TU WIEN Informatics

Advanced Tools and Methods for Treewidth-Based Problem Solving

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften (Dr.techn.)

an der Technischen Universität Wien, entsprechend dem akademischen Grad

Doktor der Naturwissenschaften (Dr.rer.nat.)

der Universität Potsdam (Cotutelle de Thèse); eingereicht von

DI Markus Hecher, BSc

an der Fakultät für Informatik der Technischen Universität Wien sowie

an der Mathematisch-Naturwissenschaftlichen Fakultät der Universität Potsdam (Cotutelle de Thèse)

Betreuung: Prof. Dr. Stefan Woltran
Zweitbetreuung: Prof. Dr. Torsten Schaub

Diese Dissertation haben begutachtet:

Mirosław Truszczyński	Heribert Vollmer

Wien, 1. Juni 2021

Markus Hecher

TU WIEN Informatics

Advanced Tools and Methods for Treewidth-Based Problem Solving

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften (Dr.techn.)

at the TU Wien, which corresponds to the degree

Doktor der Naturwissenschaften (Dr.rer.nat.)

at the University of Potsdam (Cotutelle de Thèse); by

DI Markus Hecher, BSc

to the Faculty of Informatics at the TU Wien as well as

to the Faculty of Science at the University of Potsdam (Cotutelle de Thèse)

Advisor: Prof. Dr. Stefan Woltran
Second advisor: Prof. Dr. Torsten Schaub

The dissertation has been reviewed by:

Mirosław Truszczyński	Heribert Vollmer

Vienna, 1st June, 2021

Markus Hecher

Published online on the
Publication Server of the University of Potsdam:
https://doi.org/10.25932/publishup-51251
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-512519

Erklärung zur Verfassung der Arbeit

DI Markus Hecher, BSc

Hiermit erkläre ich eidesstattlich, dass ich diese Arbeit ("Advanced Tools and Methods for Treewidth-Based Problem Solving") selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe. Des Weiteren wurde diese Arbeit an keiner anderen Hochschule als der TU Wien und der Universität Potsdam (Cotutelle de Thèse) eingereicht.

Wien, 1. Juni 2021

Markus Hecher

Acknowledgements

First of all, I would like to express my gratitude to Stefan Woltran, who supported this thesis, always lent me his ears, and gave me useful advice and motivation whenever I needed it. During my research Stefan invariably took his time for my concerns, doubts, questions, and suggestions, and he responded with nothing less than the truth. Further, I am grateful for Torsten Schaub, who assisted me with both personal and professional advice and helped me establishing the Cotutelle de Thèse program. During my stays at the university of Potsdam, he took care that I felt safe, welcome, motivated, and taken seriously.

I would also like to thank Johannes K. Fichte, to whom I often refer to by "JFK", who always believed in me and our common research. He probably unlocked many of my skills and served as a main motivator for me pursuing research. Dear JFK, over the years, you turned into one of my best friends and I am truly grateful for everything you have given without even asking. This brings me to my friend Andreas Pfandler, whose motivation and endless believe in science and research is extremely contagious. Indeed, he infected me with new motivation many times and despite his busy calendar, he always had time for me for a quick chat and some coffee ;). Thank you, Andreas, for your overwhelmingly positive personality, which makes even the worst weather feel like a sunny day. I would also like to thank Michael Morak and Michael Abseher for many productive chats and their support whenever I asked for it. Then, I am also grateful for my beer brewing buddies, Michael Tröstl and Leighton Hanson, who were responsible for us brewing more than 15 different types of beer in the last couple of months (our four batches of the third generation are already fermenting). Thank you, Michael, for always lending me eyes, ears and hands and helping me with many things over the years. You cleared many of my doubts over the years and gifted me with nothing less than the truth. Leighton, I am really grateful for you believing in us brew masters and that you always support our crazy next level brewing recipes, which seem to be not so bad after all.

Last but not least, I would like to thank all of my family and family in-law. However, almost all of my available gratitude goes to Bettina, my wife. Without you, Bettina, nothing would be possible and I can't even imagine how life would be without you. Over the years, we grew so close that I am not quite sure how to express it. Thank you, Bettina for feeling loved and like home next to you, for always believing in me, and for clearing many doubts and concerns that nobody else could have resolved but you.

Without your irreplaceable skills, your enthusiasm and your indestructible positivity for even the smallest things, I could not have finished this thesis. This brings me to my parents, who permanently believed in me and my skills, even when I did not. Thank you, mom and dad, for everything you have given me, be it an important advice or some discussion about a math problem I had in mind. I always have felt safe, warm, welcome and loved at home and I probably could not have hoped for better parents.

Finally, with this line I would like to thank everybody, who contributed to this thesis in any way without being named explicitly. One of these people is for example our beloved administrator Toni Pisjak, who had to go way too early. Toni kept all our clusters and servers at the institute in good shape and took care that everything is smooth and easy for us without a single question or doubt. Thank you very much, we deeply miss you.

Kurzfassung

In den letzten Jahrzehnten konnte ein beachtlicher Fortschritt im Bereich der *Aussagenlogik* verzeichnet werden. Dieser äußerte sich dadurch, dass für das wichtigste Problem in diesem Bereich, genannt „SAT“, welches sich mit der Fragestellung befasst, ob eine gegebene aussagenlogische Formel erfüllbar ist oder nicht, überwältigend schnelle Computerprogramme („Solver“) entwickelt werden konnten. Interessanterweise liefern diese Solver eine beeindruckende Leistung, weil sie oft selbst Probleminstanzen mit mehreren Millionen von Variablen spielend leicht lösen können. Auf der anderen Seite jedoch glaubt man in der Wissenschaft weitgehend an die *Exponentialzeithypothese (ETH)*, welche besagt, dass man im schlimmsten Fall für das Lösen einer Instanz in diesem Bereich exponentielle Laufzeit in der Anzahl der Variablen benötigt. Dieser vermeintliche Widerspruch ist noch immer nicht vollständig geklärt, denn wahrscheinlich gibt es viele ineinandergreifende Gründe für die Schnelligkeit aktueller SAT Solver. Einer dieser Gründe befasst sich weitgehend mit strukturellen Eigenschaften von Probleminstanzen, die wohl indirekt und intern von diesen Solvern ausgenützt werden.

Diese Dissertation beschäftigt sich mit solchen strukturellen Eigenschaften, nämlich mit der sogenannten *Baumweite*. Die Baumweite ist sehr gut erforscht und versucht zu messen, wie groß der Abstand von Probleminstanzen zu Bäumen ist (Baumnähe). Allerdings ist dieser Parameter sehr generisch und bei Weitem nicht auf Problemstellungen der Aussagenlogik beschränkt. Tatsächlich gibt es viele weitere Probleme, die parametrisiert mit Baumweite in polynomieller Zeit gelöst werden können. Interessanterweise gibt es auch viele Probleme in der Wissensrepräsentation (KR), von denen man davon ausgeht, dass sie härter sind als das Problem SAT, die bei beschränkter Baumweite in polynomieller Zeit gelöst werden können. Ein prominentes Beispiel solcher Probleme ist das Problem QSAT, welches sich für die Gültigkeit einer gegebenen quantifizierten, aussagenlogischen Formel (QBF), das sind aussagenlogische Formeln, wo gewisse Variablen existenziell bzw. universell quantifiziert werden können, befasst. Bemerkenswerterweise wird allerdings auch im Zusammenhang mit Baumweite, ähnlich zu Methoden der klassischen Komplexitätstheorie, die tatsächliche Komplexität (Härte) solcher Problemen quantifiziert, wo man die exakte Laufzeitabhängigkeit beim Problemlösen in der Baumweite (Stufe der Exponentialität) beschreibt.

Diese Arbeit befasst sich mit fortgeschrittenen, Baumweite-basierenden Methoden und Werkzeugen für Probleme der Wissensrepräsentation und künstlichen Intelligenz (AI).

Dabei präsentieren wir Methoden, um präzise Laufzeitresultate (*obere Schranken*) für prominente Fragmente der Antwortmengenprogrammierung (ASP), welche ein kanonisches Paradigma zum Lösen von Problemen der Wissensrepräsentation darstellt, zu erhalten. Unsere Resultate basieren auf dem Konzept der dynamischen Programmierung, die angeleitet durch eine sogenannte Baumzerlegung und ähnlich dem Prinzip „Teile-und-herrsche“ funktioniert. Solch eine *Baumzerlegung* ist eine konkrete, strukturelle Zerlegung einer Probleminstanz, die sich stark an der Baumweite orientiert.

Des Weiteren präsentieren wir einen neuen Typ von Problemreduktion, den wir als „*decomposition-guided (DG)*“, also „zerlegungsangeleitet“, bezeichnen. Dieser Reduktionstyp erlaubt es, Baumweiteerhöhungen und -verringerungen während einer Problemreduktion von einem bestimmten Problem zu einem anderen Problem präzise zu untersuchen und zu kontrollieren. Zusätzlich ist dieser neue Reduktionstyp die Basis, um ein lange offen gebliebenes Resultat betreffend quantifizierter, aussagenlogischer Formeln zu zeigen. Tatsächlich sind wir damit in der Lage, präzise *untere Schranken*, unter der Annahme der Exponentialzeithypothese, für das Problem QSat bei beschränkter Baumweite zu zeigen. Genauer gesagt können wir mit diesem Konzept der DG Reduktionen zeigen, dass das Problem QSat, beschränkt auf Quantifizierungsrang ℓ und parametrisiert mit Baumweite k, im Allgemeinen nicht besser als in einer Laufzeit, die ℓ-fach exponentiell in der Baumweite und polynomiell in der Instanzgröße ist[1], lösen. Dieses Resultat hebt auf nicht-inkrementelle Weise ein bekanntes Ergebnis für Quantifizierungsrang 2 auf beliebige Quantifizierungsränge, allerdings impliziert es auch sehr viele weitere Konsequenzen.

Das Resultat über die untere Schranke des Problems QSat erlaubt es, eine *neue Methodologie* zum Zeigen unterer Schranken einer Vielzahl von Problemen der Wissensrepräsentation und künstlichen Intelligenz, zu etablieren. In weiterer Konsequenz können wir damit auch zeigen, dass die oberen Schranken sowie die DG Reduktionen dieser Arbeit unter der Hypothese ETH „eng“ sind, d.h., sie können wahrscheinlich nicht mehr signifikant verbessert werden. Die Ergebnisse betreffend der unteren Schranken für QSat und die dazugehörige Methodologie konstituieren in gewisser Weise eine Hierarchie von über Baumweite parametrisierte Laufzeitklassen. Diese Laufzeitklassen können verwendet werden, um die Härte von Problemen für das Ausnützen von Baumweite zu quantifizieren und diese entsprechend ihrer Laufzeitabhängigkeit bezüglich Baumweite zu kategorisieren.

Schlussendlich und trotz der genannten Resultate betreffend unterer Schranken sind wir im Stande, eine effiziente Implementierung von Algorithmen basierend auf dynamischer Programmierung, die entlang einer Baumzerlegung angeleitet wird, zur Verfügung zu stellen. Dabei funktioniert unser Ansatz dahingehend, indem er probiert, passende *Abstraktionen* von Instanzen zu finden, die dann im Endeffekt sukzessive und auf rekursive Art und Weise verfeinert und verbessert werden. Inspiriert durch die enorme Effizienz und Effektivität der Sat Solver, ist unsere Implementierung ein *hybrider Ansatz*, weil

[1] „ℓ-fache Exponentialität“ meint eine Laufzeitabhängigkeit in der Baumweite k, die einem Turm der Zahlen 2 der Höhe ℓ mit k an der Spitze entspricht. Konkreter sind damit Laufzeiten der Form $\underbrace{2^{\cdot^{\cdot^{2^{\mathcal{O}(k)}}}}}_{\text{Höhe } \ell+1} \cdot \text{poly}(n)$ gemeint, wobei n die Anzahl der Variablen beschreibt.

sie den starken Gebrauch von SAT Solvern zum Lösen diverser Subprobleme, die während der dynamischen Programmierung auftreten, pflegt. Dabei stellt sich heraus, dass der resultierende Solver unserer Implementierung im Bezug auf Effizienz beim Lösen von zwei kanonischen, SAT-verwandten Zählproblemen mit bestehenden Solvern locker mithalten kann. Tatsächlich sind wir im Stande, Instanzen, wo die oberen Schranken von Baumweite 260 übersteigen, zu lösen. Diese überraschende Beobachtung zeigt daher, dass Baumweite ein wichtiger Parameter sein könnte, der wohl in modernen Designs von Solvern berücksichtigt werden sollte.

Abstract

In the last decades, there was a notable progress in solving the well-known *Boolean satisfiability (*Sat*)* problem, which can be witnessed by powerful Sat solvers that are also strikingly fast. On the one hand, these solvers can solve instances with millions of variables, but on the other hand the *exponential time hypothesis (ETH)*, which implies that in the worst case solving time is exponential in the number of variables, is widely believed among researchers. From a scientific point of view, it is still not completely clear why in practice Sat solvers are dealing so well with a large amount of instances, but there are probably many interleaving reasons for this observation. One of these reasons are *structural properties* of instances that are indirectly utilized by the solver's interna.

This thesis deals with such a structural property, which is referred to by treewidth. The *treewidth* is well-studied and measures the closeness of an instance to being a tree (tree-likeness), which is motivated by the fact that many hard problems become easy for the special case of trees or tree-like structures. This parameter, however, is quite generic and by far not limited to Boolean satisfiability. In fact, there are further problems parameterized by treewidth that are solvable in polynomial time in the instance size when parameterized by treewidth. Interestingly, also plenty of problems relevant to knowledge representation and reasoning (KR), which are believed to be even harder than Sat, can be turned tractable when utilizing treewidth. One prominent example of such a problem is QSat, which asks for deciding the validity of a quantified Boolean formula (QBF), an extension of a Boolean formula where certain variables are existentially or universally quantified. Notably, similar to complexity classes in classical complexity, the actual "hardness" of such problems when parameterized by treewidth is oftentimes quantified by studying precise runtime dependence (levels of exponentiality) on treewidth.

In this work, we study advanced treewidth-based methods and tools for problems in KR and artificial intelligence (AI). Thereby, we provide means to establish precise runtime results (*upper bounds*) for prominent fragments of the answer set programming formalism, which is a canonical paradigm for solving problems relevant to KR. Our results are obtained by relying on the concept of dynamic programming that is guided along a so-called tree decomposition in a divide-and-conquer fashion. Such a *tree decomposition* is a concrete structural decomposition of an instance, thereby adhering to the treewidth.

Then, we present a new type of problem reduction, which we call a *decomposition-guided (DG) reduction* that allows us to precisely study and monitor the treewidth increase (or

decrease) when reducing from a certain problem to another problem. This new reduction type will be the basis for proving a long-open result concerning quantified Boolean formulas. Indeed, with this reduction we are able to provide precise *conditional lower bounds* (assuming the ETH) for the problem QSAT when parameterized by treewidth. More precisely, by relying on DG reductions, we prove that QSAT when restricted to formulas of quantifier rank ℓ and treewidth k cannot be decided in a runtime that is better than ℓ-fold exponential in the treewidth and polynomial in the instance size.[2] This non-incrementally lifts a known result for quantifier rank 2 to arbitrary quantifier ranks, but yet implies many further consequences and results.

Even further, the lower bound result for QSAT allows us to design a *new methodology* for establishing lower bounds for a plethora of problems in the area of KR and AI. In consequence, we prove that all our upper bounds and DG reductions presented in this thesis are tight under the ETH. The lower bound result for QSAT and the resulting methodology also unlocks a hierarchy of dedicated runtime classes for problems parameterized by treewidth. These classes can be used to quantify their hardness for utilizing treewidth and categorize them according to their runtime dependence on the treewidth.

Finally, despite the devastating news concerning lower bounds, we are able to provide an efficient implementation of algorithms based on dynamic programming that is guided along a tree decomposition. Our approach works by finding suitable *abstractions* of instances, which is subsequently refined in a nested (recursive) fashion. Given the tremendous power of SAT solvers, our implementation is *hybrid* in the sense that it heavily uses such standard solvers for solving certain subproblems that appear during dynamic programming. It turns out that our resulting solver is quite competitive for two canonical counting problems related to SAT. In fact, we are able to solve instances with treewidth upper bounds beyond 260, which shows that treewidth might be indeed an important parameter that should be considered in modern solver designs.

[2] "ℓ-fold exponentiality" refers to a runtime dependence on the treewidth k that is a tower of 2's of height ℓ with k on top. More precisely, this indicates runtimes of the form $\underbrace{2^{\cdot^{\cdot^{2^{\mathcal{O}(k)}}}}}_{\text{height } \ell+1} \cdot \text{poly}(n)$, where n are the number of variables.

Contents

CHAPTER 1

Introduction

Every brilliant experiment, like every great work of art, starts with an act of imagination.

— Jonah Lehrer

In the last decades, there was a tremendous progress in many areas of computer science. One of these noteworthy advancements concerns the development of efficient decision procedures for problems in the area of *Boolean satisfiability (*Sat*)* [Biere et al., 2009; Garey and Johnson, 1979]. The problem Sat asks to decide whether a given Boolean formula can be satisfied, which is hard [Cook, 1971; Levin, 1973] for the well-studied complexity class NP. Problems that are hard for this class NP are bound to bad news, since such problems can probably not be solved efficiently. This excludes solving procedures for an arbitrary instance of such problems that run in polynomial time in the instance size, unless the class P of problems solvable in polynomial time coincides with the class NP. Over the time, a stronger assumption than P not being equal to NP was proposed, which is referred to by the *Exponential Time Hypothesis (ETH)* [Impagliazzo et al., 2001]. The ETH implies that in the worst case the Sat problem cannot be solved better than in single exponential time in the number of variables of a given formula. Nowadays, this stronger assumption is widely believed among theoreticians, which additionally rules out any decision procedure for Sat that runs in subexponential time in the number of variables.

Nevertheless, despite these downsides, in the past decades there were major breakthroughs in solving the satisfiability question. These breakthroughs originated from seminal research of the 60s [Davis and Putnam, 1960; Davis et al., 1962] and finally resulted in a technique that is referred to by *conflict-driven clause learning (CDCL)* [Silva and Sakallah, 1996; Bayardo and Schrag, 1997]. This technique lead to the development of efficient solvers for Sat (e.g., glucose [Audemard and Simon, 2009] and picosat [Biere, 2008]), capable of solving formulas with millions of variables. So, how is it possible that there is a major gap between theoretical limitations on the one side and efficient solvers

that are still tremendously powerful and fast on the other side? This question is still not completely solved, since Sat solvers consist of additional heuristics and a sophisticated interplay between different components. However, there are certainly connections to the underlying proof system that is actually used in CDCL-based solving [Elffers et al., 2018; Gocht et al., 2019; Kiesl et al., 2020], but also *structural properties* seem to play a role, e.g., [Samer and Szeider, 2009; Ansótegui et al., 2019]. Recently, it was argued and shown empirically that additionally the underlying hardware and optimization towards this hardware is certainly a contributing factor as well [Fichte et al., 2020e].

Still, despite the major progress and although nowadays this efficient solving technology is widely accessible, there was a notable shift towards studying, solving and applying even harder problems [Kleine Büning and Lettman, 1999; Papadimitriou, 1994; Stockmeyer and Meyer, 1973]. One way to witness this, is by a variety of works that rely on extensions of Sat over a range of formalisms and problems like the following:

- Deciding the *validity (*QSat*)* of a *quantified Boolean formula (QBF)* [Biere et al., 2009; Kleine Büning and Lettman, 1999], which is an extension of Sat that additionally allows to existentially or universally quantify over Boolean variables.
- Deciding the *consistency (*Asp*)* of an *answer set program* [Gebser et al., 2012; Brewka et al., 2011; Janhunen and Niemelä, 2016; Schaub and Woltran, 2018; Alviano et al., 2019a], which asks not only for satisfiability, but also concerns about a certain stability condition that requires a justification (proof) for every variable that is claimed to be true. Interestingly, there are many practical applications solvable with this paradigm, e.g., [Gebser et al., 2011; Abels et al., 2019].
- Counting the *number of models* of a Boolean formula (#Sat) [Gomes et al., 2009], which recently fostered different applications and solving techniques [Lagniez and Marquis, 2014; Chakraborty et al., 2016; Dueñas-Osorio et al., 2017; Lagniez et al., 2018; Fichte et al., 2018c; Sharma et al., 2019].

Studying these formalisms then lead to applicability in the context of knowledge representation and reasoning, as well as artificial intelligence in general (see, e.g., [Egly et al., 2000; Truszczyński, 2007]), which resulted in further generalizations and extensions [Eiter et al., 2007; Truszczyński, 2010; Aziz et al., 2015; Shen and Eiter, 2017; Amendola et al., 2019; Cabalar et al., 2020].

However, it turned out that among a list of different problem formalisms, quantified Boolean formulas seem to hold a key role. In particular, the problem QSat, which asks for the validity of a given QBF, serves as a natural, canonical problem in descriptive complexity theory [Grohe, 2017; Immerman, 1999]. Indeed, an encoding using a QBF allows establishing membership for certain complexity classes by using Fagin's results [Fagin, 1974]. Additionally, also hardness of a problem for a certain complexity class can be shown by reducing from QSat to the problem. This has for instance been applied

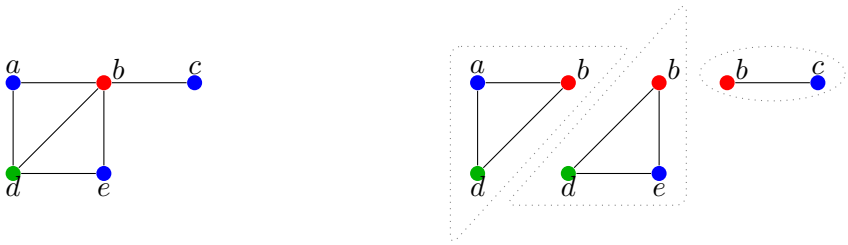

Figure 1.1: A graph (left), whose vertices can be colored with 3 colors such that no two neighboring vertices have the same color, and a corresponding 3-coloring of the graph. This is followed by (right) three individual parts or subgraphs of the graph.

to characterize the hardness of several reasoning problems (e.g., [Eiter and Gottlob, 1995a,b]).

While complexity analysis provides a way to characterize hardness, a thorough, systematic analysis of these hard problems might lead to a more precise picture of so-called "tractable" fragments of problems (e.g., [Schaefer, 1978; Hemaspaandra, 2004; Truszczyński, 2011; Bauland et al., 2011; Bulatov, 2017]). However, there are also other techniques to deal with instances of hard problems. One method to tackle hard instances is by parameterized algorithms, which originate from parameterized complexity [Cygan et al., 2015; Niedermeier, 2006; Downey and Fellows, 2013; Flum and Grohe, 2006], where certain (combinations of) parameters are taken into account. In parameterized complexity, the "hardness" of a problem is classified according to the impact of a *parameter* for solving the problem. Such studies, where the influence of different parameters for solving is systematically analyzed, have been conducted for decision problems [Lonc and Truszczyński, 2003; Lackner and Pfandler, 2012; Meier et al., 2015; Creignou and Vollmer, 2015; Fichte et al., 2019c], but also questions on counting and enumeration [Creignou and Vollmer, 2015; Creignou et al., 2017, 2019] were considered.

As already mentioned above, *structure* plays an important role in SAT solving, but this seems to hold also for other problems. A quite general, structural parameter is called *treewidth*, which was introduced specifically for graph problems [Bertelè and Brioschi, 1969, 1973; Robertson and Seymour, 1983, 1984, 1985, 1986, 1991] and it is very popular in the community of parameterized complexity. Intuitively, this parameter treewidth measures how close a given graph is to being a tree. This is motivated by the observation that problems that are typically rather hard on arbitrary graphs turn out to be simple on trees. In fact, having bounded treewidth is a combinatorial invariant that renders a large variety of NP-hard graph problems tractable [Bodlaender and Koster, 2008; Chimani et al., 2012]. Treewidth then gives rise to the concept of *tree decompositions*, which allow solving numerous NP-hard problems in parts in form of a divide-and-conquer approach, and indicates the maximum number of variables one has to investigate in such parts during evaluation. Among these problems are for example deciding whether a graph has a Hamiltonian cycle, whether a graph is 3-colorable, or determining the number of perfect matchings of a graph [Courcelle et al., 2001].

Example 1.1. *Consider the problem of deciding whether a graph is* 3-colorable*, i.e., whether it can be colored with* 3 *colors such that no two neighbor vertices obtain the same color. This question can be solved in polynomial time on a tree by traversing the (rooted) tree in post-order, which iterates from the leaves towards the root. Thereby, we compute for the current vertex possible colorings while considering computed colorings for the child vertices.*

Now, consider the given graph of Figure 1.1 (left), consisting of vertices a, b, c, d *and* e *and observe that it is* 3*-colorable. A resulting* 3*-coloring is shown, using the colors red, green, and blue. Even for such a graph, a* 3*-coloring can be computed by coloring only parts (subgraphs) of the graph and combining* 3*-colorings for these parts accordingly. These subgraphs are depicted in Figure 1.1 (right). For the given example graph, one could color the subgraphs, which are surrounded by dashed and dotted lines, corresponding to vertices* a, b, d*, vertices* b, d, e *and* b, c *independently. Then, we combine those obtained colorings, which agree on the colors of* b *and* d*. Intuitively, one can create a tree over these three subgraphs, where each subgraph corresponds to a fresh vertex that is part of the tree, which is actually a tree decomposition of the graph. The treewidth of the given graph corresponds to* 2*, which is the largest size of the subgraphs minus one (for technical reasons), since the graph contains completely connected subgraphs over three vertices (e.g., vertices* a, b, c*). For treewidth, such completely connected subgraphs are required to be considered as a whole [Kloks, 1994].*

There is a well-known meta result on treewidth, namely *Courcelle's theorem* [Courcelle, 1990] and its logspace version [Elberfeld et al., 2010], which states that whenever one can encode a problem into a formula in monadic second order logic (MSO), then the problem can be decided in time linear in the input size and some function in the treewidth. While Courcelle's theorem provides a full framework for showing the existence of a tractable algorithm, the obtained results might not be optimal in terms of runtime dependence on the treewidth. In any case, treewidth has been widely employed for important applications beyond graph problems that are defined on more general input structures such as Sat [Samer and Szeider, 2010] and constraint satisfaction [Dechter, 2006; Freuder, 1985], which relies on suitable graph representations of these structures. There is also a well-known result on the correspondence of treewidth and an important measure called resolution width that is applied in Sat solving [Atserias et al., 2011], which stresses once more the essence of structure for problem solving. For treewidth there were also dedicated competitions [Dell et al., 2017], which resulted in notable progresses in utilizing treewidth for Sat [Fichte et al., 2018b, 2019b; Charwat and Woltran, 2019] and other areas [Bannach and Berndt, 2019]. Thereby it turned out that as long as the treewidth is not excessively large, in practice one can actually exploit treewidth in order to solve instances reasonably fast. However, since the runtime dependence on the treewidth is exponential in the worst case, for large treewidth it is expected that this exponential factor outweighs any polynomial factor in the instance size. Still, even problems that are located "beyond NP" such as probabilistic inference [Ordyniak and Szeider, 2013], problems in knowledge representation and reasoning [Gottlob et al., 2010;

Pichler et al., 2010; Dvořák et al., 2012], as well as deciding the validity of QBFs can be turned tractable using treewidth. Notably, to gain tractability of the problem QSAT, we also parameterize by the number of *alternating quantifier blocks (quantifier rank)* [Chen, 2004], since treewidth alone is insufficient [Atserias and Oliva, 2014].

In order to classify problems parameterized by treewidth according to their hardness, one often distinguishes the runtime dependency of the parameter, e.g., levels of exponentiality [Lokshtanov et al., 2011; Marx and Mitsou, 2016] in the treewidth, required for solving the problem. The problem SAT for example can be solved in a runtime that is single exponential in the structural parameter treewidth and polynomial in the number of variables. In other words, for SAT we obtain an *upper bound* in the runtime that is single exponential in the treewidth. This is in contrast to deciding QSAT for QBFs of quantifier rank two, which permits an upper bound that is double exponential[1] in the treewidth k. In the light of utilizing treewidth for solving, the runtime dependence on the treewidth therefore makes a huge difference. For problems that are single exponential in the worst case, we expect that we can solve instances of larger treewidth than for problems that are double exponential. So, a challenging question is, whether a certain runtime dependence on the treewidth can be improved. Interestingly, if we assume that the ETH holds, it turns out that for treewidth neither the single exponential runtime for SAT [Impagliazzo et al., 2001] nor the double exponential runtime for QSAT on QBFs of quantifier rank two [Lampis and Mitsou, 2017] can be significantly improved. Consequently, these *(conditional) lower bounds* depending on the ETH match the available upper bounds. For the canonical QSAT problem when parameterized by treewidth, in general one obtains runtimes that are ℓ-fold exponential in the treewidth [Chen, 2004], where ℓ is the quantifier rank. However, while it was shown that evaluating QSAT has to causes some kind of hierarchy [Pan and Vardi, 2006; Atserias and Oliva, 2014] of these runtimes for treewidth, precise runtime lower bound results were left open, causing an open gap between upper and lower bounds for QSAT.

Contributions and Overview

This thesis deals with treewidth and problems in the area of knowledge representation and reasoning as well as artificial intelligence. We present advanced methods and tools for problems parameterized by treewidth, where we show how to establish parameterized algorithms for treewidth, which results in *precise upper bounds*. Our dedicated upper bounds are obtained via dedicated algorithms that go beyond an initial study of tractability using Courcelle's theorem [Courcelle, 1990], which does not guarantee optimal runtime dependence in the treewidth and therefore might not be suitable for practical applications.

Then, we develop the concept of a *decomposition-guided reduction*, which takes both a problem instance and a tree decomposition and ensures certain guarantees in terms of (tree)width increase or decrease. These decomposition-guided reductions play a

[1] Double exponential runtime refers to $2^{2^{\mathcal{O}(k)}} \cdot \text{poly}(n)$ for treewidth k and number n of variables and ℓ-fold exponentiality refers to a runtime dependence on k that is a tower of 2's of height ℓ with k on top.

crucial role in providing *conditional lower bounds* under the exponential time hypothesis (ETH), which match with our obtained upper bounds that therefore can probably not be significantly improved. More precisely, by relying on a decomposition-guided reduction we solve the open question that asks for a precise lower bound for QSat and treewidth that matches the known upper bound [Chen, 2004]. This long-open result relies on the ETH and together with a suitable decomposition-guided reduction it provides a new *methodology* for showing lower bounds for many problems relevant to this area.

Indeed, our methodology shows that quantified Boolean formulas seem to hold a key role also in parameterized complexity and it therefore confirms that QSat could serve as a canonical problem also for showing hardness results under the ETH. Both the upper bounds together with the corresponding lower bounds form completeness results for novel *treewidth classes* of problems parameterized by treewidth, which we introduce in the course of this thesis.

Further, we also show the usage of graph abstractions and present an efficient implementation of a system nestHDB that uses abstractions of graphs for solving, which is guided along a tree decomposition such that the abstraction is subsequently refined. It turns out that nestHDB is competitive and able to beat state-of-the-art solvers, especially when counting solutions as required for, e.g., model counting (#Sat) and projected model counting (#∃Sat).

In more details, after we discuss preliminaries in Chapter 2, the contributions above are subdivided into the following chapters.

Chap. 3 We establish concrete dynamic programming algorithms for important fragments related to the problem Asp, whose runtimes provide *upper bounds* on the runtime given in form of the dependence on treewidth. While Asp serves as a prototypical problem for these algorithms, the obtained findings can be transferred and further applied to problems related to Sat and QSat as well as extensions thereof. Indeed, these algorithms oftentimes yield precise and tight upper bounds and might provide first insights on the complexity of the corresponding problems for treewidth.

Chap. 4 We establish decomposition-guided reductions, which are reductions that are aware of a certain increase or decrease of (tree)width. These decomposition-guided reductions are then applied for different goals. On the one hand, this allows, e.g., to apply established formalisms like Boolean satisfiability, for which already efficient, treewidth-based solvers exist. On the other hand, our decomposition-guided reductions form a crucial element in establishing a toolkit for showing conditional lower bounds for problems parameterized by treewidth.

Chap. 5 Assuming the Exponential Time Hypothesis (ETH), we establish a meta-result for Quantified Boolean Formulas (QBFs) and treewidth, which expresses that deciding satisfiability (QSat) for a QBF of quantifier rank ℓ requires a runtime that is ℓ-fold exponential in the treewidth. Consequently, this forms a *lower bound* for QSat

under the ETH that matches the known upper bound [Chen, 2004]. We also provide concrete lower bounds under the ETH for Asp and treewidth that match with the corresponding upper bounds. Interestingly, there are certain fragments of Asp that do not admit a single exponential runtime lower bound. Therefore, for example the prominent fragment of Normal Asp, i.e., Asp restricted to normal programs, can be considered slightly harder than Sat under the ETH. This is contrast to classical complexity, where both Sat and Normal Asp are NP-complete.

Chap. 6 The previous two contributions together complete a complexity landscape for a series of problems and formalisms under bounded treewidth. Thereby, we apply the toolkit for showing lower bounds under the ETH as follows: We present decomposition-guided reductions from QSat to several core problems relevant to knowledge representation and reasoning as well as artificial intelligence in general, and apply the meta result above, which establishes hardness results under the ETH. This gives rise to a novel hierarchy of treewidth classes of problems parameterized by treewidth. Together with the upper bounds of the first contribution, the obtained hardness results form completeness for certain treewidth classes of the hierarchy.

Chap. 7 Towards practical applications and systems for problems parameterized by treewidth, we discuss a particular technique that relies on the concept of abstractions. This concept aims at finding certain graph abstractions, which is then subject to decomposition. During solving on the tree decomposition of the obtained abstraction, this abstraction is subsequently refined resulting in nested dynamic programming. This approach is combined with hybrid solving, which refers to the usage of existing, standard solvers for subproblems. Finally, we provide an efficient implementation of a system called nestHDB that relies on nested dynamic programming. The implementation uses relational database systems to manage and maintain dynamic programming, which is then combined with hybrid solving. In order to solve a problem with nestHDB, one needs to specify a (nested) dynamic programming algorithm by means of relational algebra (SQL). It turns out that common database management systems like Postgres [PostgreSQL Global Development Group, 2021] are sufficient for nestHDB to achieve competitive performance for variants of model counting (#Sat).

Chap. 8 Finally, we discuss and summarize the findings of this thesis and put it into perspective of related work. Before concluding, we present consequences of the results and the outcomes of this thesis, which give rise to plenty of future work.

As a high-level overview of the contributions of Chapters 3–6, we briefly present Table 1.1, which highlights runtime upper bounds and lower bounds for treewidth that are discussed in the course of this thesis. This table will be extended, stated more precisely, and completed in Chapter 6, which shows the full extent of the theoretical results and consequences obtained due to the advances of this work (cf. Table 6.1). Later, Chapter 7 tackles efficient implementation techniques of concepts discussed in Chapters 3 and 4, which are then applied, benchmarked and discussed.

Problem	Runtime dependence on treewidth k			
	Exponentiality	Runtime*	Upper Bound	Lower Bound
Sat, #Sat	single exponential	$2^{\Theta(k)}$	△[Samer and Szeider, 2010]	▽[Impagliazzo et al., 2001]
#∃Sat	double exponential	$2^{2^{\Theta(k)}}$	△[Fichte et al., 2018b]	▽[Fichte et al., 2018b]
Tight Asp	single exponential	$2^{\Theta(k)}$	▲ Thm. 3.8	▼ Prop. 3.9
Supported Models	single exponential	$2^{\Theta(k)}$	▲ Thm. 3.8	▼ Prop. 3.9
Normal/HCF Asp	single+ exponential	$2^{\Theta(k\cdot\log(k))}$	▲ Thm. 3.16	▼ Thm. 5.35
ι-Tight Asp	single+ exponential	$2^{\Theta(k\cdot\log(\iota))}$	▲ Thm. 4.27	▼ Corr. 4.28
Asp	double exponential	$2^{2^{\Theta(k)}}$	▲ Thm. 3.21	▼ Thm. 5.18
Disjunctive Asp	double exponential	$2^{2^{\Theta(k)}}$	△[Jakl et al., 2009]	▼ Thm. 5.18
QSat, quantifier rank ℓ	ℓ-fold exponential	$\mathsf{tower}(\ell, \Theta(k))$	△[Chen, 2004]	▼ Thm. 5.7
#Σ_ℓSat, quantifier rank ℓ	(ℓ+1)-fold exponential	$\mathsf{tower}(\ell{+}1, \Theta(k))$	△[Fichte and Hecher, 2020]	▼ Corr. 6.2

Table 1.1: Overview of runtime results (upper bounds) and hardness results (lower bounds) for problems parameterized by the treewidth of the corresponding primal graph representation, cf. Section 2.5. The column "Runtime*" does not give factors that are polynomial in the instance size. The function $\mathsf{tower}(\ell, k)$ is a tower of iterated exponentials of 2 of height ℓ with k on top, cf. Chapter 2. Known upper bounds are indicated by "△", whereas new runtime results established in the course of this thesis are marked with "▲". Runtime lower bounds are given under the assumption that the ETH holds, where "▼" indicates new lower bounds and "▽" refers to a known result.

Overall, the following publications form the basis of this thesis[2].

1. **[Fichte et al., 2017a]**: Fichte, J. K., Hecher, M., Morak, M., and Woltran, S. (2017a). Answer Set Solving with Bounded Treewidth Revisited. In *14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 10377 of *Lecture Notes in Computer Science*, pages 132–145. Springer

2. **[Fichte and Hecher, 2018]**: Fichte, J. K. and Hecher, M. (2018). Exploiting Treewidth for Counting Projected Answer Sets. In *16th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 639–640. AAAI Press

3. **[Fichte and Hecher, 2019]**: Fichte, J. K. and Hecher, M. (2019). Treewidth and Counting Projected Answer Sets. In *15th International Conference on Logic*

[2]For the technical contributions concerning algorithms for (projected) counting [Fichte et al., 2017a; Fichte and Hecher, 2018, 2019] as well as the lower bound results [Fichte et al., 2020c; Hecher, 2020; Fandinno and Hecher, 2021] I draw myself responsible. Especially the implementations of those works presenting systems [Fichte et al., 2017a, 2021b; Hecher et al., 2020b] are the result of a joint effort.

Programming and Nonmonotonic Reasoning (LPNMR), volume 11481 of *Lecture Notes in Computer Science*, pages 105–119. Springer

4. **[Fichte et al., 2020c]**: Fichte, J. K., Hecher, M., and Pfandler, A. (2020c). Lower Bounds for QBFs of Bounded Treewidth. In *35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 410–424. ACM

5. **[Hecher, 2020]**: Hecher, M. (2020). Treewidth-aware Reductions of Normal ASP to SAT - Is Normal ASP Harder than SAT after All? In *17th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 485–495. Winner of the Marco Cadoli Best Student Paper Award

6. **[Fandinno and Hecher, 2021]**: Fandinno, J. and Hecher, M. (2021). Treewidth-Aware Complexity in ASP: Not all Positive Cycles are Equally Hard. In *35th AAAI Conference on Artificial Intelligence (AAAI)*. In Press.

7. **[Fichte et al., 2021b]**: Fichte, J. K., Hecher, M., Thier, P., and Woltran, S. (2021b). Exploiting Database Management Systems and Treewidth for Counting. *Theory and Practice of Logic Programming*. In Press.

8. **[Hecher et al., 2020b]**: Hecher, M., Thier, P., and Woltran, S. (2020b). Taming High Treewidth with Abstraction, Nested Dynamic Programming, and Database Technology. In *23rd International Conference on Theory and Applications of Satisfiability Testing SAT*, volume 12178 of *Lecture Notes in Computer Science*, pages 343–360. Springer

Thereby, Chapter 3 comprises of earlier works [Fichte et al., 2017a; Fichte and Hecher, 2019]. Then, Chapter 4 is mainly based on recent studies [Hecher, 2020; Fandinno and Hecher, 2021]. Our work on lower bounds in Chapter 5 also discusses recent findings [Fichte et al., 2020c; Hecher, 2020]. This results then in Chapter 6, which establishes a new methodology [Fichte et al., 2020c] for showing lower bounds for treewidth and novel treewidth classes. Finally, in Chapter 7, we present results of works [Fichte et al., 2021b; Hecher et al., 2020b].

In the following we also list selected works, which are related and were published in the course of this research but do not form a substantial contribution to this thesis.

- **[Fichte et al., 2017b]**: Fichte, J. K., Hecher, M., Morak, M., and Woltran, S. (2017b). DynASP2.5: Dynamic Programming on Tree Decompositions in Action. In *12th International Symposium on Parameterized and Exact Computation (IPEC)*, volume 89 of *Leibniz International Proceedings in Informatics*, pages 17:1–17:13. Dagstuhl Publishing

- **[Fichte et al., 2018b]**: Fichte, J. K., Hecher, M., Morak, M., and Woltran, S. (2018b). Exploiting treewidth for projected model counting and its limits. In *21st*

International Conference on Theory and Applications of Satisfiability Testing (SAT), volume 10929 of *Lecture Notes in Computer Science*, pages 165–184. Springer

- [**Fichte et al., 2019b**]: Fichte, J. K., Hecher, M., and Zisser, M. (2019b). An Improved GPU-Based SAT Model Counter. In *25th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 11802 of *Lecture Notes in Computer Science*, pages 491–509. Springer

- [**Alviano et al., 2019b**]: Alviano, M., Dodaro, C., Fichte, J. K., Hecher, M., Philipp, T., and Rath, J. (2019b). Inconsistency Proofs for ASP: The ASP - DRUPE Format. *Theory and Practice of Logic Programming*, 19(5-6):891–907

- [**Fichte et al., 2020d**]: Fichte, J. K., Hecher, M., and Schindler, I. (2020d). Default logic and bounded treewidth. *Information and Computation*. In Press

- [**Hecher et al., 2020a**]: Hecher, M., Morak, M., and Woltran, S. (2020a). Structural Decompositions of Epistemic Logic Programs. In *34th AAAI Conference on Artificial Intelligence (AAAI)*, pages 2830–2837. AAAI Press

- [**Fichte and Hecher, 2020**]: Fichte, J. K. and Hecher, M. (2020). Counting with Bounded Treewidth: Meta Algorithm and Runtime Guarantees. In *18th International Workshop on Non-Monotonic Reasoning (NMR)*, pages 9–18

- [**Fichte et al., 2020b**]: Fichte, J. K., Hecher, M., and Kieler, M. F. I. (2020b). Treewidth-Aware Quantifier Elimination and Expansion for QCSP. In *26th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 12333 of *Lecture Notes in Computer Science*, pages 248–266. Springer

- [**Fichte et al., 2021a**]: Fichte, J. K., Hecher, M., and Meier, A. (2021a). Knowledge-Base Degrees of Inconsistency: Complexity and Counting. In *35th AAAI Conference on Artificial Intelligence (AAAI)*. In Press.

During the research, we participated in the organization of two competitions, which resulted in the following competition reports.

- [**Dzulfikar et al., 2019**]: Dzulfikar, M. A., Fichte, J. K., and Hecher, M. (2019). The PACE 2019 Parameterized Algorithms and Computational Experiments Challenge: The Fourth Iteration (Invited Paper). In *14th International Symposium on Parameterized and Exact Computation (IPEC)*, volume 148 of *Leibniz International Proceedings in Informatics*, pages 25:1–25:23. Dagstuhl Publishing

- [**Fichte et al., 2020a**]: Fichte, J. K., Hecher, M., and Hamiti, F. (2020a). The Model Counting Competition 2020. *CoRR*, abs/2012.01323

Besides being involved in the program committee and senior program committee of one of the top conferences of this area, we co-organized the following international workshops, whose focus is related to the topic of this thesis.

- 2nd Workshop on Trends and Applications of Answer Set Programming (TAASP 2018): `http://www.kr.tuwien.ac.at/events/taasp18/pc.html`
- 13th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2020) **[Dodaro et al., 2020]**: `https://sites.google.com/site/aspocp2020/`
- 14th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2021): `https://sites.google.com/view/aspocp2021`
- 1st Workshop on Model Counting (MCW 2020): `https://mccompetition.org/2020/mcw_program`
- Workshop on Counting and Sampling (MCW 2021): `https://mccompetition.org/2021/mcw_description`

Finally, in the course of this research, we participated as an assisting advisor in the following theses.

- [Besin, 2020]: Besin, V. (2020). Advancing a System for Counting Problems based on DBMS and Tree Decompositions. Bachelor's Thesis, Faculty of Informatics, TU Wien, Austria
- [Kieler, 2020]: Kieler, M. F. I. (2020). Trading Structural Dependency for Quantifier Depth on QBFs. Bachelor's Thesis, Faculty of Informatics, TU Dresden, Germany
- [Schidler, 2018]: Schidler, A. (2018). A solver for the Steiner tree problem with few terminals. Master's thesis, Faculty of Informatics, TU Wien, Austria
- [Zisser, 2018]: Zisser, M. (2018). Solving the #SAT problem on the GPU with dynamic programming and OpenCL. Master's thesis, Faculty of Informatics, TU Wien, Austria

CHAPTER 2

Preliminaries

"Obvious" is the most dangerous word in mathematics.

— Eric T. Bell

For a set X, let 2^X be the *power set of* X. Further, we let $\mathbb{N}$ contain all non-negative integers and $\mathbb{N}^+$ be the set of integers, i.e., $\mathbb{N}^+ := \mathbb{N} \setminus \{0\}$. The function $\mathsf{tower}(\ell, k)$ is a tower of iterated exponentials of 2 of height ℓ with k on top. More precisely, we inductively define for some non-negative integer k the function $\mathsf{tower} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ by $\mathsf{tower}(0, k) = k$ and $\mathsf{tower}(\ell + 1, k) = 2^{\mathsf{tower}(\ell,k)}$ for all $\ell \in \mathbb{N}$. The domain $\mathcal{D}$ of a function $f : \mathcal{D} \to \mathcal{A}$ is given by $\mathsf{dom}(f)$. By $f^{-1} : \mathcal{A} \to \mathcal{D}$ we denote the inverse function $f^{-1} := \{f(d) \mapsto d \mid d \in \mathsf{dom}(f)\}$ of a given function f, if it exists. To permit operations such as $f \cup g$ for functions f and g, the functions may be viewed as relations. We use the symbol "$\cdot$" as placeholder for a value of an argument, which is clear from the context and the actual value is negligible. Throughout this work, we refer by $\log(\cdot)$ to the binary logarithm.

2.1 Graph Theory

We recall some graph theoretical notations. For further basic terminology on graphs and digraphs, we refer to standard texts [Diestel, 2012; Bondy and Murty, 2008]. An *undirected graph* or simply a *graph* is a pair $G = (V, E)$ where $V \neq \emptyset$ is a set of *vertices* and $E \subseteq \{\{u, v\} \subseteq V \mid u \neq v\}$ is a set of *edges*. For a vertex $v \in V$, we call any $u \in V$ such that there is an edge $\{u, v\} \in E$ a *neighbor* of v. We define further for v the *degree (of v)*, which is the *number of neighbors* of v and corresponds to $|\{u \mid \{u, v\} \in E\}|$. A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$ and an *induced subgraph* if additionally for any $u, v \in V'$ and $\{u, v\} \in E$ also $\{u, v\} \in E'$. A *path of length k* is a sequence of $k + 1$ pairwise distinct vertices $v_1, \ldots, v_{k+1}$ such that there are k distinct edges $\{v_i, v_{i+1}\} \in E$ where $1 \leq i \leq k$ (possibly $k = 0$). A *cycle of length k* is a path $v_1, v_2, \ldots, v_{k+1}$ of length k such that additionally $\{v_{k+1}, v_1\} \in E$. Let $G = (V, E)$

be a graph and $A \subseteq V$ be a set of vertices. We define the *subgraph* $G - A$, which is the graph obtained by removing vertices A, by $G - A := (V \setminus A, \{e \mid e \in E, e \cap A = \emptyset\}$. Then, G is *bipartite* if the set V of vertices can be divided into two disjoint sets U and W such that there is no edge $\{u, v\} \in E$ with $u, v \in U$ or $u, v \in W$. Graph G is *complete* if for any two vertices $u, v \in V$ there is an edge $uv \in E$. G contains a *clique* on $V' \subseteq V$ if the induced subgraph (V', E') of G is a complete graph. A *(connected) component* $C \subseteq V$ of G is a $\subseteq$-largest set such that for any two vertices $u, v \in C$ there is a path from u to v in G. We say G is a *tree* if it is a connected component and G contains no cycles. We usually call the vertices of a tree *nodes*.

A *directed graph* or simply a *digraph* is a pair $G = (V, E)$ where $V \neq \emptyset$ is a set of vertices and $E \subseteq \{(u, v) \in V \times V \mid u \neq v\}$ is a set of *directed edges*. A digraph $G' = (V', E')$ is a *subdigraph* of G if $V' \subseteq V$ and $E' \subseteq E$ and an *induced subdigraph* if additionally for any $u, v \in V'$ and $(u, v) \in E$ also $(u, v) \in E'$. For a vertex $v \in V$ we call a vertex $v \in \{w \mid (w, u) \in E\}$ a *predecessor of* u and a vertex $u \in \{w \mid (v, w) \in E\}$ a *successor of* v. A *(directed) path of length* k is a sequence of $k + 1$ pairwise distinct vertices $v_1, \ldots, v_{k+1}$ such that there are k distinct edges $(v_i, v_{i+1}) \in E$ where $1 \leq i \leq k$ (possibly $k = 0$). A *(directed) cycle of length* k is a path $v_1, v_2, \ldots, v_{k+1}$ of length k such that additionally $(v_{k+1}, v_1) \in E$. Then, a vertex $u \in V$ is a *descendant* of a vertex $v \in V$ if there is a directed path $u, \ldots, v$ from u to v in G. For directed graph $G = (V, E)$, a set $C \subseteq V$ of vertices is a *(strongly-connected) component (SCC)* of G if C is a $\subseteq$-largest set such that for every two distinct vertices u, v in C there is a directed path from u to v in G. An SCC C is called *non-trivial*, if $|C| > 1$.

A *rooted tree* $T = (V, E)$ consists of (i) a digraph (V, E) whose underlying graph is a tree and (ii) a designated root vertex which has no predecessor. We call a vertex $v \in V$ *node of* T, a successor of a node *child*, the vertex $\text{root}(T)$ the *root of* T, and a vertex v that has no child a *leaf of* T. For every node t of T, we denote by $\mathsf{children}(t)$ the *set of child nodes of* t in T. Further, for a given node t of T, we let $T[t]$ be the *subtree rooted at* t, which is a rooted tree with $\text{root}(T[t]) := t$, whose digraph is an induced subdigraph of T obtained by restricting T to t and all descendants of t in T. In this work, we will traverse trees bottom-up, therefore, we let $\mathsf{post\text{-}order}(T)$ be the sequence of nodes in *post-order (bottom-up-order)* of the tree T, which is any fixed sequence of nodes in T such that for every node t' in T, nodes in $\mathsf{children}(t')$ precede t' in the sequence. Then, for every node t of T we denote by $\mathsf{post\text{-}children}(t)$ *the ordered sequence of child nodes of* t obtained by ordering all nodes in $\mathsf{children}(t)$ according to their order in $\mathsf{post\text{-}order}(T)$.

Let $G = (V, E)$ be a graph and k a positive integer. We call a function $c : V \to \{1, \ldots, k\}$ *k-coloring* of G if $c(v) \neq c(w)$ for all $\{v, w\} \in E$. Then, the problem k-Col asks to decide whether there exists a k-coloring for a given graph. As already discussed, Figure 1.1 visualizes a graph and a 3-coloring of the graph.

2.2 Computational Complexity

We assume familiarity with standard notions in computational complexity, especially, algorithms, decision, counting, and search problems, the complexity classes P and NP as well as the polynomial hierarchy. For more detailed information, we refer to standard sources [Papadimitriou, 1994]. For counting complexity we follow notions by Hemaspaandra and Vollmer [1995]. For parameterized (decision) problems we refer to standard sources [Cygan et al., 2015; Downey and Fellows, 2013; Flum and Grohe, 2006; Niedermeier, 2006].

We use the asymptotic notations $\mathcal{O}(\cdot)$, $o(\cdot)$, and $\Omega(\cdot)$ in the standard way. Let Σ and Σ' be some finite alphabets. We call $\mathcal{I} \in \Sigma^*$ an *instance* and $\|\mathcal{I}\|$ denotes the size of $\mathcal{I}$. Let $L \subseteq \Sigma^*$ and $L' \subseteq \Sigma'^*$ be decision problems. A *(non-deterministic) polynomial-time Turing reduction* from L to L' is an (non-deterministic) algorithm that decides in time $\mathcal{O}(\text{poly}(\|\mathcal{I}\|))$ whether $\mathcal{I} \in L$ using L' as an oracle. A *polynomial-time (many-to-one) reduction* from L to L' is a function $r : \Sigma^* \to \Sigma'^*$ such that for all $\mathcal{I} \in \Sigma^*$ we have $\mathcal{I} \in L$ if and only if $r(\mathcal{I}) \in L'$ and r is computable in time $\mathcal{O}(\text{poly}(\|\mathcal{I}\|))$. In other words, a polynomial-time reduction transforms instances of decision problem L into instances of decision problem L' in polynomial time.

A decision problem L is *(non-deterministically) polynomial-time solvable* if there exists a constant c such that we can decide by an (non-deterministic) algorithm whether $\mathcal{I} \in L$ in time $\mathcal{O}(\|\mathcal{I}\|^c) = \mathcal{O}(\text{poly}(\|\mathcal{I}\|))$. P is the class of all polynomial-time solvable decision problems. NP is the class of all non-deterministically polynomial-time solvable decision problems. Let C be a decision complexity class, e.g., NP. Then co-C denotes the class of all decision problems whose complement (the same problem with yes and no answers swapped) is in C.

We say that a problem L is C-*hard* if there is a polynomial-time reduction for every problem $L' \in$ C to L. If in addition $L \in$ C, then L is C-*complete*. For instance, a decision problem is NP-complete if it belongs to NP and all decision problems in NP have polynomial-time reductions to it.

Polynomial Hierarchy and PSpace. We are also interested in the polynomial hierarchy [Stockmeyer and Meyer, 1973; Stockmeyer, 1976; Wrathall, 1976; Papadimitriou, 1994]. The polynomial hierarchy consists of complexity classes Σ_i^{P} for $i \geq 0$ based on the following definitions: $\Sigma_0^{\text{P}} := \text{P}$ and $\Sigma_{i+1}^{\text{P}} = \text{NP}^{\Sigma_i^{\text{P}}}$ for all $i \geq 0$ where NP^{C} denotes the class of all decision problems such that there is a polynomial-time Turing reduction to any decision problem $L \in$ C, i.e., a decision problem $L' \in \text{NP}^{\text{C}}$ is non-deterministically polynomial-time solvable using any problem $L \in$ C as an oracle. A decision problem $L' \in \text{P}^{\text{C[log]}}$ is deterministically polynomial-time solvable doing $\mathcal{O}(\log n)$ calls to any problem $L \in$ C as an oracle. Moreover, $\Pi_i^{\text{P}} :=$ co-Σ_i^{P} for $i \geq 0$. Note that $\text{NP} = \Sigma_1^{\text{P}}$, $\text{coNP} = \Pi_1^{\text{P}}$, $\Sigma_2^{\text{P}} = \text{NP}^{\text{NP}}$, and $\Pi_2^{\text{P}} = \text{coNP}^{\text{NP}}$. Finally, we define the class PSpace, which is the class of all decision problems L, where we can decide for every instance $\mathcal{I}$, whether $\mathcal{I} \in L$ by using polynomial space $\mathcal{O}(\text{poly}(\|\mathcal{I}\|))$. Observe that the

whole polynomial hierarchy is contained in PSpace, i.e., we have that $\Sigma_i^{\mathrm{P}} \subseteq \mathrm{PSpace}$ as well as $\Pi_i^{\mathrm{P}} \subseteq \mathrm{PSpace}$ for $i \geq 0$.

Counting Complexity. A *witness function* [Durand et al., 2005] is a function $\mathcal{W} \colon \Sigma^* \to 2^{\Sigma'^*}$ that maps an instance $\mathcal{I} \in \Sigma^*$ to a finite subset of Σ'^*. We call the set $\mathcal{W}(\mathcal{I})$ the *witnesses*. Let $L : \Sigma^* \to \mathbb{N}$ be a counting problem, more precisely, a function that maps a given instance $\mathcal{I} \in \Sigma^*$ to the cardinality of its witnesses $|\mathcal{W}(\mathcal{I})|$. Let C be a decision complexity class. Then, $\#\cdot$ C denotes the class of all counting problems whose witness function $\mathcal{W}$ satisfies (i) there is a function $f : \mathbb{N} \to \mathbb{N}$ such that for every instance $\mathcal{I} \in \Sigma^*$ and every $W \in \mathcal{W}(\mathcal{I})$ we have $|W| \leq f(\|\mathcal{I}\|)$ and f is computable in time $\mathcal{O}(\mathrm{poly}(\|\mathcal{I}\|))$ and (ii) for every instance $\mathcal{I} \in \Sigma^*$ the decision problem $\mathcal{W}(\mathcal{I})$ belongs to the complexity class C. Then, $\#\cdot$ P is the complexity class consisting of all counting problems associated with decision problems in NP.

Let L and L' be counting problems with witness functions $\mathcal{W}$ and $\mathcal{W}'$. A *parsimonious reduction* from L to L' is a polynomial-time reduction $r : \Sigma^* \to \Sigma'^*$ such that for all $\mathcal{I} \in \Sigma^*$, we have $|\mathcal{W}(\mathcal{I})| = |\mathcal{W}'(r(\mathcal{I}))|$. It is easy to see that the counting complexity classes $\#\cdot$ C defined above are closed under parsimonious reductions. It is clear for counting problems L and L' that if $L \in \#\cdot$ C and there is a parsimonious reduction from L' to L, then $L' \in \#\cdot$ C.

Parameterized Complexity. An instance of a *parameterized problem* L is a pair $(\mathcal{I}, k) \in \Sigma^* \times \mathbb{N}$ for some finite alphabet Σ. For an instance $(\mathcal{I}, k) \in \Sigma^* \times \mathbb{N}$ we call $\mathcal{I}$ the *main part* and k the *parameter*. Then, $\|\mathcal{I}\|$ denotes the size of $\mathcal{I}$. We say L is *fixed-parameter tractable* if there exist a computable function f and a constant c such that we can decide whether $(\mathcal{I}, k) \in L$ in time $\mathcal{O}(f(k) \cdot \mathrm{poly}(\|\mathcal{I}\|))$. Such a (deterministic) algorithm that decides this is question called an *fpt-algorithm*. If L is a decision problem, then we identify L with the set of all yes-instances $(\mathcal{I}, k)$. FPT is the class of all fixed-parameter tractable decision problems.

Let $L \subseteq \Sigma^* \times \mathbb{N}$ and $L' \subseteq \Sigma'^* \times \mathbb{N}$ be two parameterized decision problems for some finite alphabets Σ and Σ'. An *fpt-reduction r (using g)* from L to L' is a many-to-one reduction from $\Sigma^* \times \mathbb{N}$ to $\Sigma'^* \times \mathbb{N}$ such that for all $\mathcal{I} \in \Sigma^*$ we have $(\mathcal{I}, k) \in L$ if and only if $r(\mathcal{I}, k) = (\mathcal{I}', k') \in L'$ and $k' \leq g(k)$ for fixed computable function $g : \mathbb{N} \to \mathbb{N}$ and there is a computable function f such that r is computable in time $\mathcal{O}(f(k) \cdot \mathrm{poly}(\|\mathcal{I}\|))$. We call r also an *f-bounded* fpt-reduction using g for given f and g. Thus, an fpt-reduction is, in particular, an fpt-algorithm. It is easy to see that the class FPT is closed under fpt-reductions. It is clear for parameterized problems L_1, and L_2 that if $L_1 \in \mathrm{FPT}$ and there is an fpt-reduction from L_2 to L_1, then $L_2 \in \mathrm{FPT}$.

The parameterized complexity class para-NP contains all parameterized decision problems L such that $(\mathcal{I}, k) \in L$ can be decided *non-deterministically* in time $\mathcal{O}(f(k) \cdot \mathrm{poly}\, \|\mathcal{I}\|)$ for some computable function f. A parameterized decision problem is para-NP-complete if it is in NP and NP-complete when restricted to finitely many parameter values [Flum and Grohe, 2006]. These classes can be further generalized. To this end, let C be a

decision complexity class. Recall that $\Sigma_0^{\mathrm{P}} := \mathrm{P}$ and $\Sigma_{i+1}^{\mathrm{P}} = \mathrm{NP}^{\Sigma_i^{\mathrm{P}}}$ for all $i \geq 0$ where NP^{C} denotes the class of all decision problems that are polynomial-time solvable using any problem $L \in \mathrm{C}$ as an oracle. Then, the parameterized class $\mathsf{para}\text{-}\mathrm{NP}^{\mathrm{C}}$ contains all parameterized decision problems L such that $(\mathcal{I}, k) \in L$ can be decided *non-deterministically* in time $\mathcal{O}(f(k) \cdot \mathrm{poly}\,\|\mathcal{I}\|)$ with access to a C oracle for some computable function f. Similar to above, a parameterized decision problem is $\mathsf{para}\text{-}\mathrm{NP}^{\mathrm{C}}$-complete if it is in NP^{C} and NP^{C}-complete when restricted to finitely many parameter values [Flum and Grohe, 2006].

Complexity Lower Bounds. In order to obtain conditional lower bounds, we employ the *exponential time hypothesis (ETH)* [Impagliazzo et al., 2001]. Intuitively, the ETH states a complexity theoretical lower bound on how fast satisfiability problems can be solved. More precisely, it states the following.

Hypothesis 2.1 (Exponential Time Hypothesis (ETH))**.** *One* cannot *solve 3-*Sat *in time* $2^{s \cdot n} \cdot n^{\mathcal{O}(1)}$ *for some* $s > 0$ *and an arbitrary instance with* n *variables.*

In the course of this work, we establish that many of our runtime bounds are asymptotically tight under the ETH.

2.3 (Quantified) Boolean Formulas

We define Boolean formulas and their evaluation called Boolean satisfiability (Sat) in the usual way, cf. [Biere et al., 2009; Kleine Büning and Lettman, 1999]. In particular, *literals* are variables or their negations and logical operators $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$ are used in the usual meaning. A *Boolean formula* is an expression over Boolean variables and logical operators. For such a Boolean formula F, we denote by $\mathsf{var}(F)$ the set of variables of F. A *term* is a formula that is a conjunction of literals and a *clause* is a disjunction of literals. For simplicity, both a clause and a term is oftentimes regarded as a set of literals. A Boolean formula F is in *conjunctive form (CF)* or *disjunctive form (DF)* if the formula itself is a conjunction or disjunction of formulas, respectively. In both cases, we might identify F itself by a set of formulas. Then, F is in *conjunctive normal form (CNF)* if F is a conjunction of clauses and F is in *disjunctive normal form (DNF)* if F is a disjunction of terms. We assume, unless mentioned otherwise, that a Boolean formula is either in CNF or DNF, which can therefore be regarded as a set of sets of literals. A formula is in *c-CNF* or *c-DNF* if each set in F consists of at most c many literals.

Let $\ell \geq 0$ be integer. A *quantified Boolean formula Q (in prenex normal form)* is of the form $Q_1 V_1 . Q_2 V_2 . \cdots Q_\ell V_\ell . F$ where $Q_i \in \{\forall, \exists\}$ for $1 \leq i \leq \ell$ and $Q_j \neq Q_{j+1}$ for $1 \leq j \leq \ell - 1$; and where V_i are disjoint, non-empty sets of Boolean variables with $\bigcup_{i=1}^{\ell} V_i \subseteq \mathsf{var}(F)$; and F is a Boolean formula. We call ℓ the *quantifier rank* of Q and let $\mathsf{matrix}(Q) := F$. Further, we denote the *variables* of Q by $\mathsf{var}(Q) := \mathsf{var}(\mathsf{matrix}(Q))$ and the set $\mathsf{fvar}(Q)$ of *free variables* of Q by $\mathsf{fvar}(Q) := \mathsf{var}(Q) \setminus (\bigcup_{i=1}^{\ell} V_i)$. If $\mathsf{fvar}(Q) = \emptyset$,

then Q is referred to as *closed*, otherwise we say Q is *open*. Unless stated otherwise, we assume open QBFs.

The truth (evaluation) of QBFs is defined in the standard way. An *assignment* is a mapping $\iota : X \to \{0,1\}$ defined for a set X of variables. An assignment ι' *extends* ι (by $\mathsf{dom}(\iota') \setminus \mathsf{dom}(\iota)$) if $\mathsf{dom}(\iota') \supseteq \mathsf{dom}(\iota)$ and $\iota'(y) = \iota(y)$ for any $y \in \mathsf{dom}(\iota)$. Given a Boolean formula F and an assignment ι for $\mathsf{var}(F)$. Then, for F in CNF, $F[\iota]$ is a Boolean formula obtained by removing every $c \in F$ with $x \in c$ and $\neg x \in c$ if $\iota(x) = 1$ and $\iota(x) = 0$, respectively, and by removing from every remaining clause $c \in F$ literals x and $\neg x$ with $\iota(x) = 0$ and $\iota(x) = 1$, respectively. Analogously, for F in DNF values 0 and 1 are swapped. For a given QBF Q and an assignment ι, $Q[\iota]$ is a QBF obtained from Q, where variables $x \in \mathsf{dom}(\iota)$ are removed from preceding quantifiers accordingly, and $\mathsf{matrix}(Q[\iota]) := (\mathsf{matrix}(Q))[\iota]$. A Boolean formula F *evaluates to true* if there exists an assignment ι for $\mathsf{var}(F)$ such that $F[\iota] = \emptyset$ if F is in CNF or $F[\iota] = \{\emptyset\}$ if F is in DNF. In this case, we refer to ι by *satisfying assignment* and we refer to the problem of deciding whether there exists a satisfying assignment for a Boolean formula F by *Boolean satisfiability (*Sat*)*. We define the evaluation of a closed QBF Q recursively and depending on the first quantifier of Q. More precisely, $Q = Q_1 V_1 . Q'$ *evaluates to true (or is valid)* depending on the following cases: If $Q_1 = \exists$, then Q evaluates to true if and only if there exists an assignment $\iota : V_1 \to \{0,1\}$ such that $Q[\iota]$ evaluates to true. If $Q_1 = \forall$, then $Q[\iota]$ evaluates to true if for any assignment $\iota : V_1 \to \{0,1\}$, $Q[\iota]$ evaluates to true. An (open or closed) QBF Q is *satisfiable* if there is a truth assignment $\iota : \mathsf{fvar}(Q) \to \{0,1\}$ such that resulting closed QBF $Q[\iota]$ evaluates to true. Otherwise Q is *unsatisfiable*.

Given a closed QBF Q, the *evaluation problem* QSat of QBFs asks whether Q evaluates to true; ℓ-QSat refers to the problem QSat on QBFs of quantifier rank ℓ. The problem QSat is PSpace-complete and is therefore believed to be computationally harder than Sat [Kleine Büning and Lettman, 1999; Papadimitriou, 1994; Stockmeyer and Meyer, 1973]. We refer to the problem ℓ-QSat when restricted to QBFs, whose first quantifier is existential ($\exists$) and universal ($\forall$), by Σ_ℓSat and Π_ℓSat, respectively. Then, we have that Σ_ℓSat is Σ^{P}_ℓ-complete and Π_ℓSat is Π^{P}_ℓ-complete. For more details on QBFs and known results we refer to other sources [Biere et al., 2009; Kleine Büning and Lettman, 1999].

Example 2.2. *Consider the closed QBF* $Q = \exists w, x. \forall y, z. D$*, where* $D := d_1 \vee d_2 \vee d_3 \vee d_4$*, and* $d_1 := w \wedge x \wedge \neg y$*,* $d_2 := \neg w \wedge \neg x \wedge y$*,* $d_3 := w \wedge y \wedge \neg z$*, and* $d_4 := w \wedge y \wedge z$*. Observe that* $Q[\iota]$ *is valid under assignment* $\iota = \{w \mapsto 1, x \mapsto 1\}$*. In particular,* $Q[\iota]$ *can be simplified to* $\forall y, z. (\neg y) \vee (y \wedge \neg z) \vee (y \wedge z)$*, which is valid, since for any assignment* $\kappa : \{y, z\} \to \{0,1\}$ *the formula* $Q[\iota][\kappa]$ *(and therefore* Q*) evaluates to true.*

The *projected model counting problem* PQSat *over QBFs* [Durand et al., 2005] takes an open QBF Q and asks to output the number of distinct assignments $\iota : \mathsf{fvar}(Q) \to \{0,1\}$ such that $Q[\iota]$ evaluates to true. There are well-studied fragments of the projected model counting problem. The problem PQSat over QBFs of quantifier rank ℓ, whose first quantifier is existential ($\exists$) is referred to by $\#\Sigma_\ell$Sat. Analogously, we address

the problem PQSAT when restricted to QBFs, whose the first quantifier is universal ($\forall$), with $\#\Pi_\ell$SAT. Further, there are also special cases of projected model counting over Boolean formulas, defined as follows. The problem PQSAT over QBFs of quantifier rank 0, is referred to by *model counting* #SAT [Gomes et al., 2009] and PQSAT over QBFs of quantifier rank 1 is called *projected model counting* $\#\exists$SAT.

In terms of computational complexity, #SAT is theoretically of high worst-case complexity, since it is $\#\cdot$ P-complete [Valiant, 1979a,b; Roth, 1996]. Notice that this is a very hard problem, given that there is a well-known result by Toda [1991], which states that already one call to a #SAT procedure suffices in order simulate any problem of the polynomial hierarchy.

When we consider the computational complexity of $\#\exists$SAT it turns out that under standard assumptions this problem is even harder than #SAT, more precisely, $\#\exists$SAT is complete for the class $\#\cdot$ NP [Durand et al., 2005]. Then, the full projected model counting problems $\#\Sigma_\ell$SAT and $\#\Pi_\ell$SAT are complete for the classes $\#\cdot\,\Sigma^{\mathrm{P}}_\ell$ and $\#\cdot\,\Pi^{\mathrm{P}}_\ell$, respectively [Durand et al., 2005].

2.4 Answer Set Programming

Answer Set Programming (ASP) is a declarative modeling and problem solving framework that combines techniques of knowledge representation and database theory. Two of the main advantages of ASP are its expressiveness [Brewka et al., 2011] and its advanced declarative problem modeling capability. This modeling capability is witnessed by the possibility of writing non-ground programs, where one uses first-order variables in order to write programs in an even more compact form. However, prior to solving, these compact (non-ground) programs are usually compiled into regular programs by a so-called grounder [Gebser et al., 2012].

We follow standard definitions of Boolean (non-ground) ASP [Brewka et al., 2011; Janhunen and Niemelä, 2016], i.e., we restrict ourselves to the case of plain ASP programs without first-order variables. ASP solvers often read and solve SModels input format [Syrjänen, 2002], which we will use in the course of this work. In the following, we cover the different *rule types* and fix notation of the SModels input format. To this end, let ℓ, m, n be non-negative integers such that $\ell \leq m \leq n$, and let a_1, …, a_n be distinct Boolean atoms. Moreover, we refer by *literal* to an atom or the negation thereof. A *choice rule* is an expression of the form

$$\{a_1, \ldots, a_\ell\} \leftarrow a_{\ell+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n.$$

with $\ell \geq 1$. Rules of this type intuitively enforce evidence of any subset of $\{a_1, \ldots, a_\ell\}$, if all atoms $a_{\ell+1}, \ldots, a_m$ are evident and there is no evidence for $a_{m+1}, \ldots, a_n$. A *disjunctive rule* is of the form

$$a_1 \vee \cdots \vee a_\ell \leftarrow a_{\ell+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n.$$

and intuitively requires evidence for at least one atom of $a_1, \ldots, a_\ell$ if all atoms $a_{\ell+1}, \ldots, a_m$ are evident and there is no evidence for any atoms of $a_{m+1}, \ldots, a_n$. A *weight rule* is of the form

$$a_\ell \leftarrow w \leqslant \{a_{\ell+1} = w_{\ell+1}, \ldots, a_m = w_m, \neg a_{m+1} = w_{m+1}, \ldots, \neg a_n = w_n\}.$$

where $\ell \leq 1$. Roughly, if the sum of all weights assigned to evidences that are met is at least w, then, in case of $\ell = 1$, weight rules enforce evidence of a_1, and in case of $\ell = 0$, the rule is not satisfied. A *rule* is either a disjunctive, a choice, or a weight rule, and a *program* Π is a set of these rules. For a rule r, we let $H_r := \{a_1, \ldots, a_\ell\}$, $B_r^+ := \{a_{\ell+1}, \ldots, a_m\}$, and $B_r^- := \{a_{m+1}, \ldots, a_n\}$. We denote the sets of *atoms* occurring in a rule r or in a program Π by $\mathrm{at}(r) := H_r \cup B_r^+ \cup B_r^-$ and $\mathrm{at}(\Pi) := \bigcup_{r\in\Pi} \mathrm{at}(r)$. A disjunctive rule r is *normal* if $|H_r| \leq 1$ and r is *unary* if $|B_r^+| \leq 1$. Then, a program Π is *disjunctive*, *normal* or *unary* if all its rules $r \in \Pi$ are disjunctive, normal or unary, respectively. The *positive dependency digraph* D_Π of Π is the directed graph defined on the set of atoms from $\bigcup_{r\in\Pi} H_r \cup B_r^+$, where for every rule $r \in \Pi$ two atoms $a \in B_r^+$ and $b \in H_r$ are joined by an edge (a, b). A *head-cycle* of D_Π is an $\{a, b\}$-cycle[1] for two distinct atoms a, $b \in H_r$ for some rule $r \in \Pi$. A disjunctive program Π is *head-cycle-free (HCF)* if D_Π contains no head-cycle [Ben-Eliyahu and Dechter, 1994] and we say Π is *tight*, if there is no directed cycle in D_Π [Fages, 1994].

For definition of semantics, we require auxiliary definitions for some rule types. For a weight rule $r = a_\ell \leftarrow w \leqslant \{a_{\ell+1} = w_{\ell+1}, \ldots, a_m = w_m, \neg a_{m+1} = w_{m+1}, \ldots, \neg a_n = w_n\}$, let $\mathrm{wght}(r, a)$ map atom a to its corresponding weight w_i in rule r if $a = a_i$ for $\ell+1 \leq i \leq n$ and to 0 otherwise. Moreover, let $\mathrm{wght}(r, A) := \sum_{a\in A} \mathrm{wght}(r, a)$ extend the definition for a set A of atoms, and let $\mathrm{bnd}(r) := w$ be its *bound*. An *interpretation* I is a set of atoms. Intuitively a rule enforces that whenever the body is "satisfied", the head has to be "satisfied" as well. Formally, a set $I \subseteq \mathrm{at}(\Pi)$ *satisfies* a rule r if we have the following:

(i) If r is a choice rule, then I satisfies r.

(ii) If r is a disjunctive rule, then either

  (a) the *head is satisfied*: $H_r \cap I \neq \emptyset$, or

  (b) the *negative body is dissatisfied*: $B_r^- \cap I \neq \emptyset$, or

  (c) the *positive body is dissatisfied*: $B_r^+ \not\subseteq I$.

(ii) If r is a weight rule, then either

  (a) the *head is satisfied*: $H_r \cap I \neq \emptyset$ or

  (b) the *body is dissatisfied*: $\mathrm{wght}(r, I \cap B_r^+) + \mathrm{wght}(r, B_r^- \setminus I) < \mathrm{bnd}(r)$.

[1] Let $G = (V, E)$ be a digraph and $W \subseteq V$. Then, a (directed) cycle in G is a W-cycle if it contains all vertices from W.

An interpretation I is a *model* of Π if it satisfies all rules of Π, in symbols $I \models \Pi$. Then, we refer to the problem of deciding whether there exists a model of an answer set program by MODELS. We let $\mathrm{Mod}(\mathcal{C}, \Pi) := \{C \mid C \in \mathcal{C}, C \models \Pi\}$ for a set $\mathcal{C} \subseteq 2^{\mathrm{at}(\Pi)}$ of interpretations be the set of models of program Π.

Further, we say a rule $r \in \Pi$ is *supporting* $a \in I$ *(with* I*)* if (i) $a \in H_r$, (ii) the body of r is satisfied by I, i.e., $I \not\models r'$ with r' being identical to r, except that $H_{r'} = \emptyset$, as well as (iii) $I \cap (H_r \setminus \{a\}]) = \emptyset$. Then, interpretation I is a *supported model* of Π if every $a \in I$ is *supported*, i.e., there is a rule $r \in \Pi$ supporting a with I. We refer to the problem of deciding whether a given program admits a supported model by SUPPORTED MODELS.

Towards the full semantics of answer set programs, we require to define the concept of reducts. More precisely, the *reduct* of a rule with respect to I, denoted by r^I, is a program, whose rules do not contain negative bodies, and it is defined as follows:

(i) for a choice rule r, r^I is given by $\{a \leftarrow B_r^+ \mid a \in H_r \cap I, B_r^- \cap I = \emptyset\}$,

(ii) for a disjunctive rule r, we let r^I be $\{H_r \leftarrow B_r^+ \mid B_r^- \cap I = \emptyset\}$, and

(iii) for a weight rule r, r^I is defined as $\{H_r \leftarrow w' \leqslant \{a = \mathrm{wght}(r, a) \mid a \in B_r^+\}\}$ where $w' = \mathrm{bnd}(r) - \Sigma_{a \in B_r^- \setminus I} \mathrm{wght}(r, a)$.

The *Gelfond-Lifschitz (GL) reduct*, cf. [Gelfond and Lifschitz, 1991], of Π under I is the program $\Pi^I := \bigcup_{r \in \Pi} r^I$ obtained from Π by combining the reducts of each rule. Interpretation $I \subseteq \mathrm{at}(\Pi)$ is an *answer set* of program Π if (i) $I \models \Pi$ and (ii) there does not exist a proper subset $I' \subsetneq I$ such that $I' \models \Pi^I$, that is, I is a *subset minimal model of* Π^I.

