FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Treewidth in Non-Ground Answer Set Solving and Alliance Problems in Graphs

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Bernhard Bliem
Matrikelnummer 0725395

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran

Diese Dissertation haben begutachtet:

Francesco Scarcello

Gerhard Woeginger

Wien, 3. August 2017

Bernhard Bliem

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

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Treewidth in Non-Ground Answer Set Solving and Alliance Problems in Graphs

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Bernhard Bliem
Registration Number 0725395

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran

The dissertation has been reviewed by:

Francesco Scarcello	Gerhard Woeginger

Vienna, 3rd August, 2017

Bernhard Bliem

Erklärung zur Verfassung der Arbeit

Bernhard Bliem
Lerchengasse 31/5
1080 Wien
Österreich

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

Wien, 3. August 2017

Bernhard Bliem

Acknowledgements

First and foremost, I would like to thank my advisor, Stefan Woltran, for supporting me over the last couple of years. I particularly appreciated the friendly yet professional work atmosphere and the fact that, even when things were busy, he always made me feel valued not just as a student and an employee, but also as a person. It's hard to think of a thing that I disliked about working with him – except maybe for his handwriting, which sometimes, to both our amusement, not even he could decipher.

It was also a pleasure to work with my colleagues at TU Wien. I would especially like to mention coauthors of papers I wrote during my time here: Michael Abseher, Günther Charwat, Frederico Dusberger, Markus Hecher, Marius Moldovan, Michael Morak, Sebastian Ordyniak and Reinhard Pichler. Most of the results that I present in this work would not exist without them. (Maybe they will agree that this would be a bad thing.) Also, thanks to the administrative staff who are always there to help.

In 2015, I had the pleasure of spending two months at Rolf Niedermeier's group at TU Berlin. I learned a lot in that time, and I would like to thank Rolf for his support and guidance. Thanks also to Robert Bredereck, with whom I coauthored a paper that was a lot of fun to work on but unfortunately did not make it into this thesis. Moreover, I would like to thank Torsten Schaub and his group, in particular Benjamin Kaufmann, one of my coauthors, for interesting discussions and the warm welcome in Potsdam. I would also like to thank Bruno Courcelle for clearing up a question I had about MSO transductions.

Finally, I would like to thank my friends and family for being there for me. Thanks to Manuel, my partner, for bringing so much life into my world. Also, I would like to thank my sister Dorothee for always being open to philosophical discussions or laughing about the absurdities of life. Most of all, I am indebted to my parents, Adelheid and Josef, who gave me all the freedom I needed and yet never made me doubt that I can always rely on their unconditional support and love.

Kurzfassung

Viele praktisch relevante Probleme befinden sich auf der zweiten Stufe der Polynomiellen Hierarchie und sind daher von sehr hoher Komplexität; dennoch sind Programme, die solche Probleme lösen, in der Praxis oft überraschend effizient – sogar auf großen Probleminstanzen. Einer der Gründe ist, dass in der Realität auftretende Instanzen häufig besondere strukturelle Eigenschaften haben, durch die sich manchmal die theoretisch schlimmsten Laufzeiten vermeiden lassen. Das Ziel dieser Arbeit ist es, neue Erkenntnisse über das Lösen solcher schwierigen Probleme durch das Ausnutzen des strukturellen Parameters *Baumweite* zu erhalten.

Ein beliebtes Werkzeug, um Probleme bis hin zur zweiten Stufe der Polynomiellen Hierarchie zu lösen, ist *Answer Set Programming* (ASP), welches durch eine komfortable Sprache und effiziente Implementierungen hervorsticht. Der übliche Arbeitsablauf in ASP ist, eine Problemspezifikation in ASP zu schreiben und diese zusammen mit einer Instanz in ein sogenanntes *grundiertes Programm* umzuwandeln, dessen Ergebnisse anschließend von einem *Lösungsprogramm* berechnet werden. Neueste Untersuchungen ergaben, dass die Leistung moderner ASP-Lösungsprogramme massiv davon profitiert, wenn das grundierte Programm kleine Baumweite aufweist. Leider ist das Grundieren – also die Umwandlung einer Instanz in ein grundiertes Programm – ein komplizierter, implementierungsabhängiger Prozess, weshalb es unklar ist, unter welchen Umständen das grundierte Programm kleine Baumweite besitzt. Der Einfluss des Grundierens auf die Baumweite wurde in bisherigen Untersuchungen weitgehend vernachlässigt.

Eine für die ASP-Forschung besonders interessante Klasse an Graphproblemen sind *Allianzprobleme*. Diese suchen nach Gruppen von Knoten, die einander auf eine bestimmte Weise Beistand leisten können. Einige Allianzprobleme sind auf der zweiten Stufe der Polynomiellen Hierarchie und benötigen fortgeschrittene Modellierungstechniken, um in ASP ausgedrückt werden zu können. Wichtige Allianzprobleme entbehrten bisher jedoch einer Komplexitätsanalyse, insbesondere in Bezug auf Baumweite.

Unsere Forschungsbeiträge sind die Folgenden: Wir zeigen auf, wie kleine Baumweite *implizit* in ASP genutzt werden kann – das bedeutet, ohne ein Programm schreiben zu müssen, welches Baumweite bewusst ausnutzt. Zu diesem Zweck definieren wir Klassen nichtgrundierter ASP-Programme, welche garantieren, dass das Grundieren beschränkte Baumweite beibehält. Außerdem stellen wir eine Methodik vor, mit welcher Algorithmen entworfen werden können, die *explizit* Baumweite berücksichtigen. Diese Methodik zielt eigens auf Probleme auf der zweiten Stufe der Polynomiellen Hierarchie ab und kann die Leistung im Vergleich zu naiven Ansätzen wesentlich steigern. Wir veranschaulichen einige unserer Techniken anhand von Allianzproblemen, für welche wir ferner verschiedene lange bestehende Komplexitätsfragen beantworten.

Abstract

Many practically relevant problems are at the second level of the polynomial hierarchy and thus highly intractable; yet programs solving them are often surprisingly efficient in practice even on large instances. One of the reasons is that problem instances occurring in the real world frequently have certain structural properties that sometimes allow us to avoid the worst-case running times. The goal of this thesis is to gain new insights into solving such hard problems by exploiting the structural parameter *treewidth*.

For solving problems up to the second level of the polynomial hierarchy, Answer Set Programming (ASP) has become popular due to its convenient language and efficient solvers. In ASP, the usual workflow is to write an ASP encoding for a problem and transform it together with a problem instance into a so-called *ground program*, which is then given to a *solver* that computes the solutions. Recent work has shown that the performance of state-of-the-art ASP solvers benefits greatly from the ground program having small treewidth. Unfortunately grounding, that is, the transformation of the instance into a ground program, is a complicated, implementation-dependent task, so it is not clear under which circumstances the ground program has small treewidth. The influence of grounding on the treewidth has largely been neglected in research so far.

A particularly interesting class of graph problems for research on ASP are *alliance problems*. These ask for groups of vertices that help each other out in a certain way. Some alliance problems are at the second level of the polynomial hierarchy and require advanced modeling techniques for encoding them in ASP. However, important alliance problems lacked a complexity analysis, in particular with regard to treewidth.

Our contributions are the following: We show how ASP users can take advantage of small treewidth *implicitly*, that is, without having to write a program that deliberately exploits treewidth. For this, we define classes of non-ground ASP programs that guarantee that grounding preserves bounded treewidth. Moreover, we present a methodology for designing algorithms that *explicitly* take treewidth into account. This is especially targeted at problems at the second level of the polynomial hierarchy and can substantially improve performance compared to naive approaches. We illustrate some of our techniques for alliance problems in graphs, where we also settle several long-standing questions regarding complexity.

Contents

CHAPTER 1

Introduction

In the face of vast amounts of data that are being generated and stored every day, many systems that process this data meet their limits. This is not only an issue of implementation details. Indeed, many practically relevant problems are NP-hard and, as is well known, there is strong evidence that it is impossible to write a tractable algorithm for any such problem. Yet we still need to solve hard computational tasks on large amounts of data.

One problem solving paradigm that has become quite popular for tackling computationally hard problems is Answer Set Programming (ASP) (Brewka, Eiter and Truszczyński 2011; Gebser et al. 2012; Marek and Truszczyński 1999; Lifschitz 2008). ASP is a declarative language that is especially attractive due to the fact that highly efficient systems are available (Gebser, Kaufmann and Schaub 2012; Gebser et al. 2007; Gebser et al. 2015; Alviano et al. 2013; Alviano et al. 2015; Leone et al. 2006; Alviano et al. 2011; Elkabani, Pontelli and Son 2005). These systems allow us to encode problems in a very convenient and succinct way.

The impressive performance increase of ASP systems in recent years now allows us to solve many hard problems also on quite large inputs. Indeed, ASP has been successfully employed for solving a great variety of computationally hard problems. However, there are several computational problems where even sophisticated state-of-the-art ASP systems struggle.

To see why ASP systems may perform quite well in practice on one problem whereas the performance on another problem of the same complexity can be significantly worse, it helps to consider the *parameterized* complexity of the problems (Downey and Fellows 1999; Flum and Grohe 2006; Cygan et al. 2015; Niedermeier 2006). This theoretical framework investigates the complexity of a problem not only in terms of the input size, but also of other parameters. The core idea of parameterized complexity theory is that a problem is *fixed-parameter tractable* (FPT) w.r.t. a parameter k of the instances if the problem admits an algorithm that runs in time $\mathcal{O}(f(k) \cdot n^c)$, where f is an arbitrary computable function that only depends on k, n is the input size and c is an arbitrary constant. By assuming that a certain parameter of the instances is bounded by a constant, many NP-hard problems become tractable in this sense.

In this work, we are particularly interested in the effect of *structural* parameters on the performance of ASP solvers, that is, parameters that take the relations between different parts of the input into account. We focus on the parameter *treewidth* (Robertson and Seymour 1984), which is a measure of the cyclicity of a graph. Intuitively, the smaller the treewidth of a graph, the closer the graph resembles a tree. It is well known that many graph problems become easy if we restrict the input to trees and it has turned out that for many important problems this even holds for the more general class of instances of bounded treewidth (cf., e.g., Arnborg, Lagergren and Seese 1991). Luckily, it has been observed that real-world instances exhibit small treewidth in many cases (Bodlaender 1993; Thorup 1998; Kornai and Tuza 1992). Moreover, treewidth is not only relevant for graph problems. It can also be applied to instances of all kinds of problems by choosing a suitable representation of the instance as a graph.

There have already been some investigations concerning treewidth and ASP (Gottlob, Pichler and Wei 2010a; Pichler et al. 2014; Fichte et al. 2017; Morak and Woltran 2012). These have mainly concerned ground ASP (i.e., ASP programs without variables, also known as propositional programs). Sadly, the case where ASP programs may contain variables has largely been neglected so far. Although treewidth plays a role in some work on decomposing non-ground ASP rules (Bichler, Morak and Woltran 2016; Bichler, Morak and Woltran 2017), there have been hardly any results on the effect of treewidth on ASP solving in the presence of variables. This is especially unfortunate as variables form the basis of the convenient language features provided by ASP systems, which are one of the main advantages of ASP from a user's perspective when compared to related approaches such as Boolean satisfiability (Sat).

To see why there are so far no investigations of the effect of treewidth on non-ground ASP programs, consider the typical workflow when using ASP to solve a problem: Once an encoding for a given problem is written in non-ground ASP, answer set solving is usually a two-step process. First the encoding, together with a set of input facts representing the problem instance, gets passed to a *grounder*, which transforms it into an equivalent propositional ASP program. In the second step, this ground program is evaluated by a *solver*. Since grounding can in most cases be seen as a rather simple preprocessing step whereas solving is usually the main workhorse, theoretical investigations have mainly focused on the task of solving propositional programs.

An important result regarding the complexity of ground ASP solving parameterized by treewidth is the fixed-parameter tractable algorithm by Jakl, Pichler and Woltran (2009) for deciding whether a program has an answer set. This algorithm follows the principle of *dynamic programming on tree decompositions*, which is very common for FPT algorithms parameterized by treewidth (Bodlaender 1997). The basic idea is the following: Given a graph G, a *tree decomposition* of G is a tree whose nodes correspond to subgraphs of G in such a way that certain conditions are fulfilled. (The exact conditions are not important for now and will be presented in detail in Chapter 2.) If the treewidth of a graph is bounded by a constant, then we can find (in linear time) a tree decomposition whose nodes correspond to subgraphs of constant size (Bodlaender 1996). We can then solve many problems by first applying brute force at each subgraph in order to solve a subproblem corresponding to this subgraph and then trying to combine the obtained partial solutions. If the treewidth of the graph is bounded, then we can afford this brute force approach because each of the considered subgraphs has bounded size.

The FPT algorithm by Jakl, Pichler and Woltran (2009) thus specifies a different way of ASP solving than the standard approaches, which implement a version of a concept called *conflict-driven clause learning* (Gebser, Kaufmann and Schaub 2012). This solving procedure was originally introduced for SAT solving and proved to be very successful also when adapted to ASP solving, where it now constitutes the heart of state-of-the-art solvers. Indeed the algorithm by Jakl, Pichler and Woltran (2009) has also been implemented and proposed as an alternative solver for ground ASP (Morak et al. 2010). For certain problems, this dynamic-programming-based solver was able to outperform state-of-the-art ASP solvers if the instances had a very small treewidth and the sizes of the instances were very large.

Although the encouraging results by Morak et al. (2010) confirmed that small treewidth can be successfully exploited for ASP solving in experimental settings, the restrictions on problems and instances that make this approach perform well were still too severe for most practical applications. The main obstacles that prevented this approach from being useful for a broad range of applications were the facts that, on the one hand, the naive dynamic programming approach involves an enormous overhead and, on the other hand, state-of-the-art ASP solvers often perform so well that the fixed-parameter tractability only pays off for instances of tremendous size. In fact, experiments in the work of Bliem et al. (2017) indicated that state-of-the-art ASP solvers are "sensitive" to the treewidth of their input in the sense that smaller treewidth strongly correlates with higher solving performance.

The limitations of the basic dynamic-programming-based approach hint at interesting research challenges. In particular, two approaches seem promising for successfully exploiting small treewidth for ASP solving in practice:

- The first research challenge is to improve the dynamic-programming-based methodology in order to avoid some of its overhead and redundant computations.

For solving ground ASP, these issues are especially severe compared to other problems because the corresponding computational problems are even harder than NP under standard complexity-theoretic assumptions. (In fact, deciding if a ground ASP program has an answer set is at the second level of the polynomial hierarchy.) This high complexity of ground ASP is mirrored in the dynamic programming algorithm (Jakl, Pichler and Woltran 2009), which uses brute force to first of all find all models of all parts of the decomposed program, and it subsequently uses brute force again *for each such partial model* to find all potential counterexamples that may cause the candidate to be discarded.

This pattern also frequently occurs in dynamic programming algorithms for other problems that search for solutions satisfying some form of subset minimality. Besides ground ASP, this is the case, for instance, for the problem of finding subset-minimal models of a propositional formula. Such algorithms typically store a great number of redundant objects because the subsets that may invalidate a solution candidate are themselves solution candidates. Moreover, the specifications of such algorithms themselves contain redundancies because the potential counterexamples are usually manipulated in almost the same way as the solution candidates.

- The second research challenge is to find out which ASP encoding techniques preserve bounded treewidth. Since problems are usually encoded in non-ground ASP, some language constructs may significantly blow up the treewidth of the grounding when compared to the treewidth of the input facts. Knowledge about treewidth-preserving encoding techniques would allow us to solve ASP by not doing dynamic programming at all but instead exploiting small treewidth *implicitly* by relying on the assumption that state-of-the-art solvers perform better when given ground programs of small treewidth (as indicated by the experiments in the paper by Bliem et al. (2017)).

In addition to leveraging treewidth for ASP solving, we are interested in several variants of a graph problem called Secure Set. It belongs to the class of so-called *alliance problems*, which are problems that ask for groups of vertices that help each other out in a certain way. Intuitively, a set S of vertices in a graph is secure if every subset of S has as least as many neighbors in S as neighbors not in S. The Secure Set problem asks whether a given graph contains a secure set at most of a certain size.

Alliance problems like Secure Set have various applications in practice. For instance, we could be interested in finding groups of nations, companies or individuals that depend on each other, but alliances also appear in less obvious contexts like groups of websites that form communities (Flake et al. 2002). Furthermore, alliances can be applied to computer networks in order to satisfy simultaneous requests (Haynes, S. T. Hedetniemi and Henning 2003).

The reason why we are concerned with Secure Set is that this problem has quite interesting properties, especially for ASP researchers: Attempts of encoding this problem in ASP have resulted in very involved specifications indicating that Secure Set may require the full expressive power of ASP (Abseher et al. 2015).[1] However, it is unfortunately unclear whether this is really necessary because its complexity has still remained unresolved although the problem has been introduced already in 2007 (Brigham, Dutton and S. T. Hedetniemi 2007). Moreover, there have been no investigations of treewidth as a parameter for this problem.

One of the variants of Secure Set that we consider in this work is the Defensive Alliance problem (Kristiansen, S. M. Hedetniemi and S. T. Hedetniemi 2002; Kristiansen, S. M. Hedetniemi and S. T. Hedetniemi 2004). This problem has received quite some attention in the literature (Fernau and Rodríguez-Velázquez 2014; Yero and Rodríguez-Velázquez 2013). It is known to be NP-complete, but the question of whether it is fixed-parameter tractable when parameterized by treewidth has remained open.

In this thesis, we investigate these research challenges. Our contributions are the following:

1. By restricting the syntax of non-ground ASP, we define two classes of programs called *guarded* and *connection-guarded* programs. Guarded programs guarantee that the treewidth of any fixed program stays small after grounding whenever the treewidth of the input facts is small. We formally prove this property and show that, despite their restrictions, guarded programs can still express problems that are complete for the second level of the polynomial hierarchy.

   Connection-guarded programs are even more expressive than guarded programs. We show that the treewidth of the grounding of any fixed connection-guarded program is small whenever the treewidth *and the maximum degree* of (a graph representation of) the input facts is small.

   These results bring us closer to the goal of implicitly taking advantage of the apparent sensitivity to treewidth exhibited by modern ASP solvers because they give us insight into what happens to the treewidth of the input during grounding. Thus, by writing a program in guarded ASP, we can be sure that the grounder does not destroy the property of bounded treewidth. In the case of connection-guarded ASP, the same holds for the combination of treewidth and maximum degree.

[1] In particular, the ASP encodings for Secure Set by Abseher et al. (2015) not only make use of the so-called saturation technique, which requires disjunction, but also of non-convex, recursive aggregates (Alviano, Faber and Gebser 2015; Faber 2006; Liu and Truszczyński 2006), which push ASP systems to their limits because such aggregates have not even been supported by some state-of-the-art systems until recently. The reason for this is that the semantics of such aggregates has not been agreed upon.

2. We present a complexity analysis of computational problems corresponding to guarded and connection-guarded programs, when the parameter is the treewidth of the input, the maximum degree of the input, or the combination of both. The results of this analysis show that, for any fixed guarded ASP program, answer set solving is FPT when parameterized by the treewidth of the input; moreover, for any fixed connection-guarded ASP program, answer set solving is FPT when parameterized by the combination of treewidth and maximum degree. This is not obvious because our ASP classes support weak constraints and aggregates, which are not accounted for in the FPT algorithms (Jakl, Pichler and Woltran 2009; Fichte et al. 2017) for ground ASP. Furthermore, we prove hardness results showing that for connection-guarded ASP *both* the treewidth and the maximum degree must be bounded for obtaining fixed-parameter tractability.

3. As a side-product of the investigations on ASP classes, we obtain *metatheorems* for obtaining FPT results. That is, our results on guarded ASP allow us to prove that a problem is FPT when parameterized by treewidth by simply expressing the problem in guarded ASP. We compare this metatheorem to the common approach of proving fixed-parameter tractability by expressing a problem in monadic second-order logic and invoking the well-known theorem by Courcelle (Courcelle 1990; Courcelle 1992). Similarly, we can prove that a problem is FPT when parameterized by the combination of treewidth and maximum degree by expressing the problem in connection-guarded ASP. This result is appealing because we are not aware of any metatheorems that allow us to obtain FPT results for the combination of treewidth and degree as the parameter.

4. We illustrate the use of our ASP classes as FPT classification tools by presenting simple encodings for alliance problems in graphs. By encoding the NP-complete Defensive Alliance problem in connection-guarded ASP, we easily obtain the already known result that the problem is FPT when parameterized by the combination of treewidth and maximum degree. More importantly, we obtain the new result that the co-NP-complete problem of deciding whether a given set is secure in a graph is FPT for the parameter treewidth by encoding the problem in guarded ASP.

5. We settle the complexity of the Secure Set problem by proving that the problem, along with several variants, is Σ_2^P-complete (that is, at the second level of the polynomial hierarchy).

6. We prove that both Defensive Alliance and Secure Set, as well as several problem variants, are not FPT when parameterized by treewidth alone (under commonly held complexity-theoretic assumptions). These questions have been open since the problems have been introduced in 2002 and 2007, respectively.

They have explicitly been stated as open problems by Kiyomi and Otachi (2017) (for Defensive Alliance) and by Ho and Dutton (2009) (for Secure Set).

7. We show that the Secure Set problem can still be solved in polynomial time for instances of bounded treewidth although the degree of the polynomial depends on the treewidth.

8. We present an improvement of the tree-decomposition-based dynamic programming methodology for problems that involve subset minimization (like, for instance, finding subset-minimal models of a propositional formula). Specifically, for any problem P whose solutions are exactly the subset-minimal solutions of some base problem B, we formalize how a dynamic programming algorithm for B can automatically be transformed into a dynamic programming algorithm for P. We prove that, under certain conditions, the resulting algorithm is correct if the base algorithm is correct, and it runs in FPT time if the base algorithm does. The resulting algorithm has two advantages compared to solving P directly in a naive way: first, it is usually easier to specify because we only need to design an algorithm for the base problem and need not care about subset minimization; second, it is potentially more efficient because it stores fewer redundant items. We also show how our approach can be extended to more general settings than simple subset minimization.

   Indeed, this methodology has been empirically shown to lead to significant performance benefits for several problems (Bliem et al. 2016a). An improved version of the classical dynamic programming algorithm for ground ASP has been implemented using these ideas (Fichte et al. 2017) and proved to be significantly more efficient than the algorithm by Jakl, Pichler and Woltran (2009). Our result formalizes the common scheme that underlies these algorithms. We thus provide a formal framework that makes it possible to transfer the mentioned optimizations easily to other problems. Thereby we make the impressive performance benefits that have been reported by Bliem et al. (2016a) and Fichte et al. (2017) accessible to algorithm designers working on related problems. This is primarily useful for problems on the second level of the polynomial hierarchy as subset minimization is a recurring theme in many such problems.

Research that is related to our work will be discussed in dedicated sections of our main chapters.

This work is structured as follows. We start off with the necessary background for our investigations in Chapter 2. In Chapter 3 we then introduce and analyze classes of non-ground ASP programs with the goal that grounding such programs together with an input preserves bounded treewidth of the input. This is followed by our study of alliance problems in graphs in Chapter 4. Next, in Chapter 5, we present our improvements to the algorithmic technique of dynamic programming on tree decompositions, which helps us solve problems involving subset minimization more efficiently. Finally, we summarize and discuss our results in Chapter 6.

Publications

Parts of this thesis have been published in several articles. Moreover, there are also some publications that developed as side-products during the work on this dissertation but were not included in it as they are not directly related to the core topics. We briefly list all related articles. If a paper has been superseded by a newer publication, we only mention the improved work.

Treewidth-preserving ASP classes. A paper containing a preliminary version of a part of this chapter has been accepted at IJCAI 2017 (Bliem et al. 2017). This thesis substantially extends the results from that paper as discussed in Chapter 3.

Alliance problems in graphs. Parts of this chapter have been presented at WG 2015 (Bliem and Woltran 2016a). An extended version of this paper is currently under review for a journal. Still, this thesis contains further significant extensions that have not yet been published as discussed in Chapter 4.

Advanced dynamic programming methodologies. A large part of the work in Chapter 5 has been published in Fundamenta Informaticae (Bliem et al. 2016a).

Other publications. The following publications are not part of this dissertation but developed during the work on the thesis.

- Bliem et al. (2016b): This work describes an algorithmic framework for executing tree-decomposition-based dynamic programming algorithms using lazy evaluation. In contrast to classical dynamic programming, this allows for *anytime* optimization algorithms, that is, algorithms that can report solutions before the optimum is found. This paper was presented at IJCAI 2016.
- Bliem, Ordyniak and Woltran (2016): This paper was presented at ECAI 2016 and studies the parameter clique-width as well as directed width measures for ground ASP solving.
- Bliem and Woltran (2016b): In this work, we investigated under which conditions ASP programs are *rule-equivalent*, that is, equivalent under the addition of rules that are not facts. The paper was presented at FoIKS 2016.

- Abseher et al. (2015): This article contains ASP encodings for the Secure Set problem and an experimental evaluation. The paper was accepted for publication in the Journal of Logic and Computation (JLC).
- Bliem, Hecher and Woltran (2016): This work studies a Σ_2^P-complete problem from abstract argumentation. It presents an FPT algorithm for the parameter treewidth and evaluates the performance using the techniques formalized in Chapter 5. The paper was presented at COMMA 2016.
- Abseher et al. (2014): This paper was presented at JELIA 2014 and describes the D-FLAT system for rapid prototyping of tree-decomposition-based dynamic programming algorithms. Users of the system can specify such algorithms in ASP. The paper was a continuation of work in the Master's thesis of the author of this dissertation.
- Bliem, Pichler and Woltran (2017): This is a JLC paper that shows how monadic second-order (MSO) model checking can be implemented in D-FLAT, which proves that this framework can be used for solving all MSO-definable problems in FPT time when parameterized by treewidth.
- Bliem, Bredereck and Niedermeier (2016): This work was presented at IJCAI 2016 and contains a parameterized complexity analysis of the Σ_2^P-complete problem of finding a Pareto-efficient and envy-free allocation of indivisible resources to agents.

CHAPTER 2

Background

In this chapter, we present preliminaries for our results. In Section 2.1, we define concepts from graph theory. Section 2.2 contains preliminaries from complexity theory. This is followed by background on logic in Section 2.3. Finally, we give an overview of Answer Set Programming in Section 2.4.

2.1 Graphs

Unless stated otherwise, we assume graphs to be undirected, simple and ordered, where ordered means that there is an arbitrary but fixed total order over the vertices. We denote the set of vertices and edges of a graph G by $\mathrm{V}(G)$ and $\mathrm{E}(G)$, respectively, and we denote an undirected edge between vertices u and v as (u, v) or equivalently (v, u). We thus abuse the pair notation even though undirected edges are, strictly speaking, sets. It will be clear from the context whether an edge (u, v) is directed or undirected.

Given a graph G, the *open neighborhood* of a vertex $v \in \mathrm{V}(G)$, denoted by $N_G(v)$, is the set of all vertices adjacent to v, and $N_G[v] = N_G(v) \cup \{v\}$ is called the *closed neighborhood* of v. Let $S \subseteq \mathrm{V}(G)$. We abuse notation by writing $N_G(S)$ and $N_G[S]$ to denote $\bigcup_{v \in S} N_G(v)$ and $\bigcup_{v \in S} N_G[v]$, respectively. If it is clear from the context which graph is meant, we write $N(\cdot)$ and $N[\cdot]$ instead of $N_G(\cdot)$ and $N_G[\cdot]$, respectively.

Given a graph G and a set of vertices $S \subseteq \mathrm{V}(G)$, we say that S *induces* a subgraph H of G if $\mathrm{V}(H) = S$ and $\mathrm{E}(H) = \mathrm{E}(G) \cap S^2$. Similarly, we call H the *subgraph* of G *induced by* S.

2.1.1 Treewidth

For problems whose input can be represented as a graph, *treewidth* is an important structural parameter. Intuitively, the smaller the treewidth of a graph, the closer the graph resembles a tree. The definition is based on the concept of *tree decompositions*, which have originally been introduced by Robertson and Seymour (1984). The intuition behind tree decompositions is to obtain a tree from a (potentially cyclic) graph by subsuming multiple vertices under one node and thereby isolating the parts responsible for cyclicity.

Definition 2.1. A *tree decomposition* of a graph G is a pair $\mathcal{T} = (T, \chi)$ where T is a (rooted) tree and $\chi : \mathrm{V}(T) \to 2^{\mathrm{V}(G)}$ assigns to each node of T a set of vertices of G (called the node's *bag*), such that the following conditions are met:

1. For every vertex $v \in \mathrm{V}(G)$, there is a node $t \in \mathrm{V}(T)$ such that $v \in \chi(t)$.
2. For every edge $(u, v) \in \mathrm{E}(G)$, there is a node $t \in \mathrm{V}(T)$ such that $\{u, v\} \subseteq \chi(t)$.
3. For every $v \in \mathrm{V}(G)$, the subtree of T induced by $\{t \in \mathrm{V}(T) \mid v \in \chi(t)\}$ is connected.[1]

We call $\max_{t \in \mathrm{V}(T)} |\chi(t)| - 1$ the *width* of $\mathcal{T}$. The *treewidth* of a graph is the minimum width over all its tree decompositions.

It is not hard to see that trees have treewidth 1. At the other extreme, a complete graph with n vertices (often denoted as K_n) and a complete bipartite graph with n vertices in each part (often denoted as $K_{n,n}$) have treewidth n.

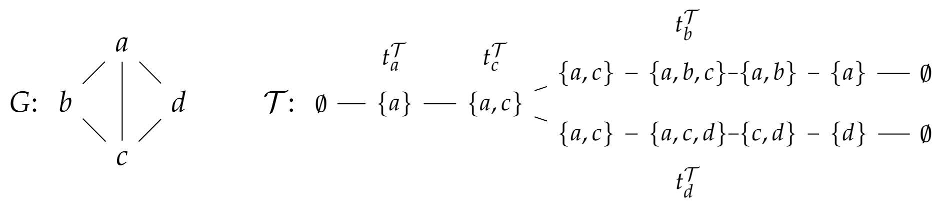
In general, constructing an optimal tree decomposition (i.e., a tree decomposition with minimum width) is intractable (Arnborg, Corneil and Proskurowski 1987). However, the problem is solvable in linear time on graphs of bounded treewidth (specifically in time $w^{\mathcal{O}(w^3)} \cdot n$, where w is the treewidth Bodlaender 1996) and there are also heuristics that offer good performance in practice (Dermaku et al. 2008; Bodlaender and Koster 2010; Bodlaender 2005).

In this work we will consider so-called *nice* tree decompositions.

Definition 2.2. A tree decomposition $\mathcal{T} = (T, \chi)$ is *nice* if each node $t \in \mathrm{V}(T)$ is of one of the following types:

1. *Leaf node:* The node t has no child nodes.

[1] Equivalently, if a vertex is contained in the bags of two nodes a, b at the same time, then it must be contained in all bags of nodes between a and b.



G: a, b, c, d
$\mathcal{T}$: $\emptyset$ — $\{a\}$ — $\{a,c\}$
$\{a,c\}$ – $\{a,b,c\}$–$\{a,b\}$ – $\{a\}$ — $\emptyset$
$\{a,c\}$ – $\{a,c,d\}$–$\{c,d\}$ – $\{d\}$ — $\emptyset$
$t_a^{\mathcal{T}}$
$t_c^{\mathcal{T}}$
$t_b^{\mathcal{T}}$
$t_d^{\mathcal{T}}$

Figure 2.1: A graph G and a nice tree decomposition $\mathcal{T}$ of G rooted at the leftmost node

$$\varphi_{Ex}\colon \quad (u \vee v) \wedge (\neg v \vee w \vee x) \wedge (\neg w) \wedge (\neg x \vee z) \wedge (\neg x \vee y \vee \neg z)$$

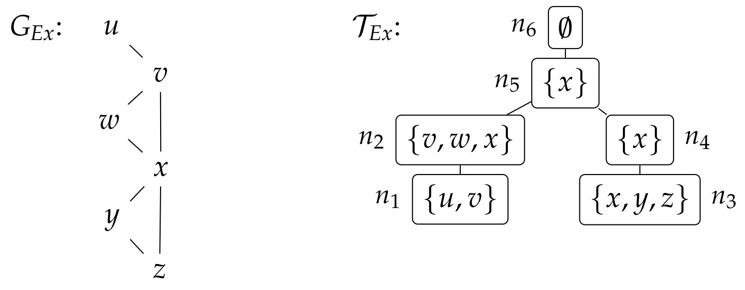


Figure 2.2: A propositional formula φ_{Ex}, the primal graph G_{Ex} of φ_{Ex} and a tree decomposition $\mathcal{T}_{Ex}$ for G_{Ex}.

2. *Introduce node:* The node t has exactly one child node t' such that $\chi(t') \subset \chi(t)$ and $|\chi(t) \setminus \chi(t')| = 1$.

3. *Forget node:* The node t has exactly one child node t' such that $\chi(t) \subset \chi(t')$ and $|\chi(t') \setminus \chi(t)| = 1$.

4. *Join node:* The node t has exactly two child nodes t_1 and t_2 with $\chi(t) = \chi(t_1) = \chi(t_2)$.

Additionally, the bags of the root and the leaves of T are empty.

A tree decomposition of width w for a graph with n vertices can be transformed into a nice one of width w with $\mathcal{O}(wn)$ nodes in linear time if w is bounded by a constant (Kloks 1994).

For any tree decomposition $\mathcal{T}$ and an element v of some bag in $\mathcal{T}$, we use the notation $t_v^{\mathcal{T}}$ to denote the unique "topmost node" whose bag contains v (i.e., $t_v^{\mathcal{T}}$ does not have a parent whose bag contains v). Figure 2.1 depicts a graph and a nice tree decomposition, where we also illustrate the $t_v^{\mathcal{T}}$ notation.

Example 2.3. Let us introduce a running example based on the Boolean satisfiability problem (Sat). Given a propositional formula φ in conjunctive normal form (CNF, i.e., a conjunction of disjunctive clauses), we first have to find an appropriate graph representation. Here, we construct the *primal graph* G of φ, that is, vertices in G represent atoms of φ, and atoms occurring together in a clause form a clique in G. An example formula φ_{Ex}, its graph representation G_{Ex} and a possible tree decomposition $\mathcal{T}_{Ex}$ are given in Figure 2.2. The width of $\mathcal{T}_{Ex}$ is 2. Note that $\mathcal{T}$ contains unnecessarily many nodes: We could obtain another valid tree decomposition for φ_{Ex} by arranging n_3, n_2 and n_1 in a path. However, we chose $\mathcal{T}$ to serve for our example because it is more suitable for illustrating dynamic programming algorithms like in Example 2.4. △

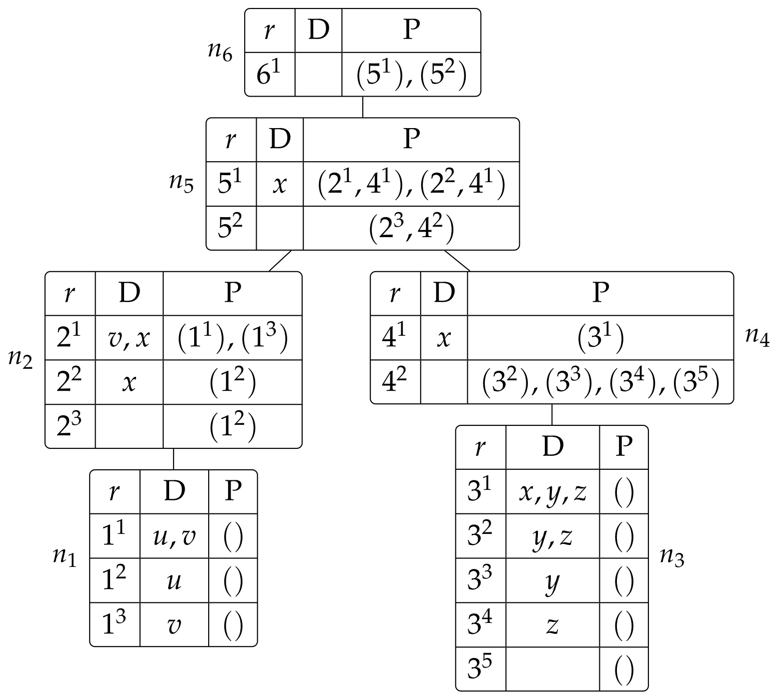
2.1.2 Dynamic Programming on Tree Decompositions

Tree decompositions have the important property that, for any tree decomposition $\mathcal{T}$ of a graph G, removing an edge (a, b) from $\mathcal{T}$ corresponds to a separation of G into two parts G_1, G_2. Each part is the subgraph of G induced by the vertices in the bags of the respective part of $\mathcal{T}$ after removing (a, b). In fact, $\chi(a) \cap \chi(b)$ is a *separator* of G, which separates G into G_1 and G_2. This means that by removing $\chi(a) \cap \chi(b)$ from G, there is no path from a vertex that is only in G_1 to a vertex that is only in G_2 or vice versa. Indeed, it is easy to verify by Definition 2.1 that there can be no edges between elements of $V(G_1) \setminus V(G_2)$ and elements of $V(G_2) \setminus V(G_1)$.

This separation property is crucial for a certain way of solving problems that follows the principle of *dynamic programming*. We briefly sketch the basic intuition. Assume we are given a graph G along with a separation into two parts G_1, G_2, and we want to decide if G has a certain property. Furthermore, suppose that we are lucky and this problem can be solved by an algorithm with the convenient property that, intuitively, the "decisions" on the status of a vertex can only affect the status of neighboring vertices (possibly in a transitive way). First we try out and record all possible decisions on the status of the vertices in G_1. In a way, we now "know everything" about the vertices in $V(G_1) \setminus V(G_2)$ because we have considered the effects of our decisions on these vertices and all their neighbors. Hence we can now forget all information that we have stored about $V(G_1) \setminus V(G_2)$ and just keep the information concerning the separator $V(G_1) \cap V(G_2)$. Using this information, we can now consider $V(G_2) \setminus V(G_1)$. Intuitively, the influence of the now forgotten part on $V(G_2)$ is all captured by the information we stored about $V(G_1) \cap V(G_2)$.

This dynamic-programming-based approach allows us to solve many problems, but the running times are clearly infeasible in general. However, in many cases this idea leads to linear-time algorithms under the condition that the graph can be separated into parts whose sizes are bounded by a constant. In particular, dynamic programming on tree decompositions is a common technique for obtaining algorithms whose running times are often linear if the treewidth of the instances is bounded by a constant.

In tree-decomposition-based dynamic programming algorithms, we usually traverse



n_6

r	D	P
6^1		$(5^1), (5^2)$

n_5

r	D	P
5^1	x	$(2^1, 4^1), (2^2, 4^1)$
5^2		$(2^3, 4^2)$

n_2

r	D	P
2^1	v, x	$(1^1), (1^3)$
2^2	x	(1^2)
2^3		(1^2)

n_4

r	D	P
4^1	x	(3^1)
4^2		$(3^2), (3^3), (3^4), (3^5)$

n_1

r	D	P
1^1	u, v	$()$
1^2	u	$()$
1^3	v	$()$

n_3

r	D	P
3^1	x, y, z	$()$
3^2	y, z	$()$
3^3	y	$()$
3^4	z	$()$
3^5		$()$

Figure 2.3: Dynamic programming computation for Sat

the tree decomposition in post-order. At each node, we compute partial solutions for the subgraph induced by the vertices encountered so far and we store them in a *table* associated with the node. For many problems, the time to compute the table of a tree decomposition node only depends on the treewidth. Albeit this running time may be exponential in the treewidth, it is still constant if the treewidth is bounded by a constant. As we may assume that the size of a tree decomposition is linear in the size of the input, we can thus compute all tables in linear time and finally decide the problem by just inspecting the table at the root node.

For some problems, we may be slightly less lucky and obtain an algorithm where the time for computing a table depends on the treewidth *and* the input size, but the dependence on the input size is only polynomial. In this case, even though the running time is not linear anymore, such an algorithm may still be attractive for intractable problems because it solves the problem in polynomial time on instances of bounded treewidth.

To illustrate dynamic programming with an example, we now describe an algorithm for the Sat problem (Samer and Szeider 2010).

Example 2.4. Figure 2.3 illustrates a dynamic programming computation for the Sat instance from Figure 2.2. We compute the tables as follows. For a tree decomposition node n, each table row r consists of a set $\mathrm{D}(r)$ and a set $\mathrm{P}(r)$. The set $\mathrm{D}(r)$, where D stands for "data", stores partial truth assignments over atoms in $\chi(n)$. We represent a truth assignment as the set of atoms it sets to "true". Hence $\mathrm{D}(r)$ contains the atoms that get assigned "true", whereas atoms in $\chi(n) \setminus \mathrm{D}(r)$ get assigned "false". We only store a row r if all clauses covered by $\chi(n)$ are satisfied by the respective partial truth assignment. The set $\mathrm{P}(r)$ contains so-called "extension pointer tuples" (EPTs) that denote rows in the child tables that we used for constructing r.

First consider node n_1. Here, $\chi(n_1) = \{u, v\}$ covers the clause $(u \lor v)$, which yields three partial assignments for φ_{Ex}. In n_2, the child rows are extended, the partial assignments are updated (by removing atoms not contained in $\chi(n_2)$ and guessing truth assignments for atoms in $\chi(n_2) \setminus \chi(n_1)$). Here, the clauses $(\neg v \lor w \lor x)$ and $(\neg w)$ have to be satisfied. Observe that row 2^1 is constructed from two different child rows. In n_3 we proceed as described before. In n_4, data concerning the removed vertices y and z are projected away. In n_5, we additionally only join partial assignments that agree on the truth assignment for common atoms. We continue this procedure recursively until we reach the root of the tree decomposition.

To decide whether the formula is satisfiable, it suffices to check if the table in the root node is non-empty. The overall procedure runs in linear time on instances of bounded treewidth because the number of nodes in the tree decomposition is bounded by the input size (i.e., the number of atoms), and each node t is associated with a table of size at most $\mathcal{O}(2^{|\chi(t)|})$ (i.e., the number of possible truth assignments).

In our example, φ_{Ex} is satisfiable due to existence of row 6^1. Solutions (i.e., models of φ_{Ex}) can even be enumerated with linear delay by starting at the root and following the EPTs while combining the partial assignments associated with the rows. For instance, we can obtain the model $\{u, v, x, y, z\}$ by starting at row 6^1 and following the EPTs (5^1), $(2^1, 4^1)$, (1^1) and (3^1), thereby combining $\mathrm{D}(6^1) \cup \mathrm{D}(5^1) \cup \mathrm{D}(2^1) \cup \mathrm{D}(1^1) \cup \mathrm{D}(4^1) \cup \mathrm{D}(3^1)$. $\triangle$

2.2 Computational Complexity

We assume basic familiarity with complexity theory and only recapitulate concepts that are important for this work. For a detailed introduction, we refer to the book by Papadimitriou (1994).

The *polynomial hierarchy* consists of the classes Σ_k^{P}, Π_k^{P} and Δ_k^{P} of decision problems, for every nonnegative integer k. These classes are inductively defined as follows:

$$\begin{aligned}
\Sigma_0^{\mathsf{P}} &= \Pi_0^{\mathsf{P}} = \Delta_0^{\mathsf{P}} = \mathsf{P}\\
\Sigma_{k+1}^{\mathsf{P}} &= \mathsf{NP}^{\Sigma_k^{\mathsf{P}}}\\
\Pi_{k+1}^{\mathsf{P}} &= \text{co-}\mathsf{NP}^{\Sigma_k^{\mathsf{P}}}\\
\Delta_{k+1}^{\mathsf{P}} &= \mathsf{P}^{\Sigma_k^{\mathsf{P}}}
\end{aligned}$$

$$\begin{tikzcd}
& \cdots & & \cdots & \\
& & \Delta_3^{\mathsf{P}} \arrow[ul] \arrow[ur] & & \\
& \Sigma_2^{\mathsf{P}} \arrow[ur] & & \Pi_2^{\mathsf{P}} \arrow[ul] & \\
& & \Delta_2^{\mathsf{P}} \arrow[ul] \arrow[ur] & & \\
\text{(equal to } \mathsf{NP}\text{)} & \Sigma_1^{\mathsf{P}} \arrow[ur] & & \Pi_1^{\mathsf{P}} \arrow[ul] & \text{(equal to co-}\mathsf{NP}\text{)} \\
& & \mathsf{P} \arrow[ul] \arrow[ur] & & \\
& & \text{(equal to } \Sigma_0^{\mathsf{P}}, \Pi_0^{\mathsf{P}}, \Delta_0^{\mathsf{P}} \text{ and } \Delta_1^{\mathsf{P}}\text{)} & &
\end{tikzcd}$$

Figure 2.4: Inclusion of classes in the polynomial hierarchy

For any complexity class C, the class P^C contains all decision problems that can be solved in polynomial time by a deterministic Turing machine with access to an oracle for C. Such an oracle can be seen as a subroutine that runs in constant time and can solve all problems in C. The class NP^C is defined in an analogous way for nondeterministic Turing machines. It is known that the following relationship holds.

$$\Sigma_0^{\mathsf{P}} \subseteq \Sigma_1^{\mathsf{P}} \subseteq \cdots \subseteq \mathsf{PSPACE}$$

The class PSPACE consists of all problems solvable in polynomial space. All inclusions are widely believed to be proper. We also get an analogous chain of inclusions for the complementary classes. Figure 2.4 depicts the relationship between the classes in the polynomial hierarchy, where an edge from class A to B denotes that A is a subclass of B. When we speak of the k-th *level* of the polynomial hierarchy, we mean the classes Σ_k^{P}, Π_k^{P} and $\Delta_{k+1}^{\mathsf{P}}$.

For any level k of the polynomial hierarchy, the following problem over quantified Boolean formulas is the canonical Σ_k^{P}-complete problem:

Qsat$_k$

Input:	A formula $\exists X_1 \forall X_2 \cdots Q X_k\ \varphi$, where each X_i is a set of propositional atoms, φ is a propositional formula over these atoms, Q is $\forall$ if k is even and Q is $\exists$ otherwise.
Question:	Is there a truth assignment to the variables in X_1 such that for all truth assignments to the variables in X_2 there is a truth assignment to the variables in X_3 such that for all … such that φ evaluates to true?

In parameterized complexity theory (Downey and Fellows 1999; Flum and Grohe 2006; Niedermeier 2006; Cygan et al. 2015), we study the complexity of problems not only in terms of the input size but also of some parameter of the input that is represented as an integer.

Definition 2.5. First we fix a finite alphabet Σ and denote the set of all finite strings over Σ by Σ^*. A *parameterized problem* is a subset of $\Sigma^* \times \mathbb{N}$. We call each element $(x, k) \in \Sigma^* \times \mathbb{N}$ an *instance* of the problem and we call k the *parameter* of the instance.

We say that an algorithm *solves* a parameterized problem P if, for any instance (x, k) of P, it correctly decides if $(x, k) \in P$.

The central concept in parameterized complexity theory is called *fixed-parameter tractability* (FPT).

Definition 2.6. A parameterized problem is in the class FPT if it can be solved in time $f(k) \cdot n^c$, where n is the input size, k is the parameter, f is a computable function that only depends on k, and c is a constant. We call a running time of this form *FPT time* and we say that such problems and algorithms are *fixed-parameter tractable* (FPT). An FPT algorithm is *fixed-parameter linear* if $c = 1$.

The nice thing about FPT algorithms is that they run in polynomial time if the parameter is bounded by a constant. This assumption may reasonable in practice because, for some parameters, real-world instances usually have small parameter values and thus do not represent the worst case. Clearly every problem in P is in FPT for any choice of parameter.

A similar class, which also offers polynomial running times for fixed parameter values, is called XP, which stands for "slice-wise polynomial".

Definition 2.7. A parameterized problem is in XP if it can be solved in time $f(k) \cdot n^{g(k)}$, where n is the input size, k is the parameter, and f and g are computable functions that only depend on k.

Note that here the degree of the polynomial may depend on k, so such algorithms are generally slower than FPT algorithms. Still, membership in XP is good news for problems that are NP-hard in the classical sense.

If we have a problem that is complete for a classical complexity class C, then bad news would be if we could prove that this problem is also C-hard for a fixed value of the parameter. In such a case, we say that the problem is para-C-hard, which means that the parameter is useless in terms of complexity. Consider the following well-known NP-complete problem for example.

Graph Coloring

Input: A graph G and an integer k

Question: Is there a proper k-coloring[a] of G?

[a] A proper k-coloring of a graph G is a mapping $\mathrm{V}(G) \to \{1, \dots, k\}$ such that any two adjacent vertices have different colors.

Since this problem is NP-hard for $k = 3$, it is para-NP-hard when we parameterize it by the number of colors. Thus assuming that the number of colors is bounded by a constant (of at least three) clearly does not help us in lowering the complexity of this problem.

The class FPT is included in XP, and there are several classes in between. In this work, we only require the class W[1]. It holds that $\mathsf{FPT} \subseteq \mathsf{W}[1] \subseteq \mathsf{XP}$, and it is commonly believed that the inclusions are proper. This means that W[1]-hard problems presumably do not admit FPT algorithms.

We can show that a problem is W[1]-hard by reducing from a known W[1]-hard problem. However, for parameterized hardness results, we need slightly different reductions than the usual polynomial-time reductions that we encounter in classical complexity theory. Intuitively, we require that a reduction is not only correct but also that it runs in FPT time (as opposed to classical polynomial-time reductions) and that it does not lead to an "excessive" increase in the parameter value.

Definition 2.8. An algorithm that transforms each instance (x, k) of a parameterized problem P into an instance (x', k') of a parameterized problem Q is an *FPT reduction* if it satisfies the following conditions:

1. It holds that $(x, k) \in P$ if and only if $(x', k') \in Q$.
2. The algorithm runs in FPT time.
3. There is a computable function f such that $k' \leqslant f(k)$.

2.3 Logic

In this thesis, we will use *monadic second-order logic* (MSO) to express properties of relational structures. In particular, we will use the notion of *MSO transductions* to formally characterize mappings that transform a structure to another structure. We define relevant concepts about structures, MSO and MSO transductions in Sections 2.3.1, 2.3.2 and 2.3.3, respectively.

2.3.1 Relational Structures

Relational structures are a basic concept in mathematical logic, (finite) model theory and databases (Libkin 2004; Abiteboul, Hull and Vianu 1995). First-order logic and its extensions, such as MSO, allow us to express properties of relational structures, which are used as interpretations for formulas. In order to state a formula in such a language, it is necessary to first fix a signature, which determines the vocabulary that we may use in the formula alongside the logical symbols. That is, we need to fix a certain set of relation symbols over which we can then define formulas and relational structures, which may or may not satisfy these formulas.

Definition 2.9. A *relational signature* (or just "signature") is a finite set σ of relation symbols, where each symbol $R \in \sigma$ has a corresponding *arity* $\rho_\sigma(R)$, which is a nonnegative integer. We write $\rho(\sigma)$ to denote the maximum arity of any relation symbol in σ.[2]

Signatures allow us to define relational structures over them.

Definition 2.10. A *relational structure* (or just "structure") $\mathcal{A}$ over a signature σ consists of a finite *domain* $\mathrm{dom}(\mathcal{A})$ and, for every $R \in \sigma$, a relation $R^{\mathcal{A}} \subseteq \mathrm{dom}(\mathcal{A})^{\rho_\sigma(R)}$. Whenever a relation $R^{\mathcal{A}}$ contains a tuple $\boldsymbol{a}$, we say that $R(\boldsymbol{a})$ is a *fact* in $\mathcal{A}$, and we say that the elements of $\boldsymbol{a}$ are the *arguments* of that fact.

Example 2.11. For representing directed graphs, it is customary to use the signature $\sigma = \{\mathrm{E}\}$ with $\rho_\sigma(\mathrm{E}) = 2$. We can now represent a directed graph G as a relational structure $\mathcal{G}$ over σ by choosing $\mathrm{dom}(\mathcal{G}) = \mathrm{V}(G)$ and $\mathrm{E}^{\mathcal{G}} = \mathrm{E}(G)$.

Alternatively, we can also represent G in a slightly more complex way as an "incidence structure": For this, we use the signature $\tau = \{\mathrm{E}, \mathrm{in}_1, \mathrm{in}_2\}$, where $\rho_\tau(\mathrm{E}) = 1$ and $\rho_\tau(\mathrm{in}_1) = \rho_\tau(\mathrm{in}_2) = 2$. The intended meaning of the abbreviation "in" is "incident". Now we can define the structure $\mathcal{S}$ over τ via $\mathrm{dom}(\mathcal{S}) = \mathrm{V}(G) \cup \mathrm{E}(G)$, and for each edge e from vertex a to b it holds that $e \in \mathrm{E}^{\mathcal{S}}$, $\langle e, a\rangle \in \mathrm{in}_1^{\mathcal{S}}$ and $\langle e, b\rangle \in \mathrm{in}_2^{\mathcal{S}}$. $\triangle$

Generalizing this example, an incidence structure is a relational structure that in turn represents a relational structure in a certain way. Namely, the domain of an incidence structure is a superset of the domain of the base structure that additionally contains all facts of the base structure (each as an "atomic element"). Each relation symbol of the base structure, regardless of its arity, is a *unary* relation symbol in the incidence structure, and can be used to express that a domain element is a fact from the respective relation of the base structure. To associate facts with their arguments, incidence structures possess binary relations in_i, which contain a pair $\langle f, a\rangle$ whenever f is a fact in the base structure that has a as its i-th argument. We now define this formally.

[2]Note that signatures are usually defined to also contain function symbols including constants, but we only require relation symbols for our purposes.

Definition 2.12. Let $\mathcal{A}$ be a relational structure over a signature σ. We define the signature $\mathrm{Inc}(\sigma) = \sigma \cup \{\mathrm{in}_1, \ldots, \mathrm{in}_{\rho(\sigma)}\}$, where $\rho_{\mathrm{Inc}(\sigma)}(R) = 1$ for every $R \in \sigma$, and $\rho_{\mathrm{Inc}(\sigma)}(\mathrm{in}_i) = 2$ for $1 \leqslant i \leqslant \rho(\sigma)$. We call $\mathrm{Inc}(\sigma)$ the *incidence signature* of σ. Now the *incidence structure* of $\mathcal{A}$ is the structure $\mathrm{Inc}(\mathcal{A})$ over $\mathrm{Inc}(\sigma)$ whose domain is $\mathrm{dom}(\mathcal{A}) \cup \{R(\boldsymbol{a}) \mid R \in \sigma,\ \boldsymbol{a} \in R^{\mathcal{A}}\}$, where each $R \in \sigma$ is interpreted as $R^{\mathrm{Inc}(\mathcal{A})} = \{R(\boldsymbol{a}) \mid \boldsymbol{a} \in R^{\mathcal{A}}\}$, and each $\mathrm{in}_i^{\mathrm{Inc}(\mathcal{A})}$ is the set of all pairs $\langle R(\boldsymbol{a}), b\rangle$ such that $R(\boldsymbol{a})$ is a fact in $\mathcal{A}$ and b is the i-th element of $\boldsymbol{a}$. We call $\mathcal{A}$ the *base structure* of $\mathrm{Inc}(\mathcal{A})$, and σ the *base signature* of $\mathrm{Inc}(\sigma)$.[3]

In order to apply some graph-theoretic concepts (like, e.g., treewidth) to structures, we use the following common way of representing a structure as a graph.

Definition 2.13. The *Gaifman graph* of a structure $\mathcal{A}$ is the undirected graph whose vertices are the domain elements of $\mathcal{A}$ and that has an edge between two different elements if they occur together in a fact of $\mathcal{A}$.

This notion can be used to define several parameters of structures, which is important because computational tasks may become easier if these parameters are bounded.

Definition 2.14. The *degree* of a domain element of a structure $\mathcal{A}$ is its degree in the Gaifman graph of $\mathcal{A}$, and the *degree* of $\mathcal{A}$ is the maximum degree of its Gaifman graph.

Definition 2.15. The *distance* between two domain elements of a structure is the their distance in the Gaifman graph of the structure.

Definition 2.16. The *treewidth* of a structure is the treewidth of its Gaifman graph.

2.3.2 MSO

Monadic second-order (MSO) logic is an extension of first-order logic by quantification over unary relations, that is, sets. We briefly state its syntax and semantics. For this, we assume familiarity with first-order logic. Detailed accounts of MSO can be found in the books by Libkin (2004), Flum and Grohe (2006) and Courcelle and Engelfriet (2012).

All logical symbols from first-order logic are also available in MSO, that is, logical connectives ($\neg$, $\wedge$, $\vee$ and $\rightarrow$), first-order quantifiers ($\forall$, $\exists$), parentheses, commas, the equality symbol and infinitely many individual variables, which are usually written in lower case. In addition, MSO has logical symbols for *second-order quantifiers* (which we

[3]Note that we concealed a detail in this definition: Name clashes may occur when the base signature contains a relation symbol of the form in_i. For this reason, relation symbols in a signature σ can be thought of technically containing σ as an additional subscript. However, we will refrain from overburdening the notation in this way. Instead we appeal to the intuition of the reader, since it will be clear from the context which signature we are referring to when we use a relation symbol.

also write as $\forall$ and $\exists$) and *set variables*, which are usually written in upper case. The available relation symbols are given by a signature as defined in Definition 2.9.[4]

Definition 2.17. Let σ be a relational signature. We inductively define the set of *MSO formulas over σ* (or just "formulas" if no confusions arise) as follows:

- The expression $x = y$, where x and y are individual or set variables, is a formula.
- The expression $x \in X$, where x is an individual variable and X is a set variable, is a formula.
- An expression of the form $R(x_1, \dots, x_{\rho_\sigma(R)})$, where $R \in \sigma$ and $x_1, \dots, x_{\rho_\sigma(R)}$ are individual variables, is a formula.
- If φ and ψ are formulas, then also $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$, $\exists x\, \varphi$, $\forall x\, \varphi$, $\exists X\, \varphi$ and $\forall X\, \varphi$ are formulas, where x and X are individual and set variables, respectively.

As in first-order logic, we say that a variable is *free* in a formula if it has an occurrence that is not bound by a quantifier.

Next we state the semantics of MSO by defining a satisfaction relation $\models$.

Definition 2.18. Let φ be an MSO formula over a signature σ, $\mathcal{A}$ be a relational structure over σ and α be a variable assignment that maps each free individual variable in φ to an element of $\mathrm{dom}(\mathcal{A})$ and each free set variable in φ to a subset of $\mathrm{dom}(\mathcal{A})$. To define when $(\mathcal{A}, \alpha) \models \varphi$ holds, we distinguish the different forms that φ can have:

- We write $(\mathcal{A}, \alpha) \models x = y$, where x and y are individual or set variables, if $\alpha(x) = \alpha(y)$.
- We write $(\mathcal{A}, \alpha) \models x \in X$ if $\alpha(x) \in \alpha(X)$.
- We write $(\mathcal{A}, \alpha) \models R(x_1, \dots, x_n)$ if $\langle \alpha(x_1), \dots, \alpha(x_n) \rangle \in R^{\mathcal{A}}$.
- We write $(\mathcal{A}, \alpha) \models \neg\psi$ if $(\mathcal{A}, \alpha) \models \psi$ does not hold.
- We write $(\mathcal{A}, \alpha) \models \psi_1 \wedge \psi_2$ if both $(\mathcal{A}, \alpha) \models \psi_1$ and $(\mathcal{A}, \alpha) \models \psi_1$.
- We write $(\mathcal{A}, \alpha) \models \exists X\, \psi$ if $(\mathcal{A}, \alpha') \models \psi$ holds for some assignment α' of the free variables occurring in ψ such that $\alpha' \overset{X}{\sim} \alpha$.

  Here, $\alpha' \overset{X}{\sim} \alpha$ denotes that $\alpha'(y) = \alpha(y)$ for all individual or set variables y different from X, and $\alpha'(X)$ is an arbitrary subset of $\mathrm{dom}(\mathcal{A})$.

[4]In general, MSO also allows for function and constant symbols, but we omit these because we will not need them.

- We write $(\mathcal{A}, \alpha) \models \exists x\, \psi$ if $(\mathcal{A}, \alpha') \models \psi$ holds for some assignment α' of the free variables occurring in ψ such that $\alpha' \overset{x}{\sim} \alpha$.

  Here, $\alpha' \overset{x}{\sim} \alpha$ denotes that $\alpha'(y) = \alpha(y)$ for all individual or set variables y different from x, and $\alpha'(x)$ is an arbitrary element of $\operatorname{dom}(\mathcal{A})$.

The semantics for formulas involving the symbols $\vee$, $\rightarrow$ and $\forall$ can be derived from the cases for $\wedge$, $\neg$ and $\exists$ just as in first-order logic. If $(\mathcal{A}, \alpha) \models \varphi$ holds for all α, then we also write $\mathcal{A} \models \varphi$ and say that φ is *true* under $\mathcal{A}$. We write $\mathcal{A} \models \varphi(\boldsymbol{a})$, where $\boldsymbol{a}$ is a tuple of elements or subsets of $\operatorname{dom}(\mathcal{A})$, to denote $(\mathcal{A}, \alpha) \models \varphi$, where α is the variable assignment that sets the free variables $\boldsymbol{x}$ in φ to $\boldsymbol{a}$.

It is well known that for any fixed MSO (and even first-order) formula φ without free variables, the problem of deciding whether $\mathcal{A} \models \varphi$ holds for a given structure $\mathcal{A}$ is PSPACE-complete.

The following result, which is known as *Courcelle's theorem*, states a useful connection between MSO and the complexity of problems parameterized by treewidth.

Theorem 2.19 (Courcelle (1990) and Courcelle (1992))**.** *Let φ be a fixed MSO formula without free variables over a signature σ. There is an algorithm that decides in fixed-parameter linear time whether $\mathcal{A} \models \varphi$ holds for a given relational structure $\mathcal{A}$ over σ, where the parameter is the treewidth of $\mathcal{A}$.*

Thus, if we can express a problem in MSO, then it is FPT when parameterized by treewidth.

2.3.3 MSO Transductions

MSO can not only be used for deriving FPT results via Courcelle's theorem, but also for studying how the treewidth changes when we transform a structure into another structure. *MSO transductions* (Courcelle and Engelfriet 2012) are such transformations on structures that guarantee that the treewidth of the "output structure" is always bounded by the treewidth of the "input structure".[5]

Definition 2.20 (Courcelle and Engelfriet (2012, Definition 7.2))**.** A *definition scheme* from a signature σ to a signature σ' is a tuple $\langle \Delta, \Theta \rangle$ whose elements are tuples of MSO formulas of the following kind, where I denotes a finite set of arbitrary objects:[6]

[5]Actually Courcelle and Engelfriet (2012) define MSO transductions in much greater generality than we do here. We restrict ourselves to a special case because it suffices for our purposes.

[6]We do not specify the exact order of the elements of Δ and Θ for the following reason: We assume that the subscripts that we use in the definition of the contained formulas are ordered in an arbitrary way. From this we can obtain the order of the elements from the order of their subscripts.

- For each $i \in I$, the tuple Δ contains a formula δ_i with one free variable x. We call these formulas *domain formulas*.
- For each $R' \in \sigma'$ of arity k and all $\langle i_1, \dots, i_k \rangle \in I^k$, the tuple Θ contains a formula $\vartheta_{R', i_1, \dots, i_k}$ with k free variables $x_1, \dots, x_k$. We call these formulas *relation formulas*.

The intended meaning of the formulas in a definition scheme $\langle \Delta, \Theta \rangle$ is that they define how to transform an "input structure" $\mathcal{A}$ into an "output structure" $\mathcal{A}'$ in the following way: For every element $a \in \text{dom}(\mathcal{A})$ and each formula δ_i such that $\mathcal{A} \models \delta_i(a)$, we put a copy of a called (a, i) into $\text{dom}(\mathcal{A}')$. The domain formulas thus define the domain of the output structure. The relation formulas in turn define the relations in the output structure by determining which of the copies of the old domain elements are together in a relation.

Definition 2.21. A function that maps a structure $\mathcal{A}$ over a signature σ to a structure $\mathcal{A}'$ over a signature σ' is an *MSO transduction* from σ to σ' if there is a definition scheme $\langle \Delta, \Theta \rangle$ that satisfies the following conditions:

- For each $a \in \text{dom}(\mathcal{A})$ and δ_i contained in Δ, if $\mathcal{A} \models \delta_i(a)$, then $\text{dom}(\mathcal{A}')$ contains an element (a, i).

  This condition exactly characterizes the domain of $\mathcal{A}'$, that is, $\text{dom}(\mathcal{A}')$ only contains such elements of the form (a, i) and no others.

- For each $R' \in \sigma'$ of arity k, all $\delta_{i_1}, \dots, \delta_{i_k}$ occurring in Δ and all $\langle a_1, \dots, a_k \rangle \in \text{dom}(\mathcal{A})^k$, if $\mathcal{A} \models \vartheta_{R', i_1, \dots, i_k}(a_1, \dots, a_k)$ holds in addition to $\mathcal{A} \models \delta_{i_j}(a_j)$ for every j, then $R'^{\mathcal{A}'}$ contains a tuple $\langle (a_1, i_1), \dots, (a_k, i_k) \rangle$.

  This condition exactly characterizes each relation of $\mathcal{A}'$.

We call $\mathcal{A}$ an *input structure* of the transduction and $\mathcal{A}'$ is the corresponding *output structure*.

