*

In this section we show that, unlike for ordinary ABPs, bounding the width of SBPs does not decrease the computational power: Polynomial size SBPs with at least 2 stack symbols and width 2 can still compute every family in VP.

Lemma 12.3.11. *Every family* $(f_n) \in \mathsf{VP}$ *can be computed by a SBP of width* 2 *with the stack symbol set* $\{0, 1\}$.

Proof. The idea of the proof is to start from the characterization of VP by SBPs from Theorem 12.3.3. We use the stack to remember which edge will be used next on a realizable path through the branching program. We will show how this can be done with width 2 SBPs with a bigger stack symbol size. In a second step we will reduce the stack symbol set to $\{0, 1\}$ with Lemma 12.3.9.

So let (G_n) be a family of SBPs. Fix n and let $G := G_n$ with vertex set V and edge set E. Furthermore, let w be the weight function, σ the stack operation labeling and S the set of stack symbols of G. Let s

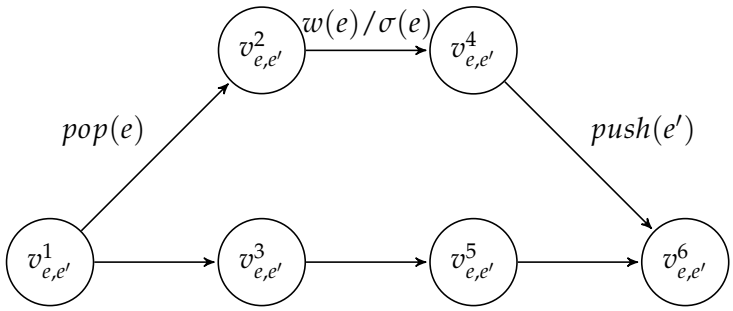

Figure 17: The gadget $G_{e,e'}$. We illustrate only the weight of the weighted edge. All edges without stack operation label are nop-edges.

and t be the source and the sink of the SBP G. We assume without loss of generality that s has one single outgoing edge e_s. Furthermore, t is only entered by one nop-edge e_t with weight 1. We will construct a new SBP G' with weight function w' and stack operation labeling σ'. The SBP G' will have $S \cup E$ as the set of stack symbols.

For each edge e with a successor edge e' the SBP G' contains a gadget $G_{e,e'}$. The vertex set of $G_{e,e'}$ is $\{v^1_{e,e'}, v^2_{e,e'}, v^3_{e,e'}, v^4_{e,e'}, v^5_{e,e'}, v^6_{e,e'}\}$. These vertices are connected to a DAG by the edges $\{v^1_{e,e'}v^2_{e,e'}, v^1_{e,e'}v^3_{e,e'}, v^2_{e,e'}v^4_{e,e'}, v^3_{e,e'}v^5_{e,e'}, v^4_{e,e'}v^6_{e,e'}, v^5_{e,e'}v^6_{e,e'}\}$. All these edges have weight 1 except for $v^2_{e,e'}v^4_{e,e'}$ which we give the weight $w'(v^2_{e,e'}v^4_{e,e'}) := w(e)$. We call $v^2_{e,e'}v^4_{e,e'}$ the *weighted edge* of $G_{e,e'}$. Furthermore, we set $\sigma(v^1_{e,e'}v^2_{e,e'}) := pop(e)$, $\sigma(v^2_{e,e'}v^4_{e,e'}) := \sigma(e)$, $\sigma(v^4_{e,e'}v^6_{e,e'}) := push(e')$. All other edges are nop-edges. The construction of $G_{e,e'}$ is illustrated in Figure 17.

We now construct order $\leq_E$ of E. Let $\leq_V$ be a topological order of V. Then we define for $uv, u'v' \in E$

$$uv \leq_E u'v' \leftrightarrow u <_V u' \vee (u = u' \wedge v \leq_V v').$$

Observe that for each pair $uv, vw \in E$ we have $uv <_E vw$.

From $\leq_E$ we construct an order $\leq_G$ of the gadgets $G_{e,e'}$ by defining

$$G_{e_1,e_2} \leq_G G_{e_3,e_4} \leftrightarrow e_1 <_E e_3 \vee (e_1 = e_3 \wedge e_2 \leq_E e_4).$$

We now connect the gadgets along the order $\leq_G$ in the following way: Let G_{e_3,e_4} be the successor of G_{e_1,e_2} in $\leq_G$. We connect $v^6_{e_1,e_2}$ to $v^1_{e_3,e_4}$ by a nop-edge of weight 1. Let $G_{e,e'}$ be the minimum of $\leq_G$. We add a new vertex s and the edge $sv^1_{e,e'}$ with weigth 1 and stack operation $\sigma(sv^1_{e,e'}) := push(e_s)$ where e_s is the single outgoing edge of s in G. Let now $G_{e,e'}$ be the maximum gadget in $\leq_G$. We add a new vertex t and the edge $v^6_{e,e'}t$ with weight 1 and stack operation $pop(e_t)$. This concludes the construction of G'.

It is easy to see that G' has indeed width 2. Thus we only need to show that G and G' compute the same polynomial. This will follow directly from the following claim:

Claim 12.3.12. *There is a bijection π between the stack-realizable paths in G and G'. Furthermore $w(P) := w'(\pi(P))$ for each stack-realizable path in G.*

Proof. Clearly every s-t-path in G' must traverse all gadgets in G'. Furthermore, whenever a gadget is entered, the stack contains only one symbol from E which lies at the top of the stack. Through each gadget $G_{e,e'}$ there are exactly the two paths $v^1_{e,e'}v^2_{e,e'}v^4_{e,e'}v^6_{e,e'}$ and $v^1_{e,e'}v^3_{e,e'}v^5_{e,e'}v^6_{e,e'}$. We call the former the *weighted path* through $G_{e,e'}$. For every stack-realizable s-t-path $P = e_1e_2 \dots e_k$ through G we define $\pi(P)$ to be the unique path through G' that takes the weighted path through exactly the gadgets $G_{e_i,e_{i+1}}$ for $i = 1, \dots, k = 1$. We have $w(P) := w'(\pi(P))$ with this definition, because only the weighted edges in the gadgets have a weight different from 1 in G'. So it suffices to show that π is indeed a bijection.

We first show that π maps stack-realizable paths in G to stack-realizable paths in G'. So let P be as before. Observe that $\pi(P)$ traverses the gadgets $G_{e_i,e_{i+1}}$ in the same order as P traverses the edges e_i. Furthermore, whenever $\pi(P)$ enters a gadget $G_{e_i,e_{i+1}}$ the top stack symbol is e_i and the rest of the stack content is exactly that on P before traversing e_i. When leaving $G_{e_i,e_{i+1}}$ the stack content is that after traversing e_i on P with an additional symbol e_{i+1} on the top. Thus all stack operations along $\pi(P)$ are legal and the stack is empty after traversing the last edge towards t. Thus $\pi(P)$ is indeed stack-realizable.

Clearly, π is injective, so to complete the proof of the claim we only need to show that it is surjective. So let P' be a stack-realizable s-t-path in G'. Let $G_{e_1,e'_1}, \dots, G_{e_k,e'_k}$ be the gadgets in which P' takes the weighted path in the order in which they are visited. We claim that the path $P := e_s e_1 \dots e_k$ is a stack-realizable s-t-path. Clearly, s is the first vertex of P. Also in P' the symbol e_t is popped in the last step by construction of G', so the last gadget in which P' took a weighted path must be one of the form G_{e,e_t}, because otherwise e_t cannot be the top symbol on the stack before the last step. Thus t is the last vertex of P.

To see that P is a path, observe that we have $e'_i = e_{i+1}$. Otherwise P cannot have the right top symbol when taking the weighted path in $G_{e_{i+1},e'_{i+1}}$. Thus e_{i+1} must be a successor of e_i in G and P is an s-t-path.

To see that P is stack-realizable observe that when P' traverses the weighted edge of a gadget G_{e_i,e'_i} it has the same stack content as when P traverses e_i in G. So P is obviously stack-realizable because P' is.

Observing that obviously $w(P) = P'$ by construction completes the proof. ∎

We now reduce the stack symbol set of G' to $\{0,1\}$ with the construction of Lemma 12.3.9. Note that because of the specific form of the gadgets $G_{e,e'}$ this construction does not increase the width of the

resulting SBP and thus it yields an SBP G'' of width 2 computing the same polynomial as G with stack symbol set $\{0,1\}$. ■

12.3.5 *Depth reduction*

In this section we sketch how the characerization of VP by SBPs allows us to directly use results from counting complexity that rely on NAuxPDAs. More specifically, we will discuss how one can adapt a proof by Niedermeier and Rossmanith [NR95] to reprove the classical parallelization theorem for VP originally proved by Valiant et al. [VSBR83].

The basic idea is the following: The realizable paths in an SBP are recursively cut into subpaths and the polynomials are then computed by combining the polynomials of the subpaths. In order to reach logarithmic depth we have to make sure that the paths are cut in paths of approximately equal length to result in a balanced computation. This is complicated by the fact that the paths have to be realizable, so we have to account for the content of the stack during the computation.

Ideally, we would like to cut each realizable path of an SBP G into subpaths of roughly equal length and then give recursion formulas similar to those in the proof of Lemma 12.3.4 to get a logarithmic depth circuit computing f_G. Unfortunately, the situation is not quite so easy for SBPs because we have to account for the stack content along the path. So we cannot simply cut all paths in the middle and then recombine these subpaths freely later as we could for ABPs, because some of these combinations might not be stack-realizable. One way to prevent these wrong combinations would be to remember the stack changes along paths explicitly, but as there can be exponentially many stack configurations this does not result in polynomial size circuits, at least if done naively.

The solution of Niedermeier and Rossmanith is different: Instead of decomposing a stack-realizable path P by cutting it once in the middle, they cut a realizable path out of the middle of P. This results in a realizable path P_1 and a path P_2 with a "gap" in the middle. This decomposition is made in such a way that whenever the gap of P_2 is filled with a realizable path, the result is a realizable path. Furthermore, P_1 and P_2 both have a constant fraction of the edges of P. Now this procedure can be recursed. P_1 and P_2 are decomposed similarly to P. During this recursion one makes sure that every path P' with gap only has a single gap and not several of them. Thus P' can be described by few parameters: One just remembers the starting and end vertex and the number of edges of P'. Furthermore, one remembers the last vertex before the gap, the first vertex after the gap and the length of the realizable path that was cut out of P'. Implementing these ideas, one gets recursion formulas for the computation of f_G that can be turned into an arithmetic circuit analogously to the proof

of Lemma 12.3.4 but this time with logarithmic depth. For details we refer the reader to [NR95].

12.4 RANDOM ACCESS MEMORY

12.4.1 *Definition*

We change the model of computation by allowing random access memory instead of a stack. We still work over a symbol set S like for SBPs but we introduce three *random access memory operations*: The operations *write* and *delete* take an argument $s \in S$ while the operation *nop* again takes no argument. Let $op(s)$ be a random access memory operation with $op \in \{write, delete\}$ and $P = op_1\, op_2 \dots op_r$ a sequence of memory operations. By $occ(P, op(s))$ we denote the number of occurences of $op(s)$ in P. We call a sequence P *realizable* if for all symbols $s \in S$ we have that $occ(P, write(s)) = occ(P, delete(s))$ and for all prefixes P' of P we have $occ(P', write(s)) \geq occ(P', delete(s))$ for all $s \in S$.

Intuitively, the random access memory operations do the following: $write(s)$ writes the symbol s into the random access memory. If s is already there it adds it another time. The operation $delete(s)$ deletes one occurence of the symbol s from the memory if there is one. Otherwise an error occurs. *nop* is the "no operation" operation again like for SBPs. A sequence of operations is realizable if no error occurs during the deletions, and moreover starting from empty memory the memory is empty again after the sequence of operations.

Definition 12.4.1. *A* random access branching program *(RABP) G is an ABP with an additional edge labeling*

$$\sigma : E \to \{op(s) \mid op \in \{write, delete\}, s \in S\} \cup \{nop\}.$$

A path $P = v_1 v_2 \dots v_r$ in G has the sequence of random access memory operations $\sigma(P) := \sigma(v_1v_2)\sigma(v_2v_3)\dots\sigma(v_{r-1}v_r)$. If $\sigma(P)$ is realizable we call P a random-access-realizable path*. The RABP G computes the polynomial*

$$f_G = \sum_P w(P),$$

where the sum is over all random-access-realizable s-t-paths P. □

In a completely analogous way to Proposition 12.3.2 we can prove that disallowing *nop*-edges does not change the computational power of RABPs.

Proposition 12.4.2. *Let G be an RABP of size s. There is an RABP G' of size $O(s^2)$ such that $f_G = f_{G'}$ and G' does not contain any nop-edges. If G is layered with width k, then G' is layered as well and has width at most k^2.*

12.4.2 *Characterizing VNP*

Intuitively, random access on the memory should allow us more fine-grained control over the paths in the branching program that contribute to the computation. While in SBPs nearly all of the memory content is hidden, in RABPs we have access to the complete memory at all times. This makes RABPs more expressive than SBPs which is formalized in the following theorem.