Example 2.3. *Assume the following program*

$$\dot{\Pi} := \{\overbrace{\{a; b\} \leftarrow c}^{r_1};\ \overbrace{c \leftarrow\ 1 \leqslant \{b = 1, \neg a = 1\}}^{r_2};\ \overbrace{d \vee a \leftarrow}^{r_3}\}.$$

Then, the set $A = \{a\}$ *is an answer set of* $\dot{\Pi}$*, since* A *is a minimal model of the reduct* $\dot{\Pi}^A = \{a \leftarrow\ c, c \leftarrow 1 \leqslant \{b = 1\}, d \vee a \leftarrow\}$*. Further,* $B = \{c, d\}$ *is an answer set, since* B *is a minimal model of* $\dot{\Pi}^B = \{c \leftarrow\ 0 \leqslant \{b = 1\}, d \vee a \leftarrow\}$*. By similar arguments one can see that also* $C = \{b, c, d\}$ *is an answer set of program* $\dot{\Pi}$*.*

The problem of deciding whether a program has an answer set is called *consistency (*ASP*)*, which is Σ_2^{P}-complete [Eiter and Gottlob, 1995a]. If the input is restricted to normal programs, the complexity drops to NP-complete [Bidoít and Froidevaux, 1991; Marek and Truszczyński, 1991]. A head-cycle-free program Π can be translated into a normal program in polynomial time [Ben-Eliyahu and Dechter, 1994]. Further, the answer sets of a tight program can be represented by means of the models of a Boolean formula, obtainable in linear time via, e.g., Clark's completion [Clark, 1977]. This thesis deals with the problems MODELS, SUPPORTED MODELS as well as ASP and ASP when

Decision Problem	Question	Fragment	Complexity
Models	Existence of a model	(full) programs	NP-co
Supported Models	Existence of a supported model	(full) programs	NP-co
Tight Asp	Existence of an answer set	tight programs	NP-co
Normal Asp	Existence of an answer set	normal programs	NP-co
HCF Asp	Existence of an answer set	head-cycle-free programs	NP-co
Disjunctive Asp	Existence of an answer set	disjunctive programs	Σ_2^{P}-co
Asp	Existence of an answer set	(full) programs	Σ_2^{P}-co

Table 2.1: Decision problems related to the answer set programming formalism that are discussed in the course of this thesis, which are described under the column "Question". The column "Fragment" states the corresponding fragment of logic programs and we list the respective (decision) complexity completeness result below the column "Complexity".

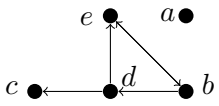

Figure 2.1: Dependency graph D_Π of Π (cf. Example 2.4).

restricted to several fragments of logic programs. As an overview, we list the resulting problems in Table 2.1. Note that in Section 4.4 we will motivate and define a novel fragment of programs, called "ι-tight" programs, which is therefore not listed in this table.

The following characterization of answer sets is often invoked when considering normal programs [Lin and Zhao, 2003]. Given a set $A \subseteq \mathrm{at}(\Pi)$ of atoms. Then, a function $\varphi : A \to \{0, \ldots, |A| - 1\}$ is a *level mapping* over A. Given a model I of a normal program Π, and a level mapping φ over I. An atom $a \in I$ is *proven* if there is a rule $r \in \Pi$ *proving* a, which is the case if r is supporting a with I and $\varphi(b) < \varphi(a)$ for every $b \in B_r^+$. Then, I is an *answer set* of Π if (i) I is a model of Π, and (ii) I *is proven*, i.e., every $a \in I$ is proven. This characterization vacuously holds also for head-cycle-free programs [Ben-Eliyahu and Dechter, 1994], since in HCF programs, vaguely speaking, all but one atom of the head of any rule can be "shifted" to the negative body [Dix et al., 1996]. More precisely, given an HCF program Π, one can create a normal program by constructing Rules (2.1) for each rule $r \in \Pi$ and $x \in H_r$. It is easy to see that the number of resulting rules is in $\mathcal{O}(|\mathrm{at}(\Pi)| \cdot |\Pi|)$ in the worst case.

$$x \leftarrow B_r^+, \overline{B_r^- \cup (H_r \setminus \{x\})} \qquad \text{for every } r \in \Pi \text{ and } x \in H_r \quad (2.1)$$

Even further, it is sufficient [Janhunen, 2006] to have individual level mappings per strongly connected component (SCC) of the dependency graph D_Π when determining (ii) provability of every atom.

Example 2.4. *Consider the program*

$$\Pi := \{\overbrace{a \vee b \leftarrow}^{r_1}; \overbrace{c \vee e \leftarrow d}^{r_2}; \overbrace{d \leftarrow b, \neg e}^{r_3}; \overbrace{e \leftarrow b, \neg d}^{r_4}; \overbrace{b \leftarrow e, \neg d}^{r_5}; \overbrace{d \leftarrow \neg b}^{r_6}\}.$$

Observe that Π *is not tight, since the dependency graph* D_Π *of Figure 2.1 contains cycle* b, d, e*. However, the program* Π *is head-cycle-free since there is neither an* $\{a, b\}$*-cycle, nor a* $\{c, e\}$*-cycle in* D_Π*. Therefore, rule* r_1 *allows shifting [Dix et al., 1996] and actually corresponds to the two rules* $a \leftarrow \neg b$ *and* $b \leftarrow \neg a$*. Analogously, rule* r_2 *can be seen as the rules* $c \leftarrow d, \neg e$ *and* $e \leftarrow d, \neg c$*. Then,* $I := \{b, c, d\}$ *is an answer set of* Π*, since* $I \models \Pi$*, and we can prove with level mapping* $\varphi := \{b \mapsto 0, d \mapsto 1, c \mapsto 2\}$ *atom* b *by rule* r_1*, atom* d *by rule* r_3*, and atom* c *by rule* r_2*. Further answer sets of* Π *are* $\{b, e\}$*,* $\{a, c, d\}$*, and* $\{a, d, e\}$*.*

The characterization above already fails for simple programs that are not HCF. Consider for example program $\Pi' := \{a \vee b \leftarrow; a \leftarrow b; b \leftarrow a\}$*, which has only one answer set* $I' = \{a, b\}$*. However,* I' *can not be proven. If the first rule* $a \vee b \leftarrow$ *shall prove* a*, we require* $b \notin I'$ *(and vice versa). Then, the remaining two rules of* Π' *can only prove either* a *or* b*, but fail to prove* I'*, since both rules proving* I' *(together) prohibits every level mapping due to the cyclic dependency.*

2.5 Tree Decompositions and Treewidth

Let $G = (V, E)$ be a graph, $T = (N, F)$ be a rooted tree, and $\chi : N \to 2^V$ be a function that maps each node $t \in N$ to a set of vertices. We call the sets $\chi(\cdot)$ *bags* and N the set of nodes. Then, the pair $\mathcal{T} = (T, \chi)$ is a *tree decomposition (TD)* of G if the following conditions hold [Robertson and Seymour, 1986]:

(i) all *vertices are covered* in some bag, that is, for every vertex $v \in V$ there is a node $t \in N$ with $v \in \chi(t)$;

(ii) all *edges are covered* in some bag, that is, for every edge $e \in E$ there is a node $t \in N$ with $e \subseteq \chi(t)$; and

(iii) the tree decomposition is *connected*, that is, for any three nodes $t_1, t_2, t_3 \in N$, if t_2 lies on the unique path from t_1 to t_3, then $\chi(t_1) \cap \chi(t_3) \subseteq \chi(t_2)$.

The *width* of the tree decomposition is defined as $\max\{|\chi(t)| - 1 \mid t \in N\}$. The *treewidth* $\mathsf{tw}(G)$ of a graph G is the minimum width over all possible tree decompositions of G. If for a tree decomposition $\mathcal{T} = (T, \chi)$, we have that the tree T actually corresponds to a path, i.e., every node in T is of degree less or equal than 2, we say that $\mathcal{T}$ is a *path decomposition*. Then, the *pathwidth* $\mathsf{pw}(G)$ of a graph G is the minimum width over all possible path decompositions of G.

Example 2.5. *Figure 2.2 illustrates a graph G and a TD $\mathcal{T}$ of G of width 2, which is also the treewidth of G, since G contains [Kloks, 1994] a complete graph (clique) among vertices e,b,d.*

Note that each graph [Robertson and Seymour, 1986] has a trivial tree decomposition (T, χ) consisting of the tree $(\{n\}, \emptyset)$ and the mapping $\chi : n \mapsto V$. It is well known that the treewidth of a tree is 1 and a graph containing a clique of size k has at least treewidth $k-1$. For some arbitrary but fixed integer k and a graph of treewidth at most k, we can compute a tree decomposition of width at most k in time $2^{\mathcal{O}(k^3)} \cdot |V|$ [Bodlaender and Koster, 2008].

For a given tree decomposition $\mathcal{T} = (T, \chi)$ with $T = (N, F)$, and an element $x \in \bigcup_{t \in N} \chi(t)$, we denote by $\mathcal{T}[x]$ the result of restricting $\mathcal{T}$ to nodes, whose bags contain x. Formally, $\mathcal{T}[x] := (T', \chi')$, where $T' := (N', F')$, $N' := \{t \mid t \in N, x \in \chi(t)\}$, $F' := F \cap (N' \times N')$, and for each $t \in N'$, we let $\chi'(t) := \chi(t)$.

Given a tree decomposition (T, χ) with $T = (N, \cdot, \cdot)$, for a node $t \in N$, we let type(t) be

leaf: if t has no children;

join: if t has children t' and t'' with $t' \neq t''$ and $\chi(t) = \chi(t') = \chi(t'') \neq \emptyset$;

intr: if t has a single child t', $\chi(t') \subseteq \chi(t)$ and $|\chi(t')| = |\chi(t)| - 1$; and

rem: if t has a single child t', $\chi(t') \supseteq \chi(t)$ and $|\chi(t')| = |\chi(t)| + 1$.

If every node $t \in N$ has at most two children, we refer to the tree decomposition as *join-nice*. If for a given constant c and every node $t \in N$ we have $\left|\chi(t) \setminus (\bigcup_{t' \in \mathsf{children}(t)} \chi(t'))\right| \leq c$, then the decomposition is referred to as *c-nice*. A tree decomposition that is join-nice such that for every node $t \in N$ we have that type$(t) \in \{\mathit{leaf}, \mathit{join}, \mathit{intr}, \mathit{rem}\}$, as well as empty bags for leaf nodes and the root node, is called *nice* [Bodlaender and Koster, 2008]. For every tree decomposition, we can compute a nice tree decomposition in linear time without increasing the width [Bodlaender and Koster, 2008]. Note that a nice TD of a graph $G = (V, E)$ of width $\mathsf{tw}(G)$, having only $\mathcal{O}(|V|)$ many nodes [Kloks, 1994, Lemma 13.1.2] always exists. Such a nice TD can be obtained from a given TD $\mathcal{T} = (T, \chi)$ with $T = (N, E)$ of width k in linear time, stated as follows.

Proposition 2.6 (nice TDs, cf. [Kloks, 1994, Lemma 13.1.3])**.** *Given a graph G and a TD $\mathcal{T} = (T, \chi)$ of G with $T = (N, E)$. Then, one can obtain a nice TD $\mathcal{T}' = (T', \chi')$ of G in time $\mathcal{O}(k^2 \cdot 2^{k^3} \cdot |N|)$.*

Proposition 2.7 (join-nice TDs)**.** *Given a graph G and a TD $\mathcal{T} = (T, \chi)$ of G with $T = (N, E)$. Then, one can obtain a join-nice TD $\mathcal{T}' = (T', \chi')$ of G in time $\mathcal{O}(|N|)$.*

Proof (Idea). We replace every node t of T with $|\mathsf{children}(t)| > 2$ many child nodes by a tree of fresh nodes as follows. We arbitrarily take two child nodes in $\mathsf{children}(t)$, which

Figure 2.2: Graph G (left) and a tree decomposition $\mathcal{T}$ of G (right).

get a fresh parent node t' with $\chi(t') := \chi(t)$. Then, node t' and an other child node of t gets a fresh parent t'' with $\chi(t'') := \chi(t)$. In turn, we create a tree over auxiliary nodes $t', t'', \cdots$ such that when we considered all child nodes of t, the auxiliary node created last serves as a "replacement" for t. Consequently, this node that is created last is the new child node of the parent of t in T. Note that we can do this for all such nodes t, where we apply this procedure in post-order of T (bottom-up) and we refer to the resulting join-nice TD of G by $\mathcal{T}' = (T', \chi')$. Observe that every node t has at most one parent node in T, and consequently this transformation can be done in time $\mathcal{O}(|N|)$. □

Proposition 2.8 (c-nice TDs)**.** *Given a graph G, a TD $\mathcal{T} = (T, \chi)$ of G of width k with $T = (N, E)$, and a constant $c \geq 1$. Then, one can obtain a c-nice TD $\mathcal{T}' = (T', \chi')$ of G in time $\mathcal{O}(\frac{k}{c} \cdot |N|)$.*

Proof (Idea). We manipulate $\mathcal{T}$, where we create auxiliary nodes for every node t with $X = \chi(t) \setminus (\bigcup_{t' \in \mathsf{children}(t))} \chi(t'))$ such that $|X| > c$. For every such node t we create at most $\frac{d}{c}$ many auxiliary nodes as follows. We chose the first $\frac{d}{c}$ many elements of X, resulting in a set X' and create a node t' such that $\chi'(t') := X' \cup (\chi(t) \cap (\bigcup_{t' \in \mathsf{children}(t)} \chi(t')))$ with $\mathsf{children}(t') := \mathsf{children}(t)$. Then, we create the next auxiliary node, taking into account $\frac{d}{c}$ many elements of X that are not already considered (not yet in X'). We create a node t'' where we set $\mathsf{children}(t'') = \{t'\}$ and $\chi'(t'') := \chi'(t') \cup X''$. In turn, the next node is a parent of t'' that adds $\frac{d}{c}$ many elements of X neither in X' nor in X'' to its bag, and so on. The resulting TD is referred to by $\mathcal{T}' = (T', \chi')$. Since we have $|N|$ many nodes in T and $d \leq k$, the procedure above runs in time $\mathcal{O}(\frac{k}{c} \cdot |N|)$. □

With the help of the parameter treewidth, one can show that the problem ℓ-QSAT for given $\ell \in \mathbb{N}^+$ can be turned tractable.

Proposition 2.9 (Runtime of ℓ-QSAT, cf. [Chen, 2004])**.** *For an arbitrary QBF Q with quantifier rank $\ell \geq 1$, the problem ℓ-QSAT can be solved in time $\mathsf{tower}(\ell, \mathcal{O}(k)) \cdot \mathrm{poly}(|\mathsf{var}(Q)|)$.*

Interestingly, under the exponential time hypothesis (ETH), one cannot significantly improve this runtime for Boolean satisfiability (SAT).

Proposition 2.10. *Unless the ETH fails, one cannot decide* SAT *for any given Boolean formula F in time $2^{o(\mathsf{tw}(\mathcal{G}_F))} \cdot \mathrm{poly}(|\mathsf{var}(F)|)$.*

Proof. Consider an arbitrary Boolean formula F. Assume towards a contradiction that one can decide Sat for F in time $2^{o(\mathsf{tw}(\mathcal{G}_F))} \cdot \text{poly}(|\mathsf{var}(F)|)$. Since $o(\mathsf{tw}(\mathcal{G}_{\mathsf{var}(F)})) \subseteq o(|\mathsf{var}(F)|)$, it follows that we can solve Sat on F in time $2^{o(|\mathsf{var}(F)|)} \cdot \text{poly}(|\mathsf{var}(F)|)$. This, however, contradicts ETH (cf. Hypothesis 2.1). □

This result was even lifted to the evaluation of the problem 2-QSat as follows.

Proposition 2.11 (cf. [Lampis and Mitsou, 2017])**.** *Given any QBF instance $Q = Q_1V_1.Q_2V_2.F$ of problem* 2-QSat *Then, unless the ETH fails, the validity of Q cannot be decided in time $2^{2^{o(k)}} \cdot \text{poly}(|\mathsf{var}(F)|)$, where k is the treewidth of the primal graph $\mathcal{G}_Q$.*

In fact, there are also further results on lower bounds that can be obtained by relying on ETH and for problems when parameterized by treewidth, e.g., [Lokshtanov et al., 2011; Marx and Mitsou, 2016; Lampis et al., 2018; Fichte et al., 2018b].

In order to use the concept of tree decompositions for solving problems, we require dedicated graph representation of instances of these problems. Thereby we mainly consider the so-called primal graph representation of an instance, whose definition heavily depends on the problem. However, as a guide through this work and without having a formal definition at hand yet, we refer by $\mathcal{G}_\mathcal{I}$ to the *primal graph (representation)* of a given problem instance $\mathcal{I}$. While the exact definition of the primal graph representation depends on the problem, intuitively, given an instance $\mathcal{I}$ of a problem, primal graph representation $\mathcal{G}_\mathcal{I}$ refers to a graph that has as vertices the elements of the instance $\mathcal{I}$ that are subject to evaluation, i.e., those elements are basic "atoms" used in formulating the instance. Then, there is an edge between two vertices of the graph of $\mathcal{G}_\mathcal{I}$, whenever the two corresponding elements of the problem are required to be evaluated "together" during the evaluation of instance $\mathcal{I}$. In order to apply a tree decomposition $\mathcal{T} = (T, \chi)$ of a graph representation, e.g., $\mathcal{G}_\mathcal{I}$, for establishing certain results for instance $\mathcal{I}$, we rely on the concept of the *canonical bag instance* $\mathcal{I}_t$, which is the instance obtained by restricting instance $\mathcal{I}$ to node t of T, i.e., the instance restricted to contents of bag $\chi(t)$. Later, any (smaller) instance $\mathcal{I}'_t$ that is part of the canonical bag instance $\mathcal{I}_t$ will also be referred to by *bag instance*.

Formal definitions of these concepts are provided below for both Boolean formulas as well as answer set programs.

Tree Decompositions for (Quantified) Boolean Formulas

The *primal graph* $\mathcal{G}_F$ of a Boolean formula F in conjunctive form (CF) or disjunctive form (DF), regarded as a set of formulas, has the variables $\mathsf{var}(F)$ of F as vertices and an edge $\{x, y\}$ if there exists $f \in F$ such that $x, y \in \mathsf{var}(f)$, respectively. For a QBF Q, we identify its primal graph with the primal graph of its matrix, i.e., let $\mathcal{G}_Q := \mathcal{G}_{\mathsf{matrix}(Q)}$.

Example 2.12. *Recall the closed QBF $Q = \exists w, x.\forall y, z.D$ from Example 2.2, where $D := d_1 \vee d_2 \vee d_3 \vee d_4$, and $d_1 := w \wedge x \wedge \neg y$, $d_2 := \neg w \wedge \neg x \wedge y$, $d_3 := w \wedge y \wedge \neg z$, and*

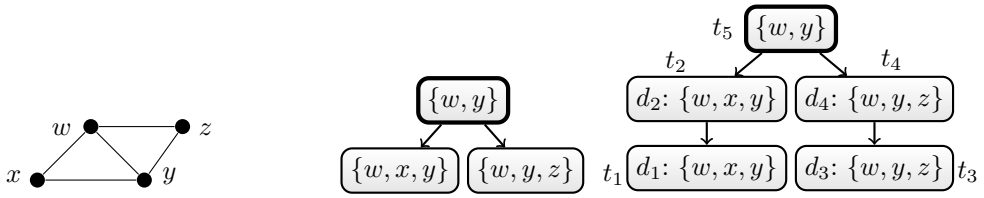

Figure 2.3: Primal graph $\mathcal{G}_Q$ of Q from Example 2.2 (left) with TDs $\mathcal{T}_1, \mathcal{T}_2$ of graph $\mathcal{G}_Q$ (right).

$d_4 := w \wedge y \wedge z$. *Observe that Figure 2.3 illustrates the primal graph $\mathcal{G}_Q$ of Q, whose matrix D is regarded as a set of sets of literals, and two tree decompositions of $\mathcal{G}_Q$ of width* 2. *The graph $\mathcal{G}_Q$ has treewidth* 2, *since the vertices w,x,y are completely connected and hence width* 2 *is optimal [Kloks, 1994].*

Then, for (Quantified) Boolean formulas, a bag instance is defined as follows. To this end let $\mathcal{T} = (T, \chi)$ be a TD of primal graph $\mathcal{G}_F$ of a Boolean formula, and let t be a node of T. The *canonical bag formula* F_t contains formulas entirely covered by the bag $\chi(t)$. Formally, $F_t := \{f \mid f \in F, \mathsf{var}(f) \subseteq \chi(t)\}$. This concept of bag formulas naturally extends to QBFs, where we define $Q_t := \mathsf{matrix}(Q)_t$. Any subset $F'_t \subseteq F_t$ or $Q'_t \subseteq Q_t$ of the canonical bag formula F_t or Q_t is also referred to by *bag formula*, respectively.

Example 2.13. *Recall matrix D of QBF Q from Example 2.2. Observe that decomposition $\mathcal{T}_2$ of Figure 2.3 (right) is a TD of the primal graph of Q. Further, we have $Q_{t_1} = D_{t_1} = Q_{t_2} = D_{t_2} = \{d_1, d_2\}$, $Q_{t_3} = D_{t_3} = Q_{t_4} = D_{t_4} = \{d_3, d_4\}$, as well as $Q_{t_5} = D_{t_5} = \emptyset$. Note that in general a rule might appear in several canonical bag formulas, which can be witnessed by the canonical bag formulas of decomposition $\mathcal{T}_1$ of Figure 2.3 (right).*

Tree Decompositions for Answer Set Programs

In order to use TDs for solving Asp, we need dedicated graph representations of programs [Jakl et al., 2009]. The *primal graph* $\mathcal{G}_\Pi$ of program Π has the atoms of Π as vertices and an edge $\{a, b\}$ if there exists a rule $r \in \Pi$ and $a, b \in \mathrm{at}(r)$. Let $\mathcal{T} = (T, \chi)$ be a TD of primal graph $\mathcal{G}_\Pi$ of a program Π, and let t be a node of T. The *canonical bag program* Π_t contains rules entirely covered by the bag $\chi(t)$. Formally, $\Pi_t := \{r \mid r \in \Pi, \mathrm{at}(r) \subseteq \chi(t)\}$. Further, we refer to any subset $\Pi'_t \subseteq \Pi_t$ of the canonical bag program Π_t by *bag program* as well.

Example 2.14. *Recall the program* $\Pi := \{\overbrace{a \vee b \leftarrow}^{r_1}; \overbrace{c \vee e \leftarrow d}^{r_2}; \overbrace{d \leftarrow b, \neg e}^{r_3}; \overbrace{e \leftarrow b, \neg d}^{r_4}; \overbrace{b \leftarrow e, \neg d}^{r_5}; \overbrace{d \leftarrow \neg b}^{r_6}\}$ *from Example 2.4. Observe that graph G of Figure 2.2 is the primal graph of Π. Further, we have $\Pi_{t_1} = \{r_2\}$, $\Pi_{t_2} = \{r_1\}$, and $\Pi_{t_3} = \{r_3, r_4, r_5, r_6\}$. Note that in general a rule might appear in several canonical bag programs.*

2.6 Labeled Tree Decompositions

While plain tree decompositions as defined above form the basis of our studies in this work, sometimes proofs can be simplified by further restricting these decompositions. This brings us to the concept of labeled tree decompositions, which is also defined specific for the corresponding problems. However, intuitively, a *labeled tree decomposition of an instance* $\mathcal{I}$ is a triple $\mathcal{T} = (T, \chi, \delta_\mathcal{I})$, where (T, χ) is a tree decomposition of primal graph $\mathcal{G}_\mathcal{I}$. Note that actually LTDs can be defined for *any graph representation* of $\mathcal{I}$, but unless mentioned otherwise this thesis deals with the primal graph $\mathcal{G}_\mathcal{I}$ of an instance $\mathcal{I}$. Then, $\delta_\mathcal{I}$ is a labeling function that assigns each node of T a label in the form of a bag instance, which is a part of the canonical bag instance $\mathcal{I}_t$. Thereby, these labeling tree decompositions enable the precise control of which parts of the instance $\mathcal{I}$ are evaluated in which node t such that indeed all relevant parts of $\mathcal{I}$ are evaluated.

Observation 2.15. *Observe that any tree decomposition (T, χ) of $\mathcal{G}_\mathcal{I}$ can be turned into the corresponding* canonical labeled tree decomposition *$\mathcal{T} = (T, \chi, \delta_\mathcal{I})$ of $\mathcal{I}$, where $\delta_\mathcal{I}$ is such that for every node t of T, $\delta_\mathcal{I}(t) := \mathcal{I}_t$.*

For instances $\mathcal{I}$, which are represented as sets, we define the following.

Definition 2.16 (Labeled TD of sets)**.** *Let* P *be a problem, where every instance $\mathcal{I}$ of* P *is a set (of elements), including every bag instance $\mathcal{I}_t$ for any node t of any TD of $\mathcal{G}_\mathcal{I}$. Then, a* labeled tree decomposition (LTD) *of such a set $\mathcal{I}$ is a tuple $\mathcal{T} = (T, \chi, \delta_\mathcal{I})$ with $T = (N, E)$, where (T, χ) is a tree decomposition of $\mathcal{G}_\mathcal{I}$, and $\delta_\mathcal{I} : N \to 2^\mathcal{I}$ is a mapping from nodes of T to subsets of $\mathcal{I}$ such that (1) for every $t \in N$, $\delta_\mathcal{I}(t) \subseteq \mathcal{I}_t$ and (2) $\bigcup_{t \in N} \delta_\mathcal{I}(t) = \mathcal{I}$.*

Then, *non-redundant* LTDs (of sets) ensure that $\delta_\mathcal{I}$ is such that each part of $\mathcal{I}$ only occurs once in $\delta_\mathcal{I}$, i.e., for every two distinct nodes t, t' of T we have $\delta_\mathcal{I}(t) \cap \delta_\mathcal{I}(t') = \emptyset$. Further, an LTD of a set is *atomic* if $|\delta_\mathcal{I}(t)| = 1$ for every node t of T. For most problems one can also turn a tree decomposition (T, χ) into a non-redundant, atomic labeled tree decomposition by copying decomposition nodes and assigning $\delta_\mathcal{I}$ accordingly.

The concrete definitions for Boolean formulas and answer set programs are given next.

Labeled Tree Decompositions for (Quantified) Boolean Formulas

The precise formal definition of labeled tree decompositions for Boolean formulas in conjunctive form (CF) or disjunctive form (DF) is as follows.

Definition 2.17. *A* labeled tree decomposition (LTD) *$\mathcal{T}$ of a Boolean formula F in CF or DF, is a labeled tree decomposition of set F, as defined in Definition 2.16.*

Example 2.18. *Consider again the QBF Q from Example 2.2 as well as Figure 2.3 (right). Recall that* $\mathsf{matrix}(Q) = D$ *with $D := d_1 \vee d_2 \vee d_3 \vee d_4$, where $d_1 := w \wedge x \wedge \neg y$, $d_2 := \neg w \wedge \neg x \wedge y$, $d_3 := w \wedge y \wedge \neg z$, and $d_4 := w \wedge y \wedge z$. Observe that $\mathcal{T}_2$ is a 3-nice,*

non-redundant, and atomic labeled tree decomposition (T, χ, δ_D) *of* $\mathcal{G}_D$*, where labeling function* δ_D *sets* $\delta_D(t_i) := \{d_i\}$*, for* $1 \leq i \leq 4$*.*

Interestingly, a non-redundant and atomic labeled tree decomposition of a Boolean formula F (in CF or DF) can be constructed easily, given a tree decomposition of $\mathcal{G}_F$.

Proposition 2.19. *Given a Boolean formula* F *in CF or DF and a tree decomposition* (T, χ) *of* $\mathcal{G}_F$ *of width* k*, where* $T = (N, E)$*. Then, one can construct (1) a non-redundant LTD* $\mathcal{T} = (T, \chi^*, \delta_F)$ *of* F *in time* $\mathcal{O}(k \cdot (|N| + |F|))$*. Further, (2) a non-redundant, atomic LTD* $\mathcal{T}' = (T', \chi', \delta'_F)$ *of* F *can be constructed in time* $\mathcal{O}(k \cdot (|N| + |F|))$*.*

Proof. For showing (1), we define for each element $v \in \mathsf{var}(F)$ a unique node t of T, defined by $rem(v) := t$, where t is such that $v \notin \chi(t)$, but there is a child node $t' \in \mathsf{children}(t)$ with $v \in \chi(t')$. Observe that by definition of tree decompositions, this is well-defined, since if there were two different nodes for v this would contradict connectedness of (T, χ). Let $\prec$ be total ordering among elements of $\mathsf{var}(F)$ such that whenever for two disjoint elements $x, y \in \mathsf{var}(F)$ we have that $rem(x)$ is a node below node $rem(y)$ in T, then $x \prec y$. Indeed, such an ordering can be computed by traversing T in post-order, which takes time $\mathcal{O}(k \cdot |N|)$. Then, we define the labeling δ_F for every node t of T as follows: $\delta_F(t) := \{c \mid c \in F, v \in \mathsf{var}(c)$ is the $\prec$-smallest element among all variables in $\mathsf{var}(c), t = rem(v)\} \cup \{c \mid c \in F_t, t = \text{root}(T)\}$. Further, we define bag $\chi^*(t)$ for each node t of T by $\chi^*(t) := \chi(t) \cup \{v \mid t = rem(v)\}$. Observe that therefore $|\chi^*(t)| \leq 2 \cdot |\chi(t)|$ for every node t of T and that (T, χ^*) is still a tree decomposition of $\mathcal{G}_F$.

We can compute δ_F by traversing over Π. Thereby, for each rule $r \in \Pi$, we determine the $\prec$-smallest element $v \in \mathsf{var}(F)$ among all variables in $\mathsf{var}(F)$ and add it to $\delta_F(rem(v))$ accordingly. This takes time $\mathcal{O}(k \cdot |\Pi|)$. Overall, computing LTD $\mathcal{T}$ takes time $\mathcal{O}(k \cdot (|N| + |\Pi|))$.

For (2), we take LTD $\mathcal{T}$ as defined above and modify T, χ^*, and δ_F as follows, which results in T', χ', and δ'_F, respectively. Whenever there is a node t of T with $|\delta_F(t)| > 1$, where $\delta_F(t) = \{c_1, \ldots, c_o\}$, we replace t by a path of o many copy nodes $\{t_1, \ldots, t_o\}$ of t with $\chi'(t_i) := \chi^*(t)$ such that $\delta_F(t_i)' := \{c_i\}$ for node $t_i \in \{t_1, \ldots, t_o\}$. Observe that thereby the number of nodes in T' is increased by at most $|F|$ and therefore the runtime claim holds as well. □

Note that the LTD of the proof of the proposition above can be slightly adapted such that we avoid doubling the bag sizes in the worst case. However, thereby the labeling for a node t as defined above needs to be "shifted" to a suitable representative child node of t, which slightly complicates the algorithm sketched above.

Example 2.20. *Consider once again the QBF* Q *and matrix* $D = \mathsf{matrix}(Q)$ *from Example 2.2 as well as Figure 2.3 (right). Recall that* $\mathcal{T}_1$ *is a tree decomposition of* $\mathcal{G}_D$*. Further,* $\mathcal{T}_2$ *is a* 3*-nice, non-redundant, and atomic labeled tree decomposition* (T, χ, δ_D)

of D that can be obtained with such a slightly adapted procedure as the one given in the proof of Proposition 2.19.

Observe that labeled tree decompositions for Boolean formulas naturally extend to the case of Quantified Boolean formulas by using the matrix of QBFs, as defined by the primal graph of a QBF in Section 2.5.

Labeled Tree Decompositions for Answer Set Programs

Next, we precisely define labeled tree decompositions for answer set programs.

Definition 2.21. *A* labeled tree decomposition (LTD) $\mathcal{T}$ *of an answer set program* Π *is a labeled tree decomposition of set* Π *of rules, as defined in Definition 2.16.*

Similar to above, we can establish the following results for LTDs.

Proposition 2.22. *Given an answer set program* Π *and a tree decomposition* (T, χ) *of* $\mathcal{G}_\Pi$ *of width* k*, where* $T = (N, E)$*. Then, one can construct (1) a non-redundant LTD* $\mathcal{T} = (T, \chi^*, \delta_\Pi)$ *of* Π *in time* $\mathcal{O}(k \cdot (|N| + |\Pi|))$*. Further, (2) a non-redundant, atomic LTD* $\mathcal{T}' = (T', \chi', \delta'_\Pi)$ *of* Π *can be constructed in time* $\mathcal{O}(k \cdot (|N| + |\Pi|))$*.*

Proof. The proof proceeds almost identical to the proof of Proposition 2.19, but formula F is replaced by program Π and instead of clauses and variables we have rules and atoms, respectively. □

Further, we also show that such a non-redundant, atomic LTD can be constructed for disjunctive programs that are head-cycle-free (HCF), where the labeling function of the LTD only refers to normal rules. More precisely, we construct a non-redundant, atomic LTD of a normal program, which is the result of reducing from a given head-cycle-free program by using Rules (2.1).

Theorem 2.23. *Given an HCF program* Π *and a tree decomposition* (T, χ) *of* $\mathcal{G}_\Pi$ *of width* k*, where* $T = (N, E)$*. Then, one can construct a non-redundant, atomic LTD* $\mathcal{T}'' = (T'', \chi'', \delta''_\Pi)$ *of the program* Π' *obtained by the reduction consisting of Rules (2.1), in time* $\mathcal{O}(k \cdot (|N| + |\Pi|))$*. Further, the number of rules in* Π' *is at most* $\mathcal{O}(k \cdot |\Pi|)$ *and the width of* $\mathcal{T}''$ *is* k*.*

Proof. For a rule $r \in \Pi$ and atom $x \in H_r$, in the following we let $R(r, x)$ refer to the rule obtained by applying the reduction consisting of Rules (2.1) for r and x. The proof takes a non-redundant, atomic LTD $\mathcal{T}' = (T', \chi', \delta'_\Pi)$ of Π as constructed in Proposition 2.22. Then, in the following we modify T', χ', and δ'_Π, which results in T'', χ'', and δ''_Π, respectively. Whenever there is a node t of T' with $|H_r| > 1$ for $r \in \delta'_\Pi(t)$, where $H_r = \{x_1, \ldots, x_o\}$, we replace t by a path of o many copy nodes $\{t_1, \ldots, t_o\}$ of t with $\chi''(t_i) := \chi'(t)$ such that $\delta_\Pi(t_i)'' := \{R(r, x_i)\}$ for node $t_i \in \{t_1, \ldots, t_o\}$.

Consequently, the number of rules and nodes in T'' is thereby increased by at most $k \cdot |\Pi|$, since LTD $\mathcal{T}'$ is non-redundant, i.e., each rule $r \in \Pi$ occurs in δ_Π at most once, and due to the observation that $|H_r| \leq k$ for rules r appearing in a bag program. Therefore the runtime claim holds. □

Note that this result also shows that the overhead of transforming a head-cycle-free program of treewidth k into a normal program using Rules (2.1), can be regarded as slightly better than the quadratic worst-case runtime overhead due to shifting [Dix et al., 1996]. Indeed, if for a program Π the treewidth $k = \mathsf{tw}(\mathcal{G}_\Pi)$ is reasonably small, i.e., $k \ll |\mathrm{at}(\Pi)|$ and $k \ll |\Pi|$, Proposition 2.23 reveals a subquadratic runtime.

CHAPTER 3

Upper Bounds for Utilizing Treewidth by Dynamic Programming

Everything should be made as simple as possible, but not simpler.

— Albert Einstein

One of the most prominent methods to utilize treewidth [Cygan et al., 2015; Downey and Fellows, 2013] is by means of *dynamic programming on tree decompositions*. Thereby, the method of dynamic programming [Bellman, 1954; Dreyfus and Law, 1977], which generally refers to breaking down problems in a divide-and-conquer fashion, is guided along a tree decomposition, where the decomposition is traversed in post-order (bottom-up traversal) such that during the traversal a table is computed for each node of the decomposition. While for a given graph the computation of a tree decomposition of minimal width (treewidth) is NP-hard, it is possible to efficiently approximate treewidth [Bodlaender et al., 2016; Bodlaender, 1996; Feige et al., 2008] and compute a tree decomposition, and there are also numerous efficient heuristics as well as exact solvers available [Abseher et al., 2017; Dell et al., 2017].

The literature distinguishes plenty of research on dynamic programming of tree decompositions for diverse problems and formalisms [Bertelè and Brioschi, 1972, 1973; Bodlaender and Kloks, 1996; Flum and Grohe, 2006; Niedermeier, 2006]. There are even existing implementations that take advantage of treewidth in the form of (a) specialized solvers such as dynasp, dynQBF, gpuSAT, and fvs-pace [Fichte et al., 2017b; Charwat and Woltran, 2017; Fichte et al., 2018b, 2019b; Kiljan and Pilipczuk, 2018] as well as (b) general systems that exploit treewidth like D-FLAT [Bliem et al., 2016a], Jatatosk [Bannach and Berndt, 2019], and sequoia [Langer et al., 2012]. Some of these systems explicitly use dynamic programming to directly exploit treewidth by means of tree decompositions,

whereas others provide some kind of declarative layer to model the problem and compute the decomposition and dynamic programming internally.

This chapter concerns the development of dynamic programming algorithms on tree decompositions, whereby we particularly focus on diverse fragments of answer set programming. While for answer set programming there is a wide range of more fine-grained studies [Truszczyński, 2011], also in parameterized complexity [Fichte et al., 2019c; Lackner and Pfandler, 2012], including treewidth [Jakl et al., 2009; Pichler et al., 2010; Gottlob et al., 2010; Pichler et al., 2014] as well as related measures [Bliem et al., 2016b], deeper studies on the complexity of the diverse fragments for treewidth have not been conducted yet. To this end, Section 3.1 starts by discussing basics on dynamic programming on tree decompositions, where this concept is exemplarily illustrated by means of Boolean satisfiability (Sat). Then, Section 3.2 concerns about dynamic programming algorithms for diverse fragments of answer set programming as follows. In Section 3.2.1, we start with extending the algorithm for Boolean satisfiability of Section 3.1 in order to end up with an algorithm for solving Asp restricted to the fragment of tight programs. Interestingly, the algorithm of Section 3.2.1 is also capable of computing supported models and thereby solves problem Supported Models for any Asp program. Then, in Section 3.2.2 we discuss an algorithm for solving Normal Asp as well as HCF Asp, which was originally published in the work [Fichte and Hecher, 2019]. This algorithm is inspired by related works on Asp for treewidth [Pichler et al., 2014] and guides the characterization of answer sets by Lin and Zhao [2003], but extended to head-cycle-free programs [Ben-Eliyahu and Dechter, 1994], along a tree decomposition. Finally, Section 3.2.3 extends an existing algorithm [Jakl et al., 2009] for Disjunctive Asp, thereby showing how to solve Asp for any logic program by means of dynamic programming. This algorithm was first discussed in the work [Fichte et al., 2017a] and we also published an improved implementation [Fichte et al., 2017b] of this algorithm later. Surprisingly, for the primal graph representation of logic programs as introduced in Section 2.5, this algorithm works fine, whereas for a different graph representation treewidth is insufficient [Pichler et al., 2014] to achieve tractability in the form of such an algorithm.

3.1 Basics on Dynamic Programming

Algorithms that utilize treewidth for solving a problem in linear time typically proceed by dynamic programming along the tree decomposition. Thereby the tree is traversed in post-order and at each node t of the tree information is gathered [Bertelè and Brioschi, 1972, 1973; Bodlaender and Kloks, 1996] in a table τ_t. A *table* τ is a set of rows, where a *row* $u \in \tau$ is a sequence or tuple of fixed length. These tables are derived by an algorithm, which we therefore call *table algorithm* A. The actual length, content, and meaning of the rows depend on the algorithm A that derives tables. Therefore, we often explicitly state A-*row* if rows of this *type* are syntactically used for table algorithm A and similar A-*table* for tables. For sake of comprehension and readability, we always specify the rows before presenting the actual table algorithm for manipulating tables. Thereby, for a given

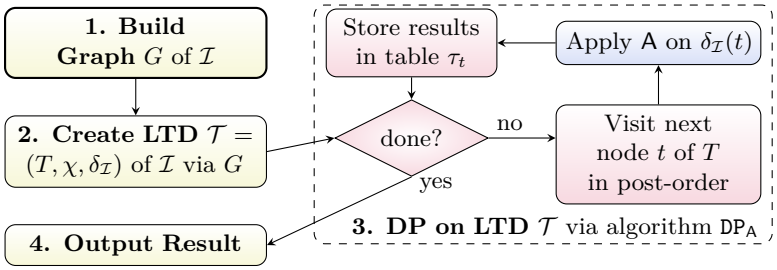

Figure 3.1: The DP approach, where table algorithm A modifies tables.

Listing 3.1: Algorithm $\mathtt{DP}_{\mathsf{A}}(\mathcal{I}, \mathcal{T})$ for computing solutions of $\mathcal{I}$ via DP on LTD $\mathcal{T}$.

In: Table algorithm A and a LTD $\mathcal{T} = (T, \chi, \delta_{\mathcal{I}})$ of $\mathcal{I}$ using a graph representation suitable for A.
Out: Table mapping A-Tabs, which maps each TD node t of T to some computed table τ_t.

```
1 A-Tabs ← {}     /* empty mapping */
2 for iterate t in post-order(T) do
3 |   Child-Tabs ← ⟨A-Tabs[t_1], ..., A-Tabs[t_ℓ]⟩ where post-children(t) = ⟨t_1, ..., t_ℓ⟩
4 |   A-Tabs[t] ← A_t(χ(t), δ_I(t), Child-Tabs)
5 return A-Tabs
```

positive integer i and a row u of a table τ, we denote by $u_{(i)}$ the i-th element of row u and further define $\tau_{(i)}$ as $\tau_{(i)} := \{u_{(i)} \mid u \in \tau\}$.

The dynamic programming approach for solving an instance $\mathcal{I}$ of a problem P relies on a table algorithm A and works as outlined in Figure 3.1, consisting of the following steps:

1. Graph: Construct a graph representation G of the instance such that table algorithm A can solve $\mathcal{I}$ via G. In this work we mostly deal with the primal graph $\mathcal{G}_{\mathcal{I}}$ of an instance $\mathcal{I}$, i.e., we mainly assume $G = \mathcal{G}_{\mathcal{I}}$, but in general the graph representation can be any graph that is suited for table algorithm A in order to solve instance $\mathcal{I}$.

2. LTD: Compute a (labeled) tree decomposition $(T, \chi, \delta_{\mathcal{I}})$ of $\mathcal{I}$ by means of the constructed graph representation G. The resulting decomposition can be obtained by using efficient heuristics via, e.g., the tool htd [Abseher et al., 2017]. Further sources of state-of-the-art decomposers can be found in the latest competition report of this area [Dell et al., 2017]. Alternatively, one can compute a decomposition, whose width is guaranteed to be in $\mathcal{O}(5 \cdot \mathsf{tw}(G))$, which can be obtained in single-exponential time in the treewidth [Bodlaender et al., 2016]. From these decompositions one can easily obtain some LTD like for example the canonical LTD as indicated by Observation 2.15.

3. DP: Run algorithm $\mathtt{DP}_{\mathsf{A}}$, which executes table algorithm A for every node t of T in post-order, and returns A-Tabs mapping every node t to its table. Algorithm DP works for given table algorithm A as presented in Listing 3.1 and takes as input instance $\mathcal{I}$ and a labeled tree decomposition $\mathcal{T}$. Table algorithm A is executed

for a specific node t of T and takes as input the corresponding bag $\chi(t)$ of t, the assigned instance $\delta_{\mathcal{I}}(t)$ for node t, as well as a sequence of child tables previously computed during the post-order traversal for child nodes of t in T, and outputs a table τ_t. For simplicity and the ease of presentation, the table algorithms presented in this thesis are *specified for nice (labeled) tree decompositions* due to clear case distinctions depending on type(t). However, recall that this is not a hard restriction, cf. Proposition 2.6 and Observation 2.15. Further, it is easy to see that for arbitrary LTDs the clear case distinctions of a table algorithm presented on nice LTDs are still valid, but are in general just overlapping.

4. Result: Print the solution to $\mathcal{I}$ by interpreting the table for root $n = \text{root}(T)$ of T. This works for decision and counting problems. If solutions of $\mathcal{I}$ shall be enumerated, one needs to retrace back the predecessors of rows in the table for the root node towards leaf nodes of T. Since this thesis mainly concerns about decision and counting problems, for details on dynamic programming on tree decompositions for enumeration problems we refer to related work [Pichler et al., 2010].

So far, we did not discuss a table algorithm in details. Indeed, such table algorithms can be designed for several problems $\mathtt{P}$. Overall, the conceptual idea of these table algorithms follows a similar pattern. Such a table algorithm A is executed for an instance $\mathcal{I}$ of $\mathtt{P}$ and an LTD $\mathcal{T} = (T, \chi, \delta_{\mathcal{I}})$ by $\mathtt{DP}_{\mathsf{A}}(\mathcal{I}, \mathcal{T})$ for each node t of the decomposition. Thereby, for each node t, solutions to bag instance $\delta_{\mathcal{I}}(t)$, which is a part of the canonical bag instance $\mathcal{I}_t$, are stored in table τ_t. However, by construction of the table algorithm, these solutions can be extended to solutions of the *instance $\mathcal{I}_{\leq t}$ below t*, which intuitively consists of all instances $\delta_{\mathcal{I}}(t')$ for all nodes t' below t in the tree of $\mathcal{T}$. This concept of an instance $\mathcal{I}_{\leq t}$ below t is formalized for Boolean formulas as follows. Given a formula F, a labeled tree decomposition $\mathcal{T} = (T, \chi, \delta_F)$ of F, and a node t of T. Then, we let the *formula $F_{\leq t}$ below t* be defined by $F_{\leq t} := \bigcup_{t' \text{ in } T[t]} \delta_F(t')$.

A Table Algorithm for Boolean Formulas

Next, we present as an example a table algorithm for model counting, i.e., for solving problem #Sat, which therefore can be used also for deciding Boolean satisfiability (Sat). The table algorithm #Sat we present uses the primal graph representation and is taken from the original source [Samer and Szeider, 2010] and slightly adapted. The main idea of algorithm #Sat is to store in table τ_t rows of the form $\langle I, c\rangle$ with I being an interpretation restricted to bag $\chi(t)$ and $c \in \mathbb{N}^+$ being a positive counter, i.e., $c > 0$. Thereby, table algorithm #Sat guarantees that for each row $\langle I, c\rangle$ in a table τ_t we have that I is also a model of $\delta_F(t)$, where we have c many distinct assignments J over variables $\mathsf{var}(F_{\leq t}) \setminus \chi(t)$ such that $I \cup J$ is a model of formula $F_{\leq t}$.

Table algorithm #Sat of Listing 3.2 transforms at node t certain row combinations of the tables (Child-Tabs) of child nodes of t into rows of table τ_t. The transformation depends on the cases of type(t), where either the decomposition node t is an empty leaf node

Listing 3.2: Table algorithm $\#\mathsf{Sat}_t(\chi_t, F'_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving #SAT [Samer and Szeider, 2010].

In: Node t, bag χ_t, bag formula F'_t, and sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child #Sat-tables of t.
Out: #Sat-table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t \leftarrow \{\langle \emptyset, 1 \rangle\}$ `/* Abbreviations` I^+, $I^\sim$ `given below. */`
2 **else if** type$(t) =$ *intr and* $a \in \chi_t$ *is the introduced variable* **then**
3 $\quad \tau_t \leftarrow \{\langle J, c \rangle \quad | \ \langle I, c \rangle \in \tau_1, J \in \{I^+_{a \mapsto 0}, I^+_{a \mapsto 1}\}, J \models F'_t\}$
4 **else if** type$(t) =$ *rem and* $a \notin \chi_t$ *is the removed variable* **then**
$\quad$ `/* C`(I) `is the set that contains the rows in` τ_1 `for assignments`
$\quad\quad J$ `that are equal to` I `after removing` a `*/`
5 $\quad \mathrm{C}(I) \leftarrow \{u \mid u \in \tau_1, u = \langle J, c \rangle, J^\sim_{\{a \mapsto 0, a \mapsto 1\}} = I^\sim_{\{a \mapsto 0, a \mapsto 1\}}\}$
6 $\quad \tau_t \leftarrow \{\langle I^\sim_{\{a \mapsto 0, a \mapsto 1\}}, \sum_{\langle J, c \rangle \in C(I)} c \rangle\} \quad | \ \langle I, \cdot \rangle \in \tau_1\}$
7 **else if** type$(t) =$ *join* **then**
8 $\quad \tau_t \leftarrow \{\langle I, c_1 \cdot c_2 \rangle \quad | \ \langle I, c_1 \rangle \in \tau_1, \langle I, c_2 \rangle \in \tau_2\}$
9 **return** τ_t

For sets S, S' and an element e, we abbreviate $S^\sim_{S'} := S \setminus S'$, $S^+_s := S \cup \{s\}$.

(*leaf*), a variable a is added to an interpretation (*intr*), a variable a is removed from an interpretation (*rem*), or where coinciding interpretations of the different child tables are required to be merged (*join*). For a leaf node t, i.e., if type$(t) =$ *leaf*, we construct only the empty (one) assignment as specified in Line 1 of Listing 3.2. Intuitively, whenever a variable a is introduced in a node t, we guess whether we assign a to true or to false and check whether the resulting assignment satisfies bag formula $F'_t = \delta_F(t)$, cf. Line 3. Then, we ensure in Line 6 that whenever an atom a is removed in node t, the assignments contained in table τ_t do not assign a. Note that this operation potentially "merges" assignments that are equal in τ_t, but were different in the table of the child node of t due to a. Consequently, one has to sum up in Line 6 corresponding counters accordingly, which is done with the help of collecting these counters in Line 5 of Listing 3.2. For join nodes, we need to multiply counters of rows with coinciding assignments accordingly, as given by Line 8. In the end, table algorithm #Sat ensures that an interpretation I from a row $u = \langle I, c \rangle$ of the table τ_n at the root $n = \mathrm{root}(T)$ proves that there are c many different assignments J to variables $\mathsf{var}(F_{\leq n}) \setminus \chi(n)$ such that $I \cup J$ is also a model of $F = F_{\leq n}$, and hence all of these c many different models prove that the formula is satisfiable.

Therefore, one can decide Boolean satisfiability with algorithm $\mathtt{DP}_{\#\mathsf{Sat}}$, even when table algorithm #Sat is restricted to only storing interpretations, i.e., for simply deciding satisfiability only the first element of each row is required. We refer by Sat to the resulting table algorithm that is obtained from table algorithm #Sat as given in Listing 3.2 by only considering the first row positions (only storing interpretations).

Example 3.1. *Consider formula*

$$F := \{\overbrace{\{\neg a, b, c\}}^{c_1}, \overbrace{\{a, \neg b, \neg c\}}^{c_2}, \overbrace{\{a, d\}}^{c_3}, \overbrace{\{a, \neg d\}}^{c_4}\}.$$

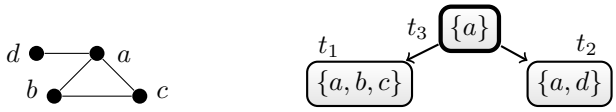

Figure 3.2: Primal graph $\mathcal{G}_F$ of F from Example 3.1 (left) with a TD $\mathcal{T}$ of $\mathcal{G}_F$ (right).

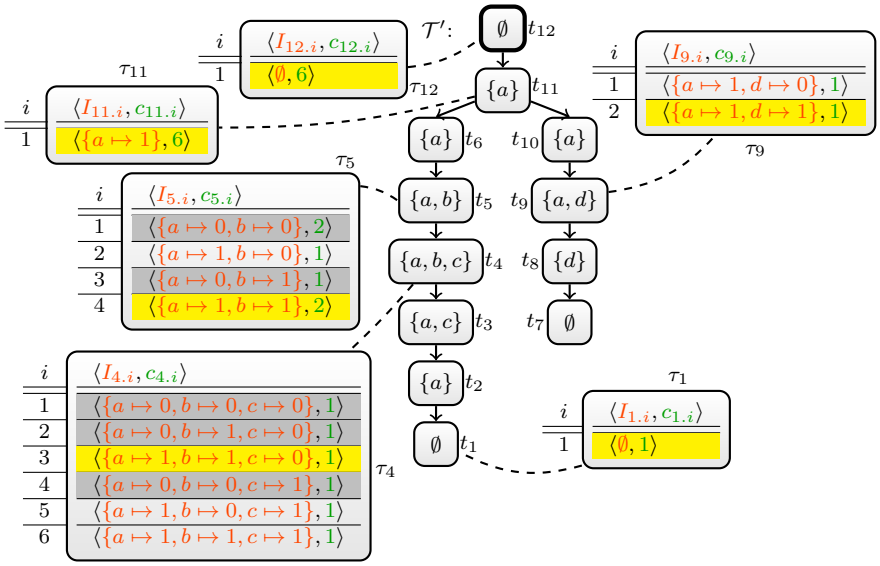

Figure 3.3: A nice TD $\mathcal{T}'$ of $\mathcal{G}_F$ (cf. formula F from Example 3.1) as well as selected tables obtained by $\mathtt{DP}_{\#\mathsf{Sat}}$ on F and the canonical LTD of $\mathcal{T}'$.

Satisfying assignments of formula F are, e.g., $\{a \mapsto 1, b \mapsto 1, c \mapsto 0, d \mapsto 0\}$, $\{a \mapsto 1, b \mapsto 0, c \mapsto 1, d \mapsto 0\}$ or $\{a \mapsto 1, b \mapsto 1, c \mapsto 1, d \mapsto 1\}$. In total, there are 6 satisfying assignments of F. Observe that Figure 3.2 depicts the primal graph $\mathcal{G}_F$ of F. Intuitively, the tree decomposition $\mathcal{T}$ of Figure 3.2 allows to evaluate formula F in parts. Figure 3.3 illustrates a nice TD $\mathcal{T}' = (T, \chi)$ of the primal graph $\mathcal{G}_F$. In this example we use the canonical LTD (T, χ, δ_F) of TD $\mathcal{T}'$ as given in Observation 2.15, where for every node t of T, labeling δ_F is defined by $\delta_F(t) := F_t$. Figure 3.3 further depicts tables $\tau_1, \ldots, \tau_{12}$ that are obtained during the execution of $\mathtt{DP}_{\#\mathsf{Sat}}$ on nodes $t_1, \ldots, t_{12}$. We assume that each row in a table τ_t is identified by a number, i.e., row i corresponds to $\vec{u_{t.i}} = \langle I_{t.i}, c_{t.i} \rangle$.

Table $\tau_1 = \{\langle \emptyset, 1 \rangle\}$ has $\mathrm{type}(t_1) = \mathit{leaf}$. Since $\mathrm{type}(t_2) = \mathit{intr}$, we construct table τ_2 from τ_1 by taking $I_{1.i} \cup \{a \mapsto 0\}$ and $I_{1.i} \cup \{a \mapsto 1\}$ for each $\langle I_{1.i}, c_{1.i} \rangle \in \tau_1$. Then, t_3 introduces c and t_4 introduces b. $F_{t_1} = F_{t_2} = F_{t_3} = \emptyset$, but since $\chi(t_4) \subseteq \mathsf{var}(c_1)$ and $\chi(t_4) \subseteq \mathsf{var}(c_2)$, we have $F_{t_4} = \{c_1, c_2\}$ for t_4. In consequence, for each $I_{4.i}$ of table τ_4, we have $\{c_1, c_2\}(I_{4.i}) = \emptyset$ since $\#\mathsf{Sat}$ enforces satisfiability of F_t in node t. Since $\mathrm{type}(t_5) = \mathit{rem}$, we remove variable c from all elements in τ_4 and sum up counters accordingly to construct τ_5. Note that we have already seen all rules where c occurs and hence c can no longer affect interpretations during the remaining traversal. We similarly create $\tau_6 = \{\langle \{a \mapsto 0\}, 3 \rangle, \langle \{a \mapsto 1\}, 3 \rangle\}$ and $\tau_{10} = \{\langle \{a \mapsto 1\}, 2 \rangle\}$. Since

$\text{type}(t_{11}) = \textit{join}$, *we build table* τ_{11} *by taking the intersection of* τ_6 *and* τ_{10}. *Intuitively, this combines assignments agreeing on* a, *where counters are multiplied accordingly. By definition (primal graph and TDs), for every* $c \in F$, *variables* $\mathsf{var}(c)$ *occur together in at least one common bag. Hence, since* $\tau_{12} = \{\langle \emptyset, 6 \rangle\}$, *we can reconstruct for example model* $\{a \mapsto 1, b \mapsto 1, c \mapsto 0, d \mapsto 1\} = I_{11.1} \cup I_{5.4} \cup I_{4.3} \cup I_{9.2}$ *of* F *using rows in Figure 3.3 that are highlighted in yellow. On the other hand, if* F *was unsatisfiable,* τ_{12} *would contain no values, i.e.,* $\tau_{12} = \emptyset$. *Rows that are marked grey in Figure 3.3 in the end do not contribute to any model of* F.

It is easy to see that the runtime of algorithm #Sat is single exponential in the treewidth. For the analysis, we assume $\gamma(n)$ to be the costs for multiplying two n-bit integers, which can be achieved in time $n \cdot \log(n) \cdot \log(\log(n))$ [Knuth, 1998; Harvey et al., 2016].

Proposition 3.2 (cf. [Samer and Szeider, 2010])**.** *Given a Boolean formula* F *and a nice LTD* $\mathcal{T} = (T, \chi, \delta_F)$ *of* $\mathcal{G}_F$ *of width* k *with* g *nodes. Then, algorithm* $\mathtt{DP}_{\#\mathsf{Sat}}$ *runs in time* $\mathcal{O}(2^k \cdot \gamma(|\mathsf{var}(F)|) \cdot g \cdot \|F\|)$.

Proof (Sketch). Let $d = k + 1$ be the maximum bag size of the labeled tree decomposition $\mathcal{T}$. By construction of #Sat, the table τ_t has at most $\mathcal{O}(2^d)$ rows. In total, with the help of efficient data structures, e.g., for nodes t with $\text{type}(t) = \textit{join}$, one can establish a runtime bound of $\mathcal{O}(2^d \cdot \gamma(|\mathsf{var}(F)|) \cdot \|F\|)$ due to the multiplication of numbers for join nodes. Then, we apply this to every node t of T, which results in a running time $\mathcal{O}(2^d \cdot \gamma(|\mathsf{var}(F)|) \cdot g \cdot \|F\|)$. □

Lower Bound via a Decomposition-Guided Reduction

Recall the exponential time hypothesis (ETH) of Hypothesis 2.1 and that under the ETH, it is not expected that one can significantly improve this result, cf. Proposition 2.10. Even further, it is easy to see that the same lower bound holds for answer set programming, namely already for the related problem Models. Recall that this problem Models is rather similar to Sat, but asks for the existence of a model of a logic program, instead of satisfiability of a Boolean formula.

Proposition 3.3. *Unless the ETH fails, one cannot decide* Models *for any given program* Π *in time* $2^{o(\mathsf{tw}(\mathcal{G}_\Pi))} \cdot \text{poly}(|\mathsf{var}(\Pi)|)$.

Proof (Sketch). Take any Boolean formula F of problem Sat. From this, we construct a tight program Π as follows. Thereby, we use as atoms of Π the variables of $\mathsf{var}(F)$ and for each variable $v \in \mathsf{var}(F)$ we require an additional atom of the form nv. For each variable $v \in \mathsf{var}(F)$, we create the rules $v \leftarrow \neg nv$, $nv \leftarrow \neg v$, as well as $\leftarrow v, nv$. Further, for each formula $c = l_1 \vee \cdots \vee l_\ell$ of F, we create the rule $\leftarrow \overline{l_1}, \ldots, \overline{l_\ell}$, where $\overline{l_i} = \neg v_i$ if $l_i = v_i$ and $\overline{l_i} = v_i$ if $l_i = \neg v_i$ for each $1 \leq i \leq \ell$. It is easy to see that for any satisfying assignment of F there is exactly one model of Π and vice versa. More precisely, for each satisfying assignment I of F, we construct a model M by letting $v \in M$ ($nv \in M$) if

and only if $I(v) = 1$ ($I(nv) = 0$) for every $v \in \mathsf{var}(F)$, respectively. Conversely, for each model M of Π, we construct a satisfying assignment I of F, where $I(v) = 1$ if and only if $v \in M$ for every $v \in \mathsf{var}(F)$.

Indeed, also the treewidth is linearly preserved. This can be witnessed by the following construction, where we assume a given tree decomposition $\mathcal{T} = (T, \chi)$ of $\mathcal{G}_F$ of width $\mathsf{tw}(\mathcal{G}_F)$ and transform it into a tree decomposition of $\mathcal{G}_\Pi$. We refer to the resulting tree decomposition of $\mathcal{G}_\Pi$ by $\mathcal{T}' = (T, \chi')$, and we let χ' be defined as follows. For each node t of T, we let $\chi'(t) := \chi(t) \cup \{nv \mid v \in \mathsf{var}(F)\}$. It is easy to see that $\mathcal{T}'$ is indeed a tree decomposition of $\mathcal{G}_\Pi$. Further, since $|\chi'(t)| = 2 \cdot |\chi(t)|$ for every node t of T, we have that $\mathsf{tw}(\mathcal{G}_\Pi) \in \mathcal{O}(\mathsf{tw}(\mathcal{G}_F))$.

Now, towards a contradiction assume the contrary of this proposition. However, with the reduction above this would result in an algorithm for solving Sat that runs in time $2^{o(\mathsf{tw}(\mathcal{G}_\Pi))} \cdot \mathrm{poly}(|\mathsf{var}(\Pi)|)$. This, however, contradicts the exponential time hypothesis, as stated in Hypothesis 2.1. □

Note that in this proof we first encountered conceptual ideas of what we will call later in Chapter 4 "decomposition-guided reduction". Such a reduction takes a problem instance, but also a tree decomposition of the corresponding primal graph representation of the instance, and the reduction is constructed in such a way that it also gives rise to a tree decomposition of the primal graph of the resulting instance. Later, in Chapters 4 and 5 we will make heavy use of this important concept for different purposes.

3.2 Dynamic Programming for ASP

Next, we present algorithms and runtimes for answer set programming and treewidth for the diverse fragments of programs. Thereby we intuitively proceed in the order of hardness of the fragments, as the dynamic programming algorithms get more involved for harder fragments. More concretely, Section 3.2.1 concerns about a table algorithm for computing supported models, which coincides with answer sets when restricting to tight programs. Then, Section 3.2.2 generalizes this algorithm to the case of normal and head-cycle-free programs. Finally, Section 3.2.3 deals with an algorithm for arbitrary logic programs, thus subsuming disjunctive programs. Each of the discussed table algorithms leads to certain runtimes and therefore one obtains runtime upper bounds.

Before we discuss table algorithms for the diverse fragments of programs, we briefly define the instance below a decomposition node for a program Π, a labeled TD $\mathcal{T} = (T, \chi, \delta_\Pi)$ of Π, and a node t of T as follows. Similar to a formula below a node, we let the *program* $\Pi_{\leq t}$ *below* t be defined by $\Pi_{\leq t} := \bigcup_{t' \text{ in } T[t]} \delta_\Pi(t')$.

3.2.1 Utilizing Treewidth for Tight Asp and Supported Models

One of the easiest fragments of programs in the context of ASP are tight programs. Indeed, the problem Tight Asp is complete for the complexity class NP and therefore considered

Listing 3.3: Table algorithm $\mathsf{SuppAsp}_t(\chi_t, \Pi'_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$.

In: Node t, bag χ_t, bag program Π'_t, and sequence $\langle \tau_1, \ldots, \tau_\ell \rangle$ of child SuppAsp-tables of t.
Out: SuppAsp-table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t \leftarrow \{\langle \emptyset, \emptyset \rangle\}$ `/* Abbreviations are given below. */`
2 **else if** type$(t) =$ *intr and* $a \in \chi_t$ *is the introduced atom* **then**
3 | $\tau_t \leftarrow \{\langle J, \mathcal{P} \cup \text{supported}(J, \Pi'_t) \rangle \mid \langle I, \mathcal{P} \rangle \in \tau_1,\ J \in \{I, I_a^+\},\ J \models \Pi'_t\}$
4 **else if** type$(t) =$ *rem and* $a \notin \chi_t$ *is the removed atom* **then**
5 | $\tau_t \leftarrow \{\langle I_a^-, \mathcal{P}_a^- \rangle \mid \langle I, \mathcal{P} \rangle \in \tau_1, I \cap \{a\} \subseteq \mathcal{P}\}$
6 **else if** type$(t) =$ *join* **then**
7 | $\tau_t \leftarrow \{\langle I, \mathcal{P}_1 \cup \mathcal{P}_2 \rangle \mid \langle I, \mathcal{P}_1 \rangle \in \tau_1, \langle I, \mathcal{P}_2 \rangle \in \tau_2\}$
8 **return** τ_t

For a set S and an element e, we abbreviate $S_e^+ := S \cup \{e\}$ and $S_e^- := S \setminus \{e\}$.

to be of similar complexity than deciding Boolean satisfiability (SAT). Consequently, the characterization of answer sets when restricted to tight programs is rather simple: An answer set I of a tight program Π (i) is a model of the program, namely, $I \models \Pi$, i.e., it satisfies all the rules and (ii) every atom $a \in I$ has to be supported, which is the case if there is a rule $r \in \Pi$ supporting a with I. Instead of dealing with truth assignments, for solving ASP we use interpretations, which are sets of those atoms that have the intended meaning of being assigned to true. Note that it is easy to see that for tight programs no level mapping for deciding provability is needed, i.e., we have provability for an atom if it is supported. Indeed in order to obtain provability, it is sufficient [Janhunen, 2006] to have level mappings per strongly connected component (SCC) of the dependency graph D_Π and for tight program Π the dependency graph D_Π does not contain any non-trivial SCC and therefore no cycle at all. The resulting characterization amounts to ideas similar to Clark's completion [Clark, 1977], which when applied to tight programs guarantees already provability of atoms.

Our approach for solving tight programs works similar to Clark's completion, but we need to guide the ideas of Clark's completion, or the determination of support to be precise, along a tree decomposition in order to not cause runtime overhead for treewidth. Recall that in Listing 3.2 above, the constructed rows consist of an assignment (plus a counter for counting). Here, instead of an assignment, every row consists of an *interpretation*, which reflects the semantics of ASP and is from a technical perspective also easier to handle than truth assignments, as well as a set of atoms for remembering *supported atoms* of the interpretation. More concretely, we design a table algorithm SuppAsp, whose rows consist of an interpretation $I \subseteq \chi(t)$ and a set of atoms $\mathcal{P} \subseteq I$ marking supported atoms of I. Let Π be a tight program, $\mathcal{T} = (T, \chi, \delta_\Pi)$ be a labeled tree decomposition of Π, and t be a node of T. Then, table algorithm SuppAsp ensures for each row $u = \langle I, \mathcal{P} \rangle$ of a SuppAsp-table τ_t the following: First, we have (1) $I \models \delta_\Pi(t)$ and that there is an interpretation $J \subseteq \text{at}(\Pi_{\leq t}) \setminus \chi(t)$ such that I can be extended to a model $I' = I \cup J$, i.e., $I' \models \Pi_{\leq t}$, where every atom in J is supported. Further, the table algorithm ensures that (2) for an atom $a \in I$ we have $a \in \mathcal{P}$ if and only if a is supported by I using $\Pi_{\leq t}$.

This leads us to the following definition of SuppAsp-rows as well as SuppAsp-tables.

Definition 3.4. *Given a program Π, an LTD (T, χ, δ_Π) of $\mathcal{G}_\Pi$, and a node t of T. Then, a* SuppAsp-row *for t is a sequence of the form $\langle I, \mathcal{P}\rangle$ with $I \subseteq \chi(t)$ and $\mathcal{P} \subseteq I$ that ensures the following:*

1. *$I \models \delta_\Pi(t)$ and*
2. *there exists $I' \subseteq \mathrm{at}(\Pi_{\leq t}) \setminus \chi(t)$ such that*
   - *$(I \cup I') \models \Pi_{\leq t}$,*
   - *each atom in I' is supported by a rule in $\Pi_{\leq t}$, and*
   - *we have that $a \in \mathcal{P}$ if and only if a is supported by a rule in $\Pi_{\leq t}$*

Finally, the SuppAsp-table *for node t is the largest set of* SuppAsp*-rows for t.*

Next, we discuss table algorithm SuppAsp as presented in Listing 3.3 in more details. To this end, we define for a given interpretation I and a program Π' the function $\mathrm{supported}(I, \Pi') := \{a \mid r \in \Pi', r \text{ supports } a \text{ with } I\}$. Then, for leaf nodes t where $\mathrm{type}(t) = \mathit{leaf}$ we only set the empty interpretation and the empty set of supported atoms as given in Line 1 of Listing 3.3. Intuitively, whenever an atom a is introduced in a node t, i.e., $\mathrm{type}(t) = \mathit{intr}$, we decide in Line 3 whether we include a in the interpretation, ensure that the resulting interpretation is a model of bag program $\Pi'_t = \delta_\Pi(t)$, and determine bag atoms that are supported in consequence of this decision. When removing an atom a in a node t of type *rem*, in Line 5 we only keep those rows where either $a \notin I$ or a has been supported and is therefore contained in $\mathcal{P}$, and then we remove from all remaining rows a accordingly. In case the node is of type *join*, we combine two rows in two different child tables, where, intuitively, in Line 7 we are enforced to agree on interpretations I. However, concerning individual supported atoms $\mathcal{P}$, it suffices that an atom is supported in at least *one* of the rows.

Example 3.5. *Recall the program $\Pi := \{\overbrace{a \vee b \leftarrow}^{r_1}; \overbrace{c \vee e \leftarrow d}^{r_2}; \overbrace{d \leftarrow b, \neg e}^{r_3}; \overbrace{e \leftarrow b, \neg d}^{r_4}; \overbrace{b \leftarrow e, \neg d}^{r_5}; \overbrace{d \leftarrow \neg b}^{r_6}\}$. from Example 2.4. Figure 3.4 depicts a TD $\mathcal{T} = (T, \chi)$ of the primal graph $\mathcal{G}_\Pi$ of Π. Further, the figure illustrates a snippet of tables, which we obtain when running $\mathsf{DP}_{\mathsf{SuppAsp}}$ on program Π and the canonical LTD (T, χ, δ_Π) of $\mathcal{T}$ (cf. Observation 2.15) according to Listing 3.3.*

In the following, we briefly discuss some selected rows of some selected tables. Note that for simplicity we write τ_j instead of τ_{t_j} and identify rows by their node and identifier i in the figure. For example, the row $\vec{u}_{13.2} = \langle I_{13.2}, \mathcal{P}_{13.2}\rangle \in \tau_{13}$ refers to the second row of table τ_{13} for node t_{13}. Node t_1 is of type leaf and consequently table τ_1 has only one row, which consists of the empty interpretation and the empty set of proven atoms (Line 1). Node t_2 is of type intr and introduces atom a. Executing Line 3 results in $\tau_2 = \{\langle\emptyset, \emptyset\rangle, \langle\{a\}, \emptyset\rangle\}$. Node t_3 is of type intr and introduces b. Then, bag program

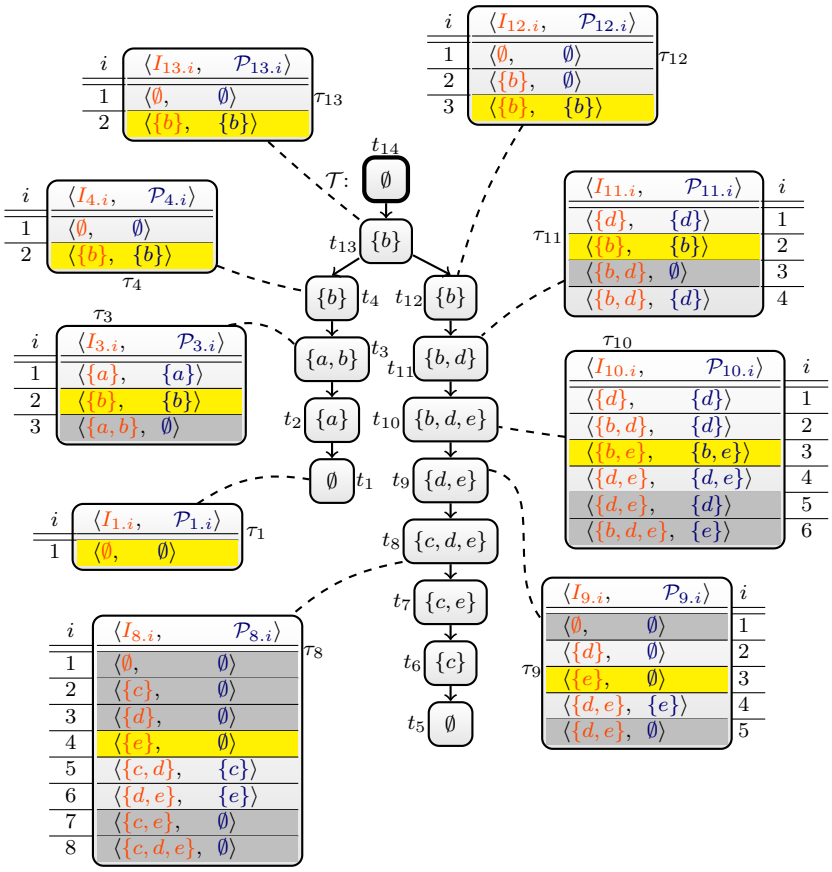

Figure 3.4: A nice TD $\mathcal{T}$ of $\mathcal{G}_\Pi$ (cf. program Π from Example 2.4) as well as selected tables obtained by $\mathsf{DP}_{\mathsf{SuppAsp}}$ on Π and the canonical LTD of $\mathcal{T}$.

at node t_3 is $\Pi_{t_3} = \delta_\Pi(t_3) = \{a \vee b \leftarrow\}$. By construction (Line 3) we ensure that interpretation $I_{3.i}$ is a model of Π_{t_3} for every row $\langle I_{3.i}, \mathcal{P}_{3.i}\rangle$ in τ_3.

Node t_4 is of type rem. Here, we restrict the rows such that they contain only atoms occurring in bag $\chi(t_4) = \{b\}$. To this end, Line 5 takes only rows $\vec{u}_{3.i}$ of table τ_3 where atoms in $I_{3.i}$ are also proven, i.e., contained in $\mathcal{P}_{3.i}$. In particular, every row in table τ_4 originates from at least one row in τ_3 where either $a \in \mathcal{P}_{3.i}$ is supported or where $a \notin I_{3.i}$. Basic conditions of a TD ensure that once an atom is removed, it will not occur in any bag at an ancestor node. Hence, we also encountered all rules where atom a occurs. Nodes t_5, t_6, t_7 are symmetric to nodes t_1, t_2, t_3. Then, node t_8 introduces an atom, t_9 removes an atom and node t_{10} again introduces an atom. Observe that $\mathcal{P}_{10.2} = \{d\}$ since d is supported by rule r_3, whereas b cannot be supported by any rule in $\delta_\Pi(t_{10}) = \Pi_{t_{10}}$. However, $\mathcal{P}_{10.3} = \{b, e\}$ since both atoms b and e are supported by rules r_5 and r_4, respectively. In particular, in row $\vec{u}_{10.3}$ atom e is used to derive b and atom b is used to derive e. Note that such a "cyclic support" is not permitted if one aims for computing answer sets. We proceed similar for nodes t_{11} and t_{12}. At node t_{13} we join tables τ_4

and τ_{12} according to Line 7.

Finally, $\tau_{14} \neq \emptyset$, i.e., Π has a supported model. One such model can be constructed by joining interpretations I of yellow marked rows of Figure 3.4 represent supported model $\{b, e\}$. Similar to previous examples, those rows that are highlighted gray do not participate in any supported model of Π. Interestingly, after recalling that program Π is not tight (cf. the positive dependency graph D_Π of Figure 2.1), one might observe that for Π its supported models coincide with the answer sets of Π. The reason is that while atom b of answer set $\{b, e\}$ is supported by rule r_5 as determined in this example, b can also be proven by rule r_1 and therefore $\{b, e\}$ is indeed also an answer set of Π. There are no further supported models of Π containing all atoms of a positive cycle and hence we conclude that the supported models coincide with the answer sets of Π.

An example of a more general case of this algorithm, namely for computing answer sets of normal and head-cycle-free programs will be discussed in the next section (cf. Example 3.12).

The following theorem on correctness assumes nice, labeled tree decompositions. However, this restriction is imposed due to the ease of presentation and it is not a hard restriction, cf. Proposition 2.7.

Theorem 3.6. *The algorithm $\mathsf{DP}_{\mathsf{SuppAsp}}$ is correct. In other words, given a tight program Π and a nice LTD $\mathcal{T} = (T, \chi, \delta_\Pi)$ of $\mathcal{G}_\Pi$. Then, $\mathsf{DP}_{\mathsf{SuppAsp}}(\Pi, \mathcal{T})$ computes for each node t of T its* SuppAsp*-table. Consequently, Π has an answer set if and only if $\langle \emptyset, \emptyset \rangle$ is a* SuppAsp*-row for* $\mathrm{root}(T)$.

Proof (Idea). The actual proof is a special case that follows from a more general result later, cf. Theorem 3.15. The idea is as follows. Correctness consists of both soundness and completeness. For soundness, we show the invariant given by Definition 3.4 for each table τ_t that is computed by table algorithm SuppAsp, thereby assuming a set of SuppAsp-rows for each child node of t. More precisely for every node t of T, we have that SuppAsp only computes SuppAsp-rows, given a set of SuppAsp-rows for each child node of t. Further, completeness is shown in a top-down manner, where we assume that SuppAsp computes a SuppAsp-table for the parent node of t and show then that also the set of SuppAsp-rows obtained by table algorithm SuppAsp for node t is actually a SuppAsp-table. □

This algorithm SuppAsp allows us to compute supported models for more general programs Π.