Thus an MSO transduction allows us to copy an input structure a fixed number of times, to filter those domain elements that satisfy a domain formula, and to define the relations of the output structure in terms of the relations of the input structure via the relation formulas. It is time for some examples (taken from the book by Courcelle and Engelfriet (2012), which contains several more).

Example 2.22. The following definition scheme formalizes an MSO transduction that transforms a structure $\mathcal{G}$ representing a directed graph into a structure $\mathcal{G}'$ representing the same graph but without without loops and isolated vertices. We represent directed graphs as structures as in Example 2.11 using the signature consisting just of the binary relation symbol E.[7]

$$\begin{aligned}\delta_1(x) &\equiv \exists y\big((\mathrm{E}(x,y) \vee \mathrm{E}(y,x)) \wedge x \neq y\big)\\ \vartheta_{1,1}(x,y) &\equiv \mathrm{E}(x,y) \wedge x \neq y\end{aligned}$$

As there is only one domain formula, we make at most one copy for each vertex in $\mathcal{G}$. In fact, by δ_1, we put a copy $(v,1)$ into $\mathcal{G}'$ for each vertex v that is adjacent to another vertex. This removes isolated vertices. The relation formula $\vartheta_{1,1}$ then makes sure that we draw an edge from a copy $(v,1)$ to a copy $(w,1)$ if and only if $\mathcal{G}$ contains an edge from v to w and $v \neq w$. △

Example 2.23. We define an MSO transduction that makes two copies of a directed graph and, for each vertex v with copies $(v,1)$ and $(v,2)$, draws an edge from $(v,1)$ to $(v,2)$.

$$\begin{aligned}\delta_1(x) &\equiv \delta_2(x) \equiv \top\\ \vartheta_{1,1}(x,y) &\equiv \vartheta_{2,2}(x,y) \equiv \mathrm{E}(x,y)\\ \vartheta_{1,2}(x,y) &\equiv x = y\\ \vartheta_{2,1}(x,y) &\equiv \bot\end{aligned}$$

The domain formulas unconditionally make two copies of each vertex. The formula $\vartheta_{1,1}$ then draws an edge from $(v,1)$ to $(w,1)$ if there was an edge from v to w, and $\vartheta_{2,2}$ does the same for the copies with number 2. So far we get two copies of the input graph. Now $\vartheta_{1,2}$ draws an edge from $(v,1)$ to $(v,2)$ for any vertex v. △

Not all transformations on structures are MSO transductions. In fact, the following observation, which follows from the definition of MSO transductions, gives us an indication of when it is not possible to find an MSO transduction for a transformation on structures.

Proposition 2.24 (Courcelle and Engelfriet (2012, Fact 1.37)). *For every MSO transduction τ there is an integer k such that every input structure $\mathcal{A}$ of τ satisfies $|\mathrm{dom}(\tau(\mathcal{A}))| \leqslant k \cdot |\mathrm{dom}(\mathcal{A})|$.*

Conversely, if we have a transformation on structures that leads to a more than linear increase in the domain size, then this transformation is no MSO transduction.

The following theorem follows from Courcelle and Engelfriet (2012, Theorem 7.47).[8]

[7]Since there is only a single relation symbol E in the signature, we write $\vartheta_{i,j}$ instead of $\vartheta_{\mathrm{E},i,j}$.

[8]Theorem 7.47 in that work is stated for transductions on (incidence) graphs, but generalizes to (incidence) structures by the following observations (Bruno Courcelle, personal communication): From

Theorem 2.25. *Let σ and σ' be relational signatures and let τ be a fixed MSO transduction from $\mathrm{Inc}(\sigma)$ to $\mathrm{Inc}(\sigma')$. For every relational structure $\mathcal{A}$ over σ, the treewidth of $\mathcal{A}'$ depends only on the treewidth of $\mathcal{A}$, where $\mathcal{A}'$ denotes the structure such that $\tau(\mathrm{Inc}(\mathcal{A})) = \mathrm{Inc}(\mathcal{A}')$ holds.*

What this theorem says is that a transformation that turns $\mathcal{A}$ into $\mathcal{A}'$ preserves bounded treewidth if we can find an MSO transduction that turns $\mathrm{Inc}(\mathcal{A})$ into $\mathrm{Inc}(\mathcal{A}')$.

Example 2.26 (based on Courcelle and Engelfriet (2012, Example 7.44))**.** Consider a mapping f that turns a directed graph G into a directed graph G' such that each edge (v, w) in G is replaced by the path (v, e, w), where e is a new vertex corresponding to (v, w). Formally, $\mathrm{V}(G') = \mathrm{V}(G) \cup \mathrm{E}(G)$ and $\mathrm{E}(G') = \{\langle v, (v, w)\rangle, \langle (v, w), w\rangle \mid (v, w) \in \mathrm{E}(G)\}$. We will show that f preserves bounded treewidth, that is, for each element G of a class of graphs of bounded treewidth, $f(G)$ also has bounded treewidth.

We will use Theorem 2.25, but it applies to MSO transductions that transform incidence structures, so we must present a definition scheme for a transduction that operates on incidence structures of directed graphs (cf. Definition 2.12). To this end, we represent a directed graph G as a structure $\mathcal{G}$ over the signature $\{\mathrm{E}, \mathrm{in}_1, \mathrm{in}_2\}$ as follows, where E is unary, and in_1 and in_2 are binary relation symbols: We use $\mathrm{dom}(\mathcal{G}) = \mathrm{V}(G) \cup \mathrm{E}(E)$, $\mathrm{E}^{\mathcal{G}} = \mathrm{E}(G)$, $\mathrm{in}_1^{\mathcal{G}} = \{\langle (v, w), v\rangle \mid (v, w) \in \mathrm{E}(G)\}$ and $\mathrm{in}_2^{\mathcal{G}} = \{\langle (v, w), w\rangle \mid (v, w) \in \mathrm{E}(G)\}$.

The transduction given by the following definition scheme makes three copies of a graph G and constructs the desired graph G' from these copies by filtering vertices and drawing appropriate edges.

$$\begin{aligned}
\delta_1(x) &\equiv \top \\
\delta_2(x) &\equiv \delta_3(x) \equiv \mathrm{E}(x) \\
\vartheta_{\mathrm{E},1}(x) &\equiv \bot \\
\vartheta_{\mathrm{E},2}(x) &\equiv \vartheta_{\mathrm{E},3}(x) \equiv \top \\
\vartheta_{\mathrm{in}_1,2,1}(x, y) &\equiv \mathrm{in}_1(x, y) \\
\vartheta_{\mathrm{in}_1,3,1}(x, y) &\equiv \vartheta_{\mathrm{in}_2,2,1}(x, y) \equiv \mathrm{E}(x) \wedge x = y \\
\vartheta_{\mathrm{in}_2,3,1}(x, y) &\equiv \mathrm{in}_2(x, y)
\end{aligned}$$

every MSO transduction that transforms $\mathrm{Inc}(\mathcal{A})$ into $\mathrm{Inc}(\mathcal{A}')$ we easily get an MSO transduction that transforms G into G', where G and G' are the incidence graphs of $\mathcal{A}$ and $\mathcal{A}'$, respectively. Thus the clique-width of G' depends only on the clique-width of G by Courcelle and Engelfriet (2012, Theorem 7.47). Since incidence structures are uniformly q-sparse (see Courcelle and Engelfriet 2012, Definition 9.36), where q depends on the signature, bounded clique-width coincides with bounded treewidth by Courcelle and Engelfriet (2012, Theorem 9.62). We thus get that the treewidth of $\mathrm{Inc}(\mathcal{A}')$ depends only on the treewidth of $\mathrm{Inc}(\mathcal{A})$. Since the treewidth of a structure is linearly related to the treewidth of its incidence structure, the treewidth of $\mathcal{A}'$ depends only on the treewidth of $\mathcal{A}$.

All other formulas $\vartheta_{\mathrm{in}_i,j,k}$ are defined as $\bot$.

For each vertex or edge in G, we have a vertex in G'. The formula δ_1 creates these elements and by $\vartheta_{\mathrm{E},1}$, which is defined as $\bot$, these elements are not edges but vertices.

For each edge e from v to w in G, we have two edges in G', namely (v, e) and (e, w). Since $\delta_2(e)$ and $\delta_3(e)$ are true (under $\mathcal{G}$), we correctly create these elements, and they are indeed edges as $\vartheta_{\mathrm{E},2}(e)$ and $\vartheta_{\mathrm{E},3}(e)$ are true. The formulas $\vartheta_{\mathrm{in}_i,j,k}$ make sure that these edges are incident to the correct vertices.

For instance, let e be an edge from v to w in G. Since $\delta_1(e)$, $\delta_1(v)$ and $\delta_2(e)$ are true, we create the elements $(e, 1)$, $(v, 1)$ and $(e, 2)$. The elements $(e, 1)$ and $(v, 1)$ are the vertices in G' corresponding to e and v, respectively. The element $(e, 2)$ shall represent the edge from $(v, 1)$ to $(e, 1)$ in G', and indeed $\vartheta_{\mathrm{in}_1,2,1}(e, v)$ and $\vartheta_{\mathrm{in}_2,2,1}(e, e)$ are true. The former dictates that the edge $(e, 2)$ starts at $(v, 1)$, and the latter makes this edge go to $(e, 1)$.

Note that we would not have been able to define this graph transformation without incidence signatures: We had to create a vertex in the output for every edge in the input, but without using an incidence signature the edges of the input would not have been available in the domain. The only possibility of obtaining vertices in the output would have been to copy *vertices* of the input – but the number of edges in a graph can be quadratic in the number of vertices. Since MSO transductions can only copy each domain element a constant number of times, there would have been no way to make enough copies. Indeed, Proposition 2.24 tells us that this transformation is not an MSO transduction if we only have vertices in the domain of the input structure. $\triangle$

2.4 Answer Set Programming

Answer Set Programming (ASP) (Gelfond and Lifschitz 1988; Marek and Truszczyński 1999) is a declarative problem solving paradigm that enjoys considerable popularity for solving computationally hard combinatorial problems. Introductions can be found in works by (Brewka, Eiter and Truszczyński 2011; Gebser et al. 2012; Lifschitz 2008). ASP offers a convenient, powerful language for encoding problems in a succinct way. Its main feature is that the user only needs to write a formal specification of the problem, whereas the actual computation is delegated to dedicated systems (see, e.g., Gebser et al. 2007; Alviano et al. 2013; Alviano et al. 2015; Leone et al. 2006; Elkabani, Pontelli and Son 2005). ASP has been used in many areas. Prominent applications include problems in bioinformatics (Gebser et al. 2011), product configuration (Soininen and Niemelä 1998) and decision support systems for space shuttle controllers (Nogueira et al. 2001).

When solving a problem in ASP, we specify a set of *rules* that formalize conditions that the solutions must fulfill. As we will see, this allows us to solve problems on a very high level of abstraction. The high declarativity is amplified by the fact that the order of the rules (and the order of the elements within a rule) is irrelevant. This is in contrast to related languages such as Prolog, where order does matter in general.

2.4.1 Historical Influences on ASP

Rules in ASP in fact look very similar to logic programs in Prolog. It is important to note, however, that the two languages are quite different. Although one of the roots of ASP is indeed logic programming as understood by Prolog, the focus of the latter is on finding *proofs* of certain goals (Lloyd 1987), whereas with ASP the focus is on finding *models* of logical expressions. In this respect, ASP resembles more Boolean satisfiability (Sat) solving (Biere et al. 2009). Indeed, state-of-the-art solvers for ASP use techniques that are heavily influenced by Sat solvers and often perform quite well on very hard problems even on instances of considerable size. Moreover, ASP solvers have also been influenced by techniques from constraint satisfaction (Rossi, Beek and Walsh 2006; Dechter 2003).

Beside logic programming (as understood by Prolog), Sat solving and constraint satisfaction, ASP also has roots in knowledge representation (Harmelen, Lifschitz and Porter 2008) – in particular in non-monotonic reasoning. In fact, ASP has a non-monotonic semantics, that is, solutions may be invalidated by adding new information.

Finally, one of the major influences of ASP have been deductive databases and especially the well-known query language *Datalog* (Ceri, Gottlob and Tanca 1990; Abiteboul, Hull and Vianu 1995). Indeed, Datalog can be seen as a restricted version of ASP. The restrictions of Datalog compared to ASP guarantee that a Datalog program always has exactly one solution, whereas ASP programs may have zero, one or many solutions.

2.4.2 Examples for Problem Solving in ASP

The fact that ASP programs may have many solutions in general can actually be considered a very useful feature. This typically allows us to write ASP programs in such a way that solution candidates may be non-deterministically guessed and, by subsequently enforcing the constraints that solutions of the problem must obey, the solutions of the problem correspond exactly to the solutions of the ASP program. In ASP, we can thus solve many problem with a guess-and-check approach, which allows for succinct specifications that are easy to understand.

Example 2.27. Even though we yet have to define the syntax and semantics of ASP, we present a program to give an intuition of problem solving with ASP. Consider the following ASP program, which encodes the NP-complete 3-Colorability problem, which is a special case of the Graph Coloring problem that we considered in Section 2.2.

$$\begin{aligned}
\texttt{red(X)} \lor \texttt{green(X)} \lor \texttt{blue(X)} &\leftarrow \texttt{vertex(X)}.\\
\bot &\leftarrow \texttt{edge(X,Y), red(X), red(Y)}.\\
\bot &\leftarrow \texttt{edge(X,Y), green(X), green(Y)}.\\
\bot &\leftarrow \texttt{edge(X,Y), blue(X), blue(Y)}.
\end{aligned}$$

In the first line, we guess a color for each vertex. The remaining lines enforce that adjacent vertices do not have the same color. We can give this program to an ASP system together with an input graph specified using the `vertex` and `edge` predicates, and the system will print all proper 3-colorings of the graph. (As we will see, solutions of ASP programs must satisfy a certain kind of minimality as per the answer-set semantics. This ensures that a vertex cannot have two colors.) △

Next we give an example containing advanced ASP language constructs called *weak constraints* and *aggregates*. With weak constraints, we can compute solutions that minimize a certain function. Intuitively, weak constraints specify conditions that, when violated, incur a penalty on the cost of a solution, and we are looking for solutions of minimum cost. Aggregates are constructs that, for instance, allow us to compare a value to a certain sums of integers.

Example 2.28. Suppose we want to solve a knapsack problem where we are given a *capacity* c as a positive integer, a set S of objects, and for each object $x \in S$, a positive integer w_x for the *weight* of x and a positive integer v_x for the *value* of x. The objective is to find a subset K of S such that $\sum_{x\in K} w_x \leqslant c$ and the sum $\sum_{x\in K} v_x$ is maximal over all such subsets. The following ASP program encodes this problem. We assume that each object x is declared in the input as `object`(x), its weight and value are given by `weight`(x, w_x) and `value`(x, v_x), respectively, and the capacity c is given by `capacity`(c).

$$\begin{aligned}
\texttt{take(X)} \lor \texttt{leave(X)} &\leftarrow \texttt{object(X)}.\\
\bot &\leftarrow \texttt{\#sum}\{\ \texttt{W,X : take(X), weight(X,W)}\ \} > \texttt{C, capacity(C)}.\\
&\leftsquigarrow \texttt{take(X), value(X,V)}.\quad [-\texttt{V,X}]
\end{aligned}$$

In the first line, we guess for each object if we take it into our subset K or not.

The constraint on the second line enforces the capacity bound. Intuitively, the : symbol can be read as "such that". In natural language, the aggregate `#sum{ W,X : take(X), weight(X,W) }` $>$ `C` expresses "the sum of all weights `W` of objects `X` such that we take `X` into K and `X` has weight `W` is greater than `C`". The reason that we write `W,X` and not just `W` is that otherwise duplicate weights would be eliminated; this is not what we want because multiple objects may have the same weight. The `#sum` aggregate always sums over just the first element in each tuple (in the set of tuples satisfying the condition to the right of the : symbol) while the remaining elements of the tuples are just for avoiding elimination of duplicates.

The third line is a weak constraint, indicated by the $\rightsquigarrow$ symbol. Intuitively, it says that "if we take an object X of value V into our solution candidate K, then add $-$V to the cost of K". Thus the cost of K will always be $-\sum_{x \in K} v_x$. Since solutions of the program minimize the total cost of the solution candidates, this only gives us solutions K where $\sum_{x \in K} v_x$ is maximal. Note that again we use $-$V, X instead of just $-$V for the same reason as before. $\triangle$

2.4.3 Syntax and Semantics

We now briefly present syntax and semantics of ASP as defined in the *ASP-Core-2* language specification (Calimeri et al. 2015).[9] This specification is based on the semantics defined in the paper by Faber, Pfeifer and Leone (2011), which extends the classical ASP semantics (Gelfond and Lifschitz 1988) to programs containing aggregates.[10] Moreover, the specification (Calimeri et al. 2015) includes weak constraints. To keep the presentation clear, we will first introduce ASP without weak constraints and then note how this concept can be added.

Syntax

A *program* in ASP is a set of *rules*, which have the following form:

$$a_1 \vee \ldots \vee a_n \leftarrow b_1, \ldots, b_k, \texttt{not } b_{k+1}, \ldots, \texttt{not } b_m.$$

The *head* of a rule r is the set denoted by $H(r) = \{a_1, \ldots, a_n\}$, the *positive body* of r is the set $B^+(r) = \{b_1, \ldots, b_k\}$, and the *negative body* of r is the set $B^-(r) = \{\texttt{not } b_{k+1}, \ldots, \texttt{not } b_m\}$. The *body* of r is now defined as $B(r) = B^+(r) \cup B^-(r)$.

If the head of a rule is empty, then we call the rule a *constraint*. When writing a constraint, we may write $\perp$ before the $\leftarrow$ symbol or we may just omit $\perp$. If the body of a rule is empty, then we we may omit the $\leftarrow$ symbol. If the body of a rule is empty and the head consists of a single atom, then we call the rule a *fact*. A program Π is called *positive* if the negative body of each rule in Π is empty.

All elements of the heads or the bodies of rules are called *atoms*. An atom can either be a *predicate atom* or an *aggregate atom*. In rule heads, we only allow predicate atoms. A *predicate atom* has the form $p(t_1, \ldots, t_n)$, where p is called a *predicate*. The elements $t_1, \ldots, t_n$ in a predicate atom are called *terms*. A term is either a *constant* or a *variable*.

[9]We reuse substantial parts of the language specification (Calimeri et al. 2015) in our statement of the syntax and semantics, and we also base parts of our exposition on a paper by Faber, Pfeifer and Leone (2011). In order to keep the presentation as clear as possible, however, we make some slight simplifications by omitting some constructs that are allowed by the language specification but not used in this work.

[10]Many other semantics have been proposed for aggregates in ASP (see., e.g., Pelov, Denecker and Bruynooghe 2004; Pelov, Denecker and Bruynooghe 2007; Ferraris 2011; Simons, Niemelä and Soininen 2002), but the semantics in the language specification (Calimeri et al. 2015) defines the language such that various important semantics are in agreement.

It is customary to write predicates and constants as (strings starting with) lower-case symbols and variables as (strings starting with) upper-case symbols. Moreover, constants may be integers. We call a program or a part of a program (like atoms, rules, etc.) *ground* if it contains no variables. A predicate is called *extensional* in a program Π if it only occurs in rule bodies of Π. A *literal* is an atom a or its negated form **not** a.

An *aggregate atom* has the form

$$\texttt{\#sum}\{\ e_1;\ldots;e_n\ \} \prec u,$$

where $e_1, \ldots, e_n$ are *aggregate elements*, $\prec$ is either $<$, $\leqslant$ $=$, $\neq$, $\geqslant$ or $>$, and u is is a term.[11] An *aggregate element* has the form

$$t_1, \ldots, t_m : l_1, \ldots, l_n,$$

where $t_1, \ldots, t_m$ are terms and $l_1, \ldots, l_n$ are predicate atoms (possibly negated by putting **not** in front). We call a predicate atom *nested* if it occurs in an aggregate element, and we call it *unnested* otherwise.

We say that a variable occurring in a rule r is *global* if it occurs in an unnested predicate atom in r, otherwise it is *local*.[12]

A rule r is *safe* if it satisfies the following conditions:[13]

1. Every global variable that occurs in r occurs in an unnested predicate atom a in the positive body of r.
2. Every local variable in an aggregate element $t_1, \ldots, t_m : l_1, \ldots, l_n$ of r occurs in a positive atom l_i.

A program is safe if all its rules are safe. We only admit ASP programs that are safe.

Semantics

We define the semantics of ASP in terms of ground programs. For this, we first show how arbitrary programs can be transformed into ground programs.

[11]The ASP language specification (Calimeri et al. 2015) also allows for other aggregate functions than `#sum`, namely also `#count`, `#max` and `#min`. We will only describe `#sum` because it is the only aggregate function that we use in this work.

[12]Local variables that appear in different aggregate elements of the same rule are different objects even if we use the same symbol for them.

[13]Actually Calimeri et al. (2015) use a less restrictive version of safety by also allowing variables to be bound by aggregate atoms involving equality as the comparison operator, but we omit this to keep the presentation clearer. Such aggregates do not appear in this thesis. Moreover, the language specification allows variables to be bound by so-called built-in atoms, which we also omitted because we do not use them.

The *Herbrand universe* of a program Π, denoted by U_Π, is the set of all constants occurring in Π. The *Herbrand base* of Π is the set of all predicate atoms that can be constructed using predicates occurring in Π and constants in U_Π. We denote this set by B_Π. Subsets of B_Π are called *interpretations* and intuitively correspond to the predicate atoms that we set to true while all other predicate atoms are set to false.

A *substitution* σ in a program Π maps a set of variables to U_Π. For any object x that is Π itself or part of Π (like atoms, aggregate elements, rules, etc.), we write $\sigma(x)$ to denote the result of replacing each variable v in x by $\sigma(v)$ if σ is defined for this variable. We say that a substitution γ in Π is a *global substitution for a rule* r if the domain of γ is exactly the set of global variables in r. A substitution λ in Π is a *local substitution for an aggregate element* e in some rule if the domain of λ is exactly the set of local variables in e.

Given a set E of aggregate elements from a rule without global variables, we write $\text{inst}(E)$ to denote the following set: For each aggregate element $e \in E$ and each local substitution λ for e, the ground aggregate element $\lambda(e)$ is in $\text{inst}(E)$. Now, for any global substitution γ for a rule r, we can obtain a corresponding *ground instance* of r by the following two steps:

1. We first replace r with $\gamma(r)$.
2. We replace each aggregate atom $\texttt{\#sum}\{\, e_1; \dots; e_n \,\} \prec u$ occurring in $\gamma(r)$ by the ground aggregate atom $\texttt{\#sum}\{\, \text{inst}(\{e_1, \dots, e_n\}) \,\} \prec u$.[14]

We now define the *ground instantiation* $\text{Ground}(\Pi)$ of a program Π as the set of all ground instances of all rules in Π. It is easy to see that $\text{Ground}(\Pi)$ is indeed a ground program.

The following example of ground instantiations is from Faber, Pfeifer and Leone (2011, Example 2.7).

Example 2.29. Let Π be the following program:

$\texttt{q}(1) \vee \texttt{p}(2,2).$
$\texttt{q}(2) \vee \texttt{p}(2,1).$
$\texttt{t}(\texttt{X}) \leftarrow \texttt{q}(\texttt{X}),\ \texttt{\#sum}\{\ \texttt{Y} : \texttt{p}(\texttt{X},\texttt{Y})\ \} > 1.$

The Herbrand universe U_Π is $\{1, 2\}$ and the ground instantiation $\text{Ground}(\Pi)$ is the following ground program:

$\texttt{q}(1) \vee \texttt{p}(2,2).$
$\texttt{q}(2) \vee \texttt{p}(2,1).$
$\texttt{t}(1) \leftarrow \texttt{q}(1),\ \texttt{\#sum}\{\ 1 : \texttt{p}(1,1);\ 2 : \texttt{p}(1,2)\ \} > 1.$
$\texttt{t}(2) \leftarrow \texttt{q}(2),\ \texttt{\#sum}\{\ 1 : \texttt{p}(2,1);\ 2 : \texttt{p}(2,2)\ \} > 1.$ $\triangle$

[14]For this, we write the resulting aggregate elements in any order and separate them with ; symbols.

Given an interpretation I of a program Π, we say that a ground predicate atom a is *true* under I if $a \in I$, otherwise it is *false*. Similarly, we say that a negated ground predicate atom a, written as $\texttt{not}\ a$, is *true* under I if $a \notin I$, otherwise it is *false*. Before we define the conditions under which aggregate atoms are true, we define $\text{eval}(E, I)$, where E is a set of ground aggregate elements, as the set consisting of those tuples $\langle t_1, \ldots, t_m \rangle$ such that E contains an element $t_1, \ldots, t_m : l_1, \ldots, l_n$ and all of $l_1, \ldots, l_n$ are true under I. Now we define that a ground aggregate atom $\texttt{\#sum}\{\ e_1; \ldots; e_n\ \} \prec u$ is *true* under I if $s \prec u$, where s is the sum of all integers in the multiset $\{t_1 \mid (t_1, \ldots, t_m) \in \text{eval}(\{e_1, \ldots, e_n\})\}$.

An interpretation I of a program Π *satisfies* a rule r in $\text{Ground}(\Pi)$ if $B(r)$ being true under I implies $H(r)$ being true under I. We say that I is a *model* of Π if it satisfies every rule in $\text{Ground}(\Pi)$.

Only models satisfying a certain minimality condition are answer sets of a program. For this, we define Π^I, called the *reduct* of a program Π w.r.t. an interpretation I, as the set of those rules in $\text{Ground}(\Pi)$ whose body elements are all true under I. Now I is an *answer set* of Π if I is a model of Π and no proper subset of I is a model of Π. Given an answer set I and a predicate p, we say that the set $\{\langle t_1, \ldots, t_n \rangle \mid p(t_1, \ldots, t_n) \in I\}$ is the *extension* of p under I.

Weak Constraints

Beside rules, ASP programs may contain *weak constraints*. These are expressions of the following form:

$$\leftsquigarrow b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m.\ [w@l, t_1, \ldots, t_n]$$

All definitions about the syntax of rules also apply to weak constraints except for the fact that weak constraints always have an empty head and they contain the expression $[w@l, t_1, \ldots, t_n]$, where w and l are terms called the *weight* and *level* of the weak constraint, respectively, and $t_1, \ldots, t_n$ are terms. We may omit the expression $@l$ if $l = 0$, so by default we assume that a weak constraint has level 0.

It is straightforward to extend our earlier definition of the ground instantiation of a program to programs with weak constraints. Answer sets of programs with weak constraints are defined in the same way as without weak constraints, that is, weak constraints are not taken into account for determining whether an interpretation is an answer set.

For any interpretation I of a program Π, we define $\text{viol}(\Pi, I)$ as the set of all elements $(w@l, t_1, \ldots, t_n)$ such that $[w@l, t_1, \ldots, t_n]$ occurs in a weak constraint r in $\text{Ground}(\Pi)$ and every body element of r is true under I. Intuitively, these are the weights (and levels, etc.) of the weak constraints violated by I.

The idea behind the levels is that minimizing the sum of the weights from weak constraints with higher levels has priority over minimizing that sum for lower levels.

We formalize this by defining $\text{cost}_l(\Pi, I)$ for every integer l as the sum of all integers in the multiset $\{w \mid (w@l, t_1, \dots, t_n) \in \text{viol}(\Pi, I)\}$.

Now we say that an answer set I of Π *dominates* an answer set J of Π if $\text{cost}_l(\Pi, I) < \text{cost}_l(\Pi, J)$ holds for some integer l and at the same time $\text{cost}_{l'}(\Pi, I) = \text{cost}_{l'}(\Pi, J)$ holds for all $l' > l$. An answer set I of Π is *optimal* if no answer set of Π dominates it. We also call optimal answer sets the *solutions* of Π.

Example 2.30. Consider the following ground program Π, which has three weak constraints at priority level 2 and one weak constraint at level 1.

$\texttt{p} \lor \texttt{q}.$

$\texttt{q} \lor \texttt{r}.$

$\texttt{r} \lor \texttt{p}.$

$\leftsquigarrow \texttt{r}.$	$[6@2, \texttt{a}]$
$\leftsquigarrow \texttt{p}.$	$[3@2, \texttt{b}]$
$\leftsquigarrow \texttt{q}, \textbf{not } \texttt{r}.$	$[3@2, \texttt{c}]$
$\leftsquigarrow \textbf{not } \texttt{p}.$	$[8@1, \texttt{d}]$

For computing the three answer sets $\{\texttt{p}, \texttt{q}\}$, $\{\texttt{q}, \texttt{r}\}$ and $\{\texttt{p}, \texttt{r}\}$ of this program, the weak constraints play no role. We now determine the costs of these answer sets.

- The answer set $\{\texttt{p}, \texttt{q}\}$ violates the second and the third weak constraint, so $\text{cost}_1(\Pi, \{\texttt{p}, \texttt{q}\}) = 0$ and $\text{cost}_2(\Pi, \{\texttt{p}, \texttt{q}\}) = 6$. Note that here it is important that the two different constants $\texttt{b}$ and $\texttt{c}$ are present in order two distinguish the two summands, which are both 3. Otherwise the cost at level 2 would only be 3 due to the discussed elimination of duplicates.

- The answer set $\{\texttt{q}, \texttt{r}\}$ violates the first and the last weak constraint, so we get $\text{cost}_1(\Pi, \{\texttt{q}, \texttt{r}\}) = 8$ and $\text{cost}_2(\Pi, \{\texttt{q}, \texttt{r}\}) = 6$.

- The answer set $\{\texttt{p}, \texttt{r}\}$ violates the first two weak constraints, so $\text{cost}_1(\Pi, \{\texttt{p}, \texttt{r}\}) = 0$ and $\text{cost}_2(\Pi, \{\texttt{p}, \texttt{r}\}) = 9$.

Clearly $\{\texttt{p}, \texttt{r}\}$ is dominated by the other two answer sets because its cost at level 2 is higher. The answer sets $\{\texttt{p}, \texttt{q}\}$ and $\{\texttt{q}, \texttt{r}\}$ have the same cost at level 2, but $\{\texttt{p}, \texttt{q}\}$ dominates $\{\texttt{q}, \texttt{r}\}$ because its cost at level 1 is lower. Hence $\{\texttt{p}, \texttt{q}\}$ is the unique optimal answer set of Π. △

Although weak constraints are substantially different from rules, in the following we often say "rules" when in fact we mean "rules or weak constraints". We thus regard weak constraints as a special kind of rules in the remainder of this work.

2.4.4 Complexity

There have been numerous investigations about the complexity of ASP under various restrictions. For a survey of important results (but without regard to aggregates), see the work by Dantsin et al. (2001). Here, we only mention those results that are of direct relevance for our work.

We are mainly interested in the complexity of the following decision problem.

Answer Set Existence

Input:	An ASP program Π
Question:	Does Π have an answer set?

Sometimes it makes sense to also consider a slightly different problem:

Brave Reasoning

Input:	An ASP program Π and a ground predicate atom p
Question:	Does an optimal answer set of Π contain p?

First we consider the special case where the programs in the input of these problems are ground. According to classical results by Eiter and Gottlob (1995), Answer Set Existence and Brave Reasoning are at the second level of the polynomial hierarchy, namely complete for Σ_2^P, if the programs are ground and contain neither weak constraints nor aggregates. Faber et al. (2008) showed that adding a restricted version of aggregates does not increase the complexity. This has been extended by Faber, Pfeifer and Leone (2011), where the same result was obtained without this restriction. If we add weak constraints (that support different priority levels as we defined them here), then the complexity of Brave Reasoning increases slightly, namely from Σ_2^P to Δ_3^P (Buccafurri, Leone and Rullo 2000) – but note that this class is still part of the second level of the polynomial hierarchy. Also note that the complexity of Answer Set Existence remains the same with or without weak constraints because weak constraints do not play a role for the question of whether an answer set exists.

Next we consider ASP programs with variables. Note that the size of ground instantiations can be exponential. It is thus not surprising that generally the complexity increases in the presence of variables. Indeed, Eiter, Gottlob and Mannila (1997) showed that then the problems are $\mathsf{NEXPTIME}^{\mathsf{NP}}$-complete, but they did not consider weak constraints or aggregates. For the case where aggregates are allowed, the same completeness results were obtained by Faber et al. (2008). The same work also showed that additionally allowing weak constraints leaves Answer Set Existence complete for $\mathsf{NEXPTIME}^{\mathsf{NP}}$, but it makes Brave Reasoning complete for $\mathsf{EXPTIME}^{\Sigma_2^{\mathsf{P}}}$.

In this work, we are more interested in the *data complexity* of ASP. The term data complexity comes from the study of query languages, where we are faced with problem instances consisting of a query and data, and the question is whether the query holds on the data (or, more generally, to evaluate the query on the data). Indeed, ASP fits well into this framework since it is commonly used for encoding a problem as a non-ground program and then combining this encoding with ground facts describing an actual instance of that problem. When we are dealing with the data complexity of Answer Set Existence or Brave Reasoning, we mean the complexity of a variant of the respective problem where the program in the input is the union of a non-ground part, which is fixed, and input facts, which may vary.

As the non-ground part Π is fixed and only the input facts F vary when we consider data complexity, the size of ground instantiations of $\Pi \cup F$ is always polynomial in the size of F. The reason is that then each rule in $\Pi \cup F$ contains a bounded number of variables, so each rule can only generate a polynomial number of ground instances. Eiter, Gottlob and Mannila (1997) and Faber et al. (2008) showed that the data complexity of Answer Set Existence and Brave Reasoning is in fact the same as the complexity in the ground case. We summarize this in the following theorems.

Theorem 2.31 (Eiter, Gottlob and Mannila (1997) and Faber et al. (2008))**.** *It is Σ_2^{P}-complete to decide for a fixed ASP program Π and a given set F of input facts whether $\Pi \cup F$ has an answer set.*

Theorem 2.32 (Eiter, Gottlob and Mannila (1997) and Faber et al. (2008))**.** *It is Δ_3^{P}-complete to decide for a fixed ASP program Π and a given set F of input facts whether a given ground atom is true in an optimal answer set of $\Pi \cup F$.*

There have also been some investigations about the expressive power of ASP. Eiter, Gottlob and Mannila (1997) showed that ASP captures Σ_2^{P} (even without weak constraints or aggregates). This means that ASP, when viewed as a database query language, can express all queries whose corresponding evaluation problem is in Σ_2^{P}.

2.4.5 Treewidth of Ground ASP Programs

We can easily apply the parameter treewidth to ground ASP programs by defining a suitable representation as a graph. For this, we use a notion that is quite similar to the primal graph of Sat instances that appeared in Example 2.3.

Definition 2.33. The *primal graph* of a ground ASP program Π is the graph whose vertices are the predicate atoms occurring in Π and that has an edge between two atoms if they appear together in a rule in Π.

On ground ASP programs, the problems Answer Set Existence and Brave Reasoning, parameterized by the treewidth of the primal graph, are fixed-parameter tractable (Gottlob, Pichler and Wei 2010a; Pichler et al. 2014; Fichte et al. 2017). In fact, these problems can even be solved in *linear* time when the treewidth is bounded by a constant.

2.4.6 Input Facts as Relational Structures

Every set of input facts for a program Π can be represented as a relational structure in various ways. The straightforward way is to choose the extensional predicates in Π as the signature and to interpret each such predicate as its extension in the input facts. In addition, since ASP systems always assume that the Herbrand base is ordered (in an arbitrary way), we can assume that the signature also contains a binary successor relation denoted by succ, which is interpreted according to an arbitrary order of the domain elements.

Definition 2.34. Let Π be an ASP program whose set of extensional predicates we denote by τ. For every set F of input facts for Π, we define the *fact structure* of F to be the structure $\mathcal{F}$ over $\tau \cup \{\text{succ}\}$ where $\text{dom}(\mathcal{F})$ consists of all ASP constants occurring in F, and for every predicate $p \in \tau$ it holds that $p^{\mathcal{F}}$ is the set of all k-tuples $\boldsymbol{a}$ such that $p(\boldsymbol{a})$ is a fact in F. We interpret the succ relation as an arbitrary, fixed successor relation of the ASP constants. By slight abuse of notation, we sometimes write $\mathcal{F}$ in place of F. For instance, we may write $\text{Ground}(\Pi \cup \mathcal{F})$ instead of $\text{Ground}(\Pi \cup F)$ to denote the ground instantiation of Π together with the input facts F. The *input structures* of a program Π comprise the fact structures of all sets of input facts for Π.

Example 2.35. Let Π be an ASP program whose only extensional predicate is p, and let F be the set of input facts for Π consisting of $p(a,b)$ and $p(a,c)$. The fact structure $\mathcal{F}$ of F contains the domain elements a, b and c, and it interprets the binary relation symbol p as $\{\langle a,b\rangle, \langle a,c\rangle\}$. Moreover, it contains an arbitrary interpretation of the succ relation, for instance $\text{succ}^{\mathcal{F}} = \{\langle a,b\rangle, \langle b,c\rangle\}$. △

In this work, we are also interested in representing input facts as *incidence structures* (Definition 2.12) of fact structures. Now we choose the extensional predicates (and the successor relation succ) as a *base* signature for our incidence structures.

Example 2.36. Continuing Example 2.35, we denote the facts $\langle a,b\rangle$ and $\langle a,c\rangle$ in $p^{\mathcal{F}}$ by ab_p and ac_p, respectively, and we denote the facts $\langle a,b\rangle$ and $\langle b,c\rangle$ in $\text{succ}^{\mathcal{F}}$ by ab_{succ} and bc_{succ}, respectively. Now the domain of $\text{Inc}(\mathcal{F})$ is $\{a, b, c, ab_p, ac_p, ab_{\text{succ}}, bc_{\text{succ}}\}$. The relations are interpreted as follows:

$$\begin{aligned}
p^{\mathrm{Inc}(\mathcal{F})} &= \{ab_p, ac_p\} \\
\mathrm{succ}^{\mathrm{Inc}(\mathcal{F})} &= \{ab_{\mathrm{succ}}, bc_{\mathrm{succ}}\} \\
\mathrm{in}_1^{\mathrm{Inc}(\mathcal{F})} &= \{\langle ab_p, a\rangle, \langle ac_p, a\rangle, \langle ab_{\mathrm{succ}}, a\rangle, \langle bc_{\mathrm{succ}}, b\rangle\} \\
\mathrm{in}_2^{\mathrm{Inc}(\mathcal{F})} &= \{\langle ab_p, b\rangle, \langle ac_p, c\rangle, \langle ab_{\mathrm{succ}}, b\rangle, \langle bc_{\mathrm{succ}}, c\rangle\}
\end{aligned}$$

△

CHAPTER 3

Treewidth-Preserving Classes of Answer Set Programs

The performance of modern ASP solvers is heavily influenced by the treewidth of the given ground input program. Indeed, an empirical evaluation in a paper by Bliem et al. (2017) revealed that the solving time increases drastically when the treewidth of the input increases but the size and the manner of construction of the programs remain the same. The objective of this chapter is to take advantage of this insight by leveraging the treewidth-sensitivity of ASP solvers when designing ASP encodings.

To solve a non-ground ASP program, ASP systems usually first invoke a *grounder* that transforms an ASP program into an equivalent set of ground rules. A "naive" grounder blindly instantiates variables in a program Π by all possible ground terms and thus produces the complete ground instantiation $\text{Ground}(\Pi)$ as defined in Section 2.4.3. Grounders in practice, on the other hand, employ sophisticated techniques in order to keep the resulting ground program as small as possible. As these techniques differ between systems, we define a simplified notion of grounding that is easier to study. Intuitively, our notion of grounding omits rules whose positive body contains an atom that cannot possibly be derived, and it also omits aggregate elements whose condition cannot possibly be satisfied.

It is usually possible to model the same problem in different ways as a non-ground ASP program. For some problems, we may be able to come up with an encoding that behaves nicely in the sense that the treewidth of the grounding is small whenever the input has small treewidth. We illustrate this with the following example.

Example 3.1. Reachability can be modeled in different ways using ASP. One way would be to use the transitive closure of the edge relation of a graph as follows (where e is the predicate representing graph edges and r the predicate to mark reachable vertices):

```
t(X,Y) ← e(X,Y).
t(X,Z) ← t(X,Y), e(Y,Z).
  r(Y) ← t(X,Y), start(X).
```

Such an encoding, however, causes any two (connected) vertices in the input graph to appear together in a rule after grounding (in place of the variables X and Z in the second rule). This then causes the primal graph of the ground program to contain a clique whose size equals the number of vertices of the original input graph, resulting in a high treewidth. It is more advisable to use the following encoding instead:

```
r(X) ← start(X).
r(Y) ← e(X,Y), r(X).
```

Here, not only is the grounding smaller, but also the treewidth decreases dramatically. In fact, it now solely depends on the treewidth (and not the size) of the input graph. △

Hence, the way a problem is encoded can influence the treewidth of the ground program considerably, and as the experiments in the work by Bliem et al. (2017) have shown, this also has a massive impact on the solving performance. Due to the grounding step, however, it is not obvious at the time of writing a non-ground ASP encoding how to achieve a low-treewidth grounding and the benefits that come with it. This is in contrast to, for example, Sat formulas that can be generated directly while keeping treewidth in mind.

The contribution of this chapter is a study of the following two classes of non-ground ASP programs in terms of the effect of grounding on the treewidth.

Guarded ASP Programs: Programs from this class guarantee that the treewidth of the program after grounding does not increase arbitrarily but only depends on the treewidth of the input facts.

Connection-Guarded ASP Programs: This class is more general than guarded ASP, but it only preserves bounded treewidth if, in addition, also the degree of the input is bounded.

This chapter defines these classes and contains the following contributions:

1. We prove that the grounding process has the claimed effect on the treewidth for these classes.
2. We discuss why the restrictions imposed by our classes cannot be dropped without generally “destroying” bounded treewidth by grounding.

3. We show that guarded programs are, despite their rather restrictive syntax, expressive enough to encode relevant problems from the second level of the polynomial hierarchy. Since every guarded program is connection-guarded, this clearly also holds for connection-guarded ASP.

This chapter is structured as follows: First, in Section 3.1, we formally define a notion of grounding that performs simplifications that can be assumed to be done by all state-of-the-art grounders. In Section 3.2, we express some basic properties of relational structures in MSO, which we will need for our transductions. Next, in Section 3.3, we define the class of guarded ASP programs and prove that grounding them preserves bounded treewidth of the input. This is followed in Section 3.4 by our definition of connection-guarded ASP and the proof that grounding such programs preserves bounded treewidth of the input if additionally the degree is bounded. In Section 3.5, we analyze the complexity of ASP solving in our classes. Finally, we discuss our results in Section 3.6 and investigate how they relate to existing research.

3.1 Program Simplifications in Grounding

The naive ground instantiation $\text{Ground}(\Pi)$ of a program Π, as defined in Section 2.4.3, is useful for the definition of the ASP semantics, but grounders in practice may omit large parts of $\text{Ground}(\Pi)$ in order to keep the grounding as small as possible while preserving equivalence to $\text{Ground}(\Pi)$. The techniques performed by state-of-the-art grounders are quite sophisticated and differ between systems, so we define a simplified notion of grounding for our study.

For a meaningful investigation of the relationship between the treewidth of input facts (as defined in Section 2.4.6 via fact structures) and the treewidth of the grounding (as defined in Section 2.4.5 via the primal graph), we need to assume that the grounder performs some basic simplifications. These simplifications are so basic that they can be assumed to be implemented by all reasonable grounders. The intuition is that a rule from the "naive" grounding is omitted in our grounding whenever its positive body contains an atom that cannot possibly be derived. Moreover, we also omit an aggregate element $t_1, \ldots, t_m : l_1, \ldots, l_n$ if some l_i cannot possibly be satisfied.

Definition 3.2. Let Π be an ASP program, let Π^+ denote the positive program obtained from Π by removing the negative bodies of all rules, removing all aggregate atoms and replacing disjunctions in the heads with conjunctions (that is, we replace a rule r whose head is $h_1 \vee \cdots \vee h_k$ by rules $r_1, \ldots, r_k$ such that $H(r_i) = \{h_i\}$ and the body of r_i is $B^+(r_i)$ without the aggregate atoms). We say that an atom is *possibly true in* Π if it is contained in the unique minimal model of Π^+.

```
take(X) ∨ leave(X) ← object(X).
    overburdened ← #sum{ W,X : take(X), weight(X,W) } > C, capacity(C).
             okay ← not overburdened.
                  ← not okay.
                  ⇜ take(X), value(X,V).   [−V,X]
capacity(3). object(a). object(b). object(c).
weight(a,1). weight(b,2). weight(c,3). value(a,5). value(b,6). value(c,9).
```

Listing 3.1: A knapsack problem encoding together with an instance

```
      take(X) ← object(X).
     leave(X) ← object(X).
overburdened ← capacity(C).
        okay.
capacity(3). object(a). object(b). object(c).
weight(a,1). weight(b,2). weight(c,3). value(a,5). value(b,6). value(c,9).
```

Listing 3.2: The positive program Π^+ obtained from the program Π from Listing 3.1

In the following definition of grounding, we now formalize the idea that reasonable grounders will not produce rules whose body is obviously false under every answer set.

Definition 3.3. Let Π be an ASP program. For any rule r in the ground instantiation $\text{Ground}(\Pi)$ as defined in Section 2.4.3, we write r^* to denote the rule obtained from r by removing each aggregate element $t_1, \ldots, t_m : l_1, \ldots, l_n$ that contains a positive atom l_i that is not possibly true. Now we define the *grounding* of Π, denoted by $\text{gr}(\Pi)$, as the set of all rules r^* such that r is a rule in $\text{Ground}(\Pi)$ and every predicate atom in $B^+(r)$ is possibly true.

Example 3.4. Recall the encoding of the knapsack problem from Example 2.28. The program Π shown in Listing 3.1 is a variation of that encoding together with input facts. This encoding is deliberately more complicated than necessary in order to illustrate what happens to negation and aggregates. We obtain the program Π^+, which is depicted in Figure 3.2, from Π as described in Definition 3.3.

The unique minimal model M^+ of Π^+ contains, for each object x, the atoms `object`(x), `take`(x), `leave`(x) as well as the atoms indicating its weight and value. Furthermore, M^+ contains the atoms `overburdened`, `okay` and `capacity(3)`.

This now allows us to construct the grounding $\text{gr}(\Pi)$, which is the program shown in Figure 3.3. Observe that, for instance, the rule involving the aggregate has only one ground instance in $\text{gr}(\Pi)$, namely the one where we instantiate the variable `C` with the actual capacity. In contrast, the complete, naive ground instantiation $\text{Ground}(\Pi)$ would

```
take(a) ∨ leave(a) ← object(a).
take(b) ∨ leave(b) ← object(b).
take(c) ∨ leave(c) ← object(c).
    overburdened ← #sum{ 1,a : take(a), weight(a,1);
                         2,b : take(b), weight(b,2);
                         3,c : take(c), weight(c,3) } > 3, capacity(3).
            okay ← not overburdened.
                 ← not okay.
                 ⇜ take(a), value(a,5).   [−5,a]
                 ⇜ take(b), value(b,6).   [−6,b]
                 ⇜ take(c), value(c,9).   [−9,c]
capacity(3). object(a). object(b). object(c).
weight(a,1). weight(b,2). weight(c,3). value(a,5). value(b,6). value(c,9).
```

Listing 3.3: The grounding of the program from Listing 3.1

contain as many instances of this rule as there are elements of the Herbrand universe – in this case nine. Also observe that the aggregate elements in every ground instance of that rule in Ground(Π) would contain literals like, e.g., take(1) or weight(3, a), which are not possibly true. △

3.2 Basic Properties of Relational Structures

In this section, we shall use MSO logic to express properties of arbitrary structures. In order to do this for any signature upon which the structures are based, we actually present *schemata* for producing MSO formulas in the following way: By a *signature-parameterized MSO formula* we mean an expression with a parameter σ, which can be instantiated by a relational signature subject to certain conditions, such that any permissible instantiation of the parameter yields an MSO formula over this particular signature. In our case, every incidence signature is a permissible instantiation of σ under the condition that the base signature contains a successor relation denoted by succ, which is used to specify an order on the domain elements. In this way, we can express properties of incidence structures of fact structures, as defined in Definition 2.34. The condition that the base signature contains a successor relation is reasonable because we express properties of ASP programs and ASP solvers automatically assume an order of the elements of the Herbrand universe (cf. Section 2.4).

We first define several signature-parameterized formulas that express properties of incidence structures in general (as long as the base structure contains a successor relation). These formulas will be used later, when we apply incidence structures for representing input facts of ASP programs. In the following, we write $\rho_{\max}$ to denote

$\rho(\text{Base}(\sigma))$, i.e., the maximum arity of a relation in the base signature.

We present our formulas in two groups. Both groups of formulas serve the same purpose: to represent certain tuples of domain elements as a combination of a single domain element with an element of a certain countable set.

The formulas in the first group, presented in Section 3.2.1, will allow us to extract from every fact $R(\boldsymbol{a})$ all tuples constituted of elements of $\boldsymbol{a}$. Subsequently, we can identify certain tuples $\langle a_1, \ldots, a_k \rangle$ of domain elements by a fact and a tuple of integers $\langle i_1, \ldots, i_k \rangle$ such that each a_j is the i_j-th argument of that fact.

The second group, presented in Section 3.2.2, concerns tuples $\langle a_1, \ldots, a_k \rangle$ of domain elements such that there is a domain element from which every a_i is reachable. Here we can identify such a tuple $\langle a_1, \ldots, a_k \rangle$ by a "source" element d together with a tuple of objects $\langle \pi_1, \ldots, \pi_k \rangle$, where each π_j determines a path from d to a_j. (By "reachable" and "path" we mean the straightforward generalizations of the graph-theoretic terms to structures.)

The motivation for representing tuples of domain elements in such a way is the following: The class of *guarded* ASP programs, which we will define in Section 3.3, will have the property that, for every atom $p(\boldsymbol{a})$ in the grounding, the tuple $\boldsymbol{a}$ can be identified by a single domain element of the input together with an element of a certain set S, which has constant size whenever the program is fixed, by virtue of the restrictions that guardedness imposes on the ASP language. For any fixed guarded program Π, given input $\mathcal{A}$, we can thus represent each atom $p(\boldsymbol{a})$ that occurs in $\text{gr}(\Pi \cup \mathcal{A})$ as one of *constantly many copies* of some element of $\text{dom}(\mathcal{A})$. Hence, for fixed guarded programs, the number of different atoms in the grounding can only be linear in the input. This is important because it allows us to formalize the grounding process as an MSO transduction, as transforming an input structure $\mathcal{I}$ into an output structure $\mathcal{O}$ is only possible via an MSO transduction if $|\text{dom}(\mathcal{O})|$ is linear in $|\text{dom}(\mathcal{I})|$ by Proposition 2.24.[1]

The input structures for our transductions will be incidence graphs of fact structures, as defined in Definition 2.34. Recall that fact structures contain a successor relation succ. In our MSO formulas, we will use the symbol $<$ as shorthand for the strict total order on the domain elements of the input structures given by the succ relation.[2]

[1]For the class of *connection-guarded* ASP programs, which we will define in Section 3.4, we will additionally require the degree of the input to be bounded in order to ensure linear grounding size. We will explain this in detail when we introduce the class.

[2]We can easily define $<$ from the succ relation because transitive closure can be defined in MSO (Courcelle and Engelfriet 2012, Section 1.3.1). Moreover, note that succ only gives us an ordering on the ASP constants but not on the facts, which are also present in the domain of the input structure (as it is an incidence structure). This is not a problem, however, because we can define a lexicographical order on tuples of constants based on the ordering on the individual constants.

3.2.1 Extracting Tuples From Facts

The following formulas will allow us to identify certain tuples $\langle a_1, \ldots, a_k \rangle$ of domain elements by a fact and a tuple of integers $\langle i_1, \ldots, i_k \rangle$ such that each a_j is the i_j-th argument of that fact.

We begin with the formula $\mathrm{fact}(x)$, which is true under a structure $\mathrm{Inc}(\mathcal{A})$ if x is a fact in the base structure $\mathcal{A}$. Recall that the domain of $\mathrm{Inc}(\mathcal{A})$ consists of the domain of $\mathcal{A}$ and the facts in $\mathcal{A}$, and that the relation symbols in the base signature $\mathrm{Base}(\sigma)$ are all unary in σ.

$$\mathrm{fact}(x) \;\equiv\; \bigvee_{R \in \mathrm{Base}(\sigma)} R(x)$$

The following formula expresses that x is a fact that contains y as some argument.

$$\mathrm{in}(x,y) \;\equiv\; \mathrm{in}_1(x,y) \vee \cdots \vee \mathrm{in}_{\rho_{\max}}(x,y)$$

If a fact in a base structure $\mathcal{A}$ contains every element of a tuple $\boldsymbol{a}$ of elements of $\mathrm{dom}(\mathcal{A})$, then we say that this fact *covers* $\boldsymbol{a}$. For each nonnegative integer k, we define the following formula, which states that x is a fact that covers $\langle y_1, \ldots, y_k \rangle$.

$$\mathrm{covers}_k(x, y_1, \ldots, y_k) \;\equiv\; \mathrm{fact}(x) \wedge \mathrm{in}(x, y_1) \wedge \cdots \wedge \mathrm{in}(x, y_k)$$

Furthermore, if some fact of a base structure $\mathcal{A}$ covers $\boldsymbol{a}$, then we define the *first cover* of $\boldsymbol{a}$ in $\mathcal{A}$ to be the smallest fact covering $\boldsymbol{a}$ according to the order of the domain elements, which we can construct from the successor relation that is guaranteed to be present in the input structure. For each nonnegative integer k, we therefore define a formula to express that x is the first cover of $\langle y_1, \ldots, y_k \rangle$:

$$\mathrm{fcov}_k(x, y_1, \ldots, y_k) \;\equiv\; \mathrm{covers}_k(x, y_1, \ldots, y_k) \wedge \neg\exists z\big(z < x \wedge \mathrm{covers}_k(z, y_1, \ldots, y_k)\big)$$

We say that a tuple $\langle i_1, \ldots, i_k \rangle$ of integers *extracts* a tuple $\langle a_1, \ldots, a_k \rangle$ of domain elements from a fact $R(d_1, \ldots, d_\ell)$ if both $1 \leqslant i_j \leqslant \ell$ and $a_j = d_{i_j}$ hold for every j. Clearly there is a tuple of integers that extracts a tuple $\boldsymbol{a}$ of domain elements from a fact x if and only if x covers $\boldsymbol{a}$. For each nonnegative integer k and every tuple $\langle i_1, \ldots, i_k \rangle$ of integers between 1 and $\rho_{\max}$, we define the following formula to express that $\langle i_1, \ldots, i_k \rangle$ extracts $\langle y_1, \ldots, y_k \rangle$ from x:

$$\mathrm{extract}_{\langle i_1, \ldots, i_k \rangle}(x, y_1, \ldots, y_k) \;\equiv\; \mathrm{in}_{i_1}(x, y_1) \wedge \cdots \wedge \mathrm{in}_{i_k}(x, y_k)$$

If a fact x covers $\boldsymbol{a}$, then we define the *first tuple* extracting $\boldsymbol{a}$ from x as the lexicographically smallest tuple of integers between 1 and $\rho_{\max}$ that extracts $\boldsymbol{a}$ from x. For this we first define the relation $\prec_{\rho_{\max}}$ among tuples of integers between 1 and $\rho_{\max}$ such that

$a \prec_{\rho_{\max}} b$ holds if a is lexicographically smaller than b. For every nonnegative integer k and every k-tuple $\boldsymbol{i}$ of integers between 1 and $\rho_{\max}$, we now define the following formula, which is true if and only if $\boldsymbol{i}$ is the first tuple extracting $\langle y_1, \dots, y_k \rangle$ from x:

$$\mathrm{fext}_{\boldsymbol{i}}(x, y_1, \dots, y_k) \;\equiv\; \mathrm{extract}_{\boldsymbol{i}}(x, y_1, \dots, y_k) \wedge \bigwedge_{\boldsymbol{j} \prec_{\rho_{\max}} \boldsymbol{i}} \neg\, \mathrm{extract}_{\boldsymbol{j}}(x, y_1, \dots, y_k)$$

If a fact in a base structure $\mathcal{A}$ covers a tuple $\boldsymbol{a}$ of domain elements, we define the *cover-based identifier* of $\boldsymbol{a}$ in $\mathcal{A}$ to be the unique combination of a fact x and a tuple $\boldsymbol{i}$ of integers such that x is the first cover of $\boldsymbol{a}$ in $\mathcal{A}$ and $\boldsymbol{i}$ is the first tuple extracting $\boldsymbol{a}$ from x. For each nonnegative integer k and every k-tuple $\boldsymbol{i}$ of integers between 1 and $\rho_{\max}$, we now define the following formula to express that x together with $\boldsymbol{i}$ is the cover-based identifier of $\langle y_1, \dots, y_k \rangle$:

$$\mathrm{cid}_{\boldsymbol{i}}(x, y_1, \dots, y_k) \;\equiv\; \mathrm{fcov}_k(x, y_1, \dots, y_k) \wedge \mathrm{fext}_{\boldsymbol{i}}(x, y_1, \dots, y_k)$$

Whenever a fact covers $\boldsymbol{a}$, the cover-based identifier of $\boldsymbol{a}$ clearly exists because then $\boldsymbol{a}$ has a first cover and $\boldsymbol{a}$ can be extracted from this cover. For every fact x and each tuple $\boldsymbol{i}$ of integers, the combination of x and $\boldsymbol{i}$ is the cover-based identifier of at most one tuple of domain elements. Hence there is a bijection between the set of all tuples that have a cover-based identifier and the set of all cover-based identifiers.

Example 3.5. Let $\mathcal{A}$ be a structure over a signature consisting of the binary relation symbols R and succ such that $\mathrm{dom}(\mathcal{A}) = \{a, b\}$, $R^{\mathcal{A}} = \{\langle a, a \rangle\}$ and $\mathrm{succ}^{\mathcal{A}} = \{\langle a, b \rangle\}$. We denote the domain elements of $\mathrm{Inc}(\mathcal{A})$ corresponding to the facts $\langle a, a \rangle$ in $R^{\mathcal{A}}$ and $\langle a, b \rangle$ in $\mathrm{succ}^{\mathcal{A}}$ by aa and ab, respectively. We assume that the ordering of the domain elements of $\mathrm{Inc}(\mathcal{A})$ that we extracted from $\mathrm{succ}^{\mathcal{A}}$ is $a < b < aa < ab$. The following formulas are true under $\mathrm{Inc}(\mathcal{A})$:

- $\mathrm{fact}(aa)$, $\mathrm{fact}(ab)$.
- $\mathrm{in}(aa, a)$, $\mathrm{in}(ab, a)$, $\mathrm{in}(ab, b)$.
- $\mathrm{covers}_1(aa, a)$, $\mathrm{covers}_2(aa, a, a)$, $\mathrm{covers}_3(aa, a, a, a)$, …

  $\mathrm{covers}_1(ab, a)$, $\mathrm{covers}_1(ab, b)$. $\mathrm{covers}_2(ab, a, a)$, $\mathrm{covers}_2(ab, a, b)$, $\mathrm{covers}_2(ab, b, a)$, $\mathrm{covers}_2(ab, b, b)$. $\mathrm{covers}_3(ab, a, a, a)$, …
- $\mathrm{fcov}_1(aa, a)$, $\mathrm{fcov}_2(aa, a, a)$, $\mathrm{fcov}_3(aa, a, a, a)$, …

  $\mathrm{fcov}_1(ab, b)$. $\mathrm{fcov}_2(ab, a, b)$, $\mathrm{fcov}_2(ab, b, a)$, $\mathrm{fcov}_2(ab, b, b)$, $\mathrm{fcov}_3(ab, a, a, b)$, …

  But note that, e.g., $\mathrm{fcov}_1(ab, a)$ is not true even though ab covers a, since a is also covered by aa, which is less than ab.

- $\mathrm{extract}_{\langle 1\rangle}(aa,a)$, $\mathrm{extract}_{\langle 2\rangle}(aa,a)$, $\mathrm{extract}_{\langle 1,1\rangle}(aa,a,a)$, $\mathrm{extract}_{\langle 1,2\rangle}(aa,a,a)$, $\mathrm{extract}_{\langle 2,1\rangle}(aa,a,a)$, $\mathrm{extract}_{\langle 2,2\rangle}(aa,a,a)$, $\mathrm{extract}_{\langle 1,1,1\rangle}(aa,a,a,a)$, ...

  $\mathrm{extract}_{\langle 1\rangle}(ab,a)$, $\mathrm{extract}_{\langle 2\rangle}(ab,b)$, $\mathrm{extract}_{\langle 1,1\rangle}(ab,a,a)$, $\mathrm{extract}_{\langle 1,2\rangle}(ab,a,b)$, $\mathrm{extract}_{\langle 2,1\rangle}(ab,b,a)$, $\mathrm{extract}_{\langle 2,2\rangle}(ab,b,b)$, $\mathrm{extract}_{\langle 1,1,1\rangle}(ab,a,a,a)$, ...

- $\mathrm{fext}_{\langle 1\rangle}(aa,a)$, $\mathrm{fext}_{\langle 1,1\rangle}(aa,a,a)$, $\mathrm{fext}_{\langle 1,1,1\rangle}(aa,a,a,a)$, ...

  $\mathrm{fext}_{\langle 1\rangle}(ab,a)$, $\mathrm{fext}_{\langle 2\rangle}(ab,b)$, $\mathrm{fext}_{\langle 1,1\rangle}(ab,a,a)$, $\mathrm{fext}_{\langle 1,2\rangle}(ab,a,b)$, $\mathrm{fext}_{\langle 2,1\rangle}(ab,b,a)$, $\mathrm{fext}_{\langle 2,2\rangle}(ab,b,b)$, $\mathrm{fext}_{\langle 1,1,1\rangle}(ab,a,a,a)$, ...

  But note that, e.g., $\mathrm{fext}_{\langle 2\rangle}(aa,a)$ is not true even though the tuple $\langle 2\rangle$ extracts the tuple $\langle a\rangle$ from aa, since we can also extract $\langle a\rangle$ from aa with the tuple $\langle 1\rangle$, which is lexicographically smaller than $\langle 2\rangle$.

- $\mathrm{cid}_{\langle 1,1\rangle}(aa,a,a)$, $\mathrm{cid}_{\langle 1,2\rangle}(ab,a,b)$, $\mathrm{cid}_{\langle 2,1\rangle}(ab,b,a)$, $\mathrm{cid}_{\langle 2,2\rangle}(ab,b,b)$, $\mathrm{cid}_{\langle 1,1,1\rangle}(aa,a,a,a)$, $\mathrm{cid}_{\langle 1,1,2\rangle}(ab,a,a,b)$, ...

  But note that, e.g., $\mathrm{cid}_{\langle 1,1\rangle}(ab,a,a)$ is not true since ab is not the first cover of $\langle a,a\rangle$. Also, $\mathrm{cid}_{\langle 2,2\rangle}(aa,a,a)$ is not true since $\langle 2,2\rangle$ is not the first tuple that extracts $\langle a,a\rangle$ from aa. △

3.2.2 Tuples Reachable From a Starting Element

We now present formulas that enable us to identify certain tuples $\langle a_1,\dots,a_k\rangle$ by a "source" element s together with a tuple of objects $\langle \pi_1,\dots,\pi_k\rangle$, where each π_j determines a path from s to a_j.

First we define a formula to express that x and y are neighbors in the sense that they are adjacent to each other in the Gaifman graph of the base structure.

$$\mathrm{neigh}(x,y) \;\equiv\; x \neq y \wedge \exists z\big(\,\mathrm{in}(z,x) \wedge \mathrm{in}(z,y)\big)$$

The order of the domain elements of a structure induces an order on the neighborhood of each domain element. For each positive integer i, the following formula expresses that y is the i-th neighbor of x according to this order.

$$\mathrm{neigh}_i(x,y) \;\equiv\; \mathrm{neigh}(x,y) \wedge \forall z\big(z < y \wedge \mathrm{neigh}(x,z) \rightarrow \bigvee_{1\leqslant j<i} \mathrm{neigh}_j(x,z)\big)$$

A *relative path* of length k is a k-tuple of positive integers. It is called a *relative d-path* if each integer in the tuple is less than or equal to d. Moreover, we say that a relative d-path is a *relative (ℓ,d)-path* if its length is at most ℓ. A *path* of length k between two domain elements x and y of $\mathcal{A}$ is a sequence of domain elements $a_0,a_1,\dots,a_k$ such that $a_0 = x$, $a_k = y$, and a_j is a neighbor of a_{j-1}, for $1 \leqslant j \leqslant k$. Each path can be uniquely identified by a relative path π together with a starting point $s \in \mathrm{dom}(\mathcal{A})$ in the obvious way, and we also write s^{π} to denote the end point of this path. For a tuple $\boldsymbol{\pi} = \langle \pi_1, \dots, \pi_k \rangle$ of relative paths, we write $s^{\boldsymbol{\pi}}$ to denote $\langle s^{\pi_1}, \dots, s^{\pi_k} \rangle$. We say that a path $\boldsymbol{p}$ is a *d-path* if its corresponding relative path is a relative d-path, and we call $\boldsymbol{p}$ an *(ℓ, d)-path* if it is a d-path of length at most ℓ.