Theorem 12.4.3. $(f_n) \in \mathsf{VNP}$ *if and only if there is a family of polynomial size RABPs computing* (f_n).

Again we prove the theorem in two independent lemmas, starting with the upper bound, which is very easy.

Lemma 12.4.4. *If* (f_n) *is computed by a family of polynomial size RABPs, then* $(f_n) \in \mathsf{VNP}$.

Proof. This is easy to see with Valiant's criterion (Lemma 9.2.3) and the fact that checking if a path through a RABP is realizable is certainly in P. ■

We will now show the lower bound of Theorem 12.4.3, which we will prove directly for bounded width RABPs. To do so we reduce from the dominating-set polynomial $GF(., \phi_{DS})$ which we have shown to be VNP-complete in Corollary 11.4.8.

Lemma 12.4.5. *For each family* $(f_n) \in \mathsf{VNP}$ *there is a family of width* 2 *RABPs of polynomial size computing* (f_n).

Proof. We will show that for a graph $G = (V, E)$ with n vertices there is a RABP of size $n^{O(1)}$ and width 2 that computes $GF(G, \phi_{DS})$. The RABP works in two stages. The symbol set of the RABP will be V. In a first stage it iteratively selects vertices v and writes v and all of its neighbors into the memory. In a second stage it checks that each vertex v was written at least once into the memory, i.e., either v or one of its neighbors was chosen in the first phase. Thus the set of chosen vertices must have been a dominating set.

So fix a graph G and set $w(v) = X_v$ for each $v \in V$. For each vertex v with neighbors $v_1, \ldots, v_d$ we construct a *choose gadget* G_v as shown in Figure 18. We call the path through G_v consisting of the edges that have memory operations the *choosing path*. Moreover, for each vertex v we construct a *check gadget* G'_v that is shown in Figure 19.

Choose an order on the vertices. For each non-maximal vertex v in the order with successor u, we connect the sink of G_v to the source of G_u and the sink of G'_v to the source of G'_u with a *nop*-edge of weight 1. Finally, let x be the maximal vertex in the order and y the

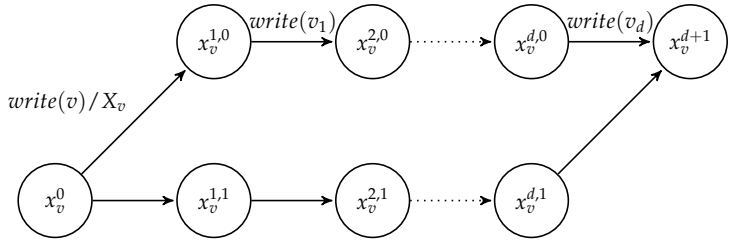

Figure 18: The gadget G_v. Let v be a vertex with neighbors $v_1, \ldots, v_d$. The weight of $x_v^0 x_v^{1,0}$ is X_v while all other edges have weight 1. G_v has two paths. Every realizable path that traverses G_v on the upper path writes v and all of its neighbors into the memory. This path has weight X_v. Realizable paths through the lower path do not change the memory in G_v and have a weight weight contribution of 1 in G_v.

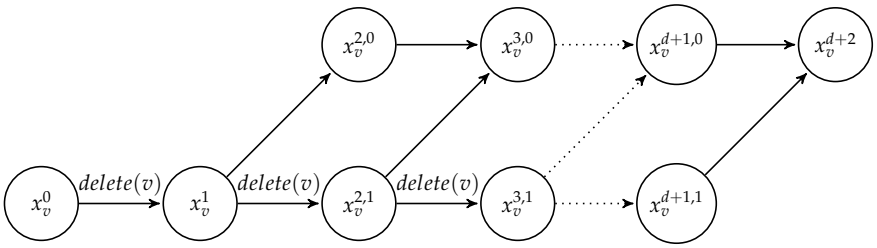

Figure 19: The gadget G'_v. Let d be the degree of v, then G'_v has $d+3$ layers. All edges have weight 1. The edges connecting vertices in the lower level have operation $delete(v)$ while all other edges have no memory operation. Every realizable path through G'_v has weight 1 and deletes between 1 and $d+1$ occurences of the symbol v from memory.

minimal vertex. Connect the sink of G_x to the source of G'_y again by a nop-edge of weight 1.

We claim that G' computes $GF(G, \phi_{DS})$. To see this, define the weight of a vertex set D in G to be $w(D) := \prod_{v \in D} X_v$. The following claim completes the proof.

Claim 12.4.6. *There is a bijection π between dominating sets in G and random-access-realizable paths in G' such that for each dominating set D in G we have $w(D) := w(\pi(D))$.*

Proof. Observe that for random-access-realizable paths through G' once the path through the gadgets G_v is chosen, then rest of the path is fixed. So each random-access-realizable path P can be described completely by the v for which the choosing paths through G_v is taken.

Let D be a dominating set. Let $\mathcal{P}$ be the set of s-t-paths in G' that for each $v \in D$ take the choosing path through G_v and for each $v \notin D$ take the other path. Because D is dominating, after a path $P \in \mathcal{P}$ has passed through the choose gadgets G_v, it contains each symbol $v \in V$ at least once. Then there is a unique path through the delete gadgets

such that every memory symbol is deleted at its end. This path is the unique random-access realizable path in $\mathcal{P}$ which we call $\pi(D)$.

Obviously, π is injective. To show that it is also surjective, consider a random-access-realizable path P in G'. Let D be the set of $v \in V$ for which P takes the choosing path in the choose gadgets. The path P passes every delete gadget G'_v, so each element $v \in V$ gets deleted from the memory at least once. It follows that each $v \in V$ must have been written to memory at least once before. So for $v \in V$ the path P must go through the choosing path in the choose gadget G_v or through the choosing path in G_u for a neighbor u of v. It follows that D is a dominating set. Furthermore, $\pi(D) = P$, so π is surjective.

Finally, $w(D) := w(\pi(D))$ is true, because the only weighted edges in G' are in the gadgets G_v and for each v the weighted edge in G_v has the weight X_v. ■

Observing that G' has width 2 completes the proof. ■

Part III

MONOMIALS IN ARITHMETIC CIRCUITS

13

INTRODUCTION AND PRELIMINARIES

13.1 INTRODUCTION

In this part of the thesis we study arithmetic circuits from a completely different perspective than in Part ii. Instead of studying how hard it is to compute a given polynomial by arithmetic circuits, we now ask how hard it is to decide properties of the polynomial computed by a given arithmetic circuit. We study mainly two problems: The first one is to decide whether a given monomial has a zero coefficient in the polynomial computed by a given circuit, while the second consists of counting the number of computed monomials. We characterize their complexity using the counting hierarchy.

The *counting hierarchy* refers to the family of classes $\mathsf{PP} \cup \mathsf{PP}^{\mathsf{PP}} \cup \mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}} \cup \ldots$ which has appeared in several recent papers. For example, Bürgisser [Bür09] uses these classes to connect computing integers to computing polynomials, while Jansen and Santhanam [JS11]—building on results by Koiran and Perifel [KP11]—use them to derive lower bounds from derandomization. The counting hierarchy was originally introduced by Wagner [Wag86] to classify the complexity of combinatorial problems. Curiously, after Wagner's paper and another by Torán [Tor88], this original motivation of the counting hierarchy has to the best of our knowledge not been pursued for more than twenty years. Instead, research focused on structural properties and the connection to threshold circuits [AW93]. As a result, very few natural complete problems for classes in the counting hierarchy are known: for instance, Kwisthout et al. give in [KBvdG11] "the first problem with a practical application that is shown to be $\mathsf{FP}^{\mathsf{PP}^{\mathsf{PP}}}$-complete". The related class $\mathsf{C}_{=}\mathsf{P}$ appears to have no natural complete problems at all (see [HO02, p. 293]). It is however possible to define generic complete problems by starting with a #P-complete problem and considering the variant where an instance and a positive integer are provided and the question is to decide whether the number of solutions for this instance is equal to the integer. We consider these problems to be counting problems disguised as decision problems and thus not as natural complete problems for $\mathsf{C}_{=}\mathsf{P}$, in contrast to the questions studied here. Note that the corresponding logspace counting class $\mathsf{C}_{=}\mathsf{L}$ is known to have interesting complete problems from linear algebra [ABO99].

In this part of this thesis we follow Wagner's original idea and show that the counting hierarchy is a helpful tool for classifying the complexity of several natural problems on arithmetic circuits by show-

ing complete problems for the classes $\mathsf{PP}^{\mathsf{PP}}$, $\mathsf{C}_{=}\mathsf{P}$ and some related classes.The common setting of these problems is the use of circuits or straight-line programs to represent polynomials. Such a representation can be much more efficient than giving the list of monomials, but common operations on polynomials may become more difficult. An important example is the question of determining whether the given polynomial is identically zero. This is easy to do when the polynomial is given as a list of monomials. When the polynomial is given as a circuit however, this problem, called ACIT for *arithmetic circuit identity testing*, is not known to be in P, though it is in coRP. In fact, derandomizing this problem would imply significant circuit lower bounds, as shown in [KI04]. This question thus plays a crucial part in complexity and it is natural to consider other problems on polynomials represented as circuits. In this part of the thesis we consider mainly two questions.

The first main problem, called ZMC for *zero monomial coefficient*, is to decide whether a given monomial in a circuit has coefficient 0 or not. This problem has already been studied by Koiran and Perifel [KP07]. They showed that when the formal degree of the circuit is polynomially bounded, the problem is complete for $\mathsf{P}^{\#\mathsf{P}}$. Unfortunately this result is not fully convincing, because it is formulated with the rather obscure notion of strong non-deterministic Turing reductions. We remedy this situation by proving a completeness result for the class $\mathsf{C}_{=}\mathsf{P}$ under the traditional logarithmic-space many-one reductions. This also provides a natural complete problem for this class. Koiran and Perifel also considered the general case of ZMC, where the formal degree of the circuit is not bounded. They showed that ZMC is in the counting hierarchy. We provide a better upper bound by proving that ZMC is in $\mathsf{coRP}^{\mathsf{PP}}$.

The second main problem is to count the number of monomials in the polynomial computed by a circuit. This seems like a natural question whose solution should not be too hard, but in the general case it turns out to be $\mathsf{PP}^{\mathsf{PP}}$-complete, and the hardness holds even for bounded depth circuits. We thus obtain another natural complete problem, in this case for the second level of the counting hierarchy. We remark that if a polynomial bound is known on the number of monomials, both the problem ZMC and the one of counting monomials become easy since an explicit description of the polynomial can be computed in polynomial time [GS09]. The related problem of enumerating the monomials of a given polynomial, in the black-box model, is addressed in [Strar].

Then we study the two above problems in the case of circuits computing multilinear polynomials. We show that our first problem becomes equivalent to the fundamental problem ACIT and that counting monomials becomes PP-complete.

Finally, we consider the case of univariate multiplicatively disjoint circuits. We show that these problems and several related ones are equivalent and complete for LOGCFL in the monotone case, or close to $\mathsf{C}_=\mathsf{LOGCFL}$ in the general case.

13.2 PRELIMINARIES

We assume that the reader is familiar with basic concepts of computational complexity theory (see e.g. [AB09]). All reductions in this chapter will be logspace many-one unless stated otherwise.

We consider different counting decision classes in the counting hierarchy [Wag86]. These classes are defined analogously to the quantifier definition of the polynomial hierarchy but, in addition to the quantifiers $\exists$ and $\forall$, the counting quantifiers C, $\mathsf{C}_=$ and $\mathsf{C}_{\neq}$ are used.

Definition 13.2.1. *Let $\mathcal{C}$ be a complexity class containing* P.

- $A \in \mathsf{C}\mathcal{C}$ *if and only if there is* $B \in \mathcal{C}$, $f \in \mathsf{FP}$ *and a polynomial* p *such that*

$$x \in A \Leftrightarrow \left|\left\{y \in \{0,1\}^{p(|x|)} \mid (x,y) \in B\right\}\right| \geq f(x),$$

- $A \in \mathsf{C}_=\mathcal{C}$ *if and only if there is* $B \in \mathcal{C}$, $f \in \mathsf{FP}$ *and a polynomial* p *such that*

$$x \in A \Leftrightarrow \left|\left\{y \in \{0,1\}^{p(|x|)} \mid (x,y) \in B\right\}\right| = f(x),$$

- $A \in \mathsf{C}_{\neq}\mathcal{C}$ *if and only if there is* $B \in \mathcal{C}$, $f \in \mathsf{FP}$ *and a polynomial* p *such that*

$$x \in A \Leftrightarrow \left|\left\{y \in \{0,1\}^{p(|x|)} \mid (x,y) \in B\right\}\right| \neq f(x). \qquad \square$$

Observe that $\mathsf{C}_{\neq}\mathcal{C} = \mathsf{co}\mathsf{C}_=\mathcal{C}$ where to a complexity class $\mathcal{C}$ we define $\mathsf{co}\mathcal{C}$ as usual by $\mathsf{co}\mathcal{C} = \{L^c \mid L \in \mathcal{C}\}$, where L^c is the complement of L. That is why the quantifier $\mathsf{C}_{\neq}$ is often also written as $\mathsf{co}\mathsf{C}_=$, so $\mathsf{C}_{\neq}\mathsf{P}$ is sometimes called $\mathsf{co}\mathsf{C}_=\mathsf{P}$.