Corollary 3.7. *Let Π be a program and $\mathcal{T} = (T, \chi, \delta_\Pi)$ be a nice LTD of Π. Then, Π has a supported model if and only if $\langle \emptyset, \emptyset \rangle$ is a* SuppAsp*-row for* $\mathrm{root}(T)$.

Proof. The proof immediately follows by the proof of Theorem 3.6 and the definition of supported models. □

Theorem 3.8. *Given a program* Π *and an LTD* $\mathcal{T} = (T, \chi, \delta_\Pi)$ *of* $\mathcal{G}_\Pi$ *of width* k *with* g *nodes. Then, algorithm* $\mathsf{DP}_{\mathsf{SuppAsp}}$ *runs in time* $\mathcal{O}(3^k \cdot g \cdot \|\Pi\|) = 2^{\mathcal{O}(k)} \cdot g \cdot \|\Pi\|$.

Proof (Sketch). The actual proof follows from a more general result later, cf. Theorem 3.16 and therefore we only provide a brief sketch here. Let $d = k + 1$ be the maximum bag size of the labeled tree decomposition $\mathcal{T}$. The table τ_t has at most $\mathcal{O}(3^d)$ rows, since for a SuppAsp-row $\langle I, \mathcal{P}\rangle$ we have by construction of algorithm SuppAsp that an atom can be either in I, both in I and $\mathcal{P}$, or neither in I nor in $\mathcal{P}$. In total, with the help of efficient data structures, e.g., for nodes t with type$(t) = \mathit{join}$, one can establish a runtime bound of $\mathcal{O}(3^d \cdot \|\Pi\|)$. Then, we apply this to every node t of T, which results in a running time $\mathcal{O}(3^d \cdot g \cdot \|\Pi\|) \subseteq 2^{\mathcal{O}(k)} \cdot g \cdot \|\Pi\|$. □

Proposition 3.9. *Unless the ETH fails, neither problem* Supported Models *nor problem* Tight Asp *can be solved in time* $2^{o(k)} \cdot \mathrm{poly}(|\mathrm{at}(\Pi)|)$*, where* Π *is a program (instance) of the respective problem* Supported Models *or* Tight Asp *and* k *is the treewidth of* $\mathcal{G}_\Pi$.

Proof. This corollary is a direct consequence of Proposition 3.3, since Sat can be solved with the same program Π, as constructed in Proposition 3.3, via problem Supported Models or Tight Asp. □

3.2.2 Solving Normal Asp and HCF Asp for Treewidth

The idea of the previous subsection, namely table algorithm SuppAsp, can be generalized in order to decide Asp for normal and even for head-cycle-free (HCF) programs. Recall that compared to tight programs, a normal program can have cyclic dependencies in the positive dependency graph, which makes it harder to analyze whether an atom is indeed justified (proven). Our approach to resolve this challenge is to implicitly apply along the TD the characterization of answer sets by Lin and Zhao [2003] relying on level mappings, but extended to head-cycle-free programs [Ben-Eliyahu and Dechter, 1994]. To this end, let $\mathcal{T} = (T, \chi, \delta_\Pi)$ be an LTD of Π and t be a node of $\mathcal{T}$. Then, we store in table τ_t at each node t rows of the form $\langle I, \mathcal{P}, \sigma\rangle$. The first position consists of an interpretation I restricted to the bag $\chi(t)$. The second position consists of a set $\mathcal{P} \subseteq I$ that represents atoms in I for which we know that they have already been proven. For the third position σ, we need to define I-distinct level mappings as follows.

Definition 3.10. *Let* Π *be a program,* $\mathcal{T} = (T, \chi)$ *be a TD of* $\mathcal{G}_\Pi$*,* t *be a node of* T*, and* $I \subseteq \chi(t)$*. Then, we refer to a level mapping* $\sigma : I \to \mathbb{N}$ *over* I *such that* $\sigma(a) \neq \sigma(b)$ *for every* $a, b \in \chi(t)$ *with* $a \neq b$*, by* I-distinct level mapping*. The* I*-distinct level mapping* σ *is* compatible *with a* J*-distinct level mapping* σ' *if for every* $a, b \in I \cap J$ *with* $a \neq b$ *we have* $\sigma(a) < \sigma(b)$ *whenever* $\sigma'(a) < \sigma'(b)$*.*

So, the third position σ is an I-distinct level mapping. Then, our table algorithm HCFAsp ensures for a row $\langle I, \mathcal{P}, \sigma\rangle$ in a node t, $I \models \delta_\Pi(t)$ and that I can be extended to an

Listing 3.4: Table algorithm $\mathsf{HCFAsp}_t(\chi_t, \Pi'_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$.

In: Node t, bag χ_t, bag program Π'_t, and sequence $\langle \tau_1, \ldots, \tau_\ell \rangle$ of child HCFAsp-tables of t.
Out: HCFAsp-table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t \leftarrow \{\langle \emptyset, \emptyset, \emptyset \rangle\}$ `/* Abbreviations are given below. */`
2 **else if** type$(t) =$ *intr and* $a \in \chi_t$ *is the introduced atom* **then**
3 $\quad \tau_t \leftarrow \{\langle J, \mathcal{P} \cup \text{proven}(J, \varphi', \Pi'_t), \varphi' \rangle$
$\qquad\qquad | \langle I, \mathcal{P}, \varphi \rangle \in \tau_1,\ J \in \{I, I_a^+\},\ J \models \Pi'_t,\ \varphi' \in \text{levelmaps}(\varphi, J)\}$
4 **else if** type$(t) =$ *rem and* $a \notin \chi_t$ *is the removed atom* **then**
5 $\quad \tau_t \leftarrow \{\langle I_a^-, \mathcal{P}_a^-, \varphi_a^- \rangle \quad | \langle I, \mathcal{P}, \varphi \rangle \in \tau_1, I \cap \{a\} \subseteq \mathcal{P}\}$
6 **else if** type$(t) =$ *join* **then**
7 $\quad \tau_t \leftarrow \{\langle I, \mathcal{P}_1 \cup \mathcal{P}_2, \varphi \rangle \quad | \langle I, \mathcal{P}_1, \varphi \rangle \in \tau_1, \langle I, \mathcal{P}_2, \varphi \rangle \in \tau_2\}$
8 **return** τ_t

For a set S and an element e, we abbreviate $S_e^+ := S \cup \{e\}$ and $S_e^- := S \setminus \{e\}$. For an S-distinct level mapping φ, we let φ_e^- be the S_e^--distinct level mapping compatible with σ.

interpretation $J := I \cup I'$, where $I' \subseteq \text{at}(\Pi_{\leq t}) \setminus \chi(t)$ such that $J \models \Pi_{\leq t}$. Even further, it guarantees that σ can be extended to a level mapping σ' over $\text{at}(\Pi_{\leq t})$ such that every atom in $I' \cup \mathcal{P}$ is proven by a rule in $\Pi_{\leq t}$ with σ'. In the end, an interpretation I of a row $\vec{u}$ of the table τ_n at the root n proves that there is a superset $I' \supseteq I$ that is an answer set of $\Pi = \Pi_{\leq n}$.

Definition 3.11. *Given an HCF program Π, an LTD (T, χ, δ_Π) of $\mathcal{G}_\Pi$, and a node t of T. Then, an* HCFAsp-row (for t) *is a sequence of the form $\langle I, \mathcal{P}, \sigma \rangle$ with $I \subseteq \chi(t)$, $\mathcal{P} \subseteq I$, and σ being an I-distinct level mapping, such that the following holds:*

1. *$I \models \delta_\Pi(t)$ and*
2. *there exists $I' \subseteq \text{at}(\Pi_{\leq t}) \setminus \chi(t)$ and an $(I \cup I')$-distinct level mapping σ' such that*
   - *$(I \cup I') \models \Pi_{\leq t}$,*
   - *σ is compatible with σ',*
   - *each atom in $\mathcal{P} \cup I'$ is proven by a rule in $\Pi_{\leq t}$ with σ', and*
   - *we have that $a \in \mathcal{P}$ if and only if a is proven by a rule in $\Pi_{\leq t}$ with σ'*

We refer to $\langle I \cup I', \mathcal{P} \cup I', \sigma' \rangle$ by a (corresponding) HCFAsp-solution *of $\langle I, \mathcal{P}, \sigma \rangle$.*

Further, the HCFAsp-table (for t) *is the largest set of* HCFAsp*-rows for t.*

Listing 3.4 presents the algorithm HCFAsp. Intuitively, whenever an atom a is introduced (*intr*), we decide whether we include a in the interpretation, determine bag atoms that can be proven in consequence of this decision, and update the level mapping σ accordingly. To this end, we define for a given interpretation I and an I-distinct level mapping σ the set $\text{proven}(I, \sigma, \Pi_t) := \cup_{r \in \Pi_t, a \in H_r} \{a \mid B_r^+ \subseteq I, I \cap B_r^- = \emptyset, I \cap (H_r \setminus \{a\}) = \emptyset, B_r^+ \prec_\sigma a\}$ where $B_r^+ \prec_\sigma a$ holds if $\sigma(b) < \sigma(a)$ is true for every $b \in B_r^+$. Moreover, given an I-distinct level mapping σ and a set $I' \supseteq I$ of atoms, we compute

an I'-distinct level mappings extending σ. Therefore, we let levelmaps$(\sigma, I') := \{\sigma' \mid \sigma'$ is an I'-distinct level mapping compatible with $\sigma\}$. When removing (*rem*) an atom a, we only keep those rows where a has been proven (contained in $\mathcal{P}$) and then restrict remaining rows to the bag (not containing a). In case the node is of type *join*, we combine two rows in two different child tables, intuitively, we are enforced to agree on interpretations I and level mappings σ. However, concerning individual proofs $\mathcal{P}$, it suffices that an atom is proven in *one* of the rows.

Example 3.12. *Recall the program* $\Pi := \{\overbrace{a \vee b \leftarrow}^{r_1}; \overbrace{c \vee e \leftarrow d}^{r_2}; \overbrace{d \leftarrow b, \neg e}^{r_3}; \overbrace{e \leftarrow b, \neg d}^{r_4}; \overbrace{b \leftarrow e, \neg d}^{r_5}; \overbrace{d \leftarrow \neg b}^{r_6}\}$. *from Example 2.4. Figure 3.5 depicts a TD* $\mathcal{T} = (T, \chi)$ *of the primal graph* $\mathcal{G}_\Pi$ *of* Π. *Further, the figure illustrates a snippet of tables, which we obtain when running* $\mathsf{DP}_{\mathsf{HCFAsp}}$ *on program* Π *and the canonical LTD of TD* $\mathcal{T}$ *according to Listing 3.4. For simplicity, for every row* $\langle I, \mathcal{P}, \sigma\rangle$ *the* I*-distinct level mapping* σ *is depicted as a sequence in Figure 3.5. In the following, we again briefly discuss some selected rows of those tables. Note that for brevity, we write* τ_j *instead of* τ_{t_j} *and identify rows by their node and identifier* i *in the figure. For example, the row* $\vec{u}_{13.3} = \langle I_{13.3}, \mathcal{P}_{13.3}, \sigma_{13.3}\rangle \in \tau_{13}$ *refers to the third row of table* τ_{13} *for node* t_{13}.

Node t_1 *is of type leaf. Table* τ_1 *has only one row, which consists of the empty interpretation, empty set of proven atoms, and the empty sequence (Line 1). Node* t_2 *is of type intr and introduces atom* a. *Executing Line 3 results in* $\tau_2 = \{\langle\emptyset, \emptyset, \langle\rangle\rangle, \langle\{a\}, \emptyset, \langle a\rangle\rangle\}$. *Node* t_3 *is of type intr and introduces* b. *Then, bag program at node* t_3 *is* $\Pi_{t_3} = \{a \vee b \leftarrow\}$. *By construction (Line 3) we ensure that interpretation* $I_{3.i}$ *is a model of* Π_{t_3} *for every row* $\langle I_{3.i}, \mathcal{P}_{3.i}, \sigma_{3.i}\rangle$ *in* τ_3. *Node* t_4 *is of type rem. Here, we restrict the rows such that they contain only atoms occurring in bag* $\chi(t_4) = \{b\}$. *To this end, Line 5 takes only rows* $\vec{u}_{3.i}$ *of table* τ_3 *where atoms in* $I_{3.i}$ *are also proven, i.e., contained in* $\mathcal{P}_{3.i}$. *In particular, every row in table* τ_4 *originates from at least one row in* τ_3 *that either proves* $a \in \mathcal{P}_{3.i}$ *or where* $a \notin I_{3.i}$. *Basic conditions of a TD ensure that once an atom is removed, it will not occur in any bag at an ancestor node. Hence, we also encountered all rules where atom* a *occurs.*

Nodes t_5, t_6, t_7, *and* t_8 *are symmetric to nodes* t_1, t_2, t_3, *and* t_4. *Nodes* t_9 *and* t_{10} *again introduce atoms. Observe that* $\mathcal{P}_{10.4} = \{e\}$ *since* $\sigma_{10.4}$ *does not allow to prove* b *using atom* e. *However,* $\mathcal{P}_{10.5} = \{b, e\}$ *as the sequence* $\sigma_{10.5}$ *allows to prove* b. *In particular, in row* $\vec{u}_{10.5}$ *atom* e *is used to derive* b. *As a result, atom* b *can be proven, whereas ordering* $\sigma_{10.4} = \langle b, e\rangle$ *does not serve in proving* b. *We proceed similar for nodes* t_{11} *and* t_{12}. *At node* t_{13} *we join tables* τ_4 *and* τ_{12} *according to Line 7. Finally,* $\tau_{14} \neq \emptyset$, *i.e.,* Π *has an answer set and in fact joining interpretations* I *of yellow marked rows of Figure 3.5 leads to* $\{b, e\}$. *Recall that similar to before, rows that are highlighted in grey do not contribute to any answer set of* Π.

We are now in the position to state the correctness of our algorithm $\mathsf{DP}_{\mathsf{HCFAsp}}$, consisting of both soundness (Lemma 3.13) as well as completeness (Lemma 3.14). However, note

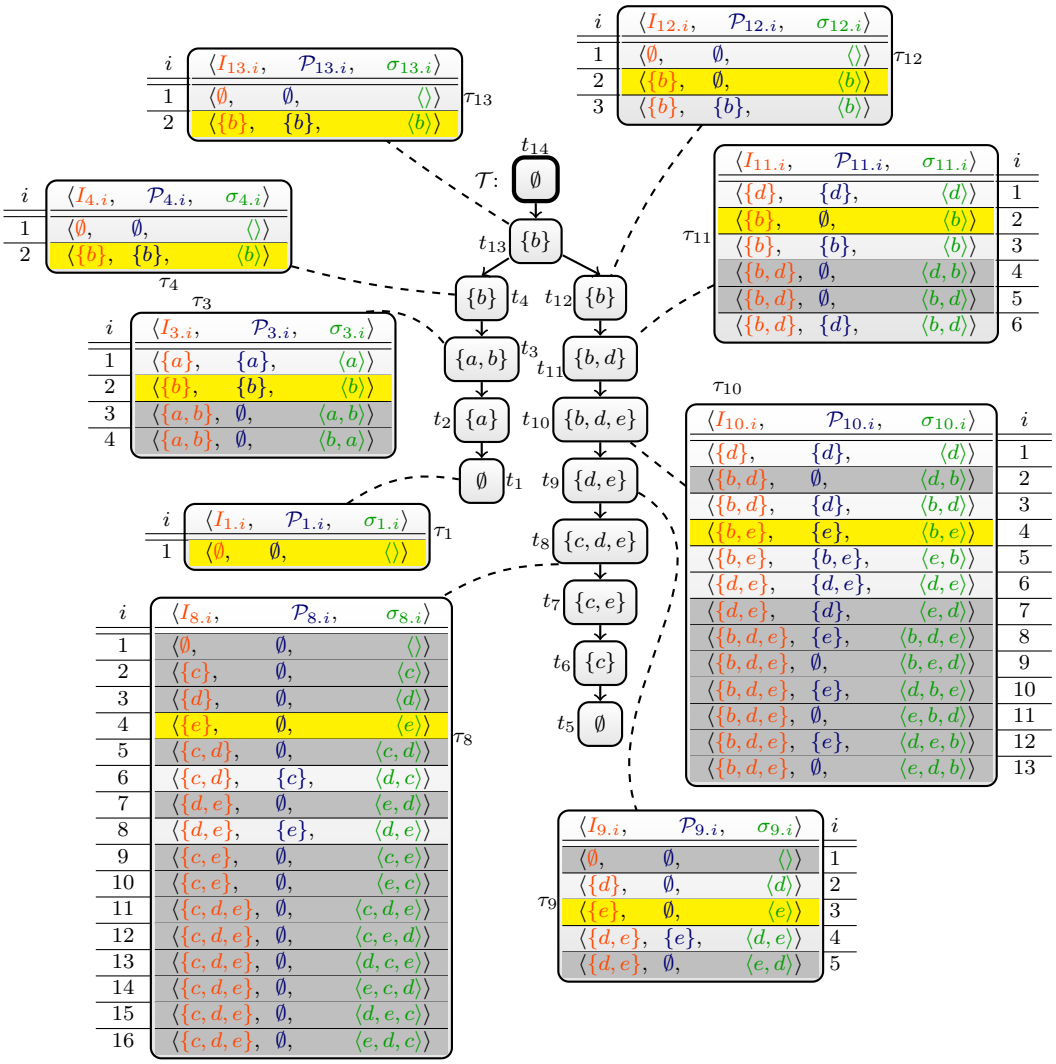

Figure 3.5: A nice TD $\mathcal{T}$ of $\mathcal{G}_\Pi$ (cf. program Π from Example 2.4) as well as selected tables obtained by $\mathsf{DP}_{\mathsf{HCFAsp}}$ on Π and the canonical LTD of $\mathcal{T}$.

that the proofs of these two lemmas are basically technical case distinctions and do not provide further insights than the conditions as already given in Definition 3.11.

Lemma 3.13 (Soundness)**.** *Given an HCF program Π, a nice LTD (T, χ, δ_Π) of Π, and let t be a node of T with* post-children$(t) = \langle t_1, \ldots, t_\ell \rangle$*. Further, let v_i be an* HCFAsp*-row at t_i for $1 \leq i \leq \ell$. Then, each row $u = \langle I, \mathcal{P}, \sigma \rangle$ obtained by* HCFAsp_t*, i.e., contained in* $\mathsf{HCFAsp}_t(\chi(t), \delta_\Pi(t), \langle \{v_1\}, \ldots, \{v_\ell\} \rangle)$ *is also an* HCFAsp*-row at node t.*

Proof (Sketch). We proceed by case distinctions.

Assume Case (i): type(t) = *leaf*. Then, $\langle \emptyset, \emptyset, \emptyset \rangle$ is an HCFAsp-row at t. This concludes Case(i).

Assume Case (ii): type(t) = *intr* and $\chi(t) \setminus \chi(t_1) = \{a\}$. Let $v_1 = \langle I, \mathcal{P}, \sigma \rangle$ be any HCFAsp-row at child node t_1, and $\hat{v_1} = \langle \hat{I}, \hat{\mathcal{P}}, \hat{\sigma} \rangle$ be any corresponding HCFAsp-solution of v_1, which exists by Definition 3.11. In the following, we show that the way HCFAsp transforms HCFAsp-row v_1 at t_1 to an HCFAsp-row $u = \langle I', \mathcal{P}', \sigma' \rangle$ at t is sound. We identify two sub-cases.

Case (a): Atom $a \notin I'$ is set to false. Then, HCFAsp constructs u where $I' = I, \sigma' = \sigma$ and $\mathcal{P}' = \mathcal{P} \cup \text{proven}(I', \sigma', \delta_\Pi(t))$. Note that by construction $I' \models \delta_\Pi(t)$. Towards showing soundness, we define how to transform $\hat{v_1}$ into $\hat{u}$ such that $\hat{u}$ is indeed a corresponding HCFAsp-solution of u constructed by HCFAsp. To this end, we define $\hat{u}$ as follows: $\hat{u} = \langle \hat{I}, \hat{\mathcal{P}} \cup \text{proven}(I', \sigma', \delta_\Pi(t)), \hat{\sigma} \rangle$. Observe that $\hat{u}$ is a corresponding HCFAsp-solution of u according to Definition 3.11. This concludes Case (a).

Case (b): Atom $a \in I'$ is set to true. Conceptually, the case works analogously to Case (a). This concludes Cases (b) and (ii).

Assume Case (iii): type(t) = *rem* and $\chi(t_1) \setminus \chi(t) = \{a\}$. Let $v_1 = \langle I, \mathcal{P}, \sigma \rangle$ be any HCFAsp-row at child node t_1, and $\hat{v_1} = \langle \hat{I}, \hat{\mathcal{P}}, \hat{\sigma} \rangle$ be any corresponding HCFAsp-solution of v_1, which exists by Definition 3.11. Similar to above, we show that the way HCFAsp transforms HCFAsp-row v_1 at t_1 to an HCFAsp-row $u = \langle I', \mathcal{P}', \sigma' \rangle$ at t is sound. Algorithm HCFAsp constructs u where $I' = I \setminus \{a\}, \sigma'$ is the I'-distinct level mapping compatible with σ and $\mathcal{P}' = \mathcal{P} \setminus \{a\}$. Therefore, it is easy to see that $\hat{v_1}$ can be transformed into $\hat{u}$ such that $\hat{u}$ is indeed a corresponding HCFAsp-solution of u constructed by HCFAsp. Observe that already $\hat{u} := \hat{v_1}$ is a corresponding HCFAsp-solution of u according to Definition 3.11.

Assume Case (iv): type(t) = *join*. Observe that $\chi(t_1) = \chi(t_2) = \chi(t)$. Let $v_1 = v_2 = \langle I, \mathcal{P}, \sigma \rangle$ be any HCFAsp-row at child nodes t_1, t_2, and $\hat{v_1} = \langle \hat{I_1}, \hat{\mathcal{P}_1}, \hat{\sigma_1} \rangle$, $\hat{v_2} = \langle \hat{I_2}, \hat{\mathcal{P}_2}, \hat{\sigma_2} \rangle$ be any corresponding HCFAsp-solution of v_1 and v_2, respectively, which exist by Definition 3.11. Similar to above, we show that there is a corresponding HCFAsp-solution $\hat{u}$ of u. To this end, let $\hat{u} := \langle \hat{I_1} \cup \hat{I_2}, \hat{\mathcal{P}_1} \cup \hat{\mathcal{P}_2}, \sigma' \rangle$, where σ' is an $(\hat{I_1} \cup \hat{I_2})$-distinct level mapping that is compatible with both $\hat{\sigma_1}$ and $\hat{\sigma_2}$. Such a level mapping σ' can be easily obtained by viewing $\hat{\sigma_1}$ as a sequence of atoms, where atoms are ordered in ascending order of their level. Then, we add to sequence $\hat{\sigma_1}$ those atoms of $\hat{\sigma_2}$ that are not in $\chi(t)$, but in the exact same order as in sequence $\hat{\sigma_2}$ and in relation to atoms $\chi(t)$ of $\hat{\sigma_1}$. Finally, the result is a sequence σ', which is clearly compatible with both $\hat{\sigma_1}$ and $\hat{\sigma_2}$. □

Lemma 3.14 (Completeness)**.** *Let* Π *be an HCF program,* (T, χ, δ_Π) *be a nice LTD of* Π*, and* t *be node of* T*, where* type(t) $\neq$ *leaf and* post-children(t) $= \langle t_1, \ldots, t_\ell \rangle$*. Given an* HCFAsp*-row* $u = \langle I, \mathcal{P}, \sigma \rangle$ *at node* t*. Then, there exists* $s = \langle \{v_1\}, \ldots, \{v_\ell\} \rangle$ *where* v_i *is an* HCFAsp*-row at* t_i *for* $1 \leq i \leq \ell$ *such that* $u \in \text{HCFAsp}_t(\chi(t), \delta_\Pi(t), s)$*.*

Proof (Sketch). Since u is an HCFAsp-row at t, there is by Definition 3.11 a corresponding HCFAsp-solution $\hat{u} = \langle \hat{I}, \hat{\mathcal{P}}, \hat{\sigma} \rangle$ of u. We proceed again by case distinction.

Assume that type(t) = *intr*. Then we define $\hat{v_1} := \langle \hat{I} \setminus \{a\}, \dot{\mathcal{P}}, \dot{\sigma} \rangle$, where $\dot{\sigma}$ is the $(\hat{I} \setminus \{a\})$-distinct level mapping compatible with $\hat{\sigma}$ (similar to $\hat{\sigma}$, but without mapping a) and $\dot{\mathcal{P}} := \text{proven}(\hat{I} \setminus \{a\}, \dot{\sigma}, \Pi_{\leq t_1})$. From this, we define $v_1 := \langle I \setminus \{a\}, \dot{\mathcal{P}} \cap \chi(t_1), \sigma' \rangle$, where σ' is the $(I \setminus \{a\})$-distinct level mapping compatible with $\dot{\sigma}$. Observe that all the conditions of Definition 3.11 are met, i.e., v_1 is an HCFAsp-row at t_1 and by construction, $\hat{v_1}$ is a corresponding HCFAsp-solution of v_1. By construction of $\hat{v_1}$ and by the definition of "proven", we conclude that u can be constructed with HCFAsp using v_1, i.e., $u \in \mathsf{HCFAsp}_t(\chi(t), \delta_\Pi(t), s)$.

Assume that type(t) = *rem*. We define $\hat{v_1} := \hat{u}$ and from this, we let $v_1 := \langle \hat{I} \cap \chi(t_1), \hat{\mathcal{P}} \cap \chi(t_1), \sigma' \rangle$, where σ' is the $(\hat{I} \cap \chi(t_1))$-distinct level mapping compatible with $\hat{\sigma}$. Observe that all the conditions of Definition 3.11 are met, i.e., v_1 is an HCFAsp-row at t_1 and by construction, $\hat{v_1}$ is a corresponding HCFAsp-solution of v_1. Similar to above, we conclude that u can be constructed with HCFAsp using v_1, i.e., $u \in \mathsf{HCFAsp}_t(\chi(t), \delta_\Pi(t), s)$.

Assume that type(t) = *join*. Then, we define $\hat{v_1} := \langle \hat{I} \cap \text{at}(\Pi_{\leq t_1}), \hat{\mathcal{P}} \cap \text{at}(\Pi_{\leq t_1}), \sigma_1 \rangle$, where σ_1 is the $(\hat{I} \cap \text{at}(\Pi_{\leq t_1}))$-distinct level mapping compatible with $\hat{\sigma}$. Analogously, we define $\hat{v_2} := \langle \hat{I} \cap \text{at}(\Pi_{\leq t_2}), \hat{\mathcal{P}} \cap \text{at}(\Pi_{\leq t_2}), \sigma_2 \rangle$, where σ_2 is the $(\hat{I} \cap \text{at}(\Pi_{\leq t_2}))$-distinct level mapping compatible with $\hat{\sigma}$. With the help of these HCFAsp-solutions, we are able to define corresponding HCFAsp-row at t_1 by $v_1 := \langle \hat{I} \cap \chi(t_1), \hat{\mathcal{P}} \cap \chi(t_1), \sigma_1' \rangle$, where σ_1' is the $(\hat{I} \cap \chi(t_1))$-distinct level mapping compatible with σ_1, and HCFAsp-row at t_2 by $v_2 := \langle \hat{I} \cap \chi(t_2), \hat{\mathcal{P}} \cap \chi(t_2), \sigma_2' \rangle$, where σ_2' is the $(\hat{I} \cap \chi(t_2))$-distinct level mapping compatible with σ_2. Observe that all the conditions of Definition 3.11 are met, i.e., v_1, v_2 is an HCFAsp-row at t_1, t_2, respectively, and by construction, $\hat{v_1}$, $\hat{v_2}$ is a corresponding HCFAsp-solution of v_1, v_2, respectively. Again, we finally conclude that u can be constructed with HCFAsp using v_1 and v_2, i.e., $u \in \mathsf{HCFAsp}_t(\chi(t), \delta_\Pi(t), s)$. □

Theorem 3.15 (Correctness)**.** *The algorithm* $\mathtt{DP}_{\mathsf{HCFAsp}}$ *is correct. In other words, given a program* Π *and a nice LTD* $\mathcal{T} = (T, \chi, \delta_\Pi)$ *of* Π*. Then,* $\mathtt{DP}_{\mathsf{HCFAsp}}(\Pi, \mathcal{T})$ *computes for each node* t *of* T *its* HCFAsp*-table. Consequently,* Π *has an answer set if and only if* $\langle \emptyset, \emptyset, \emptyset \rangle \in \tau_{\text{root}(T)}$*, where* $\tau_{\text{root}(T)}$ *is the obtained* HCFAsp*-table for the root of* T*.*

Proof (Sketch). The proof establishes both soundness and completeness.

For soundness, we inductively apply Lemma 3.13 for every node t of T in post-order, starting with the empty leafs of T. By Lemma 3.13, HCFAsp is correct for the empty leave nodes of T, i.e., HCFAsp only computes HCFAsp-rows at t for every leaf node t of T. Then, we apply Lemma 3.13 subsequently for every node t of T, where soundness has already been established for every child node in $\mathsf{children}(t)$. Consequently, every row that is computed by table algorithm HCFAsp is correct, and therefore table algorithm HCFAsp called with a certain node t only computes HCFAsp-rows at t.

For completeness, we need to make sure that indeed for every HCFAsp-row at a node t there are suitable predecessor HCFAsp-rows at child nodes of t. We show this by induction starting from the empty root node root(T) towards the leave nodes of T. More concretely, by Lemma 3.14, for every row in the HCFAsp-table at root(T), there is a HCFAsp-row

for every child node of root(T). We continue applying Lemma 3.14, until we reach the leaves of T.

Consequently, the combination of soundness and completeness establishes that HCFAsp computes for every node t of T the HCFAsp-table at t.

Finally, we establish the claim as follows.

"$\Longrightarrow$": Assume that Π has an answer set. By definition 3.11, there is an HCFAsp-row at root(T). As a result, by soundness and completeness with the help of Lemmas 3.13 and 3.14, respectively, we have that $\langle \emptyset, \emptyset, \emptyset \rangle$ is an HCFAsp-row at root(T) obtained by algorithm $\mathsf{DP}_{\mathsf{HCFAsp}}$ for root node root(T).

"$\Longleftarrow$": Assume that $\langle \emptyset, \emptyset, \emptyset \rangle$ is an HCFAsp-row at root(T) obtained by algorithm $\mathsf{DP}_{\mathsf{HCFAsp}}$ for root node root(T). Then, by soundness and completeness as above, there is a corresponding HCFAsp-solution $\langle M, \mathcal{P}, \sigma \rangle$ such that indeed M is an answer set of Π. □

The runtime behavior of the algorithm above is as follows.

Theorem 3.16. *Given an HCF program Π and an LTD $\mathcal{T} = (T, \chi, \delta_\Pi)$ of Π of width k with g nodes. Then, algorithm $\mathsf{DP}_{\mathsf{HCFAsp}}$ runs in time $\mathcal{O}(3^k \cdot k! \cdot g \cdot \|\Pi\|) = \mathcal{O}(2^{k \cdot \log(k)} \cdot g \cdot \|\Pi\|)$.*

Proof (Sketch). Let $d = k + 1$ be the maximum bag size of the labeled tree decomposition $\mathcal{T}$. The table τ_t has at most $3^d \cdot d!$ rows, since for a row $\langle I, \mathcal{P}, \sigma \rangle$ we have at most $d!$ many I-distinct level mappings σ, and by construction of algorithm HCFAsp, an atom can be either in I, both in I and $\mathcal{P}$, or neither in I nor in $\mathcal{P}$. In total, with the help of efficient data structures, e.g., for nodes t with type(t) = *join*, one can establish a runtime bound of $\mathcal{O}(3^d \cdot d! \cdot \|\Pi\|)$. Then, we apply this to every node t of the tree decomposition, which results in running time $\mathcal{O}(3^d \cdot d! \cdot g \cdot \|\Pi\|) \subseteq \mathcal{O}(3^k \cdot k! \cdot g \cdot \|\Pi\|)$. □

In fact, also this runtime can probably not be significantly improved, assuming the ETH. Indeed, one can obtain runtime lower bounds under the ETH and show that the level mappings are essential and cannot be avoided. However, the proof is a bit more involved and therefore shifted to Section 5.2, which is dedicated to lower bounds (cf. Theorem 5.35).

3.2.3 Solving (Disjunctive) Asp for Treewidth

Next, we are going to solve the full problem Asp, which refers to arbitrary logic programs. Of course, we thereby also obtain an algorithm for solving Disjunctive Asp, which is the problem Asp when restricted to disjunctive programs.

Recall the semantics of answer set programming. For an interpretation M to be an answer set of a program Π, (i) the interpretation M has to be a model of the program and (ii) there is no smaller model $C \subsetneq M$ with C being a model of the reduct Π^M. In this section, we present a table algorithm Asp, which is heavily based on the characterization

Listing 3.5: Table algorithm $\mathsf{Asp}_t(\chi_t, \Pi'_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$.

In: Node t, bag χ_t, bag program Π'_t, and sequence $\langle \tau_1, \ldots, \tau_\ell \rangle$ of child Asp-tables of t.
Out: Asp-table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t \leftarrow \{\langle \emptyset,\ \emptyset \rangle\}$ `/* Abbreviations are given below. */`
2 **else if** type$(t) =$ *intr and* $a \in \chi_t$ *is the introduced atom* **then**
3 $\quad \tau_t \leftarrow \{\langle M_a^+,\ \mathrm{Mod}(\{M\} \cup [\mathcal{C} \sqcup \{a\}] \cup \mathcal{C}, \Pi_t'^{M_a^+}) \rangle \mid \langle M, \mathcal{C} \rangle \in \tau_1, M_a^+ \models \Pi'_t\} \cup$
4 $\qquad \{\langle M,\ \mathrm{Mod}(\mathcal{C}, \Pi_t'^{M}) \rangle \mid \langle M, \mathcal{C} \rangle \in \tau_1, M \models \Pi'_t\}$
5 **else if** type$(t) =$ *rem and* $a \notin \chi_t$ *is the removed atom* **then**
6 $\quad \tau_t \leftarrow \{\langle M_a^-,\ \{C_a^- \mid C \in \mathcal{C}\} \rangle \mid \langle M, \mathcal{C} \rangle \in \tau_1\}$
7 **else if** type$(t) =$ *join* **then**
8 $\quad \tau_t \leftarrow \{\langle M,\ (\mathcal{C}_1 \cap \mathcal{C}_2) \cup (\mathcal{C}_1 \cap \{M\}) \cup (\{M\} \cap \mathcal{C}_2) \rangle \mid \langle M, \mathcal{C}_1 \rangle \in \tau_1, \langle M, \mathcal{C}_2 \rangle \in \tau_2\}$
9 **return** τ_t

For a set S, a set $\mathcal{S}$ of sets, and an element e, we abbreviate $\mathcal{S} \sqcup \{e\} := \{S \cup \{e\} \mid S \in \mathcal{S}\}$, $S_e^+ := S \cup \{e\}$, and $S_e^- := S \setminus \{e\}$.

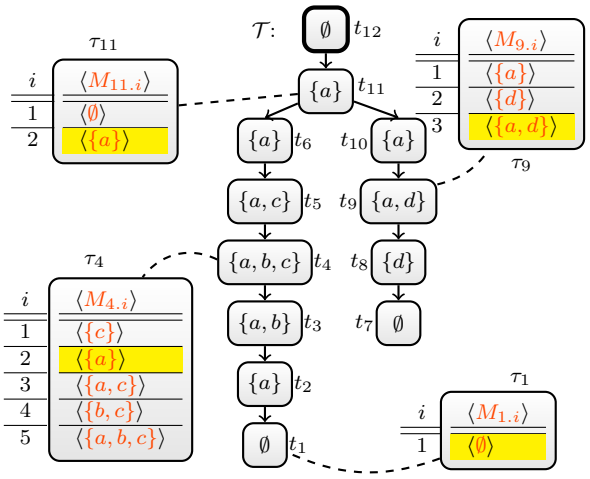

Figure 3.6: A nice TD $\mathcal{T}$ of the primal graph $\mathcal{G}_{\dot{\Pi}}$ of program $\dot{\Pi}$ from Example 2.3 as well as selected tables obtained by $\mathsf{DP}_{\mathsf{ModAsp}}$ on $\dot{\Pi}$ and the canonical LTD of $\mathcal{T}$.

of semantics above. Our algorithm is given in Listing 3.5 and it will be discussed in two parts: (i) finding models of Π and (ii) finding models which are subset minimal with respect to Π^M. For the sake of clarity and towards a comprehensive example, we first present only the first tuple positions (red text) of algorithm Asp to solve Task (i). The resulting (reduced) table algorithm aims at computing models of the program Π. We call this table algorithm ModAsp, which is quite similar to the table algorithm #Sat of Listing 3.2 as already discussed above, but without the counting.

Example 3.17. *Consider program* $\dot{\Pi} := \{\overbrace{\{a; b\} \leftarrow c;}^{r_1}\ \overbrace{c \leftarrow 1 \leqslant \{b = 1, \neg a = 1\};}^{r_2}\ \overbrace{d \vee a \leftarrow}^{r_3}\}$ *from Example 2.3 and in Figure 3.6 tree decomposition* $\mathcal{T} = (T, \chi)$ *of* $\mathcal{G}_{\dot{\Pi}}$ *and the tables* τ_1, $\ldots$, τ_{12}, *which illustrate computation results obtained during post-order traversal of the canonical LTD of* $\mathcal{T}$ *by* $\mathsf{DP}_{\mathsf{ModAsp}}$.

Table $\tau_1 = \{\langle\emptyset\rangle\}$ *as* $\mathrm{type}(t_1) = \mathit{leaf}$. *Since* $\mathrm{type}(t_2) = \mathit{intr}$, *we construct table* τ_2 *from* τ_1 *by taking* $M_{1.i}$ *and* $M_{1.i} \cup \{a\}$ *for each* $M_{1.i} \in \tau_1$ *(corresponding to a guess on* a*). Then,* t_3 *introduces* b *and* t_4 *introduces* c. $\dot{\Pi}_{t_1} = \dot{\Pi}_{t_2} = \dot{\Pi}_{t_3} = \emptyset$, *but since* $\chi(t_4) \subseteq \mathrm{at}(r_1) \cup \mathrm{at}(r_2)$ *we have* $\dot{\Pi}_{t_4} = \{r_1, r_2\}$ *for* t_4. *In consequence, for each* $M_{4.i}$ *of table* τ_4, *we have* $M_{4.i} \models \{r_1, r_2\}$ *since* ModAsp *enforces satisfiability of* $\dot{\Pi}_t$ *in node* t. *We derive tables* τ_7 *to* τ_9 *similarly. Since* $\mathrm{type}(t_5) = \mathit{rem}$, *we remove atom* b *from all elements in* τ_4 *to construct* τ_5. *Note that we have already seen all rules where* b *occurs and hence* b *can no longer affect witnesses during the remaining traversal. We similarly construct* $\tau_{t_6} = \tau_{10} = \{\langle\emptyset\rangle, \langle a\rangle\}$. *Since* $\mathrm{type}(t_{11}) = \mathit{join}$, *we construct table* τ_{11} *by taking the intersection* $\tau_6 \cap \tau_{10}$. *Intuitively, this combines witnesses agreeing on* a. *Node* t_{12} *is again of type rem. By definition (primal graph and tree decompositions) for every* $r \in \dot{\Pi}$, *atoms* $at(r)$ *occur together in at least one common bag. Hence,* $\dot{\Pi} = \dot{\Pi}_{t_{12}}$ *and since* $\tau_{12} = \{\langle\emptyset\rangle\}$, *and we can construct a model of* $\dot{\Pi}$ *from the tables. For example, we obtain the model* $\{a, d\} = M_{11.2} \cup M_{4.2} \cup M_{9.3}$, *as highlighted in the figure (yellow). Again, rows colored in grey are not part of any answer set of* $\dot{\Pi}$

The complete Asp algorithm is given in Listing 3.5. Tuples in τ_t are of the form $\langle M, \mathcal{C}\rangle$. We refer by *witness* to $M \subseteq \chi(t)$, which represents a model of Π_t witnessing the existence of $M' \supseteq M$ with $M' \models \Pi_{\leq t}$. The family $\mathcal{C} \subseteq 2^M$ contains sets of models $C \subseteq M$ of the GL reduct $(\Pi_t)^M$. A set C witnesses the existence of a set C' where $C \subseteq C' \subsetneq M'$ and $C' \models (\Pi_{\leq t})^{M'}$. Therefore, we refer to each element $C \in \mathcal{C}$ by *counter-witness* and $\mathcal{C}$ is called *counter-witness set.*

Concretely, we define Asp-rows and Asp-tables as follows.

Definition 3.18. *Given a program* Π, *an LTD* (T, χ, δ_Π) *of* Π, *and a node* t *of* T. *Then, an* Asp-row (for t) *is a sequence of the form* $\langle M, \mathcal{C}\rangle$ *with* $M \subseteq \chi(t)$ *and* $\mathcal{C} \subseteq 2^M$ *such that the following holds:*

1. $M \models \delta_\Pi(t)$,
2. *for every* $C \in \mathcal{C}$ *we have that* $C \models \delta_\Pi(t)^M$, *and*
3. *there exists* $M' \subseteq \mathrm{at}(\Pi_{\leq t}) \setminus \chi(t)$ *such that*
   - $(M \cup M') \models \Pi_{\leq t}$ *and*
   - $\mathcal{C}$ *is the largest set with* $\mathcal{C} \subseteq 2^M$ *such that for each* $C \in \mathcal{C}$ *there exists* $C' \subseteq \mathrm{at}(\Pi_{\leq t}) \setminus \chi(t)$ *with*
     - $C \cup C' \subsetneq M \cup M'$ *as well as*
     - $(C \cup C') \models \Pi_{\leq t}^{(M \cup M')}$

Further, we let the Asp-table (for t) *be the largest set of* Asp-*rows for* t.

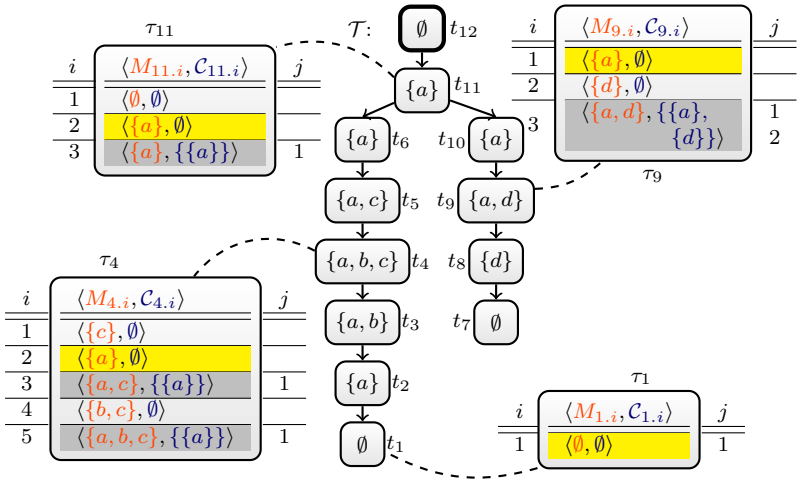

Figure 3.7: Selected tables obtained by $\mathtt{DP}_{\mathsf{Asp}}$ on $\dot{\Pi}$ of Example 2.3 and the canonical LTD of nice TD $\mathcal{T}$ of $\mathcal{G}_{\dot{\Pi}}$.

In consequence, the definition above ensures that there is an answer set of Π if table $\tau_{\mathrm{root}(T)}$ for the root of the decomposition contains $\langle\emptyset,\emptyset\rangle$, i.e., if $\langle\emptyset,\emptyset\rangle$ is an Asp-row for $\mathrm{root}(T)$.

In Example 3.17 we already explained the first tuple position and thus the witness part, we only briefly describe the parts for counter-witnesses. In the introduce case, we want to store only counter-witness sets for not being minimal with respect to the GL reduct of the bag-program. Therefore, in Line 3 we construct for M_a^+ counter-witness sets from either some witness M ($M \subsetneq M_a^+$), or of any $C \in \mathcal{C}$, or of any $C \in \mathcal{C}$ extended by a (every $C \in \mathcal{C}$ was already a counter-witness before). Line 4 ensures that only counter-witness sets that are models of the GL reduct Π_t^M are stored (via $\mathrm{Mod}(\cdot,\cdot)$). Line 6 restricts counter-witness sets to its bag content, and Line 8 enforces that child tuples agree on counter-witness sets.

Example 3.19. *Consider Example 3.17, its tree decomposition $\mathcal{T} = (\cdot, \chi)$, Figure 3.7, and the tables $\tau_1, \ldots, \tau_{12}$ obtained by $\mathtt{DP}_{\mathsf{Asp}}$. Since we have $\mathrm{at}(r_1) \cup \mathrm{at}(r_2) \subseteq \chi(t_4)$, we require $C_{4.i.j} \models \{r_1, r_2\}^{M_{4.i}}$ for each counter-witness set $C_{4.i.j} \in \mathcal{C}_{4.i}$ in tuples of τ_4. For $M_{4.5} = \{a, b, c\}$ observe that the only counter-witness set of $\{r_1, r_2\}^{M_{4.5}} = \{a \leftarrow c, b \leftarrow c, c \leftarrow 1 \leq \{b = 1\}\}$ is $C_{4.5.1} = \{a\}$. Note that witness $M_{11.2}$ of table τ_{11} is the result of joining $M_{4.2}$ with $M_{9.1}$ and witness $M_{11.3}$ (counter-witness set $C_{11.3.1}$) is the result of joining $M_{4.3}$ with $M_{9.3}$ ($C_{4.3.1}$ with $C_{9.3.1}$), and $M_{4.5}$ with $M_{9.3}$ ($C_{4.5.1}$ with $C_{9.3.2}$). $C_{11.3.1}$ witnesses that neither $M_{4.3} \cup M_{9.3}$ nor $M_{4.5} \cup M_{9.3}$ forms an answer set of $\dot{\Pi}$. Since τ_{12} contains $\langle\emptyset,\emptyset\rangle$ there is no counter-witness set for $M_{11.2}$, and we can construct an answer set of $\dot{\Pi}$ from the tables, e.g., $\{a\}$ can be constructed from $M_{4.2} \cup M_{9.1}$.*

Theorem 3.20. *The algorithm $\mathtt{DP}_{\mathsf{Asp}}$ is correct. In other words, given a program Π and a nice LTD $\mathcal{T} = (T, \chi, \delta_\Pi)$ of Π. Then, $\mathtt{DP}_{\mathsf{Asp}}(\Pi, \mathcal{T})$ computes for each node t of T its Asp-table. Consequently, Π has an answer set if and only if $\langle\emptyset,\emptyset\rangle$ is an Asp-row for* $\mathrm{root}(T)$.

Proof (Idea). The result is a slight generalization of an already established algorithm for Disjunctive Asp [Jakl et al., 2009]. The proof establishes both soundness and completeness, where we use Definition 3.18 similarly as in Lemma 3.13 and Lemma 3.14, respectively For soundness, we show the invariant given by Definition 3.18 for each table τ_t that is computed by table algorithm Asp, thereby assuming a set of Asp-rows for each child node of t. More precisely for every node t of T, we have that Asp only computes Asp-rows, given a set of Asp-rows for each child node of t. Further, completeness is shown in a top-down manner, where we assume that Asp computes a Asp-table for the parent node of t and show then that also the set of Asp-rows obtained by table algorithm Asp for node t is actually a Asp-table. □

Note that similar ideas were applied in an algorithm based on dynamic programming [Jakl et al., 2009]. However, this algorithm is restricted to disjunctive programs [Jakl et al., 2009]. Even further, it was shown that for a particular graph representation, treewidth alone is insufficient in order to design an FPT algorithm capable of evaluating weight rules [Pichler et al., 2014]. Still, the primal graph representation $\mathcal{G}_\Pi$ of any program Π, as discussed, analyzed and presented in this work, allows for an FPT algorithm, also when considering weight rules. More precisely, we obtain the following result.

Theorem 3.21. *Given a program* Π *and an LTD* $\mathcal{T} = (T, \chi, \delta_\Pi)$ *of* Π *of width* k *with* g *nodes. Then, algorithm* $\mathtt{DP}_{\mathsf{Asp}}$ *runs in time* $\mathcal{O}(2^{2^{k+2}} \cdot g \cdot \|\Pi\|) = 2^{2^{\mathcal{O}(k)}} \cdot g \cdot \|\Pi\|$.

Proof (Sketch). Let $d = k + 1$ be the maximum bag size of the labeled tree decomposition $\mathcal{T}$. The table τ_t has at most $2^d \cdot 2^{2^d} = 2^{2^d+d}$ many rows, since for a row $\langle M, \mathcal{C} \rangle$ for each of the 2^d many witnesses we can have at most 2^{2^d} many counter-witness sets. In total, with the help of efficient data structures, e.g., for nodes t with $\text{type}(t) = \textit{join}$, one can establish a runtime bound of $\mathcal{O}(2^{2^{d+1}} \cdot \|\Pi\|)$ since $\mathcal{O}(2^{2^d+d} \cdot \|\Pi\|) \subseteq \mathcal{O}(2^{2^{d+1}} \cdot \|\Pi\|)$. Then, we apply this to every node t of the tree decomposition, which results in running time $\mathcal{O}(2^{2^{d+1}} \cdot g \cdot \|\Pi\|) \subseteq \mathcal{O}(2^{2^{k+2}} \cdot g \cdot \|\Pi\|)$. □

However, also the runtime above cannot be significantly improved under the ETH (cf. Hypothesis 2.1). To this end, we refer once more to Chapter 5, which is dedicated to lower bounds, where, the lower bound for Asp is given in Theorem 5.18.

3.3 Outlook on Dynamic Programming For Other Formalisms

The idea of dynamic programming can be extended to further problems and formalisms. While it is beyond the focus of this thesis, table algorithms similar to those above can be designed for many more formalisms and problems. Thereby, we can reuse similar ideas and techniques as discussed in this chapter. In particular, the concept of witnesses and counter-witnesses as discussed above reappears in further works on

- problems for default logic [Fichte et al., 2020d], where counter-witnesses are applied several times, also to solve entailment (sub-)problems,
- problems relevant to abstract argumentation [Dvořák et al., 2012; Fichte et al., 2019a], where subset-minimization is also achieved by means of counter-witnesses,
- description logics and non-ground logic programs over bounded arity and fixed domain size [Fichte et al., 2021a],
- epistemic extensions of logic programs [Hecher et al., 2020a],
- and many more, e.g., [Bodlaender, 1988; Chen, 2004; Bodlaender and Koster, 2008].

Further, projected model counting for Boolean formulas (#∃Sat) served as a kind of a canonical problem to show that when utilizing treewidth, projected solution counting is oftentimes harder than plain counting. Indeed, this is the case for the problem #∃Sat [Fichte et al., 2018b], but also for many other formalisms, e.g., abstract argumentation [Fichte et al., 2019a], answer set programming [Fichte and Hecher, 2018, 2019], and quantified Boolean formulas [Capelli and Mengel, 2019; Fichte and Hecher, 2020]. Recently, these ideas lead to a meta algorithm [Fichte and Hecher, 2020] for projected solution counting for problems when parameterized by treewidth in general.

CHAPTER 4

Decomposition-Guided Reductions for Treewidth

Structure is not just a means to a solution. It is also a principle and a passion.

— Marcel Breuer

As already discussed in Chapter 3, there are many approaches to utilize treewidth. One particular approach, namely dynamic programming on tree decompositions turned into a rather prominent and flexible technique to utilize treewidth. Indeed, while this approach was developed quite early, nowadays it is considered to be a vital tool for showing tractability results for treewidth, see, e.g., [Bertelè and Brioschi, 1972, 1973; Bodlaender and Kloks, 1996; Flum and Grohe, 2006; Niedermeier, 2006]. This is witnessed by several specialized implementations that take advantage of treewidth [Charwat and Woltran, 2017; Fichte et al., 2018b, 2019b; Kiljan and Pilipczuk, 2018] as well as general systems and frameworks [Bliem et al., 2016a; Bannach and Berndt, 2019; Langer et al., 2012] based on tree decompositions.

Inspired by this direct usage of tree decompositions in order to solve problems, in this chapter we introduce reductions from a *source problem* to a *target problem* that do not only take an instance of the source problem, but also take into account a *tree decomposition* of a graph representation of the instance. At first, such a reduction might seem straightforward from a theoretical perspective since treewidth can be easily approximated [Bodlaender et al., 2016; Bodlaender, 1996; Feige et al., 2008]. However, there are still several advantages of these reductions. On the one hand, the way we use and define these decomposition-guided reductions, they give rise to a decomposition of the reduced instance of the target problem. This allows to easily visualize and derive relations between the given tree decomposition and the resulting decomposition, assuming that the bags of the resulting decomposition functionally depend on the given decomposition. In turn, such reductions might almost serve as a proof for certain treewidth guarantees

when reducing problems, or they might even contribute to irreducibility results for treewidth if used in the context of lower bounds (e.g., Propositions 2.11 or 3.3). On the other hand, especially in the light of efficient techniques to approximate treewidth or to obtain decompositions via decent heuristics, e.g., [Abseher et al., 2017; Dell et al., 2017], results using such decomposition-guided reductions are not limited to treewidth or to decompositions of smallest widths. Indeed, decomposition-guided reductions that obey certain treewidth-guarantees might open a broad range of practical applications by implementing certain problems via treewidth-aware translations to problem formalisms, for which efficient implementations[1] based on dynamic programming already exist.

After introducing the concept of decomposition-guided reductions in Section 4.1, in order to demonstrate the applicability of decomposition-guided reductions, we focus in this chapter on reductions for diverse fragments of answer set programming (ASP). So, once again deciding the consistency of an answer set program (Asp) serves as a prototypical problem, whose fragments seem have several interesting facets and different obstacles. More concretely, in Section 4.2 we provide a decomposition-guided reduction from Tight Asp to Boolean satisfiability (Sat) that linearly preserves the treewidth (cf. [Fandinno and Hecher, 2021]), which together with an efficient implementation to solve Sat by utilizing treewidth [Samer and Szeider, 2010], establishes the same runtime upper bound for Tight Asp as for Sat. Then, Section 4.3 is also based on recent work [Hecher, 2020] and concerns about a decomposition-guided reduction from Asp for normal and head-cycle free programs to Sat, which increases the treewidth slightly superlinearly. This result complements and completes the runtime insights gained by Theorem 3.16. However, unfortunately, we will see in the next chapter that it is not expected that this decomposition-guided reduction can be significantly improved. Still, there is a chance to improve this reduction, which is via an additional parameter dedicated to chaining or confining the cyclicity of Asp.

More concretely, in Section 4.4, we focus on a decomposition-guided reduction from Normal Asp to the novel class of *almost tight* programs, which stem from recent work [Fandinno and Hecher, 2021]. This class of almost tight programs is motivated by a novel measure called *tightness width*, which corresponds to the level of tightness and is bounded both by the treewidth as well as the additional parameter for chaining the "cyclicity" of programs. While there are measures related to this idea [Lin and Zhao, 2004a; Gebser et al., 2007; Fassetti and Palopoli, 2010], also from parameterized complexity [Gottlob et al., 2002b; Lonc and Truszczyński, 2003; Fichte and Szeider, 2015], to the best of our knowledge this is the first approach that analyzes such a measure in *combination* with treewidth. This reduction shows that even programs with large cycles could be solved relatively fast, as long as the tightness width is small. Interestingly, our decomposition-guided reduction can be used to decrease the tightness width, but it also allows us to obtain even a completely tight program, i.e., the reduction can be used in the context of reducing Normal Asp to Tight Asp. This shows the usage of decomposition-guided reductions in order to prove upper bound results, which therefore

[1] Such efficient implementation techniques will be discussed later in Chapter 7.

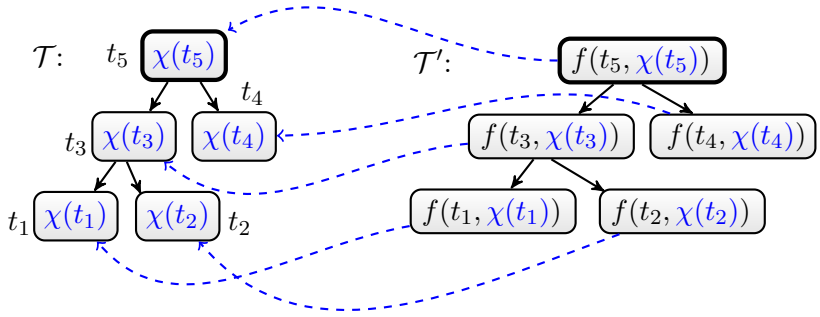

Figure 4.1: Illustration of a decomposition-guided reduction from problem P to problem P′, where we assume a given instance $\mathcal{I}$ of a problem P and a tree decomposition $\mathcal{T} = (T, \chi)$ of $\mathcal{G}_\mathcal{I}$. Then, since the decomposition-guided reduction is constructed for each node t of T, it immediately gives rise to a tree decomposition $\mathcal{T}' = (T, \chi')$ of $\mathcal{G}_{\mathcal{I}'}$ of the resulting instance $\mathcal{I}'$ of problem P. Further, each bag $\chi'(t)$ of a node t of T functionally depends on t and $\chi(t)$.

complements the usage of dynamic programming techniques of Chapters 3. We finally conclude this chapter with a brief summary and discussion in Section 4.5.

While this chapter mainly introduces decomposition-guided reductions and deals with exemplary reductions for Asp, later in Chapter 5 we use these decomposition-guided reductions in order to establish conditional lower bounds for problems parameterized by treewidth. Thereby, these reductions aid in solving a problem concerning the lower bound for evaluating quantified Boolean formulas (QSat) when parameterized by treewidth, which has been open [Chen, 2004] since 2004. Further, decomposition-guided reductions also help in proving lower bounds for fragments of ASP and treewidth, and in Chapter 6, they act as an important driver for establishing a novel tool that simplifies lower bound proofs for problems parameterized by treewidth.

4.1 Basic Definitions

Before we discuss the different use cases on where to utilize decomposition-guided reductions, we briefly introduce the concept of the former more precisely. Inspired by an intuitive account of decomposition-guided reductions as already encountered in the proof of Proposition 3.3, in this section we discuss a more formal notion of this concept. A *decomposition-guided (DG) reduction* R is a function that takes both an instance $\mathcal{I}$ of a problem P and a tree decomposition $\mathcal{T} = (T, \chi)$ of a graph representation, e.g., the primal graph $\mathcal{G}_\mathcal{I}$, of $\mathcal{I}$, and returns an instance $\mathcal{I}'$ of a problem P′. Then, the reduction directly yields also a tree decomposition $\mathcal{T}' = (T, \chi')$ of the same graph representation ($\mathcal{G}_{\mathcal{I}'}$) of $\mathcal{I}'$. Thereby, the reduction is focused to be constructed from a decomposition node's point of view, i.e., for each node t of T, the constructed bag $\chi'(t)$ functionally

depends on the original bag $\chi(t)$. Figure 4.1 provides a simplified picture of this and uses a function f taking a node t and its original bag $\chi(t)$, in order to construct each bag $\chi'(t) = f(t, \chi(t))$. Intuitively, decomposition-guided reductions are *guided* by a TD $\mathcal{T} = (T, \chi)$ and adhere to core ideas of dynamic programming along TD $\mathcal{T}$ to ensure certain properties of the treewidth of the resulting instance $\mathcal{I}'$ of $\mathtt{P}'$. More concretely, since $\mathsf{width}(\mathcal{T})$ is bounded by $\mathcal{O}(\max_{t \text{ of } T}(|\chi(t)|))$, also the treewidth of the resulting instance is at most $\mathcal{O}(\max_{t \text{ of } T}(|f(t, \chi(t))|))$.

However, this provides only a simplified and abstract picture of these reductions. Decomposition-guided reductions have to guarantee that indeed $\mathcal{T}'$ is a tree decomposition of $\mathcal{G}_{\mathcal{I}'}$. In order to make decomposition-guided reductions more accessible as well as more applicable in regard of this guarantee, such a reduction can be constructed either bottom-up or top-down. A *bottom-up decomposition-guided reduction* $R_\uparrow$ is a decomposition-guided reduction, where for each node t of T, the construction of $\chi'(t)$ functionally depends on the node t, the bag $\chi(t)$, but also on the the constructed bags $\chi'(t_1), \ldots, \chi'(t_o)$ of its child nodes $\{t_1, \ldots, t_o\} = \mathsf{children}(t)$. This gives rise to a function $f_\uparrow$ that takes a tree decomposition node t, its bag $\chi(t)$ and a set $\chi'(\mathsf{children}(t) := \{\chi'(t_i) \mid t_i \in \mathsf{children}(t)\}$ of constructed bags for the child nodes of t. Then, $R_\uparrow$ constructs $\chi'(t) = f_\uparrow(t, \chi(t), \chi'(\mathsf{children}(t)))$ for each node t in post-order or in a bottom-up manner. Observe that this reduction works in post-order as dynamic programming does (cf. Chapter 3). However, in contrast to dynamic programming, bottom-up decomposition-guided reductions are not supposed to compute tables, but are designed to translate from problem $\mathtt{P}$ to problem $\mathtt{P}'$. Figure 4.2 illustrates this concept, which is then formalized in the following definition.

Definition 4.1 (Bottom-Up DG Reduction)**.** *A bottom-up DG reduction $R_\uparrow$ from a problem $\mathtt{P}$ to a problem $\mathtt{P}'$ takes an instance $\mathcal{I}$ of $\mathtt{P}$ and a TD $\mathcal{T} = (T, \chi)$ of $\mathcal{G}_{\mathcal{I}}$. Then, $R_\uparrow$ constructs an instance $\mathcal{I}'$ of $\mathtt{P}'$ and guarantees the existence of a TD $\mathcal{T}' = (T, \chi')$ of $\mathcal{G}_{\mathcal{I}'}$. Further, there exists a function $f_\uparrow$ that takes any node t of T, bag $\chi(t)$ and the set $\chi'(\mathsf{children}(t))$ such that $\chi'(t) = f_\uparrow(t, \chi(t), \chi'(\mathsf{children}(t)))$.*

Observe that while the tree of $\mathcal{T}$ and $\mathcal{T}'$ is identical, their bag contents might differ. Next, we briefly provide a simple example that clarifies bottom-up DG reductions.

Example 4.2. *Consider a Boolean formula F and a tree decomposition $\mathcal{T} = (T, \chi)$ of $\mathcal{G}_F$. From this we define a bottom-up DG reduction $R_\uparrow$, which is then used to construct a fresh instance $F' = R_\uparrow(F, \mathcal{T})$, where each variable of $\mathsf{var}(F')$ appears in at most $1+c$ many bags of $\mathcal{T}'$, where c is the maximal number of child nodes per node of T. To this end, we require auxiliary variables of the form x_t for each node t of T and variable $x \in \mathsf{var}(\chi(t))$. Then, the construction is for each node t of T as follows. We construct $x_t \longleftrightarrow x_{t'}$ (4.1) for every $t' \in \mathsf{children}(t)$ and for each clause $c \in F_t$ with $c = l_1 \vee l_2 \vee l_2$, we add $l_1^t \vee l_2^t \vee l_3^t$ (4.2), where function $\cdot^t$ takes a literal over some variable x and replaces the occurrence of x by x_t. Observe that the resulting formula F' is satisfiable if and only if F is satisfiable and there is even a bijective correspondence between models of F and models of F'. Moreover, if $\mathcal{T}$ is a nice tree decomposition, every variable occurs in at most three bags of $\mathcal{T}'$. Consequently,*

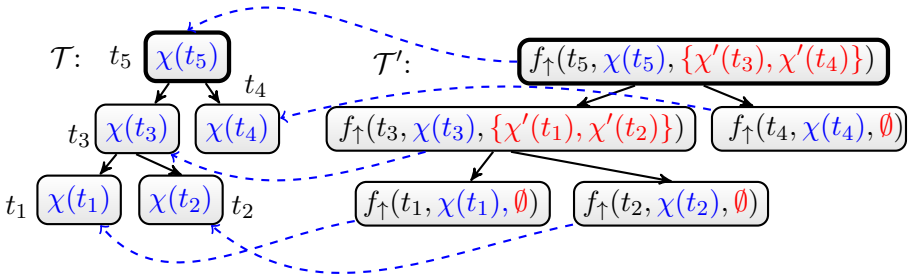

Figure 4.2: Illustration of a bottom-up decomposition-guided reduction from problem $\mathtt{P}$ to problem $\mathtt{P}'$, where we assume a given instance $\mathcal{I}$ of a problem $\mathtt{P}$ and a tree decomposition $\mathcal{T} = (T, \chi)$ of $\mathcal{G}_\mathcal{I}$. Then, the reduction is constructed for each node t of T and it immediately gives rise to a tree decomposition $\mathcal{T}' = (T, \chi')$ of $\mathcal{G}_{\mathcal{I}'}$ of the resulting instance $\mathcal{I}'$ of problem $\mathtt{P}$. Each bag $\chi'(t)$ of a node t of T functionally depends on t, $\chi(t)$, as well as $\chi'(t')$ of every child node $t' \in \mathsf{children}(t)$.

in such a case of nice decompositions, the width of $\mathcal{T}'$ is in $\mathcal{O}(3 \cdot \mathsf{width}(\mathcal{T})) = \mathcal{O}(\mathsf{width}(\mathcal{T}))$ since $\chi'(t) = \chi(t)^t \cup \chi'(\mathsf{children}(t))$ by slightly abusing $\cdot^t$ for a set of variables. Observe that therefore there exists a function $f_\uparrow$ as required by Definition 4.1 and reduction $R_\uparrow$ consisting of Formulas (4.1) and (4.2) is indeed a bottom-up DG reduction. This example shows that for Boolean satisfiability one can safely assume for a variable to occur only constantly many times in a tree decomposition of the primal graph of the given formula.

Similar to bottom-up DG reductions, we sometimes also use top-down DG reductions, which can be defined analogously as follows.

Definition 4.3 (Top-Down DG Reduction)**.** *A top-down DG reduction $R_\downarrow$ from a problem $\mathtt{P}$ to a problem $\mathtt{P}'$ takes an instance $\mathcal{I}$ of $\mathtt{P}$ and a TD $\mathcal{T} = (T, \chi)$ of $\mathcal{G}_\mathcal{I}$. Then, $R_\downarrow$ constructs an instance $\mathcal{I}'$ of $\mathtt{P}'$ and guarantees the existence of a TD $\mathcal{T}' = (T, \chi')$ of $\mathcal{G}_{\mathcal{I}'}$. Further, there exists a function $f_\downarrow$ that takes any node t of T, bag $\chi(t)$ and the set $\chi'(\mathsf{parent}(t))$ such that $\chi'(t) = f_\downarrow(t, \chi(t), \chi'(\mathsf{parent}(t)))$.*

Intuitively, top-down DG reductions might be slightly preferable over bottom-up DG reductions, since the width of the resulting decomposition $\mathcal{T}'$ of $\mathcal{G}_{\mathcal{I}'}$ only depends on the width of the parent node of t and not also on the widths of all child nodes of t.

Example 4.4. *Recall Example 4.2 from above. Observe that the DG reduction $R_\uparrow$ discussed there could be also given in form of a top-down DG reduction $R_\downarrow$, since Formulas (4.1) only require a relationship between a node and a parent node, but does not relate sibling nodes. Consequently, one could slightly modify Formulas (4.1), thereby constructing for every node t of T the formula $x_t \longleftrightarrow x_{\mathsf{parent}(t)}$ (4.3) instead of Formulas (4.1). Then, we obtain a top-down DG reduction $R_\downarrow$, which consists of Formulas (4.2) and (4.3). This allows us to define $\chi'(t) = f_\downarrow(t, \chi(t), \chi'(\mathsf{parent}(t))) = \chi(t)^t \cup \chi'(\mathsf{parent}(t))$, where*

function $\cdot^t$ *is defined as in Example 4.2. Consequently, we have that* $\mathsf{width}(\mathcal{T}')$ *is in* $\mathcal{O}(2 \cdot \mathsf{width}(\mathcal{T})) = \mathcal{O}(\mathsf{width}(\mathcal{T}))$.

In the remainder of this chapter, we use this concept of decomposition-guided reductions in order to establish upper bound results for certain fragments of Asp as well as a reduction for Asp that adheres to a novel measure in order to reduce "hardness" step-by-step when evaluating programs.

4.2 Decomposition-Guided Reduction from Tight Asp to Sat

Recall answer set programming (ASP) and its role as an active research area in knowledge representation and reasoning, cf. Chapter 2. Next, we present a treewidth-aware reduction from Tight Asp to Sat. While the step from Tight Asp to Sat might seem straightforward, in general it is not guaranteed that existing reductions, e.g., [Fages, 1994; Lin and Zhao, 2003; Janhunen, 2006], avoid a significant blowup in the treewidth. Indeed, already large rules significantly increase the treewidth in the worst case. Therefore we ensure that the treewidth only increases at most linearly by means of a decomposition-guided reduction to Sat. Naturally, this reduction also works for solving the problem Supported Models on any disjunctive program.

Let Π be any given tight program and $\mathcal{T} = (T, \chi, \delta_\Pi)$ be a labeled tree decomposition of Π. For our reduction, we use as variables besides the original atoms of Π also auxiliary variables. In order to preserve treewidth, we still need to guide the evaluation of the provability of an atom $x \in \mathrm{at}(\Pi)$ in a node t in T along the TD $\mathcal{T}$, whereby we use atoms p^x_t and $p^x_{\leq t}$ to indicate that x was proven in node t and below t, respectively. However, we do not need any level mappings, since there is no positive cycle in Π, but we still guide the idea of Clark's completion [Clark, 1977] along TD $\mathcal{T}$. Indeed, this is in line with the characterization of answer sets of tight programs and supported models of disjunctive programs, as defined in Section 2.4.

Consequently, we construct the following Boolean formula, where for each node t of T we add Formulas (4.4)–(4.8). Intuitively, Formulas (4.4) ensure that all rules are satisfied. Formulas (4.5) provide the definition for provability of an atom in a node, whereby an atom is proven whenever we encounter a rule with the atom in the head that supports this atom. Then, Formulas (4.6) guide this information from a node to its parent node. Finally, Formulas (4.7) and (4.8) take care that ultimately an atom that is set to true requires to be proven. In more details, Formulas (4.7) guarantee, for nodes t removing bag atom x, i.e., $x \in \chi(t) \setminus \chi(t')$, that x is proven if x is set to true. Similarly, this is required for atoms $x \in \chi(n)$ that are in the root node $n = \mathrm{root}(T)$ and therefore never forgotten, cf. Formulas (4.8).

Preserving answer sets: We obtain exactly one model of the resulting formula for each answer set of Π. This can be weakened by turning equivalences ($\leftrightarrow$) into implications ($\rightarrow$).

$$\bigvee_{a\in B_r^+} \neg a \vee \bigvee_{a\in B_r^- \cup H_r} a \qquad \text{for each } r \in \delta_\Pi(t) \tag{4.4}$$

$$p_t^x \longleftrightarrow \bigvee_{r\in\delta_\Pi(t), x\in H_r} \Big(\bigwedge_{a\in B_r^+} a \wedge x \wedge \bigwedge_{b\in B_r^- \cup (H_r\setminus\{x\})} \neg b \Big) \qquad \text{for each } x \in \chi(t) \tag{4.5}$$

$$p_{\leq t}^x \longleftrightarrow p_t^x \vee \Big(\bigvee_{t'\in \mathsf{children}(t), x\in\chi(t')} p_{\leq t'}^x \Big) \qquad \text{for each } x \in \chi(t) \tag{4.6}$$

$$x \longrightarrow p_{\leq t'}^x \qquad \text{for each } t' \in \mathsf{children}(t), x \in \chi(t')\setminus\chi(t) \tag{4.7}$$

$$x \longrightarrow p_{\leq n}^x \qquad \text{for each } x \in \chi(n) \text{ with } n = \mathrm{root}(T) \tag{4.8}$$

Correctness and Treewidth-Awareness

Next, we discuss the correctness of this reduction, which establishes that indeed there is a bijective correspondence between answer sets of the given program and satisfying assignments of the resulting Boolean formula.

Theorem 4.5 (Correctness)**.** *Let* Π *be a tight program, where the treewidth of* $\mathcal{G}_\Pi$ *is at most* k*. Then, the Boolean formula* F *obtained by the reduction above on* Π *and an LTD* $\mathcal{T} = (T, \chi, \delta_\Pi)$ *of* Π*, consisting of Formulas (4.4)–(4.8), is correct. Formally, for any answer set* I *of* Π *there is exactly one satisfying assignment of* F *and vice versa.*

Proof. "⇒": Given an answer set M of Π. Then, we construct a model I of F as follows. For each $x \in \mathrm{at}(\Pi)$, we let (c1) $x \in I$ if $x \in M$. For each node t of T, and $x \in \chi(t)$: (c2) If there is a rule $r \in \delta_\Pi(t)$ supporting x, we let both $p_{\leq t}^x, p_t^x \in I$. Finally, (c3) we set $p_{\leq t}^x \in I$, if $p_{\leq t'}^x \in I$ for $t' \in \mathsf{children}(t)$.

It remains to show that I is indeed a model of F. By (c1), Formulas (4.4) are satisfied by I. Further, by definition of LTDs, for each rule $r \in \Pi$ there is a node t with $r \in \delta_\Pi(t)$. Consequently, M is proven with level mapping φ, for each $x \in M$ there is a node t and a rule $r \in \delta_\Pi(t)$ proving x. Then, Formulas (4.5) are satisfied by I due to (c2), and Formulas (4.6) are satisfied by I due to (c3). Finally, by connectedness of TDs, also Formulas (4.7) and (4.8) are satisfied.

Now, in order to show that I is the only model of F with $M \subseteq I$, let us assume towards a contradiction that there is a model I' of I with $M \subseteq I'$ and $I \neq I'$. Observe, however, that Formulas (4.5) and (4.6) are equivalences and that indeed the variables of the form $p_{\leq t}^x$ and p_t^x for every $x \in \mathrm{at}(\Pi)$ and t of T are therefore uniquely determined by M. Consequently, we have that $I' = I$, which contradicts $I \neq I'$.

"⇐": Given any model I of F. Then, we construct an answer set M of Π as follows. We set $a \in M$ if $a \in I$ for any $a \in \mathrm{at}(\Pi)$. It remains to show that M is indeed an answer set of Π. Obviously, by Formulas (4.4), M is a model of Π. Further, observe that either a is

in the bag $\chi(n)$ of the root node $n = \text{root}(T)$ of T, or it is forgotten below n. In both cases we require a node t such that $p^a_{\leq t} \in I$ by Formulas (4.8) and (4.7), respectively. Consequently, by connectedness of $\mathcal{T}$ and Formulas (4.6) there is a node t', where $p^a_{t'} \in I$. But then, since Formulas (4.5) are satisfied by I, there is a rule $r \in \Pi_{t'}$ supporting a. Therefore, we have that M is an answer set of Π and it is easy to see that M is uniquely determined for the given model I of F. □

In the following, we show that the reduction consisting of Formulas (4.4)–(4.8) indeed linearly preserves the treewidth and we also state runtime properties.

Theorem 4.6 (Treewidth-Awareness)**.** *Let Π be a tight program. Then, the treewidth of Boolean formula F obtained by the reduction above, consisting of Formulas (4.4)–(4.8), by using Π and a join-nice LTD $\mathcal{T} = (T, \chi, \delta_\Pi)$ of Π of width k is in $\mathcal{O}(k)$.*

Proof. We construct a TD $\mathcal{T}' = (T, \chi')$ of $\mathcal{G}_F$ to show that the width of $\mathcal{T}'$ increases only slightly (compared to k). To this end, let t be a node of T with $\mathsf{children}(t) = \langle t_1, \ldots, t_o \rangle$ and let $\hat{t}$ be the parent of t (if exists). We inductively define $\chi'(t) := \chi(t) \cup \{p^y_{\leq t'}, p^x_t, p^x_{\leq t} \mid t' \in \{t_1, \ldots, t_o\}, x \in \chi(t), y \in \chi(t) \cap \chi(t')\}$. Observe that indeed $\mathcal{T}'$ is a TD of $\mathcal{G}_F$. Further, $|\chi'(t)| \leq k + k \cdot (o + 2)$. Thus, the width of TD $\mathcal{T}'$ is in $\mathcal{O}(k)$, since $o = 2$ in a join-nice LTD like $\mathcal{T}$. □

Corollary 4.7 (Runtime)**.** *Let Π be a tight program, where the treewidth of $\mathcal{G}_\Pi$ is at most k. Then, for a given tree decomposition $\mathcal{T}$ of primal graph $\mathcal{G}_\Pi$, the reduction above on Π and $\mathcal{T} = ((N, E), \chi)$ runs in time $\mathcal{O}(k^2 \cdot (|N| + |\Pi|))$.*

Proof. First, we compute a non-redundant, join-nice LTD $\mathcal{T}' = (T, \chi', \delta_\Pi)$ of Π in time $\mathcal{O}(k \cdot (|N| + |\Pi|))$, cf. Propositions 2.7 and 2.22. For each of the $\mathcal{O}(k \cdot (|N| + |\Pi|))$ many nodes of T, we construct Formulas (4.4)–(4.8), which runs in time $\mathcal{O}(k)$ for Formulas (4.4) and (4.6)–(4.8) since $\mathcal{T}'$ is join-nice. Then, since $\mathcal{T}'$ is non-redundant, every rule $r \in \Pi$ appears in at most $|H_r| \leq k$ many instances of Formulas (4.5), increasing for each of these appearances the formula size by $\mathcal{O}(k)$. Overall, we end up with runtime $\mathcal{O}(k^2 \cdot (|N| + |\Pi|))$, which is polynomial in $|\text{at}(\Pi)|$. □

Recall Proposition 3.9, which states that both Tight Asp and Supported Models cannot be solved in time better than single exponential in the treewidth. This also implies, that under reasonable assumptions we cannot significantly improve the reduction above. More precisely, assuming the exponential time hypothesis, there is no reduction from Tight Asp to Sat that decreases the treewidth sub-linearly, while still having polynomial runtime.