For every integer k and each relative path $\pi = \langle i_1, \dots, i_k \rangle$, we define the formula $\mathrm{reach}_{\pi}(x, y)$ to express that x^{π} is defined and equal to y. We write ϵ to denote the relative path of length 0.

$$\begin{aligned} \mathrm{reach}_{\epsilon}(x,y) &\equiv x = y \\ \mathrm{reach}_{\langle i_1,\dots,i_k \rangle}(x,y) &\equiv \exists z \big(\mathrm{reach}_{\langle i_1,\dots,i_{k-1} \rangle}(x,z) \wedge \mathrm{neigh}_{i_k}(z,y) \big) \quad \text{for } k \geqslant 1 \end{aligned}$$

Since two domain elements can be connected via more than one path, we next describe how we can designate a single representative among them. We define the strict total order $\prec_d$ over relative d-paths such that $\pi \prec_d \psi$ if π is shorter than ψ or they have the same length but π is lexicographically smaller than ψ. If there is a d-path from a to b in $\mathcal{A}$, we define the *first relative d-path* from a to b as the smallest relative d-path π such that $a^{\pi} = b$, where "smallest" refers to $\prec_d$. For each nonnegative integer d and every relative d-path π, we now define the formula $\mathrm{frp}^{d}_{\pi}(x, y)$ to represent that π is the first relative d-path from x to y. (Note that the intended meaning of the abbreviation "frp" is "first relative path".)

$$\mathrm{frp}^{d}_{\pi}(x,y) \equiv \mathrm{reach}_{\pi}(x,y) \wedge \bigwedge_{\psi \prec_d \pi} \neg\, \mathrm{reach}_{\psi}(x,y)$$

Let ℓ, d and k be arbitrary nonnegative integers, and $\boldsymbol{a}$ be a tuple of domain elements such that there is a domain element from which there is an (ℓ, d)-path to each element of $\boldsymbol{a}$. We want to be able to identify $\boldsymbol{a}$ by a combination of a domain element s and a tuple $\boldsymbol{\pi}$ of relative (ℓ, d)-paths such that $s^{\boldsymbol{\pi}} = \boldsymbol{a}$. Of course, there may be multiple choices for s and $\boldsymbol{\pi}$ that identify $\boldsymbol{a}$ in such a way, so we want to single out a unique representative. To this end, we define the *(ℓ, d)-path-based identifier* of $\boldsymbol{a}$ as the unique combination of a domain element s and a tuple of relative (ℓ, d)-paths $\boldsymbol{\pi}$ that satisfies the following properties:

1. For every i, the i-th element of $\boldsymbol{\pi}$ is the first relative d-path from s to the i-th element of $\boldsymbol{a}$.
2. There is no domain element t smaller than s for which there is a tuple $\boldsymbol{\psi}$ of relative (ℓ, d)-paths such that $t^{\boldsymbol{\psi}} = \boldsymbol{a}$.

We now define the formula $\mathrm{pid}^{\ell,d}_{\langle \pi_1,\dots,\pi_k \rangle}(x, y_1, \dots, y_k)$ to express that x together with $\langle \pi_1, \dots, \pi_k \rangle$ is the (ℓ, d)-path-based identifier of $\langle y_1, \dots, y_k \rangle$. For this, let $P^{k}_{\ell,d}$ denote the set of all k-tuples of relative (ℓ, d)-paths. (Note that this set is finite.)

$$\mathrm{pid}^{\ell,d}_{\langle \pi_1,\ldots,\pi_k\rangle}(x, y_1, \ldots, y_k) \equiv \bigwedge_{1\leqslant i\leqslant k} \mathrm{frp}^{d}_{\pi_i}(x, y_i) \wedge \neg\exists z \left(z < x \wedge \bigvee_{\langle \psi_1,\ldots,\psi_k\rangle \in P^k_{\ell,d}} \bigwedge_{1\leqslant i\leqslant k} \mathrm{frp}^{d}_{\psi_i}(z, y_i)\right)$$

Note that for the empty tuple ϵ, the formula $\mathrm{pid}^{\ell,d}_{\epsilon}(x)$ becomes $\neg\exists z\,(z < x)$, which is true if and only if x is the smallest domain element.

Any tuple $\boldsymbol{a}$ of domain elements has an (ℓ, d)-path-based identifier whenever there is a domain element s and a tuple of relative (ℓ, d)-paths $\boldsymbol{\pi}$ such that $\boldsymbol{a} = s^{\boldsymbol{\pi}}$: Then for each $a \in \boldsymbol{a}$, there is an (ℓ, d)-path from s to a, so there is also a first relative (ℓ, d)-path from s to a. Moreover, for every domain element s and each tuple $\boldsymbol{\pi}$ of relative (ℓ, d)-paths, the combination of s and $\boldsymbol{\pi}$ is the (ℓ, d)-path-based identifier of at most one tuple of domain elements. Hence there is a bijection between the set of all tuples that have an (ℓ, d)-path-based identifier and the set of all (ℓ, d)-path-based identifiers.

Example 3.6. Let $\mathcal{A}$ be a structure over a signature consisting of the binary relation symbols R and succ such that $\mathrm{dom}(\mathcal{A}) = \{a, b, c, d\}$, $R^{\mathcal{A}} = \{\langle a, b\rangle, \langle a, d\rangle, \langle b, c\rangle, \langle d, c\rangle\}$ and $\mathrm{succ}^{\mathcal{A}} = \{\langle a, b\rangle, \langle b, c\rangle, \langle c, d\rangle\}$. We assume that $a < b < c < d$ holds for the ordering of the domain elements of $\mathrm{Inc}(\mathcal{A})$, and that the domain elements corresponding to facts are all greater than d – their exact order is irrelevant for this example. The following formulas are true under $\mathrm{Inc}(\mathcal{A})$:

- $\mathrm{neigh}(a, b)$, $\mathrm{neigh}(a, d)$, $\mathrm{neigh}(b, c)$, $\mathrm{neigh}(d, c)$.

  $\mathrm{neigh}(b, a)$, $\mathrm{neigh}(d, a)$, $\mathrm{neigh}(c, b)$, $\mathrm{neigh}(c, d)$.

- $\mathrm{neigh}_1(a, b)$, $\mathrm{neigh}_2(a, d)$, $\mathrm{neigh}_1(b, a)$, $\mathrm{neigh}_2(b, c)$, $\mathrm{neigh}_1(c, b)$, $\mathrm{neigh}_2(c, d)$.
  $\mathrm{neigh}_1(d, a)$, $\mathrm{neigh}_2(d, c)$

- $\mathrm{reach}_\epsilon(a, a)$, $\mathrm{reach}_{\langle 1\rangle}(a, b)$, $\mathrm{reach}_{\langle 2\rangle}(a, d)$, $\mathrm{reach}_{\langle 1,1\rangle}(a, a)$, $\mathrm{reach}_{\langle 1,2\rangle}(a, c)$,
  $\mathrm{reach}_{\langle 2,1\rangle}(a, a)$, $\mathrm{reach}_{\langle 2,2\rangle}(a, c)$, $\mathrm{reach}_{\langle 1,1,1\rangle}(a, b)$, $\ldots$

  $\mathrm{reach}_\epsilon(b, b)$, $\mathrm{reach}_{\langle 1\rangle}(b, a)$, $\mathrm{reach}_{\langle 2\rangle}(b, c)$, $\mathrm{reach}_{\langle 1,1\rangle}(b, b)$, $\ldots$

  $\mathrm{reach}_\epsilon(c, c)$, $\mathrm{reach}_{\langle 1\rangle}(c, b)$, $\mathrm{reach}_{\langle 2\rangle}(c, d)$, $\mathrm{reach}_{\langle 1,1\rangle}(c, a)$, $\ldots$

  $\mathrm{reach}_\epsilon(d, d)$, $\mathrm{reach}_{\langle 1\rangle}(d, a)$, $\mathrm{reach}_{\langle 2\rangle}(d, c)$, $\mathrm{reach}_{\langle 1,1\rangle}(d, b)$, $\ldots$

- $\mathrm{frp}^1_\epsilon(a, a)$, $\mathrm{frp}^1_{\langle 1\rangle}(a, b)$.

  But note that, e.g., $\mathrm{frp}^1_{\langle 1,1\rangle}(a, a)$ is not true even though $\langle 1, 1\rangle$ is a relative 1-path and it leads to a when starting from a, since so does ϵ, which is lexicographically smaller than $\langle 1, 1\rangle$. Moreover, neither $\mathrm{frp}^1_\pi(a, c)$ nor $\mathrm{frp}^1_\pi(a, d)$ are true for any relative 1-path π, since going from a to c or d requires visiting the "second neighbor" of at least one element.

$\mathrm{frp}^1_{\langle 1\rangle}(b,a)$, $\mathrm{frp}^1_{\epsilon}(b,b)$.

$\mathrm{frp}^1_{\langle 1,1\rangle}(c,a)$, $\mathrm{frp}^1_{\langle 1\rangle}(c,b)$, $\mathrm{frp}^1_{\epsilon}(c,c)$.

$\mathrm{frp}^1_{\langle 1\rangle}(d,a)$, $\mathrm{frp}^1_{\epsilon}(d,d)$.

$\mathrm{frp}^2_{\epsilon}(a,a)$, $\mathrm{frp}^2_{\langle 1\rangle}(a,b)$, $\mathrm{frp}^2_{\langle 2\rangle}(a,d)$, $\mathrm{frp}^2_{\langle 1,2\rangle}(a,c)$.

But note that, e.g., $\mathrm{frp}^2_{\langle 2,2\rangle}(a,c)$ is not true even though $\langle 2,2\rangle$ is a relative 2-path and it leads to c when starting from a, since so does $\langle 1,2\rangle$, which is lexicographically smaller than $\langle 2,2\rangle$.

...

- We only illustrate the formulas concerning (ℓ,d)-path-based identifiers for $\ell = d = 1$.

  $\mathrm{pid}^{1,1}_{\epsilon}(a)$.

  $\mathrm{pid}^{1,1}_{\langle\epsilon\rangle}(a,a)$, $\mathrm{pid}^{1,1}_{\langle\epsilon\rangle}(b,b)$, $\mathrm{pid}^{1,1}_{\langle\epsilon\rangle}(c,c)$, $\mathrm{pid}^{1,1}_{\langle\epsilon\rangle}(d,d)$.

  $\mathrm{pid}^{1,1}_{\langle\epsilon,\epsilon\rangle}(a,a,a)$, $\mathrm{pid}^{1,1}_{\langle\epsilon,\langle 1\rangle\rangle}(a,a,b)$, $\mathrm{pid}^{1,1}_{\langle\langle 1\rangle,\epsilon\rangle}(d,a,d)$.

  ...

  Even though $b^{\langle\langle 1\rangle,\epsilon\rangle} = \langle a,b\rangle$, the $(1,1)$-path-based identifier of $\langle a,b\rangle$ is not constituted by b and $\langle\langle 1\rangle,\epsilon\rangle$ because $a < b$ and there is a tuple $\boldsymbol{\psi}$ of relative $(1,1)$-paths such that $a^{\boldsymbol{\psi}} = \langle a,b\rangle$, namely $\boldsymbol{\psi} = \langle\epsilon,\langle 1\rangle\rangle$. Moreover, $\langle a,c\rangle$ has no $(1,1)$-path-based identifier because the distance between a and c is 2 and for all elements that are within distance 1 of both a and c (namely b and d), there is no path leading to c that only visits the first neighbor.

  This already suggests a property that we will exploit later: A tuple $\boldsymbol{a}$ of domain elements of $\mathcal{A}$ is guaranteed to have an (ℓ,d)-path-based identifier if the distance between any two elements of $\boldsymbol{a}$ is at most ℓ and the degree of $\mathcal{A}$ is at most d. △

3.3 Guarded Answer Set Programs

In this section, we define the class of guarded ASP programs and show that they lead to groundings whose treewidth depends only on the treewidth of the input structure.

Definition 3.7. Let Π be an ASP program. A *guard* of a rule r in Π is an extensional predicate atom in the positive body of r that contains every variable occurring in r. We call Π *guarded* if every rule has a guard. We designate an arbitrary guard of r as *the guard* of r.

We next show an important property for guarded ASP programs without constants: For each possible input and every tuple $\boldsymbol{a}$ of constants that can occur in an atom of the grounding, $\boldsymbol{a}$ has a cover-based identifier.

$$\begin{tikzcd}
\Pi \cup \mathcal{A} \arrow[d] & \mathrm{Inc}(\mathcal{A}) \arrow[d] & \text{treewidth of } \mathcal{A} \text{ bounded} \arrow[dd, Rightarrow, "\text{Theorem 2.25}"] \\
\boxed{\text{Grounder}} \arrow[d] & \boxed{\text{Transduction } \gamma_\Pi} \arrow[d] & \\
\mathrm{gr}(\Pi \cup \mathcal{A}) \arrow[r, phantom, "\widehat{=}"] & \mathrm{Inc}(\mathcal{A}') & \text{treewidth of } \mathcal{A}' \text{ bounded}
\end{tikzcd}$$

Figure 3.1: Strategy for proving that grounding a fixed guarded program Π together with an input structure $\mathcal{A}$ preserves bounded treewidth of $\mathcal{A}$

Proposition 3.8. *If Π is a guarded ASP program without constants, $\mathcal{A}$ is an input structure of Π and $p(\boldsymbol{a})$ is a predicate atom that occurs in $\mathrm{gr}(\Pi \cup \mathcal{A})$, then the tuple $\boldsymbol{a}$ has a cover-based identifier in $\mathcal{A}$.*

Proof. By our observations in Section 3.2, we have seen that $\boldsymbol{a}$ has a cover-based identifier in $\mathcal{A}$ if a fact covers $\boldsymbol{a}$. Since the atom $p(\boldsymbol{a})$ is part of the grounding, it occurs in one of its rules. Let r be a rule in the grounding such that $p(\boldsymbol{a})$ occurs in r, and let r' denote the corresponding non-ground rule. The atom $p(\boldsymbol{a})$ occurs in r as an instantiation of a non-ground atom $p(\boldsymbol{X})$ in r'. Moreover, the positive body of r contains an atom $g(\boldsymbol{b})$ as an instantiation of the guard $g(\boldsymbol{Y})$ of r'. Since $g(\boldsymbol{Y})$ is the guard of r', each variable in $\boldsymbol{X}$ also occurs in $\boldsymbol{Y}$. Hence each element of $\boldsymbol{a}$ is also an element of $\boldsymbol{b}$. Since the fact $g(\boldsymbol{b})$ thus covers $\boldsymbol{a}$, the latter has a cover-based identifier in $\mathcal{A}$. □

We can also prove the related result that for all tuples $\boldsymbol{a}$ and $\boldsymbol{b}$ of constants that can occur in two atoms of the same rule in the grounding, the joint tuple $\boldsymbol{ab}$ has a cover-based identifier.

Proposition 3.9. *If Π is a guarded ASP program without constants, $\mathcal{A}$ is an input structure of Π and both $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ are predicate atoms that occur together in a rule of $\mathrm{gr}(\Pi \cup \mathcal{A})$, then the joint tuple $\boldsymbol{ab}$ has a cover-based identifier in $\mathcal{A}$.*

Proof. Since $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ occur together in a rule r of the grounding, the fact that instantiates the guard in r covers both $\boldsymbol{a}$ and $\boldsymbol{b}$, and therefore it also covers $\boldsymbol{ab}$. □

We now state the main result of this section.

Theorem 3.10. *Let Π be a fixed guarded ASP program, and $\mathcal{A}$ be an input structure of Π. If $\mathcal{A}$ has bounded treewidth, then the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$ also has bounded treewidth.*

Proof. Our proof strategy is illustrated in Figure 3.1. In order to formally represent the grounding process and investigate its influence on the treewidth, we use MSO transductions as defined in Section 2.3.3. Our methodology for proving that grounding $\Pi \cup \mathcal{A}$ preserves bounded treewidth of $\mathcal{A}$ is the following: We study the transformation of $\mathcal{A}$ into the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$. We do this by providing an MSO transduction γ_Π that transforms $\mathrm{Inc}(\mathcal{A})$ into the incidence structure of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$. By constructing the transduction in such a way that it only depends on Π (and is thus fixed as Π is fixed), we obtain the desired result by virtue of Theorem 2.25.

We will, for the moment, assume that Π is constant-free; we will show later how to handle the general case. Note that the primal graph is undirected, but we assume that for representing undirected graphs as structures we use symmetric edge relations, so we treat an undirected edge as two directed edges of opposing orientation. Thus, when we speak of the primal graph in this proof, we refer to a directed graph with a symmetric edge relation. The signature of the output structure of our transduction is thus $\{\mathrm{E}, \mathrm{in}_1, \mathrm{in}_2\}$, where E is unary and the other relations are binary.

We call the domain elements in the output structure that correspond to vertices and edges of the primal graph *vertex elements* and *edge elements*, respectively. In the following, we define γ_Π by the definition scheme$\langle \Delta, \Theta \rangle$, where the tuple Δ is the concatenation of tuples Δ_v and Δ_e, which contain domain formulas that generate the vertex and edge elements, respectively, and Θ contains relation formulas that state which vertex is incident to which edge.

Formulas in Δ_v. These formulas shall produce the vertex elements. For each predicate p of arity k, we first define O_p to be the following set of objects: If a rule r in Π is guarded by an atom $g(X_1, \dots, X_\ell)$ and contains an atom $p(X_{i_1}, \dots, X_{i_k})$, then O_p contains $\langle g, i_1, \dots, i_k \rangle$. With this, we define a formula $\mathrm{occurs}_p(\boldsymbol{x})$, where $\boldsymbol{x}$ is a k-ary tuple of variables, to express that the ground atom $p(\boldsymbol{x})$ occurs in $\mathrm{gr}(\Pi \cup \mathcal{A})$.[3]

$$\mathrm{occurs}_p(\boldsymbol{x}) \equiv \bigvee_{\langle g, i_1, \dots, i_k \rangle \in O_p} \exists \boldsymbol{y} \big(g(\boldsymbol{y}) \land \mathrm{extract}_{\langle i_1, \dots, i_k \rangle}(\boldsymbol{y}, \boldsymbol{x}) \big)$$

We now put this auxiliary formula to use: We define the following formula $\delta_{p[\boldsymbol{i}]}(x)$ to be an element of Δ_v, for every predicate p of arity k and each k-ary tuple $\boldsymbol{i}$ of integers between 1 and $\rho_{\max}$.

$$\delta_{p[\boldsymbol{i}]}(x) \equiv \exists \boldsymbol{y} \big(\mathrm{cid}_{\boldsymbol{i}}(x, \boldsymbol{y}) \land \mathrm{occurs}_p(\boldsymbol{y}) \big)$$

[3]We slightly abuse notation by sometimes using tuples where actually lists are required. Each tuple, when used like this, stands for the comma-separated list of its elements without the tuple delimiters $\langle$ and $\rangle$. Thus, for any formula φ with free variables $\boldsymbol{x} = \langle x_1, \dots, x_k \rangle$, we write $\varphi(\boldsymbol{x})$ as an abbreviation for $\varphi(x_1, \dots, x_k)$. Similarly, for tuples $\boldsymbol{x} = \langle x_1, \dots, x_k \rangle$ and $\boldsymbol{y} = \langle y_1, \dots, y_\ell \rangle$ and any formula φ whose free variables we may denote by $\langle x_1, \dots, x_k, y_1, \dots, y_\ell \rangle$, we write $\varphi(\boldsymbol{x}, \boldsymbol{y})$ as an abbreviation for $\varphi(x_1, \dots, x_k, y_1, \dots, y_\ell)$. Moreover, when we use a tuple after a quantifier, we mean that each element of the tuple is quantified as specified; for instance, $\exists \boldsymbol{x}$ denotes $\exists x_1 \cdots \exists x_k$ for any tuple $\boldsymbol{x} = \langle x_1, \dots, x_k \rangle$. We may use a tuple $\boldsymbol{x}$ in a formula without specifying its arity explicitly if the required arity is clear from the context (i.e., from the other occurrences of $\boldsymbol{x}$ in the formula).

This formula is true if and only if x together with $\boldsymbol{i}$ is the cover-based identifier of some tuple $\boldsymbol{a}$ of domain elements and $p(\boldsymbol{a})$ occurs in $\mathrm{gr}(\Pi \cup \mathcal{A})$; the resulting copy of x in the output structure then corresponds to the atom $p(\boldsymbol{a})$.

For each predicate atom $p(\boldsymbol{a})$ in the grounding, we thus produce a vertex element, since $\boldsymbol{a}$ has a cover-based identifier by Proposition 3.8. Moreover, different predicate atoms produce different vertex elements: If they have different arguments, they have different cover-based identifiers, and otherwise they differ in their predicate symbol. In both cases, they clearly produce different copies. Finally, every vertex element that we produce corresponds to an atom in the grounding by our construction of the occurs_p formulas. This proves that there is a bijection between the atoms in the grounding and the vertex elements.

Formulas in Δ_e. These formulas shall produce the edge elements. Before we define them, we introduce an auxiliary formula. For all predicates p and q of arity k and ℓ, respectively, we define $T_{p,q}$ to be the following set of objects: If there is a rule r in Π guarded by an atom $g(X_1, \dots, X_m)$ such that r contains two atoms $p(X_{i_1}, \dots, X_{i_k})$ and $q(X_{j_1}, \dots, X_{j_\ell})$, then $T_{p,q}$ contains $\langle g, \boldsymbol{i}, \boldsymbol{j} \rangle$, where $\boldsymbol{i} = \langle i_1, \dots, i_k \rangle$ and $\boldsymbol{j} = \langle j_1, \dots, j_\ell \rangle$. With this, we now define a formula $\mathrm{together}_{p,q}(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are tuples of variables with arity k and ℓ, respectively. This formula expresses that the two ground atoms $p(\boldsymbol{x})$ and $q(\boldsymbol{y})$ occur together in some rule of $\mathrm{gr}(\Pi \cup \mathcal{A})$.

$$\mathrm{together}_{p,q}(\boldsymbol{x}, \boldsymbol{y}) \equiv \bigvee_{\langle g, \boldsymbol{i}, \boldsymbol{j} \rangle \in T_{p,q}} \exists z \big(g(z) \land \mathrm{extract}_{\boldsymbol{i}}(z, \boldsymbol{x}) \land \mathrm{extract}_{\boldsymbol{j}}(z, \boldsymbol{y}) \big)$$

With this auxiliary formula in hand, we define the following formula $\delta_{p[\boldsymbol{i}]q[\boldsymbol{j}]}(x)$ to be an element of Δ_e, for all predicates p and q, and all tuples $\boldsymbol{i}$ and $\boldsymbol{j}$ of integers between 1 and $\rho_{\max}$, such that the arities of $\boldsymbol{i}$ and $\boldsymbol{j}$ are the same as those of p and q, respectively.

$$\delta_{p[\boldsymbol{i}]q[\boldsymbol{j}]}(x) \equiv \begin{cases} \bot & \text{if } p[\boldsymbol{i}] = q[\boldsymbol{j}] \\ \exists \boldsymbol{y} \exists \boldsymbol{z} \big(\mathrm{cid}_{\boldsymbol{ij}}(x, \boldsymbol{y}, \boldsymbol{z}) \land \mathrm{together}_{p,q}(\boldsymbol{y}, \boldsymbol{z}) \big) & \text{otherwise} \end{cases}$$

This formula is true if and only if there are tuples $\boldsymbol{a}$ and $\boldsymbol{b}$ of the same arity as p and q, respectively, such that (1) x together with $\boldsymbol{ij}$ is the cover-based identifier of $\boldsymbol{ab}$, and (2) the atoms $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ are different and occur together in some rule of $\mathrm{gr}(\Pi \cup \mathcal{A})$. The resulting copy of x in the output structure then corresponds to the edge from $p(\boldsymbol{a})$ to $q(\boldsymbol{b})$ in the primal graph. Due to symmetry, we can see that then also an edge in the other direction will be created. Since both atoms are different if the formula is true, we do not introduce loops.

For each pair of different predicate atoms $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ that jointly occur in a rule of the grounding, we thus correctly produce two edge elements, since $\boldsymbol{ab}$ has a cover-based identifier by Proposition 3.9. Moreover, different such pairs of atoms produce different edge elements, and every edge element that we produce corresponds to a joint occurrence of two different atoms in a rule of the grounding by our construction of the $\mathrm{together}_{p,q}$ formulas.

Formulas in Θ. These formulas shall ensure that each edge element is incident to the two appropriate vertex elements. First, let p and q be predicates occurring in Π, and let $\boldsymbol{i}$ and $\boldsymbol{j}$ be tuples of integers between 1 and $\rho_{\max}$ such that $\boldsymbol{i}$ and $\boldsymbol{j}$ have the same arity as p and q, respectively. We define a formula $\mathrm{eq}_{p[\boldsymbol{i}],q[\boldsymbol{j}]}(x,y)$ to express that the atoms $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ are equal, where $\boldsymbol{a}$ is the tuple extracted from x by $\boldsymbol{i}$, and $\boldsymbol{b}$ is the tuple extracted from y by $\boldsymbol{b}$.

$$\mathrm{eq}_{p[\boldsymbol{i}],q[\boldsymbol{j}]}(x,y) \equiv \begin{cases} \exists \boldsymbol{z}\big(\mathrm{extract}_{\boldsymbol{i}}(x,\boldsymbol{z}) \wedge \mathrm{extract}_{\boldsymbol{j}}(y,\boldsymbol{z})\big) & \text{if } p = q \\ \bot & \text{otherwise} \end{cases}$$

Let p, q and q' be predicates occurring in Π, and let $\boldsymbol{i}$, $\boldsymbol{j}$ and $\boldsymbol{j}'$ be tuples of integers between 1 and $\rho_{\max}$ with the same arity as p, q and q', respectively. We define the following formulas to be elements of Θ:[4]

$$\begin{aligned} \vartheta_{\mathrm{in}_1,\,p[\boldsymbol{i}],\,q[\boldsymbol{j}]q'[\boldsymbol{j}']}(x,y) &\equiv \mathrm{eq}_{p[\boldsymbol{i}],q[\boldsymbol{j}]}(x,y) \\ \vartheta_{\mathrm{in}_2,\,p[\boldsymbol{i}],\,q[\boldsymbol{j}]q'[\boldsymbol{j}']}(x,y) &\equiv \mathrm{eq}_{p[\boldsymbol{i}],q'[\boldsymbol{j}']}(x,y) \end{aligned}$$

We only explain the first of these formulas, as the other case is symmetric; instead of outgoing edges (due to in_1 in the subscript) it concerns incoming edges (due to in_2 in the subscript).

The formula $\vartheta_{\mathrm{in}_1,\,p[\boldsymbol{i}],\,q[\boldsymbol{j}]q'[\boldsymbol{j}']}(x,y)$ is true if and only if the atom $p(\boldsymbol{a})$ is equal to $q(\boldsymbol{b})$, where $\boldsymbol{a}$ is the tuple extracted from x by $\boldsymbol{i}$, and $\boldsymbol{b}$ is the tuple extracted from y by $\boldsymbol{j}$. If this formula is true, it makes the edge represented by the respective copy of y an *outgoing* edge of $p(\boldsymbol{a})$ because of the subscript in_1.

We first show that, whenever this formula causes an edge element to be incident to a vertex element, the corresponding edge in the primal graph is indeed an outgoing edge of the appropriate vertex. Suppose there are predicates p, q and p', tuples $\boldsymbol{i}$, $\boldsymbol{j}$ and $\boldsymbol{j}'$, as well as domain elements x and y such that (1) $\delta_{p[\boldsymbol{i}]}(x)$ is true, (2) $\delta_{q[\boldsymbol{j}]q'[\boldsymbol{j}']}(y)$ is true, and (3) $\vartheta_{\mathrm{in}_1,\,p[\boldsymbol{i}],\,q[\boldsymbol{j}]q'[\boldsymbol{j}']}(x,y)$ is true. As observed in our definition of Δ_v, (1) means that x together with $\boldsymbol{i}$ is the cover-based identifier of some tuple $\boldsymbol{a}$, and the grounding contains an atom $p(\boldsymbol{a})$. From (2) we get that there are tuples $\boldsymbol{b}$ and $\boldsymbol{b}'$ such that y together with $\boldsymbol{j}\boldsymbol{j}'$ is the cover-based identifier of $\boldsymbol{b}\boldsymbol{b}'$. We have also seen that by (2) there is a rule in the grounding that contains both $q(\boldsymbol{b})$ and $q'(\boldsymbol{b}')$, hence there is an edge from $q(\boldsymbol{b})$ to $q'(\boldsymbol{b}')$ in the primal graph. By (3) and the definition of cover-based identifiers (in particular

[4]If we do not explicitly mention relation formulas like the remaining relation formulas for defining in_1 and in_2 (those having subscripts of different forms than the shown formulas), then these are defined as $\bot$.

those of $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{b}'$), we know that $p(\boldsymbol{a})$ is equal to $q(\boldsymbol{b})$. Hence the edge in the primal graph from $q(\boldsymbol{b})$ to $q'(\boldsymbol{b}')$ is indeed an outgoing edge of $p(\boldsymbol{a})$.

Finally we prove the other direction: Whenever an edge in the primal graph is an outgoing edge of a vertex, a formula in Θ defining the relation in_1 causes the corresponding edge element to be an outgoing edge of the appropriate vertex element. Suppose that the primal graph contains an edge from atom $p(\boldsymbol{a})$ to atom $q(\boldsymbol{b})$. Then these atoms occur together in a rule of the grounding, so there is a non-ground rule r guarded by an atom $g(X_1, \dots, X_m)$ such that r contains both $p(X_{i_1}, \dots, X_{i_k})$ and $q(X_{j_1}, \dots, X_{j_\ell})$, and $\mathcal{A}$ contains a fact $g(c_1, \dots, c_m)$ such that $\boldsymbol{a} = \langle c_{i_1}, \dots, c_{i_k} \rangle$ and $\boldsymbol{b} = \langle c_{j_1}, \dots, c_{j_\ell} \rangle$. Moreover, we have seen in our definition of Δ_v and Δ_e that then our transduction produces a vertex element v for $p(\boldsymbol{a})$ and an edge element e for the edge from $p(\boldsymbol{a})$ to $q(\boldsymbol{b})$. Now let x and y be domain elements of $\mathcal{A}$, and let $\boldsymbol{i}$, $\boldsymbol{j}$ and $\boldsymbol{j}'$ be tuples of integers, such that x together with $\boldsymbol{i}$ is the cover-based identifier of $\boldsymbol{a}$, and y together with $\boldsymbol{j}\boldsymbol{j}'$ is the cover-based identifier of $\boldsymbol{a}\boldsymbol{b}$. By definition of cover-based identifiers, $\boldsymbol{i}$ extracts $\boldsymbol{a}$ from x; moreover, $\boldsymbol{j}$ and $\boldsymbol{j}'$ extract $\boldsymbol{a}$ and $\boldsymbol{b}$ from y, respectively. Since $\boldsymbol{a}$ can be extracted from x by $\boldsymbol{i}$, as well as from y by $\boldsymbol{j}$, the formula $\mathrm{eq}_{p[\boldsymbol{i}],p[\boldsymbol{j}]}(x, y)$ is clearly true. Hence the formula $\vartheta_{\mathrm{in}_1,\, p[\boldsymbol{i}],\, p[\boldsymbol{j}]q[\boldsymbol{j}']}(x, y)$ is true, which correctly makes the edge element e an outgoing edge of the vertex element v.

We are still missing the relation formulas for defining the remaining unary relation E, which identifies the edge elements. This is easy: We can just set $\vartheta_{\mathrm{E},\, p[\boldsymbol{i}]}$ to $\bot$ (for all $p[\boldsymbol{i}]$ as before) and $\vartheta_{\mathrm{E},\, p[\boldsymbol{i}]q[\boldsymbol{j}]}$ to $\top$.

This completes the construction of the MSO transduction γ_Π. Let $\mathcal{A}$ be an input structure for Π of bounded treewidth. We have argued that $\gamma_\Pi(\mathrm{Inc}(\mathcal{A}))$ yields the incidence structure of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$ as desired. By Theorem 2.25 and the fact that $\mathcal{A}$, by assumption, has bounded treewidth, this proves our claim for constant-free programs.

It is tedious, but straightforward, to generalize this proof to programs with constants, and we will only give the general idea here. For each constant occurring in Π, we add a constant symbol to our MSO signature. Then we adapt our definitions of the MSO formulas in the following ways: (1) We need to change the notion of "covering" such that a fact x covers a tuple $\boldsymbol{a}$ of domain elements if each element $\boldsymbol{a}$ occurs in x or is equal to any constant. (2) Whenever we use a tuple of integers, any element of such a tuple can also be a constant symbol. For instance, we need to define the formula $\mathrm{extract}_{\boldsymbol{i}}(x, \boldsymbol{y})$ not only for each tuple of integers between 1 and $\rho_{\max}$, but for each tuple $\boldsymbol{i}$ where every element of $\boldsymbol{i}$ is either such an integer or a constant symbol. (3) We need to change the definition of "extracting" such that constants can always be extracted from any fact; that is, we remove from the formula $\mathrm{extract}_{\langle i_1, \dots, i_j \rangle}(x, y_1, \dots, y_k)$ all conjuncts $\mathrm{in}_{i_j}(x, y_j)$ where i_j is a constant. It is easy to generalize Propositions 3.8 and 3.9 to programs with constants, and to verify that the resulting transduction is correct. □

We now illustrate this transduction for an example program.

Example 3.11. Let Π be the following guarded program for solving the 2-Colorability problem on directed graphs, where the input graph is given via the binary edge predicate $\mathtt{e}$, and the colors are $\mathtt{r}$ and $\mathtt{g}$:

$$\begin{aligned}
&\mathtt{r(X)} \vee \mathtt{g(X)} \leftarrow \mathtt{e(X,Y)}.\\
&\mathtt{r(Y)} \vee \mathtt{g(Y)} \leftarrow \mathtt{e(X,Y)}.\\
&\qquad\qquad\quad \leftarrow \mathtt{e(X,Y),\ r(X),\ r(Y)}.\\
&\qquad\qquad\quad \leftarrow \mathtt{e(X,Y),\ g(X),\ g(Y)}.
\end{aligned}$$

Next, let $\mathcal{G}$ be the input structure for Π that represents a graph consisting of vertices a and b, with an edge from b to a. Moreover, we assume that the order on the domain elements is such that $a < b$. That is, $\mathrm{dom}(\mathcal{G}) = \{a, b\}$, $\mathrm{succ}^{\mathcal{G}} = \{\langle a, b\rangle\}$ and $\mathtt{e}^{\mathcal{G}} = \{\langle b, a\rangle\}$. We denote the domain element of $\mathrm{Inc}(\mathcal{G})$ for the fact $\langle b, a\rangle$ in $\mathtt{e}^{\mathcal{G}}$ and for $\langle a, b\rangle$ in $\mathrm{succ}^{\mathcal{G}}$ by ba and ab, respectively, and we assume the ordering $a < b < ab < ba$. We show how γ_Π transforms $\mathrm{Inc}(\mathcal{G})$ into the incidence structure of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{G})$.

For the vertex elements, first observe that all of the formulas $\mathrm{occurs}_{\mathtt{e}}(b, a)$, $\mathrm{occurs}_{\mathtt{r}}(a)$, $\mathrm{occurs}_{\mathtt{r}}(b)$, $\mathrm{occurs}_{\mathtt{g}}(a)$ and $\mathrm{occurs}_{\mathtt{g}}(b)$ are true. For instance, to see that $\mathrm{occurs}_{\mathtt{e}}(b, a)$ is true, observe that $O_{\mathtt{e}} = \{\langle \mathtt{e}, 1, 2\rangle\}$ (under the assumption that the first variable of each rule is $\mathtt{X}$ and the second is $\mathtt{Y}$); now clearly the subformula $\mathtt{e}(ba) \wedge \mathrm{extract}_{\langle 1,2\rangle}(ba, b, a)$ is true.

Next we show that we correctly construct vertex elements corresponding to the atoms $\mathtt{e}(b, a)$, $\mathtt{r}(a)$ and $\mathtt{r}(b)$ in the grounding. Since the cover-based identifier of $\langle b, a\rangle$ is ab in combination with $\langle 2, 1\rangle$, we can conclude that $\delta_{\mathtt{e}[2,1]}(ab)$ is true, which produces the vertex element for $\mathtt{e}(b, a)$. Similarly, $O_{\mathtt{r}} = \{\langle \mathtt{e}, 1\rangle, \langle \mathtt{e}, 2\rangle\}$, and the cover-based identifier of $\langle a\rangle$ and $\langle b\rangle$ is ab together with $\langle 1\rangle$ and $\langle 2\rangle$, respectively. This makes $\delta_{\mathtt{r}[1]}(ab)$ and $\delta_{\mathtt{r}[2]}(ab)$ true and produces the vertex elements for $\mathtt{r}(a)$ and $\mathtt{r}(b)$, respectively.

We illustrate the edge elements just by the construction for the edge from atom $\mathtt{r}(a)$ to atom $\mathtt{e}(b, a)$. Note that $\mathrm{together}_{\mathtt{r},\mathtt{e}}(\langle a\rangle, \langle b, a\rangle)$ is true because $T_{\mathtt{r},\mathtt{e}}$ contains the tuple $\langle \mathtt{e}, \langle 2\rangle, \langle 1, 2\rangle\rangle$ and clearly $\mathtt{e}(ba) \wedge \mathrm{extract}_{\langle 2\rangle}(ba, a) \wedge \mathrm{extract}_{\langle 1,2\rangle}(ba, b, a)$ is true. Moreover, observe that ab together with $\langle 1, 2, 1\rangle$ is the cover-based identifier of $\langle a, b, a\rangle$. This allows us to conclude that $\delta_{\mathtt{r}[1]\mathtt{e}[2,1]}(ab)$ is true, which produces the desired edge element.

Finally we show that our transduction indeed makes the edge that should go from $\mathtt{r}(a)$ to $\mathtt{e}(b, a)$ an outgoing edge of $\mathtt{r}(a)$. Recall that the vertex element for $\mathtt{r}(a)$ exists due to $\delta_{\mathtt{r}[1]}(ab)$ being true, and the edge element exists due to $\delta_{\mathtt{r}[1]\mathtt{e}[2,1]}(ab)$ being true. Clearly $\mathrm{eq}_{\mathtt{r}[1],\mathtt{r}[1]}(ab, ab)$ is true, so $\vartheta_{\mathrm{in}_1,\,\mathtt{r}[1],\,\mathtt{r}[1]\mathtt{e}[2,1]}(ab, ab)$ is also true, which makes the vertex incident to the edge element as desired. $\triangle$

We can also prove that guarded encodings preserve bounded treewidth in a more direct way by modifying a tree decomposition of the input so that the result is a tree decomposition of the grounding. This gives us a simpler, more elementary proof and it also allows us to derive an explicit bound on the treewidth of the grounding. Nevertheless, the preceding proof based on MSO transductions may be of interest because it facilitates the understanding of the MSO transduction in Section 3.4, which is more complex.

Theorem 3.12. *If Π is a fixed guarded ASP program containing c constants and k predicates of arity at most ℓ, and $\mathcal{A}$ is an input structure of Π having treewidth w, then the treewidth of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$ is at most $k \cdot (w + c + 1)^\ell - 1$.*

Proof. Let $\mathcal{T}$ be a tree decomposition of $\mathcal{A}$ having width w, and let C denote the constants in Π. We construct a tree decomposition $\mathcal{T}'$ having width $k \cdot (w + c + 1)^\ell - 1$ of a supergraph of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$. Since the treewidth of a subgraph is at most the treewidth of the whole graph, the statement follows.

We define the tree in $\mathcal{T}'$ to be isomorphic to the tree in $\mathcal{T}$. Let N be a node in $\mathcal{T}$ and B be its bag. We define the bag B' of the corresponding node N' in $\mathcal{T}'$ to consist of all atoms $p(\boldsymbol{x})$ such that p is a predicate occurring in Π and $\boldsymbol{x}$ is a tuple of elements of $B \cup C$. The size of B' is then at most $k \cdot (w + c + 1)^\ell$. It remains to show that $\mathcal{T}'$ is indeed a tree decomposition of a supergraph of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$.

For every predicate atom $p(\boldsymbol{x})$ in a rule r of the grounding, we know from guardedness that there is a ground atom $g(\boldsymbol{y})$ in the positive body of r such that g is extensional and every element of $\boldsymbol{x}$ that is not a constant is also an element of $\boldsymbol{y}$. Since g is extensional, there is a node in $\mathcal{T}$ whose bag contains all elements of $\boldsymbol{y}$. By our construction, the bag of the corresponding node in $\mathcal{T}'$ contains $p(\boldsymbol{x})$.

If two predicate atoms $p(\boldsymbol{x})$ and $q(\boldsymbol{y})$ occur together in a rule r of the grounding, then from guardedness we infer that r also contains an atom $g(\boldsymbol{z})$ in the positive body of r such that g is extensional and every element of $\boldsymbol{x}$ or $\boldsymbol{y}$ that is not a constant is also an element of $\boldsymbol{z}$. As before, it follows that the bag of a node in $\mathcal{T}$ contains all elements of $\boldsymbol{x}$ and $\boldsymbol{y}$ that are not constants, and the bag of the corresponding node in $\mathcal{T}'$ contains both $p(\boldsymbol{x})$ and $q(\boldsymbol{y})$.

If the bags of two nodes N', M' of $\mathcal{T}'$ both contain an atom $p(\boldsymbol{x})$, then the bags of the corresponding nodes N, M in $\mathcal{T}$ contain all elements of $\boldsymbol{x}$ that are not constants. By the connectedness condition, every bag of each node between N and M in $\mathcal{T}$ contains all elements of $\boldsymbol{x}$ that are not constants. Hence, by our construction, the bags of all nodes between N' and M' in $\mathcal{T}'$ contain $p(\boldsymbol{x})$. This proves that $\mathcal{T}'$ is a tree decomposition of a supergraph of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$, and its width is at most $k \cdot (w + c)^\ell - 1$. □

Since the guarded program Π and thus c, k and ℓ are fixed, this shows that the treewidth of the primal graph of $\mathrm{gr}(\pi \cup \mathcal{A})$ is polynomial in the treewidth of the input $\mathcal{A}$.

3.4 Connection-Guarded Answer Set Programs

In this section, we define the class of connection-guarded ASP programs and show that they lead to groundings whose treewidth depends only on the treewidth and the degree of the input structure. We first require the following concept.

Definition 3.13. The *join structure* of a set S of predicate atoms is the following relational structure $\mathcal{J}$ over the signature consisting of the predicate symbols occurring in S, with the same respective arities as in S. The domain of $\mathcal{J}$ consists of the variables and constants occurring in S and for each predicate symbol p it holds that $p^{\mathcal{J}} = \{\boldsymbol{t} \mid p(\boldsymbol{t}) \in S\}$. The *join graph* of S is the Gaifman graph of the join structure of S.

With this notion in hand, we can define our ASP class of interest.

Definition 3.14. Let Π be an ASP program. A *connection-guard* of a rule r in Π is a set G of extensional predicate atoms occurring positively in $B^+(r)$ (nested or unnested) such that all variables that occur in r also occur in G and the join graph of G is connected. We call Π *connection-guarded* if every rule has a connection-guard. We designate an arbitrary connection-guard of r as *the connection-guard* of r.

The following proposition is rather obvious but crucial:

Proposition 3.15. *Let Π be a connection-guarded program without constants, let r be a rule in Π and let $\mathcal{A}$ be an input structure of Π. For any two constants a and b in any ground rule $r' \in \mathrm{gr}(\Pi \cup \mathcal{A})$ obtained from r during grounding, the distance between a and b in $\mathcal{A}$ is at most the number of variables in r minus one.*

Proof. If Π is connection-guarded, the distance between any two variables in the join structure of a rule r with n variables is at most $n-1$. By the nature of grounding, a ground instance of r is only produced if there is a homomorphism from the join structure of r to the input structure $\mathcal{A}$. Hence the distance between a and b in $\mathcal{A}$ is at most $n-1$. □

We next show an important property for connection-guarded ASP programs without constants: For each possible input and every tuple $\boldsymbol{a}$ of constants that can occur in an atom of the grounding, $\boldsymbol{a}$ has an (ℓ, d)-path-based identifier.

Proposition 3.16. *Let Π be a connection-guarded ASP program without constants, let ℓ be the maximum number of variables in any rule of Π, let $\mathcal{A}$ be an input structure of Π, let d be* *the degree of $\mathcal{A}$, and let $p(\boldsymbol{a})$ be a predicate atom that occurs in* $\mathrm{gr}(\Pi \cup \mathcal{A})$*. The tuple $\boldsymbol{a}$ has an (ℓ, d)-path-based identifier in $\mathcal{A}$.*

Proof. By Proposition 3.15, the distance between any two elements of $\boldsymbol{a} = \langle a_1, \dots, a_k \rangle$ is at most ℓ, so there is a domain element s such that, for each i, there is a relative (ℓ, d)-path π_i satisfying $s^{\pi_i} = a_i$. By our observations in Section 3.2, we have seen that then $\boldsymbol{a}$ has an (ℓ, d)-path-based identifier in $\mathcal{A}$. □

We can again generalize this as follows.

Proposition 3.17. *Let Π be a connection-guarded ASP program without constants, let ℓ be the maximum number of variables in any rule of Π, let $\mathcal{A}$ be an input structure of Π, let d be the degree of $\mathcal{A}$, and let $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ be predicate atoms that occur together in a rule of* $\mathrm{gr}(\Pi \cup \mathcal{A})$*. The joint tuple $\boldsymbol{ab}$ has an (ℓ, d)-path-based identifier in $\mathcal{A}$.*

Proof. Since $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ occur together in a rule r of the grounding, the distance between any two elements of $\boldsymbol{ab}$ is at most ℓ by Proposition 3.15. Hence there is an element s of $\mathrm{dom}(\mathcal{A})$ whose distance to each element of $\boldsymbol{ab}$ is at most ℓ, so there is a relative (ℓ, d)-path from s to each element of $\boldsymbol{ab}$. As we have observed in Section 3.2 when we have defined (ℓ, d)-path-based identifiers, this means that $\boldsymbol{ab}$ has an (ℓ, d)-path-based identifier in $\mathcal{A}$. □

The intention of connection-guarded programs is to guarantee that the treewidth of the grounding remains bounded, provided that the treewidth and degree of the input instance is also bounded. The following theorem is the main result of this section and states this formally.

Theorem 3.18. *Let Π be a fixed connection-guarded program and let $\mathcal{A}$ be an input structure of Π. If $\mathcal{A}$ has bounded treewidth and degree, then the primal graph of* $\mathrm{gr}(\Pi \cup \mathcal{A})$ *has bounded treewidth.*

Proof of Theorem 3.18. Our proof is similar to that of Theorem 3.10 for guarded ASP (cf. Figure 3.1), but here we use the bound d on the degree of input structures in our construction of an appropriate MSO transduction $\kappa_{\Pi,d}$. In this case the transduction thus depends on both Π and the degree bound d, and it is only defined for input structures of degree at most d. Since both Π and d are fixed, we get the desired result by Theorem 2.25.

The incidence structure $\mathrm{Inc}(\mathcal{A})$ will be an input structure of $\kappa_{\Pi,d}$, and the incidence structure of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$, which we again consider directed, will be the corresponding output structure. By Theorem 2.25, this shows our claim. Let ℓ be the maximum number of variables in any rule of Π. Furthermore, assume that $\mathcal{A}$ has bounded treewidth and degree, and let d denote the degree of $\mathcal{A}$. For each rule r of Π, we denote the set of variables of r by $\mathrm{Var}(r)$, and we assume that these variables are denoted by $X_1, \dots, X_{|\mathrm{Var}(r)|}$. We will, for the moment, assume that Π is constant-free; we will show later how to handle the general case.

We call the domain elements in the output structure that correspond to vertices and edges of the primal graph *vertex elements* and *edge elements*, respectively. In the following, we define $\kappa_{\Pi,d}$ by the definition scheme $\langle \Delta, \Theta \rangle$, where the tuple Δ is the concatenation of tuples Δ_v and Δ_e, which contain domain formulas that generate the vertex and edge elements, respectively, and Θ contains relation formulas that state which vertex is incident to which edge.

Formulas in Δ_v. These formulas shall produce the vertex elements. First we define a formula $\mathrm{inst}_r(\boldsymbol{x})$ for every rule r to express that the tuple $\boldsymbol{x}$ is an instantiation of the variables in r such that the instantiated connection-guard of r indeed appears in the input facts. To this end, let r be a rule in Π. We first define I_r to be the following set of objects: If the connection-guard of r contains an atom $g(X_{i_1}, \dots, X_{i_k})$, then I_r contains an element $\langle g, i_1, \dots, i_k \rangle$.

$$\mathrm{inst}_r(x_1, \dots, x_{|\mathrm{Var}(r)|}) \;\equiv\; \bigwedge_{\langle g, i_1, \dots, i_k \rangle \in I_r} \exists y \big(g(y) \wedge \mathrm{in}_1(y, x_{i_1}) \wedge \dots \wedge \mathrm{in}_k(y, x_{i_k}) \big)$$

For convenience, we now define the following formula for all nonnegative integers k and m, and for each tuple $\langle i_1, \dots, i_k \rangle$ of integers between 1 and m:

$$\mathrm{select}_{\langle i_1, \dots, i_k \rangle}(x_1, \dots, x_m, y_1, \dots, y_k) \;\equiv\; y_1 = x_{i_1} \wedge \dots \wedge y_k = x_{i_k}$$

The formula $\mathrm{select}_{\boldsymbol{i}}(\boldsymbol{x}, \boldsymbol{y})$ is true if and only if the elements of $\boldsymbol{y}$ are those elements of $\boldsymbol{x}$ given by the indices in $\boldsymbol{i}$.

Next, for each predicate p, we define O_p to be the following set of objects: If a rule r in Π contains an atom $p(X_{i_1}, \dots, X_{i_k})$, then the set O_p contains $\langle r, \boldsymbol{i} \rangle$, where $\boldsymbol{i} = \langle i_1, \dots, i_k \rangle$. With this, we define a formula $\mathrm{occurs}_p(\boldsymbol{x})$ to express that the ground atom $p(\boldsymbol{x})$ occurs in $\mathrm{gr}(\Pi \cup \mathcal{A})$.

$$\mathrm{occurs}_p(\boldsymbol{x}) \;\equiv\; \bigvee_{\langle r, \boldsymbol{i} \rangle \in O_p} \exists \boldsymbol{y} \big(\mathrm{inst}_r(\boldsymbol{y}) \wedge \mathrm{select}_{\boldsymbol{i}}(\boldsymbol{y}, \boldsymbol{x}) \big)$$

We now put this auxiliary formula to use: For each predicate p of arity k occurring in Π and for each k-tuple $\boldsymbol{\pi}$ of relative (ℓ, d)-paths, we define the following formula $\delta_{p[\boldsymbol{\pi}]}(x)$ to be an element of Δ_v.

$$\delta_{p[\boldsymbol{\pi}]}(x) \;\equiv\; \exists \boldsymbol{y} \big(\mathrm{pid}^{\ell,d}_{\boldsymbol{\pi}}(x, \boldsymbol{y}) \wedge \mathrm{occurs}_p(\boldsymbol{y}) \big)$$

This formula is true if and only if x together with $\boldsymbol{\pi}$ is the (ℓ, d)-path-based identifier of some tuple $\boldsymbol{a}$ of domain elements and $p(\boldsymbol{a})$ occurs in $\mathrm{gr}(\Pi \cup \mathcal{A})$; the resulting copy of x in the output structure then corresponds to the atom $p(\boldsymbol{a})$.

For each predicate atom $p(\boldsymbol{a})$ in the grounding, we thus produce a vertex element, since $\boldsymbol{a}$ has an (ℓ, d)-path-based identifier by Proposition 3.16. Moreover, different predicate atoms produce different vertex element and every vertex element that we produce corresponds to an atom in the grounding by our construction of the occurs_p formulas. This proves that there is a bijection between the atoms in the grounding and the vertex elements.

Formulas in Δ_e. These formulas shall produce the edge elements. We again define a few auxiliary formulas. For each rule r in Π, we use G_r to denote the set of all integers i such that X_i is a global variable in r. Now we define the following formula to express that two instantiations $\boldsymbol{x}$ and $\boldsymbol{y}$ of the variables in r agree on the values for the global variables.

$$\text{compat}_r(x_1, \ldots, x_{|\text{Var}(r)|}, y_1, \ldots, y_{|\text{Var}(r)|}) \equiv \bigwedge_{i \in G_r} x_i = y_i$$

Next, for all predicates p and q of arity k and m, respectively, we define $T_{p,q}$ to be the following set of objects: If there is a rule r in Π such that r contains atoms $p(X_{i_1}, \ldots, X_{i_k})$ and $q(X_{j_1}, \ldots, X_{j_m})$, then $T_{p,q}$ contains $\langle r, \boldsymbol{i}, \boldsymbol{j} \rangle$, where $\boldsymbol{i} = \langle i_1, \ldots, i_k \rangle$ and $\boldsymbol{j} = \langle j_1, \ldots, j_m \rangle$. With this, we now define a formula $\text{together}_{p,q}(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are tuples of variables with arity k and m, respectively. This formula expresses that the two ground atoms $p(\boldsymbol{x})$ and $q(\boldsymbol{y})$ occur together in some rule of $\text{gr}(\Pi \cup \mathcal{A})$.

$$\begin{aligned} \text{together}_{p,q}(\boldsymbol{x}, \boldsymbol{y}) \equiv \bigvee_{\langle r, \boldsymbol{i}, \boldsymbol{j} \rangle \in T_{p,q}} & \exists \boldsymbol{z} \exists \boldsymbol{z}' \big(\text{inst}_r(\boldsymbol{z}) \land \text{inst}_r(\boldsymbol{z}') \land \text{compat}_r(\boldsymbol{z}, \boldsymbol{z}') \land \\ & \land \text{select}_{\boldsymbol{i}}(\boldsymbol{z}, \boldsymbol{x}) \land \text{select}_{\boldsymbol{j}}(\boldsymbol{z}, \boldsymbol{y}) \big) \end{aligned}$$

The idea is that $p(\boldsymbol{x})$ and $q(\boldsymbol{y})$ appear together in r if the variables in r can be instantiated by tuples $\boldsymbol{z}$ and $\boldsymbol{z}'$ that agree on the global variables such that $p(\boldsymbol{x})$ appears when applying $\boldsymbol{z}$ and $q(\boldsymbol{y})$ appears when applying $\boldsymbol{z}'$.[5]

With these auxiliary formulas in hand, we define the following formula $\delta_{p[\boldsymbol{\pi}]q[\boldsymbol{\psi}]}(x)$ to be an element of Δ_e, for all predicates p and q, and all tuples $\boldsymbol{\pi}$ and $\boldsymbol{\psi}$ of relative (ℓ, d)-paths, such that the arities of $\boldsymbol{\pi}$ and $\boldsymbol{\psi}$ are the same as those of p and q, respectively.

$$\delta_{p[\boldsymbol{\pi}]q[\boldsymbol{\psi}]}(x) \equiv \begin{cases} \bot & \text{if } p[\boldsymbol{\pi}] = q[\boldsymbol{\psi}] \\ \exists \boldsymbol{y} \exists \boldsymbol{z} \big(\text{pid}^{\ell,d}_{\boldsymbol{\pi}\boldsymbol{\psi}}(x, \boldsymbol{y}, \boldsymbol{z}) \land \text{together}_{p,q}(\boldsymbol{y}, \boldsymbol{z}) \big) & \text{otherwise} \end{cases}$$

This formula is true if and only if there are tuples $\boldsymbol{a}$ and $\boldsymbol{b}$ of the same arity as p and q, respectively, such that (1) x together with $\boldsymbol{\pi}\boldsymbol{\psi}$ is the (ℓ, d)-path-based identifier of

[5]If r contains no aggregates, then this is equivalent to checking if there is a single instantiation of the variables in r such that both atoms appear. However, we must be more careful due to rules like $\leftarrow$ `p(X), #sum{ 1,Y : q(X,Y) }`, where atoms like `q(a,b)` and `q(a,c)` may appear together in the grounding.

$\boldsymbol{ab}$, and (2) the atoms $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ are different and occur together in some rule of $\mathrm{gr}(\Pi \cup \mathcal{A})$. The resulting copy of x in the output structure then corresponds to the edge from $p(\boldsymbol{a})$ to $q(\boldsymbol{b})$ in the primal graph. Due to symmetry, we can see that then also an edge in the other direction will be created. Since both atoms are different if the formula is true, we do not introduce loops.

For each pair of different predicate atoms $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ that jointly occur in a rule of the grounding, we thus correctly produce two edge elements, since $\boldsymbol{ab}$ has an (ℓ, d)-path-based identifier by Proposition 3.17. Moreover, different such pairs of atoms produce different edge elements, and every edge element that we produce corresponds to a joint occurrence of two atoms in a rule of the grounding by our construction of the $\mathrm{together}_{p,q}$ formulas.

Formulas in Θ*.* These formulas shall ensure that each edge element is incident to the two appropriate vertex elements. First we define the formula $\mathrm{meet}_{\boldsymbol{\pi},\boldsymbol{\psi}}(x, y)$ for each integer k and all k-tuples $\boldsymbol{\pi}$ and $\boldsymbol{\psi}$ of relative paths. The formula is true if and only if $x^{\boldsymbol{\pi}} = y^{\boldsymbol{\psi}}$.

$$\mathrm{meet}_{\langle \pi_1,\ldots,\pi_k\rangle,\langle \psi_1,\ldots,\psi_k\rangle}(x, y) \;\equiv\; \bigwedge_{1 \leqslant i \leqslant k} \exists z \big(\mathrm{reach}_{\pi_i}(x, z) \wedge \mathrm{reach}_{\psi_i}(y, z) \big)$$

Now let p and q be predicates occurring in Π, and let $\boldsymbol{\pi}$ and $\boldsymbol{\psi}$ be tuples of relative (ℓ, d)-paths. We define a formula $\mathrm{eq}_{p[\boldsymbol{\pi}],q[\boldsymbol{\psi}]}(x, y)$ to express that the atoms $p(x^{\boldsymbol{\pi}})$ and $q(y^{\boldsymbol{\psi}})$ are equal.

$$\mathrm{eq}_{p[\boldsymbol{\pi}],q[\boldsymbol{\psi}]}(x, y) \;\equiv\; \begin{cases} \mathrm{meet}_{\boldsymbol{\pi},\boldsymbol{\psi}}(x, y) & \text{if } p = q \\ \bot & \text{otherwise} \end{cases}$$

Let p, q and q' be predicates occurring in Π, and let $\boldsymbol{\pi}$, $\boldsymbol{\psi}$ and $\boldsymbol{\psi}'$ be tuples of relative (ℓ, d)-paths with the same arity as p, q and q', respectively. We define the following formulas to be an element of Θ:

$$\begin{aligned} \vartheta_{\mathrm{in}_1,\, p[\boldsymbol{\pi}],\, q[\boldsymbol{\psi}] q'[\boldsymbol{\psi}']}(x, y) \;&\equiv\; \mathrm{eq}_{p[\boldsymbol{\pi}],q[\boldsymbol{\psi}]}(x, y) \\ \vartheta_{\mathrm{in}_2,\, p[\boldsymbol{\pi}],\, q[\boldsymbol{\psi}] q'[\boldsymbol{\psi}']}(x, y) \;&\equiv\; \mathrm{eq}_{p[\boldsymbol{\pi}],q'[\boldsymbol{\psi}']}(x, y) \end{aligned}$$

We only explain the first of these formulas, as the other case is symmetric.

The formula $\vartheta_{\mathrm{in}_1,\, p[\boldsymbol{\pi}],\, q[\boldsymbol{\psi}] q'[\boldsymbol{\psi}']}(x, y)$ is true if and only if the atom $p(\boldsymbol{a})$ is equal to $q(\boldsymbol{b})$, where $\boldsymbol{a} = x^{\boldsymbol{\pi}}$ and $\boldsymbol{b} = y^{\boldsymbol{\psi}}$. If this formula is true, it makes the edge represented by the respective copy of y an *outgoing* edge of $p(\boldsymbol{a})$ because of the subscript in_1.

We first show that, whenever our transduction causes an edge element to be incident to a vertex element, the corresponding edge in the primal graph is indeed an outgoing edge of the appropriate vertex. Suppose there are predicates p, q and p', tuples $\boldsymbol{\pi}$, $\boldsymbol{\psi}$

and $\boldsymbol{\psi}'$, as well as domain elements x and y such that (1) $\delta_{p[\boldsymbol{\pi}]}(x)$ is true, (2) $\delta_{q[\boldsymbol{\psi}]q'[\boldsymbol{\psi}']}(y)$ is true, and (3) $\vartheta_{\mathrm{in}_1, p[\boldsymbol{\pi}], q[\boldsymbol{\psi}]q'[\boldsymbol{\psi}']}(x, y)$ is true. As observed in our definition of Δ_v, (1) means that x together with $\boldsymbol{\pi}$ is the (ℓ, d)-path-based identifier of some tuple $\boldsymbol{a}$, and the grounding contains an atom $p(\boldsymbol{a})$. From (2) we get that there are tuples $\boldsymbol{b}$ and $\boldsymbol{b}'$ such that y together with $\boldsymbol{\psi\psi}'$ is the (ℓ, d)-path-based identifier of $\boldsymbol{bb}'$. We have also seen that by (2) there is a rule in the grounding that contains both $q(\boldsymbol{b})$ and $q'(\boldsymbol{b}')$, hence there is an edge from $q(\boldsymbol{b})$ to $q'(\boldsymbol{b}')$ in the primal graph. By (3) and the definition of (ℓ, d)-path-based identifiers (in particular those of $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{b}'$), we know that $p(\boldsymbol{a})$ is equal to $q(\boldsymbol{b})$. Hence the edge in the primal graph from $q(\boldsymbol{b})$ to $q'(\boldsymbol{b}')$ is indeed an outgoing edge of $p(\boldsymbol{a})$.

Finally we prove the other direction: Whenever an edge in the primal graph is an outgoing edge of a vertex, a formula in Θ defining the relation in_1 causes the corresponding edge element to an outgoing edge of the appropriate vertex element. Suppose that the primal graph contains an edge from atom $p(\boldsymbol{a})$ to atom $q(\boldsymbol{b})$. Then these atoms occur together in a rule of the grounding and, as we have seen in our definition of Δ_v and Δ_e, our transduction produces a vertex element v for $p(\boldsymbol{a})$ and an edge element e for the edge from $p(\boldsymbol{a})$ to $q(\boldsymbol{b})$. Now let x and y be domain elements of $\mathcal{A}$, and let $\boldsymbol{\pi}$, $\boldsymbol{\psi}$ and $\boldsymbol{\psi}'$ be tuples of relative (ℓ, d)-paths, such that x together with $\boldsymbol{\pi}$ is the (ℓ, d)-path-based identifier of $\boldsymbol{a}$, and y together with $\boldsymbol{\psi\psi}'$ is the (ℓ, d)-path-based identifier of $\boldsymbol{ab}$. By definition of (ℓ, d)-path-based identifiers, $x^{\boldsymbol{\pi}} = \boldsymbol{a}$; moreover, $y^{\boldsymbol{\psi\psi}'} = \boldsymbol{ab}$, which entails $y^{\boldsymbol{\psi}} = \boldsymbol{a}$. Since both $\boldsymbol{a} = x^{\boldsymbol{\pi}}$ and $\boldsymbol{a} = y^{\boldsymbol{\psi}}$, the formula $\mathrm{eq}_{p[\boldsymbol{\pi}], p[\boldsymbol{\psi}]}(x, y)$ is clearly true. Hence the formula $\vartheta_{\mathrm{in}_1, p[\boldsymbol{\pi}], p[\boldsymbol{\psi}]q[\boldsymbol{\psi}']}(x, y)$ is true, which correctly makes the edge element e an outgoing edge of the vertex element v.

For the remaining relation formulas, which define the relation E, we proceed in the same way as in the proof of Theorem 3.10.

This completes the construction of the MSO transduction $\kappa_{\Pi,d}$. Let $\mathcal{A}$ be an input structure for Π with degree bounded by d. Clearly, since Π, d and ℓ are fixed, so is $\kappa_{\Pi,d}$. We have argued that $\kappa_{\Pi,d}(\mathrm{Inc}(\mathcal{A}))$ yields the incidence structure of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$ as desired. By Theorem 2.25 and the fact that $\mathcal{A}$, by assumption, has bounded treewidth, this proves our claim for constant-free programs.