The counting hierarchy CH consists of the languages from all complexity classes that we can get from P by applying the quantifiers $\exists$, $\forall$, C, $\mathsf{C}_=$ and $\mathsf{C}_{\neq}$ a constant number of times.

Remember that an oracle machine M with an oracle L is a Turing machine that during its computation may ask oracle questions to L. To do so, M writes a string o to a special oracle tape and then ask the oracle if $o \in L$. The oracle gives the correct answer to this question in a single step.

For a complexity class $\mathcal{C}$ defined by a class of machines $\mathcal{M}$ and a language L we define $\mathcal{C}^L$ to be the class of languages L' that can be decided by machines from $\mathcal{M}$ with the additional capability of asking

oracle questions to L as defined above. For two complexity classes $\mathcal{C}_1$ and $\mathcal{C}_2$ we define $\mathcal{C}_1^{\mathcal{C}_2}$ as the class of languages that are in $\mathcal{C}^L$ for a language $L \in \mathcal{C}_2$.

Remember that PP is the class of decision problems solvable by an nondeterministic polynomial time Turing machine machine such that, given a "yes" instance, strictly more than half of the computation paths accept, while given a "no" instance, strictly less than half of the computation paths accept.

Observe that with the definitions above $\mathsf{PP} = \mathsf{CP}$. Torán [Tor91] proved that this connection between PP and the counting hierarchy can be extended and that there is a characterization of CH by oracles similar to that of the polynomial hierarchy. We state some such characterizations which we will need later on, followed by other technical lemmas.

Theorem 13.2.2. *[Tor91]* $\mathsf{PP}^{\mathsf{NP}} = \mathsf{C}\exists\mathsf{P}$ *and* $\mathsf{PP}^{\mathsf{PP}} = \mathsf{CCP}$.

Lemma 13.2.3. $\mathsf{CCP} = \mathsf{CC}_{\neq}\mathsf{P}$.

Proof. This is not stated in [Tor91] nor is it a direct consequence, because Torán does not consider the $\mathsf{C}_{\neq}$-operator. It can be shown with similar techniques and we give a proof for completeness.

One inclusion is straightforward: From the definition we directly get $\mathsf{CC}_{\neq}\mathsf{P} \subseteq \mathsf{CP}^{\mathsf{C}_{\neq}\mathsf{P}} \subseteq \mathsf{PP}^{\#\mathsf{P}}$. By binary search we have $\mathsf{PP}^{\#\mathsf{P}} = \mathsf{PP}^{\mathsf{PP}} = \mathsf{CCP}$ where the latter equality is from Theorem 13.2.2.

The other inclusion needs a little more work. Let $L \in \mathsf{CCP}$, there are $A \in \mathsf{P}, f, g \in \mathsf{FP}$ and a polynomial p such that

$$
\begin{aligned}
& x \in L \\
\Leftrightarrow\ & \text{there are more than } f(x) \text{ values } y \in \{0,1\}^{p(|x|)} \text{ such that} \\
& \left|\left\{z \in \{0,1\}^{p(|x|)} \mid (x,y,z) \in A\right\}\right| \geq g(x,y) \\
\Leftrightarrow\ & \text{there are more than } f(x) \text{ values } y \in \{0,1\}^{p(|x|)} \text{ such that} \\
& \forall v \in \{1,\ldots,2^{p(|x|)}\}: \\
& \left|\left\{z \in \{0,1\}^{p(|x|)} \mid (x,y,z) \in A\right\}\right| \neq g(x,y) - v \\
\Leftrightarrow\ & \text{there are more than } 2^{p(|x|)}(2^{p(|x|)}-1) + f(x) \text{ pairs } (x,v) \\
& \text{with } y \in \{0,1\}^{p(|x|)} \text{ and } v \in \{1,\ldots,2^{p(|x|)}]\} \text{ such that} \\
& \left|\left\{z \in \{0,1\}^{p(|x|)} \mid (x,y,z) \in A\right\}\right| \neq g(x,y) - v. \qquad (14)
\end{aligned}
$$

From statement (14) we directly get $L \in \mathsf{CC}_{\neq}\mathsf{P}$ and thus the claim. To see the last equivalence we define

$$r(x,y) := \left|\left\{z \in \{0,1\}^{p(|x|)} \mid (x,y,z) \in A\right\}\right|.$$

Fix x, y, then obviously $r(x,y) \neq g(x,y) - v$ for all but at most one v. It follows that of the pairs (y,v) in the last statement $2^{p(|x|)}(2^{p(|x|)}-1)$

always lead to inequality. So statement (14) boils down to the question how many y there are such that there is no v with $r(x,y) = g(x,y) - v$. We want these to be at least $f(x)$, so we want at least $2^{p(|x|)}(2^{p(|x|)} - 1) + f(x)$ pairs such that $r(x,y) \neq g(x,y) - v$. ■

We will use the following result by Green.

Lemma 13.2.4. *[Gre93]* $\exists \mathsf{C}_{\neq}\mathsf{P} = \mathsf{C}_{\neq}\mathsf{P}$.

The following quantitative result on the distribution of primes by Schönhage is often used in the design of randomized algorithms.

Lemma 13.2.5. *[Sch79] For a large enough constant $c > 0$, it holds that for any integers $n > 0$ and x with $|x| \leqslant 2^{2^n}$ and $x \neq 0$, the number of primes p smaller than 2^{cn} such that $x \not\equiv 0 \mod p$ is at least $2^{cn}/cn$.*

Finally, we use a result on probabilistic complexity class as oracles.

Lemma 13.2.6. *[HO02, p. 81] For every oracle X we have* $\mathsf{PP}^{\mathsf{BPP}^X} = \mathsf{PP}^X$.

14 MONOMIALS IN ARITHMETIC CIRCUITS

14.1 ZERO MONOMIAL COEFFICIENT

We first consider the question of deciding if a single specified monomial occurs in a polynomial. In this problem and others regarding monomials, a monomial is encoded by giving the variable powers in binary.

ZMC **Input:** Arithmetic circuit C, monomial m. **Problem:** Decide if m has the coefficient 0 in the polynomial computed by C.

Theorem 14.1.1. ZMC *is* $\mathsf{C}_=\mathsf{P}$*-complete for both multiplicatively disjoint circuits and formulas.*

Proof. With the help of standard reduction techniques used to show the #P-completeness of the permanent (see for example [AB09]), one defines the following generic $\mathsf{C}_=\mathsf{P}$-complete problem, as mentioned in the introduction.

$\textsc{per}_=$ **Input:** Matrix $A \in \{0,1,-1\}^n$, $d \in \mathbb{N}$. **Problem:** Decide if $\textsc{per}(A) = d$.

Therefore, for the hardness of ZMC, it is sufficient to show a reduction from $\textsc{per}_=$. On input $A = (a_{ij})$ and d we compute the formula $Q := \prod_{i=1}^{n} \left(\sum_{j=1}^{n} a_{ij} Y_j \right)$. It is a classical observation by Valiant [Val79][1] that the monomial $Y_1 Y_2 \dots Y_n$ has the coefficient $\textsc{per}(A)$. Thus the coefficient of the monomial $Y_1 Y_2 \dots Y_n$ in $Q - dY_1 Y_2 \dots Y_n$ is 0 if and only if $\textsc{per}(A) = d$.

We now show that ZMC for multiplicatively disjoint circuits is in $\mathsf{C}_=\mathsf{P}$. Consider a monomial m and a constant free multiplicatively disjoint circuit C, i.e., a multiplicatively disjoint circuit in which the input gates of C are labeled either by a variable, by 1 or by -1. A parse tree T (see Section 9.2) contributes to the monomial m in the output polynomial if, when computing the weight of the tree, we get exactly the powers in m; this contribution has coefficient $+1$ if the number of gates labeled -1 in T is even and it has coefficient -1 if this number is odd. The coefficient of m is thus equal to 0 if and only if the number of trees contributing positively is equal to the number of trees contributing negatively.

1 According to [vzG87] this observation even goes back to [Ham79].

Let us represent a parse tree by a boolean word $\bar{\epsilon}$, by indicating which edges of C appear in the parse tree (the length N of the words is therefore the number of edges in C). Some of these words will not represent a valid parse tree, but this can be tested in polynomial time. Consider the following language L composed of triples $(C, m, \epsilon_0\bar{\epsilon})$ such that:

1. $\epsilon_0 = 0$ and $\bar{\epsilon}$ encodes a valid parse tree of C which contribute positively to m,
2. or $\epsilon_0 = 1$ and $\bar{\epsilon}$ does not encode a valid parse tree contributing negatively to m.

Then the number of $\bar{\epsilon}$ such that $(C, m, 0\bar{\epsilon})$ belongs to L is the number of parse trees contributing positively to m and the number of $\bar{\epsilon}$ such that $(C, m, 1\bar{\epsilon})$ belongs to L is equal to 2^N minus the number of parse trees contributing negatively to m. Thus, the number of $\epsilon_0\bar{\epsilon}$ such that $(C, m, \epsilon_0\bar{\epsilon}) \in L$ is equal to 2^N if and only if the number of trees contributing positively is equal to the number of trees contributing negatively, if and only if the coefficient of m is equal to 0 in C. Because L is in P, ZMC for multiplicatively disjoint circuits is in $\mathsf{C}_=\mathsf{P}$. ■

Theorem 14.1.2. ZMC $\in \mathsf{coRP}^{\mathsf{PP}}$.

In the proof of Theorem 14.1.2 we will use the following problem:

CoeffSLP **Input:** Arithmetic circuit C, monomial m, prime p **Problem:** Compute the coefficient of m in the polynomial computed by C modulo p

Kayal and Saha [KS11] showed an upper bound for the complexity of CoeffSLP.

Theorem 14.1.3. CoeffSLP $\in \mathsf{FP}^{\#\mathsf{P}}$.

Proof of Theorem 14.1.2. Let c be the constant given in Lemma 13.2.5. Consider the following algorithm to decide ZMC given a circuit C of size n and a monomial m, using CoeffSLP as an oracle. First choose uniformly at random an integer p smaller than 2^{cn}. If p is not prime, accept. Otherwise, compute the coefficient a modulo p of the monomial m in C with the help of the oracle and accept if $a \equiv 0 \mod p$. Since $|a| \leq 2^{2^n}$, Lemma 13.2.5 ensures that the above is a correct one-sided error probabilistic algorithm for ZMC. This yields ZMC $\in \mathsf{coRP}^{\text{CoeffSLP}}$. Hence ZMC $\in \mathsf{coRP}^{\mathsf{PP}}$ by Theorem 14.1.3. ■

We now give a result linking the ZMC problem to other questions on polynomials computed by circuits. We define the following problem.

GapMonSLP
Input: Univariate arithmetic circuit C over X, $a, b \in \mathbb{N}$.
Problem: Decide if the polynomial computed by C contains no monomial of the form X^c for $a \leq c \leq b$.

GapMonSLP can be seen as a generalization of the degree problem, called DegSLP in [ABKPM09] (see also Section 14.4). This generalization can actually be shown to be hard as it has the same complexity as ZMC.

In GapMonSLP and all other problems in this Chapter that have integers as inputs we assume that integer input are given in binary. We remark that if one encodes the inputs a and b in unary, then this modified version of GapMonSLP can be solved by an efficient randomized algorithm: To see this, observe that one can in time polynomial in $b|C|$ compute a circuit C' that computes the homogeneous parts of degree i for $a \leq i \leq b$ of the polynomial f computed by C (see for example [Büoo, Lemma 2.14]). Then one can check with the the classical Schwartz-Zippel-DeMillo-Lipton lemma (see for example [AB09]) if all the homogeneous components are zero.

Proposition 14.1.4. GapMonSLP *is equivalent to* ZMC.

Proof. The general case of ZMC easily reduces to ZMC for univariate circuits: we only briefly explain the argument below and refer to [ABKPM09] for further details. Given a circuit C over the variables $X_1, \ldots, X_n$ and a monomial $m = X_1^{d_1} \ldots X_n^{d_n}$, we can compute a circuit C' over Y and a monomial $m' = Y^d$ such that the coefficient of m' in C' is zero if and only if the coefficient of m in C is zero. Indeed, define C' by substituting each variable X_i with Y^{M^i} in C for $M := 2^{|C|} + 1$ and let $d = \sum_{i=1}^{n} d_i M^i$. The coefficient of $m' = Y^d$ in C' is zero if and only if the coefficient of m in C in zero. Since the univariate case of ZMC is a special case of GapMonSLP, this shows that ZMC reduces to GapMonSLP.