Corollary 4.8. *Let Π be a tight program, where the treewidth of $\mathcal{G}_\Pi$ is k. Then, under the ETH, the treewidth of the resulting Boolean formula of the reduction consisting of Formulas (4.4)–(4.8) can not be significantly improved, i.e., under the ETH one cannot reduce Π to Boolean formula F in time $2^{o(k)} \cdot \text{poly}(|\text{at}(\Pi)|)$ such that $\mathsf{tw}(\mathcal{G}_F)$ is in $o(k)$.*

Proof. Towards a contradiction assume the contrary of this claim. Then, we can reduce Π to a Boolean formula F, running in time $2^{o(k)} \cdot \text{poly}(|\text{at}(\Pi)|)$ with $\mathsf{tw}(\mathcal{G}_F)$ being in $o(k)$. Finally, we use an algorithm for Sat [Samer and Szeider, 2010] on F to solve F with n variables in time $2^{o(k)} \cdot \text{poly}(|n|)$, which contradicts Proposition 3.9. □

4.3 Decomposition-Guided Reduction from HCF Asp to Sat

Next, we provide a decomposition-guided reduction for HCF Asp, which therefore also works for Normal Asp. Naturally, there are numerous reductions from Asp [Clark, 1977; Ben-Eliyahu and Dechter, 1994; Lin and Zhao, 2003; Janhunen, 2006; Alviano and Dodaro, 2016] and extensions thereof [Bomanson and Janhunen, 2013; Bomanson, 2017] to Sat. Some of these existing reductions for the prominent fragment of normal programs cause only a sub-quadratic blow-up in the number of variables (auxiliary variables), which is unavoidable [Lifschitz and Razborov, 2006] if the answer sets should be preserved (bijectively). There are also further studies on certain classes of programs and their relationships in the form of whether there exist certain reductions between these classes, thereby bijectively preserving the answer sets. These studies result in an expressive power hierarchy among program classes [Janhunen, 2006]. However, structural dependency in form of, e.g., treewidth, has not been considered yet. If one considers the structural dependency in form of treewidth, existing reductions cause quadratic or even unbounded overhead in the treewidth in general.

On the contrary, we present a novel reduction for normal (and even HCF) programs that increases the treewidth k at most *sub-quadratically* ($k \cdot \log(k)$). This is indeed interesting as there is a close connection [Atserias et al., 2011] between resolution-width and treewidth, resulting in efficient Sat solver runs on instances of small treewidth. As a result, our reduction could improve solving approaches by means of Sat solvers, e.g., ASSAT [Lin and Zhao, 2004b], lp2sat [Janhunen, 2006], or lp2acyc [Gebser et al., 2014; Bomanson et al., 2016]. Later we show in Section 5.2 that under the ETH one can not significantly improve the reduction, i.e., avoid the sub-quadratic increase of treewidth.

Having the basic concept of dynamic programming and decomposition-guided reductions in mind, cf. Sections 3.2.2 and 4.1, we use these ideas to design a reduction from a normal program Π to a Boolean formula F, which only slightly increases treewidth. The reduction is inspired by the table algorithm for HCF Asp, cf. Section 3.2, and the idea of ordering atoms [Janhunen, 2006] according to their level during evaluation (level mapping). Intuitively, *global* level mappings, which refer to level mappings applied for the whole program or on an SCC-by-SCC basis, can cause already huge blowup in the treewidth. If for example all atoms are ordered at once, this can cause large rules with more than treewidth many atoms. As a result, we apply these level mappings only locally within the bags of a TD and end up with a decomposition-guided reduction. Note that while this approach might look similar to existing techniques that are applied on an SCC-by-SCC basis [Janhunen, 2006; Gebser et al., 2014; Bomanson et al., 2016],

the approach is slightly different, as different components of the positive dependency graph D_Π might be spread among different bags of a TD and a bag might only contain parts of components. If some components are required to be spread across any TD of primal graph $\mathcal{G}_\Pi$, whose width coincides with the treewidth, only parts of cycles of dependency graph D_Π can be analyzed by a table algorithm in a bag. This is exactly the underlying reason what makes the problem Asp for treewidth slightly harder than the decision problem Sat (under the ETH, cf. Section 5.2).

Consequently, we still require level mappings, but instead of global level mappings or level mappings per component, we use local level mappings, which only order within bags. This is carried out in such a way that our reduction is decomposition-guided and therefore guided by a TD $\mathcal{T} = (T, \chi)$ of primal graph $\mathcal{G}_\Pi$. It uses core ideas of dynamic programming along $\mathcal{T}$ in order to ensure only a slight increase in treewidth of the resulting Boolean formula. Intuitively, thereby the aforementioned reduction takes care to keep the increase of width local, i.e., the increase of width happens *within* the bags of $\mathcal{T}$. Concretely, if $\mathsf{width}(\mathcal{T})$ is bounded by some value $\mathcal{O}(k)$, the treewidth of the resulting formula F is at most $\mathcal{O}(k \cdot \log(k))$.

For encoding level mappings along a TD, we need the following notation.

Definition 4.9 (cf. Definition 3.10)**.** *Let Π be a program, $\mathcal{T} = (T, \chi, \delta_\Pi)$ be an LTD of Π, and t be a node of T. Then, we refer to a level mapping over $\chi(t)$ by* t-local level mapping*. Further, a* $\mathcal{T}$-local level mapping *is a set containing one t-local level mapping φ_t for every t of T such that there is an interpretation I with (1)* satisfiability*: $I \models \delta_\Pi(t)$ for every node t of T, (2)* provability*: for every $a \in I$, there is a node t of T and a rule $r \in \delta_\Pi(t)$ proving a, and (3)* compatibility*: for every nodes t, t' of T and every $a, b \in \chi(t) \cap \chi(t')$, whenever $\varphi_t(a) < \varphi_t(b)$ then $\varphi_{t'}(a) < \varphi_{t'}(b)$.*

For a level mapping φ, we use the *canonical t-local level mapping* $\hat{\varphi}_t$ for each t of T as follows. Intuitively, atoms $a \in \chi(t)$ with smallest level $\varphi(a)$ among all atoms in $\chi(t)$ get $\hat{\varphi}_t(a) = 0$, second-smallest get value 1, and so on. Formally, we define $\hat{\varphi}_t(a) := \mathrm{ord}_t(a, \varphi) - 1$ for each $a \in \chi(t)$, where $\mathrm{ord}_t(a, \varphi)$ is the ordinal number (rank) of a according to smallest level $\varphi(a)$ among $\chi(t)$.

Example 4.10. *Consider program Π, answer set $I = \{b, c, d\}$, and level mapping $\varphi = \{b \mapsto 0, d \mapsto 1, c \mapsto 2\}$ of Example 2.4. Level mapping φ can easily be extended to level mapping $\varphi' := \{a \mapsto 0, e \mapsto 0, b \mapsto 0, d \mapsto 1, c \mapsto 2\}$ over $\mathrm{at}(\Pi)$. Then, using TD $\mathcal{T}$ of $\mathcal{G}_\Pi$, we can construct $\mathcal{T}$-local level mapping $\mathcal{M} := \{\hat{\varphi}_{t_1}, \hat{\varphi}_{t_2}, \hat{\varphi}_{t_3}\}$ of φ', where $\hat{\varphi}_{t_1} = \{e \mapsto 0, d \mapsto 1, c \mapsto 2\}$, $\hat{\varphi}_{t_2} = \{a \mapsto 0, b \mapsto 0\}$, and $\hat{\varphi}_{t_3} = \{e \mapsto 0, b \mapsto 0, d \mapsto 1\}$. Consider a TD $\mathcal{T}'$ of $\mathcal{G}_\Pi$, which is similar to $\mathcal{T}$, but t_1 has a child node t', whose bag is $\{c, e\}$. Then, $\mathcal{M} \cup \{\hat{\varphi}_{t'}\}$ with $\hat{\varphi}_{t'} = \{e \mapsto 0, c \mapsto 1\}$ is a $\mathcal{T}'$-local level mapping.*

In our reduction, we use the following Boolean variables. For each atom $x \in \mathrm{at}(\Pi)$, we use x also as Boolean variable. Further, similar to above, we use variables p^x_t and $p^x_{\leq t}$ to indicate that x was proven in node t and below t, respectively. For each atom $x \in \chi(t)$

of each node t of T, we use $\lceil \log(|\chi(t)|) \rceil$ many variables of the form $b^i_{x_t}$ forming the i-th bit of the t-local level (in binary) of x. By the shortcut notation $[\![x]\!]_{t,j}$, we refer to the *conjunction of literals over bits* $b^i_{x_t}$ for $1 \leq i \leq \lceil \log(|\chi(t)|) \rceil$ according to the representation of the number j in binary. For atoms $x, x' \in \chi(t)$ of node t of T, we use the following notation to indicate that atom x is ordered before atom x':

$$x \prec_t x' := \bigvee_{1 \leq i \leq \lceil \log(|\chi(t)|) \rceil} (b^i_{x'_t} \wedge \neg b^i_{x_t} \wedge \bigwedge_{i < j \leq \lceil \log(|\chi(t)|) \rceil} (b^j_{x_t} \longrightarrow b^j_{x'_t})). \tag{4.9}$$

Example 4.11. *Consider Example 4.10 and the $\mathcal{T}$-local level mapping $\mathcal{M} = \{\varphi_{t_1}, \varphi_{t_2}, \varphi_{t_3}\}$. One could encode level $\varphi_{t_1}(e) = 0$ using two bit variables $b^1_{e_{t_1}}, b^2_{e_{t_1}}$ and forcing it to false. This results in formula $[\![e]\!]_{t_1,0} = \neg b^1_{e_{t_1}} \wedge \neg b^0_{e_{t_1}}$. Then, we formulate $\varphi_{t_1}(d) = 1$ by $[\![d]\!]_{t_1,1} = \neg b^1_{d_{t_1}} \wedge b^0_{d_{t_1}}$, and $\varphi_{t_1}(c) = 2$ by $[\![c]\!]_{t_1,2} = b^1_{c_{t_1}} \wedge \neg b^0_{c_{t_1}}$. For the whole resulting formula, $(e \prec_{t_1} d)$, $(d \prec_{t_1} c)$ as well as $(e \prec_{t_1} c)$ hold.*

4.3.1 Decomposition-Guided Reduction to Sat

For solving consistency, we require to construct the following Formulas (4.4), (4.6)–(4.8), as well as (4.10)–(4.11) below *for each TD node t* of T having child nodes $\mathsf{children}(t) = \{t_1, \ldots, t_\ell\}$. Thereby, these formulas aim at constructing $\mathcal{T}$-local level mappings along the TD $\mathcal{T}$, where Formulas (4.4) ensure satisfiability, Formulas (4.10) take care of compatibility along the TD, and Formulas (4.11) enforce provability within a node, which is then guided along the TD by Formulas (4.6) to (4.8).

$$\bigvee_{b \in B^+_r} \neg b \vee \bigvee_{a \in B^-_r \cup H_r} a \qquad \text{for each } r \in \delta_\Pi(t) \tag{4.4}$$

$$(x \prec_{t'} y) \longleftrightarrow (x \prec_t y) \qquad \text{for each } t' \in \mathsf{children}(t) \text{ and } x, y \in \chi(t) \cap \chi(t') \text{ with } x \neq y \tag{4.10}$$

$$p^x_t \longleftrightarrow \bigvee_{r \in \delta_\Pi(t), x \in H_r} (\bigwedge_{b \in B^+_r} b \wedge x \wedge (b \prec_t x) \wedge \bigwedge_{a \in B^-_r \cup (H_r \setminus \{x\})} \neg a) \qquad \text{for each } x \in \chi(t) \tag{4.11}$$

$$p^x_{\leq t} \longleftrightarrow p^x_t \vee (\bigvee_{t' \in \mathsf{children}(t), x \in \chi(t')} p^x_{\leq t'}) \qquad \text{for each } x \in \chi(t) \tag{4.6}$$

$$x \longrightarrow p^x_{\leq t'} \qquad \text{for each } t' \in \mathsf{children}(t) \text{ and } x \in \chi(t') \setminus \chi(t) \tag{4.7}$$

$$x \longrightarrow p^x_{\leq n} \qquad \text{for each } x \in \chi(n) \text{ with } n = \mathrm{root}(T) \tag{4.8}$$

Concretely, Formulas (4.4) ensure that the variables of the constructed Boolean formula F are such that all (bag) rules are satisfied. Then, whenever in node t an atom x has a

smaller level than an atom y (using $\prec_t$), this must hold also for the parent node of t and vice versa, cf. Formulas (4.10). Formulas (4.11) take care that an atom x is freshly proven in node t if and only if there is at least one rule $r \in \delta_\Pi(t)$ proving x. At the same time we ensure by Formulas (4.6) that an atom x is proven up to node t if and only if it is proven up to some child node of t or freshly proven in node t. Finally, Formulas (4.7) guarantee, for nodes t removing bag atom x, i.e., $x \in \chi(t) \setminus \chi(t')$, that x is proven if x is set to true. Similarly, this is required for atoms $x \in \chi(n)$ that are in the root node $n = \mathrm{root}(T)$ and therefore never forgotten, cf. Formulas (4.8).

Example 4.12. *Recall program* Π *from Example 2.4, and TD* $\mathcal{T}$ *of* $\mathcal{G}_\Pi$ *given in Figure 2.2. We briefly show Formula* F *for node* t_3*.*

Formulas	Formula F
(4.4)	$\neg b \vee d \vee e$; $\neg e \vee d \vee b$; $d \vee b$
(4.10)	$(d \prec_{t_1} e) \leftrightarrow (d \prec_{t_3} e)$; $(e \prec_{t_1} d) \leftrightarrow (e \prec_{t_3} d)$
(4.11)	$p^b_{t_3} \leftrightarrow [e \wedge b \wedge (e \prec_{t_3} b) \wedge \neg d]$; $p^e_{t_3} \leftrightarrow [b \wedge e \wedge (b \prec_{t_3} e) \wedge \neg d]$; $p^d_{t_3} \leftrightarrow [(b \wedge d \wedge (b \prec_{t_3} d) \wedge \neg e) \vee (d \wedge \neg b)]$
(4.6)	$p^b_{\leq t_3} \leftrightarrow (p^b_{t_3} \vee p^b_{\leq t_2})$; $p^d_{\leq t_3} \leftrightarrow (p^d_{t_3} \vee p^d_{\leq t_1})$; $p^e_{\leq t_3} \leftrightarrow (p^e_{t_3} \vee p^e_{\leq t_1})$
(4.7)	$c \rightarrow p^c_{\leq t_1}$; $a \rightarrow p^a_{\leq t_2}$
(4.8)	$b \rightarrow p^b_{\leq t_3}$; $d \rightarrow p^d_{\leq t_3}$; $e \rightarrow p^e_{\leq t_3}$

Next, we show that the reduction is indeed aware of the treewidth and that the treewidth is only slightly increased.

Theorem 4.13 (Treewidth-awareness)**.** *The reduction from an HCF program* Π *and a join-nice LTD* $\mathcal{T} = (T, \chi, \delta_\Pi)$ *of* Π *to Boolean formula* F *consisting of Formulas (4.4), (4.6)–(4.8), as well as (4.10)–(4.11) only slightly increases treewidth. Concretely, if* k *is the width of* $\mathcal{T}$*, then the treewidth of* $\mathcal{G}_F$ *is at most* $\mathcal{O}(k \cdot \log(k))$*.*

Proof. We construct a TD $\mathcal{T}' = (T, \chi')$ of $\mathcal{G}_F$ to show that the width of $\mathcal{T}'$ increases only slightly (compared to k). To this end, let t be a node of T with $\mathsf{children}(t) = \langle t_1, \ldots, t_o \rangle$ and let $\hat{t}$ be the parent of t (if exists). We define $B(t, x) := \{b^j_{x_t} \mid x \in \chi(t), 1 \leq j \leq \lceil \log(|\chi(t)|) \rceil\}$. Then, we inductively define $\chi'(t) := \chi(t) \cup (\bigcup_{x \in \chi(t)} B(t, x) \cup B(\hat{t}, x)) \cup \{p^y_{\leq t'}, p^x_t, p^x_{\leq t} \mid t' \in \{t_1, \ldots, t_o\}, x \in \chi(t), y \in \chi(t) \cap \chi(t')\}$. Observe that indeed $\mathcal{T}'$ is a TD of $\mathcal{G}_F$. Further, $|\chi'(t)| \leq k + k \cdot \lceil \log(k) \rceil \cdot 2 + k \cdot (o + 2)$. Thus, the width of TD $\mathcal{T}'$ is in $\mathcal{O}(k \cdot \log(k))$, since $o = 2$ in a join-nice LTD like $\mathcal{T}$. □

Corollary 4.14. *The reduction from an HCF program* Π *and a TD* $\mathcal{T} = (T, \chi)$ *of* $\mathcal{G}_\Pi$ *with* $T = (N, E)$ *to Boolean formula* F *consisting of Formulas (4.4), (4.6)–(4.8), as well as (4.10)–(4.11) uses at most* $\mathcal{O}(k \cdot \log(k) \cdot |N|)$ *many variables.*

Proof. The result follows immediately from the construction of Theorem 4.13. □

Later we will see the lower bound for Normal Asp, which indicates that one cannot expect to significantly improve this increase of treewidth. Next, we present consequences for the runtime required to create the Boolean formula.

Theorem 4.15 (Runtime)**.** *The reduction from an HCF program* Π *and a TD* $\mathcal{T} = (T, \chi)$ *of* $\mathcal{G}_\Pi$ *with* $T = (N, E)$ *to Boolean formula* F *consisting of Formulas (4.4), (4.6)–(4.8), as well as (4.10)–(4.11) runs in time* $\mathcal{O}(k^2 \cdot \log(k)^2 \cdot (|N| + |\Pi|))$*, where* k *forms the width of* $\mathcal{T}$*.*

Proof. First, we compute a non-redundant, join-nice LTD $\mathcal{T}' = (T, \chi', \delta_\Pi)$ of Π in time $\mathcal{O}(k \cdot (|N| + |\Pi|))$, cf. Propositions 2.7 and 2.22. Quadratic runtime $\log(k)^2$ is due to Definition 4.9, which is used k^2 many times in Formulas (4.10) for each node of T. Further, each rule $r \in \Pi$, as given by δ_Π of the non-redundant LTD, is involved once in Formulas (4.11), as well as each atom $x \in H_r$. For each rule $r \in \Pi$ and atom in H_r these formulas are of size $\mathcal{O}(k \cdot \log(k)^2)$ due to Definition 4.9, which results in runtime $\mathcal{O}(|\Pi| \cdot k \cdot (k \cdot \log(k)^2))$. Overall, we end up with runtime $\mathcal{O}(k^2 \cdot \log(k)^2(|N| + |\Pi|))$, which is polynomial in $|\mathrm{at}(\Pi)|$. □

Recall that a TD of $\mathcal{G}_\Pi$ of width $\mathsf{tw}(\mathcal{G}_\Pi)$, having only $\mathcal{O}(|\mathrm{at}(\Pi)|)$ many nodes [Kloks, 1994, Lemma 13.1.2] always exists. Further, since $k^2 \cdot \log(k)^2$ might be much smaller than $\log(|\mathrm{at}(\Pi)|)$, for some programs this reduction might pay off compared to global or component-based level mappings used in tools like lp2sat [Janhunen, 2006] or lp2acyc [Gebser et al., 2014; Bomanson et al., 2016].

4.3.2 Correctness of the Reduction

Now, we discuss the correctness of our reduction, which establishes that $\mathcal{T}$-local level mappings encoded by Formulas (4.4), (4.6)–(4.8), as well as (4.10)–(4.11) follow ideas of the characterization of answer sets for HCF programs.

Theorem 4.16 (Correctness)**.** *The reduction from an HCF program* Π *and an LTD* $\mathcal{T} = (T, \chi, \delta_\Pi)$ *of* Π *to Boolean formula* F *consisting of Formulas (4.4), (4.6)–(4.8), as well as (4.10)–(4.11) is correct. Concretely, for each answer set of* Π *there is a model of* F*. Further, for each model of* F *there is exactly one answer set of* Π *restricted to* $\mathrm{at}(\Pi)$*.*

Proof. "⇒": Given an answer set M of Π. Then, there is a level mapping φ over $\mathrm{at}(\Pi)$, where every atom of M is proven. Next, we construct a model I of F as follows. For each $x \in \mathrm{at}(\Pi)$, we let (c1) $x \in I$ if $x \in M$. For each node t of T, and $x \in \chi(t)$: (c2) For every $l \in [\![x]\!]_{t,i}$ with $i = \hat{\varphi}_t(x)$, we set $l \in I$ if l is a variable. (c3) If there is a rule $r \in \delta_\Pi(t)$ proving x, we let both $p^x_{\leq t}, p^x_t \in I$. Finally, (c4) we set $p^x_{\leq t} \in I$, if $p^x_{\leq t'} \in I$ for $t' \in \mathsf{children}(t)$.

It remains to show that I is indeed a model of F. By (c1), Formulas (4.4) are satisfied by I. Further, by (c2) of I, the order of φ is preserved among $\chi(t)$ for each node t of T, therefore Formulas (4.10) are satisfied by I. Further, by definition of TDs, for

each rule $r \in \Pi$ there is a node t with $r \in \delta_\Pi(t)$. Consequently, M is proven with level mapping φ, for each $x \in M$ there is a node t and a rule $r \in \delta_\Pi(t)$ proving x. Then, Formulas (4.11) are satisfied by I due to (c3), and Formulas (4.6) are satisfied by I due to (c4). Finally, by connectedness of TDs, also Formulas (4.7) and (4.8) are satisfied.

"$\Leftarrow$": Given any model I of F. Then, we construct an answer set M of Π as follows. We set $a \in M$ if $a \in I$ for any $a \in \mathrm{at}(\Pi)$. We define for each node t a t-local level mapping φ_t, where we set $\varphi_t(x)$ to j for each $x \in \chi(t)$ such that j is the decimal number of the binary number for x in t given by I. Concretely, $\varphi_t(x) := j$, where j is such that $I \models [\![x]\!]_{t,j}$. Then, we define a level mapping φ iteratively as follows. We set $\varphi(a) := 0$ for each $a \in \mathrm{at}(\Pi)$, where there is no node t of T with $\varphi_t(b) < \varphi_t(a)$. Then, we set $\varphi(a) := 1$ for each $a \in \mathrm{at}(\Pi)$, where there is no node t of T with $\varphi_t(b) < \varphi_t(a)$ for some $b \in \chi(t)$ not already assigned in the previous iteration, and so on. In turn, we construct φ iteratively by assigning increasing values to φ. Observe that φ is well-defined, i.e., each atom $a \in \mathrm{at}(\Pi)$ gets assigned exactly one value since it cannot be the case for two nodes t, t' and atoms $x, x' \in \chi(t) \cap \chi(t')$ that $\varphi_t(x) < \varphi_t(x')$, but $\varphi_{t'}(x) \geq \varphi_{t'}(x')$. Indeed, this is prohibited by Formulas (4.10) and connectedness of $\mathcal{T}$ ensuring that $\mathcal{T}$ restricted to x is still connected.

It remains to show that φ is a level mapping for Π proving M. Assume towards a contradiction that there is an atom $a \in M$ that is not proven. Observe that either a is in the bag $\chi(n)$ of the root node n of T, or it is forgotten below n. In both cases we require a node t such that $p^a_{\leq t} \in I$ by Formulas (4.8) and (4.7), respectively. Consequently, by connectedness of $\mathcal{T}$ and Formulas (4.6) there is a node t', where $p^a_{t'} \in I$. But then, since Formulas (4.11) are satisfied by I, there is a rule $r \in \delta_\Pi(t')$ proving a with $\varphi_{t'}$. Therefore, since by construction of φ there cannot be a node t of T with $x, x' \in \chi(t)$, $\varphi_t(x) < \varphi_t(x')$, but $\varphi(x) \geq \varphi(x')$, we have that r is proving a with φ. □

The proof above allows to conclude the following corollary.

Corollary 4.17 (Preservation of Answer Sets)**.** *The reduction from an HCF program* Π *and an LTD* $\mathcal{T} = (T, \chi, \delta_\Pi)$ *of* Π *to Boolean formula* F *consisting of Formulas (4.4), (4.6)–(4.8), as well as (4.10)–(4.11) preserves answer sets with respect to* $\mathrm{at}(\Pi)$*. Concretely, for each answer set of* Π *there is exactly one model of* F *restricted to variables in* $\mathrm{at}(\Pi)$*. Conversely, for each model of* F *there is exactly one answer set of* Π*.*

However, in general we have that for an answer set of Π, there might be several models of the Boolean formula obtained by the reduction above.

In general, we do not expect to get rid of all redundant $\mathcal{T}$-local level mappings for an answer set. The reason for this expectation lies in the fact that the different (chains of) rules required for setting the position for an atom a that is part of cycles of D_Π might be spread across the whole tree decomposition. Therefore, these local level mappings might not provide the same information that we get from global level mappings [Janhunen, 2006], where we have absolute values. Instead, these local level mappings are insufficient to conclude absolute positions without further information. This is clarified in the following example.

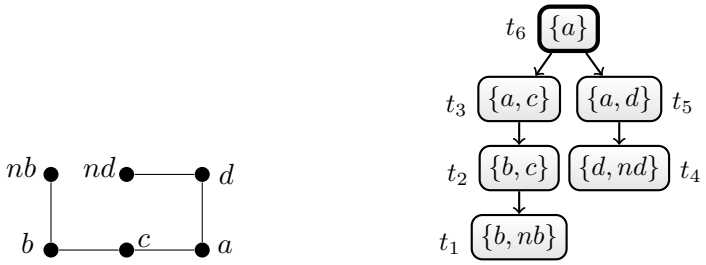

Figure 4.3: Graph $\mathcal{G}_{\Pi'}$ (left) and a tree decomposition $\mathcal{T}'$ of $\mathcal{G}_{\Pi'}$ (right), where program Π' is given in Example 4.18.

Example 4.18. *Consider the program* $\Pi' := \{b \vee nb \leftarrow; c \leftarrow b; a \leftarrow c; d \vee nd \leftarrow; a \leftarrow d\}$. *Observe that program* Π' *has the three answer sets* $\{a, b, c, nd\}$, $\{a, c, d, nb\}$, *as well as* $\{a, b, c, d\}$. *Assume the TD* $\mathcal{T}' = (T', \chi')$ *of Figure 4.3, whose width is* 1 *and equals the treewidth of* $\mathcal{G}_\Pi$. *This particular TD* $\mathcal{T}'$ *is such that* $\Pi_{t_3} = \{a \leftarrow c\}$ *and* $\Pi_{t_5} = \{a \leftarrow d\}$ *and in the following we consider the resulting canonical LTD of* $\mathcal{T}'$. *The issue is that node* t_3 *only considers atoms* a, c *and node* t_5 *only considers atoms* a, d. *Now, assume answer set* $M = \{a, b, c, d\}$. *Then, given only* t_3*-local level mappings and* t_5*-local level mappings, we cannot conclude a unique, canonical global level mapping for* $\{a, c, d\} \subseteq M$. *In particular, one could prove* a *with either* $a \leftarrow c$ *or with* $a \leftarrow d$ *(or both). From a global perspective the latter rule would be preferred to prove* a, *since it allows a level mapping with a smaller position for* a. *This is witnessed by the corresponding level mapping* $\varphi := \{b \mapsto 0, c \mapsto 1, d \mapsto 0, a \mapsto 1\}$ *for* M. *If instead we use the rule* $a \leftarrow c$ *for proving* a, *this would require level mapping* $\varphi' := \{b \mapsto 0, c \mapsto 1, a \mapsto 2, d \mapsto 0\}$, *i.e.,* φ *is preferred since* $\varphi(a) < \varphi'(a)$. *However, this information is "lost" due to the usage of local level mappings, which makes it hard to define canonical level mappings. Therefore our constructed Boolean formula yields two satisfying assignments for* M *in this case, corresponding to proving* a *either with* $a \leftarrow d$ *or* $a \leftarrow c$. *In general, a TD similar to* $\mathcal{T}'$ *can not be avoided. In particular, one can construct programs, where similar situations have to occur in every TD of smallest width.*

One can even devise further corner cases, where without absolute orders it is hard to verify whether it is indeed required that an atom precedes an other atom. This is still the case, if for each answer set M of Π, and every $a \in M$, there can be only one rule $r \in \Pi$ supporting a with M. We refer to such HCF programs Π by *uniquely provable*. Note that even for uniquely provable programs, there might be several cycles in its positive dependency graph. In fact, the program that will be used for the hardness result of Normal Asp and treewidth in Section 5.2 is uniquely provable. However, even for uniquely provable programs and any TD of $\mathcal{G}_\Pi$, there is in general no bijective correspondence between answer sets of Π and models of Formulas (4.4), (4.6)–(4.8), as well as (4.10)–(4.11). Consequently, one could compare different, *absolute* levels of level mappings, cf. [Janhunen, 2006], instead of the levels relative to one TD node as presented here, which requires to store for each atom in the worst case numbers up to $|\mathrm{at}(\Pi)| - 1$. Obviously, this number is then not bounded by the treewidth, and one cannot encode it

without increasing the treewidth in general. Observe that even if one uses level mappings on a component-by-component basis, similar to related work [Janhunen, 2006], this issue still persists in general since the whole program could be one large component.

4.3.3 Decomposition-Guided Reduction to Sat on CNFs

The reduction given by Formulas (4.4), (4.6)–(4.8), as well as (4.10)–(4.11) can be turned into conjunctive normal form without significantly worsening the results given by Theorem 4.13 or 4.15. To this end, let Π be an HCF program, $\mathcal{T} = (T, \chi, \delta_\Pi)$ be an *atomic* labeled tree decomposition of Π, and t be a node of T. Then, we use for given elements $x, x' \in \chi(t)$ and $1 \leq i \leq \lceil \log(|\chi(t)|) \rceil$ the following shortcut

$$x \prec_t^i x' := b_{x'_t}^i \wedge \neg b_{x_t}^i \wedge \bigwedge_{i<j\leq\lceil \log(|\chi(t)|)\rceil} (b_{x_t}^j \longrightarrow b_{x'_t}^j). \tag{4.12}$$

Further, we require additional auxiliary atoms of the form $(b \,\dot{\prec}_t x)$ and $(b \,\dot{\prec}_t^i x)$ for atoms $b \in B_r^+$, $x \in H_r$ with $r \in \delta_\Pi$ and $1 \leq i \leq \lceil \log(|\chi(t)|) \rceil$. The reduction above is slightly modified such that Formulas (4.10) are replaced by both Formulas (4.13) and Formulas (4.14). Indeed, Formulas (4.14) can be converted to CNFs by using rules of DeMorgan and the distributive law for $\wedge$ and $\vee$. However, this results in $2^{2\lceil \log(\chi(t)) \rceil}$ many clauses for each instance of Formulas (4.14) due to the left hand side of Formulas (4.14). In order for Formulas (4.14) to be correct, we need to exclude duplicate levels, which is guaranteed by Formulas (4.13) that result in $2^{\lceil \log(\chi(t)) \rceil}$ many clauses per instance. Then, original Formulas (4.11) are replaced by Formulas (4.15)–(4.17). Intuitively, for atomic LTDs, Formulas (4.17) work similar to Formulas (4.11), but uses auxiliary atoms as discussed above, which are defined by Formulas (4.15) and (4.16). Formulas (4.15) can be also converted to CNFs by using rules of DeMorgan and distributive law for $\wedge$ and $\vee$, whereby in particular the direction from right to left ($\longleftarrow$) results in $2^{\lceil \log(\chi(t)) \rceil}$ many clauses per instance.

$$\bigvee_{b\in B_r^+} \neg b \vee \bigvee_{a\in B_r^- \cup H_r} a \qquad \text{for each } r \in \delta_\Pi(t) \tag{4.4}$$

$$b_{x_t}^1 \neq b_{y_t}^1 \vee \ldots \vee b_{x_t}^l \neq b_{y_t}^l \qquad \text{for each } x, y \in \chi(t) \text{ with } l{=}\lceil \log(|\chi(t)|) \rceil \text{ and } x{\neq}y \tag{4.13}$$

$$(x \prec_{t'}^i y) \longrightarrow \neg(y \prec_t^j x) \qquad \text{for each } t' \in \mathsf{children}(t), x, y \in \chi(t) \cap \chi(t') \text{ with } x \neq y, \text{ and } 1 \leq i \leq \lceil \log(|\chi(t)|) \rceil, 1 \leq j \leq \lceil \log(|\chi(t')|) \rceil \tag{4.14}$$

$$(b \,\dot{\prec}_t^i x) \longleftrightarrow (b \prec_t^i x) \qquad \text{for each } r \in \delta_\Pi(t), x \in H_r, b \in B_r^+, \text{ and } 1 \leq i \leq \lceil \log(|\chi(t)|) \rceil \tag{4.15}$$

$$(b \,\dot{\prec}_t x) \longleftrightarrow \bigvee_{1\leq i\leq\lceil \log(|\chi(t)|)\rceil} (b \,\dot{\prec}_t^i x) \qquad \text{for each } r \in \delta_\Pi(t), x \in H_r, \text{ and } b \in B_r^+ \tag{4.16}$$

$$p_t^x \longleftrightarrow \bigwedge_{b \in B_r^+} b \wedge x \wedge (b \dot{\prec}_t x) \wedge \bigwedge_{a \in B_r^- \cup (H_r \setminus \{x\})} \neg a \qquad \text{for each } r \in \delta_\Pi(t) \text{ and } x \in H_r \tag{4.17}$$

$$p_{\leq t}^x \longleftrightarrow p_t^x \vee \Big(\bigvee_{t' \in \mathsf{children}(t), x \in \chi(t')} p_{\leq t'}^x \Big) \qquad \text{for each } x \in \chi(t) \tag{4.6}$$

$$x \longrightarrow p_{\leq t'}^x \qquad \text{for each } t' \in \mathsf{children}(t) \text{ and } x \in \chi(t') \setminus \chi(t) \tag{4.7}$$

$$x \longrightarrow p_{\leq n}^x \qquad \text{for each } x \in \chi(n) \text{ with } n = \mathrm{root}(T) \tag{4.8}$$

This allows us to obtain similar results as with the reduction above.

Theorem 4.19 (Treewidth-awareness)**.** *Given an HCF program* Π *and a TD* $\mathcal{T}^*$ *of* $\mathcal{G}_\Pi$*, as well as normal program* Π' *obtained by applying Rules (2.1) on* Π*. Then, the reduction from normal program* Π' *and a slightly modified LTD* $\mathcal{T}$ *of TD* $\mathcal{T}^*$ *to Boolean formula* F*, consisting of Formulas (4.4), Formulas (4.6)–(4.8), as well as Formulas (4.13)–(4.17), only slightly increases treewidth. Concretely, if* k *is the width of* $\mathcal{T}^*$*, then the treewidth of* $\mathcal{G}_F$ *is at most* $\mathcal{O}(k \cdot \log(k))$*.*

Proof. First, we transform Π and $\mathcal{T}^*$ into a normal program Π' and transform $\mathcal{T}^*$ into a non-redundant, atomic, join-nice LTD $\mathcal{T} = (T, \chi, \delta_\Pi)$ of Π' of width k by Theorem 2.23 and Proposition 2.7. From this we take program Π' and $\mathcal{T}$, and construct Boolean formula F with the reduction consisting of Formulas (4.4), Formulas (4.6)–(4.8), as well as Formulas (4.13)–(4.17). Then, we construct a TD $\mathcal{T}' = (T, \chi')$ of $\mathcal{G}_F$ to show that the width of $\mathcal{T}'$ increases only slightly (compared to k). To this end, let t be a node of T with $\mathsf{children}(t) = \langle t_1, \ldots, t_o \rangle$ and let $\hat{t}$ be the parent of t (if exists). We define $B(t,x) := \{b_{x_t}^j \mid x \in \chi(t), 1 \leq j \leq \lceil \log(|\chi(t)|) \rceil\}$, as well as $C(t) := \{(b \dot{\prec}_t x), (b \dot{\prec}_t^i x) \mid r \in \delta_\Pi, x \in H_r, b \in B_r^+, 1 \leq i \leq \lceil \log(|\chi(t)|) \rceil\}$. Then, we inductively define $\chi'(t) := \chi(t) \cup (\bigcup_{x \in \chi(t)} B(t,x) \cup B(\hat{t},x)) \cup \{p_{\leq t'}^y, p_t^x, p_{\leq t}^x \mid t' \in \{t_1, \ldots, t_o\}, x \in \chi(t), y \in \chi(t) \cap \chi(t')\} \cup C(t)$. Observe that indeed $\mathcal{T}'$ is a TD of $\mathcal{G}_F$. Further, $|\chi'(t)| \leq k + k \cdot \lceil \log(k) \rceil \cdot 2 + k \cdot (o+2) + k \cdot \lceil \log(k) \rceil$, since in particular $|C(t)|$ being in $\mathcal{O}(k \cdot \log(k))$ due to every rule in δ_Π being normal. Thus, the width of TD $\mathcal{T}'$ is in $\mathcal{O}(k \cdot \log(k))$, since $o = 2$ in a join-nice LTD like $\mathcal{T}$. □

Observe that the detour in the theorem above from Π via the normal program Π' that is obtained by the standard reduction consisting of Rules (2.1) just ensures the following: In addition, we also have the guarantee that for each node of the LTD the labeling not only refers to just one rule r, but also that we consider *at most one* head atom $x \in H_r$ of this rule in an LTD node. In our case this holds since the obtained Π' is normal. This guarantees that we do not use too many auxiliary atoms of the forms $(x \dot{\prec}_t y)$ or $(x \dot{\prec}_t^i y)$ per LTD node. Alternatively, one can also achieve this guarantee by designing a new definition of LTDs for HCF programs (cf. Definition 2.21) such that δ_Π not only assigns rules to nodes, but such that the labeling assigns rules and head atoms of these rules to nodes.

Corollary 4.20. *The reduction from an HCF program* Π *and a TD* $\mathcal{T} = (T, \chi)$ *of* $\mathcal{G}_\Pi$ *with* $T = (N, E)$ *to Boolean formula* F *consisting of Formulas (4.4), Formulas (4.6)–(4.8), as well as Formulas (4.13)–(4.17) uses at most* $\mathcal{O}(k^2 \cdot \log(k) \cdot |N|)$ *many variables. If* Π *is normal, the reduction uses at most* $\mathcal{O}(k \cdot \log(k) \cdot |N|)$ *many variables.*

Proof. The result follows immediately from the construction of Theorem 4.13 on $\mathcal{T}'$. □

Theorem 4.21 (Runtime)**.** *Let* Π *be an HCF program,* $\mathcal{T}^* = (T^*, \chi^*)$ *with* $T^* = (N, E)$ *be a TD of* $\mathcal{G}_\Pi$*, and let* Π' *be a normal program obtained by applying Rules (2.1) on* Π*. Then, the reduction from* Π' *and a slightly modified LTD* $\mathcal{T}$ *of TD* $\mathcal{T}^*$ *to Boolean formula* F *consisting of Formulas (4.4), Formulas (4.6)–(4.8), as well as Formulas (4.13)–(4.17) runs in time* $\mathcal{O}(k^4 \cdot \log(k)^3 \cdot (|N| + |\Pi|))$*, where* k *forms the width of* $\mathcal{T}^*$*.*

Proof. First, we transform Π and $\mathcal{T}^*$ into a normal program Π' and transform $\mathcal{T}^*$ into a non-redundant, atomic, join-nice LTD $\mathcal{T} = (T, \chi, \delta_\Pi)$ of Π' of width k by Theorem 2.23 and Proposition 2.7 in time $\mathcal{O}(k \cdot (|N| + |\Pi|))$. Observe that the number of nodes of T is in $\mathcal{O}(k \cdot (|N| + |\Pi|))$. From this we take program Π' and $\mathcal{T}$, and construct Boolean formula F with the reduction consisting of Formulas (4.4), Formulas (4.6)–(4.8), as well as Formulas (4.13)–(4.17). For every of the $\mathcal{O}(k^2)$ many instances of Formulas (4.13), we obtain $\mathcal{O}(k)$ many clauses of size $\mathcal{O}(\log(k))$. Runtime $\mathcal{O}(k^3 \cdot \log(k)^2 \cdot \log(k))$ is due to the $\mathcal{O}(k^2 \cdot \log(k)^2)$ many Formulas (4.14), each instance resulting in $\mathcal{O}(k)$ many clauses of size $\mathcal{O}(\log(k))$, but for each node of T. Further, each rule $r \in \Pi'$, as given by the non-redundant LTD (δ_Π), is involved once in Formulas (4.17). For each rule $r \in \Pi'$ and atom in B_r these formulas are of size $\mathcal{O}(k \cdot \log(k))$, which results in runtime $\mathcal{O}(k \cdot |\Pi| \cdot k \cdot (k \cdot \log(k)))$. Formulas (4.15) and (4.16) require runtime $\mathcal{O}(k^2 \cdot \log(k) \cdot |\Pi|)$. Overall, we end up with runtime $\mathcal{O}(k^3 \cdot \log(k)^3 \cdot k \cdot (|N| + |\Pi|))$, which is polynomial in $|\mathrm{at}(\Pi)|$. □

Similar to Theorem 4.16, one can establish correctness of the reduction consisting of Formulas (4.4), Formulas (4.6)–(4.8), as well as Formulas (4.13)–(4.17) on atomic labeled tree decompositions of HCF programs.

4.4 Decomposition-Guided Reduction from Asp to Almost Tight Asp

It turns out that the reduction above from Normal Asp to Tight Asp can not avoid a certain blowup in the treewidth. Indeed, we will see in Section 5.2, cf. Theorem 5.35, that under reasonable assumptions, namely the exponential time hypothesis (ETH), deciding the consistency of normal programs is slightly superexponential and one cannot significantly improve in the worst case. For a given normal program Π, where k is the treewidth of the primal graph of the program, this implies that one cannot decide consistency in time significantly better than $2^{k \cdot \lceil \log(k) \rceil} \cdot \mathrm{poly}(|\mathrm{at}(\Pi)|)$. Consequently, one can probably not avoid the increase of treewidth from k to $\mathcal{O}(k \cdot \log(k))$ of the DG reduction above.

While we cannot expect to significantly improve the runtime for normal programs in the worst case, it still is worth to study the underlying reason that makes the worst case bad. It is well-known that positive cycles of the dependency graph are responsible for the hardness [Lifschitz and Razborov, 2006; Janhunen, 2006] of computing answer sets of normal programs. The particular issue with programs Π in combination with treewidth and large cycles is that in a TD of $\mathcal{G}_\Pi$ it might be the case that the cycle spreads across the whole decomposition, i.e., TD bags only contain parts of such cycles, which prohibits to view these cycles (and dependencies) as a whole. This is also the reason of the hardness and explains why under bounded treewidth evaluating normal programs is expected to be slightly harder than evaluating Boolean formulas. However, if a given normal program only has positive cycles of length at most 3, and each atom appears in at most one positive cycle, the properties of TDs ensure that the atoms of each such positive cycle appear in at least one common bag. Indeed, a cycle of length at most 3 forms a completely connected subgraph and therefore it is guaranteed [Kloks, 1994] that the atoms of the cycle are in one common bag of any TD of $\mathcal{G}_\Pi$.

Example 4.22. *Recall program Π of Example 2.4. Observe that in any TD of $\mathcal{G}_\Pi$ it is required that there is a node t with $\chi(t) \subseteq \{b, d, e\}$. Indeed, any cycle of length 3 in the positive dependency graph D_Π (cf. Figure 2.1) is completely connected in $\mathcal{G}_\Pi$, cf. Figure 2.2 (left).*

In the following, we generalize this observation to cycles of length at most ℓ, where we bound the size of these positive cycles in order to improve the lower bound for programs of bounded positive cycle lengths. This provides a significant improvement in the running time on programs, where the size of positive cycles is bounded, and also shows that indeed the case of positive cycle lengths up to 3 can be generalized to lengths beyond 3.

In the remainder of this chapter, we assume an HCF program Π, whose treewidth is given by $k = \mathsf{tw}(\mathcal{G}_\Pi)$. Recall that $\mathrm{scc}(a)$ for each atom a refers to the SCC of a in D_Π. We let $\ell := \max_{a \in \mathrm{at}(\Pi)} |\mathrm{scc}(a)|$ be the *largest SCC size* among all SCCs of Π. This also bounds the lengths of positive cycles. If each atom a appears in at most one positive cycle, we have that $|\mathrm{scc}(a)|$ is the cycle length of a and then ℓ is the length of the largest cycle in Π. We refer to the class of HCF programs, whose largest SCC size is bounded by a parameter ℓ by *SCC-bounded* Asp. Observe that the largest SCC size ℓ is *incomparable* (orthogonal) to treewidth.

Example 4.23. *Consider program Π from Example 2.4. Then, $|\mathrm{scc}(a)| = |\mathrm{scc}(c)| = 1$, $|\mathrm{scc}(b)| = |\mathrm{scc}(d)| = |\mathrm{scc}(e)| = 3$, and $\ell = 3$. Assume a program Π', whose primal graph equals the dependency graph, which is just one large (positive) cycle. This program Π' has treewidth 2 and one can define a TD of $\mathcal{G}_{\Pi'}$, whose bags are constructed along the cycle. However, the largest SCC size $\ell = |\mathrm{at}(\Pi')|$. Conversely, there are programs of large treewidth with no positive cycle.*

There are also related measures from parameterized complexity [Lonc and Truszczyński, 2003], as for example the so-called feedback width [Gottlob et al., 2002b], which depends

on the atoms required to break large SCCs (positive cycles). An other such measure is the smallest backdoor size, which is the smallest size of a set of atoms such that when removed from the program, the resulting program is normal or acyclic [Fichte and Szeider, 2015]. Also programs, where the number of even and/or odd cycles is bounded, have been analyzed [Lin and Zhao, 2004a], which is orthogonal to the size of the largest cycle or largest SCC size ℓ. Indeed, in the worst case, each component might have an exponential number of cycles in ℓ.

Still, bounding sizes of SCCs seems similar to the non-parameterized context, where the consistency of normal programs is compiled to a Boolean formula (Sat) by a reduction based on level mappings that is applied on an SCC-by-SCC basis [Janhunen, 2006]. However, such techniques do not preserve the treewidth. On the other hand, while our approach also uses level mappings and proceeds on an SCC-by-SCC basis, the overall evaluation is not SCC-based, since this completely destroys the treewidth in the worst case. Indeed, already large rules with, e.g., $\mathcal{O}(\log(n))$ many atoms, where n is the number of atoms of the given program, might destroy the treewidth. Instead, the evaluation is guided along a TD by means of a DG reduction. The relaxed notions of Tight Asp allow us to balance treewidth and tightness, where our reduction is designed to increase "tightness" at the cost of an increased treewidth. Note that there are also further related works that implicitly utilize SCCs for solving logic programs [Gebser et al., 2007; Fassetti and Palopoli, 2010].

4.4.1 Towards Tightness Width and Almost Tightness

Now, we focus on the consistency problem for SCC-bounded programs. The largest SCC size in combination with treewidth leads to the development of a novel tightness measure given below. This will then result in a new class of programs that adheres to a certain "level" of tightness, which is given in terms of the following tightness measure.

Definition 4.24 (Tightness Width)**.** *Let Π be an SCC-bounded program of largest SCC size ℓ, $\mathcal{T} = (T, \chi)$ with $T = (N, E)$ be a TD of $\mathcal{G}_\Pi$ of width k, and $t \in N$ be a node of T. Then, we let $\ell_t^{\mathrm{scc}(x)}$ for each atom x of Π be the* size of the SCC *of x restricted to $\chi(t)$ in D_Π, i.e., $\ell_t^{\mathrm{scc}(x)} := |\chi(t) \cap \mathrm{scc}(x)|$. Further, we define the* local SCC size *ℓ_t for a node t as follows $\ell_t := \max\{\ell_t^{\mathrm{scc}(x)} \mid x \in \chi(t)\}$ and finally the* tightness width *ι is defined by $\iota := \max\{\ell_t \mid t \in N\}$.*

Intuitively, the local SCC sizes are bounded by the bag sizes $(k+1)$ as well as the largest SCC size ℓ, but can be even significantly smaller, especially if SCCs are spread across $\mathcal{T}$ and if tree decomposition bags contain parts of many SCCs. This observation motivates a new (relaxed) measure for almost tightness of Asp and treewidth, where even non-tight programs with large cycles are considered almost tight on $\mathcal{T}$, as long as ι is small. We say program Π is *ι-tight on $\mathcal{T}$*, i.e., "almost" tight on $\mathcal{T}$, if $\iota \geq 2$.

We say a program Π is *ι-tight* if Π is ι-tight on a tree decomposition $\mathcal{T}$ of $\mathcal{G}_\Pi$ with $\mathsf{width}(\mathcal{T}) = \mathsf{tw}(\mathcal{G}_\Pi)$, and we refer to the problem Asp restricted to ι-tight programs by ι-Tight

Asp. Notably, if $\iota = 1$, program Π is actually tight and vice versa, as established in the proposition below. Consequently, the concept of almost tightness forms a generalization of tight programs since any tight program is actually 1-tight and for any non-tight program there has to exist some $\iota \geq 2$ with the program being ι-tight.

Proposition 4.25. *A program Π is 1-tight on some tree decomposition $\mathcal{T}$ of $\mathcal{G}_\Pi$ if and only if Π is tight.*

Proof. "$\Longleftarrow$": If Π is tight, every SCC of D_Π is of size 1 and therefore it is easy to see that Π is 1-tight on every TD of $\mathcal{G}_\Pi$.

"$\Longrightarrow$": If Π is 1-tight on some TD $\mathcal{T} = (T, \chi)$ of $\mathcal{G}_\Pi$, then there is no node t of T and $x \in \chi(t)$ with $|\chi(t) \cap \mathrm{scc}(x)| > 1$. Assume towards a contradiction that $|\mathrm{scc}(x)| > 1$. Then, there has to exist an atom $y \in \mathrm{scc}(x)$ with $y \neq x$. Consequently, there is at least one edge $\{x, y\}$ of primal graph $\mathcal{G}_\Pi$. Since $\mathcal{T}$ is a TD of $\mathcal{G}_\Pi$, there has to exist a node t' of T with $x, y \in \chi(t')$. Then, we conclude that $|\chi(t') \cap \mathrm{scc}(x)| > 1$, which contradicts our assumption. Therefore, $|\mathrm{scc}(x)| = 1$ for every $x \in \mathrm{at}(\Pi)$, i.e., Π is tight. □

Observe that even for programs with very large cycles, the tightness width might be actually small.

Example 4.26. *Recall program Π from Example 2.4 and TD $\mathcal{T} = (T, \chi)$ of $\mathcal{G}_\Pi$ of Figure 2.2. Observe that Π is 3-tight on $\mathcal{T}$. Consider program Π' from Example 4.23. While ℓ is large ($\ell = |\mathrm{at}(\Pi')|$), we have that $\iota = 3$ for any TD of $\mathcal{G}_{\Pi'}$ of width 2.*

This is indeed interesting, since the consistency of program Π that is ι-tight on $\mathcal{T}$ for small values of ι can be decided in a runtime that is similar to the runtime of Sat. This significantly improves the upper bound (runtime) as presented in Corollary 4.15 and leads to the following theorem, which we establish in the course of this and the next section.

Theorem 4.27 (Runtime of ι-Tight Asp)**.** *Assume a program Π that is ι-tight on a TD $\mathcal{T}$ of width k, whose number of nodes is linear in $|\mathrm{at}(\Pi)|$. Then, there is an algorithm for deciding the consistency of Π, running in time $2^{\mathcal{O}(k \cdot \log(\iota))} \cdot \mathrm{poly}(|\mathrm{at}(\Pi)|)$.*

Observe that for ι-tight programs Π over TDs of $\mathcal{G}_\Pi$, whose widths coincide with $\mathsf{tw}(\mathcal{G}_\Pi)$, we immediately obtain the corresponding lower bound as follows.

Corollary 4.28. *Unless the ETH fails, the problem Asp on an arbitrary program Π that is ι-tight cannot be solved in time $2^{o(\mathsf{tw}(\mathcal{G}_\Pi) \cdot \log(\iota))} \cdot \mathrm{poly}(|\mathrm{at}(\Pi)|)$.*

Proof. The proof is a direct consequence of Theorem 5.35, since $\iota \leq k$. □

4.4.2 Increasing Almost Tightness via a DG Reduction

Below, we provide a reduction that reduces the tightness width of almost tight programs. To this end, we assume a program Π being ι-tight on an LTD $\mathcal{T} = (T, \chi, \delta_\Pi)$ of Π, whose width k coincides with the treewidth of $\mathcal{G}_\Pi$, and a node t of T. The reduction indirectly relies on *local level mappings* $\varphi_t : A \to \{0, \ldots, \ell_t - 1\}$ for set of atoms $A \subseteq \mathrm{at}(\Pi)$, which is a function mapping each atom $x \in A$ to a level $\varphi(x)$ such that the level does not exceed $\ell_t^{\mathrm{scc}(x)}$, i.e., $\varphi(x) < \ell_t^{\mathrm{scc}(x)}$. However, we require atoms $b^j_{x_t}$, called *local level bits*, for $x \in \mathrm{at}(\Pi)$ and $1 \leq j \leq \lceil \log(\ell_t^{\mathrm{scc}(x)}) \rceil$, for representing the level $\varphi_t(x)$ of x for node t in a mapping φ_t in binary. Further, we use auxiliary atoms of the form $(x \dot{\prec}_t y)$ to indicate that $\varphi_t(x) < \varphi_t(y)$ for $x, y \in \chi(t)$, and provability atoms p^x_t and $p^x_{\leq t}$ as before. We also use the following auxiliary Definition 4.18 for atoms $x, y \in \chi(t)$ and $1 \leq i \leq \lceil \log(\ell_t^{\mathrm{scc}(x)}) \rceil$, which is similar to Definition 4.12, but adapted for Asp and to almost tight programs. Note that Definition 4.18 uses pseudo-Boolean expressions based on math operator $\leq$, which can be easily compiled into a set of $2^{\lceil \log(|\chi(t)|) \rceil}$ many plain sets of atoms.

$$x \prec^i_t x' := \{b^i_{x'_t}, \neg b^i_{x_t}\} \cup \{b^j_{x_t} \leq b^j_{x'_t} \mid i < j \leq \lceil \log(|\chi(t)|) \rceil\}. \tag{4.18}$$

We are ready to discuss the decomposition-guided reduction from program Π being ι-tight on $\mathcal{T}$ to a program Π' that is ι'-tight on a TD $\mathcal{T}'$ with $\iota' \leq \iota$, where $\mathcal{T}'$ is the TD that is anticipated by the reduction, cf. Section 4.1. To this end, let $\mathcal{C}$ be the union of those non-trivial SCCs of D_Π that shall be eliminated. If $\mathcal{C} = \emptyset$, we obtain the program Π as result, and if $\mathcal{C}$ are all vertices of any non-trivial SCC of D_Π, we obtain a tight program, whose treewidth is bounded by $\mathcal{O}(k \cdot \log(\iota))$. Consequently, if $\mathcal{C}$ are the vertices of those non-trivial SCCs that are responsible for large local SCC sizes, one can increase the level of tightness (decrease ι) at the cost of a slight increase of treewidth. The reduction consists of Rules (4.19)–(4.30). First, truth values for each atom $x \in \chi(t) \cap \mathcal{C}$ are guessed by Rules (4.19) and by Rules (4.20) it is ensured that $\delta_\Pi(t)$ is satisfied and that those atoms in the head H_r of a rule $r \in \delta_\Pi(t)$ but not in $\mathcal{C}$ still appear in the head. The next block of Rules (4.21)–(4.24) is used for guessing local level bits for atoms $x \in \chi(t) \cap \mathcal{C}$, cf. Rules (4.21) and defining auxiliary atom $(x \dot{\prec}_t y)$ by Rules (4.22). Observe that the way Rules (4.22) are denoted, pseudo-Boolean expressions based on math operator $\leq$ are used, which can be compiled into $2^{\lceil \log(\ell_t^{\mathrm{scc}(x)}) \rceil}$ many simpler rules per instance of Rules (4.22). Rules (4.23) exclude that two atoms of the same component and in the same node have the same level. While this is not required from a conceptual point of view, it simplifies the definition of Rules (4.24), More concretely, it is not allowed that two neighboring nodes of T order two atoms of the same SCC differently. As already discussed above, if Definition 4.18 is viewed as a set of sets of atoms, every instance of Rules (4.24) can be simplified into several plain constraints, where every set of atoms of $(x \prec^i_t y)$ is combined with every set of atoms of $(y \prec^i_{t'} x)$ accordingly.

Finally, Rules (4.25)–(4.30) derive and ensure provability similar to Formulas (4.11)–(4.8), but only for atoms in $\mathcal{C}$ as well as rules, whose heads contain atoms of $\mathcal{C}$.

$$\{x\} \leftarrow \quad \text{for each } x \in \chi(t) \cap \mathcal{C} \tag{4.19}$$

$$H_r \backslash \mathcal{C} \leftarrow B_r^+, \overline{B_r^- \cup (H_r \cap \mathcal{C})} \quad \text{for each } r \in \delta_\Pi(t) \tag{4.20}$$

$$\{b_{x_t}^j\} \leftarrow \quad \text{for each } x \in \chi(t) \cap \mathcal{C} \text{ and } 1 \leq j \leq \lceil \log(\ell_t^{\mathrm{scc}(x)}) \rceil \tag{4.21}$$

$$(y \dot{\prec}_t x) \leftarrow b_{x_t}^j, \neg b_{y_t}^j, \; b_{y_t}^{j+1} \leq b_{x_t}^{j+1}, \cdots, b_{y_t}^l \leq b_{x_t}^l \quad \text{for each } r \in \delta_\Pi(t), x \in H_r \cap \mathcal{C}, y \in B_r^+ \cap C \text{ with } C = \mathrm{scc}(x), l = \lceil \log(\ell_t^C) \rceil, \text{ and } 1 \leq j \leq l \tag{4.22}$$

$$\leftarrow b_{x_t}^1 = b_{y_t}^1, \ldots, b_{x_t}^l = b_{y_t}^l \quad \text{for each } x, y \in \chi(t) \cap \mathcal{C} \text{ with } C = \mathrm{scc}(x) = \mathrm{scc}(y), l = \lceil \log(\ell_t^C) \rceil, \text{ and } x \neq y \tag{4.23}$$

$$\leftarrow (x \prec_t^i y), (y \prec_{t'}^j x) \quad \text{for each } x, y \in \chi(t) \cap \chi(t') \cap \mathcal{C}, t' \in \mathsf{children}(t), 1 \leq i \leq \lceil \log(\ell_t^C) \rceil, \text{ and } 1 \leq j \leq \lceil \log(\ell_{t'}^C) \rceil \text{ with } \mathrm{scc}(x) = \mathrm{scc}(y) \text{ and } x \neq y \tag{4.24}$$

$$p_t^x \leftarrow x, B_r^+, \overline{B_r^- \cup (H_r \backslash \{x\})}, ([B_r^+ \cap C] \dot{\prec}_t x) \quad \text{for each } r \in \delta_\Pi(t), x \in H_r \cap \mathcal{C} \text{ with } C = \mathrm{scc}(x) \text{ and } B_r^+ \cap C \neq \emptyset \tag{4.25}$$

$$p_t^x \leftarrow x, B_r^+, \overline{B_r^- \cup (H_r \backslash \{x\})} \quad \text{for each } r \in \delta_\Pi(t), x \in H_r \cap \mathcal{C} \text{ with } B_r^+ \cap \mathrm{scc}(x) = \emptyset \tag{4.26}$$

$$p_{\leq t}^x \leftarrow p_t^x \quad \text{for each } x \in \chi(t) \cap \mathcal{C} \tag{4.27}$$

$$p_{\leq t}^x \leftarrow p_{\leq t'}^x \quad \text{for each } x \in \chi(t) \cap \chi(t') \cap \mathcal{C} \text{ with } t' \in \mathsf{children}(t) \tag{4.28}$$

$$\leftarrow x, \neg p_{\leq t'}^x \quad \text{for each } x \in (\chi(t') \cap \mathcal{C}) \setminus \chi(t) \text{ with } t' \in \mathsf{children}(t) \tag{4.29}$$

$$\leftarrow x, \neg p_{\leq n}^x \quad \text{for each } x \in \chi(n) \cap \mathcal{C} \text{ with } n = \mathrm{root}(T) \tag{4.30}$$

4.4.3 Correctness and Treewidth-Awareness

Next, we show that the reduction above is correct, resulting in Lemma 4.29. Then, the DG reduction above allows us to establish Lemma 4.30, which shows the treewidth-awareness of the reduction. Further, in Theorem 4.33, we show that the reduction makes it possible to remove cyclicity.

Lemma 4.29 (Correctness)**.** *Let* Π *be a program that is* ι*-tight on an LTD* $\mathcal{T} = (T, \chi, \delta_\Pi)$ *of* Π *such that* $\mathcal{T}$ *is of width* k*. Then, the program* Π' *obtained by the reduction consisting of Rules (4.19)–(4.30) on* Π*, the union* $\mathcal{C}$ *of some non-trivial SCCs of* D_Π*, and LTD* $\mathcal{T}$ *is correct. Formally, for any answer set* I *of* Π *there is exactly one answer set of* Π' *restricted to* $\mathrm{at}(\Pi)$*. Further, for any answer set of* Π'*, there is exactly one answer set of* Π*.*

Proof. "⇒": Given an answer set M of Π. Then, there is a level mapping φ over M, where every atom of M is proven such that $\varphi(x) \neq \varphi(y)$ with $x, y \in M$. Next, we construct an answer set M' of Π' as follows. For each $x \in \mathrm{at}(\Pi)$, we let (c1) $x \in M'$ if $x \in M$. For each node t of T, and $x \in \chi(t) \cap \mathcal{C}$: (c2) For every $l \in [\![x]\!]_{t,i}$ with $i = \hat{\varphi}_t(x)$, we set $l \in M'$ if l is a variable as well as $(y \dot{\prec}_t x) \in M'$ if $\hat{\varphi}_t(y) < \hat{\varphi}_t(x)$ for $x, y \in \chi(t)$ such that there exists a rule $r \in \delta_\Pi$ with $x \in H_r$ and $y \in B_r^+ \cap \mathrm{scc}(x)$. (c3) If there is a rule $r \in \delta_\Pi(t)$ proving x, we let both $p^x_{\leq t}, p^x_t \in M'$. Finally, (c4) we set $p^x_{\leq t} \in M'$, if $p^x_{\leq t'} \in I$ for $t' \in \mathsf{children}(t)$.

We briefly show that M' is indeed an answer set of Π'. Observe that Rules (4.19) as well as (4.21) are satisfied by any interpretation and are therefore also satisfied by M'. By (c1) and since M satisfies all rules of Π, Rules (4.20) are satisfied by M' as well. By (c2), Rules (4.22) are satisfied. Further, by (c2), the order of φ is preserved among $\chi(t)$ for each node t of T, therefore Rules (4.23) as well as Rules (4.24) are satisfied by M'. Then, by definition of TDs, for each rule $r \in \Pi$ there is a node t with $r \in \delta_\Pi(t)$. Consequently, M is proven with level mapping φ, for each $x \in M$ there is a node t and a rule $r \in \delta_\Pi(t)$ proving x. Then, Rules (4.25) and (4.26) are satisfied by M' due to (c3), and Rules (4.27) and (4.28) are satisfied by M' due to (c4). Finally, by connectedness of TDs, also Rules (4.29) and (4.30) are satisfied.

It remains to show that indeed M' is an answer set of Π'. To this end, we define a level mapping φ' over M' and argue that all rules of Π' are proven by φ'. Let therefore (o1) $\varphi'(a) := \varphi(a)$ for each $a \in M'$. Further, we define for each t of T, as well as $x, y \in \chi(t) \cap \mathcal{C}$ and $1 \leq j \leq \lceil \log(\ell_t^{\mathrm{scc}(x)}) \rceil$ (o2) $\varphi'(b^j_{x_t}) := 0$ and (o3) $\varphi'(y \dot{\prec}_t x) := 1$ if there is a rule $r \in \delta_\Pi(t)$ with $y \in B_r^+ \cap \mathrm{scc}(x)$ and $x \in H_r$. Finally, for each t of T and $x \in \chi(t)$ we let (o4) $\varphi'(p^x_t) := 1 + \max\{1, \varphi(y) \mid y \in \{x\} \cup B_r, r \in \delta_\Pi(t)\}$ as well as (o5) $\varphi'(p^x_{\leq t}) := 1 + \max\{\varphi(y) \mid y \in \{p^x_t, p^x_{\leq t'} \mid t' \in \mathsf{children}(t)\}\}$. Then, atoms in $M' \setminus \mathcal{C}$ are proven by Rules (4.25) and φ', due to (o1). Further, atoms in $M' \cap \mathcal{C}$ are proven by Rules (4.19) and φ', also due to (o1). Similarly, due to (o2) and (o3), Rules (4.21) and (4.22) serve in proving corresponding head atoms in M' by φ'. Then, by (o4) we have that head atoms of Rules (4.25) and (4.26) contained in M' are proven by φ'. Finally, atoms of M' contained in the head of Rules (4.27) and (4.28) can be proven by φ' due to (o5).

"⇐": Given any answer set M' of Π'. Then, we construct an answer set M of Π as follows. We set $a \in M$ if $a \in M' \cap \mathrm{at}(\Pi)$. Observe that since M' satisfies Rules (4.20), we have that M satisfies all rules of Π. It remains to show that M is indeed an answer set of Π. To this end, we define for each node t a t-local level mapping φ_t, where we set $\varphi_t(x)$ to j for each $x \in \chi(t) \cap \mathcal{C}$ such that j is the decimal number of the binary number for x in t given by M'. Concretely, $\varphi_t(x) := j$, where j is such that $M' \models [\![x]\!]_{t,j}$. Then, we define a level mapping φ iteratively as follows. We set $\varphi(a) := 0$ for each $a \in M' \cap \mathcal{C}$, where there is no node t of T and atom $b \in \mathrm{scc}(a)$ with $\varphi_t(b) < \varphi_t(a)$. Then, we set $\varphi(a) := 1$ for each $a \in \mathrm{at}(\Pi) \cap \mathcal{C}$, where there is no node t of T and atom $b \in \mathrm{scc}(a)$, which has not been already assigned in the previous iteration, such that $\varphi_t(b) < \varphi_t(a)$, and so on. In turn, we construct φ for each $a \in M' \cap \mathcal{C}$ iteratively by assigning increasing

values to φ. Then, we complete φ by assigning values to those atoms not yet assigned, i.e., atoms in $(M' \cap \mathrm{at}(\Pi)) \setminus \mathcal{C}$. We do this by observing that since M' is an answer set of Π', there is a level mapping φ' proving M', and by assigning $\varphi(a) := \varphi'(a)$ for each $a \in \chi(t) \setminus \mathcal{C}$. Observe that φ is well-defined. More precisely, each atom $a \in M' \cap \mathcal{C}$ gets assigned exactly one value since it cannot be the case for two nodes t, t' and atoms $x, x' \in \chi(t) \cap \chi(t')$ that $\varphi_t(x) < \varphi_t(x')$, but $\varphi_{t'}(x) \geq \varphi_{t'}(x')$. Indeed, this is prohibited by Rules (4.24) as well as due to connectedness of $\mathcal{T}$ ensuring that $\mathcal{T}$ restricted to x (and x') is still connected. Observe that for each atom $a \in (M' \cap \mathrm{at}(\Pi)) \setminus \mathcal{C}$ we have that φ is well-defined as well.

It remains to show that φ is a level mapping for Π proving M. Assume towards a contradiction that there is an atom $a \in M$ that is not proven. We proceed by case distinction.
Case $a \in \mathcal{C}$: Observe that either a is in the bag $\chi(n)$ of the root node $n = \mathrm{root}(T)$ of T, or it is forgotten below n. In both cases we require a node t such that $p^a_{\leq t} \in M'$ by Rules (4.30) and (4.29), respectively. Consequently, by connectedness of $\mathcal{T}$ and Rules (4.27) and (4.28) there is a node t', where $p^a_{t'} \in M'$. But then, since $p^a_{t'} \in M'$ has to be proven due to M' being an answer set of Π', either one of Rules (4.25) or (4.26) prove $p^a_{t'} \in M'$. Consequently, there is a rule $r \in \delta_\Pi(t')$ proving a with $\varphi_{t'}$. Therefore, since by construction of φ there cannot be a node t of T with $x, x' \in \chi(t)$, $\varphi_t(x) < \varphi_t(x')$, but $\varphi(x) > \varphi(x')$, we have that r is proving a with φ.
Case $a \notin \mathcal{C}$: Since $a \in M'$ can only be proven by one of Rules (4.20), there is a node t' of T and a rule $r \in \delta_\Pi(t')$ proving a with φ_t. Consequently, by construction of φ, r is proving a with φ. □

Next, we show that the treewidth of the reduction is only slightly increased as follows.

Lemma 4.30 (Treewidth-Awareness)**.** *Let Π be a program that is ι-tight on a TD $\mathcal{T}^*$ of $\mathcal{G}_\Pi$ such that $\mathcal{T}^*$ is of width k, and let Π^* be a normal program obtained by applying Rules (2.1) on Π. Then, the treewidth of tight program Π' obtained by the reduction consisting of Rules (4.19)–(4.30) on Π^*, the union $\mathcal{C}$ of all non-trivial SCCs of D_Π, and a slightly modified LTD $\mathcal{T}$ of TD $\mathcal{T}^*$, is in $\mathcal{O}(k \cdot \log(\iota))$.*

Proof. First, we transform Π and $\mathcal{T}^*$ into a normal program Π^* and transform $\mathcal{T}^*$ into a non-redundant, atomic, join-nice LTD $\mathcal{T} = (T, \chi, \delta_\Pi)$ of Π^* of width k by Theorem 2.23 and Proposition 2.7. From this we take program Π^*, $\mathcal{C}$, and $\mathcal{T}$ in order to construct tight program Π' with the reduction consisting of Rules (4.19)–(4.30). Then, we construct a TD $\mathcal{T}' = (T, \chi')$ of $\mathcal{G}_{\Pi'}$ in order to show that the width of $\mathcal{T}'$ increases only slightly (compared to k). To this end, let t be a node of T with $\mathsf{children}(t) = \langle t_1, \ldots, t_o \rangle$ and let $\hat{t}$ be the parent of t (if exists). For every node t of T, whose parent node is t^*, we let $\chi'(t) := \chi(t) \cup \{b^j_{x_t} \mid x \in \chi(t) \cap \mathcal{C}, 1 \leq j \leq \lceil \log(\ell_t^{\mathrm{scc}(x)}) \rceil\} \cup \{p^x_t, p^x_{\leq t}, p_{\leq t^*} \mid x \in \chi(t) \cap \mathcal{C}\} \cup \{(b \dot{\prec}_t x) \mid r \in \delta_\Pi, x \in H_r \cap \mathcal{C}, b \in B^+_r \cap \mathrm{scc}(x)\}$. It is easy to see that indeed all atoms of every instance of Rules (4.19)–(4.30) appear in at least one common bag

of χ'. Further, we have connectedness of $\mathcal{T}'$, i.e., $\mathcal{T}'$ is indeed a TD of $\mathcal{G}_{\Pi'}$ and $|\chi(t)|$ is in $\mathcal{O}(k \cdot \log(\iota))$. □

Finally, we provide brief arguments on the runtime limits imposed by the reduction above.

Theorem 4.31 (Runtime)**.** *Let Π be a program that is ι-tight on a TD $\mathcal{T}^* = (T^*, \chi^*)$ of $\mathcal{G}_\Pi$ with $T^* = (N, E)$ such that $\mathcal{T}^*$ is of width k, and let Π^* be a normal program obtained by applying Rules (2.1) on Π. Then, the reduction consisting of Rules (4.19)–(4.30) on Π^*, the union $\mathcal{C}$ of all non-trivial SCCs of D_Π, and a slightly modified LTD $\mathcal{T}$ of TD $\mathcal{T}^*$, runs in time $\mathcal{O}(k \cdot (k^2 + \iota^3) \cdot \log(k)^3 \cdot (|N| + |\Pi|))$.*

Proof. First, we transform Π into a normal program Π^* and transform $\mathcal{T}^*$ into a non-redundant, atomic, join-nice LTD $\mathcal{T} = (T, \chi, \delta_\Pi)$ of Π^* of width k by Theorem 2.23 and Proposition 2.7 in time $\mathcal{O}(k \cdot (|N| + |\Pi|))$. Observe that the number of nodes of T is in $\mathcal{O}(k \cdot (|N| + |\Pi|))$. From this we take program Π^* and $\mathcal{T}$, and construct tight program Π' with the reduction consisting of Rules (4.19)–(4.30). Runtime $\mathcal{O}(\iota^3 \cdot \log(\iota)^2 \cdot \log(\iota))$ is due to the $\mathcal{O}(\iota^2 \cdot \log(\iota)^2)$ many Rules (4.24), each of size $\mathcal{O}(\iota \cdot \log(\iota))$, but for each node of T. Further, each rule $r \in \Pi'$, as given by δ_Π of the non-redundant LTD, is involved once in Rules (4.25). For each rule $r \in \Pi'$ and atom in B_r these formulas are of size $\mathcal{O}(k \cdot \log(\iota))$, which results in runtime $\mathcal{O}(k \cdot |\Pi| \cdot k \cdot (k \cdot \log(\iota)))$. Overall, we end up with runtime $\mathcal{O}((k^2 + \iota^3) \cdot \log(k)^3 \cdot k \cdot (|N| + |\Pi|))$, which is polynomial in $|\mathrm{at}(\Pi)|$. □

Corollary 4.32. *The reduction from an HCF program Π that is ι-tight on a TD $\mathcal{T}^* = ((N, E), \chi^*)$ of $\mathcal{G}_\Pi$ of width k, to tight program Π' consisting of Rules (4.19)–(4.30) uses at most $\mathcal{O}(k^2 \cdot \log(\iota) \cdot (|N| + |\Pi|))$ many variables. If Π is normal, the reduction uses at most $\mathcal{O}(k \cdot \log(\iota) \cdot |N|)$ many variables.*

Proof. The result follows immediately from the construction of Theorem 4.31 on $\mathcal{T}^*$, since we require $\mathcal{O}(k \cdot \log(\iota))$ many variables for each node of the decomposition $\mathcal{T}$. □

Theorem 4.33 (Removing Cyclicity of ι-Tight Asp)**.** *Let Π be a program being ι-tight on a TD $\mathcal{T}$ of width k. Then, there is a tight program Π' with treewidth in $\mathcal{O}(k \cdot \log(\iota))$ such that the answer sets of Π and Π' projected to $\mathrm{at}(\Pi)$ coincide.*

Proof. Observe that the reduction consisting of Rules (4.19)–(4.30) on Π and $\mathcal{T}$ with $\mathcal{C}$ being the union of all non-trivial SCCs of D_Π runs in polynomial time by Theorem 4.31. The claim follows by correctness (Lemma 4.29) and by treewidth-awareness as given by Lemma 4.30. □

Having established the results above, we are ready to show Theorem 4.27, which states that the consistency of a program Π that is ι-tight on $\mathcal{T}$ can be decided in a runtime that is similar to the runtime of Sat for small ι. This is achieved using the results of this section and the findings of Section 4.2, which together allows us to reduce from ι-Tight Asp to Sat.

Theorem 4.27 (Runtime of ι-Tight Asp)**.** *Assume a program* Π *that is* ι*-tight on a TD* $\mathcal{T}$ *of width* k*, whose number of nodes is linear in* $|\mathrm{at}(\Pi)|$*. Then, there is an algorithm for deciding the consistency of* Π*, running in time* $2^{\mathcal{O}(k \cdot \log(\iota))} \cdot \mathrm{poly}(|\mathrm{at}(\Pi)|)$*.*

Proof. First, we apply the reduction of Section 4.4 on Π and $\mathcal{T}$ on the set $\mathcal{C}$ of all non-trivial SCCs of D_Π. This results in a tight program, which is reduced by the reduction of Section 4.3 to obtain a Boolean formula F. Both reductions run in polynomial time, i.e., runtime bounded by $\mathcal{O}(\mathrm{poly}(\mathrm{at}(\Pi)))$. Finally, formula F, whose treewidth is in $\mathcal{O}(k \cdot \log(\iota))$ by Lemmas 4.30 and 4.6, are solved by an algorithm [Samer and Szeider, 2010] for Sat in time $2^{\mathcal{O}(k \cdot \log(\iota))} \cdot (|\mathrm{at}(\Pi)| + |\Pi|)$. □

Theorem 4.27 assumes a given TD, efficiently computable by means of heuristics [Abseher et al., 2017]. Alternatively, one can compute [Bodlaender et al., 2016] a TD of $\mathcal{G}_\Pi$ of width that is a 5-approximation of $\mathsf{tw}(\mathcal{G}_\Pi)$ and has a number of nodes linear in $|\mathrm{at}(\Pi)|$, in time $2^{\mathcal{O}(k)} \cdot |\mathrm{at}(\Pi)|$.

4.5 Discussion: Different Ways of Treating Hard Cycles

So far, we have seen several ways to utilize treewidth for normal and head-cycle-free (HCF) programs. Recall that Section 3.2.2 contains a quite direct method of treating these programs by means of dynamic programming, where the simple algorithm for computing supported models is extended by level mappings. The runtime of this algorithm is slightly superexponential, so the implementation of level mappings seems to cause a slight blowup in the runtime depending on the treewidth. Indeed, the treewidth dependence on the runtime increases from $2^{\mathcal{O}(k)}$ (for computing supported models) to $2^{\mathcal{O}(k \cdot \log(k))}$, where k refers to the treewidth of the primal graph of the given program, cf. also Table 1.1.

Compared to the direct implementation of the previous chapter, this chapter discusses a different approach. With the help of decomposition-guided reductions, one can implement the reduction of Section 4.3 in order to apply efficient algorithms for Boolean satisfiability (Sat). However, also this reduction comes with some overhead, where the treewidth is increased from k to $\mathcal{O}(k \cdot \log(k))$. Given that there is a known correspondence between resolution width, internally used in Sat solvers, and treewidth [Atserias et al., 2011], this treewidth blowup might be bearable for some use cases.