It is tedious, but straightforward, to generalize this proof to programs with constants, and we will only give the general idea here. For each constant occurring in Π, we add a constant symbol to our MSO signature. Then we adapt our definitions of the MSO formulas in the following ways: (1) We need to extend our definition of the MSO formulas in such a way that every relative path can also be a constant symbol instead. (2) We define $\mathrm{reach}_\pi(x, y)$ as $y = \pi$ whenever π is a constant symbol. Intuitively, we can "reach" every constant from every domain element. (3) For the formula $\mathrm{frp}^d_\pi(x, y)$, we adjust the definition of $\prec_d$ such that each constant is smaller than all relative d-paths, and the constants are ordered in an arbitrary way. (4) The set I_r in the definition of $\mathrm{inst}_r(\boldsymbol{x})$ shall contain a tuple $\langle g, e_1, \ldots, e_m \rangle$ whenever the connection-guard of r contains a predicate atom of the form $g(a_1, \ldots, a_m)$, where each a_j is either a variable X_i, in which case $e_j = i$, or a_j is a constant symbol, in which case $e_j = a_j$. In the formula $\mathrm{inst}_r(\boldsymbol{x})$, we then replace a conjunct $\mathrm{in}_j(y, x_{i_j})$ by $\mathrm{in}_j(y, i_j)$ whenever i_j is a constant. We also change the definition of the sets O_p and $T_{p,q}$ in the definitions of $\mathrm{occurs}_p(x)$ and $\mathrm{together}_{p,q}(\boldsymbol{x}, \boldsymbol{y})$ to account for constants in a similar way as in our adjusted definition of I_r. (5) In the definition of $\mathrm{select}_{\boldsymbol{i}}(\boldsymbol{x}, \boldsymbol{y})$, every element i_j of $\langle i_1, \ldots, i_k\rangle$ can also be a constant symbol, in which case the corresponding conjunct in the formula is $y_j = i_j$. It is easy to generalize Propositions 3.16 and 3.17 to programs with constants, and to verify that the resulting transduction is correct. $\square$

We now illustrate this transduction for the same program as in Example 3.11.

Example 3.19. Let Π be the connection-guarded program from Example 3.11. We denote the rules by $r_1, \ldots, r_4$ from top to bottom. Again let $\mathcal{G}$ be the input structure for Π satisfying $\mathrm{dom}(\mathcal{G}) = \{a, b\}$, $\mathrm{succ}^{\mathcal{G}} = \{\langle a, b\rangle\}$ and $\mathtt{e}^{\mathcal{G}} = \{\langle b, a\rangle\}$, and whose degree is bounded by some integer d. We denote the domain element of $\mathrm{Inc}(\mathcal{G})$ for the fact $\langle b, a\rangle$ in $\mathtt{e}^{\mathcal{G}}$ and for $\langle a, b\rangle$ in $\mathrm{succ}^{\mathcal{G}}$ by ba and ab, respectively, and we assume the ordering $a < b < ab < ba$. We show how $\kappa_{\Pi,d}$ transforms $\mathrm{Inc}(\mathcal{G})$ into the incidence structure of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{G})$.

For the vertex elements, first observe that the formula $\mathrm{inst}_r(b, a)$ is true for every rule r because each rule has the guard $\mathtt{e}(\mathtt{X}, \mathtt{Y})$ and the only input fact is $\mathtt{e}(b, a)$. Now observe that all of the formulas $\mathrm{occurs}_{\mathtt{e}}(b, a)$, $\mathrm{occurs}_{\mathtt{r}}(a)$, $\mathrm{occurs}_{\mathtt{r}}(b)$, $\mathrm{occurs}_{\mathtt{g}}(a)$ and $\mathrm{occurs}_{\mathtt{g}}(b)$ are true. For instance, observe that $O_{\mathtt{e}} = \{\langle r_1, 1, 2\rangle, \langle r_2, 1, 2\rangle, \langle r_3, 1, 2\rangle, \langle r_4, 1, 2\rangle\}$ (under the assumption that the first variable of each rule is $\mathtt{X}$ and the second is $\mathtt{Y}$); now clearly the subformula $\mathrm{inst}_r(b, a) \wedge \mathrm{select}_{\langle 1,2\rangle}(b, a, b, a)$ is true for every rule r. Hence $\mathrm{occurs}_{\mathtt{e}}(b, a)$ is true. Since the $(2, d)$-path-based identifier of $\langle b, a\rangle$ is b in combination with $\langle \epsilon, \langle 1\rangle\rangle$, we can conclude that $\delta_{\mathtt{e}[\epsilon,\langle 1\rangle]}(b)$ is true, which produces the vertex element for $\mathtt{e}(b, a)$.

Similarly, $O_{\mathtt{r}} = \{\langle r_1, 1\rangle, \langle r_2, 2\rangle, \langle r_3, 1\rangle, \langle r_3, 2\rangle\}$, by which we can see that both $\mathrm{occurs}_{\mathtt{r}}(a)$ and $\mathrm{occurs}_{\mathtt{r}}(b)$ are true due to the elements of $O_{\mathtt{r}}$ containing 2 and 1, respectively. The $(2, d)$-path-based identifiers of $\langle a\rangle$ and $\langle b\rangle$ are a together with $\langle \epsilon\rangle$ and b together with $\langle \epsilon\rangle$, respectively. This makes $\delta_{\mathtt{r}[\epsilon]}(a)$ and $\delta_{\mathtt{r}[\epsilon]}(b)$ true and produces the vertex elements for $\mathtt{r}(a)$ and $\mathtt{r}(b)$, respectively.

We illustrate the edge elements just by the construction for the edge from atom $\mathtt{r}(a)$ to atom $\mathtt{e}(b, a)$. Note that $\mathrm{together}_{\mathtt{r},\mathtt{e}}(\langle a\rangle, \langle b, a\rangle)$ is true because $T_{\mathtt{r},\mathtt{e}}$ contains the tuple $\langle r_1, \langle 2\rangle, \langle 1, 2\rangle\rangle$ as well as $\langle r_3, \langle 2\rangle, \langle 1, 2\rangle\rangle$, and clearly both subformulas $\mathrm{inst}_{r_1}(b, a) \wedge \mathrm{inst}_{r_1}(b, a) \wedge \mathrm{compat}_r(b, a, b, a) \wedge \mathrm{select}_{\langle 2\rangle}(b, a, a) \wedge \mathrm{select}_{\langle 1,2\rangle}(b, a, b, a)$ and $\mathrm{inst}_{r_3}(b, a) \wedge \mathrm{inst}_{r_3}(b, a) \wedge \mathrm{compat}_r(b, a, b, a) \wedge \mathrm{select}_{\langle 2\rangle}(b, a, a) \wedge \mathrm{select}_{\langle 1,2\rangle}(b, a, b, a)$ are true. Moreover, observe that b together with $\langle \langle 1\rangle, \epsilon, \langle 1\rangle\rangle$ is the $(2, d)$-path-based identifier of $\langle a, b, a\rangle$. This allows us to conclude that $\delta_{\mathtt{r}[\langle 1\rangle]\mathtt{e}[\epsilon,\langle 1\rangle]}(b)$ is true, which produces the desired edge element.

Finally we show that our transduction indeed makes the edge that should go from $\mathtt{r}(a)$ to $\mathtt{e}(b,a)$ an outgoing edge of $\mathtt{r}(a)$. Recall that the vertex element exists due to $\delta_{\mathtt{r}[\epsilon]}(a)$ being true, and the edge element exists due to $\delta_{\mathtt{r}[\langle 1\rangle]\mathtt{e}[\epsilon,\langle 1\rangle]}(b)$ being true. Clearly $\mathrm{eq}_{\mathtt{r}[\epsilon],\mathtt{r}[\langle 1\rangle]}(a,b)$ is true, so $\vartheta_{\mathrm{in}_1,\,\mathtt{r}[\epsilon],\,\mathtt{r}[\langle 1\rangle]\mathtt{e}[\epsilon,\langle 1\rangle]}(a,b)$ is also true, which makes the vertex incident to the edge element as desired. $\triangle$

Obviously every guarded ASP program is also connection-guarded. The other direction, however, is not true. Consider, for example, the connection-guarded program consisting of the rule $\mathtt{p(X,Z)} \leftarrow \mathtt{e(X,Y)},\ \mathtt{e(Y,Z)}$. As an input for this program, consider a path of length 2. The extension of $\mathtt{p}$ in the unique answer set contains both endpoints of the path. However, there can be no equivalent guarded program since answer sets of groundings resulting from guarded programs have the property that the extension of any predicate can only be a subset of the extension of an extensional predicate.

While connection-guarded ASP is a strict superset of guarded ASP in the sense that each guarded program is connection-guarded but not vice versa, there seems to be a price to pay for the higher generality when the objective is to keep the treewidth of the grounding low: Comparing Theorem 3.10, which concerns guarded ASP programs, with Theorem 3.18, which concerns connection-guarded programs, we notice that we rely on the input having bounded degree for showing that grounding connection-guarded programs preserves bounded degree of the input, whereas there is no such assumption for guarded programs.

It is natural to ask whether this additional condition is necessary for connection-guarded programs. Unfortunately it is, as witnessed by the rule $\mathtt{p(X,Z)} \leftarrow \mathtt{e(X,Y)},\ \mathtt{e(Y,Z)}$, where $\mathtt{e}$ is extensional: When given a tree of height 1 with n vertices (and thus of treewidth 1 and maximum degree $n-1$), the primal graph of the grounding has linear treewidth, as the complete bipartite graph $K_{n-1,n-1}$ is a subgraph of it.

Also the restrictions in Definition 3.14 cannot easily be relaxed without destroying bounded treewidth already with very simple programs: If we allow "unconnected" rules like $\mathtt{p(X,Y)} \leftarrow \mathtt{v(X)},\ \mathtt{v(Y)}$, then the complete graph K_n is a subgraph of the primal graph of the grounding for any n-vertex instance.

Moreover, if we change the definition of the extensional join graph by drawing edges also for intensional atoms, then the first encoding in Example 3.1 is allowed, which generates K_n for any connected graph.

3.5 Complexity

We have seen that most restrictions of connection-guarded ASP cannot be lifted without losing the property that grounding preserves bounded treewidth of the input when the degree is also bounded. On the other hand, guarded ASP does not require the bound

```
      t(T) ← verum(T).
      f(F) ← falsum(F).
t(X) ∨ f(X) ← exists(X).
t(Y) ∨ f(Y) ← forall(Y).
         w ← term(X,Y,Z,Na,Nb,Nc), t(X), t(Y), t(Z), f(Na), f(Nb), f(Nc).
      t(Y) ← w, forall(Y).
      f(Y) ← w, forall(Y).
           ← not w.
```

Listing 3.4: An encoding of Qsat$_2$ in guarded ASP

on the degree, but it is syntactically even more restrictive. This raises suspicions about whether the classes are too restrictive to be useful.

Luckily, straightforward encodings for problems like Graph Coloring or Hamiltonian Cycle even fall into the class of guarded ASP (cf. the second encoding in Example 3.1), and consequently also into connection-guarded ASP. Moreover, it turns out that the restrictions imposed by guardedness and connection-guardedness do not alleviate the complexity of deciding answer set existence when the program is fixed:

Theorem 3.20. *It is Σ_2^P-complete to decide for a fixed guarded or connection-guarded ASP program Π and a given input structure $\mathcal{A}$ whether $\Pi \cup \mathcal{A}$ has an answer set.*

Proof. For membership, we guess an interpretation I and then check by calling a co-NP oracle whether I is a minimal model of $\mathrm{gr}(\Pi \cup \mathcal{A})^I$. For hardness, we present a guarded encoding for the well-known Σ_2^P-complete problem Qsat$_2$. We are given a formula $\exists x_1 \cdots \exists x_k \forall y_1 \cdots \forall y_\ell\, \varphi$, where φ is a formula in 3-DNF (i.e., a disjunction of conjunctive terms, each containing at most three literals), and the question is whether there are truth values for the x variables such that for all truth values for the y variables φ is true. We assume that each disjunct in φ contains exactly three literals, which can be achieved by using the same literal multiple times in a disjunct.

Consider the ASP program in Figure 3.4, which is based on the encoding in Section 3.3.5 of the paper by Leone et al. (2006). The Qsat$_2$ formula is represented as a structure $\mathcal{A}$ as follows: The domain of $\mathcal{A}$ consists of all variables in φ and two special elements $\top$ and $\bot$. We choose $\texttt{verum}^\mathcal{A} = \{\top\}$ and $\texttt{falsum}^\mathcal{A} = \{\bot\}$. The relations $\texttt{exists}^\mathcal{A}$ and $\texttt{forall}^\mathcal{A}$ consist of all existentially and universally quantified variables, respectively. Finally, for each disjunct $l_1 \wedge l_2 \wedge l_3$ in the formula, we put an element $\langle p_1, p_2, p_3, q_1, q_2, q_3 \rangle$ into $\texttt{term}^\mathcal{A}$, where p_i denotes v_i if l_i is a positive atom v_i, otherwise $p_i = \top$, and q_i denotes v_i if l_i is an atom of the form **not** v_i, otherwise $q_i = \bot$. The element $\langle p_1, p_2, p_3, q_1, q_2, q_3 \rangle$ thus represents $p_1 \wedge p_2 \wedge p_3 \wedge \neg q_1 \wedge \neg q_2 \wedge \neg q_3$, which is equivalent to the original disjunct. This program is clearly guarded and indeed

```
strat(Y) ∨ strat(Z) ← prod_by(X,Y,Z).
          strat(W) ← contr_by(W,X,Y,Z), strat(X), strat(Y), strat(Z).
⇜ avoid(Sell,Keep,P), not strat(Sell), strat(Keep).   [1@P,Sell,Keep]
```

Listing 3.5: An encoding of PREFERRED STRATEGIC COMPANIES in guarded ASP

	treewidth	degree	treewidth + degree
guarded	FPT (3.22)	Σ_2^P-complete (3.28)	FPT
connection-guarded	NP-hard (3.26)	Σ_2^P-complete	FPT (3.23)

Table 3.1: Parameterized complexity of answer set existence for our considered classes when the program is fixed. In parentheses: Number of the theorem proving the result. Results without parentheses are implied by other results.

encodes the QSAT₂ problem, as can be seen by the arguments in the work by Leone et al. (2006). □

Note that weak constraints have no effect on whether a program *has* an answer set, but they do have an effect on *which* answer sets are admitted. Thus, if we consider a slightly different reasoning problem, we see that weak constraints increase the complexity of ASP. This fact is well known for general ASP, and it also holds for our ASP classes.

Theorem 3.21. *It is Δ_3^P-complete to decide for a fixed guarded or connection-guarded ASP program Π and a given input structure $\mathcal{A}$ whether a given ground atom is true in an optimal answer set of $\Pi \cup \mathcal{A}$.*

Proof. Membership follows from the fact that the same problem on general ASP programs is Δ_3^P-complete (Buccafurri, Leone and Rullo 2000; Leone et al. 2006). To show hardness, we present a guarded encoding of the PREFERRED STRATEGIC COMPANIES problem, which is also known to be Δ_3^P-complete. We omit a description of the problem, which can be found in Section 3.3.7 of the paper by Leone et al. (2006). The encoding in Figure 3.5 is also taken from there (with slight adaptations). The input predicates are `prod_by`, `contr_by` and `avoid`. Clearly this program is guarded. □

We now turn to the parameterized complexity of the answer set existence problem for fixed guarded and connection-guarded ASP programs. The parameters we consider are the treewidth and the degree of the input structures, as well as the combined parameter treewidth + degree. Our most important results are summarized in Table 3.1. In the following we prove these results (and more) individually.

When we consider the treewidth of the input structures as the parameter, we will see that answer-set solving for fixed guarded programs is in fact fixed-parameter tractable. It is also fixed-parameter tractable if we consider fixed connection-guarded programs and the combination of treewidth and degree as the parameter. In other words, the ASP classes we defined are not only useful for encoding problems in such a way that we implicitly benefit from the treewidth-sensitivity inherent to state-of-the-art ASP solvers due to the results in Theorem 3.10 and Theorem 3.18, but they are also amenable to algorithms that explicitly exploit small treewidth.

As we have shown in Theorem 3.10, grounding a fixed guarded ASP program together with an input structure $\mathcal{A}$ leads to a grounding whose treewidth only depends on the treewidth of $\mathcal{A}$. This is of particular interest when we consider the fact that the problem of deciding whether a ground ASP program admits an answer set is FPT when parameterized by treewidth (Gottlob, Pichler and Wei 2010a). This even holds when the language is augmented by additional constructs such as *optimization rules* and *weight rules*, which are similar to weak constraints and aggregates, respectively (Fichte et al. 2017).[6] Moreover, the algorithm by Fichte et al. (2017) clearly also works in FPT time for deciding Brave Reasoning and it can even print the cost of optimal answer sets. By combining these results, we obtain the following theorem:

Theorem 3.22. *For every fixed guarded ASP program* Π *the problem of deciding for a given input structure* $\mathcal{A}$ *whether a given ground atom is true in an optimal answer set of* $\Pi \cup \mathcal{A}$ *is fixed-parameter tractable when parameterized by the treewidth of* $\mathcal{A}$.

Proof. Let Γ denote $\mathrm{gr}(\Pi \cup \mathcal{A})$ in the following. First note that the size of Γ is linear in the size of $\mathcal{A}$, since every rule in Π is guarded and thus has at most one ground instance in Γ for every fact in $\mathcal{A}$. Moreover, by Theorem 3.10, the treewidth of Γ only depends on the treewidth of $\mathcal{A}$.

To prove the statement, we show how we can transform the weak constraints and aggregates that occur in Γ into optimization rules and weight rules (as defined in the paper by Fichte et al. (2017)). We first turn weak constraints into optimization rules and thus obtain a program Γ_o. Then we transform Γ_o into a program $\Gamma_{o,w}$ by turning aggregates into weight rules. Our transformations make sure that the size of $\Gamma_{o,w}$ is polynomial in the size of Γ, and the treewidth of $\Gamma_{o,w}$ is linear in the treewidth of Γ. This in turn implies by our initial observations that the size of $\Gamma_{o,w}$ is polynomial in the size of $\mathcal{A}$, and that the treewidth of $\Gamma_{o,w}$ only depends on the treewidth of $\mathcal{A}$. The statement then follows from a direct application of the fixed-parameter linear algorithm from Theorem 2 of the paper by Fichte et al. (2017).

First we transform weak constraints into optimization rules. Ground weak constraints have the form $\leftsquigarrow b_1, \dots, b_n\ [w@l, t_1, \dots, t_m]$, where $b_1, \dots, b_n$ are literals, w and l are integers and $t_1, \dots, t_m$ are terms. An optimization rule has the form $\leftsquigarrow b[w]$, where b is

[6]Note that weight rules make the problem W[1]-hard when the parameter is the treewidth of the *incidence graph* of the ground program (Pichler et al. 2014). Here we deal with the treewidth of the *primal graph*, where the problem is in FPT even in the presence of weight constraints.

a literal and w is a nonnegative integer. Similar to weak constraints, the meaning of optimization rules is to restrict the solutions to those answer sets that have minimum cost, where the cost is determined by adding the weight w of each optimization rule whose body is true. In contrast to weak constraints, optimization rules only have a single literal in the body, there is no possibility to specify different priority levels l and the weights w cannot be negative. We first make the following assumptions about Γ:

1. All levels l in our weak constraints are nonnegative. If there are negative levels, we can achieve this condition by adding the absolute value of the smallest level to each level.
2. There are no negative weights w in our weak constraints. If there are negative weights, we add the absolute value of the smallest weight to each weight.
3. Each level number is at most the number of weak constraints in the grounding. This is easy to achieve: Suppose there are two level numbers a and b that both occur in the ground program such that $a < b$ but there is a "gap" between them; that is, $b - a > 1$ and there is no level number occurring in the program that is between a and b. It is easy to see that, by the semantics of weak constraints, we can replace each occurrence of the level number b in the grounding by $a + 1$ without changing which answer sets are optimal. Hence we may assume that each level number is linear in the size of the grounding.

Now we rewrite Γ into an equivalent ground program Γ_o such that only a single (nonnegative) level l appears in the weak constraints of Γ_o. To achieve this, we replace each expression $w@l$ by $w \cdot (s+1)^l @0$, where s is the sum of all weights in Γ. Note that s is at most the number of rules in Γ times the highest weight. Hence the value of s is in $\mathcal{O}(n \cdot 2^n)$, where n is the size of Γ. As we have seen, l is in $\mathcal{O}(n)$. So $w \cdot (s+1)^l$ is in $\mathcal{O}(2^n \cdot (n \cdot 2^n)^n)$. We can represent this number with $\mathcal{O}(n^2)$ many bits. Since the size of Γ is linear in the size of $\mathcal{A}$, the size of Γ_o is thus still polynomial in the size of $\mathcal{A}$.

For each expression $[w@l, t_1, \dots, t_m]$ that appears in a weak constraint, we now add an optimization rule $\leftsquigarrow \text{add}_{w@l,t_1,\dots,t_m} [w]$, where $\text{add}_{w@l,t_1,\dots,t_m}$ is a new atom. Finally, we replace each weak constraint $\leftsquigarrow b_1, \dots, b_n \; [w@l, t_1, \dots, t_m]$ by the rule $\text{add}_{w@l,t_1,\dots,t_m} \leftarrow b_1, \dots, b_n$. Note that this increases the treewidth at most by one, and the size of Γ_o clearly stays polynomial in the size of $\mathcal{A}$.

Next we turn aggregates into weight rules. A weight rule has the form $a \leftarrow w \leqslant \{b_1 = w_1, \dots, b_n = w_n\}$, where a is an atom, $b_1, \dots, b_n$ are literals and $w, w_1, \dots, w_n$ are nonnegative integers. The body of a weight rule is true if w is less than of equal to the sum of all w_i such that b_i is true. To turn the ground program Π_o, which may contain aggregates, into an equivalent ground program $\Gamma_{o,w}$ that uses weight rules instead, we follow the rewriting technique from the paper by Alviano, Faber and Gebser (2015).

This rewriting is rather involved and we do not reiterate it here. We only sketch why each of its steps preserves bounded treewidth.

1. We first apply some transformations that turn all aggregates into sum aggregates with just the comparison operators $>$ and $\neq$. This can be done by splitting an aggregate into at most three aggregates,[7] performing some arithmetic on the weights and bounds, and possibly changing comparison operators. This only increases the size of $\Gamma_{o,w}$ by at most a factor of three. Moreover, for any rule r, this does not change the atoms that occur in r, so it has no impact on the treewidth. These claims can be easily verified by looking at the equivalences in Section 3.1 of the paper by Alviano, Faber and Gebser (2015).

2. We then replace each aggregate A containing the comparison operator $>$ with a new atom aux_A, and we add a rule $\text{aux}_A \leftarrow A'$, where A' is an aggregate containing only nonnegative weights, the same atoms as A and, for certain atoms p that occur in A, a copy p^F. This is formalized in rule (4) of the paper by Alviano, Faber and Gebser (2015). Adding the atoms p^F clearly increases both the program size and the treewidth at most by a factor of two. Since its weights are nonnegative, the new rule $\text{aux}_A \leftarrow A'$ can be seen as a weight rule.

3. For each new atom p^F added in the previous step, we add a constant number of new rules into the program, but these new rules only involve p, p^F and aux_A. This is formalized in rules (5)–(7) of the paper by Alviano, Faber and Gebser (2015). After rewriting an aggregate A in this way, the size of $\Gamma_{o,w}$ only increases by a constant factor. Moreover, the treewidth of the primal graph of $\Gamma_{o,w}$ is at most the treewidth of the graph that results from the primal graph of Γ_o by adding one new atom vertex p^F for every atom p that occurs in A, adding an atom vertex aux_A, and constructing a clique among aux_A and all atom vertices p and p^F such that p occurs in A. Hence this transformation only increases the treewidth at most by a factor of two.

For the aggregates over the comparison operator $\neq$, the procedure is slightly different, but the bounds on program size and treewidth can be established in the same way.

We can now apply the fixed-parameter linear algorithm described in Section 3.3 of the paper by Fichte et al. (2017) to obtain the desired result.[8]

[7]Note that in the presence of `#even` and `#odd` aggregates this bound of three does not hold. However, we can still give a linear bound, which is fine for obtaining fixed-parameter tractability. Moreover, `#even` and `#odd` predicates are not part of the ASP language specification (Calimeri et al. 2015) anyway, so we just mention this in passing here.

[8]Actually Fichte et al. (2017) prove the correctness of two algorithms: one works on the incidence graph of the grounding and the other works on the primal graph. Neither of these algorithms accounts for optimization rules. The authors do show how to extend the incidence-graph algorithm to optimization

This proves that the size of $\Gamma_{o,w}$ is polynomial in the size of Γ and that the treewidth of $\Gamma_{o,w}$ is linear in the treewidth of Γ, which only depends on the treewidth of $\mathcal{A}$ by Theorem 3.10. □

Unsurprisingly, an analogous statement can be shown for connection-guarded ASP.

Theorem 3.23. *For every fixed connection-guarded ASP program Π the problem of deciding for a given input structure $\mathcal{A}$ whether a given ground atom is true in an optimal answer set of $\Pi \cup \mathcal{A}$ is fixed-parameter tractable when parameterized by the combination of the treewidth and degree of $\mathcal{A}$.*

Proof. Since Π is connection-guarded, observe that the size of $\mathrm{gr}(\Pi \cup \mathcal{A})$ is in $\mathcal{O}(n \cdot d^\ell)$, where n and d denote the size and degree of $\mathcal{A}$, respectively, and ℓ is the maximum number of variables in a rule of Π. Hence, for bounded d and ℓ, the size of the grounding is linear in n. We can now prove the statement in the same way as Theorem 3.22, with the modification that we invoke Theorem 3.18 instead of Theorem 3.10. □

If we restrict the program a little bit further, we can even obtain fixed-parameter linear algorithms:

Corollary 3.24. *For every fixed guarded ASP program Π there is a fixed-parameter linear algorithm that decides for every input structure $\mathcal{A}$ whether a given ground atom is true in an optimal answer set of $\Pi \cup \mathcal{A}$, when the parameter is the treewidth of $\mathcal{A}$, if the levels and weights of all weak constraints in Π are bounded by a fixed constant.*

Proof. We use the same algorithm as in the proof of Theorem 3.22 and observe that it becomes fixed-parameter linear for the following reason: If the levels and weights of all weak constraints are fixed, then our "flattening" of the weights $w@l$ to $w \cdot (s+1)^l@0$ leads to integers that can be represented with a logarithmic number of bits, and these numbers can be computed in constant time under the assumption of the uniform-cost model. The size of the program Γ_o is then linear in the size of $\mathcal{A}$. Hence the size of $\Gamma_{o,w}$ is still linear in the size of $\mathcal{A}$. □

For connection-guarded ASP programs, the same restrictions also lead to fixed-parameter linearity. We omit the proof for the following corollary, as it is analogous to that of Corollary 3.24.

rules, but they omit such a discussion for the primal-graph algorithm. Nevertheless, it is clear that the ideas also work for primal graphs as the authors claim. The running time of their algorithm for primal graphs is in fact $\mathcal{O}(\log(m) \cdot f(k) \cdot n^2)$, where m is the sum of all weights of optimization rules, k is the treewidth of the primal graph, n is the size of the grounding and f is a computable function. The logarithmic factor is due to the cost of arithmetic and the square is due to the fact that their algorithm also performs counting of answer sets. Here we assume the uniform-cost model, where all arithmetic operations take constant time, and are only interested in answer set existence, so this simplifies to $\mathcal{O}(f(k) \cdot n)$.

Corollary 3.25. *For every fixed connection-guarded ASP program Π there is a fixed-parameter linear algorithm that decides for every input structure $\mathcal{A}$ whether a given atom is true in an optimal answer set of $\Pi \cup \mathcal{A}$, when the parameter is the combination of the treewidth and degree of $\mathcal{A}$, if the levels and weights of all weak constraints in Π are bounded by a fixed constant.*

Clearly our FPT results concerning Brave Reasoning carry over to Answer Set Existence.

We obtained our positive results on connection-guarded ASP by parameterizing the problem by the combination of treewidth and degree, whereas guarded ASP for any fixed non-ground encoding is already FPT when parameterized by treewidth only. It is thus natural to ask whether, for fixed encodings, connection-guarded ASP is FPT when parameterized only by either treewidth or degree.

Recall that in Section 3.4 we pointed out that grounding a connection-guarded encoding together with an input structure of arbitrary degree may lead to unbounded treewidth of the grounding. It is therefore not very surprising that the degree bound is indeed necessary for obtaining fixed-parameter tractability (unless $\mathsf{P} = \mathsf{NP}$):

Theorem 3.26. *The problem of deciding whether a fixed connection-guarded program Π together with a given input structure $\mathcal{A}$ has an answer set is* NP*-hard. This even holds if the treewidth of $\mathcal{A}$ is at most three and Π contains neither disjunctions nor aggregates nor weak constraints.*

Proof. We reduce from the following NP-complete problem.

Subgraph Isomorphism

Input:	Graphs G and H
Question:	Is there a subgraph of G that is isomorphic to H?

This problem remains NP-hard even if the treewidth of both G and H is at most two (Matoušek and Thomas 1992).

Let $\langle G, H\rangle$ be an instance of Subgraph Isomorphism. We will present an ASP encoding for Subgraph Isomorphism using the signature $\sigma = \{\mathtt{vg}, \mathtt{vh}, \mathtt{eg}, \mathtt{eh}, \mathtt{bridge}, \mathtt{eq}\}$, where `vg` and `vh` are unary predicates used to represent the vertices of G and H, respectively; `eg` and `eh` are binary predicates for the respective edges; the binary predicate `bridge` is used to connect each vertex of G with a new "bridge element", which is in turn connected to each vertex of H also via the `bridge` predicate; and the binary `eq` predicate contains all pairs of equal vertices. According to this intended meaning, we define a structure $\mathcal{A}$ over σ by $\mathrm{dom}(\mathcal{A}) = \mathrm{V}(G) \cup \mathrm{V}(H) \cup \{\mathtt{b}\}$ (where `b`

```
% Guess a subgraph S of G using predicates vs/1 and es/2.
        vs(X) ← vg(X), not not_vs(X).
    not_vs(X) ← vg(X), not vs(X).
      es(X,Y) ← eg(X,Y), vs(X), vs(Y), not not_es(X,Y).
  not_es(X,Y) ← eg(X,Y), vs(X), vs(Y), not es(X,Y).
% Guess a relation representing an isomorphism using predicate iso/2.
     iso(G,H) ← vs(G), vh(H), not not_iso(G,H), bridge(G,B), bridge(B,H).
 not_iso(G,H) ← vs(G), vh(H), not iso(G,H), bridge(G,B), bridge(B,H).
% The guessed relation must be a bijection from V(S) to V(H).
          ← iso(G,H1), iso(G,H2), not eq(H1,H2),
            bridge(G,B), bridge(B,H1), bridge(B,H2).
          ← iso(G1,H), iso(G2,H), not eq(G1,G2),
            bridge(G1,B), bridge(G2,B), bridge(B,H).
  used(G) ← iso(G,H), bridge(G,B), bridge(B,H).
  used(H) ← iso(G,H), bridge(G,B), bridge(B,H).
          ← vg(G), vs(G), not used(G).
          ← vh(H), not used(H).
% The guessed relation must be an isomorphism.
 ← iso(G1,H1), iso(G2,H2), es(G1,G2), not eh(H1,H2),
   bridge(G1,B), bridge(G2,B), bridge(B,H1), bridge(B,H2).
 ← iso(G1,H1), iso(G2,H2), eh(H1,H2), not es(G1,G2),
   bridge(G1,B), bridge(G2,B), bridge(B,H1), bridge(B,H2).
```

Listing 3.6: An encoding of Subgraph Isomorphism in connection-guarded ASP

is a new element), $\mathtt{vg}^{\mathcal{A}} = V(G)$, $\mathtt{vh}^{\mathcal{A}} = V(H)$, $\mathtt{eg}^{\mathcal{A}} = E(G)$, $\mathtt{eh}^{\mathcal{A}} = E(H)$, $\mathtt{bridge}^{\mathcal{A}} = \{(g,\mathtt{b}), (\mathtt{b},h) \mid g \in V(G),\ h \in V(H)\}$ and $\mathtt{eq}^{\mathcal{A}} = \{(v,v) \mid v \in V(G) \cup V(H)\}$. Note that for every pair (x,y) in $\mathtt{eg}^{\mathcal{A}}$ or $\mathtt{eh}^{\mathcal{A}}$ there is also (y,x) in $\mathtt{eg}^{\mathcal{A}}$ or $\mathtt{eh}^{\mathcal{A}}$, respectively, since the graphs are undirected. The connection-guarded program in Figure 3.6 encodes Subgraph Isomorphism.[9] It is easy to verify that this encoding is correct, so H is isomorphic to a subgraph of G if and only if $\Pi \cup \mathcal{A}$ has an answer set.

The treewidth of $\mathcal{A}$ is the maximum of the treewidth of G and of H plus one: Given tree decompositions $\mathcal{T}_G$ and $\mathcal{T}_H$ of G and H, respectively, we can obtain a tree decomposition of $\mathcal{A}$ by taking the disjoint union of $\mathcal{T}_G$ and $\mathcal{T}_H$, adding the bridge element b to every bag and drawing an edge between an arbitrary node from $\mathcal{T}_G$ and an arbitrary node from $\mathcal{T}_H$. □

[9]In practice, we could simplify this encoding substantially by using convenient language constructs provided by ASP systems. For the purpose of this proof, we use our rather restrictive base language. Moreover, note that the positive body of many rules contains atoms whose only purpose is to make the rules connection-guarded. Such redundant atoms could be omitted in practice.

Clearly hardness for Answer Set Existence carries over to Brave Reasoning.

Since the answer set existence problem is in NP when Π contains neither disjunctions nor weak constraints and only contains a restricted form of aggregates called *convex aggregates*, we immediately get a completeness result from Theorem 3.26. (We omit a discussion of convex aggregates and refer to works by Liu and Truszczyński (2006) and Alviano and Faber (2013) for this.)

Corollary 3.27. *It is NP-complete to decide, for a fixed connection-guarded program Π without disjunctions or weak constraints and where all aggregates are convex, whether Π together with a given input structure $\mathcal{A}$ has an answer set. Hardness even holds if the treewidth of $\mathcal{A}$ is at most three and Π contains no aggregates.*

ASP solving for connection-guarded programs thus stays hard if only the treewidth of the input is bounded. In fact our result not only rules out fixed-parameter tractable algorithms but even polynomial-time algorithms when we consider the treewidth of the input as a constant (unless $\mathsf{P} = \mathsf{NP}$).

By Corollary 3.27, the answer set existence problem is NP-complete for a fixed connection-guarded program without disjunctions, weak constraints and non-convex aggregates when the treewidth of the input is bounded. Since the problem is also NP-complete if the treewidth is unbounded, the bound on the treewidth offers no advantage at all in this case.

If we allow disjunctions, weak constraints and arbitrary aggregates, answer set existence for fixed programs becomes Δ_3^{P}-complete in general, whereas we have shown in Theorem 3.26 that the problem is NP-hard when the program is connection-guarded and the input has bounded treewidth. Since this is only a hardness result, there might still be hope for disjunctive ASP with arbitrary aggregates that bounded treewidth lowers the complexity by one level of the polynomial hierarchy. However, we consider this unlikely and we suspect that answer set existence for fixed connection-guarded programs is Δ_3^{P}-complete for bounded treewidth (and Σ_2^{P}-complete when weak constraints are not allowed). Since we are not aware of any problems that have been shown to be complete for these classes on instances of bounded treewidth, we leave this as an open question.

We now prove that the degree alone is also not sufficient for obtaining fixed-parameter tractability, even if the fixed program is guarded.

Theorem 3.28. *It is Σ_2^{P}-complete to decide for a fixed guarded program Π and a given input structure $\mathcal{A}$ whether $\Pi \cup \mathcal{A}$ has an answer set even if the degree of $\mathcal{A}$ is at most 15.*

Proof. Membership follows from the general case. For hardness, we present a reduction from the well-known Σ_2^{P}-complete problem Qsat$_2$. We are given a formula $\exists x_1 \cdots \exists x_k \forall y_1 \cdots \forall y_\ell\, \varphi$, where φ is a formula in 3-DNF, and the question is whether there are truth values for the x variables such that for all truth values for the y variables φ is true.

We may assume that every variable occurs at most three times in φ. To see this, first observe that, for each sequence of variables $z_1, \dots, z_m$, saying that two variables in this sequence have different truth values is equivalent to saying that (a) some variable z_i is false but z_{i+1} is true, or (b) z_m is false but z_1 is true. With this in mind, we obtain a formula φ' from φ by replacing every occurrence of an (either existentially or universally quantified) variable z by a new variable z^i, where i is the number of the respective occurrence in φ. (That is, the first occurrence of z in φ is replaced by z_1, the second by z_2, and so on.) To establish the connections between the copies of an old variable, we observe that the following statements are equivalent:

1. There are truth values for the x variables such that, for all truth values for the y variables, φ is true.
2. There are truth values for the x variables such that, for all truth values for the y variables and for all truth values for the new copies, the following holds: If every old variable z has the same truth value as all of its copies, then φ' is true.
3. There are truth values for the x variables such that, for all truth values for the y variables and for all truth values for the new copies, the following holds: The formula φ' is true or, for some old variable z with copies $z^1, \dots, z^m$, two variables in the sequence $z, z^1, \dots, z^m$ have different truth values.
4. There are truth values for the x variables such that, for all truth values for the y variables and for all truth values for the new copies, the following formula is true, where $\operatorname{Var}(\varphi)$ denotes the variables occurring in φ:

$$\varphi' \vee \bigvee_{z \in \operatorname{Var}(\varphi) \text{ with } m \text{ copies}} \big((\neg z \wedge z^1) \vee (\neg z^1 \wedge z^2) \vee \dots \vee (\neg z^{m-1} \wedge z^m) \vee (\neg z^m \wedge z) \big)$$

Thus we obtain an equivalent formula where each variable occurs at most three times. (This result has also been shown in a paper by Peters (2017).)

We can now use the same ASP encoding as in the proof of Theorem 3.20, but we need to choose a slightly different input structure because the domain elements $\top$ and $\bot$ from that construction have unbounded degree. Recall that the old construction puts an element $\langle p_1, p_2, p_3, q_1, q_2, q_3 \rangle$ into $\mathtt{term}^{\mathcal{A}}$ for each disjunct in φ and that some p_i or q_j may be $\top$ or $\bot$ in order to represent the equivalent term $p_1 \wedge p_2 \wedge p_3 \wedge \neg q_1 \wedge \neg q_2 \wedge \neg q_2$. The only thing that matters for $\top$ and $\bot$ is that they are always interpreted as true and false, respectively, which the old construction ensures by putting them in $\mathtt{verum}^{\mathcal{A}}$ and $\mathtt{falsum}^{\mathcal{A}}$, respectively. We can thus just use a certain number of copies of $\top$ and $\bot$ such that every copy occurs exactly once in $\mathtt{term}^{\mathcal{A}}$ and every copy is in the respective `verum`$^{\mathcal{A}}$ or `falsum`$^{\mathcal{A}}$ relation. Clearly this reduction to ASP is still correct. The degree of $\mathcal{A}$ is at most 15 because every domain element has at most five neighbors in each tuple of `term`$^{\mathcal{A}}$ and every variable occurs in at most three tuples. □

3.6 Discussion

In our investigation of the effect of grounding on the treewidth, we rely on the rather primitive notion of grounding from Definition 3.3. State-of-the-art grounders, on the other hand, produce groundings whose primal graphs are generally subgraphs of the output of our transductions. However, since degree and treewidth of a graph can only decrease for a subgraph, our results apply also to state-of-the-art grounders.

Moreover, state-of-the-art grounders are capable of solving problems without needing to call an ASP solver if the program has an answer set that is a deterministic consequence of the input, i.e., if no non-deterministic guessing is involved. This is the case, for instance, for Horn programs (that is, ASP programs without negation, disjunction and aggregates). Our notion of grounding, on the other hand, assumes that the grounder does not propagate deterministic consequences and thus cannot solve such simple problems by itself. This is in fact a reasonable assumption: The question we are concerned with in this work is which form a non-ground rule may have so it does not "destroy" bounded treewidth of the input. Any encoding can be made "nasty" in the sense that a grounder cannot solve the problem, namely by forcing atoms to be guessed. This prevents the grounder from eliminating atoms from rule bodies, and it does not change the form of rules.

Parts of this chapter have been published in the paper by Bliem et al. (2017). That paper contains a preliminary version of the transduction for connection-guarded ASP programs. The current work additionally contains the result on guarded ASP, improves some methodological details of the existing result on connection-guarded ASP, and contains a complexity analysis as well as a comparison to related formalisms such as MSO. Moreover, the transductions in this chapter can be considered more appealing than the one in that paper because they exactly produce the (incidence graph of the) primal graph of the grounding, whereas the transduction in the paper by Bliem et al. (2017) produces a supergraph. Thus the transductions in this chapter reproduce the grounding process more accurately. Finally, in contrast to the work of Bliem et al. (2017), the results of the current chapter also apply to programs containing weak constraints and aggregates.

Related Work

Investigations on treewidth for ASP solving. There have been some investigations concerning treewidth in the context of ASP. Beside parameterized complexity results (Gottlob, Pichler and Wei 2010a; Pichler et al. 2014) for ground programs, there was also work on tree-decomposition-based dynamic programming algorithms (Jakl, Pichler and Woltran 2009) and their implementations (Morak et al. 2010; Fichte et al. 2017). However, in contrast to the current work, most studies of treewidth in ASP solving only considered the ground case.

Tree decompositions have been applied in the context of non-ground ASP for rule decomposition techniques (Bichler, Morak and Woltran 2017; Bichler, Morak and Woltran 2016). The goal of this, however, is improving efficiency without explicitly aiming at fixed-parameter tractability. Hence those efforts go in a different direction than the current work.

FPT classification tools. The FPT result in Theorem 3.22 (or Theorem 3.23) has the side effect that (connection-) guarded ASP can also serve as a tool for establishing that a problem is fixed-parameter tractable when parameterized by treewidth (or by the combination of treewidth and degree). Using our ASP classes in this way is similar to a very established technique for classifying a problem parameterized by treewidth as FPT: If the problem can be expressed in MSO, then by Courcelle's theorem (Theorem 2.19) the FPT membership follows. It is therefore a natural question how our ASP classes relate to MSO.

It is well known that MSO model checking, just as first-order model checking, is PSPACE-complete (Stockmeyer 1974; Vardi 1982), and it is PSPACE-hard even if the input structure is fixed and contains only two domain elements (Kreutzer 2012). In contrast, we know that we cannot express problems harder than Δ_3^P in ASP unless the polynomial hierarchy collapses to the second level. Hence there are problems that can be expressed using MSO but not in our ASP classes.

Nevertheless, there are also problems that can be expressed in our ASP classes but not in MSO. One of the reasons is that our ASP classes provide us with aggregates and weak constraints. Additionally, in connection-guarded ASP we intuitively have the possibility to define relations of arity greater than one that are not already present in the input explicitly. (For instance, in the encoding for Subgraph Isomorphism we derived a binary relation with the `iso` predicate.) The expressive power of MSO and either of our ASP classes is therefore incomparable.

In fact, connection-guarded ASP allows us to express problems that cannot be expressed with MSO even if we disallow weak constraints and aggregates (unless $P = NP$): As we have seen in Theorem 3.26, answer set existence for fixed connection-guarded programs is NP-hard for instances of bounded treewidth even if there are neither aggregates nor weak constraints. But by Courcelle's theorem, a reduction to MSO would imply FPT membership.

Of course, the fact that connection-guarded ASP allows us to define some problems that we cannot define with MSO is only of limited significance because connection-guarded

ASP allows us to obtain FPT results when the parameter is the combination of treewidth and degree, whereas we only need treewidth as the parameter for an FPT result using MSO. Still, the class of connection-guarded programs may be of interest for algorithmic purposes because it allows us to classify a problem as FPT when parameterized by treewidth plus degree, as we have shown in Theorem 3.23. We are not aware of any extensions of MSO that allow us to obtain new FPT results using this more restrictive parameter. Hence our result may lead to an extension of MSO that can be used for classifying problems as FPT when the parameter is treewidth + degree. This is subject of future work.

MSO has been extended in several ways to increase its expressive power while retaining FPT (or at least XP) membership of model checking when parameterized by treewidth. (For overviews, see the works of Knop et al. (2017) and Langer et al. (2014).)

Specifically of interest to us is the extension *EMSO* (extended monadic second-order logic), which makes it possible to specify a linear function $\alpha(z_1, \dots, z_k)$ along with an MSO formula φ with free variables $X_1, \dots, X_k$. In addition to deciding whether a structure is a model of the formula, it is now also possible to find the maximum or minimum of $\alpha(|X_1|, \dots, |X_k|)$ for any value of the free variables such that φ is satisfied. This can be done in fixed-parameter linear time (under the uniform-cost model, otherwise there is a logarithmic factor) as shown by Arnborg, Lagergren and Seese (1991). We next compare the mentioned unique features of ASP with EMSO.

Weak constraints compared to EMSO. It is easy to see that weak constraints allow us to express problems in ASP that cannot be expressed in plain MSO: For instance, given a graph G and an integer k, deciding if G admits a vertex cover of size at most k cannot be done in MSO unless k is fixed, whereas this is possible in guarded ASP using weak constraints.

The EMSO extension is an analogue to weak constraints as it allows us to define also optimization problems like this. A notable difference is that the weight and level of a weak constraint may depend on the input, whereas in EMSO the coefficients in the linear function have to be constants. As an example, consider the following problem.

Weighted Minimum Vertex Cover

Input: A graph G and a weight function $w : \mathrm{V}(G) \to \mathbb{N}^+$

Task: Compute the minimum weight of a vertex cover in G. The weight of a set S of vertices is $\sum_{v \in S} w(v)$.

This can easily be expressed in guarded ASP by the following encoding, where a vertex v is represented by $\texttt{vertex}(v, w(v))$ and an edge (x, y) by $\texttt{edge}(x, y)$:

```
s(X) ∨ ns(X) ← vertex(X,W).
             ← edge(X,Y), not s(X), not s(Y).
             ⇝ s(X), vertex(X,W).  [W@1,X]
```

In ASP the domain of input structures may contain integers, which are treated accordingly by the semantics. Since EMSO is ignorant of numbers and only considers set cardinalities, it cannot express this problem on the same class of input structures.

Weak constraints compared to EMSO with unary numbers. We can in fact express problems like Weighted Minimum Vertex Cover above in EMSO if we represent the instances as relational structures in a different way. However, this requires us to encode the weights in unary, which exponentially increases the instance size. For example, we could represent a Weighted Minimum Vertex Cover instance $\langle G, w \rangle$ as the structure $\mathcal{A}$ where $\mathrm{dom}(\mathcal{A}) = \{v, v_1, \dots, v_{w(v)} \mid v \in \mathrm{V}(G)\}$, $\texttt{vertex}^{\mathcal{A}} = \{(v, v_1), \dots, (v, v_{w(v)}) \mid v \in \mathrm{V}(G)\}$, $\texttt{edge}^{\mathcal{A}} = \mathrm{E}(G)$, and write an MSO formula expressing "S is a vertex cover and W consists of all 'weight elements' that belong to a vertex in S".

Using such a technique we can of course also write an ASP encoding for the problem such that the weights of weak constraints do not depend on the input. In the encoding above, we merely need to change the weight `W` of the weak constraint to 1.

Even if numbers are encoded in unary, weak constraints offer more expressive power than the linear functions of EMSO in general. This is because the levels of weak constraints may still depend on the input. Even if the value of each level is thus linear in the input size, "flattening" all weight specifications to a single level leads to weights of exponential size, as we have seen in Section 3.5. A possible solution would be to extend EMSO so as to support different priority levels.

A note about numbers in the domain. If the treewidth or the degree of instances is important, we must be careful when the domain of the input structures contains integers such as the weights in the previous example. The reason is that every integer occurring in an input structure $\mathcal{A}$ is an ordinary vertex in the Gaifman graph of $\mathcal{A}$. Therefore, if $\mathcal{A}$ contains facts $\texttt{weight}(a_1, 0), \dots, \texttt{weight}(a_k, 0)$ for example, then the Gaifman graph of $\mathcal{A}$ contains a vertex 0 that is adjacent to every a_i. In general, we thus lose bounded treewidth and bounded degree compared to the case where we simply omit integers from the Gaifman graph. However, we may not omit integers from the Gaifman graph because it may well be the case that, for example, we want to derive every pair of elements that have the same weight. In such a case the integers indeed play a structural rule and thus all occurrences of the same integer should correspond to a single vertex in the Gaifman graph.

To escape this predicament, we propose to evaluate the arithmetic for optimization or counting tasks (as in weak constraints, aggregates or the linear functions of EMSO) in a slightly different way than commonly defined. For each input structure $\mathcal{A}$, we additionally supply an integer assignment $\alpha : \mathrm{dom}(\mathcal{A}) \to \mathbb{Z}$. We then use this in the definition of the semantics of the respective language constructs: We modify the notion of the grounding of a weak constraint so that we do not generate ground rules of the form $\leftharpoondown\!\!\sim \beta\ [t_w @ t_l, t_1, \dots, t_m]$ but instead $\leftharpoondown\!\!\sim \beta\ [\alpha(t_w) @ \alpha(t_l), t_1, \dots, t_m]$. For aggregates, we proceed in a similar way by replacing terms t in places where integers are expected by $\alpha(t)$. This idea also works for EMSO: Instead of passing the cardinality of a set X as argument of the linear function, we use $\sum_{x \in X} \alpha(x)$. The traditional cardinality-based semantics of EMSO is then the special case where $\alpha(x) = 1$ for all domain elements x. This in fact extends EMSO so that it "gains awareness" of integers in the sense that weights can be given in the input, as it is possible in ASP. We will not pursue these modified semantics any further in this work though.

Weak constraints compared to EMSO when the input contains no numbers. We conjecture that it is possible to prove that EMSO is strictly more expressive than guarded ASP for any class of input structures that do not contain numbers. In this case there can only be a bounded number of levels in the weak constraints, and each weight is also bounded. The formulas in our MSO transduction from Section 3.3 may then allow us to translate a program from this class into an EMSO specification. This should even be possible in the presence of aggregates: We suspect that aggregates do not add expressive power to guarded ASP because the strict requirement that each rule be guarded seems to make aggregates basically useless. We leave a closer examination of this for future work.

We expect that such a proof would most likely imply that guarded ASP without weak constraints is in fact strictly less expressive than plain MSO.

Aggregates. Aggregates are a feature that is also not present in EMSO.[10] Consider the following problem, for instance, which gives a taste of the things to come in Chapter 4.

Defensive Alliance Verification

Input: A graph G together with a set $S \subseteq V(G)$

Question: Does $|N[v] \cap S| \geqslant |N[v] \setminus S|$ hold for every $v \in S$?

The following connection-guarded ASP program encodes this problem, where the instance is given via the binary predicate e for the edges and the unary predicate s for the set S.

[10]There are, however, extensions of MSO that support certain statements about cardinalities of sets like *MSO-LCC* (Szeider 2011a) or *CardMSO* (Ganian and Obdrzálek 2013). These only allow for rather limited expressions of such a sort and we leave investigations of these extensions for future work.

$\leftarrow$ `s(X), #sum{ 1,Y : e(X,Y), s(Y); -1,Y : e(X,Y),` **`not`** `s(Y) } < -1.`

Clearly this program together with an input has an answer set if and only if the instance is positive. In MSO we cannot express this problem as it is well known that MSO does not allow us to compare the cardinality of two sets, and neither does EMSO. It also does not seem obvious that this problem can be expressed in any of the known extensions of MSO. We leave further investigations for future work.

Final remarks. The considerations in this section have been rather theoretical: Even though in many cases a problem can also be expressed in (an extension of) MSO, this is mostly interesting from a theoretical perspective, whereas the actual solving performance of algorithms based on MSO model checking is usually clearly worse than that of dedicated tools (cf. Cygan et al. 2015, pp. 184–185) even though there have been considerable advances in this effort (e.g., Langer et al. 2012). Even for rather simple problems, MSO formulas can unfortunately be quite complex. For many problems in Δ_3^P, expressing a problem in one of our classes not only yields a result about its complexity by Theorems 3.22 and 3.23, but it should in most cases also give quite good performance in practice due to the efficiency of ASP systems.

CHAPTER 4

Alliance Problems in Graphs

The objective of many problems that can be modeled as graphs is finding a group of vertices that together satisfy some property. In this respect, one of the concepts that has been quite extensively studied is the notion of a defensive alliance (Kristiansen, S. M. Hedetniemi and S. T. Hedetniemi 2004; Kristiansen, S. M. Hedetniemi and S. T. Hedetniemi 2002), which is a set of vertices such that for each element v at least half of its neighbors are also in the alliance. The name "defensive alliance" stems from the intuition that the neighbors of an element v that are also in the alliance can help out in case v is attacked by its other neighbors.

Notions like this can be applied to finding groups of nations, companies or individuals that depend on each other, but also to more abstract situations like finding groups of websites that form communities (Flake et al. 2002). Another possible application for defensive alliances are computer networks, where a defensive alliance represents computers that can provide a certain desired resource; any computer in an alliance can then, with the help of its neighbors that are also in the alliance, allow access to this resource from all of its neighbors simultaneously (Haynes, S. T. Hedetniemi and Henning 2003).

The Defensive Alliance problem can be specified as follows: Given a graph G and an integer k, is there a defensive alliance S in G such that $1 \leqslant |S| \leqslant k$? It is known that this problem is NP-complete (Jamieson 2007; Jamieson, S. T. Hedetniemi and McRae 2009) However, if we restrict ourselves to trees, Defensive Alliance becomes trivial and in fact the corresponding problems for several non-trivial variants become solvable in linear time (Jamieson 2007).

There has also been some work on the parameterized complexity of alliance problems. In particular, determining whether a defensive alliance of a given (maximum) size exists is fixed-parameter tractable when parameterized by the solution size (Fernau and Raible 2007; Enciso 2009). Also structural parameters have been considered to some extent. Kiyomi and Otachi (2017) recently proved that the Defensive Alliance problem can be solved in polynomial time if the clique-width of the instances is bounded by a constant. The authors also provided an FPT algorithm when the parameter is the size of the smallest vertex cover. Moreover, Enciso (2009) showed that Defensive Alliance is fixed-parameter tractable when parameterized by *domino treewidth*, which is a parameter that is equivalent to the combination of treewidth and maximum degree. Despite these advances, the question of whether or not Defensive Alliance parameterized by treewidth is fixed-parameter tractable has so far remained open.

The primary focus of this chapter will be a natural variant of defensive alliances called *secure sets*, which have been introduced by Brigham, Dutton and S. T. Hedetniemi (2007). While defensive alliances make sure that each element of an alliance can defend itself against attacks from its neighbors, they do not account for attacks on multiple vertices at the same time. To this end, we can employ a stronger concept: A set S of vertices is secure if for each *subset* $X \subseteq S$ at least half of the vertices adjacent to some element of X are also in S. The Secure Set problem can now be stated as follows: Given a graph G and an integer k, does there exists a secure set S of G such that $1 \leqslant |S| \leqslant k$?

It is known that the problem of deciding whether a *given* set S is secure in a graph is co-NP-complete (Ho 2011), so it would not be surprising if *finding* (non-trivial) secure sets is also a very hard problem. Moreover, Abseher et al. (2015) have reduced the problem to Answer Set Programming, which proves that it is in the class Σ_2^P. Unfortunately, the exact complexity of this problem has so far remained unresolved.

As for the parameterized complexity, it has been shown by Enciso and Dutton (2008) that Secure Set can be solved in linear time if the solution size is bounded by a constant. In a paper by Ho and Dutton (2009) it has been shown that a certain variant of Secure Set becomes easy on trees, but the complexity of Secure Set parameterized by treewidth is listed as an open problem in that work and has not been settled since.

The first main contribution of this chapter is to show that Secure Set is Σ_2^P-complete. Unlike the existing co-NP-hardness proof (Ho 2011), which uses a reduction from Dominating Set, we base our proof on a reduction from a problem in the area of logic. To be specific, we first show that the canonical Σ_2^P-complete problem Qsat$_2$ can be reduced to a variant of Secure Set, where vertices can be forced to be in or out of every solution, and pairs of vertices can be specified to indicate that every solution must contain exactly one element of each such pair. In order to prove the desired complexity result, we then successively reduce this variant to the standard Secure Set problem. At the same time, we show Σ_2^P-completeness for variants of these problems where we are interested in secure sets *exactly* of a certain size. We thus complete the picture of the precise complexity of the Secure Set problem, and we also provide completeness results for variants of the problem that have already been proposed (Ho and Dutton 2009) but for which no complexity analysis has been performed so far.

The second main contribution of this chapter is a parameterized complexity analysis of Defensive Alliance and Secure Set with treewidth as the parameter. For both of these problems, the question of whether or not they are fixed-parameter tractable when parameterized by treewidth has so far been unresolved (Ho and Dutton 2009; Kiyomi and Otachi 2017). In the current chapter, we provide a negative answer to this question: We show that both problems are hard for the class W[1], which rules out fixed-parameter tractable algorithms under commonly held complexity-theoretic assumptions. Beside these parameterized hardness results, we also give upper bounds by showing that Defensive Alliance and Secure Set are in the class XP, which means that they can be solved in polynomial time on instances of bounded treewidth. We do so by providing an algorithm for Secure Set (where the degree of the polynomial depends on the treewidth) and presenting a reduction from Defensive Alliance to Secure Set that preserves bounded treewidth, which allows us to employ this algorithm also for Defensive Alliance. Finally, we present a positive result for the co-NP-complete problem of checking whether a given set of vertices is secure in a graph: We provide an algorithm that solves the problem in linear time for graphs of bounded treewidth.

This chapter is organized as follows: First we introduce our problems of interest in Section 4.1. Then we analyze the complexity of Secure Set in Section 4.2, where we show that this problem, along with several variants, is Σ_2^P-complete. We discuss the relationship between Secure Set and Defensive Alliance in Section 4.3 and we also present a reduction from Defensive Alliance to Secure Set. In Section 4.4, we show that both problems are W[1]-hard when parameterized by treewidth. This is followed by our positive results in Section 4.5, where we present an FPT algorithm for Secure Set Verification when parameterized by treewidth, as well as an algorithm for Secure Set that runs in polynomial time if the treewidth is bounded by a constant. Section 4.6 concludes the chapter with a discussion of our results and related work.

4.1 Secure Sets and Defensive Alliances

We start with the definition of secure sets, which have the following intuition: If we consider a set S of vertices as "good" vertices and all other vertices as "bad" ones, then S being secure means that each subset of S has at least as many "good" neighbors as "bad" neighbors.

Definition 4.1. Given a graph G, a set $S \subseteq V(G)$ is *secure in G* if for each $X \subseteq S$ it holds that $|N[X] \cap S| \geqslant |N[X] \setminus S|$.

We often write "S is secure" instead of "S is secure in G" if it is clear from the context which graph is meant. By definition, the empty set is secure in any graph. Thus, in the

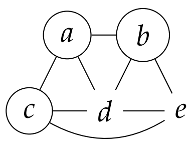


Figure 4.1: A graph with a minimum non-empty secure set indicated by circled vertices

following decision problems we ask for secure sets of size at least 1. The following is one of the main problems for this chapter:

Secure Set

Input: A graph G and an integer k with $1 \leqslant k \leqslant |V(G)|$

Question: Is there a set $S \subseteq V(G)$ with $1 \leqslant |S| \leqslant k$ that is secure?

Figure 4.1 shows a graph together with a minimum non-empty secure set $S = \{a, b, c\}$. Observe that for any $X \subseteq S$ the condition $|N[X] \cap S| \geqslant |N[X] \setminus S|$ is satisfied.

Defensive alliances are very similar, but for checking whether a set is a defensive alliance, we do not consider all subsets but only the singletons.

Definition 4.2. Given a graph G, a set $S \subseteq V(G)$ is a *defensive alliance in* G if for each $v \in S$ it holds that $|N[v] \cap S| \geqslant |N[v] \setminus S|$.

For example, in Figure 4.1, the set $S = \{a, b\}$ is a defensive alliance as $|N[v] \cap S| \geqslant |N[v] \setminus S|$ holds for each $v \in S$, but S is not a secure set, since for $X = S$ it holds that $|N[X] \cap S| < |N[X] \setminus S|$. Conversely, every secure set is also a defensive alliance.

We define the Defensive Alliance problem analogously to Secure Set.

Next we introduce several variants of Secure Set that we require in our proofs. For every such problem, we also implicitly define an analogous variant of Defensive Alliance.

The problem Secure Set$^{\mathrm{F}}$ generalizes Secure Set by designating some "forbidden" vertices that may never be in any solution. This variant can be formalized as follows:

Secure Set$^{\mathrm{F}}$

Input: A graph G, an integer k and a set $V_\square \subseteq V(G)$

Question: Does there exist a set $S \subseteq V(G) \setminus V_\square$ with $1 \leqslant |S| \leqslant k$ that is secure?

Secure Set$^{\mathrm{FN}}$ is a further generalization that, in addition, allows "necessary" vertices to be specified that must occur in every solution.

Secure Set$^{\mathrm{FN}}$

Input: A graph G, an integer k, a set $V_\square \subseteq \mathrm{V}(G)$ and a set $V_\triangle \subseteq \mathrm{V}(G)$

Question: Does there exist a set $S \subseteq \mathrm{V}(G) \setminus V_\square$ with $V_\triangle \subseteq S$ and $1 \leqslant |S| \leqslant k$ that is secure?

Finally, we introduce the generalization Secure Set$^{\mathrm{FNC}}$. Here we may state pairs of "complementary" vertices where each solution must contain exactly one element of every such pair.

Secure Set$^{\mathrm{FNC}}$

Input: A graph G, an integer k, a set $V_\square \subseteq \mathrm{V}(G)$, a set $V_\triangle \subseteq \mathrm{V}(G)$ and a set $C \subseteq \mathrm{V}(G)^2$

Question: Does there exist a set $S \subseteq \mathrm{V}(G) \setminus V_\square$ with $V_\triangle \subseteq S$ and $1 \leqslant |S| \leqslant k$ that is secure and, for each pair $(a, b) \in C$, contains either a or b?

For our results on the parameters treewidth, we need a way to represent the structure of a Secure Set$^{\mathrm{FNC}}$ instance by a graph that augments G with the information in C:

Definition 4.3. Let I be a Secure Set$^{\mathrm{FNC}}$ instance, let G be the graph in I and let C the set of complementary vertex pairs in I. By the *primal graph* of I we mean the undirected graph G' with $\mathrm{V}(G') = \mathrm{V}(G)$ and $\mathrm{E}(G') = \mathrm{E}(G) \cup C$.

When we speak of the treewidth of an instance of Secure Set, Secure Set$^{\mathrm{F}}$ or Secure Set$^{\mathrm{FN}}$, we mean the treewidth of the graph in the instance. For an instance of Secure Set$^{\mathrm{FNC}}$, we mean the treewidth of the primal graph.

While the Secure Set problem asks for secure sets of size *at most k*, we also consider the Exact Secure Set problem that concerns secure sets of size *exactly k*. Analogously, we also define exact versions of the three generalizations of Secure Set presented above.

When the task is not to find secure sets but to verify whether a given set is secure, the following problem is of interest:

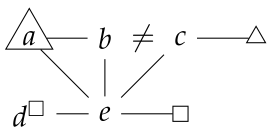


Figure 4.2: Illustration of forbidden, necessary and complementary vertices

Secure Set Verification

Input: A graph G and a set $S \subseteq \mathrm{V}(G)$

Question: Is S secure in G?

This problem is known to be co-NP-complete (Ho 2011). It is easy to see that the analogous problem of verifying whether a given set is a defensive alliance can be solved in polynomial time.

In the figures of this chapter, we often indicate necessary vertices by means of a triangular node shape, and forbidden vertices by means of either a square node shape or a superscript square in the node name. If two vertices are complementary, we often express this in the figures by putting a $\neq$ sign between them. For example, in Figure 4.2, the vertices b and c are complementary and occur in no solution together; a and the "anonymous" vertex adjacent to c are necessary and occur in every solution; $d^\square$ and the "anonymous" vertex adjacent to e are forbidden and occur in no solution. In this figure, the unique minimum non-empty secure set satisfying the conditions of forbidden, necessary and complementary vertices consists of a, b and the "anonymous" necessary vertex adjacent to c.

The following terminology will be helpful: We often use the terms *attackers* and *defenders* of a subset X of a secure set candidate S. By these we mean the sets $N[X] \setminus S$ and $N[X] \cap S$, respectively. To show that a subset X of a secure set candidate S is *not* a witness to S being insecure, we sometimes employ the notion of a *defense* of X w.r.t. S, which assigns to each attacker a dedicated defender: If we are able to find an injective mapping $\mu : N[X] \setminus S \to N[X] \cap S$, then obviously $|N[X] \setminus S| \leqslant |N[X] \cap S|$, and we call μ a *defense* of X w.r.t. S. Given such a defense μ, we say that a defender d *repels* an attack on X by an attacker a whenever $\mu(a) = d$. Consequentially, when we say that a set of defenders D *can repel* attacks on X from a set of attackers A, we mean that there is a defense that assigns to each element of A a dedicated defender in D.

To warm up, we make some easy observations that we will use in our proofs. First, for every set R consisting of a majority of neighbors of a vertex v, whenever v is in a defensive alliance, also some element of R must be in it:

Observation 4.4. *Let S be a defensive alliance in a graph, let $v \in S$ and let $R \subseteq N(v)$. If $|R| > \frac{1}{2}N[v]$, then S contains an element of R.*

Proof. Suppose that $|R| > \frac{1}{2}|N[v]|$ and S contains no element of R. Since all elements of R attack v, $|N[v] \setminus S| > \frac{1}{2}|N[v]|$. Hence $2|N[v] \setminus S| > |N[v]| = |N[v] \cap S| + |N[v] \setminus S|$, and we obtain the contradiction $|N[v] \setminus S| > |N[v] \cap S|$. □

Clearly this also holds if S is a secure set because every secure set is also a defensive alliance.