For the other direction, consider an instance (C, a, b) of GapMonSLP over the variable X. Let $f' = f \cdot (1 + XY)^{b-a}$ where f is the polynomial computed by C. The circuit C' computing f' has polynomial size since $(1 + XY)^{b-a}$ can be computed with a circuit of size $O(\log b - a)$. Let $f = \sum_l \lambda_l X^l$, then

$$f' = \left(\sum_l \lambda_l X^l \right) \left(\sum_{i=0}^{b-a} \binom{b-a}{i} Y^i X^i \right).$$

It follows that the coefficient $P(Y)$ of X^b in f' is

$$P(Y) = \sum_{i=0}^{b-a} \binom{b-a}{i} \lambda_{b-i} Y^i.$$

Thus $P(Y)$ is the zero polynomial if and only if (C, a, b) is a positive instance of GapMonSLP. Now replace Y in C' with $B := 2^{2^{|C'|^2}}$

(obtained by repeated squaring from 2) to obtain a circuit C''. Note that the polynomial P has at most $2^{|C'|}$ monomials, with coefficients bounded by $2^{2^{2|C'|}}$. From the proof of [ABKPM09, Prop. 2.2], it follows that $P(B) = 0$ if and only if P is the zero polynomial. That is, the coefficient of X^b in C'' is zero if and only if (C, a, b) is a positive instance of GAPMONSLP. ■

14.2 COUNTING MONOMIALS

We now turn to the problem of counting the monomials of a polynomial represented by a circuit.

COUNTMON
Input: Arithmetic circuit C, $d \in \mathbb{N}$.
Problem: Decide if the polynomial computed by C has at least d monomials.

To study the complexity of COUNTMON we will look at what we call extending polynomials. Given two monomials M and m, we say that M is m-extending if $M = mm'$ and m and m' have no common variable. We start by studying the problem of deciding the existence of an extending monomial.

EXISTEXTMON
Input: Arithmetic circuit C, monomial m.
Problem: Decide if the polynomial computed by C contains an m-extending monomial.

Proposition 14.2.1. EXISTEXTMON *is in* $\mathsf{RP}^{\mathsf{PP}}$. *For multiplicatively disjoint circuits it is* $\mathsf{C}_{\neq}\mathsf{P}$*-complete.*

Proof. We first show the first upper bound. So let (C, m) be an input for EXISTEXTMON where C is a circuit in the variables $X_1, \ldots, X_n$. Without loss of generality, suppose that $X_1, \ldots, X_r$ are the variables appearing in m. Let f be the polynomial computed by C and let $d = 2^{|C|}$. Clearly, d is a bound on the degree of the polynomial computed by C. We define $f' = \prod_{i=r+1}^{n}(1 + Y_i X_i)^d$ for new variables Y_i. Clearly, f' can be computed by an arithmetic circuit C' of polynomial size. Let

$$f = \sum_{\alpha} \lambda_\alpha X^\alpha$$

in multi-index notation. Then

$$\begin{aligned} f' &:= \left(\sum_{\alpha} \lambda_\alpha X^\alpha\right)\left(\prod_{i=r+1}^{n}(1 + Y_i X_i)^d\right) \\ &= \left(\sum_{\alpha} \lambda_\alpha X^\alpha\right)\left(\prod_{i=r+1}^{n}\left(\sum_{k_i=0}^{d}\binom{d}{k_i} Y_i^{k_i} X_i^{k_i}\right)\right) \end{aligned}$$

$$\left(\sum_{\alpha} \lambda_\alpha X^\alpha\right) \left(\begin{array}{c} \displaystyle\sum_{0\le k_{r+1}\le d} \left(\prod_{r+1}^{n} \binom{d}{k_i} Y_i^{k_i}\right) X^{k_{r+1}} \dots X^{k_n} \\ \vdots \\ 0\le k_n \le d \end{array} \right)$$

Let $P(Y_{r+1}, \dots, Y_n)$ be the coefficient of $m \prod_{i=r+1}^{n} X_i^d$ in f'. We have that f has an m-extending monomial if $P(Y_{r+1}, \dots, Y_n)$ is not identically 0. Observe that P is not given explicitly but can be evaluated modulo a random prime with an oracle for CoeffSLP. Thus it can be checked if P is identically 0 with the classical Schwartz-Zippel-DeMillo-Lipton lemma (see for example [AB09]). Using that CoeffSLP $\in \mathsf{FP}^{\#\mathsf{P}} = \mathsf{FP}^{\mathsf{PP}}$ by Lemma 14.1.3, we get ExistExtMon $\in \mathsf{RP}^{\mathsf{PP}}$.

The upper bound in the multiplicatively disjoint setting is easier: we can guess an m-extending monomial M and then output the answer of an oracle for the complement of ZMC, to check whether M appears in the computed polynomial. This establishes containment in $\exists \mathsf{C}_{\neq}\mathsf{P}$ which by Lemma 13.2.4 is $\mathsf{C}_{\neq}\mathsf{P}$.

For hardness we reduce to ExistExtMon the $\mathsf{C}_{\neq}\mathsf{P}$-complete problem $\textsc{per}_{\neq}$, i.e., the complement of the $\textsc{per}_{=}$ problem introduced for the proof of Theorem 14.1.1. We use essentially the same reduction constructing a circuit $Q := \prod_{i=1}^{n} \left(\sum_{j=1}^{n} a_{ij} Y_j\right)$. Observe that the only potential extension of $m := Y_1 Y_2 \dots Y_n$ is m itself and has the coefficient $\textsc{per}(A)$. Thus $Q - d Y_1 Y_2 \dots Y_n$ has an m-extension if and only if $\textsc{per}(A) \neq d$. ∎

CountExtMon
Input: Arithmetic circuit C, $d \in \mathbb{N}$, monomial m.
Problem: Decide if the polynomial computed by C has at least d m-extending monomials.

Proposition 14.2.2. CountExtMon *is* $\mathsf{PP}^{\mathsf{PP}}$*-complete.*

Proof. Clearly CountExtMon belongs to $\mathsf{PP}^{\mathsf{ZMC}}$ and thus with Theorem 14.1.2 it is in $\mathsf{PP}^{\mathsf{coRP}^{\mathsf{PP}}}$. Using Lemma 13.2.6 we get membership in $\mathsf{PP}^{\mathsf{PP}}$.

To show hardness, we reduce the canonical $\mathsf{CC}_{\neq}\mathsf{P}$-complete problem $\mathsf{CC}_{\neq}$3-SAT to CountExtMon. With Lemma 13.2.3 the hardness for $\mathsf{PP}^{\mathsf{PP}}$ then follows.

$\mathsf{CC}_{\neq}$3-SAT
Input: 3-CNF-formula $F(\bar{x}, \bar{y})$, $k, \ell \in \mathbb{N}$.
Problem: Decide if there are at least k assignments to $\bar{x}$ such that there are not exactly ℓ assignments to $\bar{y}$ such that F is satisfied.

Let $(F(\bar{x},\bar{y}),k,\ell)$ be an instance for $\mathsf{CC}_{\neq}3\mathsf{SAT}$. Without loss of generality we may assume that $\bar{x}=(x_1,\ldots,x_n)$ and $\bar{y}=(y_1,\ldots,y_n)$ and that no clause contains a variable in both negated and non-negated form. Let $\Gamma_1,\ldots,\Gamma_c$ be the clauses of F.

For each literal u of the variables in $\bar{x}$ and $\bar{y}$ we define a monomial $I(u)$ in the variables $X_1,\ldots,X_n,Z_1,\ldots,Z_c$ in the following way:

$$I(x_i)=X_i\prod_{\{j\mid x_i\in\Gamma_j\}}Z_j,\qquad I(\neg x_i)=\prod_{\{j\mid \neg x_i\in\Gamma_j\}}Z_j,$$

$$I(y_i)=\prod_{\{j\mid y_i\in\Gamma_j\}}Z_j,\qquad I(\neg y_i)=\prod_{\{j\mid \neg y_i\in\Gamma_j\}}Z_j.$$

From these monomials we compute the polynomial f by

$$f:=\prod_{i=1}^{n}(I(x_i)+I(\neg x_i))\prod_{i=1}^{n}(I(y_i)+I(\neg y_i))\,. \tag{15}$$

We fix a mapping *mon* from the assignments of F to the monomials of C: Let $\bar{\alpha}$ be an assignment to $\bar{x}$ and $\bar{\beta}$ be an assignment to $\bar{y}$. We define $mon(\bar{\alpha}\bar{\beta})$ as the monomial obtained in the expansion of f by choosing the following terms. If $\alpha_i=0$, choose $I(\neg x_i)$, otherwise choose $I(x_i)$. Similarly, if $\beta_i=0$, choose $I(\neg y_i)$, otherwise choose $I(y_i)$.

The monomial $mon(\bar{\alpha}\bar{\beta})$ has the form $\prod_{i=1}^{n}X_i^{\alpha_i}\prod_{j=1}^{c}Z_j^{\gamma_j}$, where γ_j is the number of true literals in Γ_j under the assignment $\bar{\alpha}\bar{\beta}$. Then F is true under $\bar{\alpha}\bar{\beta}$ if and only if $mon(\bar{\alpha}\bar{\beta})$ has the factor $\prod_{j=1}^{c}Z_j$. Thus F is true under $\bar{\alpha}\bar{\beta}$ if and only if $mon(\bar{\alpha}\bar{\beta})\prod_{j=1}^{c}\left(1+Z_j+Z_j^2\right)$ has the factor $\prod_{i=1}^{n}X_i^{\alpha_i}\prod_{j=1}^{c}Z_j^3$. We set $f'=f\prod_{j=1}^{c}\left(1+Z_j+Z_j^2\right)$.

Consider an assignment $\bar{\alpha}$ to $\bar{x}$. The coefficient of the monomial $\prod_{i=1}^{n}X_i^{\alpha_i}\prod_{j=1}^{c}Z_j^3$ in C' is the number of assignments $\bar{\beta}$ such that $\bar{\alpha}\bar{\beta}$ satisfies F. Thus we get

$(F(\bar{x},\bar{y}),k,\ell)\in\mathsf{CC}_{\neq}3\mathsf{SAT}$

$\Leftrightarrow$ there are at least k assignments $\bar{\alpha}$ to $\bar{x}$ such that the coefficient of the monomial $\prod_{i=1}^{n}X_i^{\alpha_i}\prod_{j=1}^{c}Z_j^3$ in f' is different from ℓ

$\Leftrightarrow$ there are at least k assignments $\bar{\alpha}$ to $\bar{x}$ such that the monomial $\prod_{i=1}^{n}X_i^{\alpha_i}\prod_{j=1}^{c}Z_j^3$ occurs in $f'':=f'-\ell\prod_{i=1}^{n}(1+X_i)\prod_{j=1}^{c}Z_j^3$

$\Leftrightarrow$ there are at least k tuples $\bar{\alpha}$ such that f'' contains the monomial $\prod_{i=1}^{n}X_i^{\alpha_i}\prod_{j=1}^{c}Z_j^3$

$\Leftrightarrow$ f'' has at least k $(\prod_{j=1}^{c}Z_j^3)$-extending monomials.

Observing that f'' can easily be computed by a depth 4 formula C'' completes the proof. ■

Theorem 14.2.3. CountMon *is* $\mathsf{PP}^{\mathsf{PP}}$*-complete. It is* $\mathsf{PP}^{\mathsf{PP}}$*-hard even for unbounded fan-in formulas of depth* 4*.*

Proof. CountMon can be easily reduced to CountExtMon since the number of monomials of a polynomial is the number of 1-extending monomials. Therefore CountMon is in $\mathsf{PP}^{\mathsf{PP}}$.

To show hardness, we prove that the instances of CountExtMon constructed in Proposition 14.2.2 can be reduced to CountMon in logarithmic space. The idea of the proof is that we make sure that the polynomial for which we count all monomials contains all monomials that are not m-extending. Thus we know how many non-m-extending monomials it contains and we can compute the number of m-extending monomials from the number of all monomials. We could use the same strategy to show in general that CountExtMon reduces to CountMon but by considering the instance obtained in the proof of Proposition 14.2.2 and analyzing the extra calculations below we get hardness for unbounded fanin formulas of depth 4.