Still, in Section 4.4 we finally deal with an orthogonal approach that tries to overcome this blowup. To this end, we define a new measure, the so-called tightness width, which combines treewidth of the primal graph of a program with the largest size of any SCC of the dependency graph of the program. Interestingly, tight program always have tightness width 1 and non-tight programs are linked to a larger tightness width, but there are even programs with large cycles that still have a rather small tightness width. With this additional measure, we are able to solve normal and HCF programs via Sat solving, with a treewidth blowup from k to $\mathcal{O}(k \cdot \log(\iota))$, where k refers to the treewidth and ι indicates the tightness width. Even further, the reduction of Section 4.4 allows us to just increase

"tightness", where the tightness width is reduced at the cost of an increased treewidth. So, instead of reducing to a fully tight program that then has tightness width 1, one can also decide to just break the cycles of certain SCCs in order to find the golden mean, where the resulting program is almost tight and the treewidth is still reasonably small.

In the next chapter we show how to use decomposition-guided reductions in order to establish a methodology for proving lower bounds. This allows us to provide lower bound results similar to, e.g., Proposition 2.10 and Corollary 3.9, but for further formalisms and problems. There, we also see that it is quite unexpected that the reductions of this chapter or the algorithms of Chapter 3 can be significantly improved, unless the exponential time hypothesis fails.

CHAPTER 5

Lower Bounds by Decomposition-Guided Reductions

The only way to discover the limits of the possible is to go beyond them into the impossible.

— Arthur C. Clarke

Having already shown upper bounds in Chapters 3 and 4, the natural question, whether one can improve these results or establish certain lower bounds for problems parameterized by treewidth, arises. While there exist several works in this direction for specific problems, also for problems on higher levels of the polynomial hierarchy [Marx and Mitsou, 2016; Lokshtanov et al., 2011], little work has been done to provide some meta tool for establishing lower bounds. The need for this is even more evident for problems higher on the polynomial hierarchy, which are more likely to have a large dependence on the treewidth [Marx and Mitsou, 2016]. Recall that indeed, treewidth has been widely employed for applications such as Boolean satisfiability (Sat) [Samer and Szeider, 2010] and constraint satisfaction (CSP) [Dechter, 2006; Freuder, 1985], but also for problems believed beyond NP such as probabilistic inference [Ordyniak and Szeider, 2013] as well as problems in knowledge representation and reasoning [Gottlob et al., 2010; Pichler et al., 2010; Dvořák et al., 2012]. Also for the prominent problem QSat, which asks for deciding the validity of a quantified Boolean formula (QBF), there are tractability results using an additional parameter [Chen, 2004] or some extension of treewidth [Eiben et al., 2018, 2020].

The meta results on treewidth are the well-known *Courcelle's theorem* [Courcelle, 1990] and its logspace version [Elberfeld et al., 2010], which states that whenever one can encode a problem into a formula in monadic second order logic (MSO), then the problem can be decided in time linear in the input size and some function in the treewidth. While

Courcelle's theorem provides a full framework for classifying problems concerning the existence of a tractable algorithm, its practical application is limited due to potentially huge constants, and the exponential runtime in the treewidth (upper bound) may result in a tower of exponents that is far from optimal. In contrast, the available upper bounds are more immediate for QSat. Recall that QSat can be turned tractable using treewidth, if additionally we parameterize by the number of alternating quantifier blocks (quantifier rank) [Chen, 2004], as stated in Proposition 2.9. Since QSat is a quite natural and prominent, prototypical problem for descriptive complexity [Grohe, 2017; Immerman, 1999] and also due to results by Fagin [1974], we aim for establishing QSat as a prototypical problem for showing lower bounds for treewidth and in the context of parameterized complexity in general.

While precise lower bounds for QSat and treewidth have been left open since the dynamic programming algorithm serving as upper bound [Chen, 2004], several related work in this direction exists. Indeed, it has been proven that QSat remains intractable when parameterized by treewidth alone [Atserias and Oliva, 2014]. Recently, Lampis and Mitsou [2017] established that 2-QSat ($\exists\forall$-Sat and $\forall\exists$-Sat) cannot be solved by an algorithm that runs in time single exponential in the treewidth of the primal graph when assuming the *Exponential Time Hypothesis (ETH)* [Impagliazzo et al., 2001], cf. Proposition 2.11. In an earlier work [Pan and Vardi, 2006], it was mentioned that this extends to 3-QSat ($\forall\exists\forall$-Sat and $\exists\forall\exists$-Sat), and ℓ-QSat, if ℓ is an *odd* number. However, it does not extend constructively to the case, where ℓ is even. While Marx and Mitsou [2016] considered certain graph problems that are located on the third level of the polynomial hierarchy [Stockmeyer and Meyer, 1973], they emphasize that the classical complexity results do not provide sufficient explanation why double- or triple-exponential dependence on treewidth is needed and one requires quite involved proofs for each problem separately. They state that intuitively the quantifier rank of the problem definitions are the common underlying reason for being on higher levels of the polynomial hierarchy and for requiring high dependence on treewidth. Lampis, Mitsou, and Mengel [2018] employed the known runtime result [Chen, 2004] and proposed reductions from a collection of reasoning problems in AI to QSat that yield quite precise upper bounds on the runtime. In consequence, given the lower bound for 2-QSat these results highlight QBF encodings as a very handy and an alternative to Courcelle's theorem. We generalize this observation and also confirm that the usage of ℓ-QSat as the target of reductions might be suited in order to obtain precise runtime upper bounds.

Lower Bounds for QSat. More concretely, in Section 5.1, which is based on recent work [Fichte et al., 2020c], we first establish results for QBFs of bounded treewidth and of *arbitrary*, bounded quantifier rank, thereby providing a novel method to generalize the result for 2-QSat in a non-incremental way.

Claim 5.1. *Under the ETH,* QSat *for a closed formula Q in prenex normal form with n variables, primal treewidth k, and quantifier rank $\ell \geq 1$* cannot *be decided in time* $\mathsf{tower}(\ell, o(k)) \cdot \mathrm{poly}(n)$.

We prove Claim 5.1, which complements Proposition 2.9 and strengthens the importance of QBF encodings for problems parameterized by treewidth. Thereby, we establish the full picture of runtime lower bounds for QSat parameterized by treewidth in connection to the quantifier rank of the formula. This is possible due to our main idea of *exponentially decreasing* the dependency on treewidth at the cost of a slight increase in the quantifier rank, which shows the dependence on the treewidth for QSat and leads to concrete lower bounds when assuming the ETH. Intuitively, we present a decomposition-guided reduction, as introduced in Chapter 4, that significantly *compresses* treewidth and applies to any instance of QSat without restricting the quantifier rank while only assuming the ETH. Note that this "compression" is constructive and independent of the original instance size, which is different from existing methods, e.g., [Lampis and Mitsou, 2017; Marx and Mitsou, 2016; Pan and Vardi, 2006].

Later, Chapter 6 will catch up on the proof of Claim 5.1 in order to provide a description of a general methodology for establishing lower bounds for problems parameterized by treewidth. In more detail, equipped with our newly established lower bound results for QSat, there we propose reductions from QSat as a general toolkit for proving lower bounds assuming the ETH and discuss several showcases to illustrate this methodology.

Lower Bounds for Asp. Then, in Section 5.2 we apply the result of Claim 5.1 in order to show lower bounds for various fragments of answer set programming (ASP), which serves the purpose of a prototypical problem. Our thereby obtained results show that the upper bounds as established in Chapter 3 and 4 probably cannot be significantly improved, when assuming the ETH. In particular, we show that for Disjunctive Asp we do not expect a runtime that is better than double exponential in the treewidth of the primal graph representation. Even more interesting, however, is the fragment Normal Asp. To this end, recall the runtime for solving Normal Asp, cf. Sections 3.2.2, being slightly superexponential in the treewidth. Analogously, we presented in Section 4.3 a reduction from Normal Asp to Boolean satisfiability (Sat), where the treewidth increase is sub-quadratic (slightly superlinear).

Based on recent work [Hecher, 2020], we show in Section 5.2.1 that for Normal Asp, we certainly cannot avoid a slightly superexponential runtime, which also rules out reductions to Sat with a treewidth increase that is asymptotically better than our sub-quadratic increase. Concretely, we establish that under the ETH, one cannot decide Normal Asp in time $2^{o(k \cdot \log(k))} \cdot n$, with treewidth k and program size n. This is in contrast to the runtime for deciding Sat: $2^{\mathcal{O}(k)} \cdot n$ with treewidth k and size n of the formula. As a result, this establishes that deciding the consistency of normal programs is already harder than Sat using treewidth. Note that this is surprising as both problems are of similar hardness according to classical complexity (NP-complete). While existing results [Lifschitz and Razborov, 2006] and the expressive power hierarchy [Janhunen, 2006] weakly indicate that Normal Asp might be slightly harder than Sat, these existing works mostly deal with bijectively preserving all answer sets. However, compared to these existing works, our result relies on the ETH, but is not restricted to, e.g., modular reductions [Janhunen,

Existing works	Our approach	Work for 2-QSat	**Our QSat approach**
		[Lampis and Mitsou, 2017]	
(Sat, k)	(Sat, k)	(Sat, k)	(Sat, k)
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow_R$
$(\mathtt{P}, f(n))$	$(\mathtt{P}, g(k))$	$(2\text{-QSat}, \log(n))$	$(2\text{-QSat}, \log(k))$
		(Sat, k)	$(\ell\text{-QSat}, k)$
		$\downarrow_?$	$\downarrow_R$
		$(3\text{-QSat}, \log(\log(n))$	$(\ell+1\text{-QSat}, \log(k))$

Table 5.1: Comparing existing works of the literature, e.g., [Lokshtanov et al., 2011; Marx and Mitsou, 2016; Lampis and Mitsou, 2017], (first column) to our approach (second column) for showing lower bounds for a problem $\mathtt{P}$. Table entries are of the form $(\mathtt{P}, k)$, which refers to problem $\mathtt{P}$ when parameterized by treewidth k. Existing works depicted in the first column mainly reduce from an arbitrary Sat instance F with n variables and treewidth k of $\mathcal{G}_F$ and construct an instance $\mathcal{I}$ of $\mathtt{P}$ with treewidth $f(n)$ of $\mathcal{G}_\mathcal{I}$. Note that therefore treewidth $f(n)$ does not directly relate to the original treewidth k, which is in contrast to our approach (second column). However, our approach has advantages, which becomes obvious when comparing it to the work [Lampis and Mitsou, 2017] for 2-QSat (third column). Since there is no direct relation between k and $f(n)$, it is rather hard to reuse ideas from [Lampis and Mitsou, 2017] for constructing, e.g., a reduction from Sat to 3-QSat, i.e., one basically requires new ideas from scratch. The fourth column, which depicts our approach, is different in this regard and allows us to reuse the same idea as in the reduction from Sat to 2-QSat for designing a reduction from ℓ-QSat to $(\ell+1)$-QSat.

2006], or involves the need of auxiliary variables [Lifschitz and Razborov, 2006], and already holds for the consistency problem Normal Asp. Indeed, our result might have theoretical consequences also for Asp solvers, as these solvers [Gebser et al., 2012; Alviano et al., 2019a] are heavily based on Sat solvers and there is a close connection [Atserias et al., 2011] between resolution-width and treewidth, resulting in efficient Sat solver runs on instances of small treewidth.

5.1 Lower Bounds for QBFs and Treewidth via Decoupling Dependencies

Next, we prepare a new approach in order to establish the lower bound of Claim 5.1. Our approach relies on the idea of transforming a given input QBF of treewidth k into an instance of exponentially smaller treewidth compared to k. Thereby, we trade the exponential decrease of the parameter for the cost of additional computation power required to solve the reduced instance. For the canonical QSat problem, this additional computation power results in an increase of the quantifier rank. Thereby, we constructively

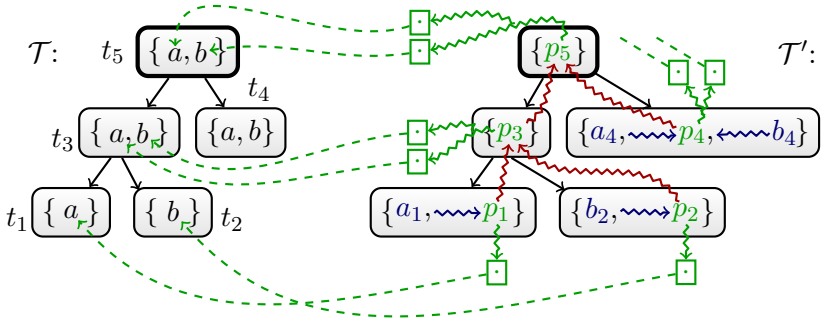

Figure 5.1: Simplified illustration of a certain tree decomposition $\mathcal{T}' = (T, \chi')$ of $\mathcal{G}_{R(Q,\mathcal{T})}$ (yielded by reduction R), and its relation to tree decomposition $\mathcal{T} = (T, \chi)$ of $\mathcal{G}_Q$. Each bag $\chi'(t_i)$ of a node t_i of $\mathcal{T}'$ contains variable x_i for any variable x introduced in $\chi(t_i)$ and $\lceil \log(\mathsf{width}(\mathcal{T})) \rceil$ many (green) pointer variables p_i selecting *one* variable in $\chi(t_i)$ of $\mathcal{T}$. Squiggly red arrows indicate the propagation between pointers p_i, p_j and ensure consistency. In particular, although truth values for variable a are "guessed" using a_1 and a_4 (and "propagated" via blue squiggly arrows to corresponding pointers p_1 and p_4, respectively), these red arrows ensure via pointers p_1, p_3, p_4, p_5 that truth values for a_1 and a_4 coincide.

encode core ideas of a dynamic programming algorithm on a tree decomposition into a QBF that expresses solving an instance of QSAT by means of a QSAT oracle of *one level* higher in the hierarchy (self-reduction) while achieving a certain compression of treewidth. More precisely, we provide a decomposition-guided reduction (cf. Chapter 4) that reduces any instance Q of QSAT of treewidth k and quantifier rank ℓ to an instance Q' of QSAT of treewidth $\mathcal{O}(\log k)$ and quantifier rank $\ell + 1$, while the size of Q' is linearly bounded in the size of Q. Notice that the treewidth of Q' only depends on the treewidth of Q, but is independent of, e.g., the number of variables and quantifier rank of Q. Hence, we say that the *treewidth* of Q' is *compressed* compared to the original treewidth of Q.

This approach, as given in the second column of Table 5.1, is indeed different from existing works (first column of the table). In existing works, we observed that when reducing from SAT to a target problem, the treewidth of the source formula does not necessarily relate to the treewidth of the constructed instance of the target problem. Instead, often the treewidth of the reduced instance depends on number n of variables of the source Boolean formula. The lower bound result for 2-QSAT [Lampis and Mitsou, 2017], given in the third column of Table 5.1, reduces from SAT to 2-QSAT, where for any instance of SAT with n variables, the resulting 2-QSAT instance has treewidth $\mathcal{O}(\log(n))$. It is, however, rather hard to generalize this reduction [Lampis and Mitsou, 2017] to reduce from SAT to 3-QSAT or even to ℓ-QSAT, as the treewidth would have to be in $\mathcal{O}(\log(\log(n)))$ or even be ℓ-fold logarithmic in n, respectively. Our approach of exponentially decreasing treewidth for QSAT, as depicted in the fourth column of Table 5.1, can be easily generalized. Thereby,

we instantly obtain a reduction R from ℓ-QSat to $(\ell+1)$-QSat, where the target QBF with quantifier rank $\ell+1$ has an exponentially smaller treewidth compared to the source QBF of quantifier rank ℓ. This is achieved with the help of decomposition-guided reductions, since actually R is such a reduction as introduced in Chapter 4.

Next, we introduce our decomposition-guided reduction R that takes both an instance Q of ℓ-QSat and a corresponding tree decomposition $\mathcal{T}$ of the primal graph $\mathcal{G}_Q$. The reduction returns a compressed instance $Q' = R(Q, \mathcal{T})$ of $(\ell+1)$-QSat such that the width of $\mathcal{G}_{R(Q,\mathcal{T})}$ is in $\mathcal{O}(\log(\mathsf{width}(\mathcal{T})))$. The reduction R, which is guided by TD $\mathcal{T}$, yields a new *compressed tree decomposition* $\mathcal{T}'$ of $\mathcal{G}_{R(Q,\mathcal{T})}$ of width $\mathcal{O}(\log(\mathsf{width}(\mathcal{T})))$. For that it is crucial to balance introducing copies of variables (redundancy) and saving treewidth (structural dependency), such that, intuitively, we can still evaluate $R(Q, \mathcal{T})$ given the limitation of treewidth $\mathcal{O}(\log(\mathsf{width}(\mathcal{T})))$. To keep this balance, we can only analyze in a bag in $\mathcal{T}'$ a *constant* number of elements of the corresponding original bag of $\mathcal{T}$. Still, considering $\log(\mathsf{width}(\mathcal{T}))$ many elements in a bag at once allows us to represent one "pointer" to address at most $\mathsf{width}(\mathcal{T})$ many elements of each bag of $\mathcal{T}$ and, consequently, the restriction to $\mathcal{O}(\log(\mathsf{width}(\mathcal{T})))$ many elements in a bag at once enables *constantly* many such pointers. To give a first glance at the idea of the reduction R, Figure 5.1 provides an intuition and illustrates a tree decomposition $\mathcal{T}$ of $\mathcal{G}_Q$ together with a corresponding compressed tree decomposition $\mathcal{T}'$ of $\mathcal{G}_{R(Q,\mathcal{T})}$, whose bags contain pointers to original bags of $\mathcal{T}$. Actually, we can encode the *propagation* of information from one bag of $\mathcal{T}'$ to its parent bag with the help of these pointers. Thereby, we ensure that information is consistent and this consistency can be preserved, even though we *guess* in R truth values for copies of the same variable in Q independently. Note that these "local" pointers for each bag are essential to achieve treewidth compression.

Below, we discuss the decomposition-guided reduction R in more detail, which enables the use of QSat as the source of reductions for showing lower bounds under the ETH.

5.1.1 A Decomposition-Guided Reduction for Reducing Treewidth Dependencies

The formula $R(Q, \mathcal{T})$ constructed by R mainly consists of three interacting parts. In the presentation, we refer to them as *guess*, *check*, and *propagate* part.

- "Guess" ($R_{\mathcal{G}}$): Contains clauses responsible for guessing truth values of variables occurring in the original QBF Q.

- "Check" ($R_{\mathcal{CK}}$): These clauses ensure that there is at least one 3-DNF term in Q that is satisfied, thereby maintaining 3 pointers for each node as discussed above.

- "Propagate" ($R_{\mathcal{P}}$): These clauses ensure consistency using a pointer for each node of the tree decomposition.

We commence with the formal description of R. Given a QBF Q of the form $Q := Q_1 V_1 . Q_2 V_2 . \cdots \forall V_\ell . D$, where D is in 3-DNF such that the quantifier blocks are alternat-

ing, i.e., quantifiers of quantifier blocks with even indices are equal, which are different from those of blocks with odd indices. Further, assume an atomic, c-nice labeled tree decomposition $\mathcal{T} = (T, \chi, \delta_Q)$, where $T = (N, \cdot)$, of Q, which can be computed in polynomial time from a given tree decomposition by Proposition 2.8 and Theorem 2.19. Notice that by Definition 2.17 and the definition of atomic LTDs, for all terms $d \in D$, the inverse function $\delta_Q^{-1}(d)$ is well-defined. Further, actually R can deal with open QBFs, i.e., QBF Q does not necessarily have to be closed. Open formulas are needed later to simplify the correctness proof of Section 5.1.2.

We use the following definitions. Let $\mathit{NodeI}(x) := \{t \mid t \in N, x \in \chi(t) \setminus (\bigcup_{t_i \in \mathsf{children}(t)} \chi(t_i))\}$ be the set of nodes, where a given element x is *introduced*. For a set $V \subseteq \mathsf{var}(D)$ of variables, we denote by $\mathit{VarI}(V) := \{x_t \mid x \in V, t \in \mathit{NodeI}(x)\}$ the set of fresh variables generated for each original variable x and node t, where x is introduced. Later, we need to distinguish whether the set V_i of variables is universally or existentially quantified. Universal quantification requires to shift for each $x \in V_i$ all but one representative of $\{x_t \mid t \in \mathit{NodeI}(x)\}$ to the next existential quantifier block Q_{i+1}. The representative variable that is not shifted is denoted by $\mathsf{rep}(x)$. In particular, given a quantifier block Q_2, its variables V_2 and the variables V_1 of the preceding quantifier block, we define: $\mathit{VarI}(Q_2, V_2, V_1) := \{x_t \mid x \in V_2, t \in \mathit{NodeI}(x), Q_2 = \exists\} \cup \{x_t \mid x \in V_2, t \in \mathit{NodeI}(x), x_t = \mathsf{rep}(x), Q_2 = \forall\} \cup \{x_t \mid x \in V_1, t \in \mathit{NodeI}(x), x_t \neq \mathsf{rep}(x), Q_2 = \exists\}$. We denote by $\mathit{VarSat} := \{\mathit{sat}_t, \mathit{sat}_{\leq t} \mid t \in N\}$ the set of fresh decision variables responsible for storing for each node $t \in N$ whether any term at t or at any node below t is satisfied, respectively. Finally, we denote by $\mathit{VarB} := \{b_t^0, \ldots, b_t^{\lceil \log(|\chi(t)|) \rceil - 1} \mid t \in N\}$, and $\mathit{VarBV} := \{v_t \mid t \in N\}$ the set of fresh variables for each node $t \in N$ that will be used to address particular elements of the corresponding bags (pointer as depicted in Figure 5.1 in binary representation), and to assign truth values for these elements, respectively. Overall, the variables in VarB allow us to guide the evaluation of formula D along the tree decomposition $\mathcal{T}$. For checking 3-DNF terms, we need the same functionality three more times, resulting in the sets $\mathit{VarB3} := \{b_{t,j}^0, \ldots, b_{t,j}^{\lceil \log(|\chi(t)|+1) \rceil - 1} \mid t \in N, 1 \leq j \leq 3\}$ that additionally may refer to a special fresh element nil (therefore the $+1$ in the exponent in definition of $\mathit{VarB3}$), and $\mathit{VarBV3} := \{v_{t,j} \mid t \in N, 1 \leq j \leq 3\}$ of fresh variables. Notice that the construction is designed in such a way that the focus lies only on certain elements of the bag (one at a time, and independent of other elements within the same bag). In the end, this ensures that the treewidth of our reduced instance is only logarithmic in the original treewidth of the primal graph of D. Reduction $R(Q, \mathcal{T})$ creates $Q' :=$

$$Q_1\ \mathit{VarI}(Q_1, V_1, \emptyset).\ Q_2\ \mathit{VarI}(Q_2, V_2, V_1). \cdots \forall\ \mathit{VarI}(\forall, V_\ell, V_{\ell-1}),\ \mathit{VarB}.$$
$$\exists\ \mathit{VarI}(\exists, \emptyset, V_\ell),\ \mathit{VarBV}, \mathit{VarBV3}, \mathit{VarB3}, \mathit{VarSat}.\ C,$$

where C is a CNF formula consisting of guess, check and propagate parts, i.e., sets $R_\mathcal{G}$, $R_{\mathcal{CK}}$, and $R_\mathcal{P}$ of clauses, respectively.

Example 5.2. *Consider again the QBF $Q = \exists w, x. \forall y, z. D$ from Example 2.2, where $D := d_1 \vee d_2 \vee d_3 \vee d_4$, and $d_1 := w \wedge x \wedge \neg y$, $d_2 := \neg w \wedge \neg x \wedge y$, $d_3 := w \wedge y \wedge \neg z$, and $d_4 := w \wedge y \wedge z$. Further, recall the labeled tree decomposition $\mathcal{T}_2$ of Q of Figure 2.3*

(right). The resulting instance $R(Q, \mathcal{T}_2)$ looks as follows assuming that $\mathsf{rep}(y) = y_{t_1}$*, where C consists of a guess, check and, propagate part.*

$$\exists \underbrace{w_{t_1}, w_{t_3}, x_{t_1}}_{VarI(\exists,\{w,x\},\emptyset)}. \forall \underbrace{y_{t_1}, z_{t_3},}_{VarI(\forall,\{y,z\},\{w,x\})} \underbrace{b^0_{t_1}, b^1_{t_1}, b^0_{t_2}, \ldots, b^1_{t_4}, b^0_{t_5}}_{VarB}. \exists \underbrace{y_{t_3},}_{VarI(\exists,\emptyset,\{y,z\})}$$

$$\underbrace{v_{t_1}, \ldots, v_{t_5}}_{VarBV}, \underbrace{v_{t_1,1}, v_{t_1,2}, v_{t_1,3}, v_{t_2,1}, \ldots, v_{t_5,3},}_{VarBV3}$$

$$\underbrace{b^0_{t_1,1}, b^1_{t_1,1}, b^0_{t_1,2}, \ldots, b^1_{t_5,3},}_{VarB3} \underbrace{sat_{t_1}, \ldots, sat_{t_5}, sat_{\leq t_1}, \ldots, sat_{\leq t_5}.}_{VarSat} C$$

In the following, we define sets $R_{\mathcal{G}}$, $R_{\mathcal{CK}}$, and $R_{\mathcal{P}}$ of clauses. To this end, we require for the pointers a bit-vector (binary) representation of the elements in a bag of $\mathcal{T}$, and a mapping that assigns bag elements to its corresponding binary representation. In particular, we assume an arbitrary, but fixed total order $\prec$ of elements of a bag $\chi(t)$ of any given node $t \in N$. With $\prec$, we can then assign each element x in $\chi(t)$ its unique (within the bag) induced ordinal number $o(t, x)$. This ordinal number $o(t, x)$ is expressed in binary. For that we need precisely $\lceil \log(|\chi(t)|) \rceil$ many bit-variables $B := \{b^0_t, \ldots, b^{\lceil \log(|\chi(t)|) \rceil - 1}_t\}$. We denote by $[\![x]\!]_t$ the (consistent) set of literals over variables in B that encode (in binary) the ordinal number $o(t, x)$ of $x \in \chi(t)$ in t, such that whenever a literal b^i_t or $\neg b^i_t$ is contained in the set $[\![x]\!]_t$, the i-th bit in the unique binary representation of $o(t, x)$ is 1 or 0, respectively. Analogously, for $1 \leq j \leq 3$ we denote by $[\![x]\!]_{t,j}$ the (consistent) set of literals over variables in $B_j := \{b^0_{t,j}, \ldots, b^{\lceil \log(|\chi(t)|+1) \rceil - 1}_{t,j}\}$ that either binary-encode the ordinal number $o(t, x)$ of $x \in \chi(t)$ in t, or these literals binary-encode number $\max_{y \in \chi(t)} o(t, y) + 1$ for $x = nil$.

The guess part $\mathcal{G}$. The clauses in $R_{\mathcal{G}}$, which we denote as implications, are defined as follows.

$$x_t \wedge \bigwedge_{b \in [\![x]\!]_t} b \longrightarrow v_t \qquad \text{for each } x_t \in VarI(\mathsf{var}(D)) \tag{5.1}$$

$$\neg x_t \wedge \bigwedge_{b \in [\![x]\!]_t} b \longrightarrow \neg v_t \qquad \text{for each } x_t \in VarI(\mathsf{var}(D)) \tag{5.2}$$

Intuitively, this establishes that whenever a certain variable x_t for an introduced variable $x \in \chi(t)$ is assigned to true (false) and all the corresponding literals in $[\![x]\!]_t$ of the binary representation of $o(t, x)$ are satisfied (i.e., x is "selected"), then also $v_t \in VarBV$ of node t has to be set to true (false).

Analogously, set $R_{\mathcal{G}}$ further contains the following clauses:

$$x_t \wedge \bigwedge_{b \in [\![x]\!]_{t,j}} b \longrightarrow v_{t,j} \qquad \text{for each } x_t \in VarI(\mathsf{var}(D)), 1 \leq j \leq 3 \tag{5.3}$$

$$\neg x_t \wedge \bigwedge_{b \in [\![x]\!]_{t,j}} b \longrightarrow \neg v_{t,j} \qquad \text{for each } x_t \in VarI(\mathsf{var}(D)), 1 \leq j \leq 3 \tag{5.4}$$

Example 5.3. *Consider formula C from Example 5.2. Let $1 \leq j \leq 3$. Further, assume the following mapping of bag contents to bit-vector assignments. For any variable $a \in \mathsf{var}(D)$ with $t \in \mathit{NodeI}(a)$ and for $a = \mathit{nil}$ with $t \in N$, we arbitrarily fix the total ordering $\prec$ and have $[\![a]\!]_t$ and $[\![a]\!]_{t,j}$ as follows.*

a	$t \in \{t_1, t_2\}$		$t \in \{t_3, t_4\}$		$t = t_5$	
	$[\![a]\!]_t$	$[\![a]\!]_{t,j}$	$[\![a]\!]_t$	$[\![a]\!]_{t,j}$	$[\![a]\!]_t$	$[\![a]\!]_{t,j}$
w	$\{\neg b_t^0, \neg b_t^1\}$	$\{\neg b_{t,j}^0, \neg b_{t,j}^1\}$	$\{\neg b_t^0, \neg b_t^1\}$	$\{\neg b_{t,j}^0, \neg b_{t,j}^1\}$	$\{\neg b_t^0\}$	$\{\neg b_{t,j}^0, \neg b_{t,j}^1\}$
x	$\{\neg b_t^0, b_t^1\}$	$\{\neg b_{t,j}^0, b_{t,j}^1\}$	-		-	
y	$\{b_t^0, \neg b_t^1\}$	$\{b_{t,j}^0, \neg b_{t,j}^1\}$	$\{\neg b_t^0, b_t^1\}$	$\{\neg b_{t,j}^0, b_{t,j}^1\}$	$\{b_t^0\}$	$\{\neg b_{t,j}^0, b_{t,j}^1\}$
z	-		$\{b_t^0, \neg b_t^1\}$	$\{b_{t,j}^0, \neg b_{t,j}^1\}$	-	
nil	-	$\{b_{t,j}^0, b_{t,j}^1\}$	-	$\{b_{t,j}^0, b_{t,j}^1\}$	-	$\{b_{t,j}^0, \neg b_{t,j}^1\}$

The guess part of C contains for example for variable $w \in \mathsf{var}(D)$ the following clauses.

$$w_{t_1} \wedge \neg b_{t_1}^0 \wedge \neg b_{t_1}^1 \longrightarrow v_{t_1}, \quad \neg w_{t_1} \wedge \neg b_{t_1}^0 \wedge \neg b_{t_1}^1 \longrightarrow \neg v_{t_1},$$

$$w_{t_3} \wedge \neg b_{t_3}^0 \wedge \neg b_{t_3}^1 \longrightarrow v_{t_3}, \quad \neg w_{t_3} \wedge \neg b_{t_3}^0 \wedge \neg b_{t_3}^1 \longrightarrow \neg v_{t_3}.$$

Thereby, whenever we guess a certain truth value for w_{t_1} (w_{t_3}) it is ensured that there is a certain bit-vector, namely $[\![w]\!]_{t_1}$ ($[\![w]\!]_{t_3}$) such that v_{t_1} (v_{t_3}) has to be set to the same truth value. Analogously, clauses of the form (5.3) and (5.4) are in $R_{\mathcal{G}}$.

The check part $R_{\mathcal{CK}}$. In the following, we assume an arbitrary, but fixed total order of the (three) literals of each (3-DNF) term $d \in D$. We refer to the first, second, and third literal of d by $\mathsf{tlit}(d,1), \mathsf{tlit}(d,2)$, and $\mathsf{tlit}(d,3)$, respectively. Analogously, $\mathsf{tvar}(d,1), \mathsf{tvar}(d,2)$, and $\mathsf{tvar}(d,3)$ refers to the variable of the first, second, and third literal, respectively. Further, for a given term $d \in D$ and $1 \leq j \leq 3$, let $\mathsf{bv}(d,t,j)$ denote $v_{t,j}$ if $\mathsf{tlit}(d,j)$ is a variable, and $\neg v_{t,j}$ otherwise. Set $R_{\mathcal{CK}}$ contains the following clauses:

$$\mathit{sat}_{\leq t} \longrightarrow \mathit{sat}_{\leq t_1} \vee \cdots \vee \mathit{sat}_{\leq t_s} \vee \mathit{sat}_t \qquad \begin{array}{l}\text{for each } t \in N,\\ \mathsf{children}(t) = \{t_1, \ldots, t_s\}\end{array} \tag{5.5}$$

Informally speaking, for any node t this ensures the propagation of whether we satisfied at least one term directly in node t, or in any descendant of t.

In order to check whether a particular term is satisfied, we add for each term $d \in D$ clauses encoding the implication $\mathit{sat}_{\delta_Q^{-1}(d)} \longrightarrow \bigwedge_{1 \leq j \leq 3} \left[\bigwedge_{b \in [\![\mathsf{tvar}(d,j)]\!]_{\delta_Q^{-1}(d),j}} b \wedge \mathsf{bv}(d, \delta_Q^{-1}(d), j) \right]$ as follows:

$$\mathit{sat}_{\delta_Q^{-1}(d)} \longrightarrow b \qquad \text{for each } d \in D, 1 \leq j \leq 3,$$

$$b \in [\![\mathsf{tvar}(d,j)]\!]_{\delta_Q^{-1}(d),j} \tag{5.6}$$

$$sat_{\delta_Q^{-1}(d)} \longrightarrow \mathsf{bv}(d, \delta_Q^{-1}(d), j) \qquad \text{for each } d \in D, 1 \leq j \leq 3 \tag{5.7}$$

Finally, we add $sat_{\leq r}$ for root $r = \mathrm{root}(T)$, and $\neg sat_t$ for each node t in $N \setminus \bigcup_{d \in D} \{\delta_Q^{-1}(d)\}$ since these nodes are not used for checking satisfiability of any term.

$$sat_{\leq r} \tag{5.8}$$

$$\neg sat_t \qquad \text{for each } t \in N \setminus \bigcup_{d \in D} \{\delta_Q^{-1}(d)\} \tag{5.9}$$

Example 5.4. *Consider again formula C from Example 5.2. We discuss clauses of the check part for node $t_2 = \delta_Q^{-1}(d_2)$ and root node t_5. Thereby, we encode satisfiability of term $d_2 = \neg w \wedge \neg x \wedge y$ assuming $\mathsf{tlit}(d_2, 1) = \neg w, \mathsf{tlit}(d_2, 2) = \neg x$, and $\mathsf{tlit}(d_2, 3) = y$.*

$$sat_{\leq t_2} \longrightarrow sat_{\leq t_1} \vee sat_{t_2},$$

$$sat_{t_2} \longrightarrow \neg b^0_{t_2,1}, \quad sat_{t_2} \longrightarrow \neg b^1_{t_2,1}, \quad sat_{t_2} \longrightarrow \neg v_{t_2,1},$$

$$sat_{t_2} \longrightarrow \neg b^0_{t_2,2}, \quad sat_{t_2} \longrightarrow b^1_{t_2,2}, \quad sat_{t_2} \longrightarrow \neg v_{t_2,2},$$

$$sat_{t_2} \longrightarrow b^0_{t_2,3}, \quad sat_{t_2} \longrightarrow \neg b^1_{t_2,3}, \quad sat_{t_2} \longrightarrow v_{t_2,3},$$

$$sat_{\leq t_5} \longrightarrow sat_{\leq t_2} \vee sat_{\leq t_4} \vee sat_{t_5}, \quad sat_{\leq t_5}, \quad \neg sat_{t_5}$$

The propagate part $R_{\mathcal{P}}$. The sets $R_{\mathcal{G}}$ and $R_{\mathcal{CK}}$ contain clauses responsible for guessing truth values and checking that at least one term of the original formula D is satisfied accordingly. In particular, the guess of truth values for $\mathsf{var}(D)$ happens at different tree decomposition nodes "independently", whereas checking whether at least one term $d \in D$ is satisfied is achieved in exactly one tree decomposition node $\delta_Q^{-1}(d)$. Intuitively, in order to ensure that these independent guesses of truth values for $\mathsf{var}(D)$, are consistent, clauses in $R_{\mathcal{P}}$ make use of the connectedness condition of TDs in order to guide the comparison of these independent guesses along the TD. More precisely, for each tree decomposition node $t \in N$, every node $t_i \in \mathsf{children}(t)$, and every variable $x \in \chi(t) \cap \chi(t_i)$ that both nodes t and t_i have in common, the set $R_{\mathcal{P}}$ contains clauses:

$$v_t \wedge \bigwedge_{b \in [\![x]\!]_t} b \wedge \bigwedge_{b \in [\![x]\!]_{t_i}} b \longrightarrow v_{t_i} \qquad \begin{array}{l}\text{for each } t \in N, t_i \in \mathsf{children}(t),\\ x \in \chi(t) \cap \chi(t_i)\end{array} \tag{5.10}$$

$$\neg v_t \wedge \bigwedge_{b \in [\![x]\!]_t} b \wedge \bigwedge_{b \in [\![x]\!]_{t_i}} b \longrightarrow \neg v_{t_i} \qquad \begin{array}{l}\text{for each } t \in N, t_i \in \mathsf{children}(t),\\ x \in \chi(t) \cap \chi(t_i)\end{array} \tag{5.11}$$

Further, for each clause $d \in D$, every node t_i in $\mathsf{children}(\delta_Q^{-1}(d))$, and $1 \leq j \leq 3$ such that $\mathsf{tvar}(d, j) \in \chi(t_i)$, set $\mathcal{P}$ contains:

$$\bigwedge_{b' \in [\![\mathsf{tvar}(d,j)]\!]_{t,j}} b' \longrightarrow b \qquad \text{for each } d \in D \text{ with } 1 \leq j \leq 3,$$

$$t = \delta_Q^{-1}(d), t_i \in \mathsf{children}(t), \mathsf{tvar}(d,j) \in \chi(t_i),$$
$$b \in [\![\mathsf{tvar}(d,j)]\!]_{t_i,j} \tag{5.12}$$
$$v_{t,j} \longleftrightarrow v_{t_i,j} \qquad \text{for each } t \in N, t_i \in \mathsf{children}(t), 1 \leq j \leq 3 \tag{5.13}$$

Vaguely speaking, this construction ensures that whenever a bag element (using *VarB3*) or a truth value (using *VarBV3*) is "selected" in node t, we also have to select the same (if exists) below in children of t.

Example 5.5. *Consider once more C from Example 5.2. We illustrate the propagate part for node $t_4 = \delta_Q^{-1}(d_4)$ and variable w assuming that $w = \mathsf{tvar}(d_4, 1)$. Observe that $d_4 = w \wedge y \wedge z$, and $w \in \chi(t_4) \cap \chi(t_3)$.*

$$v_{t_4} \wedge \underbrace{\neg b^0_{t_4} \wedge \neg b^1_{t_4}}_{[\![w]\!]_{t_4}} \wedge \underbrace{\neg b^0_{t_3} \wedge \neg b^1_{t_3}}_{[\![w]\!]_{t_3}} \longrightarrow v_{t_3},$$
$$\neg v_{t_4} \wedge \underbrace{\neg b^0_{t_4} \wedge \neg b^1_{t_4}}_{[\![w]\!]_{t_4}} \wedge \underbrace{\neg b^0_{t_3} \wedge \neg b^1_{t_3}}_{[\![w]\!]_{t_3}} \longrightarrow \neg v_{t_3},$$
$$\neg b^0_{t_3,1} \wedge \neg b^1_{t_3,1} \longrightarrow \neg b^0_{t_4,1}, \quad \neg b^0_{t_3,1} \wedge \neg b^1_{t_3,1} \longrightarrow \neg b^1_{t_4,1},$$
$$v_{t_4,1} \longleftrightarrow v_{t_3,1}, \quad v_{t_4,2} \longleftrightarrow v_{t_3,2}, \quad v_{t_4,3} \longleftrightarrow v_{t_3,3}$$

Remark 5.6. *Recalling Figure 5.1, we would like to highlight the relation between elements of the figure and variables or clauses of reduction R introduced above. Blue elements a_1, b_2, a_4, b_4 represent "introduce variables" $VarI(\mathsf{var}(D))$ and the blue squiggly arrows visualize the guess part $R_{\mathcal{G}}$. Green elements p_1, p_2, p_3, p_4, p_5 represent "pointer variables" VarB and VarB3 and the green squiggly arrows point to elements of tree decomposition $\mathcal{T}$. Finally, red squiggly arrows visualize the propagate part $R_{\mathcal{P}}$. (The check part $R_{\mathcal{CK}}$ is not explicitly visualized.)*

Converting C to 3-CNF formula C'. Observe that one can transform the CNF formula C of the QBF $R(Q, \mathcal{T})$ into 3-CNF. This results in an reduction R' that takes QBF $R(Q, \mathcal{T})$ and transforms the CNF formula C of the QBF $R(Q, \mathcal{T})$ into 3-CNF, resulting in $Q'' = R'(R(Q, \mathcal{T}))$ such that $\mathsf{tw}(\mathcal{G}_{Q''}) \leq \mathsf{tw}(\mathcal{G}_{R(Q,\mathcal{T})}) + 2$. To this end, one has to perform the following standard reduction (cf. [Lampis and Mitsou, 2017] and [Samer and Szeider, 2010, Lemma 4]): As long as there exists a clause $c \in C$ consisting of more than 3 literals, we introduce a fresh existentially quantified variable v, remove c from C and replace it with two new clauses. The first new clause contains v and two literals of c, while the second clause contains $\neg v$ and the remaining literals of c. Note that this standard reduction R' does not affect satisfiability, and it can be done such that it causes only constant increase of the treewidth (cf. Lemma 5.11 and [Lampis and Mitsou, 2017]). Observe that by construction, the same argument actually holds for pathwidth.

5.1.2 Correctness, Compression, and Runtime

The reduction discussed above allows us to confirm Claim 5.1 and establish our main result, which is the following theorem that will be shown in the course of this section.

Theorem 5.7 (QBF lower bound)**.** *Given any QBF of the form* $Q = Q_1V_1.Q_2V_2.Q_3V_3 \cdots Q_\ell V_\ell.F$ *where* $\ell \geq 1$*, and* F *is a 3-CNF formula (if* $Q_\ell = \exists$*), or* F *is a 3-DNF formula (if* $Q_\ell = \forall$*). Then, unless the ETH fails,* Q *cannot be solved in time* $\mathsf{tower}(\ell, o(k)) \cdot \mathrm{poly}(|\mathsf{var}(F)|)$*, where* k *is the treewidth of the primal graph* $\mathcal{G}_Q$*.*

In the following, we show correctness and properties of our reduction presented in Section 5.1. Therefore, we assume a given QBF $Q := Q_1V_1.Q_2V_2.\cdots\forall V_\ell.D$, where D is in 3-DNF. Further, let $\mathcal{T} = (T, \chi, \delta_Q)$ such that $T = (N, \cdot)$ be an atomic, c-nice labeled labeled tree decomposition of Q of width k. The reduced instance is addressed by $R(Q, \mathcal{T})$, where reduction R is defined as in Section 5.1. The resulting QBF of quantifier rank $\ell + 1$ is referred to by $R'(R(Q, \mathcal{T}))$ and its matrix in 3-CNF is given by $C = \mathsf{matrix}(R'(R(Q, \mathcal{T})))$.

To simplify presentation, we introduce the following definitions. Let $d \in D$ be a term, $t \in N$ be a node of the tree decomposition, and $1 \leq j \leq 3$, then $\mathsf{bit\text{-}term}(d, t) := \bigcup_{1\leq j\leq 3}[[\![\mathsf{tvar}(d, j)]\!]_{t,j} \cup \{\mathsf{bv}(d, t, j)\}]$. Further, given an assignment $\alpha : \mathsf{var}(D) \to \{0, 1\}$, we define a function $\mathsf{local}(\cdot)$ that produces new assignments to copies of the variables, therefore let $\mathsf{local}(\alpha) := \{x_t \mapsto \alpha(x) \mid x \in \mathsf{dom}(\alpha), t \in \mathit{NodeI}(x)\}$ denote the *matching* assignment of the corresponding guess variables. Further, for a set $\mathcal{S}$ of literals and an assignment ι, we say assignment ι *respects* $\mathcal{S}$, if $(\bigwedge_{l\in\mathcal{S}} l)[\iota]$ evaluates to true.

Correctness. Next, we establish correctness of reduction R.

Lemma 5.8. *Let* κ *be any assignment of at least two variables* $x_t, x_{t'} \in \mathit{VarI}(\mathsf{var}(D))$ *such that* $\kappa(x_t) \neq \kappa(x_{t'})$ *for nodes* $t, t' \in \mathit{NodeI}(x)$ *with* $x \in \mathsf{var}(D)$*. Then,* $R(Q, \mathcal{T})[\kappa]$ *is invalid.*

Proof. We construct an assignment κ' which extends κ and sets certain variables in VarB. Then, we show that $R(Q, \mathcal{T})[\kappa']$ is invalid, which suffices since VarB is universally quantified. The construction of κ' is as follows: for every $b \in [\![x]\!]_{t''}$ and every $t'' \in N$ where $x \in \chi(t'')$, we set $\kappa'(b) := 1$, if b is a variable; and $\kappa'(b) := 0$, otherwise. Assume towards contradiction that there is an assignment $\kappa'' : \mathit{VarI}(\mathsf{var}(D)) \cup \mathit{VarB} \cup \mathit{VarBV} \to \{0, 1\}$ that extends κ' such that $R(Q, \mathcal{T})[\kappa'']$ is valid. In particular, the assignment κ'' sets variables in VarBV such that every clause in the assigned variables of $R(Q, \mathcal{T})$, in particular, parts $R_\mathcal{G}$ and $R_\mathcal{P}$, is valid under the assignment κ''. By Condition (ii) of the definition of a tree decomposition (connectedness), $\mathcal{T}[x]$ induces a connected tree as well. In consequence, irrelevant of how κ'' assigns variable $v_{r'}$ for the root node r' of $\mathcal{T}[x]$, the clauses in Formulas (5.10) and (5.11) enforce that exactly the same truth value $v = \kappa''(v_n)$ has to be set for any node $n \in \mathit{NodeI}(x)$. Then, $\kappa''(v_t) = v$ and $\kappa''(v_{t'}) = v$ holds. By part $R_\mathcal{G}$ of our reduction, more precisely, Formulas (5.1) and (5.2), we conclude that both $\kappa''(x_t) = v$ and $\kappa''(x_{t'}) = v$, which contradicts that $\kappa(x_t) \neq \kappa(x_{t'})$. □

Lemma 5.9. *Given an assignment* $\iota : \mathit{VarI}(\mathsf{var}(D)) \cup \mathit{VarB} \cup \mathit{VarBV} \to \{0, 1\}$*. Then, for any assignment* $\kappa : \mathit{VarI}(\mathsf{var}(D)) \cup \mathit{VarB} \cup \mathit{VarBV} \cup \mathit{VarB3} \cup \mathit{VarBV3} \to \{0, 1\}$ *that extends* ι*,* $R(Q, \mathcal{T})[\kappa]$ *is invalid, if (a) there is no term* $d_i \in D$ *with* $t = \delta_Q^{-1}(d_i)$

such that κ respects $\mathsf{bit\text{-}term}(d_i, t)$*. Now assume that there is $d_i \in D$ with κ respecting* $\mathsf{bit\text{-}term}(d_i, \delta_Q^{-1}(d_i))$*, then $R(Q, \mathcal{T})[\kappa]$ is also invalid, if (b) $\kappa(v_{t,j}) \neq \kappa(x_{t'})$, where $x = \mathsf{tvar}(d_i, j)$ for some $1 \leq j \leq 3$ and $t' \in NodeI(x)$.*

Proof. Assume towards a contradiction that (a) is not the case, i.e., there is no $d_i \in D$ such that κ respects $\mathsf{bit\text{-}term}(d_i, t)$ for $t = \delta_Q^{-1}(d_i)$ and still $R(Q, \mathcal{T})[\kappa]$ is valid. Observe that by $R(Q, \mathcal{T})$, in particular, by construction of the check part $R_{\mathcal{CK}}$ of R, $\kappa(sat_{\leq r}) = 1$ by (5.8) and therefore $\kappa(sat_t) = 1$ by (5.5) for at least one node $t \in N$ has to be set in κ. This, however, implies by (5.9) that $t = \delta_Q^{-1}(d_i)$ for some $d_i \in D$. In consequence, by construction of (5.6) and (5.7), κ respects $\mathsf{bit\text{-}term}(d_i, t)$, where $t = \delta_Q^{-1}(d_i)$, contradicting the assumption.

Towards contradicting (b), assume that there is $d_i \in D$ with $t = \delta_Q^{-1}(d_i)$ and $x = \mathsf{tvar}(d_i, j)$ as well as $t' \in NodeI(x)$ such that $\kappa(v_{t,j}) \neq \kappa(x_{t'})$ and still $R(Q, \mathcal{T})[\kappa]$ is valid. Observe that for any two nodes $t'', t''' \in \mathcal{T}[x]$, κ respects $[\![x]\!]_{t'',j}$ and $[\![x]\!]_{t''',j}$ by (5.12) and connectedness of $\mathcal{T}[x]$. Further, for any $t'', t''' \in \mathcal{T}[x]$, $\kappa(v_{t'',j}) = \kappa(v_{t''',j})$ by (5.13). Then, since (5.3) and (5.4) ensure that $\kappa(x_{t'}) = \kappa(v_{t',j})$, ultimately by connectedness of $\mathcal{T}[x]$, $\kappa(v_{t,j}) = \kappa(x_{t'})$ holds. □

Theorem 5.10 (Correctness)**.** *Let Q be a QBF of the form $Q = Q_1 V_1 . Q_2 V_2 . \cdots \forall V_\ell . D$ where D is in DNF. Then, for any assignment $\alpha : \mathsf{fvar}(Q) \to \{0, 1\}$, we have $Q[\alpha]$ is valid if and only if $R(Q, \mathcal{T})[\alpha']$ is valid, where assignment: $\mathsf{fvar}(R(Q, \mathcal{T})) \to \{0, 1\}$ is such that $\alpha' = \mathsf{local}(\alpha)$.*

Proof. Let $\mathcal{T} = (T, \chi, \delta_Q)$ be the labeled tree decomposition that is computed when constructing R, where $T = (N, A)$. We proceed by induction on the quantifier rank ℓ.

Base case. Assume $\ell = 1$.

"$\Longrightarrow$": Let α be an assignment to the free variables of Q for which $Q[\alpha]$ is valid. Further, let $\alpha' := \mathsf{local}(\alpha)$. We show that $R(Q, \mathcal{T})[\alpha']$ is valid as well. Let therefore ι be an arbitrarily chosen assignment to the variables in V_1. Since $\ell = 1$, we have $Q_1 = \forall$. We define an assignment $\kappa : VarI(\forall, V_1, \emptyset) \to \{0, 1\}$ such that $\kappa(x_t) := \iota(x)$ for every $x_t \in VarI(\forall, V_1, \emptyset)$ with $t \in N$ and $x \in \mathsf{var}(D)$. Next, we define an assignment $\kappa' : VarI(\forall, V_1, \emptyset) \cup VarI(\exists, \emptyset, V_1) \to \{0, 1\}$ that extends κ and sets $\kappa'(x_{t'}) := \iota(x)$ for every $x_{t'} \in VarI(\exists, \emptyset, V_1)$ with $t' \in N$. Assignment κ' has by construction the same truth value for each of the copies x_t of x, which is needed for $R(Q, \mathcal{T})[\alpha' \cup \kappa]$ to be valid in order to not contradict Lemma 5.8.

Then, we construct an assignment κ'', which extends κ' by the variables in *VarB3*, *VarBV3*, and *VarSat*. By construction of ι and since $Q[\alpha]$ is valid, $Q[\alpha \cup \iota]$ is valid, which is the same as $D[\alpha \cup \iota]$ is valid. In consequence, as D is in DNF, there is at least one term $d \in D$ such that $d[\alpha \cup \iota]$ is valid. Depending on the term d, we assign the variables in *VarB3*, *VarBV3*, and *VarSat* with assignment κ''. By Definition 2.17, there is a unique node $t = \delta_Q^{-1}(d)$ in the labeled tree decomposition for the term d. Then, we set $\kappa''(sat_{\leq t}) := \kappa''(sat_t) := 1$. For every ancestor t' of $t \in N$, we assign $\kappa''(sat_{\leq t'}) := 1$.

For every node $s \in N$ that is not an ancestor of t, we set $\kappa''(sat_{\leq s}) := 0$. Finally, for every node u, where $u \neq t$, we set $\kappa''(sat_u) := 0$. For every node $t \in N$ and $1 \leq j \leq 3$ with $\mathsf{tvar}(d,j) \notin \chi(t)$, we set κ'' such that it respects $[\![nil]\!]_{t,j} \cup \{\mathsf{bv}(d,t,j)\}$. Finally, for every node t and $1 \leq j \leq 3$ with $\mathsf{tvar}(d,j) \in \chi(t)$, we set κ'' such that it respects $\mathsf{bit\text{-}term}(d,t)$.

It remains to prove that for every assignment $\beta : \mathit{VarB} \to \{0,1\}$, there is an assignment $\zeta : \mathit{VarBV} \to \{0,1\}$ for which $R(Q,\mathcal{T})[\alpha' \cup \kappa'' \cup \beta \cup \zeta]$ is valid. For every variable $x \in \chi(t)$, if for every node $t \in N$, assignment β respects $[\![x]\!]_t$, then we set $\zeta(v_t) := (\alpha \cup \iota)(x)$. Otherwise, $\zeta(v_t) := 0$, since we can assign any truth value here. By construction of α' and κ'', clauses in Formulas (5.3) and (5.4) are satisfied of $R_{\mathcal{G}}$. Every clause of $R_{\mathcal{CK}}$ of $R(Q,\mathcal{T})$ is satisfied by construction of $\kappa'' \setminus \kappa'$ (and also by κ'') and ζ. Clauses in Formulas (5.12) and (5.13) of $R_{\mathcal{P}}$ are satisfied by $\kappa'' \setminus \kappa'$. Further, clauses in Formulas (5.1) and (5.2) of $R_{\mathcal{G}}$ are satisfied because of β, κ'', α', and ζ. Finally, the clauses in Formulas (5.10) and (5.11) of $R_{\mathcal{P}}$ are satisfied by construction of β, ζ, and κ'.

"$\Longleftarrow$": Let α be an assignment to the free variables of Q for which $Q[\alpha]$ is invalid. We show that if QBF $Q[\alpha]$ is invalid, then $R(Q,\mathcal{T})[\alpha']$ is invalid as well. Since $Q[\alpha]$ is invalid, $Q[\alpha \cup \iota]$ is invalid for any assignment $\iota : V_1 \to \{0,1\}$. Assume towards a contradiction that $R(Q,\mathcal{T})[\alpha']$ is valid. We define an assignment $\kappa := \mathsf{local}(\iota)$, which is $\kappa : \mathit{VarI}(\forall, V_1, \emptyset) \cup \mathit{VarI}(\exists, \emptyset, V_1) \to \{0,1\}$ such that $\kappa(x_t) := \iota(x)$ for every $x_t \in \mathit{VarI}(\forall, V_1, \emptyset) \cup \mathit{VarI}(\exists, \emptyset, V_1)$ with $t \in N$ and $x \in \mathsf{var}(D)$. Observe that by Lemma 5.8, κ is the only remaining option to obtain valid $R(Q,\mathcal{T})[\alpha]$. As a result, since $R(Q,\mathcal{T})[\alpha']$ is claimed valid, $R(Q,\mathcal{T})[\alpha' \cup \kappa]$ is valid as well. In consequence, by Lemma 5.9 Statement (a), there has to exist an extension κ' of $\alpha' \cup \kappa$ such that for some $d \in D$, κ' respects $\mathsf{bit\text{-}term}(d,t)$, where $t = \delta_Q^{-1}(d)$. By Lemma 5.9 Statement (b), for $1 \leq j \leq 3$ and every node $t' \in \mathit{NodeI}(y)$, where $y := \mathsf{tvar}(d,j)$, we have $\kappa'(v_{t,j}) = \kappa'(y_{t'})$. However, by construction of κ' and connectedness of $\mathcal{T}[y]$, then $(\alpha \cup \iota)$ respects d. In consequence, this contradicts our assumption that $Q[\alpha \cup \iota]$ is invalid.

Induction step ($\ell > 1$): We assume that the theorem holds for a given $\ell - 1$ and it remains to prove that it then holds for ℓ.

"$\Longrightarrow$": We proceed by case distinction on the first quantifier, i.e., (Case 1) $Q_1 = \exists$ and (Case 2) $Q_1 = \forall$. Thereby, we show that if $Q[\alpha]$ is valid and has quantifier rank ℓ, then $R(Q,\mathcal{T})[\alpha']$ is valid as well.

(Case 1) $Q_1 = \exists$: Since $Q[\alpha]$ is valid, we can construct at least one assignment $\iota : V_1 \to \{0,1\}$ such that $Q[\alpha \cup \iota]$ is valid. By induction hypothesis, since the QBF $Q[\alpha \cup \iota]$ has quantifier rank $\ell - 1$, and is valid, there are α', ι' such that $R(Q,\mathcal{T})[\alpha' \cup \iota']$ is valid as well. In particular, by induction hypothesis, $\alpha' = \mathsf{local}(\alpha), \iota' = \mathsf{local}(\iota)$ and therefore $R(Q,\mathcal{T})[\alpha']$ is valid as well.

(Case 2) $Q_1 = \forall$: Since $Q[\alpha]$ is valid, for any assignment ι of V_1, we obtain that $Q[\alpha \cup \iota]$ is valid. In the following, we denote by $R''(Q)$ the QBF that is obtained from $R(Q,\mathcal{T})$, where variables in $\mathit{VarI}(\exists, \emptyset, V_1)$ do not appear in the scope of a quantifier, i.e., these variables, while existentially quantified in $R(Q,\mathcal{T})$, are free variables in $R''(Q)$. By induction

hypothesis, since the QBF $Q[\alpha \cup \iota]$ has quantifier rank $\ell - 1$, and is valid, there are α', ι' with $\alpha' = \mathsf{local}(\alpha), \iota' = \mathsf{local}(\iota)$, such that $R''(Q)[\alpha' \cup \iota']$ is valid as well. Then, since ι was chosen arbitrarily, for every assignment κ of variables in $\mathsf{dom}(\iota') \cap \mathit{VarI}(Q_1, V_1, \emptyset)$, there is (by Lemma 5.8, since $R''(Q)[\alpha' \cup \iota']$ is valid) an assignment κ' of variables in $\mathsf{dom}(\iota') \cap \mathit{VarI}(\exists, \emptyset, V_1)$ such that $R(Q, \mathcal{T})[\alpha' \cup \kappa \cup \kappa']$ is valid. In consequence, $R(Q, \mathcal{T})[\alpha']$ is valid as well.

"$\Longleftarrow$": Again, we proceed by case distinction in order to show that if $Q[\alpha]$ is invalid and has quantifier rank ℓ, $R(Q, \mathcal{T})[\alpha']$ is invalid as well.

(Case 1) $Q_1 = \exists$: Since $Q[\alpha]$ is invalid, for every assignment ι of variables V_1 we have that $Q[\alpha \cup \iota]$ is also invalid. By induction hypothesis, since the QBF $Q[\alpha \cup \iota]$ has quantifier rank $\ell - 1$, and is invalid, there are assignments α' and ι' such that $R(Q, \mathcal{T})[\alpha' \cup \iota']$ is invalid as well, where $\alpha' = \mathsf{local}(\alpha), \iota' = \mathsf{local}(\iota)$. Therefore $R(Q, \mathcal{T})[\alpha']$ is invalid, since ι was chosen arbitrarily and by Lemma 5.8 ι' covers all relevant cases, where $R(Q, \mathcal{T})[\alpha']$ could be valid.

(Case 2) $Q_1 = \forall$: Since $Q[\alpha]$ is invalid, there is at least one assignment ι of variables V_1, such that $Q[\alpha \cup \iota]$ is invalid. By induction hypothesis, since QBF $Q[\alpha \cup \iota]$ has quantifier rank $\ell - 1$, and is invalid, there are assignments α' and ι' with $\alpha' = \mathsf{local}(\alpha), \iota' = \mathsf{local}(\iota)$, such that $R''(Q)[\alpha' \cup \iota']$ is invalid (R'' defined above), either. By Lemma 5.8, even for an assignment ι'' that restricts ι' to variables in $\mathsf{dom}(\iota') \cap \mathit{VarI}(Q_1, V_1, \emptyset)$, there cannot be an assignment κ to variables in $\mathsf{dom}(\iota') \cap \mathit{VarI}(\exists, \emptyset, V_1)$ such that $R(Q, \mathcal{T})[\alpha' \cup \iota'' \cup \kappa]$ is valid. In consequence, $R(Q, \mathcal{T})[\alpha']$ is invalid as well. □

Compression and Runtime. After having established the correctness of reduction R, we move on to showing that this reduction indeed compresses the treewidth of the resulting QBF $R(Q, \mathcal{T})$, as depicted in Figure 5.1. In particular, we prove this claim by constructing a tree decomposition $\mathcal{T}'$ of the primal graph of $R(Q, \mathcal{T})$ and show its relation to atomic, c-nice labeled tree decomposition $\mathcal{T}$ of Q, where $\mathsf{width}(\mathcal{T}) = \mathsf{tw}(\mathcal{G}_Q)$. Then, we discuss runtime properties of the reduction.

Lemma 5.11 (Compression)**.** *The reduction R exponentially decreases treewidth. In particular, $R'(R(Q, \mathcal{T}))$ constructs a QBF such that the treewidth of the primal graph of $R'(R(Q, \mathcal{T}))$ is $12 \cdot \lceil \log(k+1) \rceil + 7c + 6$, where k is the treewidth of $\mathcal{G}_Q$ and $c \leq k$.*

Proof. Assume an atomic, c-nice labeled tree decomposition $\mathcal{T} = (T, \chi, r)$ of Q of width k, where $T = (N, E)$. From this we will construct a tree decomposition $\mathcal{T}' = (T, \chi', r)$ of the primal graph of $R(Q, \mathcal{T})$. For each tree decomposition node $t \in N$ with $\mathsf{children}(t) = \{t_1, \ldots, t_s\}$, we set its bag $\chi'(t) := \{b \mid x \in \chi(t), b \in [\![x]\!]_t \cup [\![x]\!]_{t_1} \cup \cdots \cup [\![x]\!]_{t_s}\} \cup \{b \mid x \in \chi(t), 1 \leq j \leq 3, b \in [\![x]\!]_{t,j} \cup [\![x]\!]_{t_1,j} \cup \cdots \cup [\![x]\!]_{t_s,j}\} \cup \{x_{t'} \mid x_{t'} \in \mathit{VarI}(V), t' = t\} \cup \bigcup_{t' \in \{t, t_1, \ldots, t_s\}} \{v_{t',1}, v_{t',2}, v_{t',3}, v_{t'}, \mathit{sat}_{t'}, \mathit{sat}_{\leq t'}\}$. Observe that all the properties of tree decompositions are satisfied. In particular, connectedness is not destroyed since the only elements that are shared among (at most two) different tree decompositions nodes are in *VarB*, *VarBV* and in *VarB3*, and *VarBV3*.

Each bag $\chi'(t)$ contains bit-vectors $[\![x]\!]_t, [\![x]\!]_{t_1}, \ldots, [\![x]\!]_{t_s}$ for each $x \in \chi(t)$, resulting in at most $3 \cdot \lceil \log(k) \rceil$ many elements, since each node can have at most $s = 2$ many children. Further, each bag additionally consists of bit-vectors $[\![x]\!]_{t,j}, [\![x]\!]_{t_1,j}, \ldots, [\![x]\!]_{t_s,j}$ for each $x \in \chi(t) \cup \{nil\}$, where $1 \leq j \leq 3$, which are at most $3 \cdot 3 \cdot \lceil \log(k+1) \rceil$ many elements. In total everything sums up to at most $12 \cdot \lceil \log(k+1) \rceil + 7c + 6$ many elements per node, since $|\{x_{t'} \mid x_{t'} \in \mathit{VarI}(V), t = t'\}| \leq c$, and moreover $|\bigcup_{t' \in \{t,t_1,\ldots,t_s\}} \{v_{t'}, 1, v_{t'}, 2, v_{t'}, 3, v_{t'}, \mathit{sat}_{t'}, \mathit{sat}_{\leq t'}\}| \leq 6 \cdot (c+1)$ due to $\mathcal{T}$ being c-nice. Note that the treewidth of $R'(R(Q, \mathcal{T}))$ only marginally increases, since there are at most $\mathcal{O}(k \cdot \lceil \log(k) \rceil)$ many clauses in each bag of $\chi'(t)$ for any node $t \in N$, each of size at most $\mathcal{O}(\lceil \log(k) \rceil)$. However, the fresh variables, that were introduced during the 3-CNF reduction only turn up in at most two new clauses (that is, they have degree two in the primal graph). Further, the construction can be controlled in such a way, that each new clause consists of at most two fresh variables. In consequence, one can easily modify $\mathcal{T}'$, by adding at most $\mathcal{O}(k \cdot \lceil \log(k) \rceil^2)$ many intermediate nodes for each node $t \in N$, such that the width of $\mathcal{T}'$ is at most $12 \cdot \lceil \log(k+1) \rceil + 7c + 6$. □

Theorem 5.12 (Runtime)**.** *Given a QBF Q, where $D = \mathsf{matrix}(Q)$, k is the treewidth of the primal graph of Q. Then, constructing $R'(R(Q, \mathcal{T}))$ takes time $\mathcal{O}(2^{k^4} \cdot \|D\| \cdot c)$, where $c \leq k$.*

Proof. First, we construct [Bodlaender, 1996] a tree decomposition of the primal graph of Q of width k in time $2^{\mathcal{O}(k^3)} \cdot |\mathsf{var}(D)|$, consisting of at most $\mathcal{O}(2^{k^3} \cdot |\mathsf{var}(D)|)$ many nodes. Then, we compute an atomic, c-nice labeled tree decomposition in time $\mathcal{O}(k^2 \cdot 2^{k^3} \cdot (\|D\|)$, cf. Proposition 2.8 and [Kloks, 1994, Lemma 13.1.3], without increasing the width k, resulting in decomposition $\mathcal{T} = (T, \chi)$, where $T = (N, E)$ of the primal graph of Q. Note that thereby the number of nodes is at most $\mathcal{O}(k \cdot 2^{k^3} \cdot \|D\|)$. The reduction $R(Q, \mathcal{T})$ then uses at most $\mathcal{O}(k \cdot 2^{k^3} \cdot \|D\| \cdot c)$ many variables in $\mathit{VarI}(V)$ since in c-nice tree decompositions one node "introduces" at most c variables. The other sets of variables used in R are bounded by $\mathcal{O}(\lceil \log(k+1) \rceil \cdot k^2 \cdot 2^{k^3} \cdot \|D\|)$. Overall, there are $\mathcal{O}(\lceil \log(k+1) \rceil \cdot k^2 \cdot 2^{k^3} \cdot \|D\| \cdot c)$ many clauses constructed by $R(Q, \mathcal{T})$. Hence, the claim follows, since $R'(R(Q, \mathcal{T}))$ runs in time $\mathcal{O}(\lceil \log(k+1) \rceil^2 \cdot k^2 \cdot 2^{k^3} \cdot \|D\| \cdot c) \subseteq \mathcal{O}(2^{k^4} \cdot \|D\| \cdot c)$. □

Proof of the main result. We are in position to prove the main result of this section. To this end, we show that the lower bounds are closed under negation and restate Theorem 5.7.

Lemma 5.13. *Assume a given closed QBF of the form $Q = Q_1 V_1 . Q_2 V_2 . Q_3 V_3 \cdots Q_\ell V_\ell . F$, where $\ell \geq 1$ and F is in CNF if $Q_\ell = \exists$, and F is in DNF if $Q_\ell = \forall$. under the ETH, one cannot solve Q in time $\mathsf{tower}(\ell, o(k)) \cdot \mathrm{poly}(|\mathsf{var}(F)|)$ if and only if one cannot solve the negation $\neg Q$ in the same time.*

Proof. Assume towards a contradiction that $\neg Q$ can be solved in time $\mathsf{tower}(\ell, o(k)) \cdot \mathrm{poly}(|\mathsf{var}(F)|)$ under the ETH. But then, since inverting the result can be achieved in

constant time, under the ETH we can solve Q in time $\mathsf{tower}(\ell, o(k)) \cdot \mathrm{poly}(|\mathsf{var}(F)|)$. Hence, we arrive at a contradiction. □

Theorem 5.7 (QBF lower bound)**.** *Given any QBF of the form $Q = Q_1V_1.Q_2V_2.Q_3V_3 \cdots Q_\ell V_\ell.F$ where $\ell \geq 1$, and F is a 3-CNF formula (if $Q_\ell = \exists$), or F is a 3-DNF formula (if $Q_\ell = \forall$). Then, unless the ETH fails, Q cannot be solved in time $\mathsf{tower}(\ell, o(k)) \cdot \mathrm{poly}(|\mathsf{var}(F)|)$, where k is the treewidth of the primal graph $\mathcal{G}_Q$.*

Proof. We assume that Q is closed, i.e., for Q we have $\mathsf{fvar}(Q) = \emptyset$. We show the theorem by induction on the quantifier rank ℓ. For the induction base, where $\ell = 1$, the result follows from the ETH in case of $Q_\ell = \exists$ since $k \leq |\mathsf{var}(F)|$. If $Q_\ell = \forall$, by Lemma 5.13, the result follows. Note that for the case of $\ell = 2$, the result has already been shown [Lampis and Mitsou, 2017] as well.

For the induction step, we assume that the theorem holds for given Q of quantifier rank $\ell \geq 1$, where $Q_\ell = \forall$, the treewidth of primal graph $\mathcal{G}_Q$ is k, and F is in 3-DNF. We show that then the theorem also holds for quantifier rank $\ell + 1$. Towards a contradiction, we assume that in general we can solve any QBF Q' of quantifier rank $\ell+1$, in time $\mathsf{tower}(\ell + 1, o(\mathsf{tw}(\mathcal{G}_{Q'}))) \cdot \mathrm{poly}(|\mathsf{var}(C')|)$, where $C' = \mathsf{matrix}(Q')$. We compute an atomic c-nice labeled TD $\mathcal{T}$ of Q of width k, where c is in $\mathcal{O}(\log(k))$. We proceed by case distinction on the last quantifier $Q'_{\ell+1}$ of Q'.

(Case 1) $Q'_{\ell+1} = \exists$: Let $Q' = R'(R(Q, \mathcal{T}))$, $C' = \mathsf{matrix}(Q')$ be the matrix of Q', and k' be the treewidth of the primal graph of C'. Observe that Q' has quantifier rank $\ell + 1$ and is of the required form. By Lemma 5.11, $k' = 12 \cdot \lceil \log(k + 1) \rceil + 7c + 6$. As a result, since R is an fpt-reduction (including time for computing $\mathcal{T}$) according to Theorem 5.12, one can solve Q' in time $\mathsf{tower}(\ell + 1, o(12 \cdot \lceil \log(k + 1) \rceil + 7c + 6)) \cdot \mathrm{poly}(|\mathsf{var}(C')|) = \mathsf{tower}(\ell + 1, o(\log(k))) \cdot \mathrm{poly}(|\mathsf{var}(C')|)$. Therefore, by Theorem 5.10 we can solve Q in time $\mathsf{tower}(\ell, o(k)) \cdot \mathrm{poly}(|\mathsf{var}(F)|)$, which contradicts the induction hypothesis.

(Case 2) $Q'_{\ell+1} = \forall$: By Lemma 5.13 one can decide in time $\mathsf{tower}(\ell+1, o(k)) \cdot \mathrm{poly}(|\mathsf{var}(C')|)$ whether Q' is valid if and only if we can decide in time $\mathsf{tower}(\ell + 1, o(k)) \cdot \mathrm{poly}(|\mathsf{var}(C')|)$ whether $\neg Q'$ is valid. Note that after bringing $\neg Q'$ into prenex normal form, the last quantifier is $\exists$. Therefore, the remainder of this case is (Case 1). Hence, we have established the second case and this concludes the proof. □

Observe that therefore the result of Proposition 2.11 follows as a corollary from Theorem 5.7.

Further Consequences. We can further generalize Theorem 5.7 to the so-called incidence graph representation. The *incidence graph* of a formula F in CNF or DNF is the bipartite graph, which has as vertices the variables and clauses (terms) of F and an edge vc between every variable v and clause (term) c whenever v occurs in c in F [Samer and Szeider, 2010]. With this definition at hand, our result yields the following.

Corollary 5.14. *Given an arbitrary QBF Q of quantifier rank $\ell \geq 1$. Then, under the ETH one cannot solve Q in time* $\mathsf{tower}(\ell, o(k)) \cdot \mathrm{poly}(|\mathsf{var}(\mathsf{matrix}(Q))|)$*, where k is the treewidth of the incidence graph of* $\mathsf{matrix}(Q)$*.*

Proof. The claim follows from Theorem 5.7, since, in general, the treewidth k of the incidence graph of Q is bounded [Samer and Szeider, 2010; Fichte and Szeider, 2015] by treewidth k' of $\mathcal{G}_Q$, i.e., $k \leq k' + 1$. As a result, if the weaker lower bound of this corollary did not hold, Theorem 5.7 would be violated. □

Corollary 5.15 (Pathwidth bound)**.** *Given an arbitrary QBF of the form $Q = Q_1 V_1 . Q_2 V_2 . Q_3 V_3 \cdots Q_\ell V_\ell . F$, where $\ell \geq 1$ and F in 3-CNF (if $Q_\ell = \exists$) or 3-DNF (if $Q_\ell = \forall$). Then, unless the ETH fails, Q cannot be solved in time* $\mathsf{tower}(\ell, o(k)) \cdot \mathrm{poly}(|\mathsf{var}(F)|)$*, where k is the pathwidth of graph $\mathcal{G}_Q$.*

Proof. First, we show the claim for *pathwidth* by induction, which can be easily established for base case $\ell = 1$. For the case of $\ell = 2$, related work [Lampis and Mitsou, 2017] holds only for treewidth. However, the cases for $\ell \geq 2$ follow from the proof of Theorem 5.7, since every path decomposition is also a tree decomposition, and the proofs of lemmas and theorems used intermediately only rely on an *arbitrary* tree decomposition. To be more concrete, the proof of Theorem 5.7 relies on Lemma 5.11, whose proof shows compression for any tree decomposition, which hence also works for any path decomposition (PD) as well. Similarly, Theorem 5.12 also holds for pathwidth, since a PD of fixed pathwidth can be computed [Bodlaender, 1996] even in time $\mathcal{O}(2^{k^2} \cdot |\mathsf{var}(F)|)$, and since computation of atomic, c-nice labeled decompositions works analogously for path decompositions. Further, the remainder of the proof holds for the thereby obtained PD, since the construction works for any TD. Finally, Lemma 5.13 holds independently of the parameter. As a result, reductions R' and R used by Theorem 5.7 indeed are sufficient for PDs. □

Remark 5.16. *We remark that reduction R can be generalized to finite,* non-Boolean *domains (QCSP, e.g., [Ferguson and O'Sullivan, 2007]). For given variables V of a QCSP formula Q, the variables in VarI(V), VarBV, and VarBV3 have to be made non-Boolean, whereas the other variables used in R stay Boolean. Consequently, one obtains similar results as in related work [Lampis and Mitsou, 2017], but for quantifier rank $\ell \geq 3$. Indeed, under the ETH, the validity of QCSPs Q over domain $\mathcal{D}$ of quantifier rank ℓ, where $k = \mathsf{pw}(\mathcal{G}_Q)$, cannot be decided in time* $\mathsf{tower}(\ell - 1, |\mathcal{D}|^{o(k)}) \cdot \mathrm{poly}(|\mathsf{var}(Q)|)$*, see [Fichte et al., 2020b].*

Finally, we establish a corollary that improves a result from the literature. To this end, we denote for given positive number n by $\log^*(n)$ the smallest value i such that $\mathsf{tower}(i, 1) \geq n$. A known result [Atserias and Oliva, 2014, Corollary 1] states Σ_ℓ^{P}-hardness for instances Q of $(4 \cdot \log^*(|\mathsf{var}(\mathsf{matrix}(Q))|))$-QSAT, whereas here we establish para-Σ_ℓ^{P}-hardness for instances Q' of $(\log^*(|\,\mathsf{var}(\,\mathsf{matrix}(Q'))|))$-QSAT. This is possible by applying our established reduction R, which is rather fine-grained since it only increases quantifier rank by one, and it works indeed for any QBF, and not just for a certain classes of QBFs

in contrast to the known result. As a consequence, whenever a new class of ℓ-QBFs with a certain treewidth or pathwidth guarantee was discovered, which is still Σ_ℓ^{P}-hard, one *immediately* obtains para-Σ_ℓ^{P}-hardness by using reduction R. Then, one could potentially further improve quantifier alternations by applying reduction R, which is (asymptotically) tight under the ETH.