Next, if one half of the neighbors of an element v of a defensive alliance attacks v, then the other half of the neighbors must be in the defensive alliance:

Observation 4.5. *Let S be a defensive alliance in a graph, let $v \in S$ and let $N(v)$ be partitioned into two equal-sized sets A, D. If $A \cap S = \emptyset$, then $D \subseteq S$.*

Proof. Since $N(v)$ is partitioned into A and D such that $A \cap S = \emptyset$, we get $N(v) \cap S = D \cap S$. If some element of D is not in S, then $D \cap S \subset D$ and $A \subset N[v] \setminus S$. By $|D| = |A|$, we get $|D \cap S| + 2 \leqslant |N[v] \setminus S|$. From $|N[v] \cap S| = 1 + |N(v) \cap S| = 1 + |D \cap S|$ we now obtain the contradiction $|N[v] \cap S| < |N[v] \setminus S|$. □

In particular, if half of the neighbors of v are forbidden, then v can only be in a defensive alliance if all non-forbidden neighbors are also in the defensive alliance.

4.2 Complexity of the Secure Set Problem

This section is devoted to proving the following theorem:

Theorem 4.6. Secure Set *and* Exact Secure Set *are both Σ_2^P-complete.*

We prove hardness by providing a chain of polynomial reductions from Qsat$_2$ to the problems under consideration. Membership is easy to see and in fact follows from the reduction from Secure Set to Answer Set Programming by Abseher et al. (2015). It is actually quite easy to generalize that algorithm so that it also supports forbidden, necessary and complementary vertices. As a consequence, clearly also all problem variants that we consider in this work are Σ_2^P-complete.

4.2.1 Hardness of Secure Set with Forbidden, Necessary and Complementary Vertices

Lemma 4.7. Secure Set$^{\text{FNC}}$ *and* Exact Secure Set$^{\text{FNC}}$ *are Σ_2^P-hard.*

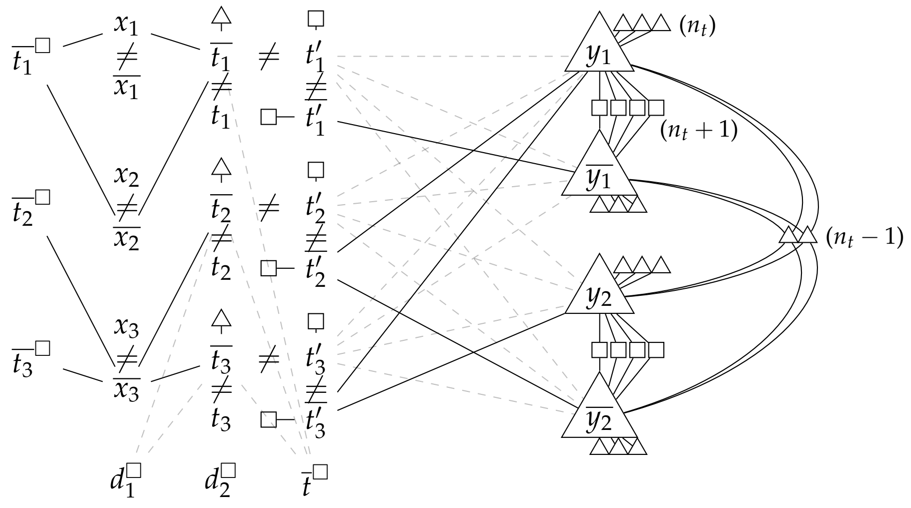


Figure 4.3: Graph corresponding to the Qsat$_2$ formula $\exists x_1 \exists x_2 \exists x_3\ \forall y_1 \forall y_2\ \big((\neg x_1 \wedge x_2 \wedge y_1) \vee (x_3 \wedge \neg y_1 \wedge y_2) \vee (x_3 \wedge \neg y_1 \wedge \neg y_2)\big)$. To avoid clutter, we omit labels for the vertices from $Y_\triangle$, $Y'_\triangle$, $Y_\square$, $\overline{T}_\triangle$, $T'_\square$ and $\overline{T'}_\square$, and we draw some edges in a dashed style.

Proof. We reduce from Qsat$_2$ to Secure Set$^{\text{FNC}}$. This also proves Σ_2^{P}-hardness for the exact variant because our reduction makes sure that all solutions of the Secure Set$^{\text{FNC}}$ instance have the same size. We are given a quantified Boolean formula $\varphi = \exists x_1 \dots \exists x_{n_x} \forall y_1 \dots \forall y_{n_y} \psi$, where ψ is in DNF and contains n_t terms of exactly three literals. We assume that no term contains both a variable and its complement (since such a term can never be satisfied) and that each term contains at least one universally quantified variable (since φ is trivially true otherwise).

We construct an instance $(G, k, V_\triangle, V_\square, C)$ of Secure Set$^{\text{FNC}}$ in the following. For an illustration, see Figure 4.3. We define a graph G by choosing the union of the following sets as $\mathrm{V}(G)$:

$$\begin{aligned}
X &= \{x_1, \dots, x_{n_x}\} & \overline{X} &= \{\overline{x_1}, \dots, \overline{x_{n_x}}\} \\
Y &= \{y_1, \dots, y_{n_y}\} & \overline{Y} &= \{\overline{y_1}, \dots, \overline{y_{n_y}}\} \\
Y_\triangle &= \{y_{i,j}^\triangle, \overline{y_{i,j}}^\triangle \mid 1 \leqslant i \leqslant n_y,\ 1 \leqslant j \leqslant n_t\} & Y'_\triangle &= \{y_j^\triangle \mid 1 \leqslant j \leqslant n_t - 1\} \\
Y_\square &= \{y_{i,j}^\square \mid 1 \leqslant i \leqslant n_y,\ 1 \leqslant j \leqslant n_t + 1\} & H &= \{d_1^\square, d_2^\square, \overline{t}^\square\} \\
T &= \{t_1, \dots, t_{n_t}\} & \overline{T} &= \{\overline{t_1}, \dots, \overline{t_{n_t}}\} \\
\overline{T}_\square &= \{\overline{t_1}^\square, \dots, \overline{t_{n_t}}^\square\} & \overline{T}_\triangle &= \{\overline{t_1}^\triangle, \dots, \overline{t_{n_t}}^\triangle\} \\
T' &= \{t'_1, \dots, t'_{n_t}\} & \overline{T'} &= \{\overline{t'_1}, \dots, \overline{t'_{n_t}}\} \\
T'_\square &= \{t_1'^\square, \dots, t_{n_t}'^\square\} & \overline{T'}_\square &= \{\overline{t'_1}^\square, \dots, \overline{t'_{n_t}}^\square\}
\end{aligned}$$

Next we define the set of edges. In the following, whenever we sloppily speak of a literal in the context of the graph G, we mean the vertex corresponding to that literal (i.e., some x_i, $\overline{x_i}$, y_i or $\overline{y_i}$), and we proceed similarly for terms. Furthermore, when we are dealing with a (vertex corresponding to a) literal l, then $\bar{l}$ shall denote the (vertex corresponding to the) complement of l. For any term t_i, let $L_X(t_i)$ and $L_Y(t_i)$ denote the set of existentially and universally quantified literals, respectively, in t_i.

$$\begin{aligned}
\mathrm{E}(G) = {} & \Big\{(\overline{t_i}, \overline{t}^{\square}), (\overline{t_i}, \overline{t_i}^{\triangle}), (t'_i, t'^{\square}_i), (\overline{t'_i}, \overline{t'_i}^{\square}) \mid t_i \in T\Big\} \cup \big(T' \times (Y \cup \overline{Y})\big) \\
& \cup \Big\{(\bar{l}, \overline{t_i}^{\square}), (\bar{l}, \overline{t_i}) \mid t_i \in T,\ l \in L_X(t_i)\Big\} \\
& \cup \Big\{(\bar{l}, \overline{t'_i}) \mid t_i \in T,\ l \in L_Y(t_i)\Big\} \\
& \cup \Big\{(d^{\square}_1, \overline{t_i}) \mid t_i \in T,\ |L_X(t_i)| \leqslant 1\Big\} \cup \Big\{(d^{\square}_2, \overline{t_i}) \mid t_i \in T,\ L_X(t_i) = \emptyset\Big\} \\
& \cup \Big\{(y_i, y^{\triangle}_{i,j}), (\overline{y_i}, \overline{y_{i,j}}^{\triangle}) \mid 1 \leqslant i \leqslant n_y,\ 1 \leqslant j \leqslant n_t\Big\} \\
& \cup \Big\{(y_i, y^{\square}_{i,j}), (\overline{y_i}, y^{\square}_{i,j}) \mid y^{\square}_{i,j} \in Y_{\square}\Big\} \cup \Big(Y'_{\triangle} \times (Y \cup \overline{Y})\Big)
\end{aligned}$$

Finally, we define

$$\begin{aligned}
V_{\triangle} &= Y \cup \overline{Y} \cup Y_{\triangle} \cup Y'_{\triangle} \cup \overline{T}_{\triangle}, \quad V_{\square} = Y_{\square} \cup \overline{T}_{\square} \cup T'_{\square} \cup \overline{T'}_{\square} \cup H, \\
C &= \{(x_i, \overline{x_i}) \mid 1 \leqslant i \leqslant n_x\} \cup \{(t_i, \overline{t_i}), (\overline{t_i}, t'_i), (t'_i, \overline{t'_i}) \mid 1 \leqslant i \leqslant n_t\},
\end{aligned}$$

and $k = |V_{\triangle}| + n_x + 2n_t$.

The following observations are crucial: Elements of $X \cup \overline{X}$ are only adjacent to vertices from $\overline{T}_{\square}$ (which are forbidden) and $\overline{T}$. For any i, each element of $X \cup \overline{X}$ is adjacent to $\overline{t_i}^{\square} \in \overline{T}_{\square}$ if and only if it is adjacent to $\overline{t_i} \in \overline{T}$. Furthermore, for any i, j, if x_i or $\overline{x_i}$ is adjacent to $\overline{t_j}$, then setting the variable x_i to true or false, respectively, falsifies the term t_j. Finally, for any i, j, if y_i or $\overline{y_i}$ is adjacent to $\overline{t'_j}$, then setting the variable y_i to true or false, respectively, falsifies the term t_j.

The intuition is that the complementary pairs $(x_i, \overline{x_i})$ guess a truth assignment to the existentially quantified variables. We now need to check if such a truth assignment has the property that the formula ψ is true for all extensions of this assignment to the universally quantified variables. Trying out all these extensions amounts to going through all subsets of a solution candidate and comparing the numbers of attackers and defenders.

To illustrate, let S be a solution candidate (i.e., a set of vertices) and suppose S satisfies the conditions on forbidden, necessary and complementary vertices. We denote the truth assignment to $x_1, \dots, x_{n_x}$ encoded in S by I_S. Moreover, let R be a subset of S containing either y_j or $\overline{y_j}$ for each universally quantified variable y_j. We denote the extension of I_S to $y_1, \dots, y_{n_y}$ encoded in R by $I_{S,R}$. For any term t_i that is falsified already by I_S, the vertex t'_i attacks all vertices y_j and $\overline{y_j}$. At the same time, for any term

t_i that is not falsified by I_S, the vertex $\overline{t'_i}$ attacks y_j or $\overline{y_j}$ if setting the variable y_j to true or false, respectively, falsifies t_i. Hence, the number of attacks from vertices of the form t'_i or $\overline{t'_i}$ on R is exactly the number of terms that are falsified by $I_{S,R}$. With the help of the vertices in $Y'_\triangle$, we can afford up to $n_t - 1$ falsified terms, but if we falsify all n_t terms, then R is a witness that S is not secure.

The Secure Set$^{\text{FNC}}$ instance $(G, k, V_\triangle, V_\square, C)$ can be constructed in time polynomial in the size of φ. We claim that φ is true if and only if $(G, k, V_\triangle, V_\square, C)$ is a positive instance of Secure Set$^{\text{FNC}}$.

"Only if" direction. If φ is true, then there is an assignment I to $x_1, \dots, x_{n_x}$ such that, for all assignments extending I to $y_1, \dots, y_{n_y}$, some term in ψ is satisfied. We define a set

$$\begin{aligned} S = V_\triangle &\cup \{x_i \in X \mid I(x_i) = \text{true}\} \cup \{\overline{x_i} \in \overline{X} \mid I(x_i) = \text{false}\} \\ &\cup \{\overline{t_i} \in \overline{T},\ \overline{t'_i} \in \overline{T'} \mid \text{there is some } l \in L_X(t_i) \text{ such that } I \not\models l\} \\ &\cup \{t_i \in T,\ t'_i \in T' \mid \text{for all } l \in L_X(t_i) \text{ it holds that } I \models l\}. \end{aligned}$$

We observe that $|S| = k$, $V_\square \cap S = \emptyset$, $V_\triangle \subseteq S$, and that for any $(a, b) \in C$ it holds that $a \in S$ if and only if $b \notin S$. By construction, whenever some element of $X \cup \overline{X}$ is in S, then all its neighbors in $\overline{T}$ are in S; and whenever some $\overline{t_i}$ is in S, then some neighbor of $\overline{t_i}$ in $X \cup \overline{X}$ is in S.

We claim that S is a secure set in G. Let R be an arbitrary subset of S. We show that R has at least as many defenders as attackers by constructing a defense, which assigns to each attacker of R a dedicated defender in $N[R] \cap S$. We distinguish cases regarding the origins of the attacks on R.

- We repel each attacker $\overline{t_i}^\square \in \overline{T}_\square$ using $\overline{t_i}$. Since $\overline{t_i}^\square$ attacks R, R must contain some element of $X \cup \overline{X}$ that is adjacent to $\overline{t_i}^\square$ and thus also to $\overline{t_i}$, so $\overline{t_i} \in N[R] \cap S$.
- Each attacker from $X \cup \overline{X} \cup \{d_1^\square, d_2^\square\}$ is adjacent to some $\overline{t_i} \in \overline{T} \cap R$. We repel that attacker using $\overline{t_i}^\triangle$, which is adjacent to $\overline{t_i}$. Note that it cannot be the case that $\overline{t_i}$ is attacked by more than one vertex in $X \cup \overline{X} \cup \{d_1^\square, d_2^\square\}$ because $\overline{t_i}$ has exactly two neighbors from that set and would not be in S if neither of these neighbors was in S.
- If $\overline{t}^\square$ attacks R, then it attacks at least one element of $\overline{T} \cap R$, which is adjacent to some element of $X \cup \overline{X}$ that is also in S. We repel $\overline{t}^\square$ using any such element of $X \cup \overline{X}$.
- Any attack from some $\overline{t_i} \in \overline{T}$ on R must be on $\overline{t_i}^\triangle$. Since $\overline{t_i} \notin S$, $\overline{t_i}^\triangle$ is not consumed for repelling an attack on $\overline{t_i}$, so we repel $\overline{t_i}$ with $\overline{t_i}^\triangle$.
- If some $t_i'^\square \in T'_\square$ attacks R (by attacking t'_i), we repel $t_i'^\square$ with t'_i.

- Analogously, we repel each attacker $\overline{t'_i}^{\square} \in \overline{T'}_{\square}$ with $\overline{t'_i}$.
- If, for some i with $1 \leqslant i \leqslant n_y$, the vertices $y^{\square}_{i,j}$ for $1 \leqslant j \leqslant n_t + 1$ attack R, then we distinguish the following cases: If y_i is in R, then the adjacent vertices $y^{\triangle}_{i,j}$ for $1 \leqslant j \leqslant n_t$ are in the neighborhood of R, too. We then repel each $y^{\square}_{i,j}$ with $y^{\triangle}_{i,j}$ for $1 \leqslant j \leqslant n_t$, and we repel $y^{\square}_{i,n_t+1}$ with y_i. Otherwise, $\overline{y_i}$ is in R, and we proceed symmetrically using $\overline{y_{i,j}}^{\triangle}$ and $\overline{y_i}$ as dedicated defenders.
- In order to account for attacks from $T' \cup \overline{T'}$ on R, we distinguish two cases.
  - If, for some i with $1 \leqslant i \leqslant n_y$, both y_i and $\overline{y_i}$ are in R, then, in the step before, we have repelled each $y^{\square}_{i,j}$ with the respective $y^{\triangle}_{i,j}$ or y_i, but all $\overline{y_{i,j}}^{\triangle}$ are still free. These vertices can repel all attacks from $T' \cup \overline{T'}$, as there are at most n_t such attacks.
  - Otherwise we show that there are at most $n_t - 1$ attacks from $T' \cup \overline{T'}$, and they can be repelled using $Y'_{\triangle}$. Consider the (partial) assignment J that assigns the same values to the variables $x_1, \dots, x_{n_x}$ as the assignment I above, and, for any variable y_i, sets y_i to true or false if R contains the vertex y_i or $\overline{y_i}$, respectively. By assumption we know that our assignment to $x_1, \dots, x_{n_x}$ is such that for all assignments to $y_1, \dots, y_{n_y}$ some term t_i in ψ is true. In particular, it must therefore hold that J falsifies no existentially quantified literal in t_i. Then, by construction of S, the vertex $\overline{t'_i}$ is not in S. We also know that J falsifies no universally quantified literal in t_i. But then the vertices from $Y \cup \overline{Y}$ adjacent to the vertex $\overline{t'_i}$ are not in R due to our construction of J, so $\overline{t'_i}$ does not attack any vertex in R. From this it follows that there are at most $n_t - 1$ attacks from $T' \cup \overline{T'}$ on R. We can repel all these attacks using the vertices $y^{\triangle}_1, \dots, y^{\triangle}_{n_t-1}$.

This allows us to conclude $|N[R] \cap S| \geqslant |N[R] \setminus S|$. Therefore S is secure.

"If" direction. Suppose S is a secure set in G satisfying the conditions regarding forbidden, necessary and complementary vertices. First observe that $|S| = k$ because the complementary vertex pairs make sure that S contains exactly half of $\mathrm{V}(G) \setminus (V_{\triangle} \cup V_{\square})$.

If S contains some $l \in X \cup \overline{X}$, then $N(l) \cap \overline{T} \subseteq S$ by Observation 4.5. If S contains some $\overline{t_i} \in \overline{T}$, then $\overline{t_i}$ must be adjacent to some element of $X \cup \overline{X}$ that is also in S by Observation 4.4.

We construct an interpretation I on the variables $x_1, \dots, x_{n_x}$ that sets exactly those x_i to true where the corresponding vertex x_i is in S, and we claim that for each extension of I to the universally quantified variables there is a satisfied term in ψ. To see this, suppose to the contrary that some assignment J to all variables extends I but falsifies all terms in ψ. Then we define a set R consisting of all vertices y_i such that $J(y_i) = \text{true}$, all vertices $\overline{y_i}$ such that $J(y_i) = \text{false}$, and all vertices in $(T' \cup \overline{T'}) \cap S$ that are adjacent to these vertices y_i or $\overline{y_i}$. We show that this contradicts S being secure: Clearly, R is a subset of S and has $|R|$ defenders due to itself, $n_t - 1$ defenders due to $Y'_\triangle$, and $n_y \cdot n_t$ defenders due to $N(R) \cap Y_\triangle$. This amounts to $|N[R] \cap S| = |R| + n_t - 1 + n_y \cdot n_t$. On the other hand, there are n_t attacks on R from $T' \cup \overline{T'}$. This is because for any term t_i in ψ one of the following cases applies:

- The term t_i is falsified already by I. Then $\overline{t_i'} \in S$ and thus $t_i' \notin S$. The vertex t_i', however, is adjacent to every element of $Y \cup \overline{Y}$, so it attacks R.
- The term t_i is not falsified by I but by J. Then $\overline{t_i'} \notin S$, and $L_Y(t_i)$ contains some literal l with $\bar{l} \in N(\overline{t_i'})$ and $J \models \bar{l}$, so $\bar{l}$ is in R and attacked by $\overline{t_i'}$.

In addition to these n_t attackers, R has $|R \cap (T' \cup \overline{T'})|$ attackers in $N(R) \cap (T'_\square \cup \overline{T'}_\square)$, as well as $n_y \cdot (n_t + 1)$ attackers in $Y_\square$. As $|R| = n_y + |R \cap (T' \cup \overline{T'})|$, we obtain in total

$$|N[R] \setminus S| = n_t + |R \cap (T' \cup \overline{T'})| + n_y \cdot (n_t + 1) = |R| + n_t + n_y \cdot n_t > |N[R] \cap S|.$$

This contradicts S being secure, so for each extension of I to the universally quantified vertices, ψ is true; hence φ is true. ☐

4.2.2 Hardness of Secure Set with Forbidden and Necessary Vertices

Next we present a transformation τ^{FNC} that eliminates complementary vertex pairs by turning a Secure Set$^{\text{FNC}}$ instance into an equivalent Secure Set$^{\text{FN}}$ instance. Along with τ^{FNC}, we define a function σ_I^{FNC}, for each Secure Set$^{\text{FNC}}$ instance I, such that the solutions of I are in a one-to-one correspondence with those of $\tau^{\text{FNC}}(I)$ in such a way that any two solutions of I have the same size if and only if the corresponding solutions of $\tau^{\text{FNC}}(I)$ have the same size. We use these functions to obtain a polynomial-time reduction from Secure Set$^{\text{FNC}}$ to Secure Set$^{\text{FN}}$ as well as from Exact Secure Set$^{\text{FNC}}$ to Exact Secure Set$^{\text{FN}}$.

Before we formally define our reduction, we briefly describe the intuition behind the used gadgets. The gadget in Figure 4.4 adds neighbors $a_1, \dots, a_n, a_1^\square, \dots, a_n^\square$ to every vertex a, which are so many that a can only be in a solution if some of the new neighbors are also in the solution. The new vertices are structured in such a way that every solution must in fact either contain all of $a, a_1, \dots, a_n$ or none of them. Next, the gadget in Figure 4.5 is added for every complementary pair (a, b). This gadget is constructed in such a way that every solution must either contain all of $a_n, a^{ab}, a_1^{ab}, \dots, a_{n^2+n}^{ab}$ or none of them, and the same holds for $b_n, b^{ab}, b_1^{ab}, \dots, b_{n^2+n}^{ab}$. By making the vertex $\triangle^{ab}$ necessary, every solution must contain one of these two sets. At the same time, the bound on the solution size makes sure that we cannot afford to take both sets for any complementary pair.

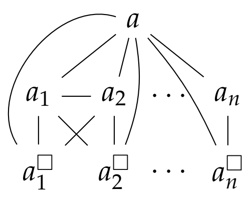


Figure 4.4: Gadget for each vertex a of the original graph in the reduction from Secure Set$^{\text{FNC}}$ to Secure Set$^{\text{FN}}$. The vertex a may have additional neighbors from the original graph, and the vertices a_n and $a_n^\square$ may have additional neighbors as depicted in Figure 4.5.

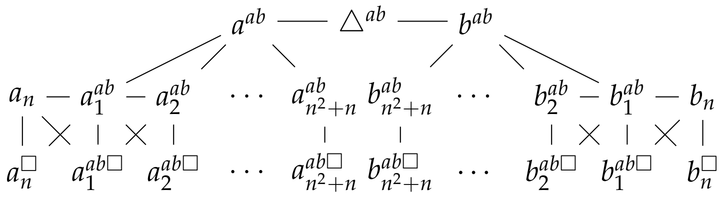


Figure 4.5: Gadget for each pair of complementary vertices (a, b) in the reduction from Secure Set$^{\text{FNC}}$ to Secure Set$^{\text{FN}}$. The vertices a_n, $a_n^\square$, b_n and $b_n^\square$ have additional neighbors as depicted in Figure 4.4.

Definition 4.8. We define a function τ^{FNC}, which assigns a Secure Set$^{\text{FN}}$ instance to each Secure Set$^{\text{FNC}}$ instance $I = (G, k, V_\square, V_\triangle, C)$. For this, we use n to denote $|V(G)|$ and first define a function

$$\sigma_I^{\text{FNC}} : x \mapsto x \cdot (n+1) + |C| \cdot (n^2 + n + 2).$$

For each $v \in V(G)$, we introduce the following sets of new vertices.

$$Y_v^\circ = \{v_1, \dots, v_n\} \qquad Y_v^\square = \{v_1^\square, \dots, v_n^\square\}$$

Next, for each $(a, b) \in C$, we introduce new vertices a^{ab}, b^{ab} and $\triangle^{ab}$ as well as, for any $x \in \{a, b\}$, the following sets of new vertices.

$$Z_{x\circ}^{ab} = \{x_1^{ab}, \dots, x_{n^2+n}^{ab}\} \qquad Z_{x\square}^{ab} = \{x_1^{ab\square}, \dots, x_{n^2+n}^{ab\square}\}$$

We use the notation $u \oplus v$ to denote the set of edges $\{(u, v), (u, u^\square), (v, v^\square), (u, v^\square), (v, u^\square)\}$. Now we define the Secure Set$^{\text{FN}}$ instance $\tau^{\text{FNC}}(I) = (G', k', V'_\square, V'_\triangle)$, where $k' = \sigma_I^{\text{FNC}}(k)$, $V'_\square = V_\square \cup \bigcup_{v \in V(G)} Y_v^\square \cup \bigcup_{(a,b) \in C} (Z_{a\square}^{ab} \cup Z_{b\square}^{ab})$, $V'_\triangle = V_\triangle \cup \bigcup_{(a,b) \in C} \{\triangle^{ab}\}$ and G' is the graph defined by

$$\begin{aligned} V(G') = V(G) \cup & \bigcup_{v \in V(G)} (Y_v^\circ \cup Y_v^\square) \cup \\ \cup & \bigcup_{(a,b) \in C} (\{\triangle^{ab}, a^{ab}, b^{ab}\} \cup Z_{a\circ}^{ab} \cup Z_{b\circ}^{ab} \cup Z_{a\square}^{ab} \cup Z_{b\square}^{ab}), \end{aligned}$$

$$\begin{aligned}\mathrm{E}(G') = \mathrm{E}(G) \cup \bigcup_{v \in \mathrm{V}(G)} & \Big((\{v\} \times Y_v^{\circ}) \cup (\{v\} \times Y_v^{\square}) \cup \bigcup_{1 \leqslant i < n} v_i \oplus v_{i+1} \Big) \cup \\ \cup \bigcup_{(a,b) \in C} \bigcup_{x \in \{a,b\}} & \Big(\{(\triangle^{ab}, x^{ab})\} \cup (\{x^{ab}\} \times Z^{ab}_{x\circ}) \cup \\ & \cup x_n \oplus x_1^{ab} \cup \bigcup_{1 \leqslant i < n^2+n} x_i^{ab} \oplus x_{i+1}^{ab} \Big).\end{aligned}$$

We illustrate our construction in Figures 4.4 and 4.5.

Lemma 4.9. *Let $I = (G, k, V_\square, V_\triangle, C)$ be a* Secure Set$^{\mathrm{FNC}}$ *instance, let A be the set of solutions of I and let B be the set of solutions of the* Secure Set$^{\mathrm{FN}}$ *instance $\tau^{FNC}(I)$. There is a bijection $f : A \to B$ such that $|f(S)| = \sigma_I^{FNC}(|S|)$ holds for every $S \in A$.*

Proof. We use the same auxiliary notation as in Definition 4.8 and we define f as $S \mapsto S \cup \bigcup_{v \in S} Y_v^{\circ} \cup \bigcup_{(a,b) \in C, x \in S \cap \{a,b\}} (\{\triangle^{ab}, x^{ab}\} \cup Z^{ab}_{x\circ})$. For every $S \in A$, we thus obtain $|f(S)| = \sigma_I^{\mathrm{FNC}}(|S|)$, and we first show that indeed $f(S) \in B$.

Let $S \in A$ and let S' denote $f(S)$. Obviously S' satisfies $V'_\square \cap S' = \emptyset$ and $V'_\triangle \subseteq S'$. To see that S' is secure in G', let X' be an arbitrary subset of S'. Since S is secure in G and $X' \cap \mathrm{V}(G) \subseteq S$, there is a defense $\mu : N_G[X' \cap \mathrm{V}(G)] \setminus S \to N_G[X' \cap \mathrm{V}(G)] \cap S$. We now construct a defense $\mu' : N_{G'}[X'] \setminus S' \to N_{G'}[X'] \cap S'$. For any attacker v of X' in G', we distinguish four cases.

- If v is some $x_i^{\square} \in Y_x^{\square}$, then we set $\mu'(v) = x_i$. This element is in $N_{G'}[X']$ since v is only adjacent to x_i or neighbors of it.
- If v is some $x_i^{ab\square} \in Z^{ab}_{x\square}$ for some $(a,b) \in C$ and $x \in \{a,b\}$, we set $\mu'(v) = x_i^{ab}$. This element is in $N_{G'}[X']$ since v is only adjacent to x_i^{ab} or neighbors of it.
- If v is a^{ab} or b^{ab} for some $(a,b) \in C$, its only neighbor in X' can be $\triangle^{ab}$ and we set $\mu'(v) = \triangle^{ab}$.
- Otherwise v is in $N_G[X' \cap \mathrm{V}(G)] \setminus S$ (by our construction of S'). Since the codomain of μ is a subset of the codomain of μ', we may set $\mu'(v) = \mu(v)$.

Since μ' is injective, each attack on X' in G' can be repelled by S'. Hence S' is secure in G'.

Clearly f is injective. It remains to show that f is surjective. Let S' be a solution of $\tau^{\mathrm{FNC}}(I)$. First we make the following observations for each $v \in \mathrm{V}(G)$:

- If $v \in S'$, then $Y_v^{\circ} \cap S' \neq \emptyset$ due to Observation 4.4, since $Y_v^{\circ} \cup Y_v^{\square}$ contains a majority of neighbors of v, and the vertices in $Y_v^{\square}$ are forbidden.

- For each $v^{ab} \in S'$, where $(a,b) \in C$ such that $v = a$ or $v = b$, it holds that $Z^{ab}_{v\circ} \cap S' \neq \emptyset$ again due to Observation 4.4.
- If S' contains an element of Y°_{v}, then $\{v\} \cup Y^{\circ}_{v} \cup \bigcup_{(v,z)\in C} Z^{vz}_{v\circ} \cup \bigcup_{(z,v)\in C} Z^{zv}_{v\circ} \subseteq S'$ by repeated applications of Observation 4.5. To see this, note in particular that $N(v_n)$ can be partitioned into the two equal-sized sets $\{v, v_{n-1}\} \cup \{v^{vz}_1 \mid (v,z) \in C\} \cup \{v^{zv}_1 \mid (z,v) \in C\}$ and $\{v^{\square}_{n-1}, v^{\square}_{n}\} \cup \{v^{vz\square}_1 \mid (v,z) \in C\} \cup \{v^{zv\square}_1 \mid (z,v) \in C\}$, and all vertices in the latter set are forbidden.
- If S' contains an element of $Z^{ab}_{v\circ}$, where $(a,b) \in C$ such that $v = a$ or $v = b$, then $\{v^{ab}\} \cup Y^{\circ}_{v} \cup \bigcup_{(v,z)\in C} Z^{vz}_{v\circ} \cup \bigcup_{(z,v)\in C} Z^{zv}_{v\circ} \subseteq S'$ for similar reasons.

It follows that for each $v \in V(G)$, S' contains either all or none of $\{v\} \cup Y^{\circ}_{v} \cup \bigcup_{(v,z)\in C} \left(\{v^{vz}\} \cup Z^{vz}_{v\circ}\right) \cup \bigcup_{(z,v)\in C} \left(\{v^{zv}\} \cup Z^{zv}_{v\circ}\right)$.

For every $(a,b) \in C$, S' contains a^{ab} or b^{ab}, since $\triangle^{ab} \in S'$, whose neighbors are a^{ab} and b^{ab}. It follows that $|S'| > |C| \cdot (n^2 + n + 2)$ even if S' contains only one of each $(a,b) \in C$. If, for some $(a,b) \in C$, S' contained both a and b, we could derive a contradiction to $|S'| \leqslant \sigma^{\mathrm{FNC}}_{I}(k) = k \cdot (n+1) + |C| \cdot (n^2 + n + 2)$ because then $|S'| > (|C| + 1) \cdot (n^2 + n + 2) > \sigma^{\mathrm{FNC}}_{I}(k)$. So S' contains either a or b for any $(a,b) \in C$.

We construct $S = S' \cap V(G)$ and observe that $S' = f(S)$, $V_{\triangle} \subseteq S$, $V_{\square} \cap S = \emptyset$, and $|S \cap \{a,b\}| = 1$ for each $(a,b) \in C$. It remains to show that S is secure in G. Let X be an arbitrary subset of S. For each $x \in X$, the set X has n additional defenders in G' compared to G (namely $x_1, \dots, x_n$), and X has n additional attackers in G' (namely $x^{\square}_1, \dots, x^{\square}_n$); so $|N_{G'}[X] \cap S'| - |N_G[X] \cap S| = |N_{G'}[X] \setminus S'| - |N_G[X] \setminus S|$. Clearly X is a subset of S', so $|N_{G'}[X] \cap S'| \geqslant |N_{G'}[X] \setminus S'|$ as S' is secure in G'. We conclude that $|N_G[X] \cap S| \geqslant |N_G[X] \setminus S|$. Hence S is secure in G. □

As τ^{FNC} is clearly computable in polynomial time, we obtain the following result:

Corollary 4.10. Secure Set$^{\mathrm{FN}}$ *is* Σ^{P}_2*-hard.*

The instances of Secure Set$^{\mathrm{FNC}}$ are identical to the instances of the exact variant, so τ^{FNC} is also applicable to the exact case. In fact it turns out that this gives us also a reduction from Exact Secure Set$^{\mathrm{FNC}}$ to Exact Secure Set$^{\mathrm{FN}}$.

Lemma 4.11. Exact Secure Set$^{\mathrm{FN}}$ *is* Σ^{P}_2*-hard.*

Proof. Let I and $I' = \tau^{\mathrm{FNC}}(I)$ be our Exact Secure Set$^{\mathrm{FNC}}$ and Exact Secure Set$^{\mathrm{FN}}$ instances, respectively, and let k and k' denote their respective solution sizes. By Lemma 4.9, there is a bijection f between the solutions of I and the solutions of I' such that, for every solution S of I, $f(S)$ has $\sigma^{\mathrm{FNC}}_{I}(k) = k'$ elements, and for every solution S' of I', $f^{-1}(S')$ has k elements since $\sigma^{\mathrm{FNC}}_{I}$ is invertible. □

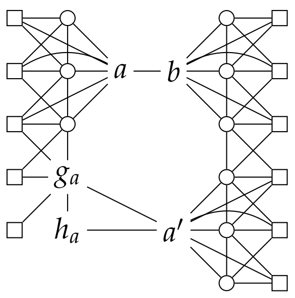


Figure 4.6: Result of the transformation τ^{FN} applied to an example graph with two adjacent vertices a and b, where b is necessary. Every solution in the depicted graph contains a', h_a and b.

4.2.3 Hardness of Secure Set with Forbidden Vertices

Now we present a transformation τ^{FN} that eliminates necessary vertices. Our transformation not only operates on a problem instance, but also requires an ordering $\preceq$ of the non-forbidden vertices of the graph. For now, we can consider this as an arbitrary ordering. It will become more important in Section 4.4, where we reuse this transformation for showing W[1]-hardness w.r.t. treewidth.

Before formally defining the transformation τ^{FN}, we refer to Figure 4.6, which shows the result for a simple example graph with only two vertices a and b, of which b is necessary. The basic idea is that the vertex a' must be in every solution S: If a or any vertex to the left of a is in S, it eventually forces a' to be in S as well. Likewise, if b or any vertex to the right of b is in S, it also forces a' to be in S. Once $a' \in S$, the construction to the right of a' makes sure that $b \in S$. We will generalize this to instances containing more vertices so that every necessary vertex as well as the primed copy of each non-necessary vertex is in every solution.

Definition 4.12. We define a function τ^{FN}, which assigns a Secure Set$^{\mathrm{F}}$ instance to each pair $(I, \preceq)$, where $I = (G, k, V_\square, V_\triangle)$ is a Secure Set$^{\mathrm{FN}}$ instance and $\preceq$ is an ordering of the non-forbidden elements of $\mathrm{V}(G)$. For this, let $V_\circ$ denote $\mathrm{V}(G) \setminus (V_\square \cup V_\triangle)$. We use n to denote $|\mathrm{V}(G)|$, and we first define a function $\sigma_I^{\mathrm{FN}} : x \mapsto (n+3) \cdot (x + |V_\circ|) - |V_\triangle|$. We use H to denote the set of new vertices $\{v', g_v, h_v, g_v^\square, h_v^\square \mid v \in V_\circ\}$. The intention is for each $g_v^\square$ and $h_v^\square$ to be forbidden, for each v' and h_v to be in every solution, and for g_v to be in a solution if and only if v is in it at the same time. We write V^+ to denote $V_\triangle \cup V_\circ \cup \{v' \mid v \in V_\circ\}$; for each $v \in V^+$, we use A_v to denote the set of new vertices $\{v_1, \dots, v_{n+1}, v_1^\square, \dots, v_{n+1}^\square\}$, and we use shorthand notation $A_v^\circ = \{v_1, \dots, v_{n+1}\}$ and $A_v^\square = \{v_1^\square, \dots, v_{n+1}^\square\}$. The intention is for each $v_i^\square$ to be forbidden and for each v_i to be in a solution if and only if v is in it at the same time. We use the notation $u \oplus v$ to denote the set of edges $\{(u,v), (u,u^\square), (v,v^\square), (u,v^\square), (v,u^\square)\}$. For any vertex $v \in V_\circ \cup V_\triangle$, we define $p(v) = v$ if $v \in V_\triangle$ and $p(v) = v'$ if $v \in V_\circ$. Let P be the set consisting

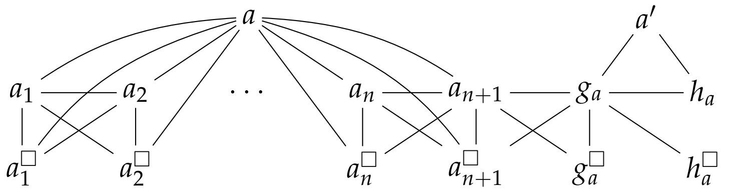


Figure 4.7: Illustration of the gadget that makes sure that every solution containing a also contains f_a, g_a and a'. The vertex a is a non-necessary, non-forbidden vertex from the Secure Set$^{\text{FN}}$ instance and may have other neighbors from this instance. The vertex a' additionally has the neighbors depicted in Figure 4.8.

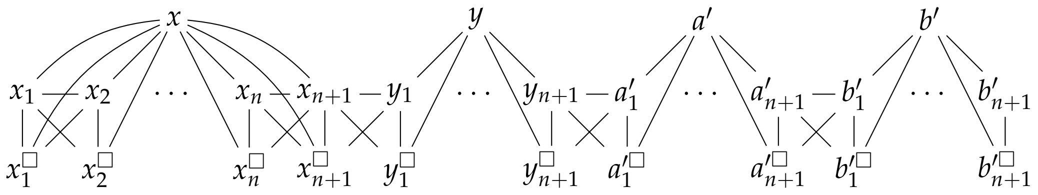


Figure 4.8: Illustration of the gadget that makes sure that every solution contains all necessary vertices if it contains some necessary vertex or if it contains v' for some non-necessary vertex v. Here we assume there are the four vertices a, b, x, y, among which x and y are necessary, and we use the ordering $x \preceq y \preceq a \preceq b$.

of all pairs $(p(u), p(v))$ such that v is the direct successor of u according to $\preceq$. Now we define $\tau^{\text{FN}}(I, \preceq) = (G', k', V'_\square)$, where $V'_\square = V_\square \cup \{g_v^\square, h_v^\square \mid v \in V_\circ\} \cup \bigcup_{v \in V^+} A_v^\square$, $k' = \sigma_I^{\text{FN}}(k)$, and G' is the graph defined by

$$\begin{aligned}
\mathrm{V}(G') &= \mathrm{V}(G) \cup H \cup \bigcup_{v \in V^+} A_v, \\
\mathrm{E}(G') &= \mathrm{E}(G) \cup \{(v, v_i), (v, v_i^\square) \mid v \in V^+,\ 1 \leqslant i \leqslant n+1\} \\
&\cup \bigcup_{v \in V^+,\, 1 \leqslant i \leqslant n} v_i \oplus v_{i+1} \cup \bigcup_{(u,v) \in P} u_{n+1} \oplus v_1 \\
&\cup \bigcup_{v \in V_\circ} v_{n+1} \oplus g_v \cup \{(v', g_v), (v', h_v), (g_v, h_v), (g_v, h_v^\square) \mid v \in V_\circ\}.
\end{aligned}$$

We illustrate our construction in Figure 4.7 and 4.8.

We now prove that τ^{FN} yields a correct reduction for any ordering $\preceq$.

Lemma 4.13. *Let $I = (G, k, V_\square, V_\triangle)$ be a* Secure Set$^{\text{FN}}$ *instance, let $\preceq$ be an ordering of $\mathrm{V}(G) \setminus V_\square$, let A be the set of solutions of I and let B be the set of solutions of the* Secure Set$^{\text{F}}$ *instance $\tau^{FN}(I, \preceq)$. There is a bijection $f : A \to B$ such that $|f(S)| = \sigma_I^{FN}(|S|)$ holds for every $S \in A$.*

Proof. We use the same auxiliary notation as in Definition 4.12 and we define f as

$$f(S) = S \cup \bigcup_{v \in S} A_v^{\circ} \cup \{v', h_v \mid v \in V_{\circ}\} \cup \bigcup_{v \in V_{\circ}} A_{v'}^{\circ} \cup \{g_v \mid v \in S \cap V_{\circ}\}.$$

For every $S \in A$, we thus obtain $|f(S)| = |S| + |S|(n+1) + 2|V_{\circ}| + |V_{\circ}| \cdot (n+1) + (|S| - |V_{\triangle}|) = \sigma_I^{\mathrm{FN}}(|S|)$, and we first show that indeed $f(S) \in B$.

Let $S \in A$ and let S' denote $f(S)$. Obviously S' satisfies $V'_{\square} \cap S' = \emptyset$. To see that S' is secure in G', let X' be an arbitrary subset of S'. Since S is secure in G and $X' \cap \mathrm{V}(G) \subseteq S$, there is a defense $\mu : N_G[X' \cap \mathrm{V}(G)] \setminus S \to N_G[X' \cap \mathrm{V}(G)] \cap S$. We now construct a defense $\mu' : N_{G'}[X'] \setminus S' \to N_{G'}[X'] \cap S'$. For any attacker a of X' in G', we distinguish the following cases:

- If a is some $v_i^{\square} \in A_v^{\square}$ for some $v \in V^+$, then a can only attack either v_i or a neighbor of v_i, all of which are in S', and we set $\mu'(a) = v_i$.
- Similarly, if a is $g_v^{\square}$ for some $v \in V_{\circ}$, then we set $\mu'(a) = g_v$.
- If a is $h_v^{\square}$ for some $v \in V_{\circ}$, then a attacks g_v and we set $\mu'(a) = h_v$.
- If a is g_v for some $v \in V_{\circ}$, then it attacks v' or h_v, which is not used for repelling any other attack because $h_v^{\square}$ cannot attack X', so we set $\mu'(a) = h_v$.
- Otherwise a is in $N_G[X' \cap \mathrm{V}(G)] \setminus S$ (by our construction of S'). Since the codomain of μ is a subset of the codomain of μ', we may set $\mu'(a) = \mu(a)$.

Since μ' is injective, each attack on X' in G' can be repelled by S'. Hence S' is secure in G'.

Clearly f is injective. It remains to show that f is surjective. Let S' be a solution of $\tau^{\mathrm{FN}}(I, \preceq)$. We first show that $V_{\triangle} \cup \{v', h_v \mid v \in V_{\circ}\} \subseteq S'$:

- If S' contains some $v \in V^+$, then S' contains an element of A_v° by Observation 4.4.
- If S' contains an element of A_v° for some $v \in V^+$, then $\{v\} \cup A_v^{\circ} \subseteq S'$ by Observation 4.5.
- If $v_{n+1} \in S'$ for some $v \in V_{\circ}$, then $g_v \in S'$ for the same reason.
- Furthermore, if S' contains an element of A_v° for some $v \in V_{\triangle} \cup \{v' \mid v \in V_{\circ}\}$, then also $A_u^{\circ} \subseteq S'$ for every $u \in V_{\triangle} \cup \{v' \mid v \in V_{\circ}\}$ for the same reason.
- If $g_v \in S'$ for some $v \in V_{\circ}$, then $\{h_v, v', v_{n+1}\} \subseteq S'$ by Observation 4.5.
- If $h_v \in S'$ for some $v \in V_{\circ}$, then $a' \in S'$ because at least g_v or v' must be in S' and the former implies $v' \in S'$ as we have seen.

- Since S' is nonempty, the previous observations show that for every $v \in V_\triangle \cup \{v' \mid v \in V_\circ\}$ it holds that $\{v\} \cup A^\circ_v \subseteq S'$. Finally, we show that $\{h_v \mid v \in V_\circ\} \subseteq S'$. Suppose, for the sake of contradiction, that there is some $v \in V_\circ$ such that $h_v \notin S'$. We have seen that the latter can only be the case if $g_v \notin S'$, and we know that $v' \in S'$. We obtain the contradiction that v' is attacked by g_v, h_v and $A^\square_{v'}$, whereas its only defenders are v' itself and $A^\circ_{v'}$.

Let $S = S' \cap \mathrm{V}(G)$. By the previous observations, it is easy to see that $S' = f(S)$. It remains to show that S is secure in G. Let X be an arbitrary subset of S. Observe that the number of additional defenders of X in G' compared to G is equal to the number of additional attackers; formally $|N_{G'}[X] \cap S'| - |N_G[X] \cap S| = |N_{G'}[X] \setminus S'| - |N_G[X] \setminus S|$. Since S' is secure in G', it holds that $|N_{G'}[X'] \cap S'| \geqslant |N_{G'}[X'] \setminus S'|$. Consequently $|N_G[X] \cap S| \geqslant |N_G[X] \setminus S|$. Hence S is secure in G. □

Given an ordering $\preceq$, clearly $\tau^{\mathrm{FN}}(I, \preceq)$ is computable in polynomial time. We can thus easily obtain a reduction from Secure Set$^{\mathrm{FN}}$ to Secure Set$^{\mathrm{F}}$ by first computing an arbitrary ordering $\preceq$ of the non-forbidden vertices. This also gives us a hardness result for the exact case, analogous to Lemma 4.11.

Corollary 4.14. Secure Set$^{\mathrm{F}}$ *and* Exact Secure Set$^{\mathrm{F}}$ *are* Σ^{P}_2*-hard.*

4.2.4 Hardness of Secure Set

We now introduce a transformation τ^{F} that eliminates forbidden vertices. The basic idea is that we ensure that a forbidden vertex f is never part of a solution by adding so many neighbors to f that we could only defend f by exceeding the bound on the solution size.

Definition 4.15. We define a function τ^{F}, which assigns a Secure Set instance to each Secure Set$^{\mathrm{F}}$ instance $I = (G, k, V_\square)$. For each $f \in V_\square$, we introduce new vertices $f', f_1, \dots, f_{2k}$. Now we define $\tau^{\mathrm{F}}(I) = (G', k)$, where G' is the graph defined by

$$\begin{aligned} \mathrm{V}(G') &= \mathrm{V}(G) \cup \{f', f_1, \dots, f_{2k} \mid f \in V_\square\}, \\ \mathrm{E}(G') &= \mathrm{E}(G) \cup \{(f, f_i),\ (f', f_i) \mid f \in V_\square,\ 1 \leqslant i \leqslant 2k\}. \end{aligned}$$

Lemma 4.16. *Every* Secure Set$^{\mathrm{F}}$ *instance I has the same solutions as the* Secure Set *instance $\tau^{\mathrm{F}}(I)$.*

Proof. Let $I = (G, k, V_\square)$ and $\tau^{\mathrm{F}}(I) = (G', k)$. Each solution S of I is also solution of $\tau^{\mathrm{F}}(I)$ because the subgraph of G induced by $N_G[S]$ is equal to the subgraph of G' induced by $N_{G'}[S]$. Now let S' be a solution of $\tau^{\mathrm{F}}(I)$. For every $f \in V_\square$, neither f nor f' are in S' because each of these vertices has at least $2k$ neighbors, and S' cannot contain any f_i because $N_{G'}(f_i) = \{f, f'\}$. Hence S' is also a solution of I as the subgraphs induced by the respective neighborhoods are again equal. □

Since $k \leqslant |\mathrm{V}(G)|$, the function τ^{F} can clearly be computed in polynomial time. This immediately yields the following result.

Corollary 4.17. Secure Set *and* Exact Secure Set *are* Σ_2^{P}*-hard.*

4.3 Relationship Between Secure Set and Defensive Alliance

In this section, we present a reduction from Defensive Alliance to Secure Set that also works for our considered variants and in fact preserves bounded treewidth. This serves as preparation for Section 4.4, where we prove a hardness result for Defensive Alliance when parameterized by treewidth. This result will then carry over to Secure Set by virtue of the reduction that we present in this section. First we briefly discuss the relationship between those two problems in order to argue that it is not trivial that hardness results for the seemingly easier problem carry over to the other problem.

It is sometimes stated (Ho and Dutton 2009) that Secure Set is a generalization of Defensive Alliance as the latter only consider subsets of size 1 of a set S of vertices for checking whether S is a solution, whereas the former considers all subsets of S. Indeed every secure set is also a defensive alliance, so that is a reasonable statement. It is also hardly surprising that Secure Set is at least as hard as Defensive Alliance, as we proved in Section 4.2. However, we would like to emphasize that, although these problems are clearly related, the relationship between the two problems is not as simple as it may seem. Clearly there are problems where hardness results trivially carry over from the special case to the more general problem (and conversely for algorithms). This is the case, for example, for the 3-Colorability problem and its generalization Graph Coloring, where the number of colors is part of the input.

In the case of Secure Set, hardness results do not immediately carry over from Defensive Alliance but require some additional work. This becomes especially apparent when we consider the exact problem variants or parameterized complexity results, where it is important that reductions preserve small parameter values. Due to the strong likeness of these two base problems, it makes sense to study them in the framework of a more general problem that encompasses them both. Hence we will shortly define the more general problem Generalized Secure Set of which Secure Set and Defensive Alliance are special cases. For this, we first define the following notion.

Definition 4.18. Given a graph G and an integer s, a set $S \subseteq \mathrm{V}(G)$ is *s-secure in G* if for each $X \subseteq S$ of size at most s it holds that $|N[X] \cap S| \geqslant |N[X] \setminus S|$.

Obviously a set S is secure if and only if it is $|S|$-secure, and a set S is a defensive alliance if and only if it is 1-secure. We now use this notion to define the generalized problem.

Generalized Secure Set

Input: A graph G and integers k, s both at least 1 and at most $|V(G)|$

Question: Does there exist a set $S \subseteq V(G)$ with $1 \leqslant |S| \leqslant k$ that is s-secure?

Both Defensive Alliance and Secure Set are special cases of this problem. The former is the case where the parameter s has the fixed value of 1; in the latter the *dual parameter* $|V(G)| - s$ has a fixed value of 1. Hence we suggest to consider Defensive Alliance as a *dual* problem to Secure Set and not as a special case.

In the following, we present a reduction from Defensive Alliance to Secure Set that also works for the considered variants and is in fact an FPT reduction when the problems are parameterized by treewidth. To simplify this reduction, we first define the following auxiliary problem and show that it can be reduced to Secure Set$^{\text{FNC}}$ and subsequently, as we have seen, to Secure Set. Our new problem adds the possibility of specifying pairs of *equivalent vertices*. The idea is that equivalent vertices are "glued together" in the sense that for each solution S and every vertex v, either all vertices that are equivalent to v are together with v in S, or none of them is.

Secure Set$^{\text{FNCE}}$

Input: A graph G, an integer k, a set $V_\square \subseteq V(G)$, a set $V_\triangle \subseteq V(G)$, a set $C \subseteq V(G)^2$ and a set $Q \subseteq V(G)^2$

Question: Does there exist a set $S \subseteq V(G) \setminus V_\square$ with $V_\triangle \subseteq S$ and $1 \leqslant |S| \leqslant k$ that is secure, contains either a or b for each pair $(a, b) \in C$ and, for each pair $(x, y) \in Q$, contains either both or none of x and y?

The treewidth of a Secure Set$^{\text{FNCE}}$ instance is the treewidth of its primal graph, which is defined as follows:

Definition 4.19. Let I be a Secure Set$^{\text{FNCE}}$ instance, let G be the graph in I, let C the set of complementary vertex pairs in I and let Q be the set of equivalent vertex pairs in I. By the *primal graph* of I we mean the undirected graph G' with $V(G') = V(G)$ and $E(G') = E(G) \cup C \cup Q$.

We also define the problem Exact Secure Set$^{\text{FNCE}}$ as the exact version of Secure Set$^{\text{FNCE}}$ in the obvious way.

Now we present our reduction from Secure Set$^{\text{FNCE}}$ to Secure Set$^{\text{FNC}}$.

Definition 4.20. We define a function τ^{FNCE}, which assigns a Secure Set$^{\mathrm{FNC}}$ instance to each Secure Set$^{\mathrm{FNCE}}$ instance $I = (G, k, V_\square, V_\triangle, C, Q)$. For this, we first define a function

$$\sigma_I^{\mathrm{FNCE}} : x \mapsto |\mathrm{V}(G)| + x.$$

Now we define the Secure Set$^{\mathrm{FNC}}$ instance $\tau^{\mathrm{FNCE}}(I) = (G', k', V_\square, V_\triangle, C')$, where $k' = \sigma_I^{\mathrm{FNCE}}(k)$ and G' is the graph defined as follows:

$$\begin{aligned} \mathrm{V}(G') &= \{v, \bar{v}, v' \mid v \in \mathrm{V}(G)\} \\ \mathrm{E}(G') &= \mathrm{E}(G) \end{aligned}$$

The pairs of complementary vertices are given by

$$C' = C \cup \{(v, \bar{v}), (\bar{v}, v') \mid v \in \mathrm{V}(G)\} \cup \{(x, \bar{y}) \mid (x, y) \in Q\}.$$

Lemma 4.21. *Let $I = (G, k, V_\square, V_\triangle, C, Q)$ be a* Secure Set$^{\mathrm{FNCE}}$ *instance, let A be the set of solutions of I and let B be the set of solutions of the* Secure Set$^{\mathrm{FNC}}$ *instance $\tau^{FNCE}(I)$. There is a bijection $f : A \to B$ such that $|f(S)| = \sigma_I^{FNCE}(|S|)$ holds for every $S \in A$.*

Proof. We define f as $S \mapsto \{v, v' \mid v \in S\} \cup \{\bar{v} \mid v \in \mathrm{V}(G) \setminus S\}$. For every $S \in A$, we thus obtain $|f(S)| = \sigma_I^{\mathrm{FNCE}}(|S|)$, and we first show that indeed $f(S) \in B$.

Let $S \in A$ and let S' denote $f(S)$. Obviously S' satisfies $V'_\square \cap S' = \emptyset$ and $V'_\triangle \subseteq S'$. Observe that S is secure in G' because it is secure in G and the subgraph of G induced by $N_G[S]$ is equal to the subgraph of G' induced by $N_{G'}[S]$. Moreover, S' consists of the vertices in S and additionally only vertices $\bar{v}$ or v' that have no neighbors. Hence S being secure in G' implies that S' is secure in G'.

Clearly f is injective. It remains to show that f is surjective. Let S' be a solution of $\tau^{\mathrm{FNCE}}(I)$. Note that for each pair of equivalent vertices $(x, y) \in Q$, S' contains x if and only if it contains y since x and $\bar{y}$, as well as $\bar{y}$ and y, are complementary.

We construct $S = S' \cap \mathrm{V}(G)$ and observe that $S' = f(S)$, $V_\triangle \subseteq S$, $V_\square \cap S = \emptyset$, $|S \cap \{x, y\}| = 1$ for each $(x, y) \in C$, and either $\{x, y\} \subseteq S$ or $\{x, y\} \cap S = \emptyset$ for every equivalent pair $(x, y) \in Q$. It is easy to see that a set is secure if and only if this set restricted to vertices that have at least one neighbor is secure. Hence S is secure in G'. Since the subgraph of G induced by $N_G[S]$ is equal to the subgraph of G' induced by $N_{G'}[S]$, S is also secure in G. $\square$

Lemma 4.22. *There is an FPT reduction that runs in polynomial time from* Secure Set$^{\mathrm{FNCE}}$ *to* Secure Set$^{\mathrm{FNC}}$ *as well as from* Exact Secure Set$^{\mathrm{FNCE}}$ *to* Exact Secure Set$^{\mathrm{FNC}}$ *when all problems are parameterized by treewidth.*

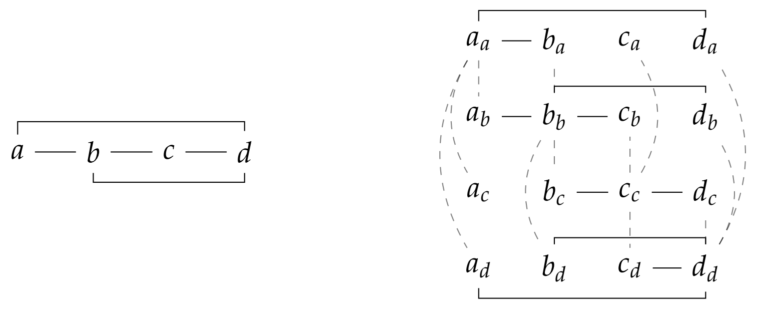


Figure 4.9: A graph (left) and the result of applying the transformation τ^{DA} to it (right). Equivalent vertex pairs are drawn as dashed lines.

Proof. Let $I = (G, k, V_\square, V_\triangle, C, Q)$ be a Secure Set$^{\mathrm{FNCE}}$ instance and let I' denote the Secure Set$^{\mathrm{FNC}}$ instance $\tau^{\mathrm{FNCE}}(I)$. By Lemma 4.21, τ^{FNCE} yields correct reductions. Clearly we can also compute τ^{FNCE} in polynomial time. We show that the treewidth of I' depends only on the treewidth of I. We can obtain a tree decomposition $\mathcal{T}'$ of I' by modifying an optimal tree decomposition $\mathcal{T}$ of I as follows: To each bag containing a vertex $v \in \mathrm{V}(G)$, we add the elements $\bar{v}$ and v'. The width of $\mathcal{T}'$ is at most three times the treewidth of I. Moreover, every pair of complementary vertices of I' is covered by a bag: The pairs already in C are already covered by a bag in $\mathcal{T}$, the pairs $(v, \bar{v})$ and $(\bar{v}, v')$ are covered by our construction, and the pairs $(x, \bar{y})$ for $(x, y) \in Q$ are also covered by a bag since $\mathcal{T}$ contains a node whose bag contains both x and y. The same reasoning also works for reducing Exact Secure Set$^{\mathrm{FNCE}}$ to Exact Secure Set$^{\mathrm{FNC}}$ by Lemma 4.21. $\square$

Now we present our reduction from Defensive Alliance to Secure Set$^{\mathrm{FNCE}}$. The idea is to form the disjoint union of all neighborhoods in the original graph G and require all copies of the same vertex to be equivalent to each other.

Definition 4.23. We define a function τ^{DA}, which assigns a Secure Set$^{\mathrm{FNCE}}$ instance to each Defensive Alliance instance $I = (G, k)$. For this, we first define a function

$$\sigma_I^{\mathrm{DA}} : x \mapsto |\mathrm{V}(G)| \cdot x.$$

Now we define the Secure Set$^{\mathrm{FNCE}}$ instance $\tau^{\mathrm{DA}}(I) = (G', k', \emptyset, \emptyset, \emptyset, Q)$, where $k' = \sigma_I^{\mathrm{DA}}(k)$ and G' is the graph defined as follows:

$$\mathrm{V}(G') = \{x_y \mid x, y \in \mathrm{V}(G)\}$$
$$\mathrm{E}(G') = \{(x_x, y_x) \mid (x, y) \in \mathrm{E}(G)\}$$

The pairs of equivalent vertices are given by

$$Q = \{(x_x, x_y) \mid x, y \in \mathrm{V}(G)\}.$$

We illustrate this construction in Figure 4.9.

Lemma 4.24. *Let $I = (G, k)$ be a* Defensive Alliance *instance, let A be the set of solutions of I and let B be the set of solutions of the* Secure Set$^{\text{FNCE}}$ *instance $\tau^{DA}(I)$. There is a bijection $f : A \to B$ such that $|f(S)| = \sigma_I^{DA}(|S|)$ holds for every $S \in A$.*

Proof. We define f as $S \mapsto \{x_y \mid x \in S,\ y \in \mathrm{V}(G)\}$. For every $S \in A$, we thus obtain $|f(S)| = \sigma_I^{\text{FNCE}}(|S|)$, and we first show that indeed $f(S) \in B$.

Let $S \in A$ and let S' denote $f(S)$. Obviously either $\{x,y\} \subseteq S'$ or $\{x,y\} \cap S' = \emptyset$ holds for every equivalent pair $(x,y) \in Q$. We next show that S' is secure in G'. For this we assume that S' does not contain vertices from different components of G' because a set is secure if and only if each of its restrictions to vertices from the same component is secure. Let X' be an arbitrary subset of S'. Suppose that X' contains a vertex x_y such that $x \neq y$. Such a vertex only has one neighbor in G', so the set Y obtained by removing x_y from X' would decrease the number of attackers at most by one. In fact, if this decreases the number of attackers by one, then it also decreases the number of defenders by one. Hence, if we can show that $|N_{G'}[Y] \cap S'| \geqslant |N_{G'}[Y] \setminus S'|$, then it follows that $|N_{G'}[X'] \cap S'| \geqslant |N_{G'}[X'] \setminus S'|$. We may thus assume that X' only contains a single vertex x_x. Note that x is an element of S and any neighbor y of x in G is in S if and only if $y_x \in S'$. Since S is a defensive alliance in G, it follows that S' is secure in G'.

Clearly f is injective. It remains to show that f is surjective. Let S' be a solution of $\tau^{\text{FNCE}}(I)$. We construct $S = \{x \in \mathrm{V}(G) \mid x_x \in S'\}$ and observe that $S' = f(S)$. Since S' is secure in G', each $x_x \in S'$ has at least as many defenders as attackers in G'. The subgraph induced by $N_{G'}[x_x]$ is isomorphic to the subgraph induced by $N_G[x]$, and any neighbor y_x of x_x in G' is in S' if and only if y is a neighbor of x in G and $y \in S$. Hence S is a defensive alliance in G. ☐

Lemma 4.25. *There is an FPT reduction that runs in polynomial time from* Defensive Alliance *to* Secure Set$^{\text{FNCE}}$ *as well as from* Exact Defensive Alliance *to* Exact Secure Set$^{\text{FNCE}}$ *when all problems are parameterized by treewidth.*

Proof. Let $I = (G, k)$ be a Defensive Alliance instance and let I' denote the Secure Set$^{\text{FNCE}}$ instance $\tau^{\text{DA}}(I)$. By Lemma 4.24, τ^{DA} yields a correct reduction. Clearly it is also computable in polynomial time.

Let P denote the primal graph of I'. We show that the treewidth of P depends only on the treewidth of G. We can obtain a tree decomposition $\mathcal{T}'$ of P by modifying an optimal tree decomposition $\mathcal{T}$ of G as follows:

1. We replace each bag element x by x_x.

2. For each edge $(x, y) \in \mathrm{E}(G)$, we pick an arbitrary node in $\mathcal{T}$ whose bag contains both x and y. The bag of its corresponding node in $\mathcal{T}'$ contains both x_x and y_y. We add to the children of this node in $\mathcal{T}'$ a new node with bag $\{x_x, y_y, x_y, y_x\}$.

3. For each pair (x, y) of non-adjacent vertices in G, we pick an arbitrary node in $\mathcal{T}$ whose bag contains x. The bag of its corresponding node in $\mathcal{T}'$ contains x_x. We add to the children of this node in $\mathcal{T}'$ a new node with bag $\{x_x, x_y\}$. (Since G is undirected, this also takes care of y.)

It is easy to see that $\mathcal{T}'$ is a valid tree decomposition of P. Moreover, its width is the maximum of three and the treewidth of G. □

By our other reductions, we immediately get the following result as well.

Corollary 4.26. *There is an FPT reduction from* Defensive Alliance *to* Secure Set *as well as from* Exact Defensive Alliance *to* Exact Secure Set *when all problems are parameterized by treewidth.*

Note that this FPT reduction does not run in polynomial time. While the FPT reductions from Defensive Alliance to Secure Set$^{\mathrm{FNCE}}$, from Secure Set$^{\mathrm{FNCE}}$ to Secure Set$^{\mathrm{FNC}}$ and then from Secure Set$^{\mathrm{FNC}}$ to Secure Set$^{\mathrm{FN}}$ do work in polynomial time, the FPT reduction from Secure Set$^{\mathrm{FN}}$ to Secure Set$^{\mathrm{F}}$ does not. The reason is that there we have to compute an appropriate ordering of the non-forbidden vertices, which really requires "FPT power". However, we can still get a polynomial-time reduction by choosing an *arbitrary* ordering of the non-necessary vertices, as we have shown in Lemma 4.13. This will not be an FPT reduction because it does not preserve bounded treewidth in general, but if we are just interested in a polynomial-time reduction, then this is fine.