So let (C'', k, m) be the instance of CountExtMon constructed in the proof of Proposition 14.2.2, with $m = \prod_{j=1}^{c} Z_j^3$. Let f'' be the polynomial computed by C''. We need to count the monomials in f'' which are of the form $f(X_1, \ldots, X_n) \prod_{j=1}^{c} Z_j^3$. The polynomial f'' is multilinear in the X_i, and the Z_j can only appear with powers in $\{0, 1, 2, 3, 4, 5\}$. So the non-m-extending monomials computed by f'' are all products of a multilinear monomial in the X_i and a monomial in the Z_j where at least one Z_j has a power in $\{0, 1, 2, 4, 5\}$. Fix j, then all monomials that are not m-extending because of Z_j are computed by the polynomial

$$\tilde{f}_j := \left(\prod_{i=1}^{n}(X_i + 1)\right)\left(\prod_{j' \neq j}\sum_{p=0}^{5} Z_{j'}^{p}\right)\left(1 + Z_j + Z_j^2 + Z_j^4 + Z_j^5\right). \quad (16)$$

Thus the polynomial $\tilde{f} := \sum_j \tilde{f}_j$ computes all non-m-extending monomials that f'' can compute. The coefficients of monomials in f'' cannot be smaller than $-\ell$ where ℓ is part of the instance of $\mathsf{CC}_{\neq 3}$SAT from which we constructed (C'', k, m) before. So the polynomial $f^* := f'' + (\ell + 1)\tilde{f}$ contains all non-m-extending monomials that f'' can compute and it contains the same extending monomials. There are $2^n 6^c$ monomials of the form that f'' can compute, only 2^n of which are m-extending, which means that there are $2^n(6^c - 1)$ monomials computed by f^* that are not m-extending. As a consequence, f'' has at least k m-extending monomials if and only if f^* has at least $2^n(6^c - 1) + k$ monomials. Now observing that f^* can be computed by a formula of depth 4 completes the proof. ■

14.3 MULTILINEARITY

In this section we consider the effect of multilinearity on our problems. We will not consider promise problems and therefore for the multilinear variants of our problems we must first check if the computed polynomial is multilinear. We start by showing that this step is not difficult, indeed, it is equivalent to the problem ACIT.

ACIT
Input: Arithmetic circuit C.
Problem: Decide if the polynomial computed by C is the zero polynomial.

CHECKML
Input: Arithmetic circuit C.
Problem: Decide if the polynomial computed by C is multilinear.

Proposition 14.3.1. CHECKML *is equivalent to* ACIT.

Proof. Reducing ACIT to CHECKML is easy: Let f be the polynomial computed by C. Then$X^2 f$ for an arbitrary variable X is multilinear if and only if f is identically zero.

For the other direction the idea is to compute the second derivatives of the polynomial computed by the input circuit and check if they are 0.

So let C be a circuit in the variables $X_1, \ldots, X_n$ that is to be checked for multilinearity. For each i we inductively compute a circuit C_i that computes the second derivative with respect to X_i. To do so, for each gate v in C, the circuit C' has three gates v_i, v_i' and v_i''. The polynomial in v_i is that of v, v_i' computes the first derivative and v_i'' the second. For the input gates the construction is obvious. If v is a $+$-gate with children u and w we have $v_i = u_i + w_i$, $v_i' = u_i' + w_i'$ and $v_i'' = u_i'' + w_i''$. If v is a $\times$-gate with children u and w we have $v_i = u_i w_i$, $v_i' = u_i' w_i + u_i w_i'$ and $v_i'' = u_i'' w_i + 2u_i' w_i' + u_i w_i''$. It is easy to see that the constructed circuit computes indeed the second derivative with respect to X_i.

Next we compute $C' := \sum_{i=1}^{n} Y_i C_i$ for new variables Y_i. We have that the polynomial computed by C' is identically zero if and only if C is multilinear. Also C' can easily be constructed in logarithmic space. ■

Next we show that the problem gets much harder if, instead of asking whether *all* the monomials in the polynomial computed by a circuit are multilinear, we ask whether at least *one* of the monomials is multilinear.

MONML
Input: Arithmetic circuit C.
Problem: Decide if the polynomial computed by C contains a multilinear monomial.

The problem MonML lies at the heart of fast exact algorithms for deciding k-paths by Koutis and Williams [Kou08, Wil09] (although in these papers the polynomials are in characteristic 2 which changes the problem a little). This motivated Chen and Fu [CF10, CF11] to consider monML, to show that it is #P-hard and to give algorithms for its bounded depth version. We provide further information on the complexity of this problem.

Proposition 14.3.2. MonML *is in* $\mathsf{RP}^{\mathsf{PP}}$. *It is* $\mathsf{C}_{\neq}\mathsf{P}$*-complete for multiplicatively disjoint circuits.*

Proof. For the upper bound, let C be the input in variables $X_1, \ldots, X_n$. We set $f' = f \cdot \prod_{i=1}^{n}(1 + X_i Y_i)$ where f is the polynomial computed by C. Then f is multilinear monomial if and only if in f' the coefficient polynomial $P(Y_1, \ldots, Y_n)$ of $\prod_{i=1}^{n} X_i$ is not identically 0. This can be tested as in the proof of Proposition 14.2.1, thus establishing $\text{MonML} \in \mathsf{RP}^{\mathsf{PP}}$.

The $\mathsf{C}_{\neq}\mathsf{P}$-completeness in the multiplicatively disjoint case can be proved in the same way as in Proposition 14.2.1. ■

We now turn to our first problem, namely deciding whether a monomial appears in the polynomial computed by a circuit, in the multilinear setting.

ML-ZMC
Input: Arithmetic circuit C, multilinear monomial m.
Problem: Decide if C computes a multilinear polynomial in which the monomial m has coefficient 0.

Proposition 14.3.3. ML-ZMC *is equivalent to* ACIT.

Proof. We first show that ACIT reduces to ML-ZMC. So let C be an input for ACIT. Allender et al. [ABKPM09] have shown that ACIT reduces to a restricted version of ACIT in which all inputs are -1 and thus the circuit computes a constant. Let C_1 be the result of this reduction. Then C computes identically 0 if and only if the constant coefficient of C_1 is 0. This establishes the first direction.

For the other direction let (C, m) be the input, where C is an arithmetic circuit and m is a monomial using Proposition 14.3.1. First check if m is multilinear, if not output 1 or any other nonzero polynomial. Next we construct a circuit C_1 that computes the homogeneous component of degree $\deg(m)$ of C with the classical method (see for example [Büoo, Lemma 2.14]). Observe that if C computes a multilinear polynomial, so does C_1. We now plug in 1 for the variables that appear in m and 0 for all other variables, call the resulting (constant) circuit C_2. If C_1 computes a multilinear polynomial, then C_2 is zero if and only if m has coefficient 0 in C_1. The end result of the reduction is $C^* := C_2 + ZC_3$ where Z is a new variable and C_3 is a circuit

which is identically 0 iff C computes a multilinear polynomial (obtained via Proposition 14.3.1). C computes a multilinear polynomial and does not contain the monomial m if and only if both C_2 and ZC_3 are identically 0, which happens if and only if their sum is identically zero. ■

In the case of our second problem, counting the number of monomials, the complexity drops down to PP.

ML-CountMon
Input: Arithmetic circuit C, $d \in \mathbb{N}$.
Problem: Decide if the polynomial computed by C is multilinear and has at least d monomials.

Proposition 14.3.4. ML-CountMon *is* PP*-complete (for Turing reductions).*

Proof. We first show ML-CountMon $\in$ PP. To this end, we first use CheckML to check that the polynomial computed by C is multilinear. Then counting monomials can be done in $\mathsf{PP}^{\text{ML-ZMC}}$, and ML-ZMC is in coRP. By Lemma 13.2.6 the class $\mathsf{PP}^{\mathsf{coRP}}$ is simply PP.

For hardness we reduce the computation of the $\{0,1\}$-permanent to ML-CountMon. The proposition then will follow, because the $\{0,1\}$-permanent is #P-complete for Turing reductions. So let A be a 0-1-matrix and $d \in \mathbb{N}$ and we have to decide if $\textsc{per}(A) \geq d$. We get a matrix B from A by setting $b_{ij} := a_{ij}X_{ij}$. Because every entry of B is either 0 or a distinct variable, we have that, when we compute the permanent of B, every permutation that yields a non-zero summand yields a unique monomial. This means that there are no cancellations, so that $\textsc{per}(A)$ is the number of monomials in $\textsc{per}(B)$.

The problem is now that no small circuits for the permanent are known and thus $\textsc{per}(B)$ is not a good input for ML-CountMon. But because there are no cancellations, we have that $\textsc{det}(B)$ and $\textsc{per}(B)$ have the same number of monomials. So take a small circuit for the determinant (for instance the one given in [MV97]) and substitute its inputs by the entries of B. The result is a circuit C which computes a polynomial whose number of monomials is $\textsc{per}(A)$. Observing that the determinant, and thus the polynomial computed by C, is multilinear completes the proof. ■

14.4 UNIVARIATE CIRCUITS

In this section we briefly study decision problesm on univariate, multiplicativelye disjoint circuits. One problem related to ZMC is to compute the degree of a polynomial given by an arithmetic circuit. This problem was first introduced in [ABKPM09] under the name DegSLP.

DEGSLP
Input: Arithmetic circuit C, $d \in \mathbb{N}$.
Problem: Decide if the degree of the polynomial computed by C is smaller than d.

Allender et al. [ABKPM09] also introduced the problem EQUSLP, which is the problem ACIT restricted to circuits with no indeterminates (i.e., computing integers).

EQUSLP
Input: Arithmetic circuit C computing an integer.
Problem: Decide if the integer computed by C is 0.

In the general case, Allender et al. remark that EQUSLP and ACIT are equivalent and they are known to be in coRP. For DEGSLP, the best known upper bound is $\mathsf{coRP}^{\mathsf{PP}}$ [KS11] and it is an open problem to obtain a lower bound better than P which was obtained by Koiran and Perifel [KP07]. For ZMC we have given a $\mathsf{coRP}^{\mathsf{PP}}$ upper bound and a $\mathsf{C}_{=}\mathsf{P}$ lower bound. Finally, we have shown that COUNTMON is $\mathsf{PP}^{\mathsf{PP}}$-complete.

In contrast to these differing complexities, we first show that in the case of univariate, multiplicatively disjoint circuits, all these problems have equivalent complexities (the case of COUNTMON is slightly different and treated after the others).

Proposition 14.4.1. *For univariate multiplicatively disjoint circuits, the problems* DEGSLP, ZMC, EQUSLP *and* ACIT *are equivalent under logspace reductions. This holds in the monotone and in the general case.*

Proof. The proof we give works both in the general case and in the monotone case.

Clearly, EQUSLP is a special case of ACIT and it can also be decided by asking for the constant coefficient in the problem ZMC, or by asking whether the degree is smaller than 0 in DEGSLP. Thus EQUSLP reduces to all other problems of the claim.

We will reduce ACIT, ZMC and DEGSLP to EQUSLP. To do so, we first remark that given a univariate multiplicatively disjoint circuit C of size s, we can construct a multiplicatively disjoint circuit C' of size $O(s^3)$, with $s+1$ output gates computing the coefficients of the polynomial computed by C (the degree of C cannot be greater than s, because C is multiplicatively disjoint). This is done by the classical argument for computing the homogeneous components of a circuit, noting that if we start from a multiplicatively disjoint circuit we get a multiplicatively disjoint circuit. Indeed, for each gate α in C, we have in C' the gates $\alpha_0, \ldots, \alpha_s$ computing the relevant coefficients of the polynomial computed by α. Then if α is an addition gate with arguments β and γ the gate α_i in C' is also an addition gate with arguments β_i and γ_i. If α is a multiplication gate with arguments β and γ

the gate α_i in C' computes $\sum_{k=0}^{i} \beta_k \gamma_{i-k}$. Each product in this sum multiplies a "β" gate with a "γ" gate, so that multiplicative disjointness is maintained in the construction.

It is now easy to show the reductions to EquSLP. In particular, the above construction directly gives the reduction from ZMC to EquSLP.

To reduce ACIT to EquSLP we apply the above construction, then square the outputs $\alpha_0, \ldots, \alpha_s$ and add the results up. The resulting circuit computes 0 if and only if the starting circuit computed the 0 polynomial.

To reduce DegSLP to EquSLP, we just need to check wether all coefficients of degree at least d are 0, which can be done in a way similar to the reduction from ACIT to EquSLP. ■

We now show that, for univariate multiplicatively disjoint circuits, all the problems considered above are complete for LOGCFL in the monotone case and complete for $\mathsf{C}_=$LOGCFL in the general case. We first recall basic facts about these classes.

We assume the reader to be familiar with the basics of Boolean circuit complexity (see e.g. [Vol99]). We only consider circuits in normal form, i.e., for every $\neg$-gate the child is an input gate.

A Boolean circuit is called $\wedge$-*disjoint* if for each $\wedge$-gate v the subcircuits that have the children of v as output-gates are disjoint. LOGCFL is defined as the class of languages $L \subseteq \{0,1\}^*$ accepted by $\wedge$-disjoint, logspace-uniform Boolean circuits [MP08].

We consider the following version of the circuit value problem.

$\wedge$-DISJOINT-CV **Input:** a $\wedge$-disjoint, monotone Boolean circuit C and an input a of C. **Problem:** Decide if C is satisfied by a.

The following observation is apparent.

Observation 14.4.2. *A language $L \subseteq \{0,1\}^*$ is in* LOGCFL *if and only if L can be reduced in logarithmic space to* $\wedge$-DISJOINT-CV.