Corollary 5.17. *Given any integer $\ell \geq 1$. Then, deciding* QSAT *is para-Σ_ℓ^{P}-hard when parameterized by pathwidth of the primal graph $\mathcal{G}_Q$ for input QBFs of the form $Q = Q_1 V_1 . Q_2 V_2 . Q_3 V_3 \cdots Q_{\ell+\log^*(|\mathsf{var}(F)|)} V_{\ell+\log^*(|\mathsf{var}(F)|)} . F$, where is F in 3-CNF (if $Q_\ell = \exists$) or 3-DNF (if $Q_\ell = \forall$).*

Proof. Given a closed QBF of the form $Q' = Q_1 V_1' . Q_2 V_2' . Q_3 V_3' \cdots Q_\ell V_\ell' . F'$, where $\ell \geq 1$ and F' is in 3-CNF if $Q_\ell = \exists$, and F' is in 3-DNF if $Q_\ell = \forall$, and k' is the pathwidth of $\mathcal{G}_{Q'}$. Then, we apply our reduction R followed by R' on Q' and iteratively apply R and R'. We repeat this step exactly $\log^*(k')$ many times and refer to the final result by Q''. Note that the solutions to problem QSAT on Q' and Q'' are equivalent by Theorem 5.10. Then, the resulting pathwidth k'' of $\mathcal{G}_{Q''}$ is in $\mathcal{O}(1)$ by Lemma 5.11, i.e., parameter k'' is constant. Hence, since Q' is hard for Σ_ℓ^{P}, also Q'' is hard for Σ_ℓ^{P}, and Q'' is para-Σ_ℓ^{P}-hard since k'' is a constant. Observe that $k' \leq |\mathsf{var}(F')| \leq |\mathsf{var}(\mathsf{matrix}(Q''))|$. As a result, QSAT for QBFs of the form Q above is hard for para-Σ_ℓ^{P}. □

5.1.3 Discussion and Outlook

In this section we presented a lower bound for deciding the validity (QSAT) of quantified Boolean formulas (QBFs). Thereby, we have significantly extended the current state-of-the-art of this line of research: So far, lower bound results under the ETH for QSAT parameterized by treewidth were not available for all levels of the polynomial hierarchy. The generalization of this result in Theorem 5.7 does not only cover QBFs, parameterized by treewidth and an arbitrary quantifier rank, but solves a natural question for a well-known problem in complexity theory. Interestingly, the result confirms the (asymptotic) optimality of the algorithm by Chen [2004] for solving QSAT and thereby answers a longstanding open question. Indeed, one cannot expect to solve QSAT of quantifier rank ℓ significantly better than in time $\Omega^*(\mathsf{tower}(\ell, k))$ in the treewidth k. The proof of this result relies on a novel reduction approach that makes use of a fragile balance between redundancy and structural dependency (captured by treewidth) and uses tree decompositions as a "guide" in order to achieve exponential *compression* of the parameter treewidth.We encode core ideas of dynamic programming on tree decompositions and obtain a technique for compressing treewidth. Note that both our technique and the results naturally carry over to path decompositions and pathwidth.

Given the nature of our reduction, we observe that the reduction might also serve in reducing treewidth in practice. In particular, solvers based on tree decompositions such as the QBF solver dynQBF [Charwat and Woltran, 2019] could benefit from significantly reduced treewidth; at the cost of increased quantifier rank by one. Since dynQBF is

capable [Lonsing and Egly, 2018a] of solving instances up to treewidth 80 with quantifier rank more than two, slightly increasing the quantifier rank might be in practice a good trade-off for decreasing the treewidth significantly.

Another advantage of our reduction is that it gives rise to a versatile *methodology* for showing lower bounds for arbitrary problems by reduction from ℓ-QSat, parameterized by treewidth, which will be discussed in detail in Section 6.1. Thereby we avoid tedious reductions from Sat (directly using ETH), which involves problem-tailored gadgets to construct instances whose treewidth is ℓ-fold logarithmic in the number of variables or clauses of the given Sat formula. Further, we will list in Section 6.2 a number of showcases to illustrate the applicability of this approach to natural problems that are beyond the second level of the polynomial hierarchy.

One direction for future work is to explore further problems parameterized by treewidth and to establish tightness of the so far existing upper bounds. Another important direction is to work out techniques and showcases for "non-canonical" lower bounds, where fptl-reductions are not sufficient and using a customized function g is necessary. Hence, our goal is to continue this line of research in order to use this toolkit for problems that do not exhibit (e.g., [Lokshtanov et al., 2018]) canonical runtimes, where fptl-reductions suffice. We hope this work will foster research and new insights on lower bounds.

5.2 Lower Bounds for Asp and Treewidth

This section concerns the hardness of Asp when considering treewidth. To this end, we again assume that the exponential time hypothesis (cf. Hypothesis 2.1) holds. Recall, that under this assumption, neither Models, nor problems Tight Asp, or Supported Models can be solved in better than single exponential time in the treewidth, as stated in Propositions 3.3 and 3.9. This also shows that the reduction of Section 4.2 consisting of Formulas (4.4)–(4.8) cannot be significantly improved, cf. Corollary 4.8.

Now, we also present such a result for the richer fragment of disjunctive programs, i.e., we focus on the problem Disjunctive Asp. With the help of the main result of the previous section as given in Theorem 5.7, we obtain the following lower bound, thereby assuming that the exponential time hypothesis holds.

Theorem 5.18. *Unless the ETH fails,* Disjunctive Asp *cannot be solved in time* $2^{2^{o(k)}} \cdot \mathrm{poly}(|\mathrm{at}(\Pi)|)$ *for an arbitrary program instance* Π *where* k *is the treewidth of the primal graph* $\mathcal{G}_\Pi$.

Proof. Assume for proof by contradiction that there is such an algorithm. We show that this contradicts a special case of Theorem 5.7, namely Proposition 2.11, which states that one cannot decide the validity of a QBF $\forall V_1.\exists V_2.F$ in time $2^{2^{o(k)}} \cdot \mathrm{poly}(|F|)$, where F is in CNF. Let $Q = \forall V_1.\exists V_2.F$ be an instance of 2-QSat parameterized by the treewidth k (of $\mathcal{G}_Q$). Then, we reduce to an instance Π of Asp when parameterized by treewidth of $\mathcal{G}_\Pi$ such that Π is as follows. We employ a well-known reduction R [Eiter and Gottlob,

1995a, Theorem 3], which transforms $\exists V_1.\forall V_2.F$, where F is in 3-DNF, into $\Pi = R(Q)$ and gives a yes instance Π of Asp if and only if $\exists V_1.\forall V_2.F$ is a yes instance of 2-QSat. To this end, we use as atoms v_i as well as nv_i for each $v_i \in V_1 \cup V_2$ and an additional atom w. More precisely, we construct for each $v_i \in V_1 \cup V_2$ the rule $v_i \vee nv_i \leftarrow$ and for each $v_i \in V_2$ we additionally construct $v_i \leftarrow w$ as well as $nv_i \leftarrow w$. Further, for each $v_i \in V_2$, we add the rule $w \leftarrow v_i, nv_i$. For each term $d \in F$ with $d = l_1 \wedge l_2 \wedge l_3$ we construct $w \leftarrow l_1^*, l_2^*, l_3^*$, where for $1 \leq j \leq 3$ we let $l_j^* := v_i$ if $l_j = v_i$ and otherwise (if $l_j = \neg v_i$) we define $l_j^* := nv_i$. Finally, we construct the rule $\leftarrow \neg w$. The proof of correctness follows exactly as in the original source of the reduction [Eiter and Gottlob, 1995a, Theorem 3].

However, it remains to show that the reduction R indeed increases the treewidth only linearly and that this suffices to conclude the proof. Therefore, let $\mathcal{T} = (T, \chi)$ be TD of $\mathcal{G}_F$ of width k. We transform $\mathcal{T}$ into a TD $\mathcal{T}' = (T, \chi')$ of $\mathcal{G}_\Pi$ as follows. For each bag $\chi(t)$ of $\mathcal{T}$, we define $\chi'(t) := \chi(t) \cup \{nv_i \mid v_i \in \chi(t)\} \cup \{w\}$. Observe that for the width k' of $\mathcal{T}'$, $k' \leq 2 \cdot k + 1$ holds. By construction of R, $\mathcal{T}'$ is a TD of $\mathcal{G}_\Pi$.

Observe that R is an *fptl*-reduction, which we require, as results do not carry over from plain fpt-reductions. We finally have to argue that the result holds if R is an fptl-reduction. Assume towards a contradiction that there is an algorithm solving Asp in time $2^{2^{o(k)}} \cdot \text{poly}(|\text{at}(\Pi)|)$. However, then we can use this algorithm in order to construct an algorithm for solving any 2-QSat instance $Q = \exists V_1.\forall V_2.F$ via reduction R. Obviously, the resulting algorithm runs in time $2^{2^{o(2k+1)}} \cdot \text{poly}(|\mathsf{var}(F)|)$, where k is the treewidth of $\mathcal{G}_F$. Under the ETH, the existence of such an algorithm contradicts Proposition 2.11, which prohibits an algorithm running in time $2^{2^{o(k)}} \cdot \text{poly}(|\mathsf{var}(F)|)$. This concludes the proof and establishes the theorem. □

From this, we automatically obtain the following result, since problem Asp is strictly more general than problem Disjunctive Asp.

Proposition 5.19. *Unless the ETH fails,* Asp *cannot be solved in time* $2^{2^{o(k)}} \cdot \text{poly}(|\text{at}(\Pi)|)$ *for an arbitrary program instance* Π *where* k *is the treewidth of the primal graph* $\mathcal{G}_\Pi$.

While this result can be obtained by just reducing from 2-QSat, the precise lower bound for Normal Asp or HCF Asp requires more attention.

5.2.1 Towards a Lower Bound for Normal Asp and Treewidth

Next, we focus on establishing a lower bound that matches the runtime of our algorithm as discussed in Section 3.2.2. To this end, recall that for deciding Asp for a normal or head-cycle-free program we obtain runtime upper bounds of $2^{k \cdot \log(k)} \cdot (\text{at}(\Pi) + |\Pi|)$, as presented in Theorem 3.16. Now, the high-level reason for Asp being harder than Sat when assuming bounded treewidth, lies in the issue that a TD, while capturing the structural dependencies of a program, might force an evaluation that is completely different from the level mappings proving answer sets. Consequently, during dynamic

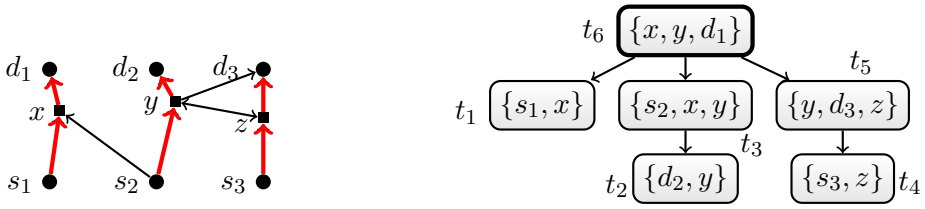

Figure 5.2: An instance $\mathcal{I} = (G, P)$ (left) of the Disjoint Paths problem and a TD of G (right).

programming for Asp, one needs to store in each table τ_t for each node t during post-order traversal, in addition to an interpretation (candidate answer set), also a level mapping among the atoms in those interpretations, cf. Chapter 3. We show that under reasonable assumptions in complexity theory, this worst case cannot be avoided. Then, the resulting runtime consequences cause Asp to be slightly harder than Sat, where in contrast to Asp storing a table τ_t of only assignments for each node t suffices.

The Disjoint Paths Problem. We show our novel hardness result by reducing from the (directed) Disjoint Paths problem, which is a graph problem defined as follows. Given a directed graph $G = (V, E)$, and a set $P \subseteq V \times V$ of disjoint pairs of the form (s_i, d_i) consisting of *source* s_i and *destination* d_i, where $s_i, d_i \in V$ such that each vertex occurs at most once in P, i.e., $\left|\bigcup_{(s_i,d_i)\in P}\{s_i, d_i\}\right| = 2 \cdot |P|$. Then, (G, P) is an instance of the Disjoint Paths problem, asking whether there exist $|P|$ many (vertex-disjoint) paths from s_i to d_i for $1 \leq i \leq |P|$. Concretely, each vertex of G is allowed to appear in at most one of these paths. For the ease of presentation, we assume without loss of generality [Lokshtanov et al., 2011] that sources s_i have no incoming edge (x, s_i), and destinations d_i have no outgoing edge (d_i, x).

Example 5.20. *Figure 5.2 (left) shows an instance $\mathcal{I} = (G, P)$ of the* Disjoint Paths *problem, where P consists of pairs of the form (s_i, d_i). The only solution to $\mathcal{I}$ is both emphasized and colored in red. Figure 5.2 (right) depicts a TD of G.*

While under the ETH, Sat cannot be solved in time $2^{o(k)} \cdot \text{poly}(|\mathsf{var}(F)|)$, where k is the treewidth of the primal graph of a given Boolean formula F, the Disjoint Paths problem is considered to be even harder. Concretely, the problem has been shown to be slightly superexponential as stated in the following proposition.

Proposition 5.21 ([Lokshtanov et al., 2011])**.** *Under the ETH, the* Disjoint Paths *problem is slightly superexponential, i.e., any instance (G, P) with $G = (V, E)$ cannot be solved in time $2^{o(k \cdot \log(k))} \cdot \text{poly}(|V|)$, where $k = \mathsf{tw}(G)$.*

It turns out that the Disjoint Paths problem is a suitable problem candidate for showing the hardness of Asp. Next, we require the following notation of open pairs, whose result is then applied in our reduction. Given an instance (G, P) of the Disjoint

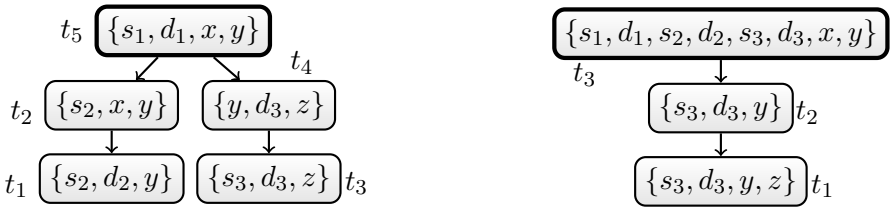

Figure 5.3: A pair-respecting TD (left), and a pair-connected TD $\mathcal{T}$ (right) of (G, P) of Figure 5.2.

Paths problem, a TD $\mathcal{T} = (T, \chi)$ of G, and a node t of T. Then, a pair $(s, d) \in P$ is *open in node* t, if either $s \in \chi_{\leq t}$ ("*open due to source* s") or $d \in \chi_{\leq t}$ ("*open due to destination* d"), but not both.

Proposition 5.22 ([Scheffler, 1994])**.** *An instance* (G, P) *of the* Disjoint Paths *problem does not have a solution if there is a TD* $\mathcal{T} = (T, \chi)$ *of* G *and a bag* $\chi(t)$ *with more than* $|\chi(t)|$ *many pairs in* P *that are open in a node* t *of* T*.*

Proof. The result, cf. [Scheffler, 1994], boils down to the fact that each bag $\chi(t)$, when removed from G, results in a disconnected graph consisting of two components. Between these components can be at most $|\chi(t)|$ different paths. □

Preparing pair-connected TDs. Before we present the actual reduction, we need to define a *pair-respecting* tree decomposition of an instance (G, P) of the Disjoint Paths problem. Intuitively, such a TD of G additionally ensures that each pair in P is encountered together in some TD bag.

Definition 5.23. *A TD* $\mathcal{T} = (T, \chi)$ *of* G *is a* pair-respecting TD *of* (G, P) *if for any pair* $p = (s, d)$ *with* $p \in P$*, (1) whenever* p *is open in a node* t *due to* s*, or due to* d*, then* $s \in \chi(t)$*, or* $d \in \chi(t)$*, respectively. Further, (2) whenever* p *is open in a node* t*, but not open in the parent* t' *of* t *("*p *is* closed in t'*"), both* $s, d \in \chi(t')$*.*

We observe that such a pair-respecting TD can be computed with only a linear increase in the (tree)width in the worst case. Concretely, we can turn any TD $\mathcal{T} = (T, \chi)$ of G into a pair-respecting TD $\mathcal{T}' = (T, \chi')$ of (G, P). Thereby, the tree T is traversed for each t of T in post-order, and vertices of P are added to $\chi(t)$ accordingly, resulting in $\chi'(t)$, such that conditions (1) and (2) of pair-respecting TDs are met. Observe, that this doubles the sizes of the bags in the worst case, since by Proposition 5.22 there can be at most bag-size many open pairs.

Example 5.24. *Figure 5.3 (left) shows a pair-respecting TD of* (G, P) *of Figure 5.2, which can be obtained by transforming the TD of Figure 5.2 (right), followed by simplifications.*

Given a sequence σ of pairs of P in the order of closure with respect to the post-order of T. We refer to σ by the *closure sequence* of $\mathcal{T}$. We denote by $p \in_i \sigma$ that pair p is the *pair closed i-th* in the order of σ. Intuitively, e.g., the first pair $p \in_1 \sigma$ indicates that pair $p \in P$ is the first to be closed when traversing T in post-order.

Definition 5.25. *A* pair-connected TD $\mathcal{T}=(T,\chi)$ *of* (G,P) *is a pair-respecting TD of* (G,P)*, if, whenever a pair* $p \in_i \sigma$ *with* $i>1$ *is closed in a node* t *of* T*, also for the pair* $(s,d) \in_{i-1} \sigma$ *closed directly before* p *in* σ*, both* $s,d \in \chi(t)$*.*

We can turn any pair-respecting, *nice* TD $\mathcal{T}'=(T,\chi')$ of width k into a pair-connected TD $\mathcal{T}''=(T,\chi'')$ with constant increase in the width. Let therefore pair $p \in_i \sigma$ be closed ($i>1$) in a node t, and pair $(s,d) \in_{i-1}$ be closed before p in node t'. Intuitively, we need to add s,d to all bags $\chi'(t'),\ldots,\chi'(t)$ of nodes encountered after node t' and before node t of the post-order tree traversal, resulting in χ''. However, the width of $\mathcal{T}''$ is at most $k+3\cdot|\{s,d\}| = k+6$, since in the tree traversal each node of T is passed at most 3 times, namely when traversing down, when going from the left branch to the right branch, and then also when going upwards. Indeed, to ensure $\mathcal{T}''$ is a TD (connectedness condition), we add at most 6 additional atoms to every bag.

Example 5.26. *Figure 5.3 (right) depicts a pair-connected TD of* (G,P) *of Figure 5.2, obtainable by transforming the pair-respecting TD of Figure 5.3 (left), followed by simplifications.*

5.2.2 Reducing from Disjoint Paths to Asp

In this section, we show the main reduction R of this section, assuming any instance $\mathcal{I} = (G,P)$ of the Disjoint Paths problem. Before we construct our program Π, we require a nice, pair-connected TD $\mathcal{T} = (T,\chi)$ of G, whose width is k and a corresponding closure sequence σ. By Proposition 5.22, for each node t of $\mathcal{T}$, there can be at most k many open pairs of P, which we assume in the following. If this was indeed not the case, we can immediately output, e.g., $\{a \leftarrow \neg a\}$.

Then, we use the following atoms in our reduction. Atoms $e_{u,v}$, or $ne_{u,v}$ indicate that edge $(u,v) \in E$ is used, or unused, respectively. Further, atom r_u for any vertex $u \in V$ indicates that u is reached via used edges. Finally, we also need atom f_t^u for a node t of T, and vertex $u \in \chi(t)$, to indicate that vertex u is already finished in node t, i.e., u has one used, outgoing edge. The presence of this atom f_t^u in an answer set prohibits to take additional edges of u in parent nodes of t, which is needed due to the need of disjoint paths of the Disjoint Paths problem.

The instance $\Pi = R(\mathcal{I},\mathcal{T})$ constructed by reduction R consists of three program parts, namely *reachability* $\Pi_{\mathcal{R}}$, *linking* $\Pi_{\mathcal{L}}$ of two pairs in P, as well as *checking* $\Pi_{\mathcal{C}}$ of disjointness of constructed paths. Consequently, $\Pi = \Pi_{\mathcal{R}} \cup \Pi_{\mathcal{L}} \cup \Pi_{\mathcal{C}}$. All three programs $\Pi_{\mathcal{R}}$, $\Pi_{\mathcal{L}}$, and $\Pi_{\mathcal{C}}$ are guided along TD $\mathcal{T}$, which ensures that the width of Π is only linearly increased. Note that this has to be carried out carefully. In particular, since the number

of atoms of the form $e_{u,v}$ using only vertices u, v that appear in one bag, can be already quadratic in the bag size. The goal of this reduction, however, admits only a linear overhead in the bag size. Consequently, we are, e.g., not allowed to construct rules in Π that require more than $\mathcal{O}(k)$ edges in one bag of a TD of $\mathcal{G}_\Pi$.

To this end, let the *ready edges* E_t^{re} *in node* t be the set of edges $(u, v) \in E$ not present in t anymore, i.e., $\{u, v\} \subseteq \chi(t') \setminus \chi(t)$ for any child node $t' \in \mathsf{children}(t)$. Further, let E_n^{re} for the root node $n = \mathrm{root}(T)$ additionally contain also all edges of n, i.e., $E \cap (\chi(n) \times \chi(n))$. Intuitively, ready edges for t will be processed in node t. Note that each edge occurs in exactly one set of ready edges. Further, for nice TDs $\mathcal{T}$, we always have $|E_t^{\mathrm{re}}| \leq k$, i.e., ready edges are linear in k.

Example 5.27. *Recall instance* $\mathcal{I}=(G, P)$ *with* $G=(V, E)$ *of Figure 5.2, and pair-connected TD* $\mathcal{T}=(T, \chi)$ *of* $\mathcal{I}$ *of Figure 5.3 (right). Then,* $E_{t_1}^{re}=\emptyset$, $E_{t_2}^{re}=\{(y, z), (z, y), (z, d_3), (s_3, z)\}$, *since* $z \notin \chi(t_2)$, *and* $E_{t_3}^{re}=E \setminus E_{t_2}^{re}$ *for root* t_3 *of* $\mathcal{T}$.

Reachability $\Pi_\mathcal{R}$. Program $\Pi_\mathcal{R}$ is constructed as follows.

$$e_{u,v} \leftarrow r_u, \neg ne_{u,v} \qquad \text{for each } (u, v) \in E_t^{\mathrm{re}} \tag{5.14}$$

$$ne_{u,v} \leftarrow \neg e_{u,v} \qquad \text{for each } (u, v) \in E_t^{\mathrm{re}} \tag{5.15}$$

$$r_v \leftarrow e_{u,v} \qquad \text{for each } (u, v) \in E_t^{\mathrm{re}} \tag{5.16}$$

Rules (5.14) and (5.15) ensure that there is a partition of edges in used edges $e_{u,v}$ and unused edges $ne_{u,v}$. Additionally, Rules (5.14) take care that only edges of adjacent, reachable vertices are used. Naturally, this requires that initially at least one vertex is reachable (constructed below). Rules (5.16) ensure reachability r_v over used edges $e_{u,v}$ for a vertex v.

Linking of pairs $\Pi_\mathcal{L}$. Program $\Pi_\mathcal{L}$ is constructed as follows.

$$\leftarrow \neg r_d \qquad \text{for each } (s, d) \in P \tag{5.17}$$

$$r_{s_1} \leftarrow \qquad \text{for } (s_1, d) \in_1 \sigma \tag{5.18}$$

$$r_{s_i} \leftarrow r_{s_{i-1}}, r_{d_{i-1}} \qquad \text{for each } (s_i, d) \in_i \sigma, (s, d_{i-1}) \in_{i-1} \sigma \tag{5.19}$$

Rules (5.17) make sure that, ultimately, destination vertices of all pairs are reached. As an initial, reachable vertex, Rule (5.18) sets the source vertex s reachable, whose pair is closed first. Then, the linking of pairs is carried out along the TD in the order of closure, as given by σ. Thereby, Rules (5.19) conceptually construct auxiliary links (similar to edges) between different pairs, in the order of σ, which is guided along the TD to ensure only a linear increase in treewidth of $\mathcal{G}_\Pi$ of the resulting program Π. Interestingly, these additional dependencies, since guided along the TD, do not increase the treewidth by much as we will see in the next subsection.

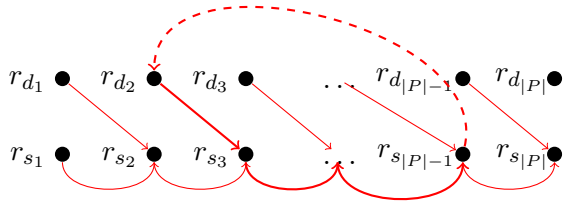

Figure 5.4: Positive dependency graph $D_{R_\mathcal{L}}$ (indicated by solid red edges) of Rules (5.19) constructed for any closure sequence σ such that $(s_i, d_i) \in_i \sigma$. If a source s_i reaches a destination d_j of a preceding pair, i.e., $j < i$, (depicted via the dashed red edge), this results in a cycle (consisting of all bold-faced edges) such that none of the atoms of the cycle can be proven.

Then, it is *crucial* that we prevent a source vertex s_i of a pair $(s_i, d_i) \in_i \sigma$ from reaching a destination vertex d_j of a pair $(s_j, d_j) \in_j \sigma$ preceding (s_i, d_i) in σ, i.e., $j < i$. To this end, we need to construct parts of cycles that prevent this. Concretely, if some source s_i reaches to d_j, i.e., d_j is reachable via s_i, the goal is to have a cyclic reachability from d_j to s_i, with no provability for corresponding reachability atoms of the cycle. Actually, Rules (5.19) also have the purpose of aiding in the construction of these potential positive cycles. Thereby we achieve that if d_j is reachable, this cannot be due to s_i, since reachability of $d_j, s_{j+1}, \ldots, s_i$ (therefore s_i itself) is required for reachability of s_i. Consequently, assuming that there is no further rule proving any of these reachability atoms, which we will ensure in the construction of program $\Pi_\mathcal{C}$ below, we end up with cyclic reachability if s_i is reaches d_j, such that none of the atoms of the cycle are proven. Figure 5.4 shows the positive dependency graph $D_{R_\mathcal{L}}$ of Rules (5.19), where pairs $(s_i, d_i) \in_i \sigma$, as discussed in the following example.

Example 5.28. *Consider the dependency graph $D_{R_\mathcal{L}}$ of Rules (5.19), as depicted in Figure 5.4. Observe that whenever s_i reaches some d_j with $j < i$, this causes a cycle $C{=}r_{s_i}, \ldots, r_{d_j}, r_{s_{j+1}}, \ldots, r_{s_{i-1}}, r_{s_i}$ over reachability atoms in $D_{R_\mathcal{L}}$ (cyclic dependency).*

If each vertex u of G can have at most one outgoing edge, i.e., only one atom $e_{u,v}$ in an answer set of $\Pi = R(\mathcal{I}, \mathcal{T})$, no atom of C can be proven (no further rule allows provability). Note that C could also be constructed by causing in the positive dependency graph $\mathcal{O}(|P|^2)$ many edges from r_{d_j} to r_{s_i} for $j < i$. This could be achieved, e.g., by constructing large rules, where reachability r_{d_j} of every preceding destination vertex is required in the positive body in order to reach a certain source vertex s_i, i.e., in order to obtain reachability r_{s_i}. However, this would cause an increase of structural dependency, and in fact, the treewidth increase would be beyond linear.

Checking of disjointness $\Pi_\mathcal{C}$. Finally, we create rules in Π that enforce at most one outgoing, used edge per vertex. This is required to ensure that we do not use a vertex twice, as required by the Disjoint Paths problem. We do this by guiding the

information, whether the corresponding outgoing edge was used, via atoms f_t^u along the TD to ensure that the treewidth is not increased significantly. Having at most one outgoing, used edge per vertex of G further ensures that when a source of a pair p reaches a destination of a pair preceding p in σ, then no atom of the resulting cycle as constructed in $\Pi_{\mathcal{L}}$ will be provable. Consequently, in the end every source of p has to reach the destination of p by the pigeon hole principle. Program $\Pi_{\mathcal{C}}$ is constructed for every node t with $t', t'' \in \mathsf{children}(t)$, if t has child nodes, as follows.

$$f_t^u \leftarrow e_{u,v} \qquad \text{for each } (u,v) \in E_t^{\text{re}}, u \in \chi(t) \tag{5.20}$$
$$f_t^u \leftarrow f_{t'}^u \qquad \text{for each } u \in \chi(t) \cap \chi(t') \tag{5.21}$$
$$\leftarrow f_{t'}^u, f_{t''}^u \qquad \text{for each } u \in \chi(t') \cap \chi(t''), t' \neq t'' \tag{5.22}$$
$$\leftarrow f_{t'}^u, e_{u,v} \qquad \text{for each } (u,v) \in E_t^{\text{re}}, u \in \chi(t') \tag{5.23}$$
$$\leftarrow e_{u,v}, e_{u,w} \qquad \text{for each } (u,v),(u,w) \in E_t^{\text{re}},\ v \neq w \tag{5.24}$$

Rules (5.20) ensure that the finished flag f_t^u is set for used edges $e_{u,v}$. Then, this information of $f_{t'}^u$ is guided along the TD from child node t' to parent node t by Rules (5.21). If for a vertex $u \in V$ we have $f_{t'}^u$ and $f_{t''}^u$ for two different child nodes $t', t'' \in \mathsf{children}(t)$, this indicates that two different edges were encountered both below t' and below t''. Consequently, this situation is avoided by Rules (5.22). Rules (5.23) make sure to disallow additional edges for vertex u in a TD node t, if the flag $f_{t'}^u$ of child node t' is set. Finally, Rules (5.24) prohibit two different edges for the same vertex u within a TD node.

Example 5.29. *Recall instance $\mathcal{I} = (G, P)$ with $G = (V, E)$ of Figure 5.2, pair-connected TD $\mathcal{T} = (T, \chi)$ of $\mathcal{I}$ of Figure 5.3 (right), and $E_{t_2}^{re} = \{(y,z),(z,y),(z,d_3),(s_3,z)\}$. We briefly present the construction of $\Pi_{\mathcal{C}}$ for node t_2.*

Rules	$\Pi_{\mathcal{L}}$
(5.20)	$f_{t_2}^y \leftarrow e_{y,z}$; $f_{t_2}^{s_3} \leftarrow e_{s_3,z}$
(5.21)	$f_{t_2}^{s_3} \leftarrow f_{t_1}^{s_3}$; $f_{t_2}^{d_3} \leftarrow f_{t_1}^{d_3}$; $f_{t_2}^{y} \leftarrow f_{t_1}^{y}$
(5.23)	$\leftarrow f_{t_1}^y, e_{y,z}$; $\leftarrow f_{t_1}^z, e_{z,y}$; $\leftarrow f_{t_1}^z, e_{z,d_3}$; $\leftarrow f_{t_1}^{s_3}, e_{s_3,z}$
(5.24)	$\leftarrow e_{z,y}, e_{z,d_3}$

The resulting program of the reduction consisting of Rules (5.14)–(5.24) is not unary. However, only Rules (5.19) as well as (5.22)–(5.24) are not unary. Still, Rules (5.23) and (5.24) can be turned unary by replacing the occurrence of $e_{u,v}$ in these two rules by $\neg ne_{u,v}$. Further, Rules (5.22) can be replaced by the following rules, which use an additional auxiliary atom "bad".

$$\text{bad} \leftarrow f_t^u, f_{t'}^u \qquad \text{for each } u \in \chi(t') \cap \chi(t''), t' \neq t'' \tag{5.25}$$
$$\leftarrow \text{bad} \tag{5.26}$$

On the other hand, for Rules (5.19) the resulting (positive) cycles of the dependency graph are required for the whole construction, cf. Figure 5.4. More precisely, it is indeed essential for the whole construction that reachability of a source s_i requires both reachability of the preceding source s_{i-1} and destination d_{i-1}. Otherwise we cannot prevent a source from reaching a preceding destination via cyclic reachability without provability and still linearly preserve the treewidth. Consequently, Rules (5.19) are *not unary* and we expect that this is crucial. Nevertheless, it was shown that non-unary programs are more expressive than unary programs [Janhunen, 2006]. Still, we are convinced that exploiting cyclic, unproven reachability such that the treewidth is not increased more than linearly, actually requires the usage of non-unary rules.

Example 5.30. *Consider again Figure 5.4, depicting the positive dependency graph $D_{R_\mathcal{L}}$ of Rules (5.19), as well as Example 5.28. More concretely, consider the same situation of Example 5.28, where a source s_i reaches some destination d_j with $j < i$, which causes a cycle $C{=}r_{s_i}, \ldots, r_{d_j}, r_{s_{j+1}}, \ldots, r_{s_i}$ over reachability atoms. Then, it is crucial for the construction that Rules (5.19) are not unary. To be more concrete, for the instantiated rule r with $r_{s_{j+1}} \in H_r$, we require that both $r_{s_j}, r_{d_j} \in B_r^+$. If instead of r we constructed two rules $r_{s_{j+1}} \leftarrow r_{s_j}$ and $r_{s_{j+1}} \leftarrow r_{d_j}$, every atom of the cycle C could be provable since $r_{s_{j+1}}$ can already be proven by the former rule. Further, also for the instantiated rule r' of Rules (5.19) with $r_{s_o} \in H_{r'}$ for every $j+1 < o \leq i$, we require that the body is not unary. If instead of such a rule r', we constructed two rules $r_{d_o} \leftarrow r_{s_{o-1}}$ and $r_{d_o} \leftarrow r_{d_{o-1}}$, every atom of the cycle C could be provable since r_{d_o} is already proven by the latter rule. Since, in particular the result should hold for any such cycle C, we rely on non-unary rules for our reduction to work.*

5.2.3 Correctness, Runtime Analysis, and Consequences

First, we show that the reduction is indeed correct, followed by a result stating that the treewidth of the reduction is at most linearly worsened, which is crucial for the runtime lower bound to hold. Then, we present the runtime and the (combined) main result of this section.

Lemma 5.31 (≤ 1 Outgoing Edge)**.** *Given any instance $\mathcal{I} = (G, P)$ of the* Disjoint Paths *problem, and any answer set M of $R(\mathcal{I}, \mathcal{T})$ using any pair-connected TD $\mathcal{T}$ of (G, P). Then, there cannot be two edges of the form $e_{u,v}, e_{u,w} \in M$.*

Proof. Assume towards a contradiction that there are three different vertices $u, v, w \in V$ with $e_{u,v}, e_{u,w} \in M$. Then, by Rules (5.24) there cannot be a node t with $(u, v), (u, w) \in E_t^{\text{re}}$. However, by the definition of TDs, there are nodes t', t'' with $(u, v) \in E_{t'}^{\text{re}}$ and $(u, w) \in E_{t''}^{\text{re}}$. By connectedness of TDs, u appears in each bag of any node of the path X between t' and t''. Then, either t' is an ancestor of t'' (or vice versa, symmetrical) or there is a common ancestor t. In the former case, $f_{t''}^u$ is justified by Rules (5.20) and so is $f_{\hat{t}}^u$ on each node $\hat{t}$ of X by Rules (5.21) and therefore ultimately Rules (5.23) fail due to $f_{t'}^u, e_{u,w} \in M$. In the latter case, $f_{t'}^u, f_{t'}^u$ is justified by Rules (5.20) and so is $f_{\hat{t}}^u$ on each node $\hat{t}$ of X by Rules (5.21). Then, Rules (5.22) fail due to $f_{t'}^u, f_{t''}^u \in M$. □

Theorem 5.32 (Correctness)**.** *Reduction R as proposed in this section is correct. More concretely, given an instance $\mathcal{I} = (G, P)$ of the* Disjoint Paths *problem, and a pair-connected TD $\mathcal{T} = (T, \chi)$ of G. Then, $\mathcal{I}$ has a solution if and only if the program $R(\mathcal{I}, \mathcal{T})$ admits an answer set.*

Proof. "⇒": Given any yes-instance $\mathcal{I}$ of Disjoint Paths problem. Then, there are disjoint paths $P_1, \ldots, P_i, \ldots P_{|P|}$ from s_1 to d_1, …, s_i to d_i, …, $s_{|P|}$ to $d_{|P|}$ for each pair $(s_i, d_i) \in P$. Assuming further pair-connected TD $\mathcal{T}$ of $\mathcal{I}$, we construct in the following an answer set M of $\Pi = R(\mathcal{I}, \mathcal{T})$. To this end, we collect reachable atoms $A := \{u \mid u \text{ appears in some } P_i, 1 \leq i \leq |P|\}$ and used edges $U := \{(u, v) \mid v \text{ appears immediately after } u \text{ in some } P_i, 1 \leq i \leq |P|\}$. Then, we construct answer set candidate $M := \{r_u \mid u \in A\} \cup \{e_{u,v} \mid (u, v) \in U\} \cup \{ne_{u,v} \mid (u, v) \in E \setminus U\} \cup \{f_t^u \mid (u, v) \in U \cap E_t^{\text{re}}\} \cup \{f_t^u \mid (u, v) \in U \cap E_{t'}^{\text{re}}, u \in \chi(t), t' \text{ is a descendant of } t \text{ in } T\}$. It remains to show that M is an answer set of Π. Observe that M indeed satisfies all the rules of $\Pi_{\mathcal{R}}$. In particular, by construction, we have reachability r_v for every vertex v of every pair in P, and the partition in used edges $e_{u,v}$ and unused edges $ne_{u,v}$ is ensured. Further, $\Pi_{\mathcal{L}}$ is satisfied, as, again by construction, for each vertex v of every pair in P, we have $r_v \in M$. Finally, $\Pi_{\mathcal{C}}$ is satisfied as by construction $f_t^u \in M$ iff $e_{u,v} \in M \cap E_t^{\text{re}}$ or $e_{u,v} \in M \cap E_{t'}^{\text{re}}$ for any descendant node t' of t with $u \in \chi(t)$. It is easy to see that M is indeed a $\subseteq$-smallest model of the reduct Π^M, since, atoms for used and unused edges form a partition of E.

"⇐": Given any answer set M of Π. First, we observe that we can only build paths from sources towards destinations, as sources have only outgoing edges and destinations allow only incoming edges. Further, by construction, vertices can only have one used, outgoing edge, cf. Lemma 5.31. Consequently, if a vertex had more than one used, incoming edge, one cannot match at least one pair of P (by combinatorial pigeon hole principle). Hence, in an answer set M of Π, there is at most one incoming edge per vertex. By construction of Π, in order to reach each d_i with $(s_i, d_i) \in_i \sigma$, s_i cannot reach some $d_{j'}$ with $j' < i$. Towards a contradiction assume otherwise, i.e., s_i reaches $d_{j'}$. But then, by construction of the reduction, we also have a reachable path from $d_{j'}$ to s_i, consisting of $d_{j'}, d_{j'+1}, \ldots, d_{i-1}, s_i$. Since every vertex has at most one incoming edge, $d_{j'}$ cannot have any other justification for being reachable, nor does any source on this path. Hence, this forms a cycle such that no atom of the cycle is proven, which can not be present in an answer set. Therefore, s_i only reaches d_i, since otherwise there would be at least one vertex s_j required to reach $s_{i'}$ with $(s_{i'}, d_{i'}) \in_{i'} \sigma$, $i' < j$. Consequently, we construct a witnessing path P_i for each pair $(s, d) \in_i \sigma$ as follows: $P_i := s, p_1, \ldots, p_m, d$ where $\{e_{s,p_1}, e_{p_1,p_2}, \ldots, e_{p_{m-1},p_m}, e_{p_m,d}\} \subseteq M$. Thus, P_i starts with s, follows used edges in M and reaches d. □

Lemma 5.33 (Treewidth-awareness)**.** *Given an instance $\mathcal{I} = (G, P)$ of the* Disjoint Paths *problem, and a pair-connected, nice TD $\mathcal{T}$ of $\mathcal{I}$ of width k. Then, the treewidth of $\mathcal{G}_\Pi$, where $\Pi = R(\mathcal{I}, \mathcal{T})$ is obtained by R, is at most $\mathcal{O}(k)$.*

Proof. Given any pair-connected, nice TD $\mathcal{T} = (T, \chi)$ of $\mathcal{I} = (G, P)$. Since $\mathcal{T}$ is nice, a node in T has at most $\ell = 2$ many child nodes. From $\mathcal{T}$ we construct a TD $\mathcal{T}' = (T, \chi')$ of $\mathcal{G}_\Pi$. Thereby we set for every node t of T, $\chi'(t) := \{r_u, f_t^u \mid u \in \chi(t)\} \cup \{e_{u,v}, ne_{u,v}, r_u, r_v, f_{t'}^u \mid (u,v) \in E_t^{\mathrm{re}}, t' \in \mathsf{children}(t), u \in \chi(t')\} \cup \{f_{t'}^u, f_t^u \mid t' \in \mathsf{children}(t), u \in \chi(t) \cap \chi(t')\}$. Observe that $\mathcal{T}'$ is a valid TD of $\mathcal{G}_\Pi$. Further, by construction we have $|\chi'(t)| \leq 2 \cdot |\chi(t)| + (4 + \ell) \cdot k + (\ell + 1) \cdot |\chi(t)|$, since $|E_t^{\mathrm{re}}| \leq k$. The claim sustains for nice TDs ($\ell = 2$). □

Corollary 5.34 (Runtime)**.** *Reduction R as proposed in this section runs for a given instance $\mathcal{I} = (G, P)$ of the* Disjoint Paths *problem with $G = (V, E)$, and a pair-connected, nice TD $\mathcal{T}$ of $\mathcal{I}$ of width k and h many nodes, in time $\mathcal{O}(k \cdot h)$.*

Next, we are in the position of showing the main result, namely the Normal Asp lower bound.

Theorem 5.35 (Lower bound)**.** *Given an arbitrary normal or HCF program Π, where k is the treewidth of the primal graph of Π. Then, unless the ETH fails, the consistency problem for Π cannot be solved in time $2^{o(k \cdot \log(k))} \cdot \mathrm{poly}(|\mathrm{at}(\Pi)|)$.*

Proof. Let (G, P) be an instance of the Disjoint Paths problem. First, we construct [Bodlaender et al., 2016] a nice TD $\mathcal{T}$ of $G = (V, E)$ of treewidth k in time $c^k \cdot |V|$ for some constant c such that the width of $\mathcal{T}$ is at most $5k + 4$. Then, we turn the result into a pair-connected TD $\mathcal{T}' = (T', \chi')$, thereby having width at most $k' = 2 \cdot (5k + 4) + 6$. Then, we construct program $\Pi = R(\mathcal{I}, \mathcal{T}')$. By Lemma 5.33, the treewidth of $\mathcal{G}_\Pi$ is in $\mathcal{O}(k')$, which is in $\mathcal{O}(k)$. Assume towards a contradiction that consistency of Π can be decided in time $2^{o(k \cdot \log(k))} \cdot \mathrm{poly}(|\mathrm{at}(\Pi)|)$. By correctness of R (Theorem 5.32), this solves (G, P), contradicting Proposition 5.21. □

Our reduction works by construction for any pair-connected TD. Consequently, this immediately yields a lower bound for *pathwidth*, which is similar to treewidth, but admits only *path decompositions* (TDs whose tree is just a path).

Corollary 5.36 (Pathwidth lower bound)**.** *Given any normal or HCF program Π, where k is the pathwidth of the primal graph of Π. Then, unless the ETH fails, the consistency problem for Π cannot be solved in time $2^{o(k \cdot \log(k))} \cdot \mathrm{poly}(|\mathrm{at}(\Pi)|)$.*

From Theorem 5.35, we follow that a general reduction from normal or HCF programs to Boolean formulas can probably not avoid the treewidth (pathwidth) overhead, which renders our reduction from the previous section ETH-tight.

Corollary 5.37 (ETH-tightness of the Reduction to Sat)**.** *Under the ETH, the increase of treewidth of the reduction using Formulas (4.4), (4.6)–(4.8), as well as (4.10)–(4.11) cannot be significantly improved.*

Proof. Assume towards a contradiction that one can reduce from an arbitrary normal logic program Π, where k is the treewidth of $\mathcal{G}_\Pi$ to a Boolean formula, whose treewidth is in $o(k \cdot \log(k))$. Then, this contradicts Theorem 5.35, as we can use an algorithm [Samer and Szeider, 2010; Fichte et al., 2018b, 2019b] for Sat being single exponential in the treewidth, thereby deciding consistency of Π in time $2^{o(k \cdot \log(k))} \cdot \text{poly}(|\text{at}(\Pi)|)$. □

Knowing that under the ETH, Tight Asp has roughly the same hardness for treewidth as Sat, cf. Proposition 3.9, we can derive the following corollary, which completes Theorem 5.35 and proves that under the ETH the reduction from a normal program using Formulas (4.19)–(4.30) for constructing a tight program, cannot be significantly improved.

Corollary 5.38 (ETH-tightness of the Reduction to Tight Asp)**.** *Let* Π *be any normal program, where the treewidth of* $\mathcal{G}_\Pi$ *is* k*. Then, under the ETH, one cannot reduce* Π *to a tight program* Π' *in time* $2^{o(k \cdot \log(k))} \cdot \text{poly}(|\text{at}(\Pi)|)$ *such that* $\mathsf{tw}(\mathcal{G}_{\Pi'})$ *is in* $o(k \cdot \log(k))$*.*

5.2.4 Discussion and Outlook

The curiosity of studying and determining the hardness of Asp and the underlying reasons has attracted the attention of the KR community for a long time. The section above discusses this question from a different angle, which hopefully will provide new insights into the hardness of Asp and foster follow-up work. The results indicate that, at least from a structural point of view, deciding Asp is already harder than Sat, since programs might compactly represent structural dependencies within the formalism. More concretely, compiling the hidden structural dependencies of a program to a Boolean formula, measured in terms of the well-studied parameter treewidth, most certainly causes a blow-up of the treewidth of the resulting formula. In the light of a known result [Atserias et al., 2011] on the correspondence of treewidth and the resolution width applied in Sat solving, this reveals that Asp might be indeed harder than solving Sat. Recall that in Section 4.3 we present a reduction from Asp to Sat that is aware of the treewidth in the sense that the reduction causes not more than this inevitable blow-up of the treewidth in the worst case.

The results in this section gives rise to plenty of future work. On the one hand, we are currently working on the comparison of different treewidth-aware reductions to Sat and variants thereof, and how these variants perform in practice. Further, we are curious about treewidth-aware reductions to Sat, which preserve answer sets bijectively or are modular [Janhunen, 2006]. We hope this work might reopen the quest to study the correspondence of treewidth and solving Asp similarly to [Atserias et al., 2011] for Sat. Also investigating further structural parameters "between" treewidth and directed variants of treewidth could lead to new insights, since for Asp directed measures [Bliem et al., 2016b] often do not yield efficient algorithms. Given the fine-grained expressiveness results for different (sub-)classes of normal programs and the resulting expressive power hierarchy [Janhunen, 2006], we are curious to see also studies in this direction and to

which extent results might differ, when further restricting to treewidth-aware reductions. Of particular interest might be the question, whether one can devise a different hardness proof for Normal Asp and treewidth (cf. Section 5.2), such that only unary rules are used.

CHAPTER 6

A Complexity Landscape for Treewidth

Out of complexity, find simplicity!

— Albert Einstein

The complexity results of Section 5.1 establish a conditional lower bound for the canonical problem QSat that matches and confirms existing upper bounds [Chen, 2004]. While the known upper bound was shown a while ago, the efforts to provide some kind of insights on hardness or lower bound results took more time, cf. [Pan and Vardi, 2006; Atserias and Oliva, 2014; Lampis and Mitsou, 2017]. These efforts finally result in Theorem 5.7, which is probably the most important result of the previous chapter. This theorem itself is quite powerful and it actually is the result of solving a long-open question, but the consequences and applications of this finding reach far beyond single statements. Indeed, it turns out that this result is the basis of a new methodology that lifts the status of QSat from being one canonical problem in classical complexity [Grohe, 2017; Immerman, 1999] to being the key problem in order to prove conditional lower bounds for problems parameterized by treewidth that are higher in the polynomial hierarchy. Together with the concept of decomposition-guided reductions of Chapter 4, this result is the last missing ingredient in order to unlock the full potential of quantified Boolean formulas (QBFs) for treewidth, which we demonstrate in the form of lower bound results for different problems and formalisms.

So far, the problem QSat already has an important role in descriptive complexity for characterizing hardness of a problem for a certain level of the polynomial hierarchy, since a polynomial-time reduction from QSat when restricted to quantifier rank ℓ suffices to establish hardness for the complexity class Σ_ℓ^{P} or Π_ℓ^{P} of the ℓ-th level of the polynomial hierarchy. Our methodology keeps this workflow, but slightly adapts it towards a more fine-grained version, where, e.g., decomposition-guided reductions can be used instead of plain polynomial-time reductions. Then, one can use decomposition-guided reductions

to reduce from QSat to the desired problem, in order to show that under reasonable assumptions it is not expected that the problem can be solved in a runtime that is polynomial in the instance size and better than ℓ-fold exponential in the treewidth. Later, we will say that this problem is then complete for the class $\mathtt{TW}^k_\ell$, which is a new runtime class that is motivated by the applicability of our methodology.

Interestingly, for treewidth already certain restrictions of the problem QSat seem to be a fairly prominent. This can be witnessed by further works in these directions that were presented around the time of the availability of precise lower bound results [Lampis and Mitsou, 2017] for QBFs of quantifier rank 2 under the exponential time hypothesis (ETH). Indeed, restricted fragments of the problem QSat were used to characterize further problems parameterized by treewidth in terms of both upper and lower bounds, cf. [Fichte and Hecher, 2019; Fichte et al., 2018b; Lampis et al., 2018], where Lampis, Mitsou, and Mengel [2018] even refer to the problem QSat as a potential "alternative" to Courcelle's theorem [Courcelle, 1990] due to its well-known, precise upper bounds [Chen, 2004]. Notably, Marx and Mitsou [2016] even refer to the intuitive meaning of implicit "quantifier alternation" of problems higher on the polynomial hierarchy, when arguing about treewidth lower bounds.

In this chapter of the thesis, we show that in general the problem QSat plays indeed a crucial role in characterizing hardness for treewidth, i.e., for establishing tight lower bounds under the exponential time hypothesis. First, we apply QSat and the findings of Section 5.1 in order to develop in Section 6.1 a methodology for showing conditional lower bounds for treewidth, which stems from a recent work [Fichte et al., 2020c]. Decomposition-guided reductions as introduced in Chapter 4 form a canonical tool for applying this lower bound methodology. Then, in Section 6.2 we use this methodology for lower bounds together with the techniques for upper bounds gained in Chapters 3 and 4 in order to classify problems according to the required runtime depending on treewidth when solving these problems. Inspired by works on general FPT runtime classes [Weyer, 2004; Downey et al., 2007], we define runtime classes for treewidth that are expected to form a strict hierarchy, unless Hypothesis 2.1, i.e., the exponential time hypothesis (ETH), fails. Section 6.2 also discusses some hardness results for these classes that can be obtained for a variety of problems relevant to knowledge representation and reasoning, and artificial intelligence in general. These hardness results together with the results of Chapters 3 and 4, allow us to present completeness results for these classes for problems discussed in the course of this thesis. Thereby, we are able to finally complete Table 1.1 of Chapter 1 and state the intuitive meaning of this table more precisely.

6.1 A Methodology for Lower Bounds

Recall that Theorem 5.7 of the previous chapter showed the following lower bound for evaluating QBFs, which we will use in order to establish a novel methodology for proving lower bounds for problems located on higher levels of the polynomial hierarchy, when parameterized by treewidth.

Theorem 5.7 (QBF lower bound)**.** *Given any QBF of the form* $Q = Q_1V_1.Q_2V_2.Q_3V_3 \cdots Q_\ell V_\ell.F$ *where* $\ell \geq 1$*, and* F *is a 3-CNF formula (if* $Q_\ell = \exists$*), or* F *is a 3-DNF formula (if* $Q_\ell = \forall$*). Then, unless the ETH fails,* Q *cannot be solved in time* $\mathsf{tower}(\ell, o(k)) \cdot \mathrm{poly}(|\mathsf{var}(F)|)$*, where* k *is the treewidth of the primal graph* P_Q*.*

The result for $\ell = 2$ (cf. [Lampis and Mitsou, 2017]) has already been applied as a strategy to show lower bound results for problems in artificial intelligence, as for example abstract argumentation, abduction, circumscription, and projected model counting, that are hard for the second level of the polynomial hierarchy when parameterized by treewidth [Fichte and Hecher, 2019; Fichte et al., 2018b; Lampis et al., 2018]. With the generalization to an *arbitrary* quantifier rank in Theorem 5.7, one can obtain lower bounds for variants of these problems and even more general problems on the third level or higher levels of the polynomial hierarchy.

The Methodology

This motivates our methodology to show lower bounds for problems parameterized by treewidth. To this end, we make use of a stricter notion of fpt-reductions as defined in Section 2.2, where the fpt-reduction is guaranteed to *linearly preserve* the parameter. Given functions $f, g : \mathbb{N} \to \mathbb{N}$, where g is *linear*, and recall the concept of an f-bounded fpt-reduction r using g. Recall that both functions f and g are in relation to the parameter of the original, i.e., given instance. Further, function f refers to the part of the runtime (bound) of r that is beyond polynomial in the instance size, and g refers to the parameter of the resulting instance. Then, for simplicity, we call r an *(f-bounded) fptl-reduction*. Recall decomposition-guided reductions of Chapter 4 and observe that these reductions can serve as an effective tool in order to construct such an f-bounded fptl-reduction, since it is often easy to see that g is indeed linear.

Next, we discuss the methodology for proving lower bounds of a problem P for treewidth consisting of the following three steps.

1. **Graph Representation:** Pick a graph representation $G(\mathcal{I})$ that can be constructed for an arbitrary instance $\mathcal{I}$ of the considered problem P in polynomial time.

2. **Establish Reduction:** Fix a suitable quantifier rank ℓ and let Q be an arbitrary QBF of this quantifier rank ℓ, where treewidth $k = \mathsf{tw}(\mathcal{G}_Q)$. Then, establish an f-bounded fptl-reduction from the arbitrary QBF Q, to an instance $\mathcal{I}$ of P parameterized by the treewidth of $G(\mathcal{I})$, where function $f : \mathbb{N} \to \mathbb{N}$ is such that $f(k)$ is in $\mathsf{tower}(\ell, o(k))$.

Intuitively, this fptl-reduction is required to be f-bounded, since otherwise one could already solve QSAT for Q within the reduction.

3. **Conclude lower bound:** Then, by applying Theorem 5.7 conclude that unless the ETH fails, an arbitrary instance $\mathcal{I}$ of problem P cannot be solved in time $\mathsf{tower}(\ell, o(k)) \cdot \mathrm{poly}(\|\mathcal{I}\|)$ where $k = \mathsf{tw}(G(\mathcal{I}))$.

We can generalize this to "non-canonical" lower bounds. To this end, one aims in Step 2 for a lower bound of the form $\mathsf{tower}\,(\ell, \Omega(g^{-1}(k)) \cdot \mathrm{poly}(\|\mathcal{I}\|)$ for some function $g : \mathbb{N} \to \mathbb{N}$ such that g^{-1} is well-defined, and $f(k)$ is in $\mathsf{tower}(\ell, o(g^{-1}(k)))$. Then, one needs to establish an f-bounded fpt-reduction using g accordingly, in order to conclude in Step 3 that under the ETH an arbitrary instance $\mathcal{I}$ of P cannot be solved in time $\mathsf{tower}(\ell, o(g^{-1}(k))) \cdot \mathrm{poly}(\|\mathcal{I}\|)$, where $k = \mathsf{tw}(G(\mathcal{I}))$.

With the help of this methodology one can show lower bounds $f(k)$ for certain problems P, parameterized by treewidth, by reducing from the canonical ℓ-QSAT problem parameterized by treewidth k as well. Thus, one avoids directly using ETH via tedious reductions from SAT, which involves problem-tailored constructions of instances of P whose treewidth is ℓ-fold logarithmic in the number of variables or clauses of the given Boolean formula.

Note that the methodology naturally extends to pathwidth, since the result of Theorem 5.7 easily extends to pathwidth by construction of our reduction R, which works for tree decompositions including the special case of path decompositions. Formal details on correctness have been already provided in Section 5.1.2 (cf. Corollary 5.15), which also discusses further consequences of Theorem 5.7.

Applications and Showcases

The proof of Theorem 6.1 below serves as an example for applying the methodology, showing that Theorem 5.7 also allows for quite general results on projection. Note that these bounds are tight under the ETH.

Theorem 6.1. *Given an open QBF of the form $Q = Q_1V_1.Q_2V_2.\ Q_3V_3 \cdots Q_\ell V_\ell.F$ where $\ell \geq 1$, and F is a 3-CNF formula (if $Q_\ell = \exists$), or a 3-DNF formula (if $Q_\ell = \forall$). Then, under the ETH,* PQSAT *is indeed harder than deciding validity of $Q[\iota]$ for any assignment $\iota : \mathsf{fvar}(Q) \to \{0,1\}$. In particular, assuming the ETH,* PQSAT *cannot be solved in time $\mathsf{tower}(\ell + 1, o(k)) \cdot \mathrm{poly}(|\mathsf{var}(F)|)$, where k is the pathwidth of the primal graph P_Q.*

Proof. Assume towards a contradiction that under the ETH one can solve projected model counting of Q in time $\mathsf{tower}(\ell+1, o(k)) \cdot \mathrm{poly}(|\mathsf{var}(F)|)$. In the following, we define an fptl-reduction r from QSAT to the decision variant PQSAT-at-least-u of PQSAT, where a given open QBF Q is a yes-instance if and only if the solution (count) to PQSAT of Q is at least u. More precisely, we transform a closed QBF $Q' = Q_0V_0.Q_1V_1.\ Q_2V_2.Q_3V_3 \cdots Q_\ell V_\ell.F$,

where k is the pathwidth of P_Q to an instance $Q = Q_1.V_1.Q_2V_2.Q_3V_3 \cdots Q_\ell V_\ell.F$ of PQSAT-at-least-u, where $\mathsf{fvar}(Q) = V_0$, and we set $u := 1$ if $Q_0 = \exists$ and $u := 2^{|V_0|}$, otherwise. The reduction is indeed correct, since Q' is a yes-instance of QSAT if and only if $Q = r(Q')$ is a yes-instance of PQSAT-at-least-u. Then, one can solve Q' of quantifier rank $\ell + 1$ in time $\mathsf{tower}(\ell + 1, o(k)) \cdot \mathrm{poly}(|\mathsf{var}(F)|)$, which contradicts Theorem 5.7 and Corollary 5.15. □

Corollary 6.2. *Under the ETH, an instance Q of the problem $\#\Sigma_\ell$SAT or $\#\Pi_\ell$SAT cannot be solved in time $\mathsf{tower}(\ell+1, o(k)) \cdot \mathrm{poly}(|\mathsf{var}(\mathsf{matrix}(Q))|)$, where k is the pathwidth of $\mathcal{G}_Q$.*

Next, we list further selected examples in order to demonstrate the applicability of our methodology. These results and further are listed in an overview of complexity results later, cf. Table 6.1. For corresponding problem definitions, we refer to the respective original sources given below.

Proposition 6.3 (cf. [Fichte and Hecher, 2019])**.** *Unless the ETH fails,* PASP *for given ASP program Π and a set $P \subseteq \mathsf{var}(\Pi)$ of projection variables cannot be solved in time $\mathsf{tower}(3, o(k)) \cdot \mathrm{poly}(|\Pi|)$, where k is the pathwidth of the primal graph of Π.*

Proof (Idea). Fptl-reduction from ∀∃∀-SAT to PASP, both parameterized by the pathwidth of its primal graph. □

Proposition 6.4 (cf. [Fichte et al., 2019a])**.** *Let $\mathcal{S} \in \{\text{pref}, \text{semi-st}, \text{stage}\}$ and F be an argumentation framework. Unless the ETH fails, we cannot solve the problem $\#\text{PCRED}_\mathcal{S}$ in time $\mathsf{tower}(3, o(k)) \cdot \mathrm{poly}(\|F\|)$ where k is the pathwidth of F (underlying graph).*

Proof (Idea). Fptl-reduction from ∀∃∀-SAT parameterized by pathwidth of primal graph, to $\#\text{PCRED}_{SEM}$ (parameterized by pathwidth of the underlying graph). □

Theorem 6.5 (cf. [Hecher et al., 2020a])**.** *Given an epistemic program Π and a variable $a \in \mathsf{var}(\Pi)$. Then, unless the ETH fails, deciding the problem* CANDIDATE WORLD VIEW CHECK *cannot be solved in time $\mathsf{tower}(3, o(k)) \cdot \mathrm{poly}(|\Pi|)$, and the problem* WORLD VIEW CHECK *for a cannot be solved in time $\mathsf{tower}(4, o(k)) \cdot \mathrm{poly}(|\Pi|)$, where k is the pathwidth of the primal graph of Π.*

Proof (Idea). Fptl-reduction from ∃∀∃-SAT, or ∃∀∃∀-SAT (parameterized by pathwidth of primal graph), respectively. Actually the reductions from the literature for showing Σ_3^P-hardness and Σ_4^P-hardness [Shen and Eiter, 2017] form fptl-reductions. □

6.2 Complexity Characterization for Treewidth

The methodology above for showing lower bounds, or that a problem is "at least as hard", gives rise to a natural classification of problems based on the runtime for treewidth, where for simplicity we focus here on the treewidth of the primal graph representation $\mathcal{G}_\mathcal{I}$ of a problem instance $\mathcal{I}$. Note, however, that one could also consider the treewidth of different (further) graph representations. In addition to hardness results and in order to characterize problems accordingly, one also requires results providing that a problem is "at most as hard" as a representative of some group or class of problems, thereby establishing upper bounds or membership for this class. Recall that indeed techniques for showing upper bounds involving treewidth were already provided in Chapters 3 and 4 and that decomposition-guided reductions therefore seem to form a suitable tool for showing membership.

Next, we combine these findings with the methodology above. This is done by grouping problems according to the required runtime for treewidth when solving them, which will result in classes of problems, referred to by *treewidth classes* below. To this end, we define the following class of problems for treewidth.

Class:	$\mathtt{TW}_1^k$
Definition:	Class $\mathtt{TW}_1^k$ is the set of all problems parameterized by treewidth such that every instance $\mathcal{I}$ of these problems can be solved in time $2^{\mathcal{O}(k)} \cdot \text{poly}(\|\mathcal{I}\|)$, where $k = \mathsf{tw}(\mathcal{G}_\mathcal{I})$

This class can be further generalized in order to obtain the following classes below. The definition of these classes is inspired by the work on general FPT runtime classes [Weyer, 2004; Downey et al., 2007] and these classes are therefore obviously contained in the broader class FPT. We define a class for every non-negative integer $i \in \mathbb{N}$ as follows:

Class:	$\mathtt{TW}_i^k$, for every $i \in \mathbb{N}$
Definition:	Class $\mathtt{TW}_i^k$ is the set of all problems parameterized by treewidth such that every instance $\mathcal{I}$ of these problems can be solved in time $\mathsf{tower}(i, \mathcal{O}(k)) \cdot \text{poly}(\|\mathcal{I}\|)$, where $k = \mathsf{tw}(\mathcal{G}_\mathcal{I})$

Observe that indeed $\mathtt{TW}_0^k = \mathrm{P}$. It is also immediate by the definition of these classes that for any $i \in \mathbb{N}$ we have $\mathtt{TW}_i^k \subseteq \mathtt{TW}_{i+1}^k$. Even further, under the exponential time hypothesis, the inclusions between those classes $\mathtt{TW}_i^k$ are strict. Consequently, we show next that these $\mathtt{TW}_i^k$ for $i \in \mathbb{N}$ form a strict hierarchy assuming the exponential time hypothesis sustains.

Proposition 6.6. *Let $i \in \mathbb{N}$. Then, under the ETH we have that $\mathtt{TW}_i^k \subsetneq \mathtt{TW}_{i+1}^k$.*

Proof. Assume towards a contradiction that the ETH holds and at the same time we have that $\mathtt{TW}_i^k \not\subseteq \mathtt{TW}_{i+1}^k$. Consequently, due to $\mathtt{TW}_i^k \subseteq \mathtt{TW}_{i+1}^k$, we have that $\mathtt{TW}_i^k = \mathtt{TW}_{i+1}^k$. However, this contradicts Theorem 5.7 for problem $(i+1)$-QSat, which is in $\mathtt{TW}_{i+1}^k$ (cf. Proposition 2.9), but under the ETH it is not in $\mathtt{TW}_i^k$ since any function in $\mathsf{tower}(i, \mathcal{O}(k))$ is also in the set $\mathsf{tower}(i+1, o(k))$ of functions. □

The concept of these classes can be further generalized to cover also "non-canonical" running times, which results in the following definition.

Class:	$\mathtt{TW}_i^f$, for every $i \in \mathbb{N}$ and function $f : \mathbb{N} \to \mathbb{N}$
Definition:	Class $\mathtt{TW}_i^f$ is the set of all problems parameterized by treewidth such that every instance $\mathcal{I}$ of these problems can be solved in time $\mathsf{tower}(i, \mathcal{O}(f(k)))\cdot\mathrm{poly}(\|\mathcal{I}\|)$, where $k = \mathsf{tw}(\mathcal{G}_\mathcal{I})$

This definition allows us to also cover classes like $\mathtt{TW}_1^{k\cdot\log(k)}$, which is, assuming the ETH, located between $\mathtt{TW}_1^k$ and $\mathtt{TW}_2^k$. Recall that problems of this class $\mathtt{TW}_1^{k\cdot\log(k)}$ are for example the problem Disjoint Paths [Scheffler, 1994], which is also hard for this class under the ETH, cf. Proposition 5.21. Therefore, it is quite unexpected that Disjoint Paths can be a member of any class $\mathtt{TW}_1^f$ with $f \in o(k \cdot \log(k))$. Similar, as already shown, for answer set programming the problem Asp restricted to normal or head-cycle-free programs is in this class (cf. Theorem 3.16), but it is also hard for this class under the ETH as established in Theorem 5.35. Observe that for two functions f, g and an integer $i \in \mathbb{N}$, if $f \in \Theta(g)$ the two problem classes $\mathtt{TW}_i^f$ and $\mathtt{TW}_i^g$ actually coincide, i.e., $\mathtt{TW}_i^f = \mathtt{TW}_i^g$.

Finally, in order to cover also problems that work with non-binary domains sizes d, where $d \in \mathbb{N}^+$, we further generalize the classes above. This results in the final and most general definition of our *treewidth classes* as follows.

Class:	*Treewidth class* $\mathtt{TW}_i^{d,f}$, for every $i \in \mathbb{N}$, $d \in \mathbb{N}^+$, and function $f : \mathbb{N} \to \mathbb{N}$
Definition:	Class $\mathtt{TW}_i^{d,f}$ is the set of all problems parameterized by treewidth such that every instance $\mathcal{I}$ of these problems can be solved in time $\mathsf{tower}(i, d^{\mathcal{O}(f(k))})\cdot\mathrm{poly}(\|\mathcal{I}\|)$, where $k = \mathsf{tw}(\mathcal{G}_\mathcal{I})$

Observe that for $i \in \mathbb{N}$ and a function f, we have that $\mathtt{TW}_i^{2,f} = \mathtt{TW}_{i+1}^{1,f} = \mathtt{TW}_{i+1}^f$.

Since the treewidth classes are now defined, we still need to formally state hardness and completeness for these classes as motivated by the methodology above.

Definition 6.7 (Completeness for Treewidth Classes)**.** *Given integers $i \in \mathbb{N}$, $d \in \mathbb{N}^+$, and a function $f : \mathbb{N} \to \mathbb{N}$. Then, a problem* P *is* hard *for the class $\mathtt{TW}_i^{d,f}$, if for an arbitrary instance $\mathcal{I}$ of this problem* P*, the problem cannot be solved in time $\mathsf{tower}(i, d^{o(f(\mathsf{tw}(\mathcal{G}_\mathcal{I})))}) \cdot \mathrm{poly}(\|\mathcal{I}\|)$, unless the ETH fails. If a problem is both in the class $\mathtt{TW}_i^{d,f}$, i.e., a member of $\mathtt{TW}_i^{d,f}$, and also hard for this class, we say that problem* P *is* complete *for $\mathtt{TW}_i^{d,f}$.*

Indeed, having both membership and hardness for a problem and a treewidth class $\mathtt{TW}_i^f$ leads to quite precise runtime results for treewidth.

Observation 6.8. *Let $i \in \mathbb{N}$ and $d \in \mathbb{N}^+$ be given integers, and let $f : \mathbb{N} \to \mathbb{N}$ be a function. Then, we have that unless the ETH fails, an arbitrary instance $\mathcal{I}$ of a problem being complete for the class $\mathtt{TW}_i^{d,f}$, requires runtime in $\mathsf{tower}(i, d^{\Theta(f(\mathsf{tw}(\mathcal{G}_\mathcal{I})))}) \cdot \mathrm{poly}(\|\mathcal{I}\|)$.*

Proof. The claim follows by the definition of class $\mathtt{TW}_i^f$ as well as Definition 6.7. □

The definitions and results above can be easily lifted to cover pathwidth as well.

Class:	*Pathwidth class* $\mathtt{PW}_i^{d,f}$, for every $i \in \mathbb{N}$, $d \in \mathbb{N}^+$, and function $f : \mathbb{N} \to \mathbb{N}$
Definition:	Class $\mathtt{PW}_i^{d,f}$ is the set of all problems parameterized by pathwidth such that every instance $\mathcal{I}$ of these problems can be solved in time $\mathsf{tower}(i, d^{\mathcal{O}(f(k))}) \cdot \mathrm{poly}(\|\mathcal{I}\|)$, where k is the pathwidth of $\mathsf{tw}(\mathcal{G}_\mathcal{I})$

Observe that membership results for treewidth immediately carry over to the smaller parameter pathwidth. Further, hardness results for pathwidth immediately carry over to the larger parameter treewidth.

Observation 6.9. *Given integers $i \in \mathbb{N}$, $d \in \mathbb{N}^+$, and a function $f : \mathbb{N} \to \mathbb{N}$. Then, a problem that is a member of treewidth class $\mathtt{TW}_i^{d,f}$ is also a member of pathwidth class $\mathtt{PW}_i^{d,f}$, i.e., $\mathtt{PW}_i^{d,f} \subseteq \mathtt{TW}_i^{d,f}$. Further, if a problem is hard for $\mathtt{PW}_i^{d,f}$ it is hard for $\mathtt{TW}_i^{d,f}$ as well. Consequently, if a problem is in $\mathtt{TW}_i^{d,f}$ and hard for $\mathtt{PW}_i^{d,f}$, this problem is complete for both $\mathtt{TW}_i^{d,f}$ and $\mathtt{PW}_i^{d,f}$.*

Proof. The result follows immediately by the fact that for any graph G we have that $\mathsf{tw}(G) \leq \mathsf{pw}(G)$ and that $\mathsf{pw}(G)$ is bounded by the pathwidth of G as well. □

Overview of Complexity Results for Treewidth

Table 6.1 gives a brief overview of selected problems and their respective runtime upper bounds as well as runtime lower bounds under the ETH. All the results given in this table form completeness results for respective treewidth and pathwidth classes, i.e., the given bounds are asymptotically tight under ETH. Concrete references to table entries of Table 6.1 are given in Footnotes[1–19] below.

[1] See: [Cygan et al., 2015].
[2] Consequence of the ETH [Impagliazzo et al., 2001].
[3] See: [Scheffler, 1994].
[4] See: [Lokshtanov et al., 2018].
[5] See: [Samer and Szeider, 2010].
[6] See: [Lampis et al., 2018].
[7] See: [Fichte et al., 2020c].
[8] See: [Fichte et al., 2018b].
[9] See: [Jakl et al., 2009].
[10] See: [Fichte and Hecher, 2018, 2019].
[11] See: [Fichte et al., 2021a].
[12] See: [Hecher et al., 2020a].
[13] See: [Fellows et al., 2011].
[14] See: [Marx and Mitsou, 2016].
[15] See: [Dvořák et al., 2012].
[16] See: [Fichte et al., 2019a].
[17] See: [Chen, 2004].
[18] See: [Capelli and Mengel, 2019] or [Fichte and Hecher, 2020], which elaborates on combining an established algorithm [Chen, 2004] with an approach for projected solution counting [Fichte et al., 2018b].
[19] See: [Fichte et al., 2020b].

Problem P	Completeness for $\mathtt{TW}_i^{d,f}$ and $\mathtt{PW}_i^{d,f}$				
	i	d	f	Membership	Hardness
Min Vertex Cover [Garey and Johnson, 1979]	1	1	k	$\triangle^{1}$	$\triangledown^{2}$
Min Dominating Set [Garey and Johnson, 1979]	1	1	k	$\triangle^{1}$	$\triangledown^{2}$
Max Independent Set [Garey and Johnson, 1979]	1	1	k	$\triangle^{1}$	$\triangledown^{2}$
Hamiltonian Cycle [Garey and Johnson, 1979]	1	1	k	$\triangle^{1}$	$\triangledown^{2}$
3-Colorability [Garey and Johnson, 1979]	1	1	k	$\triangle^{1}$	$\triangledown^{2}$
Disjoint Paths [Scheffler, 1994]	1	1	$k{\cdot}\log(k)$	$\triangle^{3}$	$\triangledown^{4}$
Sat, #Sat	1	1	k	$\triangle^{5}$	$\triangledown^{2}$
Circumscription [McCarthy, 1980]	2	1	k	$\triangle^{6}$	$\triangledown^{6}_{\mathtt{TW}}$,$\blacktriangledown^{7}$
Mus [Lampis et al., 2018]	2	1	k	$\triangle^{6}$	$\triangledown^{6}_{\mathtt{TW}}$,$\blacktriangledown^{7}$
Pap [Eiter and Gottlob, 1995b]	2	1	k	$\triangle^{6}$	$\triangledown^{6}_{\mathtt{TW}}$,$\blacktriangledown^{7}$
#∃Sat	2	1	k	$\triangle^{8}$	$\triangledown^{8}_{\mathtt{TW}}$,$\blacktriangledown^{7}$
Models, #Models	1	1	k	▲ Thm. 3.8	▼ Prop. 3.3
Tight Asp/#Asp	1	1	k	▲ Thm. 3.8	▼ Prop. 3.9
Supported Models/#Models	1	1	k	▲ Thm. 3.8	▼ Prop. 3.9
Normal/HCF Asp	1	1	$k{\cdot}\log(k)$	▲ Thm. 3.16	▼ Thm. 5.35
ι-Tight Asp (cf., Definition 4.24)	1	1	$k\cdot\log(\iota)$	▲ Thm. 4.27	▼ Corr. 4.28
Disjunctive Asp/#Asp	2	1	k	$\triangle^{9}$	▼ Thm. 5.18
Asp, #Asp	2	1	k	▲ Thm. 3.21	▼ Thm. 5.18
PAsp [Aziz, 2015; Fichte and Hecher, 2019]	3	1	k	$\triangle^{10}$	$\blacktriangledown^{10}$
Nonground Asp [Eiter et al., 2007], arity 3, domain size d	3	d	k	$\triangle^{11}$	$\blacktriangledown^{11}$
Cand. World View Check [Shen and Eiter, 2017]	3	1	k	$\triangle^{12}$	$\blacktriangledown^{12}$
World View Check [Shen and Eiter, 2017]	4	1	k	$\triangle^{12}$	$\blacktriangledown^{12}$
s-Choosability [Fellows et al., 2011], $s \geq 3$	2	1	k	$\triangle^{13}$	$\triangledown^{14}$
s-Choosability Deletion [Marx and Mitsou, 2016], $s \geq 4$	3	1	k	$\triangle^{14}$	$\triangledown^{14}$
$\text{Skep}_{\text{pref}}$, $\text{Skep}_{\text{semi-st}}$, $\text{Cred}_{\text{semi-st}}$ [Dung, 1995]	2	1	k	$\triangle^{15}$	$\triangledown^{16}_{\mathtt{TW}}$, $\blacktriangledown^{7}$
$\#\text{PCred}_{\mathcal{S}}$ [Fichte et al., 2019a], $\mathcal{S} \in \{\text{pref}, \text{semi-st}, \text{stage}\}$	3	1	k	$\triangle^{16}$	$\triangledown^{16}_{\mathtt{TW}}$, $\blacktriangledown^{7}$
ℓ-QSat, ℓ-#QSat [Gomes et al., 2009], $\ell \geq 1$	ℓ	1	k	$\triangle^{17}$	▼ Thm. 5.7
PQSat: $\#\Sigma_{\ell-1}\text{Sat}, \#\Pi_{\ell-1}\text{Sat}, \ell \geq 2$	ℓ	1	k	$\triangle^{18}$	▼ Corr. 6.2
$(\ell-1)$-QCSat [Dechter, 2006], $\ell \geq 2$, domain size d	ℓ	d	k	$\triangle^{17}$	$\blacktriangledown^{19}$

Table 6.1: Completeness results of selected problems for treewidth classes $\mathtt{TW}_i^{d,f}$ and pathwidth classes $\mathtt{PW}_i^{d,f}$, where k refers to the treewidth and the pathwidth of the primal graph, respectively. References of the results are provided in Footnotes[1–19]. Membership upper bounds are mostly known by the literature, which is indicated by "$\triangle$". New upper bounds established in the course of this thesis, i.e., Chapters 3 and 4, are indicated by "▲". Runtime lower bounds (under the ETH) are given under column "hardness" and k refers to the treewidth ("$\triangledown_{\mathtt{TW}}$"), pathwidth ("$\triangledown$" or "▼") of the corresponding (primal) graph of $\mathcal{I}$. Results known from the literature are marked by "$\triangledown_{\mathtt{TW}}$" and "$\triangledown$". By "▼", we indicate that the result holds due to lower bound advancements and the methodology described in Chapter 5. We obtain results for "▼", with known lower bound ("$\triangledown_{\mathtt{TW}}$", "$\triangledown$"), by the existing lower bound proof together with our methodology for pathwidth.

CHAPTER 7

Efficiently Implementing Treewidth-Aware Algorithms

Knowledge is not power; Implementation is power.

— Garrison Wynn

Recall dynamic programming on tree decompositions as one of the most prominent methods to utilize treewidth [Cygan et al., 2015; Downey and Fellows, 2013]. Besides theoretical work in this direction, which dates back to the early 70s [Bertelè and Brioschi, 1972, 1973; Bodlaender and Kloks, 1996; Flum and Grohe, 2006; Niedermeier, 2006], this technique has also been the focus of several practical implementations and empirical studies. Indeed, while for many problems parameterized by treewidth the best-known upper bounds and especially corresponding lower bounds might seem devastating (cf. also Chapters 3–6), there are several solvers exploiting treewidth. On the one hand, there are specialized solvers such as dynasp [Fichte et al., 2017b], dynQBF [Charwat and Woltran, 2017], gpuSAT [Fichte et al., 2018b, 2019b], and fvs-pace [Kiljan and Pilipczuk, 2018] that utilize treewidth supported by problem-specific implementations, techniques, and fine-tunings. Some of these parameterized solvers are particularly efficient for certain fragments [Lonsing and Egly, 2018a], and even successfully participated in problem-specific competitions [Pulina and Seidl, 2019]. However, on the other hand, the literature also distinguishes quite generic systems that exploit treewidth like D-FLAT [Bliem et al., 2016a], Jatatosk [Bannach and Berndt, 2019], and sequoia [Langer et al., 2012]. Independent of how these systems work internally, a crucial key ingredient is oftentimes the component or the software library that computes tree decompositions. Nowadays, it is possible to efficiently approximate treewidth [Bodlaender et al., 2016; Bodlaender, 1996; Feige et al., 2008] and to benefit from several efficient heuristics, see, e.g., [Abseher et al., 2017; Dell et al., 2017], which is also the result of dedicated competitions [Dell et al., 2017] for treewidth. This is indeed quite surprising, as for a given graph the computation of a tree decomposition of minimal width (treewidth) is NP-hard.