Corollary 4.27. *There is a polynomial-time reduction from* Defensive Alliance *to* Secure Set *as well as from* Exact Defensive Alliance *to* Exact Secure Set.

4.4 Complexity of Alliance Problems Parameterized by Treewidth

We now turn to the parameterized complexity of the Secure Set and Defensive Alliance problems, as well as their variants, when treewidth is the parameter. This section is devoted to proving the following theorem:

Theorem 4.28. Secure Set, Exact Secure Set, Defensive Alliance *and* Exact Defensive Alliance *are all* W[1]*-hard when parameterized by treewidth.*

As a consequence, clearly also all problem variants that we considered in this work are W[1]-hard. Under the widely held assumption that FPT $\neq$ W[1], this rules out fixed-parameter tractable algorithms for these problems.

First we show hardness for Defensive Alliance$^{\text{FNC}}$. Then we successively reduce this problem to Defensive Alliance by reusing some reductions from Section 4.2. We have originally defined these reductions for variants of Secure Set, but we will prove that they also work for the respective variants of Defensive Alliance. Furthermore, we will we show that they preserve bounded treewidth.

4.4.1 Hardness of Defensive Alliance with Forbidden, Necessary and Complementary Vertices

To show W[1]-hardness of Defensive Alliance$^{\text{FNC}}$, we reduce from the following problem (Asahiro, Miyano and Ono 2011), which is known to be W[1]-hard (Szeider 2011b) parameterized by the treewidth of the graph:

Minimum Maximum Outdegree	
Input:	A graph G, an edge weighting $w : \mathrm{E}(G) \to \mathbb{N}^+$ given in unary and a positive integer r
Question:	Is there an orientation of the edges of G such that, for each $v \in \mathrm{V}(G)$, the sum of the weights of outgoing edges from v is at most r?

Lemma 4.29. Defensive Alliance$^{\text{FNC}}$ *and* Exact Defensive Alliance$^{\text{FNC}}$*, both parameterized by the treewidth of the primal graph, are* W[1]*-hard.*

Proof. Let an instance of Minimum Maximum Outdegree be given by a graph G, an edge weighting $w : \mathrm{E}(G) \to \mathbb{N}^+$ in unary and a positive integer r. From this we construct an instance of both Defensive Alliance$^{\text{FNC}}$ and Exact Defensive Alliance$^{\text{FNC}}$. An example is given in Figure 4.10. For each $v \in \mathrm{V}(G)$, we define the set of new vertices $H_v = \{h_1^v, \dots, h_{2r-1}^v\}$, and for each $(u, v) \in \mathrm{E}(G)$, we define the sets of new vertices $V_{uv} = \{u_1^v, \dots, u_{w(u,v)}^v\}$, $V_{uv}^{\square} = \{u_1^{v\square}, \dots, u_{w(u,v)}^{v\square}\}$, $V_{vu} = \{v_1^u, \dots, v_{w(u,v)}^u\}$ and $V_{vu}^{\square} = \{v_1^{u\square}, \dots, v_{w(u,v)}^{u\square}\}$. We now define the graph G' with

$$\begin{aligned} \mathrm{V}(G') &= \mathrm{V}(G) \cup \bigcup_{v \in \mathrm{V}(G)} H_v \cup \bigcup_{(u,v) \in \mathrm{E}(G)} (V_{uv} \cup V_{uv}^{\square} \cup V_{vu} \cup V_{vu}^{\square}), \\ \mathrm{E}(G') &= \{(v, h) \mid v \in \mathrm{V}(G),\ h \in H_v\} \\ &\quad \cup \{(u, x) \mid (u, v) \in \mathrm{E}(G),\ x \in V_{uv} \cup V_{uv}^{\square}\} \\ &\quad \cup \{(x, v) \mid (u, v) \in \mathrm{E}(G),\ x \in V_{vu} \cup V_{vu}^{\square}\}. \end{aligned}$$

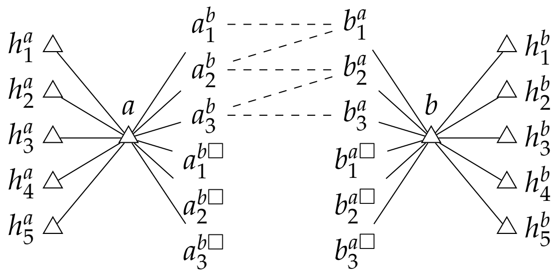


Figure 4.10: Result of our transformation on a sample Minimum Maximum Outdegree instance with $r = 3$ and two vertices a, b that are connected by an edge of weight 3. Complementary vertex pairs are shown via dashed lines. Necessary and forbidden vertices have a △ and □ symbol next to their name, respectively.

We also define the set of complementary vertex pairs $C = \{(u_i^v, v_i^u) \mid (u,v) \in \mathrm{E}(G),\ 1 \leqslant i \leqslant w(u,v)\} \cup \{(v_i^u, u_{i+1}^v) \mid (u,v) \in \mathrm{E}(G),\ 1 \leqslant i < w(u,v)\}$. Finally, we define the set of necessary vertices $V_\triangle = \mathrm{V}(G) \cup \bigcup_{v \in \mathrm{V}(G)} H_v$, the set of forbidden vertices $V_\Box = \bigcup_{(u,v) \in \mathrm{E}(G)} (V_{uv}^\Box \cup V_{vu}^\Box)$ and $k = |V_\triangle| + \sum_{(u,v) \in \mathrm{E}(G)} w(u,v)$. We use I to denote $(G', k, C, V_\triangle, V_\Box)$, which is an instance of Defensive Alliance$^{\mathrm{FNC}}$ and also of Exact Defensive Alliance$^{\mathrm{FNC}}$.

Clearly I can be computed in polynomial time. We now show that the treewidth of the primal graph of I depends only on the treewidth of G. We do so by modifying an optimal tree decomposition $\mathcal{T}$ of G as follows:

1. For each $(u,v) \in \mathrm{E}(G)$, we take an arbitrary node whose bag B contains both u and v and add to its children a chain of nodes $N_1, \dots, N_{w(u,v)-1}$ such that the bag of N_i is $B \cup \{u_i^v, u_{i+1}^v, v_i^u, v_{i+1}^u\}$.
2. For each $(u,v) \in \mathrm{E}(G)$, we take an arbitrary node whose bag B contains u and add to its children a chain of nodes $N_1, \dots, N_{w(u,v)}$ such that the bag of N_i is $B \cup \{u_i^{v\Box}\}$.
3. For each $(u,v) \in \mathrm{E}(G)$, we take an arbitrary node whose bag B contains v and add to its children a chain of nodes $N_1, \dots, N_{w(u,v)}$ such that the bag of N_i is $B \cup \{v_i^{u\Box}\}$.
4. For each $v \in \mathrm{V}(G)$, we take an arbitrary node whose bag B contains v and add to its children a chain of nodes $N_1, \dots, N_{r-1}$ such that the bag of N_i is $B \cup \{h_i^v\}$.

It is easy to verify that the result is a valid tree decomposition of the primal graph of I and its width is at most the treewidth of G plus four.

It remains to show that our reduction is correct. Obviously I is a positive instance of Defensive Alliance$^{\mathrm{FNC}}$ if and only if it is a positive instance of Exact Defensive

Alliance$^{\text{FNC}}$ because the forbidden, necessary and complementary vertices make sure that every solution of the Defensive Alliance$^{\text{FNC}}$ instance I has exactly k elements. Hence we only consider Defensive Alliance$^{\text{FNC}}$.

The intention is that for each orientation of G we have a solution candidate S in I such that an edge orientation from u to v entails $V_{vu} \subseteq S$ and $V_{uv} \cap S = \emptyset$, and the other orientation entails $V_{uv} \subseteq S$ and $V_{vu} \cap S = \emptyset$. For each vertex $v \in \mathrm{V}(G)$ and every incident edge $(v,u) \in \mathrm{E}(G)$ regardless of its orientation, the vertex v is attacked by the forbidden vertices $V_{vu}^{\square}$. So every vertex $v \in \mathrm{V}(G)$ has as least as many attackers as the sum of the weights of all incident edges. If in the orientation of G all edges incident to v are incoming edges, then each attack on v from $V_{vu}^{\square}$ can be repelled by V_{vu}, since $V_{vu} \subseteq S$. Due to the fact that the helper vertices H_v consist of exactly $2r-1$ elements, v can afford to have outgoing edges of total weight at most r.

We claim that (G, w, r) is a positive instance of Minimum Maximum Outdegree if and only if I is a positive instance of Defensive Alliance$^{\text{FNC}}$.

"Only if" direction. Let D be the directed graph given by an orientation of the edges of G such that for each vertex the sum of weights of outgoing edges is at most r. The set $S = V_{\triangle} \cup \{v_1^u, \dots, v_{w(u,v)}^u \mid (u,v) \in \mathrm{E}(D)\}$ is a defensive alliance in G': Let x be an arbitrary element of S. If x is an element of a set H_v or V_{uv}, then the only neighbor of x in G' is a necessary vertex, so x can trivially defend itself; so suppose $x \in \mathrm{V}(G)$. Let the sum of the weights of outgoing and incoming edges be denoted by w_{out}^x and w_{in}^x, respectively. The neighbors of x that are also in S consist of the elements of H_x and all elements of sets V_{xv} such that $(v,x) \in \mathrm{E}(D)$. Hence, including itself, x has $2r + w_{\text{in}}^x$ defenders in G'. The attackers of x consist of all elements of sets V_{xv} such that $(x,v) \in \mathrm{E}(D)$ (in total w_{out}^x) and all elements of sets $V_{xv}^{\square}$ such that either $(v,x) \in \mathrm{E}(D)$ or $(x,v) \in \mathrm{E}(D)$ (in total $w_{\text{in}}^x + w_{\text{out}}^x$). Hence x has $w_{\text{in}}^x + 2w_{\text{out}}^x$ attackers in G'. This shows that x has at least as many defenders as attackers, as by assumption $w_{\text{out}}^x \leqslant r$. Finally, it is easy to verify that $|S| = k$, $V_{\square} \cap S = \emptyset$, $V_{\triangle} \subseteq S$, and exactly one element of each pair of complementary vertices is in S.

"If" direction. Let S be a solution of I. For every $(u,v) \in \mathrm{E}(G)$, either $V_{uv} \subseteq S$ or $V_{vu} \subseteq S$ due to the complementary vertex pairs. We define a directed graph D by $\mathrm{V}(D) = \mathrm{V}(G)$ and $\mathrm{E}(D) = \{(u,v) \mid V_{vu} \subseteq S\} \cup \{(v,u) \mid V_{uv} \subseteq S\}$. Suppose there is a vertex x in D whose sum of weights of outgoing edges is greater than r. Clearly $x \in S$. Let the sum of the weights of outgoing and incoming edges be denoted by w_{out}^x and w_{in}^x, respectively. The defenders of x in G' beside itself consist of the elements of H_x and of w_{in}^x neighbors due to incoming edges in D. These are in total $2r + w_{\text{in}}^x$ defenders. The attackers of x in G' consist of $2w_{\text{out}}^x$ elements (of the form x_i^v as well as $x_i^{v\square}$) due to outgoing edges in D and w_{in}^x elements (of the form $x_i^{v\square}$) due to incoming edges. These are in total $2w_{\text{out}}^x + w_{\text{in}}^x$ attackers. But then x has more attackers than defenders, as by assumption $w_{\text{out}}^x > r$. □

4.4.2 Hardness of Defensive Alliance with Forbidden and Necessary Vertices

Now we reduce from Defensive Alliance$^{\text{FNC}}$ to Defensive Alliance$^{\text{FN}}$ to show W[1]-hardness of the latter problem. Since Secure Set$^{\text{FNC}}$ and Secure Set$^{\text{FN}}$ have the same instances as Defensive Alliance$^{\text{FNC}}$ and Defensive Alliance$^{\text{FN}}$, respectively, we can reuse the function τ^{FNC} from Definition 4.8. First we show that τ^{FNC} indeed specifies a correct reduction from Defensive Alliance$^{\text{FNC}}$ to Defensive Alliance$^{\text{FN}}$.

Lemma 4.30. *Let $I = (G, k, V_\square, V_\triangle, C)$ be a* Defensive Alliance$^{\text{FNC}}$ *instance, let A be the set of solutions of I and let B be the set of solutions of the* Defensive Alliance$^{\text{FN}}$ *instance $\tau^{FNC}(I)$. There is a bijection $f : A \to B$ such that $|f(S)| = \sigma_I^{FNC}(|S|)$ holds for every $S \in A$.*

Proof. We use the same auxiliary notation as in Definition 4.8 and we define f as $S \mapsto S \cup \bigcup_{v \in S} Y_v^\circ \cup \bigcup_{(a,b) \in C,\, x \in S \cap \{a,b\}} (\{\triangle^{ab}, x^{ab}\} \cup Z_{x\circ}^{ab})$. For every $S \in A$, we thus obtain $|f(S)| = \sigma_I^{\text{FNC}}(|S|)$, and we first show that indeed $f(S) \in B$.

Let $S \in A$ and let S' denote $f(S)$. Obviously S' satisfies $V'_\square \cap S' = \emptyset$ and $V'_\triangle \subseteq S'$. To see that S' is a defensive alliance in G', let x be an arbitrary element of S'. If $x \notin S$, then x clearly has as least as many neighbors in S' as neighbors not in S' by construction of f, so suppose $x \in S$. There is a defense $\mu : N_G[x] \setminus S \to N_G[x] \cap S$ since S is a defensive alliance in G. We use this to construct a defense $\mu' : N_{G'}[x] \setminus S' \to N_{G'}[x] \cap S'$. For any attacker v of x in G', we distinguish two cases.

- If v is some $x_i^\square \in Y_x^\square$ for some $x \in \mathrm{V}(G)$, we set $\mu'(v) = x_i$. This element is in $N_{G'}[x]$ by construction.
- Otherwise v is in $N_G[x] \setminus S$ (by our construction of S'). Since the codomain of μ is a subset of the codomain of μ', we may set $\mu'(v) = \mu(v)$.

Since μ' is injective, each attack on x in G' can be repelled by S'. Hence S' is a defensive alliance in G'.

Clearly f is injective. It remains to show that f is surjective. Let S' be a solution of $\tau^{\text{FNC}}(I)$. As in the proof of Lemma 4.9, for every $(a,b) \in C$, we can see that S' contains either a or b, and that for each $x \in \{a,b\}$, S' contains either all or none of $\{x, x^{ab}\} \cup Y_x^\circ \cup Z_{x\circ}^{ab}$. Moreover, S' contains either all or none of $\{v, v_1, \dots, v_n\}$ for every $v \in \mathrm{V}(G)$.

We construct $S = S' \cap \mathrm{V}(G)$ and observe that $S' = f(S)$, $V_\triangle \subseteq S$, $V_\square \cap S = \emptyset$, and $|S \cap \{a,b\}| = 1$ for each $(a,b) \in C$. It remains to show that S is a defensive alliance in G. Let x be an arbitrary element of S. We observe that $N_{G'}[x] \cap S' = (N_G[x] \cap S) \cup Y_x^\circ$ and similarly $N_{G'}[x] \setminus S' = (N_G[x] \setminus S) \cup Y_x^\square$. Since the cardinality of each set Y_x° is equal to the cardinality of $Y_x^\square$, this implies $|N_{G'}[x] \cap S'| - |N_G[x] \cap S| = |N_{G'}[x] \setminus S'| -$

$|N_G[x] \setminus S|$. Since S' is a defensive alliance in G' and $x \in S'$, it holds that $|N_{G'}[x] \cap S'| \geqslant |N_{G'}[x] \setminus S'|$. We conclude that $|N_G[x] \cap S| \geqslant |N_G[x] \setminus S|$. Hence S is a defensive alliance in G. □

To obtain the hardness result for Defensive Alliance$^{\mathrm{FN}}$ parameterized by treewidth, it remains to show that the reduction specified by τ^{FNC} preserves bounded treewidth.

Lemma 4.31. Defensive Alliance$^{\mathrm{FN}}$*, parameterized by the treewidth of the graph, is* W[1]*-hard.*

Proof. Let I be a Defensive Alliance$^{\mathrm{FNC}}$ instance whose primal graph we denote by G. We obtain an equivalent Defensive Alliance$^{\mathrm{FN}}$ instance $\tau^{\mathrm{FNC}}(I)$, whose graph we denote by G'. This reduction is correct, as shown in Lemma 4.30. It remains to show that the treewidth of G' is bounded by a function of the treewidth of G. Let $\mathcal{T}$ be an optimal nice tree decomposition of G. We build a tree decomposition $\mathcal{T}'$ of G' by modifying a copy of $\mathcal{T}$ in the following way: For each vertex $v \in \mathrm{V}(G)$, we add v_n and $v_n^{\square}$ to every bag containing v. Then we pick an arbitrary node t in $\mathcal{T}$ whose bag contains v, and we add new children $N_1, \dots, N_{n-1}$ to t such that the bag of N_i is $\{v, v_i, v_i^{\square}, v_{i+1}, v_{i+1}^{\square}\}$. Next, for every pair (a, b) of complementary vertices, we pick an arbitrary node t in $\mathcal{T}$ whose bag B contains both a_n and b_n, and we add a chain of nodes $N_1, \dots, N_{2n^2+2n-1}$ between t and its parent such that, for $1 \leqslant i < n^2 + n$, the bag of N_i is $B \cup \{a^{ab}, a_i^{ab}, a_i^{ab\square}, a_{i+1}^{ab}, a_{i+1}^{ab\square}\}$, the bag of N_{n^2+n} is $B \cup \{a^{ab}, b^{ab}, \triangle^{ab}\}$, and the bag of N_{n^2+n+i} is $B \cup \{b^{ab}, b_{n^2+n+1-i}^{ab}, b_{n^2+n+1-i}^{ab\square}, b_{n^2+n-i}^{ab}, b_{n^2+n-i}^{ab\square}\}$. It is easy to verify that $\mathcal{T}'$ is a valid tree decomposition of G'. Furthermore, the width of $\mathcal{T}'$ is at most three times the width of $\mathcal{T}$ plus five. □

Just like before, we get an analogous result for the exact variant. It can be proved in the same way as Lemma 4.11.

Lemma 4.32. Exact Defensive Alliance$^{\mathrm{FN}}$*, parameterized by the treewidth of the graph, is* W[1]*-hard.*

4.4.3 Hardness of Defensive Alliance with Forbidden Vertices

We next show W[1]-hardness of Defensive Alliance$^{\mathrm{F}}$ by reducing from Defensive Alliance$^{\mathrm{FN}}$ reusing the function τ^{FN} from Definition 4.12. This function maps a Defensive Alliance$^{\mathrm{FN}}$ instance, together with an ordering $\preceq$ of the non-forbidden vertices, to a Defensive Alliance$^{\mathrm{F}}$ instance. We show that by choosing $\preceq$ appropriately, this gives us a reduction that preserves bounded treewidth. First we show that τ^{FN} indeed specifies a correct reduction from Defensive Alliance$^{\mathrm{FN}}$ to Defensive Alliance$^{\mathrm{F}}$ for any ordering $\preceq$.

Lemma 4.33. *Let $I = (G, k, V_\Box, V_\triangle)$ be a* Defensive Alliance$^{\text{FN}}$ *instance, let $\preceq$ be an ordering of* $\mathrm{V}(G) \setminus V_\Box$*, let A be the set of solutions of I and let B be the set of solutions of the* Defensive Alliance$^{\text{F}}$ *instance $\tau^{FN}(I, \preceq)$. There is a bijection $f : A \to B$ such that $|f(S)| = \sigma_I^{FN}(|S|)$ holds for every $S \in A$.*

Proof. We use the same auxiliary notation as in Definition 4.12 and we define f as

$$f(S) = S \cup \bigcup_{v \in S} A_v^\circ \cup \{v', h_v \mid v \in V_\circ\} \cup \bigcup_{v \in V_\circ} A_{v'}^\circ \cup \{g_v \mid v \in S \cap V_\circ\}.$$

For every $S \in A$, we thus obtain $|f(S)| = |S| + |S|(n+1) + 2|V_\circ| + |V_\circ| \cdot (n+1) + (|S| - |V_\triangle|) = \sigma_I^{\text{FN}}(|S|)$, and we first show that indeed $f(S) \in B$.

Let $S \in A$ and let S' denote $f(S)$. Obviously S' satisfies $V'_\Box \cap S' = \emptyset$. To see that S' is a defensive alliance in G', let x be an arbitrary element of S'. If $x \in S$, then there is a defense $\mu : N_G[x] \setminus S \to N_G[x] \cap S$ since S is a defensive alliance in G. We use this to construct a defense $\mu' : N_{G'}[x] \setminus S' \to N_{G'}[x] \cap S'$. For any attacker a of x in G', we distinguish the following cases:

- If a is some $v_i^\Box \in A_v^\Box$ for some $v \in V^+$, then x is either v_i or a neighbor of v_i, all of which are in S', and we set $\mu'(a) = v_i$.
- Similarly, if a is $g_v^\Box$ for some $v \in V_\circ$, then we set $\mu'(a) = g_v$.
- If a is $h_v^\Box$ for some $v \in V_\circ$, then $x = g_v$ and we set $\mu'(a) = h_v$.
- If a is g_v for some $v \in V_\circ$, then x is either v' or h_v, which is not used for repelling any other attack because $h_v^\Box$ cannot attack x, so we set $\mu'(a) = h_v$.
- Otherwise a is in $N_G[x] \setminus S$ (by our construction of S'). Since the codomain of μ is a subset of the codomain of μ', we may set $\mu'(a) = \mu(a)$.

Since μ' is injective, each attack on x in G' can be repelled by S'. Hence S' is a defensive alliance in G'.

Clearly f is injective. It remains to show that f is surjective. Let S' be a solution of $\tau^{\text{FN}}(I, \preceq)$. As in the proof of Lemma 4.13, we can see that $V_\triangle \cup \{v', h_v \mid v \in V_\circ\} \subseteq S'$. Let $S = S' \cap \mathrm{V}(G)$. By the previous observations, it is easy to see that $S' = f(S)$. It remains to show that S is a defensive alliance in G. Let x be an arbitrary element of S. We observe that $N_{G'}[x] \cap S' = (N_G[x] \cap S) \cup A_x^\circ$ and similarly $N_{G'}[x] \setminus S' = (N_G[x] \setminus S) \cup A_v^\Box$. Since $|A_x^\circ| = |A_x^\Box|$, this implies $|N_{G'}[x] \cap S'| - |N_G[x] \cap S| = |N_{G'}[x] \setminus S'| - |N_G[x] \setminus S|$. Since S' is a defensive alliance in G' and $x \in S'$, it holds that $|N_{G'}[x] \cap S'| \geqslant |N_{G'}[x] \setminus S'|$. We conclude that $|N_G[x] \cap S| \geqslant |N_G[x] \setminus S|$. Hence S is a defensive alliance in G. □

To obtain the hardness result for Defensive Alliance$^{\mathrm{F}}$ parameterized by treewidth, it remains to show that we can compute an ordering $\preceq$ in FPT time such that the reduction specified by τ^{FN} together with $\preceq$ preserves bounded treewidth.

Lemma 4.34. Defensive Alliance$^{\mathrm{F}}$*, parameterized by the treewidth of the graph, is* W[1]*-hard.*

Proof. Let $I = (G, k, V_\square, V_\triangle)$ be a Defensive Alliance$^{\mathrm{FN}}$ instance and let $\mathcal{T}$ be an optimal nice tree decomposition of G. We can compute such a tree decomposition in FPT time (Bodlaender 1996). Let $\preceq$ be the ordering of the elements of $V_\triangle \cup V_\circ$ that is obtained in linear time by doing a post-order traversal of $\mathcal{T}$ and sequentially recording the elements that occur for the last time in the current bag. We obtain the Defensive Alliance$^{\mathrm{F}}$ instance $\tau^{\mathrm{FN}}(I, \preceq)$, whose graph we denote by G'. This reduction is correct, as shown in Lemma 4.33, and computable in FPT time. It remains to show that the treewidth of G' is bounded by a function of the treewidth of G. To this end, we use $\mathcal{T}$ to build a tree decomposition $\mathcal{T}'$ of G'. We initially set $\mathcal{T}' := \mathcal{T}$ and modify it by the following steps:

1. For each $v \in V_\circ$, we add g_v, $g_v^\square$, h_v, $h_v^\square$ and v' to the bag of $t_v^{\mathcal{T}'}$. Note that afterwards $t_v^{\mathcal{T}'} = t_{v'}^{\mathcal{T}'}$. After this step we increased the width of $\mathcal{T}'$ by at most five.
2. For each $v \in V^+$, we use B_v to denote the bag of $t_v^{\mathcal{T}'}$ and replace $t_v^{\mathcal{T}'}$ by a chain of nodes $N_1, \dots, N_n$, where N_n is the topmost node and the bag of N_i is $B_v \cup \{v_i, v_i^\square, v_{i+1}, v_{i+1}^\square\}$. After this step we increased the width of $\mathcal{T}'$ by at most nine. Note that the bag of the new node $t_v^{\mathcal{T}'}$ now contains v_{i+1} and $v_{i+1}^\square$. We have so far covered all edges except the ones connecting elements of two different sets A_x and A_y for $(x, y) \in P$.
3. For every $(u, v) \in P$, we add v_1 and $v_1^\square$ into the bag of every node between (and including) $t_u^{\mathcal{T}'}$ and $t_{v_1}^{\mathcal{T}'}$. Note that this preserves connectedness and afterwards the bag of $t_u^{\mathcal{T}'}$ contains u_{i+1}, $u_{i+1}^\square$, v_1 and $v_1^\square$, thus covering the remaining edges. After this step we increased the width of $\mathcal{T}'$ by at most 13. (Since the number of children of each tree decomposition node is at most two, this step enlarges every bag at most twice.)

It is easy to verify that $\mathcal{T}'$ is a valid tree decomposition of G'. Furthermore, the width of $\mathcal{T}'$ is at most the width of $\mathcal{T}$ plus 13. $\square$

We again get an analogous result for the exact variant.

Corollary 4.35. Exact Defensive Alliance$^{\mathrm{F}}$*, parameterized by the treewidth of the graph, is* W[1]*-hard.*

4.4.4 Hardness of Defensive Alliance

Next we show W[1]-hardness of Defensive Alliance by reducing from Defensive Alliance$^{\mathrm{F}}$ while preserving bounded treewidth.

Lemma 4.36. Defensive Alliance, *parameterized by the treewidth of the graph, is* W[1]-*hard.*

Proof. Let $I = (G, k, V_\square)$ be a Defensive Alliance$^{\mathrm{F}}$ instance, let G' denote the graph of $\tau^{\mathrm{F}}(I)$, where τ^{F} is the function from Definition 4.15, and let $\mathcal{T}$ be an optimal nice tree decomposition of G. We build a tree decomposition $\mathcal{T}'$ of G' by modifying a copy of $\mathcal{T}$ in the following way: For every $f \in V_\square$, we pick an arbitrary node t in $\mathcal{T}$ whose bag B contains f, and we add a chain of nodes $N_1, \dots, N_{2k}$ between t and its parent such that, for $1 \leqslant i \leqslant 2k$, the bag of N_i is $B \cup \{f', f_i\}$. It is easy to verify that $\mathcal{T}'$ is a valid tree decomposition of G'. Furthermore, the width of $\mathcal{T}'$ is at most the width of $\mathcal{T}$ plus two. Finally, we can prove that $\tau^{\mathrm{F}}(I)$ indeed specifies a correct reduction from Defensive Alliance$^{\mathrm{F}}$ to Defensive Alliance in the same way as in the proof of Lemma 4.16. □

We again get an analogous result for the exact variant.

Corollary 4.37. Exact Defensive Alliance, *parameterized by the treewidth of the input graph, is* W[1]-*hard.*

4.4.5 Hardness of Secure Set and Its Variants

Finally, hardness of Secure Set (and its variants) carries over from our hardness result for Defensive Alliance (Lemma 4.36) by our reduction from Defensive Alliance to Secure Set (Corollary 4.26). All of these reductions preserve bounded treewidth, as we have seen. We also get a result for Exact Secure Set by an analogous argument.

Corollary 4.38. Secure Set *and* Exact Secure Set *are both* W[1]-*hard when parameterized by the treewidth of the input graph.*

This proves Theorem 4.28.

4.4.6 Fixed-Parameter Tractability of Secure Set Verification

We have seen that treewidth does not lead to fixed-parameter tractability for *finding* secure sets. Now we prove an FPT result for the co-NP-complete problem of *checking* whether a given set is secure. We do this by providing an encoding to guarded ASP (see Chapter 3) and invoking Theorem 3.22.

Listing 4.1 presents a guarded ASP program that encodes an optimization variant of Secure Set Verification. It finds subsets X of a given set S of vertices in a graph G

```
 x(S) ∨ nx(S) ← s(S).
neighbor(V) ← x(X), e(X,V).
    good(V) ← v(V), x(V).
    good(V) ← neighbor(V), s(V).
     bad(V) ← neighbor(V), v(V), not s(V).
            ⇜ v(V), good(V).  [1,V]
            ⇜ v(V), bad(V).  [-1,V]
```

Listing 4.1: A guarded ASP encoding for an optimization variant of Secure Set Verification

such that $|N_G[X] \cap S| - |N_G[X] \setminus S|$ is as small as possible. The graph G is specified as follows: For each vertex v of G there is an input fact $\texttt{v}(v)$, for each edge (a, b) of G there are input facts $\texttt{e}(a, b)$ and $\texttt{e}(b, a)$, and for each element x of S there is an input fact $\texttt{s}(x)$. First the encoding guesses a subset X of S, indicated by the `x` predicate. Then it defines $N(X)$ using the `neighbor` predicate, and it defines $N[X] \cap S$ and $N[X] \setminus S$ using the `good` and `bad` predicates, respectively. Finally, it instructs the ASP solver to minimize the value of $|N[X] \cap S| - |N[X] \setminus S|$ via the weak constraints. If the solver produces an answer set where this value is negative, then clearly the answer set witnesses that the set S is not secure in G. Otherwise for any $X \subseteq S$ this value is nonnegative, which means that S is secure.

Since this ASP encoding is clearly guarded, the treewidth of the primal graph of the grounding given an input graph G only depends on the treewidth of G by Theorem 3.10. Furthermore, by Theorem 3.22 this proves that Secure Set Verification is fixed-parameter tractable when parameterized by the treewidth of the input graph. In fact, we get the following fixed-parameter linearity result from Corollary 3.24:[1]

Theorem 4.39. Secure Set Verification *parameterized by the treewidth of the graph can be solved in fixed-parameter linear time.*

It is also possible to obtain this FPT result by expressing the problem in EMSO (extended monadic second-order logic; see Section 3.6) using the same idea. For this, consider the following formula $\mathrm{ssv}(X, G, B)$, which is true if $X \subseteq S$, $G = N[X] \cap S$ and $B = N[X] \setminus S$. We assume that the input is given as a relational structure $\mathcal{G}$, where the edges of the graph are specified using the edge relation E and the vertices in S are indicated by the

[1]Strictly speaking, Corollary 3.24 considers the Brave Reasoning problem, whereas here we consider a slightly different problem where we want to check if the cost of optimal answer sets is nonnegative. However, the algorithm used in the proof of Theorem 3.22, upon which Corollary 3.24 is based, can clearly be adapted to this variant.

```
s(V) ∨ ns(V) ← v(V).
← v(V), s(V), #sum{ 1,X : e(V,X), s(X); -1,X : e(V,X), ns(X) } < -1.
⇜ s(X).  [1,X]
```

Listing 4.2: A connection-guarded ASP encoding for Defensive Alliance

unary S relation.

$$\begin{aligned}\mathrm{ssv}(X,G,B) \;\equiv\; & \forall v\big(v \in X \rightarrow S(v)\big)\\ & \wedge \forall v\Big(v \in G \leftrightarrow v \in X \vee \exists x\big(x \in X \wedge \mathrm{E}(v,x)\big)\Big)\\ & \wedge \forall v\Big(v \in B \leftrightarrow \neg S(v) \wedge \exists x\big(x \in X \wedge \mathrm{E}(v,x)\big)\Big)\end{aligned}$$

If we combine this formula with the function $\alpha(|X|,|G|,|B|) = |G| - |B|$, then by a result of Arnborg, Lagergren and Seese (1991) we can find the minimum of this function for any X, G and B such that $\mathrm{ssv}(X,G,B)$ is true under $\mathcal{G}$ in FPT time. As before, we can thus decide Secure Set Verification by checking if this value is negative.

4.4.7 Fixed-Parameter Tractability of Defensive Alliance Parameterized by Treewidth and Degree

In Section 4.4.6, we have proved that Secure Set Verification is fixed-parameter tractable when parameterized by treewidth by expressing the problem in guarded ASP. We have also seen that the same result can be shown with extended monadic second-order logic (EMSO), which is not surprising since we suspect that guarded ASP is strictly less expressive than EMSO if the input structures do not contain numbers, as we explained in Section 3.6. Next we briefly give an example of a problem where connection-guarded ASP allows us to obtain an FPT result (for the combination of treewidth and degree as the parameter) that cannot be achieved from any extension of MSO that we are aware of.

Listing 4.2 presents a connection-guarded ASP encoding of the Defensive Alliance problem. We have shown this problem to be W[1]-hard when parameterized by treewidth alone in Section 4.4.4. It is already known (Enciso 2009) that this problem is FPT when parameterized by the combination of treewidth and degree. Hence the following theorem is not new, but it serves as an example of FPT results that can be obtained by connection-guarded ASP but not, to the best of our knowledge, with extensions of MSO found in the literature.[2]

Theorem 4.40. Defensive Alliance *parameterized by the combination of the treewidth and the degree of the graph can be solved in fixed-parameter linear time.*

[2]Here the same remarks as for Theorem 4.39 apply: Theorem 3.23 and Corollary 3.25 actually concern the Brave Reasoning problem, but again we can easily adapt the algorithm so that it allows us to check the cost of optimal answer sets.

4.5 Algorithms for Alliance Problems on Graphs of Bounded Treewidth

In this section, we present some positive results in the form of algorithms that exploit bounded treewidth to obtain (fixed-parameter) tractability.

First, we provide a fixed-parameter linear algorithm that solves the Secure Set Verification problem via dynamic programming on a tree decomposition of the input graph. We have already seen in Theorem 4.39 that this problem is solvable in fixed-parameter linear time, but we obtained this result by metatheorems that, when implemented, yield algorithms with practically quite bad running time. Indeed, Courcelle's theorem is widely considered to be primarily useful as a classification tool, but if efficiency is a concern, manually crafted algorithms are to be preferred (Cygan et al. 2015). Hence our dynamic programming algorithm for Secure Set Verification is likely much more efficient than the algorithm implicit in proofs of Courcelle's theorem.

Second, we show that all the variants of both Secure Set and Defensive Alliance considered in this paper are solvable in polynomial time on instances whose treewidth is bounded by a constant. We again do this by providing a polynomial-time dynamic programming algorithm, but this time the degree of the polynomial depends on the treewidth.

4.5.1 A Fixed-Parameter Tractable Algorithm for Secure Set Verification

While we have seen in Section 4.4 that Secure Set parameterized by treewidth is most likely not FPT, we now present a positive result: The co-NP-complete (Ho 2011) Secure Set Verification problem, which consists of checking whether a given set $\widehat{S}$ is secure in a graph G, is FPT parameterized by the treewidth of G. We show this by giving a fixed-parameter linear algorithm that follows the principle of dynamic programming on a tree decomposition $\mathcal{T}$ of G. The core idea is the following: For each node t of $\mathcal{T}$ and each $X \subseteq \widehat{S} \cap \chi(t)$, we store an integer $c_{\widehat{S},t}(X)$, which indicates that X can be extended to a set $\widehat{X} \subseteq \widehat{S}$ using "forgotten" vertices from further down in $\mathcal{T}$ in such a way that the difference between defenders and attackers of $\widehat{X}$ is $c_{\widehat{S},t}(X)$ and $\widehat{X}$ is the "worst" subset of $\widehat{S}$ that can be obtained in this way. To compute these values, we traverse $\mathcal{T}$ from the bottom up and use recurrence relations to compute the values for the current node t of $\mathcal{T}$ based on the values we have computed for the children of t. If we then look at the values we have computed at the root of $\mathcal{T}$, we can decide if there is a subset of $\widehat{S}$ that is "bad enough" to witness that $\widehat{S}$ is not secure.

Dynamic programming algorithms like this are quite common for showing FPT membership w.r.t. treewidth and some examples can be found in the book by Niedermeier (2006). Proving their correctness is a usually rather tedious structural induction argument along the tree decomposition: At every node t of $\mathcal{T}$, we have to prove that the recurrence relations indeed characterize the value they are supposed to represent. Examples of such proofs can be found in the book by Cygan et al. (2015).

We now formally define the values that we will compute at each tree decomposition node. Let G be a graph with a nice tree decomposition $\mathcal{T}$ and let $\widehat{S} \subseteq V(G)$ be the candidate for which we want to check if it is secure. For each node t of $\mathcal{T}$ and each set of vertices A, we define $A_t = \{a \in A \mid a \in \chi(t'),\ t' \text{ is a descendant of } t\}$. For any $\widehat{X} \subseteq \widehat{S}$, we call $|N_G[\widehat{X}]_t \cap \widehat{S}| - |N_G[\widehat{X}]_t \setminus \widehat{S}|$ the *score* of $\widehat{X}$ w.r.t. $\widehat{S}$ at t (or just the score of $\widehat{X}$ if $\widehat{S}$ and t are clear from the context) and denote it by $\operatorname{score}_{\widehat{S},t}(\widehat{X})$. Furthermore, we call $|N_G[\widehat{X}] \cap \chi(t) \cap \widehat{S}| - |(N_G[\widehat{X}] \cap \chi(t)) \setminus \widehat{S}|$ the *local score* of $\widehat{X}$ w.r.t. $\widehat{S}$ at t and denote it by $\operatorname{lscore}_{\widehat{S},t}(\widehat{X})$. Finally, for each $X \subseteq \widehat{S} \cap \chi(t)$, we define the value

$$c_{\widehat{S},t}(X) = \min_{\widehat{X} \subseteq \widehat{S}_t,\ \widehat{X} \cap \chi(t) = X} \{\operatorname{score}_{\widehat{S},t}(\widehat{X})\}.$$

When r is the root of $\mathcal{T}$, both $\widehat{S}_r = \widehat{S}$ and $\chi(r) = \emptyset$ hold, so $\widehat{S}$ is secure if and only if $c_{\widehat{S},r}(\emptyset)$ is nonnegative.

We now describe how to compute all such values in a bottom-up manner by distinguishing the node type of t, and we prove the correctness of our computation by structural induction along the way. In this correctness proof, we use additional terminology: We say that a set $\widehat{X}$ is an *extension* of X w.r.t. $\widehat{S}$ at t if it is one of those sets considered in the definition of $c_{\widehat{S},t}(X)$ that has minimum score; formally $\widehat{X} \subseteq \widehat{S}_t$, $\widehat{X} \cap \chi(t) = X$ and $\operatorname{score}_{\widehat{S},t}(\widehat{X}) = c_{\widehat{S},t}(X)$. We may omit $\widehat{S}$ or t if they are clear from the context.

Leaf node. If t is a leaf node, then its bag is empty and obviously $c_{\widehat{S},t}(\emptyset) = 0$ holds.

Introduce node. Let t be an introduce node with child t', let v be the unique element of $\chi(t) \setminus \chi(t')$, let $X \subseteq \widehat{S} \cap \chi(t)$ and let $X' = X \setminus \{v\}$. We prove that the following equation holds:

$$c_{\widehat{S},t}(X) = \begin{cases} c_{\widehat{S},t'}(X') + 1 & \text{if } v \in N_G[X] \cap \widehat{S} \\ c_{\widehat{S},t'}(X') - 1 & \text{if } v \in N_G[X] \setminus \widehat{S} \\ c_{\widehat{S},t'}(X') & \text{otherwise} \end{cases}$$

First consider the case where $v \in N_G[X] \cap \widehat{S}$. Let $\widehat{X}$ be an extension of X at t, so $\operatorname{score}_{\widehat{S},t}(\widehat{X}) = c_{\widehat{S},t}(X)$. From $v \notin N_G[\widehat{X} \setminus \{v\}]_{t'}$ and $v \in N_G[\widehat{X}]_t \cap \widehat{S}$ we infer $\operatorname{score}_{\widehat{S},t}(\widehat{X}) = \operatorname{score}_{\widehat{S},t'}(\widehat{X} \setminus \{v\}) + 1$. Moreover, the set $\widehat{X} \setminus \{v\}$ is one of the candidates considered for an extension of X' in the definition of $c_{\widehat{S},t'}$, so we obtain $c_{\widehat{S},t'}(X') \leqslant \operatorname{score}_{\widehat{S},t'}(\widehat{X} \setminus \{v\})$. In total, this gives us $c_{\widehat{S},t}(X) \geqslant c_{\widehat{S},t'}(X') + 1$. Conversely, let $\widehat{X'}$ be an extension of X' at t', so $\operatorname{score}_{\widehat{S},t'}(\widehat{X'}) = c_{\widehat{S},t'}(X')$. We distinguish two cases.

1. If $v \in X$, then from $v \notin N_G[\widehat{X'}]_{t'}$ and $v \in N_G[\widehat{X'} \cup \{v\}]_t \cap \widehat{S}$ we infer $\mathrm{score}_{\widehat{S},t}(\widehat{X'} \cup \{v\}) = \mathrm{score}_{\widehat{S},t'}(\widehat{X'}) + 1$. Since $X = X' \cup \{v\}$ and $X' = \widehat{X'} \cap \chi(t')$, it holds that $X = (\widehat{X'} \cup \{v\}) \cap \chi(t)$. Hence the set $\widehat{X'} \cup \{v\}$ is one of the candidates considered for an extension of X in the definition of $c_{\widehat{S},t}$ and we obtain $c_{\widehat{S},t}(X) \leqslant \mathrm{score}_{\widehat{S},t}(\widehat{X'} \cup \{v\})$.
2. Otherwise $v \notin X$. In this case $X = X'$, $v \notin \widehat{X'}$ and $X = \widehat{X'} \cap \chi(t)$. Hence the set $\widehat{X'}$ is considered in the definition of $c_{\widehat{S},t}(X)$ and we get $c_{\widehat{S},t}(X) \leqslant \mathrm{score}_{\widehat{S},t}(\widehat{X'})$. Since v is adjacent to an element of X, we infer $\mathrm{score}_{\widehat{S},t}(\widehat{X'}) = \mathrm{score}_{\widehat{S},t'}(\widehat{X'}) + 1$.

In both cases, we obtain $c_{\widehat{S},t}(X) \leqslant c_{\widehat{S},t'}(X') + 1$, so indeed $c_{\widehat{S},t}(X) = c_{\widehat{S},t'}(X') + 1$.

Next consider the case where $v \in N_G[X] \setminus \widehat{S}$. Clearly $v \notin X$. Let $\widehat{X}$ be an extension of X at t, so $\mathrm{score}_{\widehat{S},t}(\widehat{X}) = c_{\widehat{S},t}(X)$. From $v \notin N_G[\widehat{X}]_{t'}$ and $v \in N_G[\widehat{X}]_t \setminus \widehat{S}$ we now infer $\mathrm{score}_{\widehat{S},t}(\widehat{X}) = \mathrm{score}_{\widehat{S},t'}(\widehat{X}) - 1$. Similar to before, by definition of $c_{\widehat{S},t'}(X')$ we obtain $c_{\widehat{S},t'}(X') \leqslant \mathrm{score}_{\widehat{S},t'}(\widehat{X})$. In total, this gives us $c_{\widehat{S},t}(X) \geqslant c_{\widehat{S},t'}(X') - 1$. Conversely, let $\widehat{X'}$ be an extension of X' at t', so $\mathrm{score}_{\widehat{S},t'}(\widehat{X'}) = c_{\widehat{S},t'}(X')$. Since $v \notin \widehat{X'}$ and $X = \widehat{X'} \cap \chi(t)$, $\widehat{X'}$ is considered in the definition of $c_{\widehat{S},t}(X)$ and we get $c_{\widehat{S},t}(X) \leqslant \mathrm{score}_{\widehat{S},t}(\widehat{X'})$. Since v is adjacent to an element of X, we infer $\mathrm{score}_{\widehat{S},t}(\widehat{X'}) = \mathrm{score}_{\widehat{S},t'}(\widehat{X'}) - 1$. We obtain $c_{\widehat{S},t}(X) \leqslant c_{\widehat{S},t'}(X') - 1$, so indeed $c_{\widehat{S},t}(X) = c_{\widehat{S},t'}(X') - 1$.

Finally consider the remaining case where $v \notin N_G[X]$ and, in particular, $v \notin X$ holds as well as $X = X'$. Using elementary set theory with $\widehat{S}_t \setminus \{v\} = \widehat{S}_{t'}$ and $\chi(t) = \chi(t') \cup \{v\}$ in mind, we can prove that $\{\widehat{X} \subseteq \widehat{S}_t \mid \widehat{X} \cap \chi(t) = X\}$ is equal to $\{\widehat{X} \subseteq \widehat{S}_{t'} \mid \widehat{X} \cap \chi(t') = X'\}$. Hence a set $\widehat{X}$ is considered in the definition of $c_{\widehat{S},t}(X)$ if and only if it is considered in the definition of $c_{\widehat{S},t'}(X')$. For every $\widehat{X} \subseteq \widehat{S}_t$ such that $\widehat{X} \cap \chi(t) = X$, observe that $v \notin N_G[\widehat{X}]_t$, since v is not adjacent to any element of X and if it were adjacent to some element of $\widehat{X} \setminus X$, then $\mathcal{T}$ would not be a valid tree decomposition. This proves that every such $\widehat{X}$ has the same score at t and t'. Hence $c_{\widehat{S},t}(X) = c_{\widehat{S},t'}(X')$.

Forget node. Let t be a forget node with child t', let v be the unique element of $\chi(t') \setminus \chi(t)$ and let $X \subseteq \widehat{S} \cap \chi(t)$. We prove that the following equation holds:

$$c_{\widehat{S},t}(X) = \begin{cases} \min\{c_{\widehat{S},t'}(X),\ c_{\widehat{S},t'}(X \cup \{v\})\} & \text{if } v \in \widehat{S} \\ c_{\widehat{S},t'}(X) & \text{otherwise} \end{cases}$$

Clearly $\widehat{S}_t = \widehat{S}_{t'}$ and all scores at forget nodes are identical to those in the respective child node. The case where $v \notin \widehat{S}$ is trivial as then $\widehat{S} \cap \chi(t) = \widehat{S} \cap \chi(t')$, i.e., the domains of $c_{\widehat{S},t}$ and $c_{\widehat{S},t'}$ are equal, and the sets considered in the definitions of $c_{\widehat{S},t}(X)$ and $c_{\widehat{S},t'}(X)$ are the same. Hence we consider the case where $v \in \widehat{S}$.

Let $\widehat{X}$ be an extension of X at t, so $c_{\widehat{S},t}(X) = \mathrm{score}_{\widehat{S},t}(\widehat{X}) = \mathrm{score}_{\widehat{S},t'}(\widehat{X})$. If $v \notin \widehat{X}$, then $\widehat{X} \cap \chi(t') = X$, so we obtain $c_{\widehat{S},t'}(X) \leqslant \mathrm{score}_{\widehat{S},t'}(\widehat{X})$ by definition of $c_{\widehat{S},t'}(X)$. On the other hand, if $v \in \widehat{X}$, then $\widehat{X} \cap \chi(t') = X \cup \{v\}$, so we obtain $c_{\widehat{S},t'}(X \cup \{v\}) \leqslant \mathrm{score}_{\widehat{S},t'}(\widehat{X})$. As one of these two inequalities applies, we get $c_{\widehat{S},t}(X) \geqslant \min\{c_{\widehat{S},t'}(X),\ c_{\widehat{S},t'}(X \cup \{v\})\}$.

Conversely, every extension $\widehat{X'}$ of X at t' is considered in the definition of $c_{\widehat{S},t}(X)$, so $c_{\widehat{S},t}(X) \leqslant \mathrm{score}_{\widehat{S},t}(\widehat{X'}) = \mathrm{score}_{\widehat{S},t'}(\widehat{X'}) = c_{\widehat{S},t'}(X)$. Moreover, every extension $\widehat{X'}$ of $X \cup \{v\}$ at t' is also considered in the definition of $c_{\widehat{S},t}(X)$, so $c_{\widehat{S},t}(X) \leqslant \mathrm{score}_{\widehat{S},t}(\widehat{X'}) = \mathrm{score}_{\widehat{S},t'}(\widehat{X'}) = c_{\widehat{S},t'}(X \cup \{v\})$. If we combine these two inequalities, we get $c_{\widehat{S},t}(X) \leqslant \min\{c_{\widehat{S},t'}(X),\ c_{\widehat{S},t'}(X \cup \{v\})\}$. Hence $c_{\widehat{S},t}(X) = \min\{c_{\widehat{S},t'}(X),\ c_{\widehat{S},t'}(X \cup \{v\})\}$.

Join node. Let t be a join node with children t', t'' such that $\chi(t) = \chi(t') = \chi(t'')$, and let $X \subseteq \widehat{S} \cap \chi(t)$. We prove that the following equation holds:

$$c_{\widehat{S},t}(X) = c_{\widehat{S},t'}(X) + c_{\widehat{S},t''}(X) - \mathrm{lscore}_{\widehat{S},t}(X)$$

Let $\widehat{X}$ be an extension of X at t, so $\mathrm{score}_{\widehat{S},t}(\widehat{X}) = c_{\widehat{S},t}(X)$. The set $\widehat{X'} = \widehat{X} \cap \widehat{S}_{t'}$ satisfies $\widehat{X'} \cap \chi(t') = X$, so $c_{\widehat{S},t'}(X) \leqslant \mathrm{score}_{\widehat{S},t'}(\widehat{X'})$. Symmetrically, for $\widehat{X''} = \widehat{X} \cap \widehat{S}_{t''}$ it holds that $c_{\widehat{S},t''}(X) \leqslant \mathrm{score}_{\widehat{S},t''}(\widehat{X''})$.

There is no element of $V(G)_{t''} \setminus \chi(t)$ that is adjacent to an element of $\widehat{X'} \setminus X$, otherwise $\mathcal{T}$ would not be a valid tree decomposition. Hence $N_G[\widehat{X'}]_t = N_G[\widehat{X'}]_{t'}$, and symmetrically $N_G[\widehat{X''}]_t = N_G[\widehat{X''}]_{t''}$. This entails $\mathrm{score}_{\widehat{S},t}(\widehat{X'}) = \mathrm{score}_{\widehat{S},t'}(\widehat{X'})$ and $\mathrm{score}_{\widehat{S},t}(\widehat{X''}) = \mathrm{score}_{\widehat{S},t''}(\widehat{X''})$.

Since $N_G[\widehat{X}]_t \cap \widehat{S}$ is the union of $N_G[\widehat{X'}]_t \cap \widehat{S}$ and $N_G[\widehat{X''}]_t \cap \widehat{S}$, and these latter two sets have $N_G[X] \cap \chi(t) \cap \widehat{S}$ as their intersection, we can apply the inclusion-exclusion principle to obtain $|N_G[\widehat{X}]_t \cap \widehat{S}| = |N_G[\widehat{X'}]_t \cap \widehat{S}| + |N_G[\widehat{X''}]_t \cap \widehat{S}| - |N_G[X] \cap \chi(t) \cap \widehat{S}|$. In a similar way, we get $|N_G[\widehat{X}]_t \setminus \widehat{S}| = |N_G[\widehat{X'}]_t \setminus \widehat{S}| + |N_G[\widehat{X''}]_t \setminus \widehat{S}| - |(N_G[X] \cap \chi(t)) \setminus \widehat{S})|$. We now can establish $\mathrm{score}_{\widehat{S},t}(\widehat{X}) = \mathrm{score}_{\widehat{S},t}(\widehat{X'}) + \mathrm{score}_{\widehat{S},t}(\widehat{X''}) - \mathrm{lscore}_{\widehat{S},t}(X)$ by putting these equations together. The inequalities we have derived before now allow us to conclude $c_{\widehat{S},t}(X) \geqslant c_{\widehat{S},t'}(X) + c_{\widehat{S},t''}(X) - \mathrm{lscore}_{\widehat{S},t}(X)$.

Now let $\widehat{X'}$ and $\widehat{X''}$ be extensions of X at t' and at t'', respectively. We have that $c_{\widehat{S},t'}(X) = \mathrm{score}_{\widehat{S},t'}(\widehat{X'})$ and $c_{\widehat{S},t''}(X) = \mathrm{score}_{\widehat{S},t''}(\widehat{X''})$. The set $\widehat{X} = \widehat{X'} \cup \widehat{X''}$ is clearly considered in the definition of $c_{\widehat{S},t}(X)$, so $c_{\widehat{S},t}(X) \leqslant \mathrm{score}_{\widehat{S},t}(\widehat{X})$. Following the same reasoning as before, we obtain $\mathrm{score}_{\widehat{S},t}(\widehat{X}) = \mathrm{score}_{\widehat{S},t}(\widehat{X'}) +$

$\mathrm{score}_{\widehat{S},t}(\widehat{X''}) - \mathrm{lscore}_{\widehat{S},t}(X)$. This allows us to conclude $c_{\widehat{S},t}(X) \leqslant c_{\widehat{S},t'}(X) + c_{\widehat{S},t''}(X) - \mathrm{lscore}_{\widehat{S},t}(X)$. Hence $c_{\widehat{S},t}(X) = c_{\widehat{S},t'}(X) + c_{\widehat{S},t''}(X) - \mathrm{lscore}_{\widehat{S},t}(X)$.

Using these recurrence relations, we can traverse the tree decomposition $\mathcal{T}$ in a bottom-up way and compute at each node t of $\mathcal{T}$ the value $c_{\widehat{S},t}(X)$ for each $X \subseteq \widehat{S} \cap \chi(t)$. Hence for each node of $\mathcal{T}$ we compute at most 2^w values, where w is the width of $\mathcal{T}$. By choosing the right data structure for adjacency tests (Cygan et al. 2015, Exercise 7.16), each value can be computed in time $\mathcal{O}(w^3)$. Since $\mathcal{T}$ has $\mathcal{O}(w \cdot |V(G)|)$ many nodes and $\mathcal{T}$ can be computed in fixed-parameter linear time (Bodlaender 1996) (in fact in time $2^{\mathcal{O}(w^3)} \cdot |V(G)|$ as observed by Bojańczyk and Pilipczuk (2017)), we thus get an algorithm with fixed-parameter linear running time for checking whether a given set $\widehat{S}$ is secure.

Theorem 4.41. *Given a graph G, a tree decomposition of G of weight w and a set $\widehat{S} \subseteq V(G)$, we can decide in time $\mathcal{O}(2^w \cdot w^4 \cdot |V(G)|)$ whether $\widehat{S}$ is secure in G.*

Our algorithm can easily be adjusted to find a witness if $\widehat{S}$ is not secure, i.e., to print a subset of $\widehat{S}$ that has more attackers than defenders. After $c_{\widehat{S},t}$ has been computed for each t, this can be done via a final top-down traversal by a standard technique in dynamic programming on tree decompositions (Niedermeier 2006): Alongside each value $c_{\widehat{S},t}(X)$, we store the "origin" of this value and recursively combine the origins of $c_{\widehat{S},r}(\emptyset)$, where r is the root of $\mathcal{T}$.

In our definition of the Secure Set Verification problem, we were only concerned with checking whether a set is secure, but we did not mention the additional constructs that we consider in this paper, like complementary vertex pairs or necessary or forbidden vertices. However, these additions pose no difficulty at all because we can just check the respective conditions in linear time.

4.5.2 A Polynomial-Time Algorithm for Secure Set on Bounded Treewidth

We now present an algorithm for finding secure sets, not just verifying whether a given set is secure. Our algorithm works by dynamic programming on a tree decomposition of the input and extends the algorithm from Section 4.5.1. For graphs of bounded treewidth, the algorithm presented in this section runs in polynomial time. However, in contrast to the algorithm in Section 4.5.1, it is not an FPT algorithm since the degree of the polynomial depends on the treewidth. This is to be expected since the problem of finding secure sets of a certain size is W[1]-hard when parameterized by treewidth, as stated in Corollary 4.38. Our algorithm provides an upper bound for the complexity of this problem, namely membership in the class XP. As Defensive Alliance can be reduced to Secure Set in FPT time while preserving bounded treewidth by Corollary 4.26, our algorithm also shows that Defensive Alliance is solvable in polynomial time on instances of bounded treewidth.

Let G be a graph with a nice tree decomposition $\mathcal{T}$, and let t be a node of $\mathcal{T}$. In Section 4.5.1, we were given one particular secure set candidate $\widehat{S}$ that we wanted to check, so we only computed one value for each $X \subseteq \widehat{S} \cap \chi(t)$, namely the lowest score of any $\widehat{X} \subseteq \widehat{S}_t$ whose intersection with $\chi(t)$ is X. Here, in contrast, we cannot restrict ourselves to only one secure set candidate, and multiple candidates may have the same intersection with $\chi(t)$. We therefore compute multiple objects for each subset of $\chi(t)$, since two subsets of $\mathrm{V}(G)_t$ that have the same intersection with $\chi(t)$ may have to be distinguished due to their subsets having different scores.

Let $S \subseteq \chi(t)$. By F_S we denote the set of functions from 2^S to an integer. Let $c \in F_S$ and let k be an integer. We say that a set $\widehat{S} \subseteq \mathrm{V}(G)_t$ is *(S, t, c, k)-characterized* if $|\widehat{S}| = k$, $\widehat{S} \cap \chi(t) = S$ and, for each $X \subseteq S$, it holds that $c(X) = c_{\widehat{S},t}(X)$, where $c_{\widehat{S},t}$ is the function defined in Section 4.5.1. For each $S \subseteq \chi(t)$, we now define the set

$$C_{S,t} = \{(c, k) \mid \text{there is a } (S, t, c, k)\text{-characterized set}\}.$$

When r is the root of $\mathcal{T}$, there is a secure set of size k in G if and only if there is an element $(c, k) \in C_{\emptyset,r}$ such that $c(\emptyset) \geqslant 0$. To see this, first suppose there is a secure set $\widehat{S}$ of size k in G. Then there is a function $c : \{\emptyset\} \to \mathbb{Z}$ such that $\widehat{S}$ is $(\emptyset, r, c, k)$-characterized, so $(c, k) \in C_{\emptyset,r}$ and $c(\emptyset) = c_{\widehat{S},r}(\emptyset)$, which means that $c(\emptyset)$ is the lowest score of any subset of $\widehat{S}$. Since $\widehat{S}$ is secure in G, this number is nonnegative. For the other direction, let $(c, k) \in C_{\emptyset,r}$ such that $c(\emptyset) \geqslant 0$. Then there is a $(\emptyset, r, c, k)$-characterized set $\widehat{S}$, obviously of size k. Since $c(\emptyset) \geqslant 0$, the lowest score of any subset of $\widehat{S}$ is nonnegative, which proves that $\widehat{S}$ is secure in G.

We now describe how to compute all such values in a bottom-up manner.

Leaf node. If t is a leaf node, its bag is empty and obviously $C_{\emptyset,t} = \{(c, 0)\}$ holds, where c maps $\emptyset$ to 0.

Introduce node. Let t be an introduce node with child t' and let v be the unique element of $\chi(t) \setminus \chi(t')$. For each $S \subseteq \chi(t)$ and each function $c \in F_{S\setminus\{v\}}$, we define a function $c \oplus_{S,t} v \colon 2^S \to \mathbb{Z}$. Its intended purpose is to obtain a version of c that applies to t instead of t'. If $v \in S$, we need to increase scores where v can serve as an additional defender, and otherwise we need to decrease scores where v can serve as an additional attacker. We now make this formal. Let $S \subseteq \chi(t)$, $X \subseteq S$, $X' = X \setminus \{v\}$ and $c \in F_{S\setminus\{v\}}$.

$$(c \oplus_{S,t} v)(X) = \begin{cases} c(X') + 1 & \text{if } v \in N_G[X] \cap S \\ c(X') - 1 & \text{if } v \in N_G[X] \setminus S \\ c(X') & \text{otherwise} \end{cases}$$

For each $S \subseteq \chi(t)$ and each function $c \in F_S$ there is a unique function $c' \in F_{S\setminus\{v\}}$ such that $c = c' \oplus_{S,t} v$, and we denote c' by $\mathrm{origin}_{S,t}(c)$.

The following statements can be proved by arguments similar to those in Section 4.5.1: Let $\widehat{S} \subseteq V(G)_{t'}$, $S = \widehat{S} \cap \chi(t')$ and $(c,k) \in C_{S,t'}$ such that $\widehat{S}$ is (S,t',c,k)-characterized. The set $\widehat{S} \cup \{v\}$ is $(S \cup \{v\}, t, c \oplus_{S\cup\{v\},t} v, k+1)$-characterized and $\widehat{S}$ is $(S,t,c \oplus_{S,t} v, k)$-characterized. Hence $(c \oplus_{S\cup\{v\},t} v, k+1) \in C_{S\cup\{v\},t}$ and $(c \oplus_{S,t} v, k) \in C_{S,t}$. Conversely, let $\widehat{S} \subseteq V(G)_t$, $S = \widehat{S} \cap \chi(t)$ and $(c,k) \in C_{S,t}$ such that $\widehat{S}$ is (S,t,c,k)-characterized, and let $c' = \mathrm{origin}_{S,t}(c)$ and $k' = k - |S \cap \{v\}|$. The set $\widehat{S} \setminus \{v\}$ is $(S \setminus \{v\}, t', c', k')$-characterized. Hence $(c',k') \in C_{S\setminus\{v\},t'}$.

From these observations, the following equation follows for every $S \subseteq \chi(t)$:

$$C_{S,t} = \{(c \oplus_{S,t} v, k + |S \cap \{v\}|) \mid (c,k) \in C_{S\setminus\{v\},t'}\}$$

Forget node. Let t be a forget node with child t' and let v be the unique element of $\chi(t') \setminus \chi(t)$. For each $S \subseteq \chi(t)$ and each function $c \in F_{S\cup\{v\}}$, we define a function $c \ominus^{\in}_{S,t} v$, and for each $S \subseteq \chi(t)$ and each function $c \in F_S$, we define a function $c \ominus^{\notin}_{S,t} v$. Each of these functions maps every subset of S to an integer.

$$(c \ominus^{\in}_{S,t} v)(X) = \min\{c(X),\ c(X \cup \{v\})\}$$

$$(c \ominus^{\notin}_{S,t} v)(X) = c(X)$$

Next we define functions $\mathrm{origin}^{\in}_{S,t}$ and $\mathrm{origin}^{\notin}_{S,t}$ that map each element of F_S to a set of elements of $F_{S\cup\{v\}}$ and F_S, respectively:

$$\mathrm{origin}^{\in}_{S,t}(c) = \{c' \in F_{S\cup\{v\}} \mid c = c' \ominus^{\in}_{S,t} v\}$$

$$\mathrm{origin}^{\notin}_{S,t}(c) = \{c' \in F_S \mid c = c' \ominus^{\notin}_{S,t} v\}$$

The following statements can be proved by arguments similar to those in Section 4.5.1: Let $\widehat{S} \subseteq V(G)_{t'}$, $S = \widehat{S} \cap \chi(t')$ and $(c,k) \in C_{S,t'}$ such that $\widehat{S}$ is (S,t',c,k)-characterized. If $v \in \widehat{S}$, then $\widehat{S}$ is $(S \setminus \{v\}, t, c \ominus^{\in}_{S,t} v, k)$-characterized and $(c \ominus^{\in}_{S,t} v, k) \in C_{S\setminus\{v\},t}$; otherwise $\widehat{S}$ is $(S,t,c \ominus^{\notin}_{S,t} v, k)$-characterized and $(c \ominus^{\notin}_{S,t} v, k) \in C_{S,t}$. Conversely, let $\widehat{S} \subseteq V(G)_t$, $S = \widehat{S} \cap \chi(t)$ and $(c,k) \in C_{S,t}$ such that $\widehat{S}$ is (S,t,c,k)-characterized. If $v \in \widehat{S}$, then there is some $c' \in \mathrm{origin}^{\in}_{S,t}(c)$ such that $\widehat{S}$ is $(S \cup \{v\}, t', c', k)$-characterized and $(c',k) \in C_{S\cup\{v\},t'}$; otherwise there is some $c' \in \mathrm{origin}^{\notin}_{S,t}(c)$ such that $\widehat{S}$ is (S,t',c',k)-characterized and $(c',k) \in C_{S,t'}$.