A function $f : \{0,1\} \to \mathbb{N}$ is defined to be in #LOGCFL if there is a logspace uniform family of multiplicatively disjoint, monotone arithmetic circuits computing f [MP08]. A function $f : \{0,1\} \to \mathbb{Z}$ is defined to be in gapLOGCFL if and only if it is the difference of two functions in #LOGCFL. Finally, a language L is defined to be in $\mathsf{C}_=$LOGCFL if and only if there is a function $f \in$ gapLOGCFL such that $x \in L$ if and only if $f(x) = 0$.

To a monotone Boolean circuit C we assign a monotone arithmetic circuit $ar(C)$ by replacing all $\vee$-gates by $+$-gates and replacing all $\wedge$-gates by $\vee$-gates. Observe that the construction is a bijection, so to a monotone arithmetic circuit C we can assign a monotone Boolean circuit $ar^{-1}(C)$.

We will use the following basic observation:

Observation 14.4.3. *Let C be a monotone Boolean circuit and $a \in \{0,1\}^*$ an input to C. Then a satisfies C if and only if $ar(C)$ on the input a evaluates to an integer strictly greater than 0*

Proposition 14.4.4. *For monotone univariate multiplicatively disjoint circuits, the problems* DegSLP, ZMC, EquSLP, ACIT *and* CountMon *are* LOGCFL*-complete.*

Proof. By Proposition 14.4.1, for all of these problems apart from CountMon, it is enough to show that in the monotone case the problem EquSLP is LOGCFL-complete.

So let C be a multiplicatively disjoint, monotone arithmetic circuit in which all inputs are labeled by 0 and 1. We can interpret C as an arithmetic circuit C' with variable inputs together with a $\{0,1\}$-assignment a. By Observation 14.4.3 we have that C' computes 0 on a if and only if a does not satisfy $ar(C')$. If follows that EquSLP reduces $\wedge$-disjoint-CV and thus it is in LOGCFL.

Again by Observation 14.4.3 $\wedge$-disjoint-CV reduces to EquSLP: Given a monotone Boolean circuit C and an assignment a, we have that a satisfies C if and only if $ar(C)$ computes a nonzero value on input a. Thus EquSLP for monotone, multiplicatively disjoint arithmetic circuits is LOGCFL-hard.

We now show the upper bound for CountMon. Given a monotone, multiplicatively disjoint arithmetic circuit C, computing a univariate polynomial f, and an integer d, we start from the construction given in Proposition 14.4.1 which yields a family of constant free, monotone arithmetic circuit C' computing the coefficients of f. Let $\alpha_0, \ldots, \alpha_s$ be the output gates such that α_i computes the coefficient of X^i. In the circuit $ar^{-1}(C')$ the gate α_i evaluates to 1 if and only if the coefficient of X^i in f is nonzero. Now interpreting the values of α_i in $ar^{-1}(C')$ as binary numbers, we have to accept (C,d) if and only if $\sum_{i=1}^{s} \alpha_i \geq d$. But iterated addition and comparison of binary numbers are in NC_1 (see e.g. [Vol99]), i.e., they can be computed by polynomial size Boolean formulas. Thus we can construct a $\wedge$-disjoint Boolean circuit C'' and an assignment a such that C'' is satisfied by a if and only if f has at least d monomials. Thus CountMon on univariate multiplicative circuits is in LOGCFL.

Finally, the complement of EquSLP trivially reduces to CountMon by asking whether the number of monomials is at least 1, so CountMon is LOGCFL-hard. ■

Proposition 14.4.5. *For univariate multiplicatively disjoint circuits, the problems* DegSLP, ZMC, EquSLP, ACIT *are* $\mathsf{C}_=$LOGCFL*-complete.*

Proof. Once again, by Proposition 14.4.1 we just need to consider EquSLP.

We first show hardness. So let first $f = f_+ - f_-$ for $f_+, f_- \in$ #LOGCFL and $a \in \{0,1\}^*$ an input to f. Then we can in logarithmic

space construct a multiplicatively disjoint arithmetic circuit C that computes $f = f_+ + (-1)f_-$. Labeling the inputs of C by the values of a completes the hardness proof.

For containment, consider a multiplicatively disjoint, constant free arithmetic circuit C with only constant inputs. Let c be the constant computed by C. We interpret C as a multivariate arithmetic circuit C' with an input $a \in \{0,1\}^*$. Let f be the function computed by C'. It is well known that from C' one can construct two monotone, multiplicatively disjoint, arithmetic circuits C_1, C_2 computing integers f_1, f_2, respectively, such that $f = f_1 - f_2$ (see e.g. [Koi12]). Clearly, $f_1, f_2 \in$ #LOGCFL and $f_1(a) - f_2(a) = f(a) = c$. This directly yields the upper bound. ■

Proposition 14.4.6. *For univariate multiplicatively disjoint circuits, the problem* CountMon *is* $\mathsf{C}_=\mathsf{LOGCFL}$*-hard and is in* $\mathsf{L}^{\mathsf{C}_=\mathsf{LOGCFL}}$.

Proof. The hardness follows from Proposition 14.4.5 and the argument given at the end of the proof of Proposition 14.4.4. The upper bound is clear using Proposition 14.4.5 for ZMC, since we only need to add up a small number of answers to ZMC and then compare to d similarly to the proof of Proposition 14.4.4. ■

14.5 CONCLUSION

In this chapter we have strengthened the known connection between the counting hierarchy and arithmetic circuits by showing that natural questions on arithmetic circuits are complete for different classes in CH. We consider it as likely that other questions on arithmetic circuits could be shown to be connected to CH with similar techniques.

Since the conference version of this chapter [FMM12] was published, Mahajan, Rao and Sreenivasaiah [MRS12] analyzed the complexity of several problems considered in this chapter for very restricted arithmetic circuit classes, so-called read-once/twice formulas and branching programs. For these classes the complexity of our problems often but not always drops considerably.

Let us also remark that the techniques from this chapter have found an application in a recent paper by Mittmann, Saxena and Scheiblechner [MSS12]: The notion of degeneracy considered there, to which algebraic independence in positive characteristic can be reduced, is shown to be hard by reduction from ZMC. It would be interesting to see if a similar hardness result can be shown for algebraic independence itself.

Let us close this chapter with some more open questions: The $\mathsf{C}_=\mathsf{P}$ lower bound for ZMC does not match the upper bound of $\mathsf{coRP}^{\mathsf{PP}}$ completely. Can this upper bound be derandomized to show that ZMC is in $\mathsf{C}_=\mathsf{P}$ also in the general case?

DegSLP is in our opinion one of the most puzzling open questions in arithmetic circuit complexity. While it is widely believed to be hard, not even conditional hardness results are known for it. Our contribution to the understanding of DegSLP has been very modest, but we feel that the direction it proposes might be promising. Maybe a better understanding of tractable classes of polynomials computed by restricted classes of circuits will lead to a better understanding of the general problem. So are there any other classes of circuits for which DegSLP is tractable? Are there any multivariate classes? We leave this as an open question.

Question 4. *What is the complexity of* DegSLP *for different classes of arithmetic circuits?*

Part IV

APPENDIX

A

THE PROOFS FOR FRACTIONAL HYPERTREE WIDTH

A.1 TRACTABLE COUNTING

We will use the following theorems.

Theorem A.1.1 ([GM06]). *The solutions of a* CQ*-instance* Φ *with hypergraph* $\mathcal{H}$ *can be enumerated in time* $\|\Phi\|^{\rho^*(\mathcal{H})+O(1)}$.

Theorem A.1.2 ([Mar10]). *Given a hypergraph* $\mathcal{H}$ *and a rational number* $w \geq 1$*, it is possible in time* $\|\mathcal{H}\|^{O(w^3)}$ *to either*

- *compute a fractional hypertree decomposition of* $\mathcal{H}$ *with width at most* $7w^3 + 31w + 7$*, or*
- *correctly conclude that* $\mathrm{fhw}(\mathcal{H}) \geq w$.

We start of with the quantifier free case which we will use as a building block for the more general result later.

Lemma A.1.3. *The solutions of a quantifier free* CQ*-instance* Φ *with hypergraph* $\mathcal{H}$ *can be counted in time* $\|\Phi\|^{\mathrm{fhw}(\mathcal{H})^{O(1)}}$.

Proof. With Theorem A.1.2 we can compute a fractional hypertree decomposition $(\mathcal{T}, (\chi_t)_{t\in T}, (\psi_t)_{t\in T})$ of width at most $k := O(\mathrm{fhw}(\mathcal{H})^3)$. For each bag χ_t we can with Theorem A.1.1 in time $\|\Phi\|^k$ compute all solutions to the CQ-instance $\Phi[\chi_t]$ that is induced by the variables in χ_t. Let these solutions form a new relation $\mathcal{R}_t$ belonging to a new atom ϕ'_t. Then $\bigwedge_{t\in T} \phi'_t$ gives a solution equivalent, acyclic, quantifier free #CQ instance of size $\|\Phi\|^{O(k)}$. Now we can count the solutions with Theorem 3.1.2. ■

Proof of Theorem 5.4.3. This is a minor modification of the proof of Lemma 5.2.1.

Let $\mathcal{H} = (V, E)$ be the hypergraph of Φ. Because of Theorem A.1.2 we may assume that we have a fractional hypertree decomposition $\Xi := (\mathcal{T}, (\chi_t)_{t\in T}, (\psi_t)_{t\in T})$ of width $k' := k^{O(1)}$ of $\mathcal{H}$. For each edge $e \in E$ we let $\varphi(e)$ be the atom of Φ that induces e.

Let $V_1, \ldots, V_m$ be the vertex sets of the components of $\mathcal{H} - S$ and let $V'_1, \ldots, V'_m$ be the vertex sets of the S-components of $\mathcal{H}$. Clearly, $V_i \subseteq V'_i$ and $V'_i - V_i = V'_i \cap S =: S_i$. Let Φ_i be the restriction of Φ to the variables in V'_i and let Ξ_i be the corresponding fractional hypertree decomposition. Then Ξ_i has a tree $\mathcal{T}_i$ that is a subtree of $\mathcal{T}$.

For each Φ_i we construct a new #CQ-instance Φ_i' by computing for each bag $t \in T$ an atom ϕ_t in the variables χ_t that contains the solutions of $\Phi_i[\chi_t]$ that is induced by the variables of χ_t. The decomposition Ξ has width at most k' so this can be done in time $\|\Phi\|^{O(k')}$ by Theorem A.1.1. Obviously Φ_i and Φ_i' are solution equivalent and Φ_i' is acyclic. Furthermore, Φ_i' has only one single S_i-component, because all the vertices in V_i are connected in Φ and thus also in Φ_i'. Let $\mathcal{H}_i$ be the hypergraph of Φ_i', then $\mathcal{H}_i$ has S_i-star size at most ℓ. Thus the vertices in S_i can be covered by at most ℓ edges in $\mathcal{H}_i$ by Lemma 4.1.1.

Now we construct a CQ-instance $(\mathcal{A}'', \phi_i'')$ such that ϕ_i'' is an atomic formula in the variables S_i exactly as in the proof of Lemma 5.2.1.

We now eliminate all quantified variables in Φ. To do so we add the atom ϕ_i'' for $i \in [m]$ and delete all atoms that contain any quantified variable, i.e., we delete all Φ_i'. Call the resulting CQ-instance Φ''. Because $(\mathcal{A}_i'', \phi_i'')$ is solution equivalent to Φ_i', we have that Φ and Φ' are solution equivalent, too.

We now construct a fractional hypertree decomposition of Φ'' by doing the following: we set $\chi_t' = (\chi_t \setminus \bigcup_{i \in I_t} V_i) \cup \bigcup_{i \in I_t} S_i$ for each bag χ_t where $I_t := \{i \mid \chi_t \cap V_i \neq 0\}$. For each bag χ_t we construct a fractional edge cover ψ_t' of χ_t' by setting $\psi_t'(e) := \psi_t(e)$ for all old edges and setting $\psi_t(S_i) = 1$ for $i \in I_t$ where S_i corresponds to the newly added constraint ϕ_i with $\chi_t \cap V_i \neq 0$. The result is indeed a fractional edge cover, because each variable not in any S_i is still covered as before and the variables in S_i are covered by definition of ψ_t. Furthermore, we claim that the width of the cover is at most k'. Indeed, for each $i \in I$ we had for each $v \in V_i$ $\sum_{e \in E : v \in e} \psi(e) \geq 1$. None of these edges appears in the new decomposition anymore. Thus adding the edge S_i with weight 1 does not increase the total weight of the cover. It is now easy to see that doing this construction for all χ_t leads to a fractional hypertree decomposition of Φ' of width at most k'.

Applying Lemma A.1.3 concludes the proof. ■

A.2 COMPUTING INDEPENDENTS SETS

Lemma A.2.1. *The independent sets of a hypergraph $\mathcal{H} = (V, E)$ can be enumerated in time $|\mathcal{H}|^{O(\rho^*_{\mathcal{H}}(V))}$.*

Proof. Let $\mathcal{H} = (V, E)$. We construct a quantifier free CQ-instance $\Phi = (\mathcal{A}, \phi)$ with the hypergraph $\mathcal{H}$. Let V be the variables of Φ, $\{0, 1\}$ the domain and add an atom ϕ_e with relation symbol $\mathcal{R}_e$ and scope e for each $e \in E$. The relation $\mathcal{R}_e^{\mathcal{A}}$ contains all tuples that contain at most one 1 entry. Finally, $\phi := \bigwedge_{e \in E} \phi_e$.