Despite these major practical progresses, systems that exploit treewidth naturally suffer from rather similar issues: for efficiently computing treewidth and tree decompositions [Abseher et al., 2017; Tamaki, 2019], these approaches based on dynamic programming reach their limits when instances have higher treewidth; a situation which can even occur in structured real-world instances [Maniu et al., 2019]. Nevertheless, in the area of Boolean satisfiability, this approach proved to be successful for counting problems, such as, e.g., (weighted) model counting [Fichte et al., 2019b; Samer and Szeider, 2010] and projected model counting [Fichte et al., 2018b]. To further increase the applicability of this paradigm, novel techniques are required which (1) rely on different levels of abstraction of the instance at hand; (2) treat subproblems originating in the abstraction by standard solvers whenever widths appear too high; and (3) use highly sophisticated data management in order to store and process tables obtained by dynamic programming.

In this chapter, we turn our focus to practically applying tree decompositions for efficient problem solving, where we treat the three above aspects as follows.

1. To tame the beast of high treewidth, based on recent work [Hecher et al., 2020b] we propose in Section 7.1 the concept of *nested dynamic programming*, where only parts of an abstraction of a graph are decomposed. Then, each tree decomposition node also needs to solve a *subproblem* residing in the graph, but may involve vertices outside the abstraction. In turn, for solving such subproblems, the idea of nested DP is to subsequently repeat decomposing and solving more fine-grained graph abstractions in a nested fashion. This results not only in elegant DP algorithms, but also allows to deal with high treewidth. While candidates for obtaining abstractions often originate naturally from the problem, nested DP may require non-obvious sub-abstractions, for which we present a generic solution.

2. To further improve the capability of handling high treewidth, we show how to apply nested dynamic programming in the context of hybrid solving, where established, standard solvers (e.g., Sat solvers) and caching are incorporated in nested dynamic programming such that the best of two worlds are combined. This leads to a technique called *hybrid dynamic programming* [Hecher et al., 2020b], which will be discussed in Section 7.2. Thereby, structured solving is applied to parts of the problem instance subject to counting or enumeration, while depending on results of subproblems. These subproblems (subject to search) reside in the abstraction only, and are solved via standard solvers.

3. Finally, in Section 7.3 we discuss how to apply database management systems (DBMS) for efficiently maintaining tables during dynamic programming. We implement a system called nestHDB that uses databases during hybrid dynamic programming, which follows recent ideas [Fichte et al., 2021b; Hecher et al., 2020b]. This system is presented in Section 7.4, thereby combining nested DP and standard solvers, where DBMS are applied for efficiently implementing the handling of tables needed by nested DP. Preliminary experiments of nestHDB indicate that nested DP with hybrid solving can be indeed fruitful.

Figure 7.1: Primal graph $\mathcal{G}_F$ of F from Example 3.1 (left), nested primal graph $\mathcal{G}_F^{\{a,b\}}$ (middle), as well as nested primal graph $\mathcal{G}_F^{\{c,d\}}$ (right).

We exemplify and detail these ideas along the canonical #Sat problem and projected model counting (#∃Sat), and we discuss adaptions required for solving other problems.

7.1 Abstractions as a Key for Nested Dynamic Programming

In the following, we discuss certain abstractions of the primal graph in the context of the Boolean satisfiability problem, namely for the problem #Sat. Afterwards we generalize the usage of these abstraction to nested dynamic programming for further problems.

To this end, let F be a Boolean formula. Now, assume the situation that a set U of variables of F, called *nesting variables*, appears *uniquely* in the bag of exactly one TD node t of a tree decomposition of $\mathcal{G}_F$. Then, observe that one could do dynamic programming on the tree decomposition as explained in Chapter 3, but no truth value for any variable in U requires to be stored. Instead, clauses involving U could be evaluated by nested DP within node t, since variables U appear uniquely in the node t. Indeed, for dynamic programming on the non-nesting (abstraction) variables, only the result of this evaluation is essential.

Before we can apply nested DP, we need abstractions with room for choosing nesting variables between the empty set and the set of all the variables. Let F be a Boolean formula and recall the primal graph $\mathcal{G}_F = (\mathsf{var}(F), E)$ of F, as defined in Section 2.5. Inspired by related work [Dell et al., 2019; Eiben et al., 2019; Ganian et al., 2017; Hecher et al., 2020a], we define the *nested primal graph* $\mathcal{G}_F^A$ for a given formula F and a given set $A \subseteq \mathsf{var}(F)$ of *abstraction variables*. To this end, we say a path P in primal graph $\mathcal{G}_F$ is a *nesting path (between u and v)* using A, if $P = u, v_1, \ldots, v_\ell, v$ ($\ell \geq 0$), and every vertex v_i is a *nesting variable*, i.e., $v_i \notin A$ for $1 \leq i \leq \ell$. Note that any path in $\mathcal{G}_F$ is nesting using A if $A = \emptyset$. Then, the vertices of nested primal graph $\mathcal{G}_F^A$ correspond to A and there is an edge between two distinct vertices $u, v \in A$ if there is a nesting path between u and v. Observe that the nested primal graph only consists of abstraction variables and, intuitively, "hides" nesting variables of nesting paths of primal graph $\mathcal{G}_F$. Even further, the connected components of $\mathcal{G}_F - A$ are hidden in the nested primal graph $\mathcal{G}_F^A$ by means of cliques among A.

Example 7.1. *Recall formula* $F := \{\overbrace{\{\neg a, b, c\}}^{c_1}, \overbrace{\{a, \neg b, \neg c\}}^{c_2}, \overbrace{\{a, d\}}^{c_3}, \overbrace{\{a, \neg d\}}^{c_4}\}$ *and primal graph $\mathcal{G}_F$ of Example 3.1, which is visualized in Figure 7.1 (left). Given abstraction variables A=$\{a, b\}$, nesting paths of $\mathcal{G}_F$ are, e.g., P_1=a, P_2=a, d, P_3=d, a, P_4=a, b,*

Figure 7.2: TD $\mathcal{T}$ (left) of the primal graph $\mathcal{G}_F$ of Figure 7.1, and a TD $\mathcal{T}'$ (right) of nested primal graph $\mathcal{G}_F^{\{a,b\}}$.

P_5=a, c, b. However, neither path P_6=b, a, d, nor path P_7=d, a, b, c is nesting using A. Nested primal graph $\mathcal{G}_F^A$ is shown in Figure 7.1 (middle) and it contains an edge $\{a, b\}$ over the vertices in A due to, e.g., paths P_4, P_5. Assume a different set $A' = \{c, d\}$. Observe that $\mathcal{G}_F^{A'}$ as depicted in Figure 7.1 (right) consists of the vertices A' and there is an edge between c and d due to, e.g., nesting path $P' = c, a, d$ using A'.

The nested primal graph provides abstractions of needed flexibility for nested DP. Indeed, if we set abstraction variables to A=$\mathsf{var}(F)$, we end up with full dynamic programming and zero nesting, whereas setting A=$\emptyset$ results in full nesting, i.e., nesting of all variables. Intuitively, the nested primal graph ensures that clauses subject to nesting (containing nesting variables) can be safely evaluated in exactly one node of a tree decomposition of the nested primal graph.

To formalize this, we assume a tree decomposition $\mathcal{T} = (T, \chi)$ of $\mathcal{G}_F^A$ and say a set $U \subseteq \mathsf{var}(F)$ of variables is *compatible* with a node t of T, and vice versa, if

(I) U is a connected component of the graph $\mathcal{G}_F - A$, which is obtained from primal graph $\mathcal{G}_F$ by removing A and

(II) all neighbor vertices of U that are in A are contained in $\chi(t)$, i.e., $\{a \mid a \in A, u \in U,$ there is a nesting path from a to u using $A\} \subseteq \chi(t)$.

If such a set $U \subseteq \mathsf{var}(F)$ of variables is compatible with a node of T, we say that U is a *compatible set*. By construction of the nested primal graph, any nesting variable is in at least one compatible set. However, a compatible set could be compatible with several nodes of T. Hence, to enable nested evaluation in general, we need to ensure that each nesting variable is evaluated only in one unique node t.

As a result, we formalize for every compatible set U, a *unique* node t of T that is compatible with U, denoted by $\mathrm{comp}_{F,A,\mathcal{T}}(U) := t$. We denote the union of all compatible sets U with $\mathrm{comp}_{F,A,\mathcal{T}}(U) = t$, by *nested bag variables* $\chi_t^A := \bigcup_{U:\mathrm{comp}_{F,A,\mathcal{T}}(U)=t} U$. Then, the *nested bag formula* F_t^A for a node t of T equals $F_t^A := \{c \mid c \in F, \mathsf{var}(c) \subseteq \chi(t) \cup \chi_t^A\} \setminus F_t$, where the canonical bag formula F_t is defined as in Section 2.5 above. Observe that the definition of nested bag formulas ensures that any connected component U of $\mathcal{G}_F - A$ "appears" among nested bag variables of some unique node of T. Consequently, each variable $a \in \mathsf{var}(F) \setminus A$ appears *only* in one nested bag formula F_t^A of a node t of T that is unique for a.

Listing 7.1: Algorithm $\texttt{NestDP}_{\mathsf{N}}(\text{depth}, \mathcal{I}, A, \mathcal{T})$ for computing solutions of $\mathcal{I}$ via nested DP on LTD $\mathcal{T}$.

In: Nested table algorithm N, nesting depth ≥ 0, a set A of vertices of $\mathcal{G}_{\mathcal{I}}$, and an LTD $\mathcal{T} = (T, \chi, \delta_{\mathcal{I}})$ of the nested primal graph $\mathcal{G}_{\mathcal{I}}^{A}$ of $\mathcal{I}$.

Out: Table mapping N-Tabs, which maps each TD node t of T to some computed table τ_t.

```
1 N-Tabs ← {}      /* empty mapping */
2 for iterate t in post-order(T) do
3 |   Child-Tabs ← ⟨N-Tabs[t_1], ..., N-Tabs[t_ℓ]⟩ where post-children(t) = ⟨t_1, ..., t_ℓ⟩
4 |   N-Tabs[t] ← N_t(depth, χ(t), I, δ_I(t), I_t^A, Child-Tabs)
5 return N-Tabs
```

Example 7.2. *Recall formula F, the tree decomposition $\mathcal{T} = (T, \chi)$ of $\mathcal{G}_F$, as depicted in Figure 7.2 (left), and abstraction variables $A = \{a, b\}$ of Example 7.1. Consider TD $\mathcal{T}' := (T, \chi')$, where $\chi'(t) := \chi(t) \cap \{a, b\}$ for each node t of T, which is given in Figure 7.2 (right). Observe that $\mathcal{T}'$ is $\mathcal{T}$, but restricted to A and that $\mathcal{T}'$ is a TD of $\mathcal{G}_F^A$ of width* 1*. There are two compatible sets, namely $\{c\}$ and $\{d\}$. Observe that only for compatible set $U = \{d\}$ we have two nodes compatible with U, namely t_2 and t_3. We assume that $\text{comp}_{F,A,\mathcal{T}'}(U) = t_2$, i.e., we decide that t_2 shall be the unique node for U. Consequently, nested bag formulas are $F_{t_1}^A = \{c_1, c_2\}$, $F_{t_2}^A = \{c_3, c_4\}$, and $F_{t_3}^A = \emptyset$.*

Nested Dynamic Programming

The notations above are by far not limited to problems related to Boolean satisfiability. Assume a given problem $\mathtt{P}$, an instance $\mathcal{I}$ of $\mathtt{P}$, and the primal graph $\mathcal{G}_{\mathcal{I}} = (V, E)$ of $\mathcal{I}$. Indeed, given a set $A \subseteq V$ of abstraction variables, it is straight forward to apply nesting paths in order to define the *nested primal graph* $\mathcal{G}_{\mathcal{I}}^{A}$ for any instance $\mathcal{I}$ of any problem $\mathtt{P}$. Further, assuming a given tree decomposition $\mathcal{T} = (T, \chi)$ of nested primal graph $\mathcal{G}_{\mathcal{I}}^{A}$ and a node t of T, we require a suitable, problem-specific definition of the *nested bag instance* $\mathcal{I}_t^A$, which lifts the concept of nested bag formulas from Boolean formulas to instances of problem $\mathtt{P}$.

Now, we have established required notation in order to discuss *nested dynamic programming (nested DP)*. Listing 7.1 presents algorithm `NestDP` for solving a given problem $\mathtt{P}$ by means of nested dynamic programming. Observe that Listing 7.1 is almost identical to algorithm `DP` as presented in Listing 3.1. The reason for this is that nested dynamic programming for $\mathtt{P}$ can be seen as a refinement of dynamic programming for $\mathtt{P}$, cf. algorithm `DP` of Listing 3.1. Indeed, the difference of `NestDP` compared to `DP` is that `NestDP` uses labeled tree decompositions of the nested primal graph and that it gets as additional parameter a set A of abstraction variables. Further, instead of a table algorithm A, algorithm `NestDP` relies on a *nested table algorithm* N during dynamic programming, which is similar to a table algorithm that gets as additional parameter an integer depth ≥ 0 that will be used later and a nested bag instance that needs to be evaluated. For simplicity and generality, also the instance $\mathcal{I}$ is passed as a parameter, which is, however, used only for passing problem-specific information of the instance.

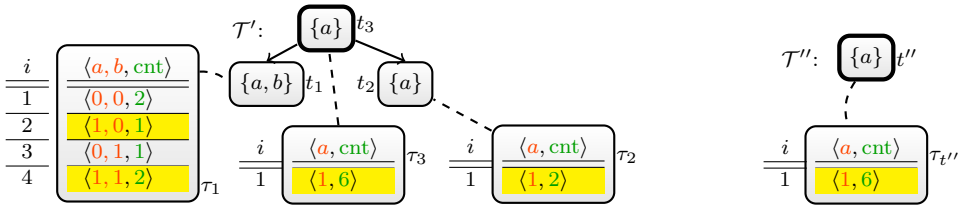

Figure 7.3: Selected tables obtained by nested DP on any LTD of TD $\mathcal{T}'$ of $\mathcal{G}_F^{\{a,b\}}$ (left) and on any LTD of TD $\mathcal{T}''$ of $\mathcal{G}_F^{\{a\}}$ (right) for F of Example 7.2 via $\texttt{NestDP}_{\mathsf{N\#Sat}}$.

Indeed, most nested table algorithm do not require this parameter, which should not be used for direct problem solving instead of utilizing the bag instance. Consequently, nested dynamic programming still follows the basic concept of dynamic programming as presented in Figure 3.1.

Similar to Chapter 3, for the ease of presentation our nested table algorithms use *nice labeled tree decompositions only*. Recall that this is not a hard restriction, cf. Proposition 2.6 and Observation 2.15. Indeed, it is easy to see that for arbitrary LTDs the clear case distinctions of nice decompositions are still valid, but are in general just overlapping. Further, without loss of generality we also assume that each compatible set U gets assigned a unique node $t = \text{comp}_{\mathcal{I},A,\mathcal{T}}(U)$ that is an *introduce node*, i.e., $\text{type}(t) = \textit{intr}$.

Nested Dynamic Programming for #Sat. In order to design a nested table algorithm for #Sat, assume a Boolean formula F as well as a given labeled tree decomposition $\mathcal{T} = (T, \chi, \delta_F)$ of $\mathcal{G}_F^A$ using any set A of abstraction variables. Recall from the discussions above, that each variable $a \in \mathsf{var}(F) \setminus A$ appears *only* in one nested bag formula F_t^A of a node t of T that is unique for a. These unique variable appearances allow us to actually nest the evaluation of nested bag formula F_t^A. This evaluation is performed by a nested table algorithm N#Sat in the context of nested dynamic programming. Listing 7.2 shows this simple nested table algorithm N#Sat for solving problem #Sat by means of algorithm $\texttt{NestDP}_{\mathsf{N\#Sat}}$. For comparison, recall table algorithm #Sat for solving problem #Sat by means of dynamic programming, as given by Listing 3.2. Observe that the main difference of N#Sat compared to #Sat is that the nested table algorithm N#Sat gets called on a nested primal graph and that it gets besides other parameters the nested bag formula as additional parameter. Then, the nested table algorithm evaluates this nested bag formula in Line 3 via any procedure #Sat for solving problem #Sat(F_t^A) on the nested bag formula F_t^A. Note that this subproblem #Sat(F_t^A) itself can be solved by again using nested dynamic programming with the help of algorithm $\texttt{NestDP}_{\mathsf{N\#Sat}}$.

In the following, we briefly show the evaluation of nested dynamic programming for #Sat on an example.

Example 7.3. *Recall formula F, set A of abstraction variables, and TD $\mathcal{T}'$ of nested primal graph $\mathcal{G}_F^A$ given in Example 7.2. Formula F has six satisfying assignments,*

Listing 7.2: Nested table algorithm $\mathsf{N}\#\mathsf{Sat}_t(\cdot, \chi_t, \cdot, F'_t, F^A_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving #SAT.

In: Node t, bag χ_t, bag formula F'_t, nested bag formula F^A_t, and sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables of t.
Out: Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t \leftarrow \{\langle \emptyset, 1 \rangle\}$
2 **else if** type$(t) =$ *intr, and* $a \in \chi(t)$ *is introduced* **then**
3 $\quad \tau_t \leftarrow \{\langle J, c' \cdot c \rangle \quad | \ \langle I, c \rangle \in \tau_1, J \in \{I^+_{a \mapsto 0}, I^+_{a \mapsto 1}\}, J \models F'_t, c'{>}0, c'{=}\#\text{SAT}(F^A_t[J])\}$
4 **else if** type$(t) =$ *rem, and* $a \notin \chi(t)$ *is removed* **then**
$\quad$ /* C(I) is the set that contains the rows in τ_1 for assignments J that are equal to I after removing a */
5 $\quad \text{C}(I) \leftarrow \{\langle J, c \rangle \mid \langle J, c \rangle \in \tau_1, J \setminus \{a \mapsto 0, a \mapsto 1\} = I \setminus \{a \mapsto 0, a \mapsto 1\}\}$
6 $\quad \tau_t \leftarrow \{\langle I \setminus \{a \mapsto 0, a \mapsto 1\}, \sum_{\langle J, c \rangle \in C(I)} c \rangle \} \mid \langle I, \cdot \rangle \in \tau_1\}$
7 **else if** type$(t) =$ *join* **then**
8 $\quad \tau_t \leftarrow \{\langle I, c_1 \cdot c_2 \rangle \quad | \ \langle I, c_1 \rangle \in \tau_1, \langle I, c_2 \rangle \in \tau_2\}$
9 **return** τ_t

namely $\{a \mapsto 1, b \mapsto 0, c \mapsto 1, d \mapsto 0\}$, $\{a \mapsto 1, b \mapsto 0, c \mapsto 1, d \mapsto 1\}$, $\{a \mapsto 1, b \mapsto 1, c \mapsto 0, d \mapsto 0\}$, $\{a \mapsto 1, b \mapsto 1, c \mapsto 0, d \mapsto 1\}$, $\{a \mapsto 1, b \mapsto 1, c \mapsto 1, d \mapsto 0\}$, *and* $\{a \mapsto 1, b \mapsto 1, c \mapsto 1, d \mapsto 1\}$.

Figure 7.3 (left) shows TD $\mathcal{T}'$ *of* $\mathcal{G}^A_F$ *and tables obtained by* $\texttt{NestDP}_{\mathsf{N}\#\mathsf{Sat}}(0, F, A, \mathcal{T}^*)$ *for model counting (*#SAT*) on* F*, where* $\mathcal{T}^*$ *is any arbitrary LTD of TD* $\mathcal{T}'$*. Observe that the canonical bag formula for every node of* $\mathcal{T}'$ *is empty and therefore we can indeed use any LTD* $\mathcal{T}^*$ *of TD* $\mathcal{T}'$ *and end up with the same result. We briefly discuss executing* #Sat *on* $\mathcal{T}^*$*, resulting in tables* τ_1*,* τ_2*, and* τ_3 *as shown in Figure 7.3 (left). Intuitively, table* τ_1 *is the result of introducing variables* a *and* b*. Recall from Example 7.2 that* $F^A_{t_1} = \{c_1, c_2\}$ *with* $c_1 = \{\neg a, b, c\}$ *and* $c_2 = \{a, \neg b, \neg c\}$*. Then, in Line 3 of algorithm* N#Sat*, for each assignment* I *to* $\{a, b\}$ *of each row* r *of* τ_1*, we compute* $\#\text{SAT}(F^A_{t_1}[I])$*. Consequently, for assignment* $I_1 = \{a \mapsto 0, b \mapsto 0\}$*, we have that there are two satisfying assignments of* $F^A_{t_1}[I_1]$*, namely* $\{c \mapsto 0\}$ *and* $\{c \mapsto 1\}$*. Indeed, this count of* 2 *is obtained for the first row of table* τ_1 *by Line 3. Analogously, one can derive the remaining tables of* τ_1 *and one obtains table* τ_2 *similarly, by using formula* $F^A_{t_2}$*. Then, table* τ_3 *is the result of removing* b *in node* t_1 *and combining agreeing assignments of rows accordingly. Consequently, we obtain that there are six satisfying assignments of* F*, which are all required to set* a *to* 1 *due to formula* $F^A_{t_2}$ *that is evaluated in node* t_2*.*

Figure 7.3 (right) shows TD $\mathcal{T}''$ *of* $\mathcal{G}^{\{a\}}_F$ *and tables obtained by* $\texttt{NestDP}_{\mathsf{N}\#\mathsf{Sat}}(0, F, \{a\}, \mathcal{T}^{**})$ *using any LTD* $\mathcal{T}^{**}$ *of TD* $\mathcal{T}''$*. Since* $F^{\{a\}}_{t''} = F$ *and* $\{a \mapsto 0\} \not\models F$*, table* $\tau_{t''}$ *does not contain an entry corresponding to assignment* $\{a \mapsto 0\}$*, cf. Condition "*$c'{>}0$*" in Line 3 of Listing 7.2. Thus, there are six satisfying assignments of* $F^{\{a\}}_{t''}[\{a \mapsto 1\}]$ *obtained by computing* $\#\text{SAT}(F^{\{a\}}_{t''}[\{a \mapsto 1\}])$*.*

While the overall concept of nested dynamic programming as given by algorithm NestDP of Listing 7.1 is quite general, sometimes in practice it is sufficient to further restrict the set of choices for abstraction vertices A when constructing the nested primal graph.

Listing 7.3: Nested table algorithm $\mathsf{N}\#\exists\mathsf{Sat}_t(\cdot, \chi_t, \cdot, F'_t, Q^A_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving problem #∃Sat.

In: Node t, bag χ_t, bag formula F'_t, nested bag QBF $Q^A_t = \exists V.F^A_t$, and sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables of t.
Out: Table τ_t.

1 **if** type(t) = *leaf* **then** $\tau_t \leftarrow \{\langle \emptyset, 1 \rangle\}$
2 **else if** type(t) = *intr, and* $a \in \chi(t)$ *is introduced* **then**
3 $\quad \tau_t \leftarrow \{\langle J, c' \cdot c \rangle$
$\quad\quad \mid \langle I, c \rangle \in \tau_1, J \in \{I^+_{a \mapsto 0}, I^+_{a \mapsto 1}\}, J \models F'_t, c'{>}0, c'{=}\#\exists\mathrm{Sat}(Q^A_t[J])\}$
4 **else if** type(t) = *rem, and* $a \notin \chi(t)$ *is removed* **then**
$\quad$ `/* C(I) is the set that contains the rows in` τ_1 `for assignments`
$\quad\quad$ `J that are equal to` I `after removing` a `*/`
5 $\quad \mathrm{C}(I) \leftarrow \{\langle J, c \rangle \mid \langle J, c \rangle \in \tau_1, J \setminus \{a \mapsto 0, a \mapsto 1\} = I \setminus \{a \mapsto 0, a \mapsto 1\}\}$
6 $\quad \tau_t \leftarrow \{\langle I \setminus \{a \mapsto 0, a \mapsto 1\}, \sum_{\langle J,c \rangle \in C(I)} c \rangle \mid \langle I, \cdot \rangle \in \tau_1\}$
7 **else if** type(t) = *join* **then**
8 $\quad \tau_t \leftarrow \{\langle I, c_1 \cdot c_2 \rangle \mid \langle I, c_1 \rangle \in \tau_1, \langle I, c_2 \rangle \in \tau_2\}$
9 **return** τ_t

Nested Table Algorithm for #∃Sat. To this end, we show the approach of nested dynamic programming for the problem #∃Sat.

Example 7.4. *Recall formula F as well as set $A = \{a, b\}$ of abstraction variables from Example 7.2. Then, we have that $Q := \exists V.F$ with $V := \{c, d\}$ is an instance of the projected model counting problem* #∃Sat. *Sometimes we refer to $A = \mathsf{fvar}(Q)$ by a set of* projection variables *or* projection set. *Restricted to projection set A, the Boolean formula F has two satisfying assignments, namely $\{a \mapsto 1, b \mapsto 0\}$ and $\{a \mapsto 1, b \mapsto 1\}$. Consequently, the solution to* #∃Sat *on Q, i.e.,* #∃Sat(Q), *is* 2.

Indeed, for solving projected model counting we mainly focus on the case, where for a given instance $Q = \exists V.F$ with Boolean formula F of problem #∃Sat, the abstraction variables A that are used for constructing the *nested primal graph* $\mathcal{G}^A_Q := \mathcal{G}^A_F$ are *among the projection variables*, i.e., $A \subseteq \mathsf{fvar}(Q)$. The approach of nested DP can then be applied for solving projected model counting such that the nested table algorithm naturally extends algorithm N#Sat of Listing 7.2. To this end, we let for Q with $A \subseteq \mathsf{fvar}(Q)$ the *nested bag QBF* $Q^A_t := \exists(V \cap \mathsf{var}(F^A_t)).F^A_t$ with nested bag formula F^A_t being defined as above. Note that if $V \cap \mathsf{var}(F^A_t) = \emptyset$, we assume by slight abuse of notation that $Q^A_t := F^A_t$.

Example 7.5. *Consider again our instance $Q = \exists V.F$ from above, where formula F, abstraction variables A, as well as TD $\mathcal{T}'$ of nested primal graph $\mathcal{G}^A_F$ are given in Example 7.2. Then, we have that the nested bag QBF $Q^A_{t_1} = \exists c.F^A_{t_1}$ and $Q^A_{t_2} = \exists d.F^A_{t_2}$. Finally, observe that $Q^A_{t_3} = F^A_{t_3} = \emptyset$.*

Then, the nested table algorithm N#∃Sat for solving projected model counting via nested dynamic programming is presented in Listing 7.3. Observe that nested table algorithm N#∃Sat does not significantly differ from algorithm N#Sat due to $A \subseteq \mathsf{fvar}(Q)$.

Listing 7.4: Algorithm $\mathtt{HybDP}_{\mathsf{H}\#\exists\mathsf{Sat}}(\mathrm{depth}, Q)$ for hybrid DP of #∃SAT based on nested DP.

In: Nesting depth ≥ 0 and a quantified Boolean formula $Q = \exists V.F$ with F being Boolean.
Out: Number $\#\exists\mathrm{SAT}(Q)$ of assignments.

```
1  P ← fvar(Q)   /*Original projection variables*/
2  Q' = ∃V'.F' ← BCP_And_Preprocessing(Q)
3  P' ← fvar(Q')   /*Projection variables after preprocessing*/
4  A ← P'
5  if F' ∈ dom(cache)   /*Cache Hit occurred*/      then return cache(F') · 2^|P\P'|
6  if P' = ∅ then return SAT(F') · 2^|P|
7  T = (T, χ, δ_Q') ← Decompose_via_Heuristics(G^A_F')      /* Decompose */
8  if width(T) ≥ threshold_hybrid or depth ≥ threshold_depth /* Standard Solver */ then
9  |  if var(F') = P' then cache ← cache∪{(F', #SAT(F'))}
10 |  else                cache ← cache∪{(F', #∃SAT(Q'))}
11 |  return cache(F') · 2^|P\P'|
12 if width(T) ≥ threshold_abstr /* Abstract via Heuristics & Decompose*/ then
13 |  A      ← Choose_Subset_via_Heuristics(A, F')
14 |  T = (T, χ, δ_Q') ← Decompose_via_Heuristics(G^A_F')
15 N-Tabs ← NestDP_{H#∃Sat}(depth, Q', A, T)   /* Nested Dynamic Programming */
16 cache ← cache∪{(F', c) | ⟨∅, c⟩ ∈ N-Tabs[root(T)]}
17 return cache(F') · 2^|P\P'|
```

Indeed, the main difference is only in Line 3 of Listing 7.2, where instead of a procedure for model counting, a procedure #∃SAT for solving a projected model counting question is called. Note that one can adapt nested table algorithm N#∃Sat of Listing 7.2 to the case where $A \not\subseteq \mathsf{fvar}(Q)$. However, this requires to use an algorithm that is more involved than algorithm $\mathtt{NestDP}_{\mathsf{N}\#\exists\mathsf{Sat}}$ when computing precise projected model counts, which is detailed in related work [Fichte et al., 2018b].

7.2 Refining Nested DP – Towards Hybrid Dynamic Programming

Now, we have definitions at hand to further refine and discuss nested dynamic programming in the context of *hybrid dynamic programming (hybrid DP)*, which combines using both standard solvers and parameterized solvers exploiting treewidth in the form of nested dynamic programming. We first illustrate the ideas for the problem #∃SAT in Section 7.2.1 and then we show how to apply hybrid dynamic programming to other problems in Section 7.2.2. After discussing in Section 7.3 how to use database management systems for these algorithms, we finally present a concrete implementation of this approach in Section 7.4.

7.2.1 Hybrid Solving based on Nested DP

Listing 7.4 depicts our algorithm $\texttt{HybDP}_{\mathsf{H}\#\exists\mathsf{Sat}}$ for solving projected model counting, i.e., problem #∃Sat. This algorithm $\texttt{HybDP}_{\mathsf{H}\#\exists\mathsf{Sat}}$ takes a quantified Boolean formula $Q = \exists V.F$ consisting of Boolean formula F and projection variables $P = \mathsf{fvar}(Q)$. The algorithm maintains a global, but simple cache mapping a formula to an integer, and consists of the following four subsequent blocks of code, which are separated by empty lines: (1) Preprocessing & Cache Consolidation, (2) Standard Solving, (3) Abstraction & Decomposition, and (4) Nested Dynamic Programming, which causes an indirect recursion through nested table algorithm H#∃Sat, as discussed later.

Block (1) spans Lines 1-5 and performs simple preprocessing techniques like *Boolean conflict propagation (BCP)*, thereby obtaining a simplified instance $Q' = \exists V'.F'$ and obtaining updated projection variables $P' \subseteq P$. Then, in Line 4, we set the set A of abstraction variables to P', and consolidate cache with the updated formula F'. Note that the operations in Line 2 are required to return a simplified instance that preserves satisfying assignments of the original formula when restricted to P. If F' is not cached, in Block (2), we do standard solving if the width is out-of-reach for nested DP, which spans over Lines 6-11. More precisely, if the updated formula F' does not contain projection variables, in Line 6 we employ a Sat solver returning integer 1 or 0. If F' contains projection variables and either the width obtained by heuristically decomposing $\mathcal{G}_{F'}$ is above $\text{threshold}_{\text{hybrid}}$, or the nesting depth exceeds $\text{threshold}_{\text{depth}}$, we use a standard #Sat or #∃Sat solver depending on P'.

Block (3) spans Lines 12-14 and is reached if no cache entry was found in Block (1) and standard solving was skipped in Block (2). If the width of the computed decomposition is above $\text{threshold}_{\text{abstr}}$, we need to use an abstraction in form of the nested primal graph. This is achieved by choosing suitable subsets $E \subseteq A$ of abstraction variables and decomposing ${F'}_t^E$ heuristically.

Finally, Block (4) concerns nested DP, cf. Lines 15-17. This block relies on nested table algorithm H#∃Sat, which is given in Listing 7.5 that is almost identical to nested table algorithm N#∃Sat as already discussed above and given in Listing 7.3. The only difference of H#∃Sat compared to N#∃Sat is that in Line 3 the nested table algorithm H#∃Sat uses the parameter depth and recursively executes algorithm $\texttt{HybDP}_{\mathsf{H}\#\exists\mathsf{Sat}}$ on the increased nesting depth of depth +1, and the same formula as the one used in the generic #∃Sat oracle call in Line 3 of Listing 7.3.

As a result, our approach deals with high treewidth by recursively finding and decomposing abstractions of the graph. If the treewidth is too high for some parts, tree decompositions of abstractions are used to guide standard solvers. Towards defining an actual implementation for practical solving, one still needs to find values for the threshold constants $\text{threshold}_{\text{hybrid}}$, $\text{threshold}_{\text{depth}}$, and $\text{threshold}_{\text{abstr}}$. The actual values of these constants will be made more precisely later in Section 7.4 when discussing our implementation and experiments.

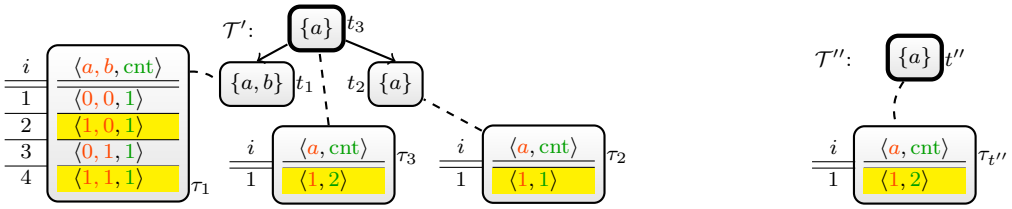

Figure 7.4: Selected tables obtained by nested DP using $\mathtt{NestDP}_{\mathsf{H}\#\exists\mathsf{Sat}}$ on any LTD of TD $\mathcal{T}'$ of $\mathcal{G}_Q^{\{a,b\}}$ (left) and on any LTD of TD $\mathcal{T}''$ of $\mathcal{G}_Q^{\{a\}}$ (right) for instance $Q = \exists c, d.F$ of Example 7.4.

Listing 7.5: Nested table algorithm $\mathsf{H}\#\exists\mathsf{Sat}_t(\text{depth}, \chi_t, \cdot, F'_t, Q_t^A, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving #∃SAT.

In: Node t, nesting depth ≥ 0, bag χ_t, bag formula F'_t, nested bag QBF $Q_t^A = \exists V.F_t^A$, and sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables of t.
Out: Table τ_t.

1 **if** type(t) = *leaf* **then** $\tau_t \leftarrow \{\langle \emptyset, 1\rangle\}$
2 **else if** type(t) = *intr, and* $a \in \chi(t)$ *is introduced* **then**
3 $\quad \tau_t \leftarrow \{\langle J, c' \cdot c\rangle \mid \langle I, c\rangle \in \tau_1, J \in \{I^+_{a\mapsto 0}, I^+_{a\mapsto 1}\}, J \models F'_t, c' > 0, c' = \mathtt{NestDP}_{\mathsf{H}\#\exists\mathsf{Sat}}(\text{depth}+1, Q_t^A[J])\}$
4 **else if** type(t) = *rem, and* $a \notin \chi(t)$ *is removed* **then**
$\quad$ /* C(I) is the set that contains the rows in τ_1 for assignments J that are equal to I after removing a */
5 $\quad \text{C}(I) \leftarrow \{\langle J, c\rangle \mid \langle J, c\rangle \in \tau_1, J \setminus \{a \mapsto 0, a \mapsto 1\} = I \setminus \{a \mapsto 0, a \mapsto 1\}\}$
6 $\quad \tau_t \leftarrow \{\langle I \setminus \{a \mapsto 0, a \mapsto 1\}, \sum_{\langle J,c\rangle \in C(I)} c\rangle \mid \langle I, \cdot\rangle \in \tau_1\}$
7 **else if** type(t) = *join* **then**
8 $\quad \tau_t \leftarrow \{\langle I, c_1 \cdot c_2\rangle \mid \langle I, c_1\rangle \in \tau_1, \langle I, c_2\rangle \in \tau_2\}$
9 **return** τ_t

Example 7.6. *Recall QBF $Q = \exists V.F$ of Example 7.4, and set A of abstraction variables as well as TD $\mathcal{T}'$ of nested primal graph $\mathcal{G}_F^A$ as given in Example 7.2. Further, recall that restricted to projection set A, F has two satisfying assignments. Figure 7.4 (left) shows TD $\mathcal{T}'$ of $\mathcal{G}_Q^A = \mathcal{G}_F^A$ and tables obtained by $\mathtt{NestDP}_{\mathsf{H}\#\exists\mathsf{Sat}}(0, Q, A, \mathcal{T}^*)$ for solving projected model counting on Q, where $\mathcal{T}^*$ is any LTD of $\mathcal{T}'$.*

Note that nested table algorithm $\mathsf{H}\#\exists\mathsf{Sat}$ of Listing 7.5 works similar to the nested table algorithm $\mathsf{N}\#\exists\mathsf{Sat}$ of Listing 7.3, but it calls $\mathtt{HybDP}_{\mathsf{H}\#\exists\mathsf{Sat}}$ recursively. We briefly discuss executing $\mathsf{H}\#\exists\mathsf{Sat}_{t_1}$ in the context of Line 15 of algorithm $\mathtt{HybDP}_{\mathsf{H}\#\exists\mathsf{Sat}}$ on node t_1, resulting in table τ_1 as shown in Figure 7.4 (left). Recall that $F_{t_1}^A = \{\{\neg a, b, c\}, \{a, \neg b, \neg c\}\}$. Then, in Line 3 of algorithm $\mathsf{H}\#\exists\mathsf{Sat}$, for each assignment J to $\{a, b\}$ of each row of τ_1, we compute $\mathtt{HybDP}_{\mathsf{H}\#\exists\mathsf{Sat}}(\text{depth}+1, \exists c.F_{t_1}^A[J])$. Each of these recursive calls, however, is already solved by Boolean conflict propagation (BCP) and preprocessing, e.g., $F_{t_1}^A[\{a \mapsto 1, b \mapsto 0\}]$ of Row 2 simplifies to $\{\{c\}\}$.

Figure 7.4 (right) shows TD $\mathcal{T}''$ of $\mathcal{G}_Q^E$ with $E := \{a\}$, and tables obtained by algorithm $\mathtt{NestDP}_{\mathsf{H}\#\exists\mathsf{Sat}}(0, Q, E, \mathcal{T}^{**})$, where $\mathcal{T}^{**}$ is any LTD of $\mathcal{T}''$. Still, $F_{t''}^E[J]$ for a given

Listing 7.6: Algorithm $\texttt{HybDP}_{\mathsf{HQSAT}}(\text{depth}, Q)$ for hybrid solving of QSAT by nested DP.

In: Nesting depth ≥ 0 and quantified Boolean formula $Q = \exists Q_1 V_1 . H$ with H being a QBF.
Out: 1 if Q is a positive instance (yes instance) of QSAT, 0 otherwise.

```
1  Q' ← BCP_And_Preprocessing(Q)
2  A ← V_1
3  if Q' ∈ dom(cache)  /*Cache Hit occurred*/   then return cache(Q')
4  T = (T, χ, δ_Q') ← Decompose_via_Heuristics(G^A_Q')   /* Decompose */
5  if width(T) ≥ threshold_hybrid or depth ≥ threshold_depth /* Standard Solver */ then
6  |  cache ← cache ∪ {(Q', QSAT(Q'))}
7  |  return cache(Q')
8  if width(T) ≥ threshold_abstr /* Abstract via Heuristics & Decompose*/ then
9  |  A  ← Choose_Subset_via_Heuristics(A, Q')
10 |  T = (T, χ, δ_Q') ← Decompose_via_Heuristics(G^A_Q')
11 N-Tabs ← NestDP_HQSAT(depth, Q', A, T)   /* Nested Dynamic Programming */
12 cache ← cache ∪ {(Q', 1) | N-Tabs[root(T)] ≠ ∅} ∪ {(Q', 0) | N-Tabs[root(T)] = ∅}
13 return cache(Q')
```

assignment J to $\{a\}$ of any row $r \in \tau_{t''}$ can be simplified. Concretely, $F^E_{t''}[\{a \mapsto 0\}]$ evaluates to $\emptyset$ and $F^E_{t''}[\{a \mapsto 1\}]$ evaluates to clause $\{b, c\}$. Thus, there are 2 satisfying assignments $\{b \mapsto 0\}$, $\{b \mapsto 1\}$ of $\exists c.F^E_{t''}[\{a \mapsto 1\}]$ restricted to A.

7.2.2 Applying Hybrid Dynamic Programming for Variants of SAT

Hybrid dynamic programming as proposed above is by far not restricted to (projected) model counting, or counting problems in general. In fact, one can easily generalize this approach further to other relevant problems, which is briefly sketched for deciding the validity of quantified Boolean formulas (QBFs).

Quantified Boolean Formulas (QBFs). We assume QBFs of the form

$$Q = Q_1 V_1 . Q_2 V_2 . \ldots Q_\ell V_\ell . F$$

using quantifiers $\exists, \forall$, where F is a CNF formula and $\mathsf{var}(Q) = \mathsf{var}(F) = V_1 \cup V_2 \cdots \cup V_\ell$. Hybrid solving by nested DP can be extended to problem QSAT. To the end of using this approach for QBFs, let $A \subseteq \mathsf{var}(Q)$ be a set of abstraction variables, and $\mathcal{T} = (T, \chi)$ be a TD of $\mathcal{G}^A_Q$ and t be a node of T. Then, the *nested bag QBF* Q^A_t for a set $A \subseteq \mathsf{var}(Q)$ amounts to $Q^A_t := Q_1 V'_1 . Q_2 V'_2 . \ldots Q_\ell V'_\ell . F^A_t$ with V'_i for $1 \leq i \leq \ell$ being $V'_i := V_i \cap \mathsf{var}(Q)$. For simplicity, we assume that every quantifier Q_i of Q^A_t is removed in case of $V'_i = \emptyset$.

The algorithm $\texttt{HybDP}_{\mathsf{HQSAT}}$ of Listing 7.6 is similar to $\texttt{HybDP}_{\#\exists\mathrm{SAT}}$ of Listing 7.4, where projection variables are not used and, initially, abstraction variables A coincide with the variables of the outermost quantifier of Q after preprocessing, cf. Line 2 of Listing 7.6. Further, this algorithm is adapted to QSAT, where, in particular Line 6 calls a QSAT

Listing 7.7: Nested table algorithm $\mathsf{HQSAT}_t(\text{depth}, \chi_t, Q, F'_t, Q^A_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving QSAT.

In: Node t, nesting depth ≥ 0, QBF $Q = Q_1 V_1 . H$, bag χ_t, bag formula F'_t, nested bag QBF Q^A_t, and sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables of t.
Out: Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t \leftarrow \{\langle \emptyset \rangle\}$
2 **else if** type$(t) =$ *intr, and* $a \in \chi(t)$ *is introduced* **then**
3 $\quad \tau_t \leftarrow \{\langle J \rangle \quad | \ \langle I \rangle \in \tau_1, J \in \{I^+_{a \mapsto 0}, I^+_{a \mapsto 1}\}, J \models F'_t, \mathtt{NestDP}_{\mathsf{HQSAT}}(\text{depth}+1, Q^A_t[J]) = 1\}$
4 $\quad$ **if** $Q_1 = \forall$ *and* $|\tau_t| \neq 2^{|\chi_t|}$ **then** $\tau_t \leftarrow \emptyset$
5 **else if** type$(t) =$ *rem, and* $a \notin \chi(t)$ *is removed* **then**
6 $\quad \tau_t \leftarrow \{\langle I \setminus \{a \mapsto 0, a \mapsto 1\} \rangle \quad | \ \langle I \rangle \in \tau_1\}$
7 **else if** type$(t) =$ *join* **then**
8 $\quad \tau_t \leftarrow \{\langle I \rangle \quad | \ \langle I \rangle \in \tau_1, \langle I \rangle \in \tau_2\}$
9 **return** τ_t

solver in Line 11 we rely on a fresh nested table algorithm HQSAT as presented in Listing 7.7.

This nested table algorithm HQSAT is depicted in Listing 7.7 and it is of similar shape as algorithm #∃Sat, cf. Listing 7.5, but does not maintain counters. Further, Line 4 of algorithm HQSAT intuitively filters τ_t fulfilling the outer-most quantifier, and keeps those rows r of τ_t, where the recursive call to $\mathtt{HybDP}_{\mathsf{HQSAT}}$ on nested bag formula simplified by the assignment J of r succeeds. For ensuring that the outer-most quantifier Q_1 is fulfilled, we are either in the situation that $Q_1 = \exists$, which immediately is fulfilled for every row r in τ_t since r itself serves as a witness. If $Q_1 = \forall$, we need to check that τ_t contains $2^{|\chi(t)|}$ many rows, i.e., all rows of the current bag. Notably, if $Q_1 = \forall$, we do not need to check in Line 8 of Listing 7.7, whether all rows sustain in table τ_t since this is already ensured for both child tables τ_1, τ_2 of t.

Finally, if in the end the computed table for the root node of $\mathcal{T}$ is not empty, it is guaranteed that either the table contains some (if $Q_1 = \exists$) or all (if $Q_1 = \forall$) rows and that Q is valid. Note that this has to be checked by algorithm $\mathtt{HybDP}_{\mathsf{HQSAT}}$, cf. Line 12 of Listing 7.6.

Hybrid Dynamic Programming Beyond QSAT. Note that the algorithm HQSAT above can be extended to also consider more fine-grained quantifier dependency schemes. Further, by combining ideas of this subsection and the previous subsection, one can easily design an algorithm based on hybrid dynamic programming for projected solution counting over QBFs, i.e., problem PQSAT. Indeed, such a hybrid DP algorithm needs to set abstraction variables as in Line 4 of Listing 7.4 in case of non-empty free variables, and set those variables to the variables of the outermost quantifier, as in Line 2 of Listing 7.6, otherwise. A corresponding nested table algorithm for solving PQSAT requires to use counters as in Listing 7.5, but also needs to consider the outer-most quantifier in case the free variables of the QBF are empty, as in Line 4 of Listing 7.7.

Compared to other algorithms for QSAT using treewidth [Charwat and Woltran, 2017;

Chen, 2004], nested DP and hybrid DP is quite compact without the need of nested tables. Instead of rather involved data structures (nested tables), we use here plain tables that can be handled by modern database systems efficiently. Indeed, this approach of keeping plain tables seems to be rather promising and recently, it was also discussed for problems related to answer set programming (ASP) [Hecher et al., 2020a]. Further, note that one can also use reductions from certain fragments of Asp to Sat that preserve the treewidth, as discussed, e.g., in Chapter 4. Then, this allows us to immediately apply these reductions in order to use hybrid solving dynamic programming via nested DP for Sat or QSat in general.

7.3 Dynamic Programming with Database Management Systems

While algorithms that run dynamic programming on bounded treewidth can be quite useful for efficient problem solving in practice, implementations turn out to be tedious already for problems such as the propositional satisfiability problem. In the following, we aim for rapid prototyping with dynamic programming by a declarative approach that ideally uses existing systems, gets parallel execution for free, and remains fairly efficient.

In the previous section, we explained that the traversal of the tree decomposition and the overall methodology of the procedure stays the same. But the core of dynamic programming on tree decompositions for various problems is mostly the specification of the table algorithm that modifies a table based on previously computed tables. Hence, one can often focus on the table algorithms and their descriptions. When recalling basics from databases [Elmasri and Navathe, 2016] and taking a closer look on table algorithms of Chapter 3 like Listing 3.2, we can immediately spot that these algorithms are effectively describing a query on existing data that produces a new table. This motivates our idea to use a database management system to execute the query and specify the query in SQL. Before we can proceed with our idea to use databases for executing DP algorithms, we take a step back and recall that the theory of SQL queries is based on relational algebra.

Relational algebra is discussed in Section 7.3.1, which allows us to describe our algorithms and later use SQL encodings for specifying table algorithms. The intermediate step of stating the algorithm in a relation algebra description is twofold. First, we can immediately see the connection between the algorithms given in the literature, which allows us to use the existing algorithms without reproving all properties. Second, we obtain a compact mathematical description, which is not just a lengthy and technical SQL query that might be hard to understand to researchers from the community who are usually not very familiar with practical databases and the usage of query languages.

Then, in Section 7.3.2 we discuss an architecture for dynamic programming based on relational algebra (SQL) and show how to specify diverse (nested) table algorithms in Section 7.3.3.

7.3.1 Relational Algebra

Before we start with details on our approach, we briefly recall basics in relational algebra. The classical relational algebra was introduced by Codd [1970] as a mathematical framework for manipulating relations (tables). Since then, relational algebra serves as the formal background and theoretical basis in relational databases and their standard language *SQL (Structured Query Language)* for querying tables [Ullman, 1989]. In fact, in the following, we need extended constructs, which have not been defined in the original framework by Codd, but are standard notions in databases nowadays [Elmasri and Navathe, 2016]. For the understanding later, we would like to mention that the SQL table model and relational algebra model slightly differ. The SQL table model is a bag (multiset) model, rather than a set [Garcia-Molina et al., 2009, Chapter 5]. Below we also use extended projection and aggregation by grouping. Sometimes these are defined on bags. We avoid this in the definitions in order to keep the algorithms close to the formal set based notation. Finally, we would like to emphasize that we are not using relation algebra here as developed by Alfred Tarski for the field of abstract algebra, but really relational algebra as used in database applications and theory.

A *column* a is of a certain finite *domain* $\mathsf{dom}(a)$. Then, a *row* r over set $\mathrm{cols}(r)$ of columns is a set of pairs of the form (a, v) with $a \in \mathrm{cols}(r), v \in \mathsf{dom}(a)$ such that for each $a \in \mathrm{cols}(r)$, there is exactly one $v \in \mathsf{dom}(a)$ with $(a, v) \in r$. In order to *access* the value v of an attribute a in a row r, we sometimes write $r.a$, which returns the unique value v with $(a, v) \in r$. A *table* τ is a finite set of rows r over set $\mathrm{cols}(\tau) := \mathrm{cols}(r)$ of columns, using domain $\mathsf{dom}(\tau) := \bigcup_{a \in \mathrm{cols}(\tau)} \mathsf{dom}(a)$. We define *renaming* of τ, given a set A of columns and a bijective mapping $m : \mathrm{cols}(\tau) \to A$ with $\mathsf{dom}(a) = \mathsf{dom}(m(a))$ for $a \in \mathrm{cols}(\tau)$, by $\rho_m(\tau) := \{(m(a), v) \mid (a, v) \in \tau\}$. In SQL, renaming can be achieved by means of the `AS` keyword.

In order to select certain rows of a table, we require Boolean formulas extended with expressions using equality. Such a resulting *equality formula* is a Boolean formulas, where variables are expressions using equality. In more detail: Let d be a fixed constant over domain $\mathsf{dom}(v)$, where we call d *term constant*. Let v and v' be variables over some domain $\mathsf{dom}(v)$ and $\mathsf{dom}(v')$, where we call v and v' *term variables*. Then, an *equality formula* β is an expression of the form $v{=}d$ or $v{=}v'$. A *term assignment* J of equality formula β over term variables of β assigns each domain variable v of β a value over domain $\mathsf{dom}(v)$. The Boolean formula $\beta(J)$ *under term assignment* J is obtained as follows. First, we replace all expressions $v{=}d$ in β by 1 if $J(v) = d$, all expressions $v{=}v'$ by 1 if $J(v) = J(v')$, and by 0 otherwise. Second, we remove from the resulting clauses in $\beta(J)$ each clause c that contains a literal set to 1. Finally, we remove from every remaining clause in $\beta(J)$ every literal that is set to 0. We say a term assignment J is *satisfying* if $\beta(J) = \emptyset$.

Selection of rows in τ according to a given equality formula F over term variables $\mathrm{cols}(\tau)$ is defined[1] by $\sigma_F(\tau) := \{r \mid r \in \tau, F(\mathrm{ass}(r)) = \emptyset\}$, where function ass provides the

[1] We abbreviate for binary $v \in \mathrm{cols}(\tau)$ with $\mathsf{dom}(v) = \{0, 1\}$, $v{=}1$ by v and $v{=}0$ by $\neg v$.

corresponding term assignment of a given row $r \in \tau$. Selection in SQL is specified using keyword `WHERE`. Given a relation τ' with $\mathrm{cols}(\tau') \cap \mathrm{cols}(\tau) = \emptyset$. Then, we refer to the *cross-join* by $\tau \times \tau' := \{r \cup r' \mid r \in \tau, r' \in \tau'\}$. Further, a *$\theta$-join* (according to F) corresponds to $\tau \bowtie_F \tau' := \sigma_F(\tau \times \tau')$. Interestingly, in SQL a θ-join can be achieved by specifying the two tables (cross-join) and adding the selection according to F by means of `WHERE`.

Assume in the following a set $A \subseteq \mathrm{cols}(\tau)$ of columns. Then, we let table τ *projected to* A be given by $\Pi_A(\tau) := \{r_A \mid r \in \tau\}$, where $r_A := \{(a, v) \mid (a, v) \in r, a \in A\}$. This concept of projection can be lifted to *extended projection* $\dot{\Pi}_{A,S}$, where we assume in addition to A, a set S of expressions of the form $a \leftarrow f$, such that $a \in \mathrm{cols}(\tau) \setminus A$, f is an arithmetic function that takes a row $r \in \tau$, and there is at most one such expression for each $a \in \mathrm{cols}(\tau) \setminus A$ in S. Formally, we define $\dot{\Pi}_{A,S}(\tau) := \{r_A \cup r^S \mid r \in \tau\}$ with $r^S := \{(a, f(r)) \mid a \in \mathrm{cols}(r), (a \leftarrow f) \in S\}$. SQL allows to specify (extended) projection directly after initiating an SQL query with the keyword `SELECT`.

Later, we use *aggregation by grouping* ${}_A G_{(a \leftarrow g)}$, where we assume $a \in \mathrm{cols}(\tau) \setminus A$ and a so-called *aggregate function* $g : 2^\tau \to \mathsf{dom}(a)$, which intuitively takes a table of (grouped) rows. Therefore, we let ${}_A G_{(a \leftarrow g)}(\tau) := \{r \cup \{(a, g(\tau[r]))\} \mid r \in \Pi_A(\tau)\}$, where $\tau[r] := \{r' \mid r' \in \tau, r \subseteq r'\}$. For this purpose, we use for a set $S \subseteq \mathbb{N}$ of integers, the functions `SUM` for summing up values in S, `MIN` for providing the smallest integer in S, as well as `MAX` for obtaining the largest integer in S, which are often used for aggregation in this context. The SQL standard uses projection (`SELECT`) to specify A as well as the aggregate function g, such that these two parts are distinguished by means of the keyword `GROUP BY`.

Example 7.7. *Assume a table $\tau_1 := \{r_1, r_2, r_3\}$ of 2 columns a, b over Boolean domain $\mathsf{dom}(a) = \mathsf{dom}(b) = \{0, 1\}$, where $r_1 := \{(a, 1), (b, 1)\}$, $r_2 := \{(a, 0), (b, 0)\}$, $r_3 := \{(a, 0), (b, 1)\}$.*

τ_1	a	b
r_1	1	1
r_2	0	0
r_3	0	1

τ_2	b	a
r_1	1	1
r_2	0	0
r_3	1	0

Then, $r_3.a = 0$ and $r_3.b = 1$. Rows can be swapped by renaming and we let $\tau_2 := \rho_{\{a \mapsto b, b \mapsto a\}} \tau_1$.

Observe that, e.g., $\rho_{\{a \mapsto b, b \mapsto a\}}(\{r_3\})$ corresponds to $\{\{(a, 1), (b, 0)\}\}$, i.e., considering r_3 and swapping a and b. We select rows by using the selection σ. For example, if we want to select rows where $b = 1$ (colored in blue) we can use $\sigma_{b=1}(\tau_1)$.

Hence, applying $\sigma_{b=1}(\tau_1)$ results in $\{r_1, r_3\}$. Table τ_1 can be θ-joined with τ_2, but before, we need to have disjoint columns, which we obtain by renaming each column c to a fresh column c' as below by $\rho_{a \mapsto a', b \mapsto b'} \tau_2$. Then, $\tau_3 := \tau_1 \bowtie_{a=a' \wedge b=b'} (\rho_{a \mapsto a', b \mapsto b'} \tau_2)$.

τ_1	a	b
r_1	1	1
r_2	0	0
r_3	0	1

τ_1	a	b
r_1	1	1
r_2	0	0
r_3	0	1

$\rho_{a\mapsto a',b\mapsto b'}(\tau_2)$	b'	a'
r_1	1	1
r_2	0	0
r_3	1	0

τ_3	a	b	b'	a'
r_1	1	1	1	1
r_2	0	0	0	0

Consequently, we have $\tau_3 = \{\{(a,0),(a',0),(b,0),(b',0)\},\{(a,1),(a',1),(b,1),(b',1)\}\}$. *Extended projection allows not only to filter certain columns, but also to add additional columns. As a result, if we only select column* a *of each row of* τ_1*, but add a fresh column* c *holding the sum of the values for* a *and* b*, then* $\dot{\Pi}_{\{a\},\{c\leftarrow a+b\}}\tau_1$ *corresponds to* $\{\{(a,1),(c,2)\},\{(a,0),(c,0)\},\{(a,0),(c,1)\}\}$.

$\dot{\Pi}_{\{a\},\{c\leftarrow a+b\}}\tau_1$	a	c
r_1	1	2
r_2	0	0
r_3	0	1

Grouping τ_1 *according to the value of column* a*, where we aggregate each group by summing up the values of columns* b *in a fresh column* d*, results in* ${}_{\{a\}}G_{d\leftarrow\tau\mapsto SUM(\{r.b|r\in\tau\})}(\tau_1)$*, which simplifies to* $\{\{(a,1),(d,1)\},\{(a,0),\{d,1)\}\}$ *as illustrated below.*

τ_1	a	b
r_1	1	1
r_2	0	0
r_3	0	1

${}_{\{a\}}G_{d\leftarrow\tau\mapsto\texttt{SUM}(\{r.b\|r\in\tau\})}(\tau_1)$	a	d
r_1	1	1
r_3	0	1

Instead of using set theory to describe how tables are obtained during dynamic programming, one could alternatively use relational algebra. There, tables τ_t for each TD node t are pictured as relations, where τ_t distinguishes a unique column x for each $x \in \chi(t)$. Further, there might be additional columns required depending on the problem at hand, e.g., we need a column *cnt* for counting in #Sat, or a column for modeling costs or weights in case of optimization problems. Listing 7.8 presents a table algorithm for problem #Sat that is equivalent to Listing 3.2, but relies on relational algebra only for computing tables. This step from set notation to relational algebra is driven by the observation that in these table algorithms one can identify recurring patterns and one mainly has to adjust problem-specific parts of it. We highlighted it by colors in Listing 7.8. In particular, one typically derives for nodes t with $\text{type}(t) = \textit{leaf}$, a fresh initial table τ_t,

Listing 7.8: Table algorithm $\#\mathsf{Sat}'_t(\chi(t), F_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving #Sat.

In: Node t, bag $\chi(t)$, bag formula F_t, sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables of t.
Out: Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t := \{\{(\text{cnt}, 1)\}\}$
2 **else if** type$(t) =$ *intr, and* $a \in \chi(t)$ *is introduced* **then**
3 $\quad \tau_t := \tau_1 \bowtie_{F_t} \{\{(a,0)\},\{(a,1)\}\}$
4 **else if** type$(t) =$ *rem, and* $a \notin \chi(t)$ *is removed* **then**
5 $\quad \tau_t := {}_{\chi(t)}G_{\text{cnt} \leftarrow \tau \mapsto \text{SUM}(\{r.\text{cnt} \mid r \in \tau\})}(\Pi_{\text{cols}(\tau_1)\setminus\{a\}}\tau_1)$
6 **else if** type$(t) =$ *join* **then**
7 $\quad \tau_t := \dot{\Pi}_{\chi(t),\{\text{cnt} \leftarrow \text{cnt}\cdot\text{cnt}'\}}(\tau_1 \bowtie_{\bigwedge_{u \in \chi(t)} u=u'} \rho_{\bigcup_{u \in \text{cols}(\tau_2)} \{u \mapsto u'\}}\tau_2)$
8 **return** τ_t

cf. Line 1 of Listing 7.8. Then, whenever a variable a is introduced, such algorithms often use θ-joins with a fresh initial table for the introduced variable a. Hence, the new column represents the potential values for variable a. In Line 3, the selection of the θ-join is performed according to F_t, i.e., corresponding to the bag instance of #Sat. Further, for nodes t with type$(t) =$ *rem*, these table algorithms typically need projection. In case of Listing 7.8, Line 5 also needs grouping in order to sum up the counters for those rows of τ_1 that concur in τ_t. Thereby, rows are grouped according to values of columns $\chi(t)$ and we keep only one row per group in table τ, where the fresh counter *cnt* is the sum among all counters in τ. Finally, in Line 7 for a node t with type$(t) =$ *join*, we use extended projection and θ-joins, where we join on the same truth assignments. This allows us later to leverage database technology for a usually expensive operation. Extended projection is needed for multiplying the counters of the two rows containing the same assignment.

7.3.2 Dynamic Programming via Relational Algebra and Databases

In this section, we present a general architecture to model table algorithms by means of database management systems. The architecture is influenced by the basic dynamic programming approach of Chapter 3 (cf. Figure 3.1) and works as depicted in Figure 7.5, where the steps highlighted in yellow and blue need to be specified depending on the problem P. Steps outside Step 3 are mainly setup tasks, the yellow "E"s indicate *events* that might be needed to solve more complex problems on the polynomial hierarchy. For example, one could create and drop auxiliary sub-tables for each node during Step 3 within such events. Observe that after the generation of an LTD $\mathcal{T} = (T, \chi, \delta_\mathcal{I})$ that is constructed for an instance $\mathcal{I}$ of problem P, Step 2b automatically creates tables τ_t for each node t of T, where the corresponding table columns of τ_t are specified in the blue part, i.e., within A. The *default columns* of such a table τ_t that are assumed in this section foresee one column for each element of the bag $\chi(t)$, where additional columns that are needed for solving the problem can be added. This includes additional auxiliary columns, which can be also counters or costs for counting or optimization, respectively.

Actually, the core of this architecture is focused on the table algorithm A executed

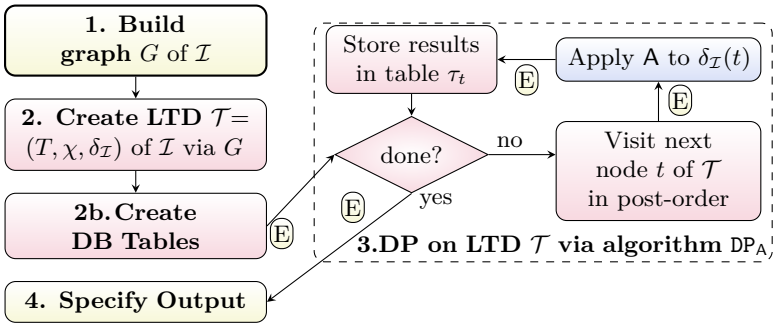

Figure 7.5: Architecture of Dynamic Programming with Databases. Steps highlighted in red are provided by the system depending on the specification of the yellow and blue steps, which are given by the user for specific problems P. The yellow "E"s represent events that can be intercepted and handled by the user. The blue step concentrates on table algorithm A, where the user specifies how SQL code is generated in a modular way.

Listing 7.9: Template table algorithm $\mathsf{A}_t(\chi(t), \mathcal{I}_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ of Figure 7.5 for solving an instance $\mathcal{I}$ of problem P.

In: Node t, bag $\chi(t)$, bag instance $\mathcal{I}_t$, and a sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables.
Out: Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t \leftarrow$ #leafTab#
2 **else if** type$(t) =$ *intr, and* $a \in \chi(t)$ *is introduced* **then**
3 $\quad \tau_t \leftarrow \dot{\Pi}_{\chi(t),\#\text{intrAddCols}\#}(\tau_1 \bowtie_{\#\text{intrFilter}\#} \#\text{intrTab}\#)$
4 **else if** type$(t) =$ *rem, and* $a \notin \chi(t)$ *is removed* **then**
5 $\quad \tau_t \leftarrow {}_{\chi(t)\cup\#\text{remGroupCols}\#}G_{\#\text{remAggr}\#}(\Pi_{\text{cols}(\tau_1)\setminus\{a,\#\text{remCols}\#\}}\sigma_{\#\text{remFilter}\#}\tau_1)$
6 **else if** type$(t) =$ *join* **then**
7 $\quad \tau_t \leftarrow \dot{\Pi}_{\chi(t),\#\text{joinAddCols}\#}(\tau_1 \bowtie_{\bigwedge_{u\in\chi(t)} u=u' \wedge \#\text{joinAddFilter}\#} \rho_{\bigcup_{u\in\text{cols}(\tau_2)}\{u\mapsto u'\}}\tau_2)$
8 **return** τ_t

for each node t of T of LTD $\mathcal{T} = (T, \chi, \delta_{\mathcal{I}})$. Besides the definition of table schemes, the blue part concerns specification of the table algorithm by means of a procedural *generator template* that describes how to obtain SQL code for each node t, thereby depending on $\chi(t)$ and on the tables for child nodes of t. This generated SQL code is then used internally for manipulation of tables τ_t during the tree decomposition traversal in Step 3 of dynamic programming. Listing 7.9 presents a general template, where parts of table algorithms for problems that are typically problem-specific are replaced by colored placeholders of the form #placeHolder#, cf. Listing 7.8. We briefly discuss the intuition behind these placeholders. For leaf nodes, the initial table (typically empty) can be specified using #leafTab#. For introduce nodes, the potential cases for the introduced vertex a are given with the help of #intrTab#. Then, according to the bag instance, we only keep those rows that satisfy #intrFilter#. Placeholder #intrAddCols# allows to add additional columns for solving problem, which are required e.g., for counting or optimization. In other words, placeholder #intrAddCols# in Line 3 of Listing 7.9 uses extended projection, which is needed for problems requiring changes on vertex

Listing 7.10: Table algorithm $\mathsf{Col}_t(\chi(t), G_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving #3-Col.

In: Node t, bag $\chi(t)$, bag graph $G_t = (V_t, E_t)$, and a sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables.
Out: Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t \leftarrow \{(\text{cnt}, 1)\}$
2 **else if** type$(t) =$ *intr, and* $a \in \chi(t)$ *is introduced* **then**
3 $\quad \tau_t \leftarrow \tau_1 \bowtie_{\bigwedge_{\{u,v\} \in E_t} u \neq v} \{\{(a,0)\}, \{(a,1)\}, \{(a,2)\}\}$
4 **else if** type$(t) =$ *rem, and* $a \notin \chi(t)$ *is removed* **then**
5 $\quad \tau_t \leftarrow {}_{\chi(t)}G_{\text{cnt} \leftarrow \tau \mapsto \texttt{SUM}(\{r.\text{cnt} | r \in \tau\})}(\Pi_{\text{cols}(\tau_1) \setminus \{a\}} \tau_1)$
6 **else if** type$(t) =$ *join* **then**
7 $\quad \tau_t \leftarrow \dot{\Pi}_{\chi(t), \{\text{cnt} \leftarrow \text{cnt} \cdot \text{cnt}'\}}(\tau_1 \bowtie_{\bigwedge_{u \in \chi(t)} u = u'} \rho_{\bigcup_{u \in \text{cols}(\tau_2)} \{u \mapsto u'\}} \tau_2)$
8 **return** τ_t

introduction. Nodes, where an atom a is removed sometimes require to filter rows, which do not lead to a solution using #remFilter#, and to remove columns concerning a by #remCols#. Further, one oftentimes needs to aggregate rows according to the values of the columns of the bag and additional columns (given by #remGroupCols#), where the aggregation is specified by #remAggr#. Finally, for join nodes, one can specify an additional filter #joinAddFilter# that goes beyond checking equivalence of row values in the θ-join operation. Further, depending on the problem one might need to add and update values of additional columns by using extended projection in form of placeholder #joinAddCols#.

7.3.3 Specifying Table Algorithms with Relational Algebra

The general template of table algorithms above works for many problems, including decision problems, counting problems as well as optimization problems. As a proof of concept, we present the relevant parts of table algorithm specification according to the template in Listing 7.9 for a selection of problems below. There, the placeholders of the form #placeHolder# given in Listing 7.9 are instantiated with concrete values in order to solve the corresponding problem. To this end, we assume a nice LTD $\mathcal{T} = (T, \chi, \delta_{\mathcal{I}})$ of the corresponding primal graph representation $\mathcal{G}_{\mathcal{I}}$ of our given instance $\mathcal{I}$.

If the given instance $\mathcal{I}$ is already a graph G, we refer to the primal graph of G by the graph itself, i.e., $\mathcal{G}_G := G$. Further, the bag instance of $G = (V, E)$, is referred to by *bag graph* G_t and it is defined for a given node t of T by $G_t := (V \cap \chi(t), E \cap [\chi(t) \times \chi(t)])$.

Problem #3-Col: Counting 3-Colorings

Recall that for a given graph instance $G = (V, E)$, a 3-*coloring* is a mapping $\iota : V \to \{0, 1, 2\}$ such that for each edge $\{u, v\} \in E$, we have $\iota(u) \neq \iota(v)$. Then, the problem #3-Col asks to count the number of 3-colorings of G. The table algorithm for this problem #3-Col is given in Listing 7.10. Similarly to Listing 7.8, for (empty) leaf nodes, the counter *cnt* is set to 1 in Line 1. Whenever a vertex a is introduced, in Line 3, one of the 3 many color values for a are guessed and θ-joined with the table τ_1 for the child

Listing 7.11: Table algorithm $\mathsf{VC}_t(\chi(t), G_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving MinVC.

In: Node t, bag $\chi(t)$, bag graph $G_t = (V_t, E_t)$, and a sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables.
Out: Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t \leftarrow \{(\text{card}, 0)\}$
2 **else if** type$(t) =$ *intr, and* $a \in \chi(t)$ *is introduced* **then**
3 $\quad \tau_t \leftarrow \tau_1 \bowtie_{\bigwedge_{\{u,v\} \in E_t} u \vee v} \{\{(a,0)\}, \{(a,1)\}\}$
4 **else if** type$(t) =$ *rem, and* $a \notin \chi(t)$ *is removed* **then**
5 $\quad \tau_t \leftarrow {}_{\chi(t)}G_{\text{card} \leftarrow \tau \mapsto \text{MIN}(\{r.\text{card} + r.a \mid r \in \tau\})}(\Pi_{\text{cols}(\tau_1) \setminus \{a\}} \tau_1)$
6 **else if** type$(t) =$ *join* **then**
7 $\quad \tau_t \leftarrow \dot{\Pi}_{\chi(t), \{\text{card} \leftarrow \text{card} + \text{card}'\}}(\tau_1 \bowtie_{\bigwedge_{u \in \chi(t)} u = u'} \rho_{\bigcup_{u \in \text{cols}(\tau_2)} \{u \mapsto u'\}} \tau_2)$
8 **return** τ_t

node of t such that only colorings with different values for two *adjacent* vertices are kept. Similarly to Listing 7.8, whenever a vertex a is removed, Line 5 ensures that the column for a is removed and that counters *cnt* are summed up for rows that concur due to the removal of column a. Then, the case for join nodes in Line 7 is again analogous to Listing 7.8, where only rows with the same colorings in both child tables are kept and counters *cnt* are multiplied accordingly.

Problem MinVC: Computing Minimum Vertex Cover Size

Given a graph $G = (V, E)$, a *vertex cover* is a set of vertices $C \subseteq V$ of G such that for each edge $\{u, v\} \in E$, we have $\{u, v\} \cap C \neq \emptyset$. Then, MinVC asks to find the minimum cardinality $|C|$ among all vertex covers C, i.e., C is such that there is no vertex cover C' with $|C'| < |C|$. We use an additional column *card* for storing cardinalities. The table algorithm for solving MinVC is provided in Listing 7.11, where, for leaf nodes the cardinality is 0, cf. Line 1. Then, when introducing vertex a, we guess in Line 3 whether a shall be in the vertex cover or not, and enforce that for each edge of the bag instance at least one of the two endpoint vertices has to be in the vertex cover. Note that the additional cardinality column only takes removed vertices into account. More precisely, when a vertex a is removed, we group in Line 5 according to the bag columns $\chi(t)$, where the fresh cardinality value is the minimum cardinality (plus 1 for a if a shall be in the vertex cover), among those rows that concur due to the removal of a. The join node is similar to before, but in Line 7 we additionally need to sum up the cardinalities of two adjoining child table rows.

7.3.4 Specifying Nested Table Algorithms with Relational Algebra

This idea of specifying table algorithms as depicted in the template table algorithm of Listing 7.9 can be even used to specify nested table algorithms. To this end, recall nested table algorithms of Section 7.1 as well as hybrid dynamic programming as discussed in Section 7.2.

Then, the implementation of nested table algorithm H#∃Sat of Listing 7.5 by means

Listing 7.12: Nested table algorithm $\mathsf{H}\#\exists\mathsf{Sat}'_t(\text{depth}, \chi_t, \cdot, F'_t, Q^A_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving #∃Sat.

In: Node t, nesting depth ≥ 0, bag χ_t, bag formula F'_t, nested bag QBF $Q^A_t = \exists V.F^A_t$, and sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables of t.
Out: Table τ_t.

1 **if** type(t) = *leaf* **then** $\tau_t \leftarrow \{\{(\text{cnt}, 1)\}\}$
2 **else if** type(t) = *intr, and* $a \in \chi_t$ *is introduced* **then**
3 $\quad \tau_t \leftarrow \tau_1 \bowtie_{F'_t} \{\{(a,0)\},\{(a,1)\}\}$
4 $\quad \tau_t \leftarrow \sigma_{\text{cnt}>0}(\dot{\Pi}_{\chi_t, \{\text{cnt} \leftarrow \text{cnt} \cdot \mathtt{HybDP}_{\mathsf{H}\#\exists\mathsf{Sat}}(\text{depth}+1,\, Q^A_t[\text{ass}])\}} \tau_t)$
5 **else if** type(t) = *rem, and* $a \notin \chi_t$ *is removed* **then**
6 $\quad \tau_t \leftarrow {}_{\chi_t}G_{\text{cnt} \leftarrow \mathtt{SUM}(\text{cnt})}(\Pi_{\text{cols}(\tau_1)\setminus\{a\}} \tau_1)$
7 **else if** type(t) = *join* **then**
8 $\quad \tau_t \leftarrow \dot{\Pi}_{\chi_t, \{\text{cnt} \leftarrow \text{cnt} \cdot \text{cnt}'\}}(\tau_1 \bowtie_{\bigwedge_{a \in \chi_t} a = a'} \rho_{\bigcup_{a \in \text{cols}(\tau_2)} \{a \mapsto a'\}} \tau_2)$

⋆) Function ass refers to the respective truth assignment $I\colon \chi_t \to \{0,1\}$ of a given row $r \in \tau_t$.

of relational algebra is sketched by algorithm $\mathsf{H}\#\exists\mathsf{Sat}'$ of Listing 7.12. Note that in Line 4, for each row $r \in \tau_t$, this algorithm recursively relies on $\mathtt{HybDP}_{\mathsf{H}\#\exists\mathsf{Sat}'}$ on nested bag QBF Q^A_t simplified by the assignment ass(r) of the current row r. Of course, the value of the recursion through $\mathtt{HybDP}_{\mathsf{H}\#\exists\mathsf{Sat}'}$ has to be computed first and in Line 4 and by slight abuse of notation the recursive expression is therefore treated as if it was a constant value. As a result, Line 4 uses extended projection, cf. Listing 7.8, where the counter *cnt* of the respective row r is updated by multiplying it with the resulting value of the recursion through $\mathtt{HybDP}_{\mathsf{H}\#\exists\mathsf{Sat}'}$. Notably, as the recursive call to $\mathtt{HybDP}_{\mathsf{H}\#\exists\mathsf{Sat}'}$ within extended projection of Line 4 implicitly takes a given current row r, the function occurrences ass in Line 4 implicitly take this row r as an argument. Further, while Lines 3 and 4 seem to be not in line with the template algorithm of Listing 7.9, one can also merge those two lines at the cost of a more involved expression. Indeed, one can reformulate and "shift" the condition "cnt>0" inside the expression of Line 3, which, however, requires duplication of parts of the expression.

Similarly, the nested table algorithm HQSat of Listing 7.7 can be implemented by means of relational algebra. The result is sketched by algorithm HQSat' in Listing 7.13. Again, similar to above, one can reformulate in order to obtain a nested table algorithm that is in line with the template algorithm of Listing 7.9.

7.4 Implementing Abstractions and Hybrid Dynamic Programming

We implemented a solver $\mathsf{nestHDB}^2$ based on hybrid dynamic programming in Python3 and using table manipulation techniques by means of SQL and the *database management system (DBMS)* Postgres. We are certain that one can easily replace PostgreSQL by any other state-of-the-art relational database that uses SQL. In the following, we discuss in

[2]Source code, instances, and detailed results are available at: tinyurl.com/nesthdb.

Listing 7.13: Nested table algorithm $\mathsf{HQSAT}'_t(\text{depth}, \chi_t, Q, F'_t, Q^A_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving QSAT.