From these observations, the following equation follows for every $S \subseteq \chi(t)$:

$$C_{S,t} = \{(c \ominus^{\in}_{S,t} v, k) \mid (c,k) \in C_{S\cup\{v\},t'}\} \cup \{(c \ominus^{\notin}_{S,t} v, k) \mid (c,k) \in C_{S,t'}\}$$

Join node. Let t be a join node with children t', t'' such that $\chi(t) = \chi(t') = \chi(t'')$. For each $S \subseteq \chi(t)$, and each $c', c'' \in F_S$, we define a function $c' \otimes_{S,t} c''$, which maps each subset of S to an integer.

$$(c' \otimes_{S,t} c'')(X) = c'(X) + c''(X) - \mathrm{lscore}_{S,t}(X)$$

Next we define a function $\text{origin}_{S,t}$ that maps each element of F_S to a subset of $F_S \times F_S$:

$$\text{origin}_{S,t}(c) = \{(c', c'') \in F_S \times F_S \mid c = c' \otimes_{S,t} c''\}$$

The following statements can be proved by arguments similar to those in Section 4.5.1: Let $\widehat{S'} \subseteq \mathrm{V}(G)_{t'}$, $\widehat{S''} \subseteq \mathrm{V}(G)_{t''}$, $S = \widehat{S'} \cap \widehat{S''}$, $(c', k') \in C_{S,t'}$ and $(c'', k'') \in C_{S,t''}$ such that $\widehat{S'}$ is (S, t', c', k')-characterized and $\widehat{S''}$ is (S, t'', c'', k'')-characterized, and let $c = c' \otimes_{S,t} c''$ and $k = k' + k'' - |S|$. The set $\widehat{S'} \cup \widehat{S''}$ is (S, t, c, k)-characterized and $(c, k) \in C_{S,t}$. Conversely, let $\widehat{S} \subseteq \mathrm{V}(G)_t$, $S = \widehat{S} \cap \chi(t)$ and $(c, k) \in C_{S,t}$ such that $\widehat{S}$ is (S, t, c, k)-characterized. There is some $(c', c'') \in \text{origin}_{S,t}(c)$ as well as integers k', k'' such that $k = k' + k'' - |S|$, the set $\widehat{S} \cap \mathrm{V}(G)_{t'}$ is (S, t', c', k')-characterized and $\widehat{S} \cap \mathrm{V}(G)_{t''}$ is (S, t'', c'', k'')-characterized. Hence $(c', k') \in C_{S,t'}$ and $(c'', k'') \in C_{S,t''}$.

From these observations, the following equation follows for every $S \subseteq \chi(t)$:

$$C_{S,t} = \{(c' \otimes_{S,t} c'', k' + k'' - |S|) \mid (c', k') \in C_{S,t'},\ (c'', k'') \in C_{S,t''}\}$$

We can now traverse the tree decomposition $\mathcal{T}$ in a bottom-up way and at each node t of $\mathcal{T}$ compute the set $C_{S,t}$ for each $S \subseteq \chi(t)$. Let n denote the number of vertices of G and w denote the width of $\mathcal{T}$. Every element of $C_{S,t}$ is a pair (c, k), where c is a function that maps each subset of S to an integer between $-n$ and n, there are at most 2^w subsets of S, and k is an integer between 0 and n. Hence there are at most $(2n+1)^{2^w} \cdot (n+1)$ elements of $C_{S,t}$. Each individual element of $C_{S,t}$ can be computed in time $\mathcal{O}(2^w)$. Finally, there are at most 2^w possible values for S and $\mathcal{O}(wn)$ many nodes in $\mathcal{T}$. We thus get an algorithm that takes as input an integer k together with a graph G whose treewidth we denote by w, and determines in time $f(w) \cdot n^{g(w)}$ whether G admits a secure set of size k, where f and g are functions that only depend on w.

This algorithm for Exact Secure Set obviously also gives us an algorithm for Secure Set by checking all solution sizes from 1 to k. By keeping track of the origins of our computed values during our bottom-up traversal of the tree decomposition, we can even adapt the algorithm without much effort to find solutions if they exist. Finally, we can easily extend it to accommodate complementary and equivalent vertex pairs as well as necessary and forbidden vertices. Hence we get the following XP membership result:

Theorem 4.42. *All of the following problems can be solved in polynomial time if the treewidth of the input is bounded by a constant:* Secure Set, Exact Secure Set, Secure Set$^{\mathrm{F}}$, Exact Secure Set$^{\mathrm{F}}$, Secure Set$^{\mathrm{FN}}$, Exact Secure Set$^{\mathrm{FN}}$, Secure Set$^{\mathrm{FNC}}$, Exact Secure Set$^{\mathrm{FNC}}$, Secure Set$^{\mathrm{FNCE}}$ *and* Exact Secure Set$^{\mathrm{FNCE}}$.

We have seen in Corollary 4.26 that Defensive Alliance can be reduced to Secure Set in FPT time while preserving bounded treewidth. Hence our algorithm can also be used for solving Defensive Alliance, which proves XP membership also for this problem (and the variants we considered). Note, however, that Defensive Alliance is in fact known to be solvable in polynomial time already on instances of bounded clique-width (Kiyomi and Otachi 2017), which is an even stronger result. We leave the question of whether our XP membership result for Secure Set can be extended to instances of bounded clique-width for future work.

4.6 Discussion

One of the central results of this chapter was the proof that the Secure Set problem is Σ_2^P-complete. This means that Secure Set is among the few rather natural problems in graph theory that are complete for the second level of the polynomial hierarchy (like, e.g., Clique Coloring (Marx 2011) or 2-Coloring Extension (Szeider 2005)). Moreover, Σ_2^P-hardness of Secure Set indicates that an efficient reduction to the Sat problem is not possible (unless the polynomial hierarchy collapses).

Beside showing that Secure Set is Σ_2^P-complete, we also proved Σ_2^P-completeness for variants where we are looking for solutions having exactly a given size. Note that a secure set may become insecure by adding or removing elements, so these are non-trivial problem variants. Indeed, exact versions of alliance problems have also been mentioned as interesting variants in Fernau and Raible (2007) because some algorithms that work in the non-exact case stop working in the exact case: A graph has a secure set of size at most k if and only if it has a *connected* secure set of size at most k since every component of a secure set is itself secure. Algorithms that exploit this by looking only for connected solutions hence fail for the exact versions. (In fact, for some of the problem variants that we introduced in this work, this connectedness property does not apply even in the non-exact case.)

Another important contribution of this chapter was the proof that both Defensive Alliance and Secure Set are hard for the class $\mathsf{W}[1]$, which rules out fixed-parameter tractable algorithms under commonly held complexity-theoretic assumptions. This result is rather surprising for two reasons: First, the problems are tractable on trees (Ho and Dutton 2009) and quite often problems that become easy on trees turn out to become easy on graphs of bounded treewidth.[3] Second, this puts Defensive Alliance and Secure Set among the very few "subset problems" that are fixed-parameter tractable w.r.t. solution size but not w.r.t. treewidth. Problems with this kind of behavior are rather rare, as observed by Dom et al. (2008).

[3]To be precise, Ho and Dutton (2009) showed that a slight variant of Secure Set is tractable on trees, since Secure Set on trees is trivial. Our results, however, also imply $\mathsf{W}[1]$-hardness for this particular variant.

```
 x(S) ∨ nx(S) ← s(S).
neighbor(V) ← x(X), e(X,V).
    good(V) ← v(V), x(V).
    good(V) ← neighbor(V), s(V).
     bad(V) ← neighbor(V), v(V), not s(V).
            ← #sum{ 1,V : good(V); -1,V : bad(V) } ⩾ 0.
```

Listing 4.3: An ASP encoding for the co-problem of Secure Set Verification

In Listing 4.1, we presented a guarded ASP encoding for the Secure Set Verification problem and thus showed that the problem is solvable in linear time on instances whose treewidth is bounded by a constant (Theorem 4.39). This ASP program relied on weak constraints to count the number of defenders and the number of attackers. To illustrate the importance of these weak constraints, we compare this to a slightly different encoding, which is presented in Listing 4.3. This encoding has an answer set if and only if there is a witness that the given set S is *not* secure. In contrast to our approach from Listing 4.1, where we compute the minimum difference of "good" and "bad" neighbors and then inspect this value, this encoding produces an answer set if and only if there is a subset of S such that the difference of "good" and "bad" neighbors is negative. Thus Listing 4.3 encodes the co-problem of Secure Set Verification.

When comparing the two ASP encodings for verifying whether a set is secure, the crucial observation is that the program in Listing 4.3 is not guarded due to the last line. The extensions of the `good` and `bad` predicates cannot be determined by the grounder because they depend on the extension of the `x` predicate, which is subject to a non-deterministic guess. Hence the grounder is forced to instantiate the variables in the last rule with each element of S, which results in a single ground rule containing a linear number of atoms (instead of a linear number of ground weak constraints, each containing two atom, as in Listing 4.1). This illustrates that it sometimes pays off to consider alternative encoding techniques in order to obtain a program that is in one of the ASP classes we defined in Chapter 3.

The present chapter extends a conference paper (Bliem and Woltran 2016a), which proved Σ_2^P-completeness of Secure Set and its variants but did not contain any results about the parameterized complexity of these problems. To obtain our W[1]-hardness results for the parameter treewidth, we modified some of the reductions that prove Σ_2^P-hardness so that they preserve bounded treewidth, which allowed us to reuse them for our parameterized hardness proofs. We also added a reduction (that eliminates necessary vertices), which made one of the reductions (from the exact variant of the problem to the non-exact variant) from the conference paper (Bliem and Woltran 2016a) redundant. A paper resulting from these changes is currently under review for a journal and has also been made publicly available (Bliem and Woltran 2017).

The current chapter contains some additional changes that were still not present in the extended paper by Bliem and Woltran (2017). There, we only considered (variants of) the Secure Set problem, whereas in the current chapter we showed W[1]-hardness not only for Secure Set (and variants) when parameterized by treewidth, but also for Defensive Alliance (and variants). Additionally, this chapter showed how Defensive Alliance can be reduced to Secure Set, and we proved that the Secure Set Verification problem is solvable in linear time for instances of bounded treewidth by expressing the problem in guarded ASP and extended monadic second-order logic (Section 4.4.6). Finally, we added Section 4.4.7, which illustrates the use of connection-guarded ASP for showing the known result (Enciso 2009) that Defensive Alliance is solvable in linear time for instances of bounded treewidth and degree.

Related Work

In the literature, several variants of defensive alliances have been studied. The papers that originally introduced defensive alliances (Kristiansen, S. M. Hedetniemi and S. T. Hedetniemi 2002; Kristiansen, S. M. Hedetniemi and S. T. Hedetniemi 2004) also proposed related notions like *offensive* and *powerful* alliances. An offensive alliance is a set S of vertices such that every *neighbor* of an element of S has at least half of its neighbors in S, and a powerful alliance is both a defensive and an offensive alliance. Any of these alliances is called *global* if it is at the same time a dominating set.

For offensive and powerful alliances, it has been shown that deciding if such an alliance of a given (maximum) size exists is fixed-parameter tractable when parameterized by the solution size (Fernau and Raible 2007; Enciso 2009). Kiyomi and Otachi (2017) proved that not only Defensive Alliance but also the corresponding problems for such other alliances can be solved in polynomial time if the clique-width of the instances is bounded by a constant. Furthermore, they provided an FPT algorithm for these problems when the vertex cover number is the parameter.

For *global* defensive alliances, Cami et al. (2006) showed that the respective decision problem is NP-complete. Moreover, in the work by Enciso (2009), we can find an FPT algorithm for finding global defensive alliances when the parameter is the domino treewidth.

Another variant is to consider alliances S where, for each vertex $v \in S$, the difference between the number of neighbors of v in S and the number of other neighbors of v is at most a given integer (Shafique and Dutton 2003). For comprehensive overviews of different kinds of alliances in graphs, we refer to the surveys by Yero and Rodríguez-Velázquez (2013) and Fernau and Rodríguez-Velázquez (2014).

CHAPTER 5

Advanced Dynamic Programming Methodologies

Many problems on the second level of the polynomial hierarchy involve some sort of subset minimization. For instance, a Σ_2^P-complete variant of SAT asks for subset-minimal models of a propositional formula, and in ground ASP we are looking for models I of a program Π that are subset-minimal models of the reduct Π^I (cf. Section 2.4). Many of these problems can be solved via dynamic programming on tree decompositions. However, algorithms for this often suffer from bad performance due to the fact that a lot of computations for subsets of the solution candidates have to be done that are often quite similar to the computations for the solution candidates themselves. Hence such algorithms often perform a great deal of redundant work, and they are also typically more difficult to specify than algorithms that do not involve subset minimization. In this chapter, we present a methodology that alleviates these issues.

There is a close relationship between the polynomial hierarchy and quantifier alternation: The canonical Σ_k^P-complete problem is QSAT$_k$, that is, the problem of deciding whether a quantified Boolean formula of the form $\exists A_1 \forall A_2 \cdots QA_k\ \varphi$ is true, where each A_i is a set of propositional atoms, Q is $\forall$ if k is even and Q is $\exists$ otherwise. Another indication of the connection to quantifier alternation is the fact that we can also define the k-th level of the polynomial hierarchy as consisting of those problems that an alternating Turing machine can solve in polynomial time with k alternations between $\exists$ and $\forall$ states. This definition is equivalent to the classical one in terms of oracle machines.

In this work, we are particularly interested in the second level of the polynomial hierarchy, where we can find many problems from artificial intelligence such as ASP, circumscription, abduction or problems from abstract argumentation (e.g., see the papers by Jakl, Pichler and Woltran (2009), Jakl et al. (2008), Gottlob, Pichler and Wei (2010b) and Dvořák, Pichler and Woltran (2012)). Due to the connection of Σ_2^P with (a single) quantifier alternation, the question asked by decision problems in this class can always be stated in the following pseudo-formal way for illustration:

$$\exists S\Big(\mathrm{P}(S) \land \neg\exists X\big(\mathrm{Q}(X) \land \mathrm{R}(S,X)\big)\Big)$$

Intuitively, $\mathrm{P}(S)$ expresses that S is a *solution candidate*, $\mathrm{Q}(X)$ expresses that X is a *counterexample candidate*, and $\mathrm{R}(S,X)$ expresses that X is a *counterexample*, which witnesses that S is in fact not a solution.[1]

For example, we can state the question for Qsat$_2$ in such a way as follows:

$\mathrm{P}(S)$	...	S is a truth assignment for the variables in A_1.
$\mathrm{Q}(X)$	...	X is a truth assignment for the variables in A_2.
$\mathrm{R}(S,X)$	...	φ is false under the truth assignment given by S and X.

In this chapter, we will focus on search problems, that is, problems that require us to actually produce a solution and not just decide if one exists. This will make presentation easier and the results of this chapter can also be applied to decision problems. For search problems, we are looking for a value for the variable S such that the following subformula of the pseudo-formal expression from before is true.

$$\mathrm{P}(S) \land \neg\exists X\big(\mathrm{Q}(X) \land \mathrm{R}(S,X)\big)$$

We illustrate this using the following search problem, which will play an important role in this chapter.

Subset-Minimal Sat

Input: A Boolean formula φ

Task: Compute a model S of φ such that no proper subset X of S is a model of φ.[a]

[a] We identify a truth assignment with the set of atoms it sets to true.

The decision variant of this problem (where we ask if there is a subset-minimal model that sets a given atom to true) is Σ_2^P-complete.

[1]Actually we could dispense with P and Q in this expression and let R take over their duties, but separating these parts better captures the intuition of many problems.

We represent Subset-Minimal Sat as follows:

$\mathrm{P}(S)$	...	S is a model of φ.
$\mathrm{Q}(X)$	...	X is a model of φ.
$\mathrm{R}(S, X)$	...	X is a proper subset of S.

In this case, the solution candidates are actually the same as the counterexample candidates. Also for many other problems, the counterexample candidates are related to the solution candidates in a certain way. For instance, in ASP a solution candidate S is a model of the program Π and a counterexample candidate X is a model of the reduct Π^S. Even though S and X intuitively refer to different programs, the set of models of Π has a nonempty intersection with the set of models of Π^S in general. The point that we would like to make is that for many problems solution candidates and counterexample candidates are not entirely different things.

Algorithm 5.1: A naive algorithm for Subset-Minimal Sat

```
Input: A Boolean formula $\varphi$ and a variable $z$
Output: A subset-minimal model containing $z$ if there is one, or "no" otherwise
for each set $S$ of variables that occur in $\varphi$ do
    if $S \models \varphi$ then
        counterexample ← false;
        for each set $X$ of variables that occur in $\varphi$ do
            if $X \models \varphi$ and $X \subset S$ then counterexample ← true;
        if counterexample = false then return $S$;
return "no";
```

We now illustrate a consequence of this relationship between solution candidates and counterexample candidates. A very naive way of finding solutions for a Subset-Minimal Sat instance is outlined in Algorithm 5.1. Note that in general the inner loop considers a set X of variables many times during the execution and solves the same model checking problem $X \models \varphi$ over and over again. Moreover, note that the algorithm also checks $X \models \varphi$ if it has already checked this for a *solution candidate* equal to X. Thus, due to the similarity of solution candidates and counterexample candidates, the algorithm performs many redundant tasks.

Usually dynamic programming algorithms use a naive brute-force approach similar to this in every tree decomposition node to produce all partial solution candidates (by which we mean the information needed at the current node to eventually compute all solutions). Moreover, for Σ_2^P-hard problems like Subset-Minimal Sat, such algorithms then typically proceed by storing all partial counterexample candidates for each partial solution candidate from the previous step. Generally the number of objects computed by such an algorithm at every tree decomposition node is thus doubly exponential in the treewidth. Theoretically this is not an issue for fixed-parameter tractable algorithms because the treewidth is considered a constant. In practice, however, the redundant computations lead to an enormous overhead in terms of running time and memory (Bliem et al. 2016a).

Beside these performance issues, also the design and implementation of such dynamic programming algorithms is often quite tedious. Especially for problems where counterexample candidates are very similar to solution candidates (or even exactly the same), care must be taken to avoid redundancies in the code, which make the programs difficult to maintain.

In this chapter, we show how we can automatically generate tree-decomposition-based dynamic programming algorithms from simpler principles. For this, we first introduce a formal model of dynamic programming computations, which allows us to abstract from concrete algorithms and makes our results generally applicable to many problems instead of just to a particular problem. We show how our model captures typical dynamic programming computations for problems on the second level of the polynomial hierarchy, and we discuss how we can ensure fixed-parameter tractability. This formal framework is the basis for the main contribution of this chapter: We give a formal definition of a transformation that turns computations for a simpler problem into computations for a derived, more complex problem. Moreover, we identify conditions under which this procedure is sound, and we give a formal correctness proof.

To facilitate presentation, we restrict ourselves to subset-minimization problems such as Subset-Minimal Sat. We thus assume that the solution candidates are exactly the same as the counterexample candidates, and we assume that checking whether X is a counterexample to S being a solution amounts to checking whether $X \subset S$. After presenting our main contribution, we will discuss how we can generalize the results by allowing for more complex relationships between solution candidates and counterexample candidates.

Subset-minimization problems such as Subset-Minimal Sat can be seen as derived from a *base problem*, by which we mean a problem whose solutions are exactly the solution *candidates* of the derived problem. In the case of Subset-Minimal Sat, the base problem is Sat: The solution candidates for Subset-Minimal Sat are exactly the solutions of Sat, and for determining if such a candidate is a solution, we just need to check for minimality.

The core idea of our transformation that generates an algorithm for the derived problem from an algorithm for the base problem is the following: We first run the algorithm for the base problem to compute all partial solution candidates. Then we discard all partial solution candidates that do not lead to solutions. Finally, we automatically obtain the relevant partial counterexample candidates from the remaining partial solution candidates.

This chapter is organized as follows. In Section 5.1, we present a formal framework for tree-decomposition-based dynamic programming algorithms. Next, in Section 5.2, we define conditions on dynamic programming computations under which the solutions produced by the computation are exactly the subset-minimal solutions of the base problem. We then show in Section 5.3 how we can compress our data structures, which is important for obtaining fixed-parameter tractability. In Section 5.4, we present the conditions that a dynamic programming algorithm for a base problem must fulfill in order to be eligible for our automatic transformation to an algorithm for the respective derived problem. The heart of this chapter is Section 5.5, where we present our transformation for such algorithms and prove its correctness as well as running time guarantees. Finally we discuss extensions of our results and relationships to other approaches in Section 5.6.

5.1 A Formal Account of Dynamic Programming on Tree Decompositions

In this section we formally describe dynamic programming on tree decompositions for subset optimization problems. First, we define our data structure, called *computation*, which is a tree of *tables* resulting from the bottom-up traversal of the tree decomposition. We then define the *extensions* of table rows, which are used to obtain (partial) solution candidates for the problem at hand. Additionally, we define their relation to counterexamples. Finally, we state how solutions are obtained and define properties that have to be fulfilled by the tables in a computation in order to yield correct results.

Definition 5.1. A *computation* is a rooted ordered tree whose nodes are called *tables*. Each table R is a set of *rows* and each row $r \in R$ possesses

- some problem-specific *data* $\mathrm{D}(r)$,
- a non-empty set of *extension pointer tuples* (EPTs) $\mathrm{P}(r)$ such that each tuple is of arity k, where k is the number of children of R, and for each $(p_1, \dots, p_k) \in \mathrm{P}(r)$ it holds that each p_i is a row of the i-th child of R,
- a *subtable* $\mathrm{S}(r)$, which is a set of *subrows*, where each subrow $s \in \mathrm{S}(r)$ possesses
  - some problem-specific data $\mathrm{D}(s)$,
  - a non-empty set of EPTs $\mathrm{P}(s)$ such that for each $(p_1, \dots, p_k) \in \mathrm{P}(s)$ there is some $(q_1, \dots, q_k) \in \mathrm{P}(r)$ with $p_i \in \mathrm{S}(q_i)$ for $1 \leqslant i \leqslant k$,
  - an inclusion status flag $\mathrm{inc}(s) \in \{\mathrm{eq}, \subset\}$.

For rows or subrows a, b we write $a \approx b$, $a \leqslant b$ and $a < b$ to denote $\mathrm{D}(a) = \mathrm{D}(b)$, $\mathrm{D}(a) \subseteq \mathrm{D}(b)$ and $\mathrm{D}(a) \subset \mathrm{D}(b)$, respectively. For sets of rows or subrows R, S we write $\mathrm{D}(R)$ to denote $\bigcup_{r \in R} \mathrm{D}(r)$, and we write $R \approx S$, $R \leqslant S$ and $R < S$ to denote $\mathrm{D}(R) = \mathrm{D}(S)$, $\mathrm{D}(R) \subseteq \mathrm{D}(S)$ and $\mathrm{D}(R) \subset \mathrm{D}(S)$, respectively.

The reason that each row possesses a subtable is that we consider subset optimization problems, and we assume that algorithms for such problems use subtables to store potential counterexamples to a solution candidate being subset-minimal. The intuition of a subrow s for a row r is that s represents solution candidates that are subsets of the candidates represented by r. If one of these subset relations is proper, we indicate this by $\mathrm{inc}(s) = \subset$.

Example 5.2. We return to Example 2.3 and recall the propositional formula φ_{Ex} defined as $(u \vee v) \wedge (\neg v \vee w \vee x) \wedge (\neg w) \wedge (\neg x \vee z) \wedge (\neg x \vee y \vee \neg z)$, its primal graph G_{Ex} and a tree decomposition $\mathcal{T}_{Ex}$ as depicted in Figure 2.2. Now we consider Subset-Minimal Sat (i.e., finding models that are subset-minimal w.r.t. the set of atoms that get assigned "true"). Figure 5.1 illustrates the computation performed by a dynamic programming algorithm for Subset-Minimal Sat at nodes n_3 and n_4 of $\mathcal{T}_{Ex}$. At n_3, the table R is computed as before (as depicted in Figure 2.3). For any $r \in R$, each subrow $s \in \mathrm{S}(r)$ represents a partial assignment that is a subset of the one in r (i.e., $\mathrm{D}(s) \subseteq \mathrm{D}(r)$) and that also satisfies all clauses covered by the bag (in this case the clauses $(\neg x \vee z)$ and $(\neg x \vee y \vee \neg z)$). Moreover, $\mathrm{inc}(s)$ is set appropriately. Now consider n_4, where y and z are removed. As in the dynamic programming algorithm for Sat, we simply project away data related to the removed vertices. Observe that for instance 4^1 now contains redundant subrows 4^1_2, 4^1_3, 4^1_4 and 4^1_5 in the sense that they contain the same data and inclusion flag (and only different EPTs). We overcome this problem in Section 5.3, where tables are *compressed* in order to remove such redundancies. △

The EPTs of a table row r are used for recursively combining the problem-specific data $\mathrm{D}(r)$ with data from "compatible" rows that are in descendant tables. The fact that each set of EPTs is required to be non-empty entails that for each (sub)row r at a leaf table it holds that $\mathrm{P}(r) = \{()\}$. We disallow rows with an empty set of EPTs because in the end we are only interested in rows that can be extended to complete solutions, consisting of one row per table. For this we introduce the notion of an *extension* of a table row.

Definition 5.3. Let $\mathcal{C}$ be a computation and R be a table in $\mathcal{C}$ with k children. We inductively define the *extensions* of a row $r \in R$ as $\mathrm{E}(r) = \big\{\{r\} \cup A \mid A \in \bigcup_{(p_1,\dots,p_k) \in \mathrm{P}(r)} \{X_1 \cup \dots \cup X_k \mid X_i \in \mathrm{E}(p_i) \text{ for all } 1 \leqslant i \leqslant k\}\big\}$.

Note that any extension $X \in \mathrm{E}(r)$ contains r and exactly one row from each table that is a descendant of R. If r is a row of a leaf table, $\mathrm{E}(r) = \big\{\{r\}\big\}$ because $\mathrm{P}(r) = \{()\}$.

n_3

R			$\mathrm{S}(r)$			
r	D	P	s	D	P	inc
3^1	x,y,z	$()$	3^1_1	x,y,z	$()$	eq
			3^1_2	y,z	$()$	$\subset$
			3^1_3	y	$()$	$\subset$
			3^1_4	z	$()$	$\subset$
			3^1_5		$()$	$\subset$
3^2	y,z	$()$	3^2_1	y,z	$()$	eq
			3^2_2	y	$()$	$\subset$
			3^2_3	z	$()$	$\subset$
			3^2_4		$()$	$\subset$
3^3	y	$()$	3^3_1	y	$()$	eq
			3^3_2		$()$	$\subset$
3^4	z	$()$	3^4_1	z	$()$	eq
			3^4_2		$()$	$\subset$
3^5		$()$	3^5_1		$()$	eq

n_4

R			$\mathrm{S}(r)$			
r	D	P	s	D	P	inc
4^1	x	(3^1)	4^1_1	x	(3^1_1)	eq
			4^1_2		(3^1_2)	$\subset$
			4^1_3		(3^1_3)	$\subset$
			4^1_4		(3^1_4)	$\subset$
			4^1_5		(3^1_5)	$\subset$
4^2		(3^2)	4^2_1		(3^2_1)	eq
			4^2_2		(3^2_2)	$\subset$
			4^2_3		(3^2_3)	$\subset$
			4^2_4		(3^2_4)	$\subset$
4^3		(3^3)	4^3_1		(3^3_1)	eq
			4^3_2		(3^3_2)	$\subset$
4^4		(3^4)	4^4_1		(3^4_1)	eq
			4^4_2		(3^4_2)	$\subset$
4^5		(3^5)	4^5_1		(3^5_1)	eq

Figure 5.1: (Partial) dynamic programming computation for Subset-Minimal Sat without compression

While the extensions from the root table of a computation represent complete solution candidates, the purpose of subtables is to represent possible counterexamples that would cause a solution candidate to be invalidated. More precisely, for each extension X that can be obtained by extending a root table row r, we check if we can find an extension Y of an element $s \in \mathrm{S}(r)$ with $\mathrm{inc}(s) = \subset$ such that every element of Y is listed as a subrow of a row in X (i.e., we check if for every $y \in Y$ there is some $x \in X$ with $y \in \mathrm{S}(x)$). If this is so, then Y witnesses that X represents no solution because Y then represents a solution candidate that is a proper subset. For this reason, we need to introduce the notion of extensions (like Y) relative to another extension (like X).

Definition 5.4. Let $\mathcal{C}$ be a computation, R be a table in $\mathcal{C}$ with k children, $r \in R$ be a row and $s \in \mathrm{S}(r)$ be a subrow of r. We first define, for any $X \in \mathrm{E}(r)$, a restriction of $\mathrm{P}(s)$ to EPTs where each element is a subrow of a row in X, as $\mathrm{P}_X(s) = \{(p_1, \dots, p_k) \in \mathrm{P}(s) \mid r_i \in X,\ p_i \in \mathrm{S}(r_i) \text{ for all } 1 \leqslant i \leqslant k\}$. Now we define the set of *extensions* of s *relative to* some extension $X \in \mathrm{E}(r)$ as $\mathrm{E}_X(s) = \{\{s\} \cup A \mid A \in \bigcup_{(p_1,\dots,p_k) \in \mathrm{P}_X(s)} \{Y_1 \cup \cdots \cup Y_k \mid Y_i \in \mathrm{E}_X(p_i) \text{ for all } 1 \leqslant i \leqslant k\}\}$.

We can now formalize that the solutions of a computation are the extensions of those rows that do not have a subrow indicating a counterexample.

Definition 5.5. Let R be the root table in a computation $\mathcal{C}$. We define the set of *solutions* of $\mathcal{C}$ as $\mathrm{sol}(\mathcal{C}) = \{\, \mathrm{D}(X) \mid r \in R,\ X \in \mathrm{E}(r),\ \nexists s \in \mathrm{S}(r) : \mathrm{inc}(s) = \subset \}$

Example 5.6. If n_4 in Figure 5.1 were the root of the tree decomposition, only 4^5 would yield a solution (namely $\{\}$, i.e., the interpretation where x, y, and z are all set to false) since all other rows contain a subrow where $\mathrm{inc}(s) = \subset$. This is indeed the only subset-minimal model of the formula consisting of the clauses encountered until n_4, i.e., $(\neg x \vee z) \wedge (\neg x \vee y \vee \neg z)$. △

For this example, the computation of our Subset-Minimal Sat algorithm yields solutions as intended. Other algorithms, however, may not be so well-behaved. We are still missing a criterion to distinguish algorithms that indeed produce computations whose solutions are minimal from algorithms that use subtables in an unintended way. The following section presents such a criterion.

5.2 Normal Computations

In this section, we formalize requirements on subrows and their inclusion status to ensure that subrows correspond to subsets of their parent row, that each potential counterexample is represented by a subrow and that $\mathrm{inc}(\cdot)$ is used as intended. We call a computation *normal* if it is well-behaved in this sense.

Definition 5.7. A table R is *normal* if the following properties hold:

1. For each $r \in R$, $s \in \mathrm{S}(r)$, $X \in \mathrm{E}(r)$ and $Y \in \mathrm{E}_X(s)$, it holds that $Y \leqslant X$, and $Y < X$ holds if and only if $\mathrm{inc}(s) = \subset$.
2. For each $r \in R$, $s \in \mathrm{S}(r)$ and $Y \in \mathrm{E}(s)$ there is some $r' \in R$ and $X' \in \mathrm{E}(r')$ such that $s \approx r'$ and $Y \approx X'$.
3. For each $q, r \in R$, $Z \in \mathrm{E}(q)$ and $X \in \mathrm{E}(r)$, if $Z \leqslant X$ holds, then there is some $s \in \mathrm{S}(r)$ and $Y \in \mathrm{E}_X(s)$ with $s \approx q$ and $Y \approx Z$.

A computation is *normal* if all its tables are normal.

This definition ensures that it suffices to examine the root table of a normal computation in order to decide a subset-minimization problem correctly, provided that the rows represent all solution candidates.

Example 5.8. The table of node n_4 in Figure 5.1 is normal. We only check the conditions for some of the (sub)rows. Let $r = 4^2$ and $s = 4^2_2$. Observe that $\mathrm{E}(r) = \{\{r, 3^2\}\}$ and $\mathrm{E}_X(s) = \{\{s, 3^2_2\}\}$. Let X and Y be the unique elements of $\mathrm{E}(r)$ and $\mathrm{E}_X(s)$, respectively. Now $\mathrm{D}(X) = \{y, z\}$ and $\mathrm{D}(Y) = \{y\}$, so $Y \leqslant X$ holds as required by condition 1. In fact, even $Y < X$, and $\mathrm{inc}(s) = \subset$ as required. As for condition 2, observe that Y is also in $\mathrm{E}(s)$, so we must find a row r' and extension X' of r' that "mirror" s and Y, respectively. Indeed, $r' = 4^3$ and $X' = \{r', 3^3\}$ (for which $X' \in \mathrm{E}(r')$ holds) satisfy $s \approx r'$ and $Y \approx X'$. As for condition 3, let $q = 4^3$ and $Z = \{q, 3^3\}$. It holds that $Z \in \mathrm{E}(r')$ and $Z \leqslant X$, so we must find a subrow s of r and extension Y of s (relative to X) that "mirror" q and Z, respectively. Indeed, $s = 4^2_2$ and $Y = \{s, 3^2_2\}$ (for which $Y \in \mathrm{E}_X(s)$ holds) satisfy $s \approx q$ and $Y \approx Z$. △

We will later show how a non-minimizing computation (i.e., one with empty subtables) satisfying certain properties can be transformed into a normal computation. In this transformation, we must avoid redundancies lest we destroy fixed-parameter tractability. For this, we first introduce how tables can be compressed without losing solution candidates.

5.3 Table Compression

In this section, we show how the tables can be compressed such that their sizes are bounded by the width of the tree decomposition and do not become exponentially large in the input size. The idea is to compress tables by merging equivalent (sub)rows. For this, we first define an equivalence relation on rows as well as one on subrows.

5.3.1 Compressing Subtables

Definition 5.9. Let R be a table and $r \in R$. We define an equivalence relation $\equiv_r$ over the subrows of r such that $s_1 \equiv_r s_2$ if $s_1 \approx s_2$ and $\mathrm{inc}(s_1) = \mathrm{inc}(s_2)$.

We use this notion of equivalence between subrows to compress subtables by merging equivalent subrows.

Definition 5.10. Let R be a table and $r \in R$. We define a subtable $\mathrm{S}^*(r)$ called the *compressed subtable* of r that contains exactly one subrow for each $\equiv_r$-equivalence class. For any $s \in \mathrm{S}(r)$, let $[s]$ denote the $\equiv_r$-equivalence class of s and let s' denote the subrow in $\mathrm{S}^*(r)$ corresponding to $[s]$. We define s' by $s' \approx s$, $\mathrm{inc}(s') = \mathrm{inc}(s)$ and $\mathrm{P}(s') = \bigcup_{t \in [s]} \mathrm{P}(t)$.

Example 5.11. Let r be the row 3^1 in Figure 5.1. Since all subrows of r have different data, each $\equiv_r$-equivalence class is a singleton, so we cannot merge any equivalent subrows of r (i.e., $\mathrm{S}^*(r)$ contains as many subrows as $\mathrm{S}(r)$). Now let $r = 4^1$. There are two $\equiv_r$-equivalence classes: one containing just 4^1_1, the other containing 4^1_2, 4^1_3, 4^1_4 and 4^1_5. Hence $S^*(r)$ contains two subrows, each corresponding to a $\equiv_r$-equivalence class and having the same data and inclusion status as all the members of this class. Thus 4^1_2, 4^1_3, 4^1_4 and 4^1_5 are merged into a single subrow $s \in S^*(r)$ such that $D(s) = \emptyset$, $\mathrm{inc}(s) = \subset$ and $P(s) = \{(3^1_2), (3^1_3), (3^1_4), (3^1_5)\}$. $\triangle$

5.3.2 Compressing Tables

Once subtables have been compressed, we can compress the table by merging equivalent rows. For this, we first need a notion of equivalence between rows that takes compressed subtables into account.

Definition 5.12. We define an equivalence relation $\equiv_R$ over the rows of a table R such that $r_1 \equiv_R r_2$ if $r_1 \approx r_2$ and there is a bijection $f : S^*(r_1) \to S^*(r_2)$ such that for any $s \in S^*(r_1)$ it holds that $s \approx f(s)$ and $\mathrm{inc}(s) = \mathrm{inc}(f(s))$.

When rows are equivalent, their compressed subtables only differ in the EPTs. We now define how such compressed subtables can be merged.

Definition 5.13. Let R be a table, $r \in R$, and let $[r]$ denote the $\equiv_R$-equivalence class of r. For any $r' \in [r]$, let $f_{r'} : S^*(r) \to S^*(r')$ be the bijection such that for any $s \in S^*(r)$ it holds that $s \approx f_{r'}(s)$ and $\mathrm{inc}(s) = \mathrm{inc}(f_{r'}(s))$. (The existence of $f_{r'}$ is guaranteed by Definition 5.12.) We define a subtable $\mathrm{mst}([r])$ (for "merged subtable") that contains exactly one subrow for each element of $S^*(r)$. For any $s \in S^*(r)$, let s' denote the subrow in $\mathrm{mst}([r])$ corresponding to s. We define s' by $s' \approx s$, $\mathrm{inc}(s') = \mathrm{inc}(s)$ and $P(s') = \bigcup_{r' \in [r]} P(f_{r'}(s))$.

We use these equivalence relations to compress tables in such a way that all equivalent (sub)rows (according to the respective equivalence relation) are merged.

Definition 5.14. Let R be a table. We now define a table $\mathrm{compr}(R)$ that contains exactly one row for each $\equiv_R$-equivalence class. For any $r \in R$, let $[r]$ denote the $\equiv_R$-equivalence class of r and let r' be the row in $\mathrm{compr}(R)$ corresponding to $[r]$. We define r' by $r' \approx r$, $P(r') = \bigcup_{q \in [r]} P(q)$ and $S(r') = \mathrm{mst}([r])$. For any computation $\mathcal{C}$, we write $\mathrm{compr}(\mathcal{C})$ to denote the computation isomorphic to $\mathcal{C}$ where each table R in $\mathcal{C}$ corresponds to $\mathrm{compr}(R)$ in $\mathrm{compr}(\mathcal{C})$.

Example 5.15. Figure 5.2 illustrates the complete dynamic programming computation for our running example, including compression. Root node n_6 contains two rows, 6^1 and 6^2. The row 6^2 represents the subset-minimal models of our running example, since it is associated with a single subrow 6^2_1, where $\mathrm{inc}(6^2_1) = \mathrm{eq}$. Opposed to that, models represented by 6^1 are not subset-minimal, due to $\mathrm{inc}(6^1_2) = \subset$. We obtain the solutions by following the EPTs and combining P of the respective rows: We have

$\mathrm{P}(6^2) \cup \mathrm{P}(5^4) \cup \mathrm{P}(2^3) \cup \mathrm{P}(1^2) \cup \mathrm{P}(4^3) \cup \mathrm{P}(3^5) = \{u\}$ and $\mathrm{P}(6^2) \cup \mathrm{P}(5^5) \cup \mathrm{P}(2^4) \cup \mathrm{P}(1^3) \cup \mathrm{P}(4^1) \cup \mathrm{P}(3^1) = \{x, v, y, z\}$. Thus the solutions are $\{u\}$ (i.e., u set to true, and v, x, y, z set to false) as well as $\{v, x, y, z\}$ (with u set to false). Indeed, these are exactly the subset-minimal models of φ_{Ex}.

Regarding compression, consider the table stored at node n_4 in Figure 5.2 in comparison to the non-compressed table for n_4 in Figure 5.1. For instance, due to Definition 5.13, 4^1 now has subrow 4^1_2 that contains a set of EPTs, i.e., $\mathrm{P}(4^1_2) = \{(3^1_2), (3^1_3), (3^1_4), (3^1_5)\}$. Additionally, due to Definition 5.14 we can merge complete rows. As an example, the single row 4^2 is obtained by compressing rows 4^2, 4^3 and 4^4 of Figure 5.1, since the data of these rows as well as the data and the inclusion flags of all subrows coincide, i.e., they are contained in the same $\equiv_R$-equivalence class. △

We spend the rest of this section proving some properties of table compression. In particular, we show that there is a one-to-one correspondence between extensions of a table and extensions of its compressed version such that the respective data are the same. Moreover, we show that compressing a table according to the definitions above preserves normality.

5.3.3 Correspondence Between Extensions

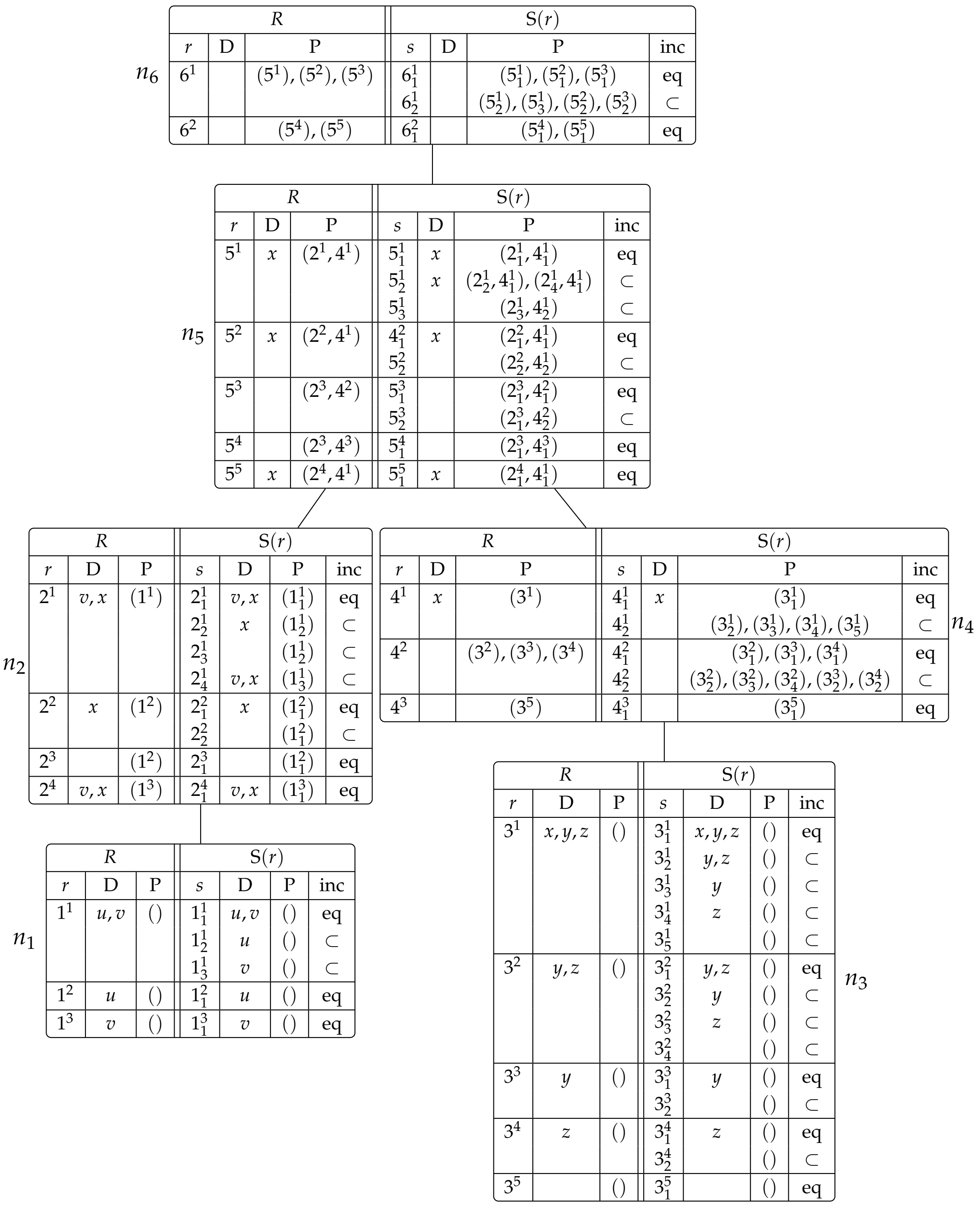
We now show that the extensions from rows in a table R are in a one-to-one correspondence to the extensions of rows from $\mathrm{compr}(R)$ such that the data of corresponding extensions is the same.

Lemma 5.16. *Let R be a table. For any $r \in R$ and $X \in \mathrm{E}(r)$ there are $r' \in \mathrm{compr}(R)$ and $X' \in \mathrm{E}(r')$ such that $r' \approx r$ and $X' \approx X$. Also, for any $r' \in \mathrm{compr}(R)$ and $X' \in \mathrm{E}(r')$ there are $r \in R$ and $X \in \mathrm{E}(r)$ such that $r \approx r'$ and $X \approx X'$.*

Proof. Let R be a table and $R' = \mathrm{compr}(R)$. For the first statement, let $r \in R$ and $X \in \mathrm{E}(r)$. There are $(r_1, \dots, r_k) \in \mathrm{P}(r)$ and $X_i \in \mathrm{E}(r_i)$ such that $X = \{r\} \cup X_1 \cup \dots \cup X_k$. By definition of $\mathrm{compr}(\cdot)$, there is $r' \in R'$ with $r' \approx r$ and $(r_1, \dots, r_k) \in \mathrm{P}(r')$. Now $X' = \{r'\} \cup X_1 \cup \dots \cup X_k$ is in $\mathrm{E}(r')$ and $X' \approx X$. The second statement is proved symmetrically. □

We can also obtain a similar lemma for subrows instead of rows.

Lemma 5.17. *Let R be a table. For any $r \in R$, $X \in \mathrm{E}(R)$, $s \in \mathrm{S}(r)$ and $Y \in \mathrm{E}_X(s)$ there are $r' \in \mathrm{compr}(R)$, $X' \in \mathrm{E}(r')$, $s' \in \mathrm{S}(r')$ and $Y' \in \mathrm{E}_{X'}(s')$ such that $r' \approx r$, $X' \approx X$, $s' \approx s$ and $Y' \approx Y$. Also, for any $r' \in \mathrm{compr}(R)$, $X' \in \mathrm{E}(r')$, $s \in \mathrm{S}(r')$ and $Y' \in \mathrm{E}_{X'}(s')$ there are $r \in R$, $X \in \mathrm{E}(r)$, $s \in \mathrm{S}(r)$ and $Y \in \mathrm{E}_X(s)$ such that $r \approx r'$, $X \approx X'$, $s \approx s'$ and $Y \approx Y'$.*



n_6

R			$S(r)$			
r	D	P	s	D	P	inc
6^1		$(5^1), (5^2), (5^3)$	6^1_1		$(5^1_1), (5^2_1), (5^3_1)$	eq
			6^1_2		$(5^1_2), (5^1_3), (5^2_2), (5^3_2)$	$\subset$
6^2		$(5^4), (5^5)$	6^2_1		$(5^4_1), (5^5_1)$	eq

n_5

R			$S(r)$			
r	D	P	s	D	P	inc
5^1	x	$(2^1, 4^1)$	5^1_1	x	$(2^1_1, 4^1_1)$	eq
			5^1_2	x	$(2^1_2, 4^1_1), (2^1_4, 4^1_1)$	$\subset$
			5^1_3		$(2^1_3, 4^1_2)$	$\subset$
5^2	x	$(2^2, 4^1)$	4^2_1	x	$(2^2_1, 4^1_1)$	eq
			5^2_2		$(2^2_2, 4^1_2)$	$\subset$
5^3		$(2^3, 4^2)$	5^3_1		$(2^3_1, 4^2_1)$	eq
			5^3_2		$(2^3_1, 4^2_2)$	$\subset$
5^4		$(2^3, 4^3)$	5^4_1		$(2^3_1, 4^3_1)$	eq
5^5	x	$(2^4, 4^1)$	5^5_1	x	$(2^4_1, 4^1_1)$	eq

n_2

R			$S(r)$			
r	D	P	s	D	P	inc
2^1	v, x	(1^1)	2^1_1	v, x	(1^1_1)	eq
			2^1_2	x	(1^1_2)	$\subset$
			2^1_3		(1^1_2)	$\subset$
			2^1_4	v, x	(1^1_3)	$\subset$
2^2	x	(1^2)	2^2_1	x	(1^2_1)	eq
			2^2_2		(1^2_1)	$\subset$
2^3		(1^2)	2^3_1		(1^2_1)	eq
2^4	v, x	(1^3)	2^4_1	v, x	(1^3_1)	eq

n_4

R			$S(r)$			
r	D	P	s	D	P	inc
4^1	x	(3^1)	4^1_1	x	(3^1_1)	eq
			4^1_2		$(3^1_2), (3^1_3), (3^1_4), (3^1_5)$	$\subset$
4^2		$(3^2), (3^3), (3^4)$	4^2_1		$(3^2_1), (3^3_1), (3^4_1)$	eq
			4^2_2		$(3^2_2), (3^2_3), (3^2_4), (3^3_2), (3^4_2)$	$\subset$
4^3		(3^5)	4^3_1		(3^5_1)	eq

n_1

R			$S(r)$			
r	D	P	s	D	P	inc
1^1	u, v	$()$	1^1_1	u, v	$()$	eq
			1^1_2	u	$()$	$\subset$
			1^1_3	v	$()$	$\subset$
1^2	u	$()$	1^2_1	u	$()$	eq
1^3	v	$()$	1^3_1	v	$()$	eq

n_3

R			$S(r)$			
r	D	P	s	D	P	inc
3^1	x, y, z	$()$	3^1_1	x, y, z	$()$	eq
			3^1_2	y, z	$()$	$\subset$
			3^1_3	y	$()$	$\subset$
			3^1_4	z	$()$	$\subset$
			3^1_5		$()$	$\subset$
3^2	y, z	$()$	3^2_1	y, z	$()$	eq
			3^2_2	y	$()$	$\subset$
			3^2_3	z	$()$	$\subset$
			3^2_4		$()$	$\subset$
3^3	y	$()$	3^3_1	y	$()$	eq
			3^3_2		$()$	$\subset$
3^4	z	$()$	3^4_1	z	$()$	eq
			3^4_2		$()$	$\subset$
3^5		$()$	3^5_1		$()$	eq

Figure 5.2: Dynamic programming computation for Subset-Minimal Sat

Proof. Let R be a table and $R' = \text{compr}(R)$. For the first statement, let $r \in R$, $X \in \mathrm{E}(r)$, $s \in \mathrm{S}(r)$ and $Y \in \mathrm{E}_X(s)$. Moreover, let $(s_1, \ldots, s_k) \in \mathrm{P}(s)$ and $Y_i \in \mathrm{E}_{X_i}(s_i)$ be such that $Y = \{s\} \cup Y_1 \cup \cdots \cup Y_k$. By Lemma 5.16, there are $r' \in R'$ and $X' \in \mathrm{E}(r')$ such that $r' \approx r$ and $X' \approx X$. By definition of $\text{compr}(\cdot)$ and $\text{mst}(\cdot)$, there is $s' \in \mathrm{S}(r')$ with $s' \approx s$ and $(s_1, \ldots, s_k) \in \mathrm{P}(s')$. Now $Y' = \{s'\} \cup Y_1 \cup \cdots \cup Y_k$ is in $\mathrm{E}_{X'}(s')$ and $Y' \approx Y$. The second statement is proved symmetrically. □

5.3.4 Preservation of Normality

Next we show that compressing a table according to the definitions above preserves normality.

Lemma 5.18. *If a table R is normal, then so is* $\text{compr}(R)$.

Proof. Let R be a normal table and $R' = \text{compr}(R)$. We denote the child tables of R (which are also the child tables of R') by $R_1, \ldots, R_k$. We prove conditions 1–3 of normality of R' separately.

1. Let $r' \in R'$, $s' \in \mathrm{S}(r')$, $X' \in \mathrm{E}(r')$ and $Y' \in \mathrm{E}_{X'}(s')$. By Lemma 5.17, we can find $r \in R$, $s \in \mathrm{S}(r)$, $X \in \mathrm{E}(r)$ and $Y \in \mathrm{E}_X(s)$ such that $r \approx r'$, $s \approx s'$, $X \approx X'$ and $Y \approx Y'$. As R is normal, $Y \leqslant X$ holds, which proves that $Y' \leqslant X'$. By definition of $\text{compr}(\cdot)$, it holds that $\text{inc}(s) = \subset$ if and only if $\text{inc}(s') = \subset$. Furthermore, due to normality of R, $Y < X$ holds if and only if $\text{inc}(s) = \subset$. This proves that $Y' < X'$ holds if and only if $\text{inc}(s') = \subset$. We have thus proved condition 1.

2. Let $r' \in R'$, $s' \in \mathrm{S}(r')$ and $Y' \in \mathrm{E}(s')$. As a consequence of Lemma 5.17, we can find $r \in R$, $s \in \mathrm{S}(r)$ and $Y \in \mathrm{E}(s)$ such that $r \approx r'$, $s \approx s'$ and $Y \approx Y'$. As R is normal, there is a row $t \in R$ and an extension $T \in \mathrm{E}(t)$ such that $t \approx s$ and $T \approx Y$. By Lemma 5.16, now there is a row $t' \in R'$ and an extension $T' \in \mathrm{E}(t')$ such that $t' \approx t$ and $T' \approx T$, which proves condition 2.

3. Let $q', r' \in R'$, $Z' \in \mathrm{E}(q')$ and $X' \in \mathrm{E}(r')$ such that $Z' \leqslant X'$. By Lemma 5.16, we can find $q, r \in R$, $Z \in \mathrm{E}(q)$ and $X \in \mathrm{E}(r)$ such that $Z \approx Z'$ and $X \approx X'$, which implies $Z \leqslant X$. As R is normal, now there is a subrow $s \in \mathrm{S}(r)$ and an extension $Y \in \mathrm{E}_X(s)$ such that $s \approx q$ and $Y \approx Z$. By Lemma 5.17, this implies existence of a subrow $s' \in \mathrm{S}(r')$ and an extension $Y' \in \mathrm{E}_{X'}(s')$ such that $s' \approx s$ and $Y' \approx Y$. This proves condition 3. □

We use table compression in our transformation from algorithms for the base problem to algorithms for the derived problem. Before we introduce this transformation, we need to define to which algorithms for the base problem it can be applied.

5.4 Augmentable Computations

In this section, we introduce conditions that make a computation without minimization eligible for transformation into a "minimizing" one. We call such a computation *augmentable*.

First we define some "forbidden" properties that may prevent a computation from being eligible for our transformation. In particular, when an element d occurs in a table, then rows from this table that do not include d must not be extended by rows that include d. Intuitively, once some row contains d, all rows in the same table also have to "decide" whether they want to include d, and this decision must not be revoked later on.

We now formalize the idea that for each table R and each element $d \in \mathrm{D}(R)$ we regard all rows in R whose data does not include d as having irrevocably decided against including d. We will use this to enforce that each extension X that includes such a row must satisfy $d \notin \mathrm{D}(X)$.

Definition 5.19. Let X be an extension of a row in a table of a computation $\mathcal{C}$. We define $\mathrm{D}^-(X)$ as the union of all sets $\mathrm{D}(R) \setminus \mathrm{D}(r)$ such that r is a row in X and R is the table containing r in $\mathcal{C}$.

We use this to define the notion of an *inconsistent* extension. The intuition is that the data of such an extension contains an item d, but at the same time some row in this extension "decided against" including d.[2]

Definition 5.20. Let r be a row in a table of a computation and $X \in \mathrm{E}(r)$ be an extension of r. We say that X is *inconsistent* if the intersection of $\mathrm{D}(X)$ and $\mathrm{D}^-(X)$ is nonempty.

We now define the notion of an *augmentable* computation, i.e., a computation that can be used in our transformation.

Definition 5.21. We call a computation $\mathcal{C}$ *augmentable* if the following conditions hold for all tables R in $\mathcal{C}$:

1. For all rows $r \in R$ it holds that $\mathrm{S}(r) = \emptyset$.
2. For all $r, r' \in R$ with $r \neq r'$ it holds that $\mathrm{D}(r) \neq \mathrm{D}(r')$.
3. No row in R has an inconsistent extension.

[2]Note that in a previously published version of our results (Bliem et al. 2016a) different concepts were used instead of inconsistent extensions. The proof of Theorem 3.24 in that work unfortunately contained an error, which we now corrected by introducing the notion of an inconsistent extension.

These requirements are satisfied by many reasonable tree-decomposition-based dynamic programming algorithms. In particular, condition 2 is usually satisfied by reasonable FPT algorithms because they avoid redundancies in order to stay fixed-parameter tractable. To understand the motivation of condition 3, note that usually dynamic programming algorithms do not put arbitrary data into the rows. Rather, the data in a row is typically restricted to information about bag elements of the respective tree decomposition node. Often every data element can be associated with a unique bag element that this element "speaks about". By the definition of tree decompositions, if an element is contained in two bags, then it must be contained in all bags between those. Similarly, if a data element is contained in two tables, then it is often also contained in all tables in between.

Indeed many dynamic programming algorithms follow the scheme of assigning some status to a bag element once it is introduced, keeping that status the same as long as that element is in the current bag and forgetting the status when the bag element is removed. Moreover, it is often the case that join nodes are processed in such a way that only rows that agree on the status of common bag elements are combined. Algorithms that work in this simple way can easily be shown to only produce computations without inconsistent extensions. Thus many dynamic programming algorithms only produce augmentable computations.

Example 5.22. The computation depicted in Figure 2.3 is augmentable. In the figure, we omitted the subtables since all of them are empty, which is what condition 1 requires. Condition 2 is also clearly satisfied. As for condition 3, consider the three extensions of 5^1 for example:

- For the extension $X = \{5^1, 2^1, 4^1, 1^1, 3^1\}$, we have $\mathrm{D}(X) = \{u, v, x, y, z\}$ and $\mathrm{D}^-(X) = \emptyset$.
- For $X = \{5^1, 2^1, 4^1, 1^3, 3^1\}$, we have $\mathrm{D}(X) = \{v, x, y, z\}$ and $\mathrm{D}^-(X) = \{u\}$.
- For $X = \{5^1, 2^2, 4^1, 1^2, 3^1\}$, we have $\mathrm{D}(X) = \{u, x, y, z\}$ and $\mathrm{D}^-(X) = \{v\}$.

None of these extensions violates condition 3. △

5.5 From Augmentable to Normal Computations

Now we describe how augmentable computations can automatically be transformed into normal computations that take minimization into account. For any table R in an augmentable computation, this allows us to compute a new table $\mathrm{aug}(R)$ if for each child table R_i the table $\mathrm{aug}(R_i)$ has already been computed and compressed to $\mathrm{compr}(\mathrm{aug}(R_i))$.

5.5.1 Augmenting Tables

We now define the function $\mathrm{aug}(\cdot)$, which is our desired transformation from "non-minimizing" to "minimizing" computations.

Definition 5.23. We inductively define a function $\mathrm{aug}(\cdot)$ that maps each table R from an augmentable computation to a new table. Let the child tables of R be called $R_1, \dots, R_k$. For any $1 \leqslant i \leqslant k$ and $r \in R_i$, we write $\mathrm{res}(r)$ to denote $\{q \in \mathrm{compr}(\mathrm{aug}(R_i)) \mid q \approx r\}$. We define $\mathrm{aug}(R)$ as the smallest table that satisfies the following conditions:

1. For each $r \in R$, $(r_1, \dots, r_k) \in \mathrm{P}(r)$ and $(c_1, \dots, c_k) \in \mathrm{res}(r_1) \times \cdots \times \mathrm{res}(r_k)$, there is a row $q \in \mathrm{aug}(R)$ with $q \approx r$ and $\mathrm{P}(q) = \{(c_1, \dots, c_k)\}$.
2. For all $q, q' \in \mathrm{aug}(R)$ such that $q' \leqslant q$, we denote the unique element of $\mathrm{P}(q)$ as $(q_1, \dots, q_k)$ and the unique element of $\mathrm{P}(q')$ as $(q'_1, \dots, q'_k)$, and we require that the following holds: If for each $1 \leqslant i \leqslant k$ there is some $s_i \in \mathrm{S}(q_i)$ with $s_i \approx q'_i$, then there is a subrow $s \in \mathrm{S}(q)$ with $s \approx q'$ and $\mathrm{P}(s) = \{(s_1, \dots, s_k)\}$. Moreover, $\mathrm{inc}(s) = \subset$ if $q' < q$ or $\mathrm{inc}(s_i) = \subset$ for some s_i, otherwise $\mathrm{inc}(s) = \mathrm{eq}$.

For any augmentable computation $\mathcal{C}$, we write $\mathrm{aug}(\mathcal{C})$ to denote the computation isomorphic to $\mathcal{C}$ where each table R in $\mathcal{C}$ corresponds to $\mathrm{aug}(R)$.

Example 5.24. In this example, for integers i and j, we use T_i to denote the table of node n_i in Figure 2.3, we use i^j to denote the respective row in Figure 2.3, and we use $\bar{i}^j$ to denote the row that is called i^j in Figure 5.2. Suppose that we have already computed $\mathrm{compr}(\mathrm{aug}(T_2))$ and $\mathrm{compr}(\mathrm{aug}(T_4))$, which are depicted in Figure 5.2. Now we want to compute $\mathrm{aug}(T_5)$.

The idea behind condition 1 of Definition 5.23 is that each row in T_5 gives rise to (possibly several) rows in $\mathrm{aug}(T_5)$ in the following way: For each row in T_5, we look at each EPT, and for each element r_i of this EPT we find out which rows $\mathrm{res}(r_i)$ resulted from r_i in the respective compressed augmented table. For each combination of one resulting row per EPT element, we add a row to $\mathrm{aug}(T_5)$. For instance, let $r = 5^1$ (i.e., the row in T_5 with $\mathrm{D}(r) = \{x\}$), and let $(r_1, r_2) = (2^1, 4^1)$, which is an element of $\mathrm{P}(r)$. It holds that $\mathrm{res}(r_1) = \{\bar{2}^1, \bar{2}^4\}$ (i.e., the two rows in $\mathrm{compr}(\mathrm{aug}(T_2))$ whose data is $\{v, x\}$), whereas $\mathrm{res}(r_2) = \{\bar{4}^1\}$ (i.e., the row in $\mathrm{compr}(\mathrm{aug}(T_4))$ whose data is $\{x\}$). Let $c_1 = \bar{2}^1$, and let $c_2 = \bar{4}^1$. Since $(c_1, c_2) \in \mathrm{res}(r_1) \times \mathrm{res}(r_2)$, condition 1 of Definition 5.23 forces us to put a row into $\mathrm{aug}(T_5)$ having the same data as r and extending (c_1, c_2). In this way, we proceed with all other rows in T_5, EPTs of these rows, etc., until we have added all rows to $\mathrm{aug}(T_5)$ that condition 1 requires. This results in rows of $\mathrm{aug}(T_5)$ like those we see in Figure 5.2, except that their subtables are still empty. (Actually, Figure 5.2 shows $\mathrm{compr}(\mathrm{aug}(T_5))$ and not $\mathrm{aug}(T_5)$, but the compression does not merge any rows of $\mathrm{aug}(T_5)$.)

Now let q be the row in $\operatorname{aug}(T_5)$ with $\mathrm{D}(q) = \{x\}$ and $\mathrm{P}(q) = \{(\bar{2}^1, \bar{4}^1)\}$, and let q' be the row in $\operatorname{aug}(T_5)$ with $\mathrm{D}(q') = \{x\}$ and $\mathrm{P}(q') = \{(\bar{2}^2, \bar{4}^1)\}$. It holds that $q' \leqslant q$. Condition 2 now checks if q' gives us reason to add an entry to the subtable of q. Indeed this is the case: Let (q_1, q_2) denote the unique element of $\mathrm{P}(q)$, and let (q'_1, q'_2) denote the unique element of $\mathrm{P}(q')$. There is a subrow s_1 in $\mathrm{S}(q_1)$ that has the same data as q'_1 (i.e., $\{x\}$). Likewise, there is a subrow s_2 in $\mathrm{S}(q_2)$ that has the same data as q'_2 (i.e., $\{x\}$). Condition 2 forces us to add a subrow s into $\mathrm{S}(q)$ that has the same data as q' and extends (s_1, s_2). Since $\operatorname{inc}(s_1) = \subset$ and $\operatorname{inc}(s_2) = \mathrm{eq}$, we have to set $\operatorname{inc}(s) = \subset$. We continue like this for all other pairs of rows from $\operatorname{aug}(T_5)$ until we have added all subrows that condition 2 requires. △

We spend the remainder of this section proving some properties of $\operatorname{aug}(\cdot)$. In particular, we still need to show that the transformation leads to correct algorithms and that it does not destroy fixed-parameter tractability of base algorithms.

5.5.2 Correspondence Between Extensions

Tables in augmentable computations never have two different rows r, r' with $r \approx r'$. Moreover, we defined $\operatorname{aug}(R)$ in such a way that for all $r \in R$ there is some $q \in \operatorname{aug}(R)$ with $r \approx q$. In the compression $\operatorname{compr}(\operatorname{aug}(R))$, we only merge rows and subrows having the same data. So with each row and subrow in $\operatorname{aug}(R)$ or $\operatorname{compr}(\operatorname{aug}(R))$ we can associate a unique originating row in R. In fact, the extensions from a table R in an augmentable computation are in a one-to-one correspondence to the extensions of rows from $\operatorname{aug}(R)$. To show these claims we need some additional formal machinery.

Definition 5.25. Let R be a table in an augmentable computation, let $Q = \operatorname{aug}(R)$ and $Q' = \operatorname{compr}(Q)$. We define a function $\operatorname{orig}_R(\cdot)$ that maps each row or subrow in Q or Q' to a row in R. For $q \in Q \cup Q'$ we define $\operatorname{orig}_R(q)$ to be the unique $r \in R$ with $r \approx q$. For $s \in \mathrm{S}(q)$ we define $\operatorname{orig}_R(s)$ to be the unique $r \in R$ with $r \approx s$.