Clearly, Φ has indeed the hypergraph $\mathcal{H}$. Furthermore the solutions of Φ are exactly the characteristic vectors of independent sets of $\mathcal{H}$.

Thus we can enumerate all independent sets of $\mathcal{H}$ in time $|\mathcal{H}|^{O(\rho^*)}$ with Theorem A.1.1. ■

Proof of Lemma 5.4.4 (Sketch). We proceed by dynamic programming along a fractional hypertree decomposition.

In a first step we compute a fractional hypertree decomposition $(\mathcal{T}, (\chi_t)_{t\in T}, (\psi_t)_{t\in T})$ of width $k' = k^{O(1)}$ of $\mathcal{H}$ with Theorem A.1.2. For each bag χ_t we then compute all all independent sets of $\mathcal{H}[\chi_t]$ by Lemma A.2.1; call this set I_t.

By dynamic programming similar to the proof of Lemma 4.2.2 we then compute a maximum independent set of $\mathcal{H}$. ■

BIBLIOGRAPHY

[AB09] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009. (Cited on pages 181, 185, 187, and 189.)

[ABKPM09] E. Allender, P. Bürgisser, J. Kjeldgaard-Pedersen, and P. B. Miltersen. On the Complexity of Numerical Analysis. *SIAM J. Comput.*, 38(5):1987–2006, 2009. (Cited on pages 12, 112, 187, 188, 193, 194, and 195.)

[ABO99] E. Allender, R. Beals, and M. Ogihara. The Complexity of Matrix Rank and Feasible Systems of Linear Equations. *Computational Complexity*, 8(2):99–126, 1999. (Cited on page 179.)

[AGG07] I. Adler, G. Gottlob, and G. Grohe. Hypertree Width and Related Hypergraph Invariants. *Eur. J. Comb.*, 28(8):2167–2181, 2007. (Cited on page 32.)

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995. (Cited on pages 17 and 105.)

[AW93] E. Allender and K. W. Wagner. Counting Hierarchies: Polynomial Time And Constant Depth Circuits. *Current Trends in Theoretical Computer Science*, 40:469–483, 1993. (Cited on page 179.)

[Bar89] D.A. Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC1. *Journal of Computer and System Sciences*, 38(1):150–164, 1989. (Cited on page 161.)

[BCC+05] M. Bauland, P. Chapdelaine, N. Creignou, M. Hermann, and H. Vollmer. An algebraic approach to the complexity of generalized conjunctive queries. In *Theory and Applications of Satisfiability Testing*, pages 30–45. Springer, 2005. (Cited on pages 37 and 105.)

[BCS97] P. Bürgisser, M. Clausen, and M.A. Shokrollahi. *Algebraic complexity theory*, volume 315. Springer, 1997. (Cited on page 110.)

[BDG07] G. Bagan, A. Durand, and G. Grandjean. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *CSL'07, 16th Annual Conference of the EACSL*, volume

4646 of *LNCS*, pages 208–222. Springer, 2007. (Cited on page 3.)

[BG08] M. Bodirsky and M. Grohe. Non-dichotomies in Constraint Satisfaction Complexity. In *ICALP 2008*, pages 184–196, 2008. (Cited on page 73.)

[BK09] I. Briquel and P. Koiran. A Dichotomy Theorem for Polynomial Evaluation. *Mathematical Foundations of Computer Science 2009*, pages 187–198, 2009. (Cited on pages 109, 114, 117, 118, 124, 152, 153, 155, and 156.)

[BKM11] I. Briquel, P. Koiran, and K. Meer. On the expressive power of CNF formulas of bounded tree- and clique-width. *Discrete Applied Mathematics*, 159(1):1–14, 2011. (Cited on pages 117 and 118.)

[BLMW11] P. Bürgisser, J. M. Landsberg, L. Manivel, and J. Weyman. An Overview of Mathematical Issues Arising in the Geometric Complexity Theory Approach to VP;VNP. *SIAM J. Comput.*, 40(4):1179–1209, 2011. (Cited on page 8.)

[BOC92] M. Ben-Or and R. Cleve. Computing Algebraic Formulas Using a Constant Number of Registers. *SIAM J. Comput.*, 21(1):54–58, 1992. (Cited on pages 128, 161, and 162.)

[Bod88] H.L. Bodlaender. NC-Algorithms for Graphs with Small Treewidth. In *International Workshop on Graph-Theoretic Concepts in Computer Science 1988*, pages 1–10, 1988. (Cited on page 26.)

[Bod93] H.L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *STOC 1993*, pages 226–234. ACM, 1993. (Cited on page 25.)

[Bod98] H.L. Bodlaender. A Partial k-Arboretum of Graphs with Bounded Treewidth. *Theor. Comput. Sci.*, 209(1-2):1–45, 1998. (Cited on pages 24 and 26.)

[Bol98] B. Bollobas. *Modern graph theory*. Springer Verlag, 1998. (Cited on page 53.)

[Bre76] R.P. Brent. The complexity of multiple-precision arithmetic. In R P Brent R S Andersson, editor, *The Complexity of Computational Problem Solving*, pages 126–165. Univ. of Queensland Press, 1976. (Cited on page 111.)

[Bul11] A.A. Bulatov. On the CSP Dichotomy Conjecture. In *International Computer Science Symposium in Russia 2011*, pages 331–344, 2011. (Cited on pages 3 and 72.)

[Büro9] P. Bürgisser. On Defining Integers And Proving Arithmetic Circuit Lower Bounds. *Computational Complexity*, 18(1):81–103, 2009. (Cited on pages 12 and 179.)

[Büoo] P. Bürgisser. *Completeness and Reduction in Algebraic Complexity Theory*. Algorithms and computation in mathematics. Springer, Berlin, New York, 2000. (Cited on pages 109, 110, 112, 114, 115, 125, 148, 152, 153, 154, 159, 187, and 193.)

[CC12] J.-Y. Cai and X. Chen. Complexity of counting CSP with complex weights. In *STOC 2012*, pages 909–920, 2012. (Cited on pages 72 and 109.)

[CD05] H. Chen and V. Dalmau. Beyond Hypertree Width: Decomposition Methods Without Decompositions. In *11th International Conference Principles and Practice of Constraint Programming*, pages 167–181, 2005. (Cited on page 5.)

[CD12] H. Chen and V. Dalmau. Decomposing Quantified Conjunctive (or Disjunctive) Formulas. Logic in Computer Science 2012, 2012. (Cited on pages 86 and 105.)

[CF10] Z. Chen and B. Fu. Approximating Multilinear Monomial Coefficients and Maximum Multilinear Monomials in Multivariate Polynomials. In *Conference on Combinatorial Optimization and Applications 2010*, pages 309–323, 2010. (Cited on page 193.)

[CF11] Z. Chen and B. Fu. The Complexity of Testing Monomials in Multivariate Polynomials. In *Conference on Combinatorial Optimization and Applications 2011*, pages 1–15, 2011. (Cited on page 193.)

[CH96] N. Creignou and M. Hermann. Complexity of Generalized Satisfiability Counting Problems. *Inf. Comput.*, 125(1):1–12, 1996. (Cited on page 117.)

[Che12] H. Chen. On the Complexity of Existential Positive Queries. *ArXiv e-prints*, June 2012. (Cited on page 105.)

[CJG08] D. Cohen, P. Jeavons, and M. Gyssens. A Unified Theory of Structural Tractability for Constraint Satisfaction Problems. *Journal of Computer and System Sciences*, 74(5):721 – 743, 2008. (Cited on pages 5, 27, and 35.)

[CM77] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC 1977*, pages 77–90. ACM, 1977. (Cited on pages 2, 7, 37, 45, 89, 96, and 100.)

[CMR01] B. Courcelle, J.A. Makowsky, and U. Rotics. On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. *Discrete Applied Mathematics*, 108(1-2):23–52, 2001. (Cited on pages 10, 117, 145, and 149.)

[Cou90] B. Courcelle. Graph Rewriting: An Algebraic and Logic Approach. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 193–242. 1990. (Cited on page 147.)

[Cou97] B. Courcelle. The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic. In *Handbook of Graph Grammars*, pages 313–400, 1997. (Cited on page 145.)

[DF99] R.G. Downey and M.R. Fellows. *Parameterized complexity*, volume 3. springer New York, 1999. (Cited on page 66.)

[DHK05] A. Durand, M. Hermann, and P.G. Kolaitis. Subtractive reductions and complete problems for counting complexity classes. *Theoretical Computer Science*, 340(3):496–513, 2005. (Cited on page 23.)

[Die05] R. Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, August 2005. (Cited on page 25.)

[DJ04] V. Dalmau and P. Jonsson. The complexity of counting homomorphisms seen from the other side. *Theor. Comput. Sci.*, 329(1-3):315–323, 2004. (Cited on pages 45, 73, 94, and 97.)

[DKV02] V. Dalmau, P.G. Kolaitis, and M.Y. Vardi. Constraint Satisfaction, Bounded Treewidth, and Finite-Variable Logics. In *International Conference on Principles and Practice of Constraint Programming 2002*, pages 310–326, 2002. (Cited on pages 7 and 94.)

[DM11] A. Durand and S. Mengel. The Complexity of Weighted Counting for Acyclic Conjunctive Queries. *CoRR*, abs/1110.4201, 2011. (Cited on pages 12 and 13.)

[DM13] A. Durand and S. Mengel. Structural tractability of counting of solutions to conjunctive queries. In *International Conference on Database Theory 2013*, pages 81–92, New York, NY, USA, 2013. ACM. (Cited on pages 12 and 13.)

[dRA12] N. de Rugy-Altherre. A Dichotomy Theorem for Homomorphism Polynomials. In *MFCS 2012*, pages 308–322, 2012. (Cited on page 114.)

[Dur13] A. Durand. personal communication, 2013. (Cited on page 149.)

[FFG02] J. Flum, M. Frick, and M. Grohe. Query Evaluation via Tree-Decompositions. *J. ACM*, 49(6):716–752, 2002. (Cited on pages 20 and 21.)

[FG04] J. Flum and M. Grohe. The Parameterized Complexity of Counting Problems. *SIAM Journal on Computing*, 33(4):892–922, 2004. (Cited on page 23.)

[FG06] J. Flum and M. Grohe. Parameterized Complexity Theory. *Texts in Theoretical Computer Science. An EATCS Series*, 2006. (Cited on pages 22, 23, 25, 91, 96, 145, and 148.)

[FKL07] U. Flarup, P. Koiran, and L. Lyaudet. On the Expressive Power of Planar Perfect Matching and Permanents of Bounded Treewidth Matrices. In Takeshi Tokuyama, editor, *International Symposium on Algorithms and Computation 2007*, volume 4835 of *Lecture Notes in Computer Science*, pages 124–136. Springer Berlin Heidelberg, 2007. (Cited on pages 10, 110, 117, 145, 150, 154, and 159.)

[FMM12] H. Fournier, G. Malod, and S. Mengel. Monomials in arithmetic circuits: Complete problems in the counting hierarchy. In *STACS 2012*, pages 362–373, 2012. (Cited on pages 13 and 198.)

[FMR08] E. Fischer, J.A. Makowsky, and E.V. Ravve. Counting truth assignments of formulas of bounded tree-width or clique-width. *Discrete Applied Mathematics*, 156(4):511–529, 2008. (Cited on page 117.)

[FV98] T. Feder and M.Y. Vardi. The computational structure of monotone monadic snp and constraint satisfaction: A study through datalog and group theory. *SIAM Journal on Computing*, 28(1):57–104, 1998. (Cited on page 72.)

[GJC94] M. Gyssens, P. Jeavons, and D.A. Cohen. Decomposing Constraint Satisfaction Problems Using Database Techniques. *Artif. Intell.*, 66(1):57–89, 1994. (Cited on page 35.)

[GLS00] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artif. Intell.*, 124(2):243–282, 2000. (Cited on pages 5, 27, 32, 35, and 71.)

[GLS01] G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001. (Cited on pages 33, 117, and 118.)

[GLS02] G. Gottlob, N. Leone, and F. Scarcello. Hypertree Decompositions and Tractable Queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002. (Cited on page 32.)

[GM06] M. Grohe and D. Marx. Constraint Solving via Fractional Edge Covers. In *SODA 2006*, pages 289–298, New York, NY, USA, 2006. ACM. (Cited on pages 76 and 203.)

[GMS09] G. Gottlob, Z. Miklós, and T. Schwentick. Generalized Hypertree Decompositions: NP-Hardness and Tractable Variants. *J. ACM*, 56(6), 2009. (Cited on page 32.)