In: Node t, nesting depth ≥ 0, bag χ_t, QBF $Q = Q_1 V_1 . Q'$, bag formula F'_t, nested bag QBF Q^A_t, and child tables $\langle \tau_1, \ldots \tau_\ell \rangle$ of t.
Out: Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t \leftarrow \{\emptyset\}$
2 **else if** type$(t) =$ *intr, and* $a \in \chi_t$ *is introduced* **then**
3 $\quad \tau_t \leftarrow \sigma_{\mathtt{HybDP}_{\mathsf{QSAT}}(\text{depth}+1, Q^A_t[\text{ass}])=1}(\tau_1 \bowtie_{F'_t} \{\{(a,0)\},\{(a,1)\}\})$
4 $\quad \tau_t \leftarrow \sigma_{Q_1 = \exists \vee |\tau_t| = 2^{|\chi_t|}}(\tau_t)$
5 **else if** type$(t) =$ *rem, and* $a \notin \chi_t$ *is removed* **then**
6 $\quad \tau_t \leftarrow \Pi_{\text{cols}(\tau_1) \setminus \{a\}} \tau_1$
7 **else if** type$(t) =$ *join* **then**
8 $\quad \tau_t \leftarrow \Pi_{\chi_t}(\tau_1 \bowtie_{\bigwedge_{a \in \chi_t} a = a'} \rho_{\bigcup_{a \in \text{cols}(\tau_2)} \{a \mapsto a'\}} \tau_2)$

⋆) The cardinality of a table τ can be obtained via relational algebra (sub-expression): $|\tau| := c$, where $\{\{(\text{card}, c)\}\} = {}_{\emptyset}G_{\text{card} \leftarrow \mathtt{SUM}(1)} \tau$

Section 7.4.1 implementation specifics that are crucial for a performant system that is still extendable and flexible. Indeed, nestHDB is quite flexible in terms of implemented problems and formalisms. However, in the course of the whole section, we mainly focus on an implementation of algorithm $\mathtt{HybDP}_{\mathsf{H\#\exists Sat}}$ of Listing 7.4, but instead of using nested table algorithm H#∃Sat we rely on H#∃Sat′ as given in Listing 7.12. Then, this algorithm will be used as the basis of comparison and experiments in Section 7.4.2, where we consider the counting problems #SAT and #∃SAT.

7.4.1 Implementation Details of nestHDB

We present a list of implementation specifics that are crucial for nestHDB to reach a performance level that is competitive with respect to state-of-the-art counting solvers.

Computing Tree Decompositions. In order to apply (hybrid) dynamic programming, the solver nestHDB requires tree decompositions. These decompositions are computed mainly with the library htd version 1.2 with default settings [Abseher et al., 2017], which finds TDs extremely quick also for interesting instances [Fichte et al., 2019b] due to heuristics. Note that nestHDB directly supports the format of tree decompositions of recent competitions [Dell et al., 2017], i.e., one could easily replace the library for computing decompositions. It is important not to enforce htd to compute nice TDs, as this would cause a lot of overhead later in nestHDB for copying tables. However, in order to benefit from the implementation of θ-joins, query optimization, and state-of-the-art database technology in general, we observed that it is crucial to limit the number of child nodes of every TD node. In result, when huge tables are involved, θ-joins among child node tables cover at most a limited number of child node tables. Hence, the query optimizer of the database system can still come up with meaningful execution plans depending on the contents of the table. Nonetheless we prefer θ-joins with more than just two tables, since binary θ-joins already fix in which order these tables shall be combined,

which already limits the query optimizer. Apart from this trade-off, we tried to outsource the task of joining tables to the DBMS, since the performance of database systems highly depends on query optimization. The actual limit, which is a restriction from experience and practice only, highly depends on the DBMS that is used. For PostgreSQL, we set a limit of at most 5 child nodes for each node of the TD, i.e., each θ-join covers at most 5 child tables.

Towards Non-Nice Tree Decompositions. Although this paper presents the algorithms for nice TDs (mainly due to simplicity), the system nestHDB interleaves these cases as presented in Listing 7.9. More precisely, the system executes one query per table τ_t for each node t during the traversal of TD $\mathcal{T}$. This query consists of several parts and we briefly explain its parts from outside to inside in accordance with Listing 7.9. First of all, the inner-most part concerns the *row candidates* for τ_t consisting of the θ-join among all child tables of τ_t as in Line 7 of Listing 7.9. If there is no child node of t, table #leafTab# of Line 1 is used instead. Next, the result is cross-joined with #intrTab# for each introduced variable as in Line 3, but without using the filter #intrFilter# yet. Then, the result is projected by using extended projection involving $\chi(t)$ as well as both #joinAddCols# and #intrAddCols#. Actually, there are different configurations of how nestHDB can deal with the resulting row candidates. For debugging (see below) one could (1) actually materialize the result in a table, whereas for performance reasons, one should use (2) *common table expressions (CTEs or `WITH`-queries)* or (3) *sub-queries (nested queries)*, which both result in one nested SQL query per table τ_t. On top of these row candidates, selection according to #intrFilter#, cf. Line 3, is executed. Finally the resulting table is plugged as table τ_1 into Line 5, where in particular the result is grouped by using both $\chi(t)$[3] and #remGroupCols# and each group is aggregated by #remAggr# accordingly. It turns out that PostgreSQL can do better with sub-queries than CTEs, since we observed that the query optimizer oftentimes pushes (parts of) outer selections and projections into the sub-query if needed, which is not the case for CTEs, as discussed in the PostgreSQL manual [PostgreSQL Global Development Group, 2021, Sec. 7.8.1]. On different DBMSs or other vendors, e.g., Oracle, it might be better to use CTEs instead.

Example 7.8. *Consider again formula* $F := \{\overbrace{\{\neg a, b, c\}}^{c_1}, \overbrace{\{a, \neg b, \neg c\}}^{c_2}, \overbrace{\{a, d\}}^{c_3}, \overbrace{\{a, \neg d\}}^{c_4}\}$ *from Example 7.1 and TD* $\mathcal{T}$ *of* $\mathcal{G}_F$*, as given in Figure 3.2. Now, let us use table algorithm* #Sat′ *with nestHDB on formula* F *of TD* $\mathcal{T}$ *and Option (3): sub-queries, where the row candidates are expressed via a sub-queries. Then, for each node* t_i *of* $\mathcal{T}$*, nestHDB generates a view* vi *as well as a table* τ_i *containing in the end the content of* vi*. Observe that each view only has one column* a *for each variable* a *of* F *since the truth assignments of the other variables are not needed later. This keeps the tables compact, only* τ_1 *has two rows,* τ_2*, and* τ_3 *have only one row. We obtain the following views.*

```
CREATE VIEW v1 AS SELECT a, sum(cnt) AS cnt FROM
```

[3] Actually, nestHDB keeps only columns relevant for the table of the parent node of t.

```
 (WITH intrTab AS (SELECT 0 AS val UNION ALL SELECT 1)
   SELECT i1.val AS a, i2.val AS b, i3.val AS c, 1 AS cnt
          FROM intrTab i1, intrTab i2, intrTab i3)
WHERE (NOT a OR b OR c) AND (a OR NOT b OR NOT c) GROUP BY a

CREATE VIEW v2 AS SELECT a, sum(cnt) AS cnt FROM
 (WITH intrTab AS (SELECT 0 AS val UNION ALL SELECT 1)
   SELECT i1.val AS a, i2.val AS d, 1 AS cnt
          FROM intrTab i1, intrTab i2)
WHERE (a OR d) AND (a OR NOT d) GROUP BY a

CREATE VIEW v3 AS SELECT a, sum(cnt) AS cnt FROM
 (SELECT τ1.a, τ1.cnt * τ2.cnt AS cnt FROM τ1, τ2 WHERE τ1.a = τ2.a)
GROUP BY a
```

Parallelization. A further reason to not over-restrict the number of child nodes within the TD, lies in parallelization. In nestHDB, we compute tables in parallel along the TD, where multiple tables can be computed at the same time, as long as the child tables are computed. Therefore, we tried to keep the number of child nodes in the TD as high as possible. In our system nestHDB, we currently allow for at most 24 worker threads for table computations and 24 database connections at the same time (both pooled and configurable). On top of that we have 2 additional threads and database connections for job assignments to workers, as well as one dedicated watcher thread for clean-up and connection termination, respectively.

Logging, Debugging and Extensions. Currently, we have two versions of the nestHDB system implemented. One version aims for performance and the other one tries to achieve comprehensive logging and easy debugging of problem (instances), thereby increasing explainability. The former does neither keep intermediate results nor create database tables in advance (Step 2b), as depicted in Figure 7.5, but creates tables according to an SQL `SELECT` statement. In the latter, we keep all intermediate results, we record database timestamps before and after certain nodes, provide statistics as, e.g., width and number of rows. Further, since for each table τ_t, exactly one SQL statement is executed for filling this table, we also have a dedicated view of the SQL `SELECT` statement, whose result is then inserted in τ_t. Together with the power and flexibility of SQL queries, we observed that this helps in finding errors in the table algorithm specifications.

Besides convenient debugging, system nestHDB immediately contains an extension for *approximation*. There, we restrict the table contents to a maximum number of rows. This allows for certain approximations on counting problems or optimization problems, where it is infeasible to compute the full tables. Further, nestHDB foresees a dedicated *randomization* on these restricted number of rows such that we obtain different approximate results on different random seeds.

Note that nestHDB can be easily extended. Each problem can overwrite existing default behavior and nestHDB also supports problem-specific argument parsers for each problem individually. Out-of-the-box, we support the formats DIMACS SAT and DIMACS graph [Liu et al., 2006] as well as the common format for TDs [Dell et al., 2017].

Our solver nestHDB builds upon our recently published prototype dpdb [Fichte et al., 2021b], which applied a DBMS for plain dynamic programming algorithms. However, we used the most-recent version 12 of Postgres and we let it operate on a tmpfs-ramdisk. In our solver, the DBMS serves the purpose of extremely efficient in-memory table manipulations and query optimization required by nested DP, and therefore nestHDB benefits from database technology.

Nested DP & Choice of Standard Solvers. We implemented dedicated nested DP algorithms for solving #Sat and #∃Sat, where we do (nested) DP up to $\text{threshold}_{\text{depth}} = 2$. Further, we set $\text{threshold}_{\text{hybrid}} = 1000$ and therefore we do not "fall back" to standard solvers based on the width (cf. Line 8 of Listing 3.1), but based on the nesting depth.

Also, the evaluation of the nested bag formula is "shifted" to the database if it uses at most 40 abstraction variables, since Postgres efficiently handles these small-sized Boolean formulas. Thereby, further nesting is saved by executing optimized SQL statements within the TD nodes. A value of 40 seems to be a nice balance between the overhead caused by standard solvers for small formulas and exponential growth counteracting the advantages of the DBMS. For the standard solvers required for hybrid dynamic programming, we use #Sat solver sharpSAT [Thurley, 2006] and for #∃Sat we employ the recently published #∃Sat solver projMC [Lagniez and Marquis, 2019], solver sharpSAT and Sat solver picosat [Biere, 2008]. Observe that our solver immediately benefits from better standard solvers and further improvements of the solvers above.

Choosing Non-Nesting Variables & Compatible Nodes. TDs are computed by means of heuristics via decomposition library htd [Abseher et al., 2017]. For finding good abstractions (crucial), i.e., abstraction variables for the nested primal graph, we use encodings for solver clingo [Gebser et al., 2019], which is based on logic programming (ASP) and therefore perfectly suited for solving reachability via nesting paths. There, among a reasonably sized subset of vertices of smallest degree, we aim for a preferably large (maximal) set A of abstraction variables such that at the same time the resulting graph N_F^A for the given formula F is reasonably sparse, which is achieved by minimizing the number of edges of N_F^A. To this end, we use built-in (cost) optimization, where we take the best results obtained by clingo after running at most 35 seconds. For the concrete encodings used in nestHDB, we refer to the online repository as stated above. We expect that this initial approach can be improved and that extending by problem-specific as well as domain-specific information might help in choosing promising abstraction variables A.

As rows of tables during (nested) DP can be independently computed and parallelized [Fichte et al., 2019b], hybrid DP solver nestHDB potentially calls standard solvers for solving subproblems in parallel using a thread pool. Thereby, the uniquely compatible

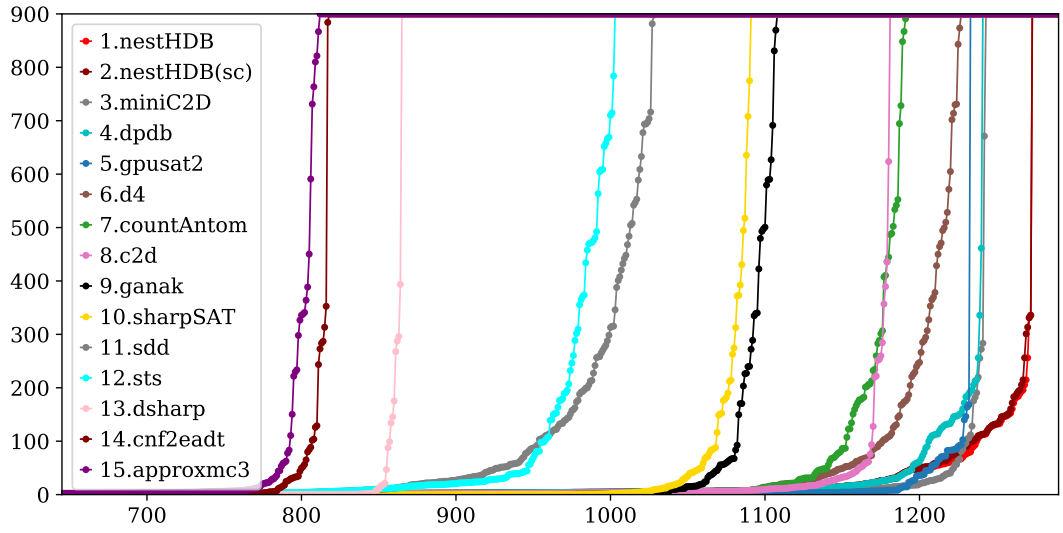

Figure 7.6: Cactus plot of instances for #SAT, where instances (x-axis) are ordered for each solver individually by runtime[seconds] (y-axis). $\text{threshold}_{\text{abstr}} = 38$.

node for relevant compatible sets U, as denoted in this paper by $\text{comp}_{F,A,\mathcal{T}}(U)$, is decided during runtime among compatible nodes on a first-come-first-serve basis.

7.4.2 Experimental Results - Hybrid Dynamic Programming in Practice

In order to evaluate the concept of hybrid dynamic programming, we conducted a series of experiments considering a variety of solvers and benchmarks, both for model counting (#SAT) as well as projected model counting (#∃SAT). During the evaluation we thereby compared the performance of algorithm $\mathsf{HybDP}_{\mathsf{H\#\exists Sat}}$ of Listing 7.4, which instead of H#∃Sat implements the nested table algorithm H#∃Sat′ as depicted in Listing 7.12. We benchmarked this algorithm both for the projected model counting problem, but also for the special case of model counting.

Benchmarked Solvers & Instances. We benchmarked nestHDB and 16 other publicly available #SAT solvers on 1,494 instances recently considered [Fichte et al., 2021b]. Among those solvers are the single-core solvers miniC2D [Oztok and Darwiche, 2015], d4 [Lagniez and Marquis, 2017], c2d [Darwiche, 2004], ganak [Sharma et al., 2019], sharpSAT [Thurley, 2006], sdd [Darwiche, 2011], sts [Ermon et al., 2012], dsharp [Muise et al., 2012], cnf2eadt [Koriche et al., 2013], cachet [Sang et al., 2004], sharpCDCL [Klebanov et al., 2013], approxmc3 [Chakraborty et al., 2014], as well as bdd_minisat [Toda and Soh, 2015]. We also included multi-core solvers dpdb [Fichte et al., 2021b], gpusat2 [Fichte et al., 2019b], as well as countAntom [Burchard et al., 2015]. Note that we excluded distributed solvers such as dCountAntom [Burchard et al., 2016] and DMC [Lagniez et al., 2018] from our experimental setup. Both solvers require a cluster with access

to the Open-MPI framework [Gabriel et al., 2004] and fast physical interconnections. Unfortunately, we do not have access to OpenMPI on our cluster. Nonetheless, our focus are shared-memory systems and since dpdb might well be used in a distributed setting, it leaves an experimental comparison between distributed solvers that also include dpdb as subsolver to future work. While nestHDB itself is a multi-core solver, we additionally included in our comparison nestHDB(sc), which is nestHDB, but restricted to a single core only. The instances [Fichte et al., 2021b] we took are already preprocessed by pmc [Lagniez and Marquis, 2014] using recommended options `-vivification -eliminateLit -litImplied -iterate=10 -equiv -orGate -affine`, which guarantee that the model counts are preserved. However, nestHDB still uses pmc with these options also in Line 2 of Listing 3.1.

Further, we considered the problem #∃Sat, where we compare solvers projMC [Lagniez and Marquis, 2019], clingo [Gebser et al., 2019], ganak [Sharma et al., 2019], nestHDB [2], and nestHDB(sc) on 610 publicly available instances[4] from projMC (consisting of 15 *planning*, 60 *circuit*, and 100 *random* instances) and Fremont, with 170 *symbolic-markov* applications, and 265 *misc* instances. For preprocessing in Line 2 of Listing 3.1, nestHDB uses pmc as before, but without options `-equiv -orGate -affine` to ensure preservation of models (equivalence).

Benchmark Setup. Solvers ran on a cluster of 12 nodes. Each node of the cluster is equipped with two Intel Xeon E5-2650 CPUs consisting of 12 physical cores each at 2.2 GHz clock speed, 256 GB RAM. For dpdb and nestHDB, we used Postgres 12 on a tmpfs-ramdisk (/tmp) that could grow up to at most 1 GB per run. Results were gathered on Ubuntu 16.04.1 LTS machines with disabled hyperthreading on kernel 4.4.0-139. We mainly compare total wall clock time and number of timeouts. For parallel solvers (dpdb, countAntom, nestHDB) we allow 12 physical cores. Timeout is 900 seconds and RAM is limited to 16 GB per instance and solver. Results for gpusat2 are taken from [Fichte et al., 2021b], where a machine equipped with a consumer GPU is used: Intel Core i3-3245 CPU operating at 3.4 GHz, 16 GB RAM, and one Sapphire Pulse ITX Radeon RX 570 GPU running at 1.24 GHz with 32 compute units, 2048 shader units, and 4GB VRAM using driver amdgpu-pro-18.30-641594 and OpenCL 1.2. The system operated on Ubuntu 18.04.1 LTS with kernel 4.15.0-34.

Benchmark Results. The results for #Sat showing the best 14 solvers are summarized in the cactus plot of Figure 7.6. Overall it shows nestHDB among the best solvers, solving 1,273 instances. The reason for this is, compared to dpdb, that nestHDB can solve instances using TDs of primal graphs of widths larger than 44, up to width 266. This limit is even slightly larger than the width of 264 that sharpSAT on its own can handle. We also tried using minic2d instead of sharpSAT as standard solver for solvers nestHDB and nestHDB(sc), but we could only solve one instance more. Notably, nestHDB(sc) has about the same performance as nestHDB, indicating that parallelism does not help

[4]Sources: tinyurl.com/projmc; tinyurl.com/pmc-fremont-01-2020.

bench-mark set	solver	tw upper bound max	0-30	31-50	>50	$\sum$	time [h]
planning	nestHDB	**30**	**7**	0	0	**7**	**2.88**
	nestHDB(sc)	**30**	**7**	0	0	**7**	3.31
	projMC	26	6	0	0	6	3.01
	ganak	19	5	0	0	5	3.36
	clingo	4	1	0	0	1	4.00
circ	nestHDB	**99**	**34**	**10**	**16**	**60**	2.10
	nestHDB(sc)	**99**	**34**	4	14	52	4.60
	projMC	91	28	**10**	11	49	6.23
	ganak	**99**	**34**	**10**	**16**	**60**	**1.21**
	clingo	**99**	31	**10**	**16**	57	4.44
random	nestHDB	79	**30**	**20**	**17**	**67**	**10.91**
	nestHDB(sc)	79	**30**	**20**	15	65	11.29
	projMC	**84**	**30**	**20**	15	65	11.09
	ganak	19	19	0	0	19	23.18
	clingo	24	25	0	0	25	21.38
markov	nestHDB	23	62	0	0	62	31.98
	nestHDB(sc)	23	61	0	0	61	32.54
	projMC	8	54	0	0	54	33.65
	ganak	**59**	**64**	0	**4**	**68**	**30.32**
	clingo	3	38	0	0	38	37.54
misc	nestHDB	47	**38**	**17**	0	**55**	46.12
	nestHDB(sc)	47	**38**	13	0	51	48.20
	projMC	47	**38**	13	0	51	45.90
	ganak	44	**38**	15	0	53	**45.72**
	clingo	**63**	**38**	15	**1**	54	44.79
Σ	nestHDB	**99**	**171**	**47**	**33**	**251**	**93.99**
	nestHDB(sc)	**99**	170	37	29	236	99.95
	projMC	91	156	43	26	225	99.88
	ganak	**99**	160	25	20	205	103.78
	clingo	**99**	133	25	17	175	112.15

Figure 7.7: Number of solved #∃SAT instances, grouped by upper bound intervals of treewidth. time[h] is cumulated wall clock time, timeouts count as 900s. $\text{threshold}_{\text{abstr}}$=8.

much on the instances. Further, we observed that the employed simple cache as used in Listing 3.1, provides only a marginal improvement.

Figure 7.7 depicts a table of results on #∃SAT, where we observe that nestHDB does a good job on instances with low widths below $\text{threshold}_{\text{abstr}} = 8$ (containing ideas of dpdb), but also on widths well above 8 (using nested DP). Notably, nestHDB is also competitive on widths well above 50. Indeed, nestHDB and nestHDB(sc) perform well on all benchmark sets, whereas on some sets the solvers projMC, clingo and ganak are faster. Overall, parallelism provides a significant improvement here, but still nestHDB(sc) shows competitive performance, which is also visualized in the cactus plot of Figure 7.8. Figure 7.9 shows scatter plots comparing nestHDB to projMC (left) and to ganak (right).

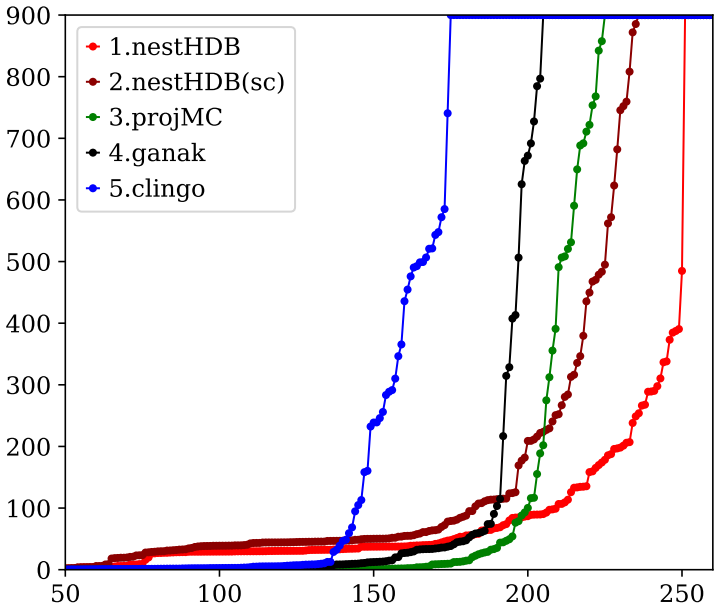

Figure 7.8: Cactus plot showing the number of solved #∃Sat instances, where the x-axis shows for each solver (configuration) individually, the number of instances ordered by increasing runtime. time[h] is cumulated wall clock time, timeouts count as 900s. $\text{threshold}_{\text{abstr}}$=8.

Overall, both plots show that nestHDB solves more instances, since in both cases the y-axis shows more black dots at 900 seconds than the x-axis. Further, the bottom left of both plots shows that there are plenty easy instances that can be solved by projMC and ganak in well below 50 seconds, where nestHDB needs up to 200 seconds. Similarly, the cactus plot given in Figure 7.8 shows that nestHDB can have some overhead compared to the three standard solvers, which is not surprising. This indicates that there is still room for improvement if, e.g., easy instances are easily detected, and if standard solvers are used for those instances. Alternatively, one could also just run a standard solver for at most 50 seconds and if not solved within 50 seconds, the heavier machinery of nested dynamic programming is invoked. Apart from these instances, Figure 7.9 shows that nestHDB solves harder instances faster, where standard solvers struggle.

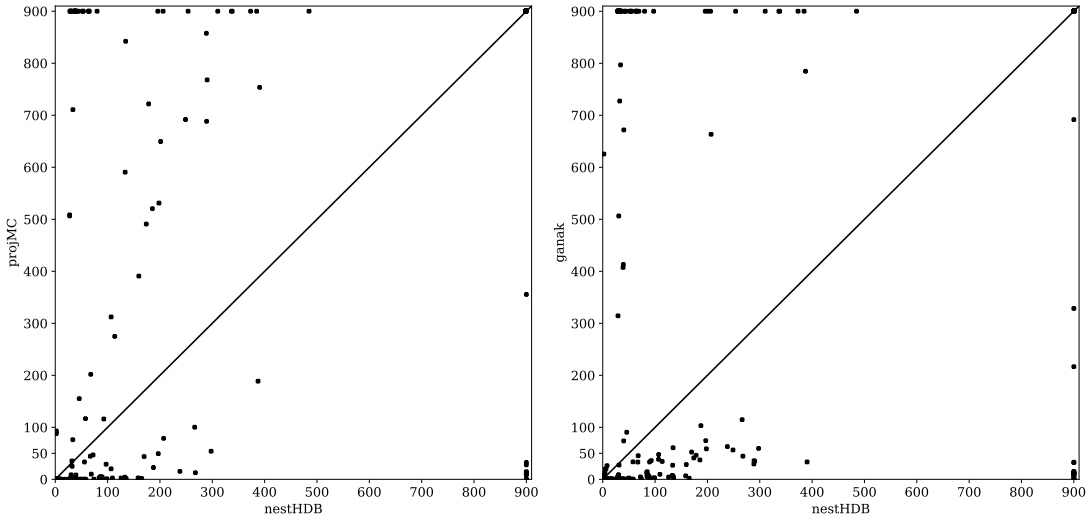

Figure 7.9: Scatter plot of instances for #∃SAT, where the x-axis shows runtime in seconds of nestHDB compared to the y-axis showing runtime of projMC (left) and of ganak (right). $\text{threshold}_{\text{abstr}} = 8$.

CHAPTER 8

Discussion

The aim of argument, or of discussion, should not be victory, but progress.

— Joseph Joubert

This thesis deals with advanced tools and methods for problems that are parameterized by treewidth. Thereby, we show in Chapter 3 how to utilize treewidth by means of dynamic programming such that the running times of the resulting algorithms with respect to treewidth are tight under reasonable assumptions in computational complexity, cf. Chapter 5. To illustrate this technique [Bertelè and Brioschi, 1972, 1973; Bodlaender and Kloks, 1996], we show how to apply it for fragments of answer set programming. Then, we provide decomposition-guided reductions in Chapter 4, which are reductions that are guided by a tree decomposition and allow us to easily show treewidth guarantees. More precisely, with these reductions, we obtain translatability results from several fragments of answer set programming to Boolean satisfiability, thereby taking care of the treewidth.

However, later in Chapter 5, it is shown that for our discussed problems parameterized by treewidth neither our dynamic programming algorithms of Chapter 3, nor the decomposition-guided reductions of Chapter 4 can be significantly improved under reasonable assumptions like the exponential time hypothesis (ETH) as stated in Hypothesis 2.1. By providing these lower bounds for the canonical PSPACE-complete problem of deciding the validity (QSAT) of a quantified Boolean formula, we solve the question of whether one can improve an existing algorithm [Chen, 2004] that has been open for a long time. Even further, it turns out that our lower bound result for QSAT serves the purpose of establishing in Chapter 6 a novel methodology for proving lower bounds for several problems parameterized by treewidth. Indeed, in Chapter 6, we reveal a table of results that completes our initial Table 1.1 and states it more precisely, which can be established using our novel methodology and further techniques of this thesis.

Finally, Chapter 7 provides means to efficiently implement treewidth-based algorithms, despite the devastating lower bound results of Chapter 6. Our approach relies on different

levels of abstractions that are inspired by related work [Ganian et al., 2017; Dell et al., 2019; Eiben et al., 2019; Hecher et al., 2020a], where we put these ideas into the context of nested dynamic programming and hybrid solving. It turns out that our approach is quite competitive compared to state-of-the-art solvers in the area of model counting and extensions thereof. Surprisingly, it seems that in practice our approach even tames the beast of high treewidth, which is witnessed by the observation that our implementation allows us to solve instances using tree decompositions of widths beyond 260. This is quite unexpected, since we also focus on problems that are double exponential [Fichte et al., 2018b] in the treewidth (under the ETH), where we apply tree decompositions of widths up to 99. While we did initial studies for prototypical counting problems, we expect that this implementation might be also of use for answer set programming and even further formalisms relevant to reasoning problems and artificial intelligence in general. Indeed, the established decomposition-guided reductions of Chapter 4 might provide a first approach into this direction. However, there are probably deeper studies and problem-specific fine-tunings required in order to competitively apply this approach for a variety of other formalisms. This can be witnessed by several works on algorithms that are based on dynamic programming [Fichte et al., 2018c; Charwat and Woltran, 2019; Tamaki, 2019; Fichte et al., 2019b; Bannach and Berndt, 2019]. There, it seems that many problem-specific optimizations are essential in order to design a solver utilizing treewidth that is competitive with the state-of-the-art. Indeed, compared to initial approaches that naively implement dynamic programming in a way that is leaning towards the worst case, it turns out that many fine-tunings are essential in order to obtain efficient solvers.

Next, we discuss a combined presentation of related work, which is then followed by outlooks into future work.

8.1 Related Work

Tractability results for treewidth have been established for many problems and formalisms, where some of them even involve dedicated dynamic programming algorithms. For answer set programming, several works in this direction are known [Jakl et al., 2009; Pichler et al., 2010], including hardness results for some graph representations [Pichler et al., 2014] and studies on non-ground programs [Bliem et al., 2020]. Similar results have been established for formalisms related to answer set programming and beyond [Fichte and Hecher, 2018, 2019; Fichte et al., 2020d; Hecher et al., 2020a]. There are also results involving these dynamic programming algorithms for further problems in knowledge representation and reasoning, e.g., [Dvořák et al., 2012; Fichte et al., 2018b, 2019a; Fichte and Hecher, 2020; Fichte et al., 2021a]. However, even parameterized results for answer set programming and related formalisms beyond treewidth have been established in a variety of works [Gottlob et al., 2002b; Lonc and Truszczyński, 2003; Lackner and Pfandler, 2012; Meier et al., 2015; Creignou and Vollmer, 2015].

Especially for deciding the consistency of logic programs (problem Asp), there are several parameters that have been considered in the past. Among those measures from

parameterized complexity [Lonc and Truszczyński, 2003], is for example the so-called feedback width [Gottlob et al., 2002b], which depends on the atoms required to break large strongly connected components (positive cycles) of the dependency graph. Another such measure is the smallest backdoor size, which is the smallest size of a set of atoms such that when removed from the program, the resulting program is normal or acyclic [Fichte and Szeider, 2015]. Note that also the special case of programs, where the number of even and/or odd cycles is bounded, has been analyzed [Lin and Zhao, 2004a], which is orthogonal to the size of the largest cycle or strongly connected component, as discussed in Section 4.4. Recently, even combinations of parameters were considered in a systematic way [Fichte et al., 2019c]. There are also studies on parameters that are strictly more general than treewidth, where still approaches similar to dynamic programming as discussed in Chapter 3 are applicable. Among these parameters are, e.g., directed treewidth-based measures [Bliem et al., 2016b], hypertree width [Gottlob et al., 2002a], and fractional hypertree width [Grohe and Marx, 2006, 2014; Fischl et al., 2018]. While the (fractional) hypertree width subsumes treewidth, there are even dedicated tools for computing such decompositions efficiently, cf. [Gottlob and Samer, 2009; Fichte et al., 2018a; Korhonen et al., 2019; Fischl et al., 2019; Fichte et al., 2020f].

Conditional runtime lower bounds that depend on the ETH and are beyond super exponentiality have been considered in a range of works, cf. [Marx and Mitsou, 2016; Lokshtanov et al., 2011]. While precise lower bounds for deciding the validity of quantified Boolean formulas (QSat) and treewidth have been left open since the dynamic programming algorithm serving as upper bound [Chen, 2004], several related work in this direction exists. Indeed, it has been proven that QSat remains intractable when parameterized by treewidth alone [Atserias and Oliva, 2014]. Thereby, Atserias and Oliva [Atserias and Oliva, 2014] cover a setting that is related to ours: showing that there exists a compression of pathwidth for a fragment of path decompositions of QBFs, thereby increasing the quantifier rank by two. However, we require a general, constructive method to compress the width of *arbitrary* tree decompositions of any QBF, thereby increasing quantifier rank by only one. Such a method is presented in Section 5.1, which allows us to improve their result [Atserias and Oliva, 2014] (cf. Corollary 5.17). Lampis and Mitsou [Lampis and Mitsou, 2017] established that QSat restricted to quantifier depth 2 cannot be solved by an algorithm that runs below double exponential in the treewidth of the primal graph when assuming the ETH. In an earlier work [Pan and Vardi, 2006], it was mentioned that this behavior extends to quantifier rank 3 and even to quantifier ranks that are an odd number. Lampis, Mitsou, and Mengel [Lampis et al., 2018] employed the known runtime result [Chen, 2004] and proposed reductions from a collection of reasoning problems in AI to QSat that yield quite precise upper bounds on the runtime, especially in the light of the corresponding lower bound [Lampis and Mitsou, 2017]. Indeed, also the establishment of the lower bound of Section 5.1 fostered many results, as already discussed, cf. Table 6.1 of Chapter 6. Runtime classes for parameterized problems have been considered before in the course of several works, e.g., [Weyer, 2004; Downey et al., 2007]. However, in this work and in particular in Chapter 6 we discuss runtime classes specific for treewidth and with respect to the ETH as given in

Hypothesis 2.1. Further, the literature also distinguishes different hierarchies of classes of parameterized problems beyond FPT. These classes are based on weighted variants of QSat [de Haan, 2019], where certain assignments are restricted to weight k, which means setting at most k variables to true. Compared to these hierarchies, the runtime classes of Chapter 6 are strictly contained in FPT and these classes even provide more fine-grained runtime guarantees.

The established lower bound for deciding the consistency of a normal program of bounded treewidth, as presented in Section 5.2 is quite surprising. In the light of the lower bound results for QSat of Section 5.1 and given that the consistency of a normal program is of the same complexity as deciding the satisfiability (Sat) of a Boolean formula (NP-complete), one might think that for treewidth the situation is similar. Instead, the problem Asp restricted to normal programs is slightly superexponential, a runtime class that has been studied before [Lokshtanov et al., 2018].

There are many implemented tools and systems that utilize treewidth supported by problem-specific implementations, techniques, and fine-tunings or tree decompositions, ranging from specialized solvers such as dynasp [Fichte et al., 2017b], dynQBF [Charwat and Woltran, 2017], and gpuSAT [Fichte et al., 2018b, 2019b] to fvs-pace [Kiljan and Pilipczuk, 2018]. Some of these parameterized solvers are particularly efficient for certain fragments [Lonsing and Egly, 2018a], and even successfully participated in problem-specific competitions [Pulina and Seidl, 2019]. Further, the literature distinguishes also generic systems that exploit treewidth like D-FLAT [Bliem et al., 2016a], Jatatosk [Bannach and Berndt, 2019], and sequoia [Langer et al., 2012]. Surprisingly, tree decompositions have been also applied to improve state-of-the-art grounders for answer set programming [Bichler et al., 2020]. The concept of abstractions as discussed in Chapter 7 was already used before [Ganian et al., 2017; Dell et al., 2019; Eiben et al., 2019], but mainly in the context of tractability results. However, nested dynamic programming was recently also discussed for an extension of answer set programming [Hecher et al., 2020a].

8.2 Future Work

This work gives rise to plenty of future work, which can be witnessed also by the increasing interest of research that is leaning towards the parameter treewidth[1].

In the last couple of years, there was a movement towards proving the correctness of solvers or showing at least that certain solver runs are correct. This resulted in several proof formats for solvers deciding satisfiability [Gelder, 2008; Goldberg and Novikov, 2003] and further problem formalisms [Heule et al., 2013; Wetzler et al., 2014; Heule et al., 2014; Lonsing and Egly, 2018b]. In the light of these works, the resulting initiative on explainability for artificial intelligence, and a recent proof logging format adapted for answer set programming [Alviano et al., 2019b], it might be interesting to consider

[1] Treewidth is mentioned in over 20,400 results on Google Scholar (queried on February 15, 2021).

applying and adapting such formats for the algorithms of Chapter 3 or further approaches based based on dynamic programming.

Decomposition-guided reductions were proposed in Chapter 4 as a tool for reducing problems to Boolean satisfiability, but also in order to establish lower bounds under the ETH. This raises the question of the particular strengths, weaknesses and limits of this type of reductions. To increase the flexibility of such reductions, one could allow to also "generate" additional auxiliary tree decomposition nodes, thereby slightly weakening the condition that the trees of the decompositions coincide, cf. Figures 4.1 and 4.2.

Chapter 5 of this work mainly focused on lower bounds under the exponential time hypothesis as given in Hypothesis 2.1. However, there are also extensions and stronger versions [Impagliazzo and Paturi, 2001] of the ETH that, while controversial, would result in more precise lower bounds. It would be interesting to establish lower bounds under this stronger assumption and then analyze how "good" existing upper bounds like those of Chapter 3 are compared to the resulting lower bounds. Recall that in Sections 5.1.3 and 5.2.4, we already discussed quite detailed potentials of future work. In particular, to further improve the methodology on showing lower bounds as discussed in Chapter 6, it might be interesting to extend it towards "non-canonical" lower bounds. For answer set programming, there is still an open question of whether the treewidth is in strong correlation with the resolution width, as this is the case for Sat solvers [Atserias et al., 2011] and Asp solvers are highly based on Sat solvers.

Finally, recall that dynamic programming can be applied to several other problems and formalisms. According to our observations of Chapter 7, especially counting problems seem to benefit from the structure that is guided by means of tree decompositions. However, nested dynamic programming in combination with hybrid solving might be also sufficiently applicable for decision problems, given that suitable abstractions are used. As a result, we propose further works in this direction in order to get a better picture of abstractions that might lead to efficient solver runs. These abstractions should be probably analyzed in a domain-specific and problem-specific setting, which should further improve the whole approach.

List of Algorithms

Bibliography

Abels, D., Jordi, J., Ostrowski, M., Schaub, T., Toletti, A., and Wanko, P. (2019). Train scheduling with hybrid ASP. In *15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 11481 of *Lecture Notes in Computer Science*, pages 3–17. Springer.

Abseher, M., Musliu, N., and Woltran, S. (2017). htd - A Free, Open-Source Framework for (Customized) Tree Decompositions and Beyond. In *14th International Conference on the Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*, volume 10335 of *Lecture Notes in Computer Science*, pages 376–386. Springer.

Alviano, M., Amendola, G., Dodaro, C., Leone, N., Maratea, M., and Ricca, F. (2019a). Evaluation of Disjunctive Programs in WASP. In *15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 11481 of *Lecture Notes in Computer Science*, pages 241–255. Springer.

Alviano, M. and Dodaro, C. (2016). Completion of Disjunctive Logic Programs. In *25th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 886–892. IJCAI/AAAI Press.

Alviano, M., Dodaro, C., Fichte, J. K., Hecher, M., Philipp, T., and Rath, J. (2019b). Inconsistency Proofs for ASP: The ASP - DRUPE Format. *Theory and Practice of Logic Programming*, 19(5-6):891–907.

Amendola, G., Ricca, F., and Truszczyński, M. (2019). Beyond NP: Quantifying over Answer Sets. *Theory and Practice of Logic Programming*, 19(5-6):705–721.

Ansótegui, C., Bonet, M. L., Giráldez-Cru, J., Levy, J., and Simon, L. (2019). Community Structure in Industrial SAT Instances. *Journal of Artificial Intelligence Research*, 66:443–472.

Atserias, A., Fichte, J. K., and Thurley, M. (2011). Clause-Learning Algorithms with Many Restarts and Bounded-Width Resolution. *Journal of Artificial Intelligence Research*, 40:353–373.

Atserias, A. and Oliva, S. (2014). Bounded-width QBF is PSPACE-complete. *Journal of Computer and System Sciences*, 80(7):1415–1429.

Audemard, G. and Simon, L. (2009). Predicting Learnt Clauses Quality in Modern SAT Solvers. In *21st International Joint Conference on Atrtificial Intelligence (IJCAI)*, pages 399–404.

Aziz, R. A. (2015). *Answer Set Programming: Founded Bounds and Model Counting*. PhD thesis, Department of Computing and Information Systems, The University of Melbourne.

Aziz, R. A., Chu, G., Muise, C., and Stuckey, P. (2015). #(∃)SAT: Projected Model Counting. In *18th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 121–137. Springer.

Bannach, M. and Berndt, S. (2019). Practical Access to Dynamic Programming on Tree Decompositions. *Algorithms*, 12(8):172.

Bauland, M., Mundhenk, M., Schneider, T., Schnoor, H., Schnoor, I., and Vollmer, H. (2011). The tractability of model checking for LTL: The good, the bad, and the ugly fragments. *ACM Transactions on Computational Logic*, 12(2):13:1–13:28.

Bayardo, R. J. and Schrag, R. (1997). Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *14th National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI)*, pages 203–208. AAAI Press / The MIT Press.

Bellman, R. (1954). Some Applications of the Theory of Dynamic Programming - A Review. *Operations Research*, 2(3):275–288.

Ben-Eliyahu, R. and Dechter, R. (1994). Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12(1):53–87.

Bertelè, U. and Brioschi, F. (1969). Contribution to nonserial dynamic programming. *Journal of Mathematical Analysis and Applications*, 28(2):313–325.

Bertelè, U. and Brioschi, F. (1972). *Nonserial Dynamic Programming*. Academic Press, Inc.

Bertelè, U. and Brioschi, F. (1973). On Non-serial Dynamic Programming. *Journal of Combinatorial Theory Series A*, 14(2):137–148.

Besin, V. (2020). Advancing a System for Counting Problems based on DBMS and Tree Decompositions. Bachelor's Thesis, Faculty of Informatics, TU Wien, Austria.

Bichler, M., Morak, M., and Woltran, S. (2020). lpopt: A Rule Optimization Tool for Answer Set Programming. *Fundamenta Informaticae*, 177(3-4):275–296.

Bidoít, N. and Froidevaux, C. (1991). Negation by default and unstratifiable logic programs. *Theoretical Computer Science*, 78(1):85–112.

Biere, A. (2008). PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97.

Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors (2009). *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Bliem, B., Charwat, G., Hecher, M., and Woltran, S. (2016a). D-FLAT^2: Subset Minimization in Dynamic Programming on Tree Decompositions Made Easy. *Fundamenta Informaticae*, 147(1):27–61.

Bliem, B., Morak, M., Moldovan, M., and Woltran, S. (2020). The Impact of Treewidth on Grounding and Solving of Answer Set Programs. *Journal of Artificial Intelligence Research*, 67:35–80.

Bliem, B., Ordyniak, S., and Woltran, S. (2016b). Clique-Width and Directed Width Measures for Answer-Set Programming. In *22nd European Conference on Artificial Intelligence (ECAI)*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, pages 1105–1113. IOS Press.

Bodlaender, H. L. (1988). Dynamic Programming on Graphs with Bounded Treewidth. In *15th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 317 of *Lecture Notes in Computer Science*, pages 105–118. Springer.

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

Bodlaender, H. L., Drange, P. G., Dregi, M. S., Fomin, F. V., Lokshtanov, D., and Pilipczuk, M. (2016). A c^k n 5-Approximation Algorithm for Treewidth. *SIAM Journal on Computing*, 45(2):317–378.

Bodlaender, H. L. and Kloks, T. (1996). Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms*, 21(2):358–402.

Bodlaender, H. L. and Koster, A. M. (2008). Combinatorial Optimization on Graphs of Bounded Treewidth. *Computer Journal*, 51(3):255–269.

Bomanson, J. (2017). lp2normal - A Normalization Tool for Extended Logic Programs. In *14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 10377 of *Lecture Notes in Computer Science*, pages 222–228. Springer.

Bomanson, J., Gebser, M., Janhunen, T., Kaufmann, B., and Schaub, T. (2016). Answer Set Programming Modulo Acyclicity. *Fundamenta Informaticae*, 147(1):63–91.

Bomanson, J. and Janhunen, T. (2013). Normalizing Cardinality Rules Using Merging and Sorting Constructions. In *12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 8148 of *Lecture Notes in Computer Science*, pages 187–199. Springer.

Bondy, J. A. and Murty, U. S. R. (2008). *Graph theory*, volume 244 of *Graduate Texts in Mathematics*. Springer.

Brewka, G., Eiter, T., and Truszczyński, M. (2011). Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103.

Bulatov, A. A. (2017). A Dichotomy Theorem for Nonuniform CSPs. In *58th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 319–330. IEEE Computer Society.

Burchard, J., Schubert, T., and Becker, B. (2015). Laissez-Faire Caching for Parallel #SAT Solving. In *18th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 9340 of *Lecture Notes in Computer Science*, pages 46–61. Springer.

Burchard, J., Schubert, T., and Becker, B. (2016). Distributed Parallel #SAT Solving. In *18th IEEE International Conference on Cluster Computing (CLUSTER)*, pages 326–335. IEEE Computer Society.

Cabalar, P., Fandinno, J., Garea, J., Romero, J., and Schaub, T. (2020). eclingo : A Solver for Epistemic Logic Programs. *Theory and Practice of Logic Programming*, 20(6):834–847.

Capelli, F. and Mengel, S. (2019). Tractable QBF by Knowledge Compilation. In *36th International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 126 of *Leibniz International Proceedings in Informatics*, pages 18:1–18:16. Dagstuhl Publishing.

Chakraborty, S., Fremont, D. J., Meel, K. S., Seshia, S. A., and Vardi, M. Y. (2014). Distribution-Aware Sampling and Weighted Model Counting for SAT. In *28th AAAI Conference on Artificial Intelligence (AAAI)*, pages 1722–1730. AAAI Press.

Chakraborty, S., Meel, K. S., and Vardi, M. Y. (2016). Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Solver Calls. In *25th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3569–3576. AAAI Press.

Charwat, G. and Woltran, S. (2017). Expansion-based QBF Solving on Tree Decompositions. In *24th International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (RCRA)*, volume 2011 of *CEUR Workshop Proceedings*, pages 16–26. CEUR-WS.org.

Charwat, G. and Woltran, S. (2019). Expansion-based QBF Solving on Tree Decompositions. *Fundamenta Informaticae*, 167(1-2):59–92.

Chen, H. (2004). Quantified Constraint Satisfaction and Bounded Treewidth. In *16th European Conference on Artificial Intelligence (ECAI)*, pages 161–165. IOS Press.

Chimani, M., Mutzel, P., and Zey, B. (2012). Improved Steiner tree algorithms for bounded treewidth. *Journal of Discrete Algorithms*, 16:67–78.

Clark, K. L. (1977). Negation as Failure. In *Logic and Data Bases*, Advances in Data Base Theory, pages 293–322. Plemum Press.

Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Journal of the ACM*, 13(6):377–387.

Cook, S. A. (1971). The Complexity of Theorem-Proving Procedures. In *3rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 151–158. ACM.

Courcelle, B. (1990). Graph Rewriting: An Algebraic and Logic Approach. In *Handbook of Theoretical Computer Science, Vol. B*, pages 193–242. Elsevier.

Courcelle, B., Makowsky, J. A., and Rotics, U. (2001). On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. *Discrete Applied Mathematics*, 108(1-2):23–52.

Creignou, N., Ktari, R., Meier, A., Müller, J., Olive, F., and Vollmer, H. (2019). Parameterised Enumeration for Modification Problems. *Algorithms*, 12(9):189.

Creignou, N., Meier, A., Müller, J.-S., Schmidt, J., and Vollmer, H. (2017). Paradigms for Parameterized Enumeration. *Theoretical Computer Science*, 60(4):737–758.

Creignou, N. and Vollmer, H. (2015). Parameterized Complexity of Weighted Satisfiability Problems: Decision, Enumeration, Counting. *Fundamenta Informaticae*, 136(4):297–316.

Cygan, M., Fomin, F. V., Kowalik, Ł., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., and Saurabh, S. (2015). *Parameterized Algorithms*. Springer.

Darwiche, A. (2004). New Advances in Compiling CNF to Decomposable Negation Normal Form. In *16th Eureopean Conference on Artificial Intelligence (ECAI)*, pages 318–322. IOS Press.

Darwiche, A. (2011). SDD: A New Canonical Representation of Propositional Knowledge Bases. In *22nd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 819–826. AAAI Press/IJCAI.

Davis, M., Logemann, G., and Loveland, D. W. (1962). A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397.

Davis, M. and Putnam, H. (1960). A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215.

de Haan, R. (2019). *Parameterized Complexity in the Polynomial Hierarchy - Extending Parameterized Complexity Theory to Higher Levels of the Hierarchy*, volume 11880 of *Lecture Notes in Computer Science*. Springer.

Dechter, R. (2006). Tractable Structures for Constraint Satisfaction Problems. In *Handbook of Constraint Programming*, volume I, chapter 7, pages 209–244. Elsevier.

Dell, H., Komusiewicz, C., Talmon, N., and Weller, M. (2017). The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. In *12th International Symposium on Parameterized and Exact Computation (IPEC)*, Leibniz International Proceedings in Informatics, pages 30:1—30:13. Dagstuhl Publishing.

Dell, H., Roth, M., and Wellnitz, P. (2019). Counting Answers to Existential Questions. In *46th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 132 of *LIPIcs*, pages 113:1–113:15. Dagstuhl Publishing.

Diestel, R. (2012). *Graph Theory, 4th Edition*, volume 173 of *Graduate Texts in Mathematics*. Springer.

Dix, J., Gottlob, G., and Marek, V. W. (1996). Reducing Disjunctive to Non-Disjunctive Semantics by Shift-Operations. *Fundamenta Informaticae*, 28(1-2):87–100.

Dodaro, C., Elder, G. A., Faber, W., Fandinno, J., Gebser, M., Hecher, M., LeBlanc, E., Morak, M., and Zangari, J., editors (2020). *International Conference on Logic Programming 2020 Workshop Proceedings co-located with 36th International Conference on Logic Programming (ICLP)*, volume 2678 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Downey, R., Flum, J., Grohe, M., and Weyer, M. (2007). Bounded fixed-parameter tractability and reducibility. *Annals of Pure And Applied Logic*, 148(1-3):1–19.

Downey, R. G. and Fellows, M. R. (2013). *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer.

Dreyfus, S. E. and Law, A. M. (1977). *Art and Theory of Dynamic Programming*. Academic Press, Inc.

Dueñas-Osorio, L., Meel, K. S., Paredes, R., and Vardi, M. Y. (2017). Counting-Based Reliability Estimation for Power-Transmission Grids. In *31th AAAI Conference on Artificial Intelligence (AAAI)*, pages 4488–4494. AAAI Press.

Dung, P. M. (1995). On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–357.

Durand, A., Hermann, M., and Kolaitis, P. G. (2005). Subtractive reductions and complete problems for counting complexity classes. *Theoretical Computer Science*, 340(3):496–513.

Dvořák, W., Pichler, R., and Woltran, S. (2012). Towards fixed-parameter tractable algorithms for abstract argumentation. *Artificial Intelligence*, 186:1–37.

Dzulfikar, M. A., Fichte, J. K., and Hecher, M. (2019). The PACE 2019 Parameterized Algorithms and Computational Experiments Challenge: The Fourth Iteration (Invited Paper). In *14th International Symposium on Parameterized and Exact Computation (IPEC)*, volume 148 of *Leibniz International Proceedings in Informatics*, pages 25:1–25:23. Dagstuhl Publishing.

Egly, U., Eiter, T., Tompits, H., and Woltran, S. (2000). Solving Advanced Reasoning Tasks Using Quantified Boolean Formulas. In *17th National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI 2000)*, pages 417–422. AAAI Press / The MIT Press.

Eiben, E., Ganian, R., Hamm, T., and Kwon, O. (2019). Measuring what Matters: A Hybrid Approach to Dynamic Programming with Treewidth. In *44th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 138 of *lipics*, pages 42:1–42:15. Dagstuhl Publishing.

Eiben, E., Ganian, R., and Ordyniak, S. (2018). Small Resolution Proofs for QBF using Dependency Treewidth. In *35th Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 96 of *Leibniz International Proceedings in Informatics*, pages 28:1–28:15. Dagstuhl Publishing.

Eiben, E., Ganian, R., and Ordyniak, S. (2020). Using decomposition-parameters for QBF: Mind the prefix! *Journal of Computer and System Sciences*, 110:1–21.

Eiter, T., Faber, W., Fink, M., and Woltran, S. (2007). Complexity results for answer set programming with bounded predicate arities and implications. *Annals of Mathematics and Artificial Intelligence*, 51(2-4):123–165.

Eiter, T. and Gottlob, G. (1995a). On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15(3–4):289–323.

Eiter, T. and Gottlob, G. (1995b). The Complexity of Logic-Based Abduction. *Journal of the ACM*, 42(1):3–42.

Elberfeld, M., Jakoby, A., and Tantau, T. (2010). Logspace versions of the theorems of Bodlaender and Courcelle. In *51st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 143–152. IEEE Computer Society.

Elffers, J., Giráldez-Cru, J., Gocht, S., Nordström, J., and Simon, L. (2018). Seeking Practical CDCL Insights from Theoretical SAT Benchmarks. In *27th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1300–1308. ijcai.org.

Elmasri, R. and Navathe, S. B. (2016). *Fundamentals of Database Systems*. Prentice Hall, 7th edition.

Ermon, S., Gomes, C. P., and Selman, B. (2012). Uniform Solution Sampling Using a Constraint Solver As an Oracle. In *28th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 255–264. AUAI Press.

Fages, F. (1994). Consistency of Clark's completion and existence of stable models. *Logical Methods in Computer Science*, 1(1):51–60.

Fagin, R. (1974). Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. In *7th Symposia in Applied Mathematics (SIAM)*. AMS.

Fandinno, J. and Hecher, M. (2021). Treewidth-Aware Complexity in ASP: Not all Positive Cycles are Equally Hard. In *35th AAAI Conference on Artificial Intelligence (AAAI)*. In Press.

Fassetti, F. and Palopoli, L. (2010). On the complexity of identifying head-elementary-set-free programs. *Theory and Practice of Logic Programming*, 10(1):113–123.

Feige, U., Hajiaghayi, M., and Lee, J. R. (2008). Improved Approximation Algorithms for Minimum Weight Vertex Separators. *SIAM Journal on Computing*, 38(2):629–657.

Fellows, M. R., Fomin, F. V., Lokshtanov, D., Rosamond, F. A., Saurabh, S., Szeider, S., and Thomassen, C. (2011). On the complexity of some colorful problems parameterized by treewidth. *Information and Computation*, 209(2):143–153.

Ferguson, A. and O'Sullivan, B. (2007). Quantified Constraint Satisfaction Problems: From Relaxations to Explanations. In *26th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 74–79. AAAI Press.

Fichte, J. K. and Hecher, M. (2018). Exploiting Treewidth for Counting Projected Answer Sets. In *16th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 639–640. AAAI Press.

Fichte, J. K. and Hecher, M. (2019). Treewidth and Counting Projected Answer Sets. In *15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 11481 of *Lecture Notes in Computer Science*, pages 105–119. Springer.

Fichte, J. K. and Hecher, M. (2020). Counting with Bounded Treewidth: Meta Algorithm and Runtime Guarantees. In *18th International Workshop on Non-Monotonic Reasoning (NMR)*, pages 9–18.

Fichte, J. K., Hecher, M., and Hamiti, F. (2020a). The Model Counting Competition 2020. *CoRR*, abs/2012.01323.

Fichte, J. K., Hecher, M., and Kieler, M. F. I. (2020b). Treewidth-Aware Quantifier Elimination and Expansion for QCSP. In *26th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 12333 of *Lecture Notes in Computer Science*, pages 248–266. Springer.

Fichte, J. K., Hecher, M., Lodha, N., and Szeider, S. (2018a). An SMT Approach to Fractional Hypertree Width. In *24th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 11008 of *Lecture Notes in Computer Science*, pages 109–127. Springer.

Fichte, J. K., Hecher, M., and Meier, A. (2019a). Counting Complexity for Reasoning in Abstract Argumentation. In *33rd AAAI Conference on Artificial Intelligence (AAAI)*, pages 2827–2834. AAAI Press.

Fichte, J. K., Hecher, M., and Meier, A. (2021a). Knowledge-Base Degrees of Inconsistency: Complexity and Counting. In *35th AAAI Conference on Artificial Intelligence (AAAI)*. In Press.

Fichte, J. K., Hecher, M., Morak, M., and Woltran, S. (2017a). Answer Set Solving with Bounded Treewidth Revisited. In *14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 10377 of *Lecture Notes in Computer Science*, pages 132–145. Springer.

Fichte, J. K., Hecher, M., Morak, M., and Woltran, S. (2017b). DynASP2.5: Dynamic Programming on Tree Decompositions in Action. In *12th International Symposium on Parameterized and Exact Computation (IPEC)*, volume 89 of *Leibniz International Proceedings in Informatics*, pages 17:1–17:13. Dagstuhl Publishing.

Fichte, J. K., Hecher, M., Morak, M., and Woltran, S. (2018b). Exploiting treewidth for projected model counting and its limits. In *21st International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 10929 of *Lecture Notes in Computer Science*, pages 165–184. Springer.

Fichte, J. K., Hecher, M., and Pfandler, A. (2020c). Lower Bounds for QBFs of Bounded Treewidth. In *35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 410–424. ACM.

Fichte, J. K., Hecher, M., and Schindler, I. (2020d). Default logic and bounded treewidth. *Information and Computation*. In Press.

Fichte, J. K., Hecher, M., and Szeider, S. (2020e). A Time Leap Challenge for SAT-Solving. In *26th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 12333 of *Lecture Notes in Computer Science*, pages 267–285. Springer.

Fichte, J. K., Hecher, M., and Szeider, S. (2020f). Breaking Symmetries with RootClique and LexTopSort. In *26th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 12333 of *Lecture Notes in Computer Science*, pages 286–303. Springer.

Fichte, J. K., Hecher, M., Thier, P., and Woltran, S. (2021b). Exploiting Database Management Systems and Treewidth for Counting. *Theory and Practice of Logic Programming*. In Press.

Fichte, J. K., Hecher, M., Woltran, S., and Zisser, M. (2018c). Weighted Model Counting on the GPU by Exploiting Small Treewidth. In *26th Annual European Symposium on Algorithms (ESA)*, volume 112 of *Leibniz International Proceedings in Informatics*, pages 28:1–28:16. Dagstuhl Publishing.

Fichte, J. K., Hecher, M., and Zisser, M. (2019b). An Improved GPU-Based SAT Model Counter. In *25th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 11802 of *Lecture Notes in Computer Science*, pages 491–509. Springer.

Fichte, J. K., Kronegger, M., and Woltran, S. (2019c). A multiparametric view on answer set programming. *Annals of Mathematics and Artificial Intelligence*, 86(1-3):121–147.

Fichte, J. K. and Szeider, S. (2015). Backdoors to Tractable Answer-Set Programming. *Artificial Intelligence*, 220(C):64–103.

Fischl, W., Gottlob, G., Longo, D. M., and Pichler, R. (2019). HyperBench: A Benchmark and Tool for Hypergraphs and Empirical Findings. In *38th ACM Symposium on Principles of Database Systems (PODS)*, pages 464–480. ACM.

Fischl, W., Gottlob, G., and Pichler, R. (2018). General and Fractional Hypertree Decompositions: Hard and Easy Cases. In *37th ACM Symposium on Principles of Database Systems (PODS)*, pages 17–32. ACM.

Flum, J. and Grohe, M. (2006). *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer.

Freuder, E. C. (1985). A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(4):755–761.

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *11th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, pages 97–104.

Ganian, R., Ramanujan, M. S., and Szeider, S. (2017). Combining Treewidth and Backdoors for CSP. In *34th Symposium on Theoretical Aspects of Computer Science (STACS)*, Leibniz International Proceedings in Informatics, pages 36:1–36:17. Dagstuhl Publishing.

Garcia-Molina, H., Ullman, J. D., and Widom, J. (2009). *Database systems: the complete book*. Pearson Education.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

Gebser, M., Janhunen, T., and Rintanen, J. (2014). Answer Set Programming as SAT modulo Acyclicity. In *21st European Conference on Artificial Intelligence (ECAI)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 351–356. IOS Press.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2012). *Answer Set Solving in Practice*. Morgan & Claypool.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2019). Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82.

Gebser, M., Lee, J., and Lierler, Y. (2007). Head-Elementary-Set-Free Logic Programs. In *9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 4483 of *Lecture Notes in Computer Science*, pages 149–161. Springer.

Gebser, M., Schaub, T., Thiele, S., and Veber, P. (2011). Detecting Inconsistencies in Large Biological Networks with Answer Set Programming. *Theory and Practice of Logic Programming*, 11(2-3):323–360.

Gelder, A. V. (2008). Verifying RUP Proofs of Propositional Unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics (ISAIM)*.

Gelfond, M. and Lifschitz, V. (1991). Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3/4):365–386.

Gocht, S., Nordström, J., and Yehudayoff, A. (2019). On Division Versus Saturation in Pseudo-Boolean Solving. In *28th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1711–1718. ijcai.org.

Goldberg, E. I. and Novikov, Y. (2003). Verification of Proofs of Unsatisfiability for CNF Formulas. In *DATE*, pages 10886–10891. IEEE Computer Society.

Gomes, C. P., Sabharwal, A., and Selman, B. (2009). Chapter 20: Model Counting. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 633–654. IOS Press.

Gottlob, G., Leone, N., and Scarcello, F. (2002a). Hypertree Decompositions and Tractable Queries. *Journal of Computer and System Sciences*, 64(3):579–627.

Gottlob, G., Pichler, R., and Wei, F. (2010). Bounded treewidth as a key to tractability of knowledge representation and reasoning. *Artificial Intelligence*, 174(1):105–132.

Gottlob, G. and Samer, M. (2009). A Backtracking-based Algorithm for Hypertree Decomposition. *ACM Journal of Experimental Algorithmics*, 13:1:1.1–1:1.19.

Gottlob, G., Scarcello, F., and Sideri, M. (2002b). Fixed-parameter complexity in AI and nonmonotonic reasoning. *Artificial Intelligence*, 138(1-2):55–86.

Grohe, M. (2017). *Descriptive Complexity, Canonisation, and Definable Graph Structure Theory*, volume 47. Cambridge University Press.

Grohe, M. and Marx, D. (2006). Constraint Solving via Fractional Edge Covers. In *17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 289–298. ACM Press.

Grohe, M. and Marx, D. (2014). Constraint solving via fractional edge covers. *ACM Transactions on Algorithms*, 11(1):Art. 4, 20.

Harvey, D., van der Hoeven, J., and Lecerf, G. (2016). Even faster integer multiplication. *Journal of Complexity*, 36:1–30.

Hecher, M. (2020). Treewidth-aware Reductions of Normal ASP to SAT - Is Normal ASP Harder than SAT after All? In *17th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 485–495. Winner of the Marco Cadoli Best Student Paper Award.

Hecher, M., Morak, M., and Woltran, S. (2020a). Structural Decompositions of Epistemic Logic Programs. In *34th AAAI Conference on Artificial Intelligence (AAAI)*, pages 2830–2837. AAAI Press.

Hecher, M., Thier, P., and Woltran, S. (2020b). Taming High Treewidth with Abstraction, Nested Dynamic Programming, and Database Technology. In *23rd International Conference on Theory and Applications of Satisfiability Testing SAT*, volume 12178 of *Lecture Notes in Computer Science*, pages 343–360. Springer.

Hemaspaandra, E. (2004). Dichotomy Theorems for Alternation-Bounded Quantified Boolean Formulas. *CoRR*, cs.CC/0406006.

Hemaspaandra, L. A. and Vollmer, H. (1995). The Satanic Notations: Counting Classes Beyond #P and Other Definitional Adventures. *SIGACT News*, 26(1):2–13.

Heule, M., Hunt Jr., W. A., and Wetzler, N. (2013). Verifying Refutations with Extended Resolution. In *24th International Conference on Automated Deduction (CADE)*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer.

Heule, M., Seidl, M., and Biere, A. (2014). A Unified Proof System for QBF Preprocessing. In *7th International Joint Conference (IJCAR)*, volume 8562 of *Lecture Notes in Computer Science*, pages 91–106. Springer.

Immerman, N. (1999). *Descriptive complexity*. Springer.

Impagliazzo, R. and Paturi, R. (2001). On the Complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375.

Impagliazzo, R., Paturi, R., and Zane, F. (2001). Which Problems Have Strongly Exponential Complexity? *Journal of Computer and System Sciences*, 63(4):512–530.

Jakl, M., Pichler, R., and Woltran, S. (2009). Answer-Set Programming with Bounded Treewidth. In *21st International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, pages 816–822.

Janhunen, T. (2006). Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16(1-2):35–86.

Janhunen, T. and Niemelä, I. (2016). The Answer Set Programming Paradigm. *AI Magazine*, 37(3):13–24.

Kieler, M. F. I. (2020). Trading Structural Dependency for Quantifier Depth on QBFs. Bachelor's Thesis, Faculty of Informatics, TU Dresden, Germany.

Kiesl, B., Rebola-Pardo, A., Heule, M. J. H., and Biere, A. (2020). Simulating Strong Practical Proof Systems with Extended Resolution. *Journal of Automated Reasoning*, 64(7):1247–1267.

Kiljan, K. and Pilipczuk, M. (2018). Experimental Evaluation of Parameterized Algorithms for Feedback Vertex Set. In *SEA*, volume 103 of *Leibniz International Proceedings in Informatics*, pages 12:1–12:12. Dagstuhl Publishing.

Klebanov, V., Manthey, N., and Muise, C. J. (2013). SAT-Based Analysis and Quantification of Information Flow in Programs. In *10th International Conference on Quantitative Evaluation of Systems (QEST)*, volume 8054 of *Lecture Notes in Computer Science*, pages 177–192. Springer.

Kleine Büning, H. and Lettman, T. (1999). *Propositional Logic: Deduction and Algorithms*, volume 48 of *Cambridge tracts in theoretical computer science*. Cambridge University Press.

Kloks, T. (1994). *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer.

Knuth, D. E. (1998). How fast can we multiply? In *The Art of Computer Programming*, volume 2 of *Seminumerical Algorithms*, chapter 4.3.3, pages 294–318. Addison-Wesley, 3 edition.

Korhonen, T., Berg, J., and Järvisalo, M. (2019). Solving Graph Problems via Potential Maximal Cliques: An Experimental Evaluation of the Bouchitté-Todinca Algorithm. *ACM Journal of Experimental Algorithmics*, 24(1):1.9:1–1.9:19.

Koriche, F., Lagniez, J.-M., Marquis, P., and Thomas, S. (2013). Knowledge Compilation for Model Counting: Affine Decision Trees. In *23rd International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI/AAAI.

Lackner, M. and Pfandler, A. (2012). Fixed-Parameter Algorithms for Finding Minimal Models. In *13th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 85–95. AAAI Press.

Lagniez, J. and Marquis, P. (2014). Preprocessing for Propositional Model Counting. In *28th AAAI Conference on Artificial Intelligence (AAAI)*, pages 2688–2694. AAAI Press.

Lagniez, J. and Marquis, P. (2019). A Recursive Algorithm for Projected Model Counting. In *33rd Conference on Artificial Intelligence*, pages 1536–1543. AAAI Press.

Lagniez, J.-M. and Marquis, P. (2017). An improved decision-DNNF compiler. In *26th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 667–673. ijcai.org.

Lagniez, J.-M., Marquis, P., and Szczepanski, N. (2018). DMC: A Distributed Model Counter. In *27th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1331–1338. AAAI Press.

Lampis, M., Mengel, S., and Mitsou, V. (2018). QBF as an Alternative to Courcelle's Theorem. In *21st International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 235–252. Springer.

Lampis, M. and Mitsou, V. (2017). Treewidth with a Quantifier Alternation Revisited. In *12th International Symposium on Parameterized and Exact Computation (IPEC)*, volume 89 of *Leibniz International Proceedings in Informatics*, pages 26:1–26:12. Dagstuhl Publishing.

Langer, A., Reidl, F., Rossmanith, P., and Sikdar, S. (2012). Evaluation of an MSO-Solver. In *14th Meeting on Algorithm Engineering & Experiments (ALENEX)*, pages 55–63. SIAM / Omnipress.

Levin, L. A. (1973). Universal Sequential Search Problems. *Problems of Information Transmission*, 9(3).

Lifschitz, V. and Razborov, A. A. (2006). Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 7(2):261–268.

Lin, F. and Zhao, J. (2003). On tight logic programs and yet another translation from normal logic programs to propositional logic. In *Proceedings of the 18th International Joint Conference on Artificial intelligence (IJCAI'03)*, pages 853–858. Morgan Kaufmann.

Lin, F. and Zhao, X. (2004a). On Odd and Even Cycles in Normal Logic Programs. In McGuinness, D. L. and Ferguson, G., editors, *19th National Conference on Artificial Intelligence (AAAI)*, pages 80–85. AAAI Press / The MIT Press.

Lin, F. and Zhao, Y. (2004b). ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137.

Liu, J., Zhong, W., and Jiao, L. (2006). Comments on "The 1993 DIMACS graph coloring Challenge" and "Energy function-based approaches to graph Coloring". *IEEE Transactional on Neural Networks and Learning Systems*, 17(2):533.

Lokshtanov, D., Marx, D., and Saurabh, S. (2011). Slightly Superexponential Parameterized Problems. In *22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 760–776. SIAM.

Lokshtanov, D., Marx, D., and Saurabh, S. (2018). Slightly Superexponential Parameterized Problems. *SIAM Journal on Computing*, 47(3):675–702.

Lonc, Z. and Truszczyński, M. (2003). Fixed-parameter complexity of semantics for logic programs. *ACM Transactions on Computational Logic*, 4(1):91–119.

Lonsing, F. and Egly, U. (2018a). Evaluating QBF Solvers: Quantifier Alternations Matter. In *24th International Conference on Principles and Practice of Constraint Programming (CP 2018)*, volume 11008 of *Lecture Notes in Computer Science*, pages 276–294. Springer.

Lonsing, F. and Egly, U. (2018b). QRAT+: Generalizing QRAT by a More Powerful QBF Redundancy Property. In *9th International Joint Conference (IJCAR)*, volume 10900 of *Lecture Notes in Computer Science*, pages 161–177. Springer.

Maniu, S., Senellart, P., and Jog, S. (2019). An Experimental Study of the Treewidth of Real-World Graph Data (Extended Version). *CoRR*, abs/1901.06862.

Marek, W. and Truszczyński, M. (1991). Autoepistemic logic. *Journal of the ACM*, 38(3):588–619.

Marx, D. and Mitsou, V. (2016). Double-Exponential and Triple-Exponential Bounds for Choosability Problems Parameterized by Treewidth. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *Leibniz International Proceedings in Informatics*, pages 28:1–28:15. Dagstuhl Publishing.

McCarthy, J. (1980). Circumscription - A Form of Non-Monotonic Reasoning. *Artificial Intelligence*, 13(1-2):27–39.

Meier, A., Schindler, I., Schmidt, J., Thomas, M., and Vollmer, H. (2015). On the parameterized complexity of non-monotonic logics. *Archive for Mathematical Logic*, 54(5-6):685–710.

Muise, Christian J .and McIlraith, S. A., Beck, J. C., and Hsu, E. I. (2012). Dsharp: Fast d-DNNF Compilation with sharpSAT. In *25th Canadian Conference on Artificial Intelligence (AI)*, volume 7310 of *Lecture Notes in Computer Science*, pages 356–361. Springer.

Niedermeier, R. (2006). *Invitation to Fixed-Parameter Algorithms*, volume 31 of *Oxford Lecture Series in Mathematics and its Applications*. Oxford University Press.

Ordyniak, S. and Szeider, S. (2013). Parameterized Complexity Results for Exact Bayesian Network Structure Learning. *Journal of Artificial Intelligence Research*, 46:263–302.

Oztok, U. and Darwiche, A. (2015). A Top-Down Compiler for Sentential Decision Diagrams. In *24th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3141–3148. AAAI Press.

Pan, G. and Vardi, M. Y. (2006). Fixed-Parameter Hierarchies inside PSPACE. In *21th IEEE Symposium on Logic in Computer Science (LICS)*, pages 27–36. IEEE Computer Society.

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

Pichler, R., Rümmele, S., Szeider, S., and Woltran, S. (2014). Tractable answer-set programming with weight constraints: bounded treewidth is not enough. *Theory and Practice of Logic Programming*, 14(2).

Pichler, R., Rümmele, S., and Woltran, S. (2010). Counting and Enumeration Problems with Bounded Treewidth. In *16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 6355 of *Lecture Notes in Computer Science*, pages 387–404. Springer.

PostgreSQL Global Development Group (2021). Postgresql documentation 12. Available at: `https://www.postgresql.org/docs/12/queries-with.html`.

Pulina, L. and Seidl, M. (2019). The 2016 and 2017 QBF solvers evaluations (QBFEVAL'16 and QBFEVAL'17). *Artificial Intelligence*, 274:224–248.

Robertson, N. and Seymour, P. D. (1983). Graph Minors. I. Excluding a Forest. *Journal of Combinatorial Theory, Series B*, 35(1):39–61.

Robertson, N. and Seymour, P. D. (1984). Graph Minors. III. Planar Tree-Width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64.

Robertson, N. and Seymour, P. D. (1985). Graph Minors – a Survey. In *Surveys in Combinatorics 1985: Invited Papers for the 10th British Combinatorial Conference*, London Mathematical Society Lecture Note Series, pages 153–171. Cambridge University Press.

Robertson, N. and Seymour, P. D. (1986). Graph Minors. II. Algorithmic Aspects of Tree-Width. *Journal of Algorithms*, 7(3):309–322.

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

Roth, D. (1996). On the Hardness of Approximate Reasoning. *Artificial Intelligence*, 82(1–2).

Samer, M. and Szeider, S. (2009). Fixed-Parameter Tractability. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 425–454. IOS Press.

Samer, M. and Szeider, S. (2010). Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50–64.

Sang, T., Bacchus, F., Beame, P., Kautz, H., and Pitassi, T. (2004). Combining Component Caching and Clause Learning for Effective Model Counting. In *7th International Conference on Theory and Applications of Satisfiability Testing (SAT)*.

Schaefer, T. J. (1978). The Complexity of Satisfiability Problems. In *10th Annual ACM Symposium on Theory of Computing (STOC)*, pages 216–226. ACM.

Schaub, T. and Woltran, S. (2018). Special Issue on Answer Set Programming. *KI - Kuenstliche Intelligenz*, 32(2-3):101–103.

Scheffler, P. (1994). A Practical Linear Time Algorithm for Disjoint Paths in Graphs with Bounded Tree-width. Technical Report Report-396-1994, TU Berlin. Available at: `http://www.redaktion.tu-berlin.de/fileadmin/i26/download/AG_DiskAlg/FG_KombOptGraphAlg/preprints/1994/Report-396-1994.ps.gz`.

Schidler, A. (2018). A solver for the Steiner tree problem with few terminals. Master's thesis, Faculty of Informatics, TU Wien, Austria.

Sharma, S., Roy, S., Soos, M., and Meel, K. S. (2019). GANAK: A Scalable Probabilistic Exact Model Counter. In *28th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1169–1176. ijcai.org.

Shen, Y. and Eiter, T. (2017). Evaluating Epistemic Negation in Answer Set Programming (Extended Abstract). In *26th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 5060–5064. ijcai.org.

Silva, J. P. M. and Sakallah, K. A. (1996). GRASP - a new search algorithm for satisfiability. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 220–227. IEEE Computer Society / ACM.

Stockmeyer, L. J. (1976). The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22.

Stockmeyer, L. J. and Meyer, A. R. (1973). Word problems requiring exponential time. In *5th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–9. ACM.

Syrjänen, T. (2002). Lparse 1.0 User's Manual. `tcs.hut.fi/Software/smodels/lparse.ps`.

Tamaki, H. (2019). Positive-instance driven dynamic programming for treewidth. *Journal of Combinatorial Optimization*, 37(4):1283–1311.

Thurley, M. (2006). sharpSAT – Counting Models with Advanced Component Caching and Implicit BCP. In *9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 424–429. Springer.

Toda, S. (1991). PP is as Hard as the Polynomial-Time Hierarchy. *SIAM Journal on Computing*, 20(5):865–877.

Toda, T. and Soh, T. (2015). Implementing Efficient All Solutions SAT Solvers. *ACM Journal of Experimental Algorithmics*, 21:1.12. Special Issue SEA 2014.

Truszczyński, M. (2007). Logic Programming for Knowledge Representation. In Dahl, V. and Niemelä, I., editors, *23rd International Conference on Logic Programming (ICLP)*, volume 4670 of *Lecture Notes in Computer Science*, pages 76–88. Springer.

Truszczyński, M. (2010). Reducts of propositional theories, satisfiability relations, and generalizations of semantics of logic programs. *Artificial Intelligence*, 174(16-17):1285–1306.

Truszczyński, M. (2011). Trichotomy and dichotomy results on the complexity of reasoning with disjunctive logic programs. *Theory and Practice of Logic Programming*, 11(6):881–904.

Ullman, J. D. (1989). *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press.

Valiant, L. (1979a). The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421.

Valiant, L. G. (1979b). The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201.

Wetzler, N., Heule, M., and Hunt Jr., W. A. (2014). DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *17th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer.

Weyer, M. (2004). Bounded fixed-parameter tractability: The case 2poly(k). In *Parameterized and Exact Computation, First International Workshop (IWPEC)*, volume 3162 of *Lecture Notes in Computer Science*, pages 49–60. Springer.

Wrathall, C. (1976). Complete Sets and the Polynomial-Time Hierarchy. *Theoretical Computer Science*, 3(1):23–33.

Zisser, M. (2018). Solving the #SAT problem on the GPU with dynamic programming and OpenCL. Master's thesis, Faculty of Informatics, TU Wien, Austria.