Let R be a table from an augmentable computation. We want to establish that for each extension in $\operatorname{aug}(R)$, we can also obtain one in R having the same data and vice versa. The following lemma will be helpful for this. It formalizes that for each (sub)row q in $\operatorname{aug}(R)$ with $(q_1, \dots, q_k) \in \mathrm{P}(q)$, the originating row $\operatorname{orig}_R(q)$ in R has a corresponding EPT $(r_1, \dots, r_k)$ such that each q_i is resulting from r_i.

Lemma 5.26. *Let R be a table from an augmentable computation and let the child tables of R be called $R_1, \dots, R_k$. For each $q \in \operatorname{aug}(R)$ and $(q_1, \dots, q_k) \in \mathrm{P}(q)$, we have $\big(\operatorname{orig}_{R_1}(q_1), \dots, \operatorname{orig}_{R_k}(q_k)\big) \in \mathrm{P}(\operatorname{orig}_R(q))$. Furthermore, for every $s \in \mathrm{S}(q)$ and for each $(s_1, \dots, s_k) \in \mathrm{P}(s)$ it holds that $\big(\operatorname{orig}_{R_1}(s_1), \dots, \operatorname{orig}_{R_k}(s_k)\big) \in \mathrm{P}(\operatorname{orig}_R(s))$.*

Proof. Let R be a table in some augmentable computation such that $R_1, \dots, R_k$ denote the child tables of R, and let $Q = \operatorname{aug}(R)$ with child tables $Q_i = \operatorname{compr}(\operatorname{aug}(R_i))$.

Moreover, let $q \in Q$ and $(q_1, \dots, q_k) \in \mathrm{P}(q)$. By construction of $\mathrm{aug}(R)$, there are $r \in R$ and $(r_1, \dots, r_k) \in \mathrm{P}(r)$ such that $r \approx q$ and $q_i \in \mathrm{res}(r_i)$ for each $1 \leqslant i \leqslant k$. From this we get $q_i \approx r_i$, so $\mathrm{orig}_{R_i}(q_i) = r_i$ holds. Furthermore, $\mathrm{orig}_R(q) = r$ holds since $q \approx r$. Because of $(r_1, \dots, r_k) \in \mathrm{P}(r)$, this entails $\big(\mathrm{orig}_{R_1}(q_1), \dots, \mathrm{orig}_{R_k}(q_k)\big) \in \mathrm{P}(\mathrm{orig}_R(q))$.

As for the second statement, let $s \in \mathrm{S}(q)$ and $(s_1, \dots, s_k) \in \mathrm{P}(s)$. By construction of $\mathrm{aug}(R)$, there are $q' \in Q$ and $(q'_1, \dots, q'_k) \in \mathrm{P}(q')$ such that $q' \approx s$ and $q'_i \approx s_i$ for each $1 \leqslant i \leqslant k$. We know that this entails $\big(\mathrm{orig}_{R_1}(q'_1), \dots, \mathrm{orig}_{R_k}(q'_k)\big) \in \mathrm{P}(\mathrm{orig}_R(q'))$ by the first statement of this lemma. As we have seen that $q'_i \approx s_i$ and $q' \approx s$, it must hold that $\mathrm{orig}_R(s) = \mathrm{orig}_R(q')$ and $\mathrm{orig}_{R_i}(s_i) = \mathrm{orig}_{R_i}(q'_i)$, which proves the second statement. □

Now we can show that the extensions from a table R in an augmentable computation are in a one-to-one correspondence to the extensions of rows from $\mathrm{aug}(R)$.

Lemma 5.27. *Let R be a table from an augmentable computation and $Q = \mathrm{aug}(R)$. For any $r \in R$ and $Z \in \mathrm{E}(r)$ there are $q \in Q$ and $X \in \mathrm{E}(q)$ such that $r \approx q$ and $Z \approx X$. Also, for any $q \in Q$ and $X \in \mathrm{E}(q)$ there are $r \in R$ and $Z \in \mathrm{E}(r)$ such that $q \approx r$ and $X \approx Z$.*

Proof. Let R be a table in an augmentable computation such that $R_1, \dots, R_k$ denote the child tables of R, and let $Q = \mathrm{aug}(R)$ with child tables $Q_i = \mathrm{compr}(\mathrm{aug}(R_i))$. We prove the lemma by induction. First suppose R and Q are leaf tables. It can be easily verified using the definition of Q that the rows in Q and R are in a one-to-one correspondence. Formally, $|Q| = |R|$ holds and for each $r \in R$ there is a $q \in Q$ with $q \approx r$. As the EPTs in R and Q always consist of just the empty tuple, the data of any extension of a row in R or Q coincides with the data of that row.

Now suppose R and Q are arbitrary tables and both statements from the lemma hold for all R_i. We first attend to the first statement. Let $r \in R$ and $Z \in \mathrm{E}(r)$. There are $(r_1, \dots, r_k) \in \mathrm{P}(r)$ and $Z_i \in \mathrm{E}(r_i)$ such that $Z = \{r\} \cup Z_1 \cup \dots \cup Z_k$. By the induction hypothesis and Lemma 5.16, there are $q_i \in Q_i$ and $X_i \in \mathrm{E}(q_i)$ such that $r_i \approx q_i$ and $X_i \approx Z_i$. As $r_i \approx q_i$, it holds that $q_i \in \mathrm{res}(r_i)$. Hence condition 1 from the construction of Q ensures that there is a row $q \in Q$ with $q \approx r$ and $\mathrm{P}(q) = \{(q_1, \dots, q_k)\}$. Now clearly $X = \{q\} \cup X_1 \cup \dots \cup X_k$ is contained in $\mathrm{E}(q)$, and $X \approx Z$.

As for the second statement, let $q \in Q$ and $X \in \mathrm{E}(q)$. There are $(q_1, \dots, q_k) \in \mathrm{P}(q)$ and $X_i \in \mathrm{E}(q_i)$ such that $X = \{q\} \cup X_1 \cup \dots \cup X_k$. By Lemma 5.16 and the induction hypothesis, there are $r_i \in R_i$ and $Z_i \in \mathrm{E}(r_i)$ such that $q_i \approx r_i$ and $X_i \approx Z_i$. Now $\mathrm{orig}_{R_i}(q_i) = r_i$ holds because of $q_i \approx r_i$. Let $r = \mathrm{orig}_R(q)$, so obviously $q \approx r$. By Lemma 5.26, $(q_1, \dots, q_k) \in \mathrm{P}(q)$ entails $\big(\mathrm{orig}_{R_1}(q_1), \dots, \mathrm{orig}_{R_k}(q_k)\big) \in \mathrm{P}(\mathrm{orig}_R(q))$, i.e., $(r_1, \dots, r_k) \in \mathrm{P}(r)$. Now clearly $Z = \{r\} \cup Z_1 \cup \dots \cup Z_k$ is contained in $\mathrm{E}(r)$, and $X \approx Z$. □

We can state a similar lemma for subrows instead of rows.

Lemma 5.28. *Let R be a table from an augmentable computation and $Q = \mathrm{aug}(R)$. For any $q \in Q$, $s \in \mathrm{S}(q)$ and $Y \in \mathrm{E}(s)$ there are $r \in R$ and $Z \in \mathrm{E}(r)$ such that $s \approx r$ and $Y \approx Z$.*

Proof. Let R be a table in an augmentable computation such that $R_1, \dots, R_k$ denote the child tables of R, and let $Q = \mathrm{aug}(R)$ with child tables $Q_i = \mathrm{compr}(\mathrm{aug}(R_i))$. We prove the lemma by induction. First suppose R and Q are leaf tables. By definition of Q, for each subrow s in Q there is some $q' \in Q$ with $s \approx q'$. Obviously, $q' \approx \mathrm{orig}_R(q')$ and $\mathrm{orig}_R(s) = \mathrm{orig}_R(q')$, so $s \approx \mathrm{orig}_R(s)$, which proves the base case.

Now suppose R and Q are arbitrary tables and the statement holds for all R_i. Let $q \in Q$, $s \in \mathrm{S}(q)$, $Y \in \mathrm{E}(s)$ and $r = \mathrm{orig}_R(s)$. Now there are $(s_1, \dots, s_k) \in \mathrm{P}(s)$ and $Y_i \in \mathrm{E}(s_i)$ such that $Y = \{s\} \cup Y_1 \cup \dots \cup Y_k$. By Lemma 5.17 and the induction hypothesis, there are $r_i \in R_i$ and $Z_i \in \mathrm{E}(r_i)$ such that $s_i \approx r_i$ and $Y_i \approx Z_i$. Then $\mathrm{orig}_{R_i}(s_i) = r_i$. By Lemma 5.26, $(s_1, \dots, s_k) \in \mathrm{P}(s)$ entails $\big(\mathrm{orig}_{R_1}(s_1), \dots, \mathrm{orig}_{R_k}(s_k)\big) \in \mathrm{P}(\mathrm{orig}_R(s))$, so $(r_1, \dots, r_k) \in \mathrm{P}(r)$. Now clearly $Z = \{r\} \cup Z_1 \cup \dots \cup Z_k$ is contained in $\mathrm{E}(r)$, and $Y \approx Z$. □

5.5.3 Correctness

The following lemma is central for showing that $\mathrm{aug}(\cdot)$ works as intended.

Lemma 5.29. *Let R be a table from an augmentable computation. Then the table $\mathrm{aug}(R)$ is normal.*

Proof. Let R be a table in an augmentable computation such that $R_1, \dots, R_k$ denote the child tables of R and let $Q = \mathrm{aug}(R)$. We use induction. If Q is a leaf table, then rows and extensions coincide and the construction of Q clearly ensures that Q is normal. So suppose that Q has child tables $Q_i = \mathrm{compr}(\mathrm{aug}(R_i))$ and that all $\mathrm{aug}(R_i)$ are normal. By Lemma 5.18, the latter implies that all Q_i are normal too. We prove that Q is normal by considering the three conditions of normality separately. For this, let $q \in Q$, $s \in \mathrm{S}(q)$, $\mathrm{P}(q) = \{(q_1, \dots, q_k)\}$, $\mathrm{P}(s) = \{(s_1, \dots, s_k)\}$, $X_i \in \mathrm{E}(q_i)$, $Y_i \in \mathrm{E}_{X_i}(s_i)$, $X = \{q\} \cup X_1 \cup \dots \cup X_k$ and $Y = \{s\} \cup Y_1 \cup \dots \cup Y_k$.

As for normality condition 1, the construction of Q ensures $s \leqslant q$ and normality of Q_i ensures $Y_i \leqslant X_i$, so $Y \leqslant X$. To show that $\mathrm{inc}(s)$ has the correct value, first suppose $Y < X$ and $\mathrm{inc}(s) = \mathrm{eq}$. The latter would entail $Y_i = X_i$, so $s < q$, but this would yield the contradiction $\mathrm{inc}(s) = \subset$. So suppose $X \approx Y$ and $\mathrm{inc}(s) = \subset$. If $s < q$, then $\mathrm{D}(q) \setminus \mathrm{D}(s)$ would contain an element d that is also present in $\mathrm{D}(Y)$ because $d \in \mathrm{D}(X)$ and $X \approx Y$. Since $d \in \mathrm{D}(Y)$ but at the same time $d \notin \mathrm{D}(s)$ and $d \in \mathrm{D}(Q)$, the origin of s in R would have an inconsistent extension by Lemma 5.28, contradicting that the computation is augmentable. So $s \approx q$; thus the reason for $\mathrm{inc}(s) = \subset$ must be that for some j there is a $d \in \mathrm{D}(X_j) \setminus \mathrm{D}(Y_j)$. Due to $X \approx Y$, we get $d \in \mathrm{D}(s)$ or $d \in \mathrm{D}(Y_h)$ for some $h \neq j$. In both cases, the origin of s in R has an inconsistent extension.

For condition 2, let $Z_i \in \mathrm{E}(s_i)$ and $Z = \{s\} \cup Z_1 \cup \cdots \cup Z_k$. As $s \in \mathrm{S}(q)$, there are $p \in Q$ and $(p_1, \dots, p_k) \in \mathrm{P}(p)$ with $p \approx s$ and $p_i \approx s_i$. This entails existence of $r \in R$ and $(r_1, \dots, r_k) \in \mathrm{E}(r)$ with $r \approx p$ and $r_i \approx p_i$. By hypothesis and Lemma 5.16, there are $q'_i \in Q_i$ and $X'_i \in \mathrm{E}(q'_i)$ with $q'_i \approx s_i$ and $X'_i \approx Z_i$. Each q'_i originates from the unique row in R having the same data as q'_i. Hence $q'_i \in \mathrm{res}(r_i)$ holds and there are $q' \in Q$ and $X' \in \mathrm{E}(q')$ with $q' \approx r \approx s$ and $X' \approx Z$.

For condition 3, let $q' \in Q$, $\mathrm{P}(q') = \{(q'_1, \dots, q'_k)\}$, $X' \in \mathrm{E}(q')$ and $X'_i \in \mathrm{E}(q'_i)$ for all $1 \leqslant i \leqslant k$. Suppose $X' \leqslant X$. First we show that $X'_i \leqslant X_i$ for all i. Suppose, for the sake of contradiction, that for some j there is a $d \in \mathrm{D}(X'_j) \setminus \mathrm{D}(X_j)$. Since $X' \leqslant X$, it must then hold that $d \in \mathrm{D}(q)$ or $d \in \mathrm{D}(X_h)$ for some $h \neq j$. In both cases, the origin of q in R has an inconsistent extension. So $X'_i \leqslant X_i$ for each i. By hypothesis there are $t_i \in \mathrm{S}(q_i)$ and $T_i \in \mathrm{E}_{X_i}(t_i)$ with $t_i \approx q'_i$ and $T_i \approx X'_i$. So there is a $t \in \mathrm{S}(q)$ with $t \approx q'$ and $\mathrm{P}(t) = \{(t_1, \dots, t_k)\}$. Now $T = \{t\} \cup T_i \cup \cdots \cup T_k$ is in $\mathrm{E}_X(t)$ and $T \approx X'$. □

We can now state the main theorem of this chapter, which says that exactly the subset-minimal solutions of an augmentable computation are the solutions of the augmented computation.

Theorem 5.30. *Let $\mathcal{C}$ be an augmentable computation. Then* $\mathrm{sol}(\mathrm{aug}(\mathcal{C})) = \{S \in \mathrm{sol}(\mathcal{C}) \mid \nexists S' \in \mathrm{sol}(\mathcal{C}) : S' \subset S\}$.

Proof. Let R be the root table of an augmentable computation $\mathcal{C}$ and R' be the root table of $\mathcal{C}' = \mathrm{aug}(\mathcal{C})$. By Lemma 5.29, $\mathcal{C}'$ is normal. We prove both directions of the equality separately by showing that each side of the equality is included in the other.

For the first direction, let $S \in \mathrm{sol}(\mathcal{C}')$. There are $r' \in R'$ and $X' \in \mathrm{E}(r')$ such that $\mathrm{D}(X') = S$ and for all $s' \in \mathrm{S}(r')$ it holds that $\mathrm{inc}(s') = \mathrm{eq}$. By Lemma 5.27, there are $r \in R$ and $X \in \mathrm{E}(r)$ such that $X \approx X'$. As $\mathcal{C}$ is augmentable, $\mathrm{S}(r)$ is empty, so $S \in \mathrm{sol}(\mathcal{C})$ by Definition 5.5. We must now show that there is no solution in $\mathcal{C}$ that is a proper subset of S. For the sake of contradiction, suppose there is some $T \in \mathrm{sol}(\mathcal{C})$ with $T \subset S$. Now there are $q \in R$ and $Z \in \mathrm{E}(q)$ such that $\mathrm{D}(Z) = T$, hence $Z < X'$. By Lemma 5.27, there are $q' \in R'$ and $Z' \in \mathrm{E}(q')$ such that $Z' \approx Z$. As R' is normal, due to $Z' < X'$, there is some $s' \in \mathrm{S}(r')$ such that $\mathrm{inc}(s') = \subset$. This contradicts $\mathrm{inc}(s') = \mathrm{eq}$, which we have seen earlier.

For the other direction, let $S \in \mathrm{sol}(\mathcal{C})$ be such that there is no $S' \in \mathrm{sol}(\mathcal{C})$ with $S' \subset S$. There are $r \in R$ and $X \in \mathrm{E}(r)$ such that $\mathrm{D}(X) = S$. By Lemma 5.27, there are $r' \in R'$ and $X' \in \mathrm{E}(r')$ such that $X' \approx X$, and there are no $q' \in R'$ and $Z' \in \mathrm{E}(q')$ with $Z' < X$. Hence, as R' is normal, there cannot be a $s' \in \mathrm{S}(r')$ with $\mathrm{inc}(s') = \subset$. This proves that $S \in \mathrm{sol}(\mathcal{C}')$. □

5.5.4 Fixed-Parameter Tractability

Finally, we sketch that $\mathrm{aug}(\cdot)$ does not destroy fixed-parameter tractability.

Theorem 5.31. *Let $\mathcal{A}$ be an algorithm that takes as input an instance of size n and treewidth w along with a tree decomposition $\mathcal{T}$ of width w. Suppose $\mathcal{A}$ produces an augmentable computation $\mathcal{C}$ isomorphic to $\mathcal{T}$ in time $f(w) \cdot n^{\mathcal{O}(1)}$, where f is a function depending only on w. Then $\mathrm{aug}(\mathcal{C})$ can be computed in time $g(w) \cdot n^{\mathcal{O}(1)}$, where g again depends only on w.*

Proof. We assume w.l.o.g. that each node in $\mathcal{T}$ has at most 2 children, as any tree decomposition can be transformed to this form in linear time without increasing the width (Kloks 1994). As $\mathcal{A}$ runs in FPT time, i.e., in $f(w) \cdot n^{\mathcal{O}(1)}$ for a function f, no table in $\mathcal{C}$ can be bigger than $f(w) \cdot n^c$ for a constant c. To recursively compute $Q = \mathrm{aug}(R)$ for some table R in $\mathcal{C}$ with child tables $R_1, \dots, R_k$ ($k \leqslant 2$), suppose we have already constructed each $Q_i = \mathrm{aug}(R_i)$ in FPT time. Then $|Q_i| = f_i(w) \cdot n^{c_i}$ for some f_i and c_i. We can clearly compute $Q_i' = \mathrm{compr}(Q_i)$ in time polynomial in $|Q_i|$. Then $|Q_i'| = f_i'(w) \cdot n^{c_i'}$ for some f_i' and c_i'. Definition 5.23 suggests a straightforward way of computing Q in time polynomial in $|R|$ and $\sum_{1 \leqslant i \leqslant k} |Q_i'|$. So we can compute Q in FPT time. As $\mathcal{T}$ has size $\mathcal{O}(n)$, we can compute $\mathrm{aug}(\mathcal{C})$ in FPT time. $\square$

5.6 Discussion

In this chapter, we presented a way of constructing tree-decomposition-based dynamic programming algorithms from simpler principles. In particular, we showed how an algorithm that produces solution satisfying a subset-minimality condition can automatically be obtained from a "non-minimizing" algorithm. This can greatly simplify the specification of dynamic programming algorithms, and it leads to algorithms whose performance is much better in terms of both running time and memory compared to naive dynamic programming implementations (Bliem et al. 2016a).

In presenting our transformation from an algorithm for a base problem into an algorithm for a derived problem, we assumed that the solutions of the derived problem are exactly the subset-minimal solutions of the base problem. Although some problems from the second level of the polynomial hierarchy are that simple, many problems are related to a base problem in a more complicated way. Our approach can, however, be easily extended in several directions. For instance, it is straightforward to generalize our transformation so that it supports problems where only a certain part of the data (instead of all data) is subject to minimization. Several problems can be handled using this generalization, for instance propositional circumscription (McCarthy 1980).

Our approach is based on the idea that counterexample candidates are also solution candidates. For many problems this is not the case. For example, when we want to compute the answer sets of a disjunctive ASP program, then solution candidates are models of the program while counterexample candidates are models of the respective *reduct* of the program and not the program itself. A quick and dirty way of dealing with this is to put all such counterexample candidates also among the solution candidates and mark them with a special flag. In the end, we suppress all solutions extending a counterexample candidate.

Finally, our approach clearly also works for finding maximal subsets instead of minimal ones by just reversing the directions of the set inclusion checks.

The discussed extensions have been implemented in the D-FLAT^2 system (Bliem et al. 2016a). Moreover, our extended approach is the theoretical basis for the *DynASP2* system (Fichte et al. 2017), which is a solver for ground disjunctive ASP programs. The performance of both of these systems is greatly enhanced by the optimizations analyzed in this chapter.

Related Work

There seems to be only little research in the direction of improving tree-decomposition-based dynamic programming algorithms by using specialized modules for certain tasks. One example is the *D-FLAT* system (Abseher et al. 2014), which allows for rapid prototyping of such algorithms by offering users to specify them in ASP. This framework provides a few facilities that take care of cost minimization or of an automatic handling of join nodes. As the name suggests, the D-FLAT^2 system mentioned above is an extension of D-FLAT.

Another related approach is called *Autograph* (Courcelle and Durand 2016). This tool makes it possible to combine pre-defined components called fly-automata and thus piece together an automaton that solves the problem at hand.

The motivation for our approach is slightly different and is inspired more by metaprogramming techniques for ASP. For exploiting the full expressive power of ASP, we often need to employ a sophisticated encoding technique called *saturation* (see, e.g., the paper by Leone et al. (2006)) in order to solve co-NP subproblems. As such encodings are quite error-prone and difficult to understand, several approaches try to relieve ASP users from this task (see, e.g., the papers by Eiter and Polleres (2006), Gebser, Kaminski and Schaub (2011) and Brewka et al. (2015)). For instance, we can compute minimal models of a propositional formula by simply expressing the Sat problem in ASP together with a special *minimize statement*, which is recognized by systems like *metasp* (Gebser, Kaminski and Schaub 2011). In this way, we obtain a program computing subset-minimal models. Our approach presented in this chapter clearly resembles this research.

Faber et al. (2016) follow an interesting different line of research on improving the performance of subset optimization problems: For finding optimal solutions w.r.t. set inclusion, they use repeated calls of algorithms that produce solutions having optimum *cardinality*, based on the observation that cardinality-minimal solutions are also subset-minimal (and similarly for maximization). The authors implement this idea using different declarative programming systems (among them ASP). That work does not aim at fixed-parameter tractability, however.

CHAPTER 6

Conclusion

6.1 Summary

The goal of this thesis was to advance the field of solving problems at the second level of the polynomial hierarchy via algorithms that exploit bounded treewidth. A focus of this work was on Answer Set Programming (ASP), which is a popular tool for solving hard combinatorial problems, in particular on the second level, and on alliance problems in graphs, among which we can find interesting cases of problems that are also on the second level.

Our contributions include complementary techniques for taking advantage of small treewidth: On the one hand, we showed how we can take advantage of small treewidth *implicitly*, that is, without consciously writing a program that deliberately exploits treewidth. On the other hand, we presented improvements to the methodology of designing algorithms that *explicitly* take treewidth into account.

We briefly summarize the main contributions of this work grouped into our three thematic areas.

Treewidth-preserving ASP classes. We defined and analyzed classes of non-ground ASP programs called *guarded* and *connection-guarded* programs. These classes provide ASP users with an effective means of encoding their problems in such a way that small treewidth of the input is preserved in grounding. This is attractive because treewidth has been shown to have a massive impact on the solving performance of state-of-the-art ASP systems (Bliem et al. 2017). Our results enable ASP users to implicitly take advantage of the improved performance on groundings of small treewidth. Furthermore, our results provide insights into what happens to the treewidth of the input during grounding.

Alliance problems in graphs. We investigated a family of showcase problems from the area of alliance problems in graphs, which are of particular interest for ASP researchers due to their tricky encodings that require advanced techniques. We provided a thorough complexity analysis, which settled several long-standing questions. In particular, we proved that the SECURE SET problem is Σ_2^P-complete and that the DEFENSIVE ALLIANCE problem is W[1]-hard when parameterized by treewidth. On the positive side, we showed an FPT result for the SECURE SET VERIFICATION problem and a polynomial-time algorithm for SECURE SET on bounded treewidth. Moreover, we illustrated the use of guarded and connection-guarded ASP programs by encoding alliance problems in these ASP classes.

Advanced dynamic programming methodologies. We formalized an advanced methodology for obtaining dynamic programming algorithms that explicitly exploit bounded treewidth, and we proved correctness and fixed-parameter tractability. This approach is tailored to problems whose solutions must be subset-minimal in some way, which is the case for many problems on the second level of the polynomial hierarchy. Specifically, for any problem P whose solutions are exactly the subset-minimal solutions of some base problem B, we formalized how a dynamic programming algorithm for B can automatically be transformed into a dynamic programming algorithm for P. Furthermore, we showed how this approach can be generalized so that it is applicable to several problems at the second level of the polynomial hierarchy. This allows us to avoid a substantial amount of the overhead and redundant computations done by naive dynamic programming algorithms. Implementations of this approach have been shown to yield significant performance benefits in practice (Fichte et al. 2017; Bliem et al. 2016a).

6.2 Future Work

Our work opens up several interesting directions for future research in each of the considered areas.

Treewidth-preserving ASP classes. In Chapter 3, we compared our ASP classes to monadic second-order logic (MSO) and some of its extensions, which are often used for obtaining FPT results for problems parameterized by treewidth. By our results, the class of connection-guarded programs can be used to classify a problem as FPT when parameterized by the combination of treewidth and maximum degree. We are not aware of extensions of MSO for this more restrictive parameter, so our work may pave the way to such an extension.

From a more practical perspective, it is promising to look closely into what ASP solvers and in particular their heuristics are doing when they are presented with a grounding of small treewidth. This could provide us insight into why state-of-the-art ASP solvers perform better on instances of small treewidth even though they do not "consciously" exploit this fact. With the gained understanding, we may be able to improve their performance by explicitly taking information from a tree decomposition into account during solving. This could perhaps lead to a hybrid ASP solving approach that uses classical conflict-driven clause learning in combination with techniques employing tree decompositions.

Finally, for some of the problems considered in Section 3.5, we only obtained complexity bounds that are not tight, in particular for variants of the Brave Reasoning problem. Obtaining exact complexity results for these cases is a task for future work.

Alliance problems in graphs. We showed that Secure Set is not FPT when parameterized by treewidth under common complexity-theoretic assumptions. It is an open question which additional restrictions beside bounded treewidth need to be imposed to achieve fixed-parameter tractability. In particular, we do not know whether the problem becomes FPT when we use the combination of treewidth and maximum degree as the parameter. Conversely, we showed that Defensive Alliance is FPT on bounded treewidth, but perhaps we could obtain FPT results for parameters that are even less restrictive.

We proved $\mathsf{W}[1]$-hardness and XP-membership of Secure Set, so a tight bound is still lacking, albeit perhaps more of theoretical interest due to the fact that problems at a certain level of the weft hierarchy generally do not admit faster algorithms than problems at higher levels. As for the polynomial-time algorithm on instances of bounded treewidth, it would be interesting to investigate if we can obtain a similar result for less restrictive parameters like clique-width.

When comparing Defensive Alliance to Secure Set, we see that the problem statements only differ in the size of the subsets of solution candidates that need to be checked. Hence it may be interesting to study the complexity of Generalized Secure Set, which generalizes both of these problems, where also the size of the subsets can be investigated as a parameter. This may be interesting because even lowering the complexity to NP would be good news due to the fact that Secure Set is Σ_2^P-complete.

Finally, we only considered secure sets and defensive alliances, but there are also other variants of alliance problems in the literature. We leave it for future work to investigate if some of our reductions and algorithms can help in the study of such related problems.

Advanced dynamic programming methodologies. Our approach of transforming an algorithm for a base problem into an algorithm for a derived problem, as we described it in Chapter 5, only supports problems that have a rather simple relationship between their solution candidates and their counterexample candidates. Although we have sketched how straightforward extensions of the presented transformation can be achieved, these extended transformations still only apply to algorithms that employ some variant of subset minimization (or maximization). One possibility for future research is to investigate different relationships between solution candidates and counterexample candidates.

Another way of extending our work is to consider problems higher in the polynomial hierarchy than the second level. Such problems usually require additional levels of nesting for the dynamic programming tables.

There is also recent work on lazy evaluation of tree-decomposition-based dynamic programming (Bliem et al. 2016b), which turned out to bring substantial performance benefits. It may be worth considering if the approach presented in this thesis can be combined with lazy evaluation.

List of Figures

List of Listings

Index

Bibliography

Abiteboul, Serge, Richard Hull and Victor Vianu (1995). *Foundations of Databases*. Addison-Wesley. ISBN: 0-201-53771-0.

Abseher, Michael, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher and Stefan Woltran (2014). "The D-FLAT System for Dynamic Programming on Tree Decompositions". In: *Proceedings of the 14th European Conference on Logics in Artificial Intelligence (JELIA 2014), September 24-26, 2014, Funchal, Madeira, Portugal*. Ed. by Eduardo Fermé and João Leite. Vol. 8761. Lecture Notes in Computer Science. Springer, pp. 558–572. DOI: `10.1007/978-3-319-11558-0_39`.

Abseher, Michael, Bernhard Bliem, Günther Charwat, Frederico Dusberger and Stefan Woltran (2015). "Computing Secure Sets in Graphs Using Answer Set Programming". In: *Journal of Logic and Computation*. Accepted for publication. DOI: `10.1093/logcom/exv060`.

Alviano, Mario, Carmine Dodaro, Wolfgang Faber, Nicola Leone and Francesco Ricca (2013). "WASP: A Native ASP Solver Based on Constraint Learning". In: *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013), September 15–19, 2013, Corunna, Spain*. Ed. by Pedro Cabalar and Tran Cao Son. Vol. 8148. Lecture Notes in Computer Science. Springer, pp. 54–66. DOI: `10.1007/978-3-642-40564-8_6`.

Alviano, Mario, Carmine Dodaro, Nicola Leone and Francesco Ricca (2015). "Advances in WASP". In: *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2015), September 27–30, 2015, Lexington, KY, USA*. Ed. by Francesco Calimeri, Giovambattista Ianni and Mirosław Truszczyński. Vol. 9345. Lecture Notes in Computer Science. Springer, pp. 40–54. ISBN: 978-3-319-23263-8. DOI: `10.1007/978-3-319-23264-5_5`.

Alviano, Mario and Wolfgang Faber (2013). "The Complexity Boundary of Answer Set Programming with Generalized Atoms under the FLP Semantics". In: *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013), September 15–19, 2013, Corunna, Spain*. Ed. by Pedro Cabalar and

Tran Cao Son. Vol. 8148. Lecture Notes in Computer Science. Springer, pp. 67–72. DOI: 10.1007/978-3-642-40564-8_7.

Alviano, Mario, Wolfgang Faber and Martin Gebser (2015). “Rewriting Recursive Aggregates in Answer Set Programming: Back to Monotonicity”. In: *Theory and Practice of Logic Programming* 15.4-5, pp. 559–573. DOI: 10.1017/S1471068415000228.

Alviano, Mario, Wolfgang Faber, Nicola Leone, Simona Perri, Gerald Pfeifer and Giorgio Terracina (2011). “The Disjunctive Datalog System DLV”. In: *Revised Selected Papers of the 1st International Workshop on Datalog Reloaded (Datalog 2010), March 16–19, 2010, Oxford, UK*. Ed. by Oege de Moor, Georg Gottlob, Tim Furche and Andrew Jon Sellers. Vol. 6702. Lecture Notes in Computer Science. Springer, pp. 282–301. DOI: 10.1007/978-3-642-24206-9_17.

Arnborg, Stefan, Derek G. Corneil and Andrzej Proskurowski (1987). “Complexity of Finding Embeddings in a k-Tree”. In: *SIAM Journal of Algebraic and Discrete Methods* 8.2, pp. 277–284. DOI: 10.1137/0608024.

Arnborg, Stefan, Jens Lagergren and Detlef Seese (1991). “Easy Problems for Tree-Decomposable Graphs”. In: *Journal of Algorithms* 12.2, pp. 308–340. DOI: 10.1016/0196-6774(91)90006-K.

Asahiro, Yuichi, Eiji Miyano and Hirotaka Ono (2011). “Graph Classes and the Complexity of the Graph Orientation Minimizing the Maximum Weighted Outdegree”. In: *Discrete Applied Mathematics* 159.7, pp. 498–508. DOI: 10.1016/j.dam.2010.11.003.

Bichler, Manuel, Michael Morak and Stefan Woltran (2016). “The Power of Non-Ground Rules in Answer Set Programming”. In: *Theory and Practice of Logic Programming* 16.5-6, pp. 552–569. DOI: 10.1017/S1471068416000338.

Bichler, Manuel, Michael Morak and Stefan Woltran (2017). “lpopt: A Rule Optimization Tool for Answer Set Programming”. In: *Revised Selected Papers of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016), September 6–8, 2016, Edinburgh, UK*. Ed. by Manuel V. Hermenegildo and Pedro López-García. Vol. 10184. Lecture Notes in Computer Science. Springer, pp. 114–130. ISBN: 978-3-319-63138-7. DOI: 10.1007/978-3-319-63139-4_7.

Biere, Armin, Marijn Heule, Hans van Maaren and Toby Walsh, eds. (2009). *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. Amsterdam, The Netherlands: IOS Press. ISBN: 978-1-58603-929-5.

Bliem, Bernhard, Robert Bredereck and Rolf Niedermeier (2016). “Complexity of Efficient and Envy-Free Resource Allocation: Few Agents, Resources, or Utility Levels”. In: *Proceedings of the 25th International Joint Conference on Artificial Intelligence*

(IJCAI 2016), July 9–15, 2016, New York, NY, USA. Ed. by Subbarao Kambhampati. AAAI Press, pp. 102–108. ISBN: 978-1-57735-770-4.

Bliem, Bernhard, Günther Charwat, Markus Hecher and Stefan Woltran (2016a). "D-FLAT^2: Subset Minimization in Dynamic Programming on Tree Decompositions Made Easy". In: *Fundamenta Informaticae* 147.1, pp. 27–61. DOI: `10.3233/FI-2016-1397`.

Bliem, Bernhard, Markus Hecher and Stefan Woltran (2016). "On Efficiently Enumerating Semi-Stable Extensions via Dynamic Programming on Tree Decompositions". In: *Proceedings of the 6th International Conference on Computational Models of Argument (COMMA 2016), September 13–16, 2016, Potsdam, Germany*. Ed. by Pietro Baroni, Thomas F. Gordon, Tatjana Scheffler and Manfred Stede. Vol. 287. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 107–118. DOI: `10.3233/978-1-61499-686-6-107`.

Bliem, Bernhard, Benjamin Kaufmann, Torsten Schaub and Stefan Woltran (2016b). "ASP for Anytime Dynamic Programming on Tree Decompositions". In: *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016), July 9–15, 2016, New York, NY, USA*. Ed. by Subbarao Kambhampati. AAAI Press, pp. 979–986. ISBN: 978-1-57735-770-4.

Bliem, Bernhard, Marius Moldovan, Michael Morak and Stefan Woltran (2017). "The Impact of Treewidth on ASP Grounding and Solving". In: *Proceedings of the 26th Joint Conference on Artificial Intelligence (IJCAI 2017), August 19–25, 2017, Melbourne, Australia*. Ed. by Carles Sierra and Fahiem Bacchus. Accepted for publication. AAAI Press.

Bliem, Bernhard, Sebastian Ordyniak and Stefan Woltran (2016). "Clique-Width and Directed Width Measures for Answer-Set Programming". In: *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016), August 29–September 2, 2016, The Hague, The Netherlands*. Ed. by Gal A. Kaminka, Maria Fox, Paolo Bouquet, Eyke Hüllermeier, Virginia Dignum, Frank Dignum and Frank van Harmelen. Vol. 285. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 1105–1113. ISBN: 978-1-61499-671-2. DOI: `10.3233/978-1-61499-672-9-1105`.

Bliem, Bernhard, Reinhard Pichler and Stefan Woltran (2017). "Implementing Courcelle's Theorem in a Declarative Framework for Dynamic Programming". In: *Journal of Logic and Computation* 27.4, pp. 1067–1094. DOI: `10.1093/logcom/exv089`.

Bliem, Bernhard and Stefan Woltran (2016a). "Complexity of Secure Sets". In: *Revised Papers of the 41st International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2015), June 17–19, 2015, Garching, Germany*. Ed. by Ernst W. Mayr. Vol. 9224. Lecture Notes in Computer Science. Springer, pp. 64–77. ISBN: 978-3-662-53173-0. DOI: `10.1007/978-3-662-53174-7_5`.

Bliem, Bernhard and Stefan Woltran (2016b). “Equivalence between Answer-Set Programs under (Partially) Fixed Input”. In: *Proceedings of the 9th International Symposium on Foundations of Information and Knowledge Systems (FoIKS 2016), March 7–11, 2016, Linz, Austria*. Ed. by Marc Gyssens and Guillermo Ricardo Simari. Vol. 9616. Lecture Notes in Computer Science. Springer, pp. 95–111. DOI: 10.1007/978-3-319-30024-5_6.

Bliem, Bernhard and Stefan Woltran (2017). “Complexity of Secure Sets”. Version 3. In: arXiv: 1411.6549 [cs.CC].

Bodlaender, Hans L. (1993). “A Tourist Guide through Treewidth”. In: *Acta Cybernetica* 11.1-2, pp. 1–21.

Bodlaender, Hans L. (1996). “A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth”. In: *SIAM Journal on Computing* 25.6, pp. 1305–1317. DOI: 10.1137/S0097539793251219.

Bodlaender, Hans L. (1997). “Treewidth: Algorithmic Techniques and Results”. In: *Proceedings of the 22nd International Symposium on Mathematical Foundations of Computer Science (MFCS 1997), August 25–29, 1997, Bratislava, Slovakia*. Ed. by Igor Prívara and Peter Ruzicka. Vol. 1295. Lecture Notes in Computer Science. Springer, pp. 19–36. DOI: 10.1007/BFb0029946.

Bodlaender, Hans L. (2005). “Discovering Treewidth”. In: *Proceedings of the 31st Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2005), January 22–28, 2005, Liptovský Ján, Slovakia*. Ed. by Peter Vojtás, Mária Bieliková, Bernadette Charron-Bost and Ondrej Sýkora. Vol. 3381. Lecture Notes in Computer Science. Springer, pp. 1–16. DOI: 10.1007/978-3-540-30577-4_1.

Bodlaender, Hans L. and Arie M. C. A. Koster (2010). “Treewidth Computations I. Upper Bounds”. In: *Information and Computation* 208.3, pp. 259–275. DOI: 10.1016/j.ic.2009.03.008.

Bojańczyk, Mikołaj and Michal Pilipczuk (2017). “Optimizing Tree Decompositions in MSO”. In: *Proceedings of the 34th Symposium on Theoretical Aspects of Computer Science (STACS 2017), March 8–11, 2017, Hannover, Germany*. Ed. by Heribert Vollmer and Brigitte Vallée. Vol. 66. LIPIcs – Leibniz International Proceedings in Informatics. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 15:1–15:13. ISBN: 978-3-95977-028-6. DOI: 10.4230/LIPIcs.STACS.2017.15.

Brewka, Gerhard, James P. Delgrande, Javier Romero and Torsten Schaub (2015). “asprin: Customizing Answer Set Preferences without a Headache”. In: *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015), January 25–30, 2015, Austin, TX, USA*. Ed. by Blai Bonet and Sven Koenig. AAAI Press, pp. 1467–1474.

Brewka, Gerhard, Thomas Eiter and Mirosław Truszczyński (2011). “Answer Set Programming at a Glance”. In: *Communications of the ACM* 54.12, pp. 92–103. DOI: 10.1145/2043174.2043195.

Brigham, Robert C., Ronald D. Dutton and Stephen T. Hedetniemi (2007). “Security in Graphs”. In: *Discrete Applied Mathematics* 155.13, pp. 1708–1714. DOI: 10.1016/j.dam.2007.03.009.

Buccafurri, Francesco, Nicola Leone and Pasquale Rullo (2000). “Enhancing Disjunctive Datalog by Constraints”. In: *IEEE Transactions on Knowledge and Data Engineering* 12.5, pp. 845–860. DOI: 10.1109/69.877512.

Calimeri, Francesco, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca and Torsten Schaub (2015). *ASP-Core-2 Input Language Format*. https://www.mat.unical.it/aspcomp2013/ASPStandardization. Version: 2.03c.

Cami, Aurel, Hemant Balakrishnan, Narsingh Deo and Ronald D. Dutton (2006). “On the Complexity of Finding Optimal Global Alliances”. In: *Journal of Combinatorial Mathematics and Combinatorial Computing* 58, pp. 23–31.

Ceri, Stefano, Georg Gottlob and Letizia Tanca (1990). *Logic Programming and Databases*. Surveys in Computer Science. Berlin, Heidelberg, Germany: Springer-Verlag. ISBN: 978-3-642-83954-2. DOI: 10.1007/978-3-642-83952-8.

Courcelle, Bruno (1990). “The Monadic Second-Order Logic of Graphs I: Recognizable Sets of Finite Graphs”. In: *Information and Computation* 85.1, pp. 12–75. DOI: 10.1016/0890-5401(90)90043-H.

Courcelle, Bruno (1992). “The Monadic Second-Order Logic of Graphs III: Tree-decompositions, Minors and Complexity Issues”. In: *RAIRO Theoretical Informatics and Applications* 26, pp. 257–286. DOI: 10.1051/ita/1992260302571.

Courcelle, Bruno and Irène Durand (2016). “Computations by Fly-Automata Beyond Monadic Second-Order Logic”. In: *Theoretical Computer Science* 619, pp. 32–67. DOI: 10.1016/j.tcs.2015.12.026.

Courcelle, Bruno and Joost Engelfriet (2012). *Graph Structure and Monadic Second-Order Logic – A Language-Theoretic Approach*. Vol. 138. Encyclopedia of Mathematics and its Applications. Cambridge University Press. ISBN: 978-0-521-89833-1. DOI: 10.1017/CBO9780511977619.

Cygan, Marek, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk and Saket Saurabh (2015). *Parameterized Algorithms*. Cham, Switzerland: Springer International Publishing. ISBN: 9783319212746. DOI: 10.1007/978-3-319-21275-3.

Dantsin, Evgeny, Thomas Eiter, Georg Gottlob and Andrei Voronkov (2001). "Complexity and Expressive Power of Logic Programming". In: *ACM Computing Surveys* 33.3, pp. 374–425. DOI: 10.1145/502807.502810.

Dechter, Rina (2003). *Constraint Processing*. Amsterdam, The Netherlands: Elsevier Morgan Kaufmann. ISBN: 978-1-55860-890-0. DOI: 10.1016/b978-1-55860-890-0.x5000-2.

Dermaku, Artan, Tobias Ganzow, Georg Gottlob, Benjamin J. McMahan, Nysret Musliu and Marko Samer (2008). "Heuristic Methods for Hypertree Decomposition". In: *Proceedings of the 7th Mexican International Conference on Artificial Intelligence (MICAI 2008), October 27–31, 2008, Atizapán de Zaragoza, Mexico*. Ed. by Alexander F. Gelbukh and Eduardo F. Morales. Vol. 5317. Lecture Notes in Computer Science. Springer, pp. 1–11. DOI: 10.1007/978-3-540-88636-5_1.

Dom, Michael, Daniel Lokshtanov, Saket Saurabh and Yngve Villanger (2008). "Capacitated Domination and Covering: A Parameterized Perspective". In: *Proceedings of the 3rd International Workshop on Parameterized and Exact Computation (IWPEC 2008), May 14–16, 2008, Victoria, Canada*. Vol. 5018. Lecture Notes in Computer Science. Springer, pp. 78–90.

Downey, Rodney G. and Michael R. Fellows (1999). *Parameterized Complexity*. Monographs in Computer Science. New York, NY, USA: Springer. ISBN: 9780387948836. DOI: 10.1007/978-1-4612-0515-9.

Dvořák, Wolfgang, Reinhard Pichler and Stefan Woltran (2012). "Towards Fixed-Parameter Tractable Algorithms for Abstract Argumentation". In: *Artificial Intelligence* 186, pp. 1–37. DOI: 10.1016/j.artint.2012.03.005.

Eiter, Thomas and Georg Gottlob (1995). "On the Computational Cost of Disjunctive Logic Programming: Propositional Case". In: *Annals of Mathematics and Artificial Intelligence* 15.3-4, pp. 289–323. DOI: 10.1007/BF01536399.

Eiter, Thomas, Georg Gottlob and Heikki Mannila (1997). "Disjunctive Datalog". In: *ACM Transactions on Database Systems* 22.3, pp. 364–418. DOI: 10.1145/261124.261126.

Eiter, Thomas and Axel Polleres (2006). "Towards Automated Integration of Guess and Check Programs in Answer Set Programming: A Meta-Interpreter and Applications". In: *Theory and Practice of Logic Programming* 6.1-2, pp. 23–60. DOI: 10.1017/S1471068405002577.

Elkabani, Islam, Enrico Pontelli and Tran Cao Son (2005). "Smodels^{A} - A System for Computing Answer Sets of Logic Programs with Aggregates". In: *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2005), September 5–8, 2005, Diamante, Italy*. Ed. by Chitta Baral, Gianluigi

Greco, Nicola Leone and Giorgio Terracina. Vol. 3662. Lecture Notes in Computer Science. Springer, pp. 427–431. DOI: 10.1007/11546207_40.

Enciso, Rosa I. (2009). "Alliances in Graphs: Parameterized Algorithms and on Partitioning Series-Parallel Graphs". PhD thesis. Orlando, FL, USA: University of Central Florida.

Enciso, Rosa I. and Ronald D. Dutton (2008). "Parameterized Complexity of Secure Sets". In: *Congressus Numerantium* 189, pp. 161–168.

Faber, Wolfgang (2006). "Decomposition of Nonmonotone Aggregates in Answer Set Programming". In: *Online Proceedings of the 20th Workshop on Logic Programming (WLP 2006), February 22–24, 2006, Vienna, Austria.* Ed. by Michael Fink, Hans Tompits and Stefan Woltran, pp. 164–171.

Faber, Wolfgang, Gerald Pfeifer and Nicola Leone (2011). "Semantics and Complexity of Recursive Aggregates in Answer Set Programming". In: *Artificial Intelligence* 175.1, pp. 278–298. DOI: 10.1016/j.artint.2010.04.002.

Faber, Wolfgang, Gerald Pfeifer, Nicola Leone, Tina Dell'Armi and Giuseppe Ielpa (2008). "Design and Implementation of Aggregate Functions in the DLV System". In: *Theory and Practice of Logic Programming* 8.5-6, pp. 545–580. DOI: 10.1017/S1471068408003323.

Faber, Wolfgang, Mauro Vallati, Federico Cerutti and Massimiliano Giacomin (2016). "Solving Set Optimization Problems by Cardinality Optimization with an Application to Argumentation". In: *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016), August 29–September 2, 2016, The Hague, The Netherlands.* Ed. by Gal A. Kaminka, Maria Fox, Paolo Bouquet, Eyke Hüllermeier, Virginia Dignum, Frank Dignum and Frank van Harmelen. Vol. 285. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 966–973. ISBN: 978-1-61499-671-2. DOI: 10.3233/978-1-61499-672-9-966.

Fernau, Henning and Daniel Raible (2007). "Alliances in Graphs: A Complexity-Theoretic Study". In: *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2007), January 20–26, 2007, Harrachov, Czech Republic.* Ed. by Jan van Leeuwen, Giuseppe F. Italiano, Wiebe van der Hoek, Christoph Meinel, Harald Sack, Frantisek Plasil and Mária Bieliková. Vol. 2. Institute of Computer Science AS CR, Prague, pp. 61–70. ISBN: 80-903298-9-6.

Fernau, Henning and Juan A. Rodríguez-Velázquez (2014). "A Survey on Alliances and Related Parameters in Graphs". In: *Electronic Journal of Graph Theory and Applications (EJGTA)* 2.1, pp. 70–86. DOI: 10.5614/ejgta.2014.2.1.7.

Ferraris, Paolo (2011). "Logic Programs with Propositional Connectives and Aggregates". In: *ACM Transactions on Computational Logic* 12.4, 25:1–25:40. DOI: 10.1145/1970398.1970401.

Fichte, Johannes Klaus, Markus Hecher, Michael Morak and Stefan Woltran (2017). "Answer Set Solving with Bounded Treewidth Revisited". In: *Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2017), July 3–6, 2017, Espoo, Finland*. Ed. by Marcello Balduccini and Tomi Janhunen. Vol. 10377. Lecture Notes in Computer Science. Springer, pp. 132–145. DOI: 10.1007/978-3-319-61660-5_13.

Flake, Gary William, Steve Lawrence, C. Lee Giles and Frans Coetzee (2002). "Self-Organization and Identification of Web Communities". In: *IEEE Computer* 35.3, pp. 66–71. DOI: 10.1109/2.989932.

Flum, Jörg and Martin Grohe (2006). *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer. DOI: 10.1007/3-540-29953-X.

Ganian, Robert and Jan Obdrzálek (2013). "Expanding the Expressive Power of Monadic Second-Order Logic on Restricted Graph Classes". In: *Revised Selected Papers of the 24th International Workshop on Combinatorial Algorithms (IWOCA 2013), Rouen, France, July 10–12, 2013*. Ed. by Thierry Lecroq and Laurent Mouchard. Vol. 8288. Lecture Notes in Computer Science. Springer, pp. 164–177. ISBN: 978-3-642-45277-2. DOI: 10.1007/978-3-642-45278-9_15.

Gebser, Martin, Roland Kaminski, Benjamin Kaufmann, Javier Romero and Torsten Schaub (2015). "Progress in clasp Series 3". In: *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2015), September 27–30, 2015, Lexington, KY, USA*. Ed. by Francesco Calimeri, Giovambattista Ianni and Mirosław Truszczyński. Vol. 9345. Lecture Notes in Computer Science. Springer, pp. 368–383. ISBN: 978-3-319-23263-8. DOI: 10.1007/978-3-319-23264-5_31.

Gebser, Martin, Roland Kaminski, Benjamin Kaufmann and Torsten Schaub (2012). *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Williston, VT, USA: Morgan & Claypool Publishers. DOI: 10.2200/S00457ED1V01Y201211AIM019.

Gebser, Martin, Roland Kaminski and Torsten Schaub (2011). "Complex Optimization in Answer Set Programming". In: *Theory and Practice of Logic Programming* 11.4-5, pp. 821–839. DOI: 10.1017/S1471068411000329.

Gebser, Martin, Benjamin Kaufmann, André Neumann and Torsten Schaub (2007). "clasp: A Conflict-Driven Answer Set Solver". In: *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007), May 15–17, 2007, Tempe, AZ, USA*. Ed. by Chitta Baral, Gerhard Brewka and John S.

Schlipf. Vol. 4483. Lecture Notes in Computer Science. Springer, pp. 260–265. ISBN: 978-3-540-72199-4. DOI: 10.1007/978-3-540-72200-7_23.

Gebser, Martin, Benjamin Kaufmann and Torsten Schaub (2012). "Conflict-Driven Answer Set Solving: From Theory to Practice". In: *Artificial Intelligence* 187, pp. 52–89. DOI: 10.1016/j.artint.2012.04.001.

Gebser, Martin, Torsten Schaub, Sven Thiele and Philippe Veber (2011). "Detecting Inconsistencies in Large Biological Networks with Answer Set programming". In: *Theory and Practice of Logic Programming* 11.2-3, pp. 323–360. DOI: 10.1017/S1471068410000554.

Gelfond, Michael and Vladimir Lifschitz (1988). "The Stable Model Semantics for Logic Programming". In: *Proceedings of the 5th International Conference and Symposium on Logic Programming (JICSLP 1988), August 15–19, 1988, Seattle, WA, USA*. Ed. by Robert A. Kowalski and Kenneth A. Bowen. Vol. 2. The MIT Press, pp. 1070–1080.

Gottlob, Georg, Reinhard Pichler and Fang Wei (2010a). "Bounded Treewidth as a Key to Tractability of Knowledge Representation and Reasoning". In: *Artificial Intelligence* 174.1, pp. 105–132. DOI: 10.1016/j.artint.2009.10.003.

Gottlob, Georg, Reinhard Pichler and Fang Wei (2010b). "Tractable Database Design and Datalog Abduction through Bounded Treewidth". In: *Information Systems* 35.3, pp. 278–298. DOI: 10.1016/j.is.2009.09.003.

Harmelen, Frank van, Vladimir Lifschitz and Bruce W. Porter, eds. (2008). *Handbook of Knowledge Representation*. Vol. 3. Foundations of Artificial Intelligence. Elsevier. ISBN: 978-0-444-52211-5. DOI: 10.1016/s1574-6526(07)x0300-6.

Haynes, Teresa W., Stephen T. Hedetniemi and Michael A. Henning (2003). "Global Defensive Alliances in Graphs". In: *Electronic Journal of Combinatorics* 10.

Ho, Yiu Yu (2011). "Global Secure Sets of Trees and Grid-Like Graphs". PhD thesis. Orlando, FL, USA: University of Central Florida.

Ho, Yiu Yu and Ronald D. Dutton (2009). "Rooted Secure Sets of Trees". In: *AKCE International Journal of Graphs and Combinatorics* 6.3, pp. 373–392.

Jakl, Michael, Reinhard Pichler, Stefan Rümmele and Stefan Woltran (2008). "Fast Counting with Bounded Treewidth". In: *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2008), November 22–27, 2008, Doha, Qatar*. Ed. by Iliano Cervesato, Helmut Veith and Andrei Voronkov. Vol. 5330. Lecture Notes in Computer Science. Springer, pp. 436–450. ISBN: 978-3-540-89438-4. DOI: 10.1007/978-3-540-89439-1_31.

Jakl, Michael, Reinhard Pichler and Stefan Woltran (2009). "Answer-Set Programming with Bounded Treewidth". In: *Proceedings of the 21st International Joint Conference* *on Artificial Intelligence (IJCAI 2009), July 11–17, 2009, Pasadena, CA, USA*. Ed. by Craig Boutilier. AAAI Press, pp. 816–822.

Jamieson, Lindsay H. (2007). "Algorithms and Complexity for Alliances and Weighted Alliances of Various Types". PhD thesis. Clemson, South Carolina, USA: School of Computing, Clemson University.

Jamieson, Lindsay H., Stephen T. Hedetniemi and Alice A. McRae (2009). "The Algorithmic Complexity of Alliances in Graphs". In: *Journal of Combinatorial Mathematics and Combinatorial Computing* 68, pp. 137–150.

Kiyomi, Masashi and Yota Otachi (2017). "Alliances in Graphs of Bounded Clique-Width". In: *Discrete Applied Mathematics* 223, pp. 91–97. DOI: `10.1016/j.dam.2017.02.004`.

Kloks, Ton (1994). *Treewidth: Computations and Approximations*. Vol. 842. Lecture Notes in Computer Science. New York, NY, USA: Springer. DOI: `10.1007/BFb0045375`.

Knop, Dusan, Martin Koutecký, Tomás Masarík and Tomás Toufar (2017). "Simplified Algorithmic Metatheorems Beyond MSO: Treewidth and Neighborhood Diversity". In: arXiv: `1703.00544 [cs.CC]`.

Kornai, András and Zsolt Tuza (1992). "Narrowness, Pathwidth, and Their Application in Natural Language Processing". In: *Discrete Applied Mathematics* 36.1, pp. 87–92. DOI: `10.1016/0166-218X(92)90208-R`.

Kreutzer, Stephan (2012). "On the Parameterized Intractability of Monadic Second-Order Logic". In: *Logical Methods in Computer Science* 8.1. DOI: `10.2168/LMCS-8(1:27)2012`.

Kristiansen, Petter, Sandra M. Hedetniemi and Stephen T. Hedetniemi (2002). "Introduction to Alliances in Graphs". In: *Proceedings of the 17th International Symposium on Computer and Information Sciences (ISCIS 2002), October 28–30, 2002, Orlando, FL, USA*. Ed. by Ilyas Cicekli, Nihan Kesim Cicekli and Erol Gelenbe. CRC Press, pp. 308–312.

Kristiansen, Petter, Sandra M. Hedetniemi and Stephen T. Hedetniemi (2004). "Alliances in Graphs". In: *Journal of Combinatorial Mathematics and Combinatorial Computing* 48, pp. 157–178.

Langer, Alexander, Felix Reidl, Peter Rossmanith and Somnath Sikdar (2012). "Evaluation of an MSO-Solver". In: *Proceedings of the 14th Meeting on Algorithm Engineering & Experiments (ALENEX 2012), January 16, 2012, Kyoto, Japan*. Ed. by David A. Bader and Petra Mutzel. SIAM / Omnipress, pp. 55–63. ISBN: 978-1-61197-212-2. DOI: `10.1137/1.9781611972924.5`.

Langer, Alexander, Felix Reidl, Peter Rossmanith and Somnath Sikdar (2014). "Practical Algorithms for MSO Model-Checking on Tree-Decomposable Graphs". In: *Computer Science Review* 13-14, pp. 39–74. DOI: 10.1016/j.cosrev.2014.08.001.

Leone, Nicola, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri and Francesco Scarcello (2006). "The DLV System for Knowledge Representation and Reasoning". In: *ACM Transactions on Computational Logic* 7.3, pp. 499–562. DOI: 10.1145/1149114.1149117.

Libkin, Leonid (2004). *Elements of Finite Model Theory*. Texts in Computer Science. Berlin, Heidelberg, Germany: Springer. ISBN: 3-540-21202-7. DOI: 10.1007/978-3-662-07003-1.

Lifschitz, Vladimir (2008). "What Is Answer Set Programming?" In: *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI 2008), July 13–17, 2008, Chicago, IL, USA*. Ed. by Dieter Fox and Carla P. Gomes. AAAI Press, pp. 1594–1597.

Liu, Lengning and Mirosław Truszczyński (2006). "Properties and Applications of Programs with Monotone and Convex Constraints". In: *Journal of Artificial Intelligence Research* 27, pp. 299–334. DOI: 10.1613/jair.2009.

Lloyd, John W. (1987). *Foundations of Logic Programming, 2nd Edition*. Berlin, Heidelberg, Germany: Springer-Verlag. ISBN: 3-540-18199-7. DOI: 10.1007/978-3-642-83189-8.

Marek, Victor W. and Mirosław Truszczyński (1999). "Stable Models and an Alternative Logic Programming Paradigm". In: *The Logic Programming Paradigm: A 25-Year Perspective*. Ed. by Krzysztof Apt, Victor W. Marek, Mirosław Truszczyński and David S. Warren. Berlin, Heidelberg, Germany: Springer-Verlag, pp. 375–398. ISBN: 978-3-642-64249-4. DOI: 10.1007/978-3-642-60085-2.

Marx, Dániel (2011). "Complexity of Clique Coloring and Related Problems". In: *Theoretical Computer Science* 412.29, pp. 3487–3500. DOI: 10.1016/j.tcs.2011.02.038.

Matoušek, Jiří and Robin Thomas (1992). "On the Complexity of Finding Iso- and Other Morphisms for Partial k-Trees". In: *Discrete Mathematics* 108.1-3, pp. 343–364. DOI: 10.1016/0012-365X(92)90687-B.

McCarthy, John (1980). "Circumscription—A Form of Non-Monotonic Reasoning". In: *Artificial Intelligence* 13.1, pp. 27–39. DOI: 10.1016/0004-3702(80)90011-9.

Morak, Michael, Reinhard Pichler, Stefan Rümmele and Stefan Woltran (2010). "A Dynamic-Programming Based ASP-Solver". In: *Proceedings of the 12th European Conference on Logics in Artificial Intelligence (JELIA 2010), September 13–15, 2010, Helsinki, Finland*. Ed. by Tomi Janhunen and Ilkka Niemelä. Vol. 6341. Lecture Notes in Computer Science. Springer, pp. 369–372. DOI: 10.1007/978-3-642-15675-5_34.

Morak, Michael and Stefan Woltran (2012). “Preprocessing of Complex Non-Ground Rules in Answer Set Programming”. In: *Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012), September 4–8, 2012, Budapest, Hungary*. Ed. by Agostino Dovier and Vítor Santos Costa. Vol. 17. LIPIcs – Leibniz International Proceedings in Informatics. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 247–258. ISBN: 978-3-939897-43-9. DOI: 10.4230/LIPIcs.ICLP.2012.247.

Niedermeier, Rolf (2006). *Invitation to Fixed-Parameter Algorithms*. Vol. 31. Oxford Lecture Series in Mathematics and its Applications. Oxford, United Kingdom: Oxford University Press. ISBN: 9780198566076. DOI: 10.1093/acprof:oso/9780198566076.001.0001.

Nogueira, Monica, Marcello Balduccini, Michael Gelfond, Richard Watson and Matthew Barry (2001). “An A-Prolog Decision Support System for the Space Shuttle”. In: *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages (PADL 2001), March 11–12, 2001, Las Vegas, NV, USA*. Ed. by I. V. Ramakrishnan. Vol. 1990. Lecture Notes in Computer Science. Springer, pp. 169–183. ISBN: 3-540-41768-0. DOI: 10.1007/3-540-45241-9_12.

Papadimitriou, Christos H. (1994). *Computational Complexity*. Reading, MA, USA: Addison-Wesley. ISBN: 0-201-53082-1.

Pelov, Nikolay, Marc Denecker and Maurice Bruynooghe (2004). “Partial Stable Models for Logic Programs with Aggregates”. In: *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2004), January 6–8, 2004, Fort Lauderdale, FL, USA*. Ed. by Vladimir Lifschitz and Ilkka Niemelä. Vol. 2923. Lecture Notes in Computer Science. Springer, pp. 207–219. ISBN: 3-540-20721-X. DOI: 10.1007/978-3-540-24609-1_19.

Pelov, Nikolay, Marc Denecker and Maurice Bruynooghe (2007). “Well-Founded and Stable Semantics of Logic Programs with Aggregates”. In: *Theory and Practice of Logic Programming* 7.3, pp. 301–353. DOI: 10.1017/S1471068406002973.

Peters, Dominik (2017). “Σ_2^p-Complete Problems on Hedonic Games”. Version 2. In: arXiv: 1509.02333 [cs.GT].

Pichler, Reinhard, Stefan Rümmele, Stefan Szeider and Stefan Woltran (2014). “Tractable Answer-Set Programming with Weight Constraints: Bounded Treewidth Is Not Enough”. In: *Theory and Practice of Logic Programming* 14.2, pp. 141–164. DOI: 10.1017/S1471068412000099.

Robertson, Neil and Paul D. Seymour (1984). "Graph Minors. III. Planar Tree-Width". In: *Journal of Combinatorial Theory, Series B* 36.1, pp. 49–64. DOI: 10.1016/0095-8956(84)90013-3.

Rossi, Francesca, Peter van Beek and Toby Walsh, eds. (2006). *Handbook of Constraint Programming*. Vol. 2. Foundations of Artificial Intelligence. Elsevier. ISBN: 978-0-444-52726-4. DOI: 10.1016/s1574-6526(06)x8001-x.

Samer, Marko and Stefan Szeider (2010). "Algorithms for Propositional Model Counting". In: *Journal of Discrete Algorithms* 8.1, pp. 50–64. DOI: 10.1016/j.jda.2009.06.002.

Shafique, Khurram H. and Ronald D. Dutton (2003). "Maximum Alliance-Free and Minimum Alliance-Cover Sets". In: *Congressus Numerantium* 162, pp. 139–146.

Simons, Patrik, Ilkka Niemelä and Timo Soininen (2002). "Extending and Implementing the Stable Model Semantics". In: *Artificial Intelligence* 138.1-2, pp. 181–234. DOI: 10.1016/S0004-3702(02)00187-X.

Soininen, Timo and Ilkka Niemelä (1998). "Developing a Declarative Rule Language for Applications in Product Configuration". In: *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL 1999), January 18–19, 1999, San Antonio, TX, USA*. Ed. by Gopal Gupta. Vol. 1551. Lecture Notes in Computer Science. Springer, pp. 305–319. ISBN: 3-540-65527-1. DOI: 10.1007/3-540-49201-1_21.

Stockmeyer, Larry J. (1974). "The Complexity of Decision Problems in Automata Theory". PhD thesis. Department of Electrical Engineering, MIT.

Szeider, Stefan (2005). "Generalizations of Matched CNF Formulas". In: *Annals of Mathematics and Artificial Intelligence* 43.1, pp. 223–238. DOI: 10.1007/s10472-005-0432-6.

Szeider, Stefan (2011a). "Monadic Second Order Logic on Graphs with Local Cardinality Constraints". In: *ACM Transactions on Computational Logic* 12.2, 12:1–12:21. DOI: 10.1145/1877714.1877718.

Szeider, Stefan (2011b). "Not So Easy Problems for Tree Decomposable Graphs". In: arXiv: 1107.1177 [cs.DS].

Thorup, Mikkel (1998). "All Structured Programs have Small Tree-Width and Good Register Allocation". In: *Information and Computation* 142.2, pp. 159–181. DOI: 10.1006/inco.1997.2697.

Vardi, Moshe Y. (1982). "The Complexity of Relational Query Languages (Extended Abstract)". In: *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC 1982), May 5–7, 1982, San Francisco, CA, USA*. Ed. by Harry R. Lewis, Barbara

B. Simons, Walter A. Burkhard and Lawrence H. Landweber. ACM, pp. 137–146. ISBN: 0-89791-067-2. DOI: `10.1145/800070.802186`.

Yero, Ismael González and Juan A. Rodríguez-Velázquez (2013). "Defensive Alliances in Graphs: A Survey". In: arXiv: `1308.2096 [math.CO]`.