[GN06] J. Guo and R. Niedermeier. Exact algorithms and applications for Tree-like Weighted Set Cover. *J. Discrete Algorithms*, 4(4):608–622, 2006. (Cited on page 53.)

[GP04] G. Gottlob and R. Pichler. Hypergraphs in Model Checking: Acyclicity and Hypertree-Width versus Clique-Width. *SIAM J. Comput.*, 33(2):351–378, 2004. (Cited on page 159.)

[GR01] C.D. Godsil and G. Royle. *Algebraic graph theory*, volume 8. Springer New York, 2001. (Cited on page 93.)

[Gre93] F. Green. On the Power of Deterministic Reductions to $C_{=}P$. *Theory of Computing Systems*, 26(2):215–233, 1993. (Cited on page 183.)

[Gro07] M. Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *J. ACM*, 54(1), 2007. (Cited on pages 7, 45, 73, 79, 89, and 94.)

[GS09] S. Garg and E. Schost. Interpolation of polynomials given by straight-line programs. *Theor. Comput. Sci.*, 410(27-29):2659–2662, 2009. (Cited on page 180.)

[GSS01] M. Grohe, T. Schwentick, and L. Segoufin. When is the evaluation of conjunctive queries tractable? In *STOC 2001*, pages 657–666. ACM, 2001. (Cited on pages 45, 47, and 79.)

[Ham79] J. Hammond. Question 6001. *Educ. Times*, 32:179, 1879. (Cited on page 185.)

[HO02] L.A. Hemaspaandra and M. Ogihara. *The Complexity Theory Companion*. Texts in theoretical computer science. Springer, Berlin, New York, 2002. (Cited on pages 179 and 183.)

[HP06] J.M. Hitchcock and A. Pavan. Comparing Reductions to NP-Complete Sets. In *ICALP 2006*, pages 465–476, 2006. (Cited on page 114.)

[Jer81] M. Jerrum. *On the Complexity of Evaluating Multivariate Polynomials*. PhD thesis, University of Edinburgh, 1981. (Cited on page 153.)

[JS11] M. Jansen and R. Santhanam. Permanent Does Not Have Succinct Polynomial Size Arithmetic Circuits of Constant Depth. In *ICALP 2011*, pages 724–735, 2011. (Cited on pages 12 and 179.)

[KBvdG11] J. Kwisthout, H. L. Bodlaender, and L. C. van der Gaag. The Complexity of Finding *k*th Most Probable Explanations in Probabilistic Networks. In *International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2011*, pages 356–367, 2011. (Cited on page 179.)

[KI04] V. Kabanets and R. Impagliazzo. Derandomizing Polynomial Identity Tests Means Proving Circuit Lower Bounds. *Computational Complexity*, 13:1–46, 2004. (Cited on page 180.)

[Kin10] S. Kintali. Realizable Paths and the NL vs L Problem. *Electronic Colloquium on Computational Complexity (ECCC)*, 17:158, 2010. (Cited on pages 162 and 169.)

[KM08] P. Koiran and K. Meer. On the expressive power of CNF formulas of bounded tree- and clique-width. In *WG08*, pages 252–263, 2008. (Cited on page 118.)

[Koi12] P. Koiran. Arithmetic circuits: The chasm at depth four gets wider. *Theor. Comput. Sci.*, 448:56–65, 2012. (Cited on pages 11, 161, and 198.)

[Kou08] I. Koutis. Faster Algebraic Algorithms for Path and Packing Problems. In *ICALP 2008*, pages 575–586, 2008. (Cited on page 193.)

[KP07] P. Koiran and S. Perifel. The complexity of two problems on arithmetic circuits. *Theor. Comput. Sci.*, 389(1-2):172–181, 2007. (Cited on pages 12, 112, 180, and 195.)

[KP11] P. Koiran and S. Perifel. Interpolation in Valiant's Theory. *Computational Complexity*, 20(1):1–20, 2011. (Cited on pages 12 and 179.)

[KPZ99] A. Kiayias, A. Pagourtzis, and S. Zachos. Cook reductions blur structural differences between functional complexity classes. In *Panhellenic Logic Symposium*, pages 132–137, 1999. (Cited on page 23.)

[Kre11] S. Kreutzer. Algorithmic meta-theorems. Number 379 in London Mathematical Society Lecture Note Series. Cambridge University Press, 2011. (Cited on page 148.)

[KS11] N. Kayal and C. Saha. On the Sum of Square Roots of Polynomials and related problems. In *IEEE Conference on Computational Complexity 2011*, pages 292–299, 2011. (Cited on pages 112, 186, and 195.)

[KV00] P.G. Kolaitis and M.Y. Vardi. Conjunctive-Query Containment and Constraint Satisfaction. *J. Comput. Syst. Sci.*, 61(2):302–332, 2000. (Cited on pages 2 and 22.)

[Lad75] R.E. Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, 22(1):155–171, 1975. (Cited on pages 72 and 73.)

[Lib04] L. Libkin. *Elements of Finite Model Theory*. EATCS Series. Springer, 2004. (Cited on pages 17 and 145.)

[Lya07] L. Lyaudet. *Graphes et hypergraphes : complexités algorithmique et algébrique*. PhD thesis, École normale supérieure de Lyon, 2007. (Cited on pages 11, 110, 145, and 150.)

[Mah12] M. Mahajan. Algebraic Complexity Classes. In *Proc. Workshop on Complexity and Logic (in celebration of the 60th birthday of Somenath Biswas)*, 2012. (Cited on page 110.)

[Mal07] Guillaume Malod. The Complexity of Polynomials and Their Coefficient Functions. In *IEEE Conference on Computational Complexity 2007*, pages 193–204, 2007. (Cited on page 109.)

[Mar10] D. Marx. Approximating fractional hypertree width. *ACM Trans. Algorithms*, 6(2):29:1–29:17, April 2010. (Cited on pages 73, 105, and 203.)

[Men11] S. Mengel. Characterizing Arithmetic Circuit Classes by Constraint Satisfaction Problems - (Extended Abstract). In *ICALP 2011*, pages 700–711, 2011. (Cited on page 13.)

[Men13] S. Mengel. Arithmetic Branching Programs with Memory. *ArXiv e-prints*, March 2013. (Cited on page 13.)

[Mik08] Z. Miklós. *Understanding Tractable Decompositions for Constraint Satisfaction*. PhD thesis, University of Oxford, 2008. (Cited on page 27.)

[MP08] G. Malod and N. Portier. Characterizing Valiant's algebraic complexity classes. *J. Complexity*, 24(1):16–38, 2008. (Cited on pages 8, 11, 109, 110, 111, 113, 114, 161, 162, 168, and 196.)

[MRS12] M. Mahajan, B. V. Raghavendra Rao, and K. Sreenivasaiah. Identity Testing, Multilinearity Testing, and Monomials in Read-Once/Twice Formulas and Branching Programs. In *MFCS 2012*, pages 655–667, 2012. (Cited on page 198.)

[MSS12] J. Mittmann, N. Saxena, and P. Scheiblechner. Algebraic Independence in Positive Characteristic – A p-adic Calculus. *ArXiv e-prints*, February 2012. (Cited on page 198.)

[Mul12] K.D. Mulmuley. The GCT program toward the P vs. NP problem. *Communications of the ACM*, 55(6):98–107, 2012. (Cited on page 8.)

[MV97] M. Mahajan and V. Vinay. Determinant: Combinatorics, Algorithms, and Complexity. *Chicago J. Theor. Comput. Sci.*, 1997, 1997. (Cited on page 194.)

[Nis91] N. Nisan. Lower bounds for non-commutative computation. In *STOC 1991*, page 418. ACM, 1991. (Cited on page 161.)

[NR95] R. Niedermeier and P. Rossmanith. Unambiguous auxiliary pushdown automata and semi-unbounded fan-in circuits. *Information and Computation*, 118(2):227–245, 1995. (Cited on pages 161, 172, and 173.)

[OPS10] S. Ordyniak, D. Paulusma, and S. Szeider. Satisfiability of Acyclic and Almost Acyclic CNF Formulas. In *Foundations of Software Technology and Theoretical Computer Science 2010*, pages 84–95, 2010. (Cited on page 106.)

[Pap94] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994. (Cited on page 73.)

[PB83] J.S. Provan and M.O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM Journal on Computing*, 12(4):777–788, 1983. (Cited on page 38.)

[PS13] R. Pichler and S. Skritek. Tractable counting of the answers to conjunctive queries. *Journal of Computer and System Sciences*, 2013. (Cited on pages 3, 6, 22, 37, 47, 105, 123, 137, and 138.)

[PSS13] D. Paulusma, F. Slivovsky, and S. Szeider. Model Counting for CNF Formulas of Bounded Modular Treewidth. In *STACS 2013*, pages 55–66, 2013. (Cited on page 106.)

[RS86] N. Robertson and P.D. Seymour. Graph Minors. II. Algorithmic Aspects of Tree-Width. *J. Algorithms*, 7(3):309–322, 1986. (Cited on page 24.)

[Sch78] T.J. Schaefer. The complexity of satisfiability problems. In *STOC 1978*, pages 216–226, 1978. (Cited on pages 3, 72, and 117.)

[Sch79] A. Schönhage. On the Power of Random Access Machines. In *ICALP 1979*, pages 520–529, 1979. (Cited on page 183.)

[SS10] M. Samer and S. Szeider. Algorithms for propositional model counting. *J. Discrete Algorithms*, 8(1):50–64, 2010. (Cited on page 106.)

[Strar] Y. Strozecki. Interpolation Meets Enumeration. *Theory of Computing Systems*, To appear. (Cited on page 180.)

[SV85] S. Skyum and L.G. Valiant. A Complexity Theory Based on Boolean Algebra. *J. ACM*, 32(2):484–502, 1985. (Cited on page 161.)

[SY10] A. Shpilka and A. Yehudayoff. Arithmetic Circuits: A survey of recent results and open questions. *Foundations and Trends in Theoretical Computer Science*, 5(3-4):207–388, 2010. (Cited on page 110.)

[Thu06] M. Thurley. Tractability and Intractability of Parameterized Counting Problems. Diploma thesis, Humbold-Universiät zu Berlin, 2006. (Cited on page 23.)

[Tod92] S. Toda. Classes of arithmetic circuits capturing the complexity of computing the determinant. *IEICE Transactions on Information and Systems*, 75(1):116–124, 1992. (Cited on pages 161 and 162.)

[Tor88] J. Torán. Succinct Representations of Counting Problems. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes 1988*, pages 415–426, 1988. (Cited on page 179.)

[Tor91] J. Torán. Complexity Classes Defined by Counting Quantifiers. *J. ACM*, 38(3):753–774, 1991. (Cited on page 182.)

[Tur50] A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):pp. 433–460, 1950. (Cited on page iii.)

[Val79] L.G. Valiant. Completeness Classes in Algebra. In *STOC 1979*, pages 249–261. ACM, 1979. (Cited on pages 109, 110, 112, 154, 161, and 185.)

[Val81] L.G. Valiant. Universality considerations in VLSI circuits. *IEEE Transactions on Computers*, 30(2):135–140, 1981. (Cited on page 49.)

[Val82] L.G. Valiant. Reducibility by algebraic projections. *Logic and Algorithmic: an International Symposium held in honor of Ernst Specker*, 30(Monogr. No. 30 de l'Enseign. Math.):365–380, 1982. (Cited on pages 109 and 110.)

[Vol99] H. Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. (Cited on pages 196 and 197.)

[VSBR83] L.G. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff. Fast Parallel Computation of Polynomials Using Few Processors. *SIAM J. Comput.*, 12(4):641–644, 1983. (Cited on pages 111, 161, and 172.)

[VT89] H. Venkateswaran and M. Tompa. A New Pebble Game that Characterizes Parallel Complexity Classes. *SIAM J. Comput.*, 18(3):533–549, 1989. (Cited on page 113.)

[vzG87] J. von zur Gathen. Feasible Arithmetic Computations: Valiant's Hypothesis. *Journal of Symbolic Computation*, 4(2):137–172, 1987. (Cited on page 185.)

[Wag86] K. W. Wagner. The Complexity of Combinatorial Problems with Succinct Input Representation. *Acta Informatica*, 23(3):325–356, 1986. (Cited on pages 12, 179, and 181.)

[Wil09] R. Williams. Finding paths of length k in $O^*(2^k)$ time. *Information Processing Letters*, 109(6):315–318, 2009. (Cited on page 193.)

[WS07] V. Weber and T. Schwentick. Dynamic Complexity Theory Revisited. *Theory Comput. Syst.*, 40(4):355–377, 2007. (Cited on page 161.)

[Yan81] M. Yannakakis. Algorithms for Acyclic Database Schemes. In *Proceedings of the seventh international conference on Very Large Data Bases*, volume 7, pages 82–94, 1981. (Cited on pages 29 and 136.)

EIDESSTATTLICHE ERKLÄRUNG

Hiermit versichere ich, dass ich die folgende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen als Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Paderborn, 21. Mai 2013

Stefan Mengel