Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

TU UB WIEN Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Optimizing Second-Level Dynamic Programming Algorithms

The D-FLAT^2 System: Encodings and Experimental Evaluation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Markus Hecher
Matrikelnummer 1026412

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Mitwirkung: Dipl.-Ing. Bernhard Bliem BSc

Wien, 29. September 2015

Markus Hecher | Stefan Woltran

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Optimizing Second-Level Dynamic Programming Algorithms

The D-FLAT^2 System: Encodings and Experimental Evaluation

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Markus Hecher

Registration Number 1026412

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Assistance: Dipl.-Ing. Bernhard Bliem BSc

Vienna, 29th September, 2015

Markus Hecher	Stefan Woltran

Erklärung zur Verfassung der Arbeit

Markus Hecher
Am Vogelsang 91
2753 Ober-Piesting

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, 29. September 2015

____
Markus Hecher

Danksagung

Dieses Mal sollten die Dankesworte wohl etwas seriöser sein, daher habe ich beschlossen, dies zu tun. Zu diesem Anlass möchte ich vorerst allen danken, die in irgendeiner Form mitgewirkt haben. Ihr habt das super gemacht. Danke, lieber Leser, liebe Leserin, für eure Aufmerksamkeit.

Acknowledgements

Since this time the acknowledgements should be a little bit more serious, I decided to do so. First of all, I want to thank everyone involved somehow. You did a good job. Thanks, dearest readers for your attention.

Kurzfassung

Für viele bekannte AI-Probleme wurde bereits gezeigt, dass sie – unter der Annahme einer beschränkten Baumweite (tree-width) – in polynomieller Zeit und mit polynomiellen Speicheranforderungen lösbar (tractable) sind. Um diesen Umstand auch in der Praxis ausnutzen zu können, wurden relativ komplizierte Algorithmen für Baumzerlegungen (tree decompositions) basierend auf Dynamischer Programmierung (DP) entwickelt und implementiert. Typischerweise zeigen diese Algorithmen wiederkehrende Muster, die beispielsweise Teilmengen-Minimierung erfordern. Gerade bei der Answer-Set Programmierung (ASP) zeigt sich beispielsweise folgender Umstand: Um die volle Ausdrucksstärke dieser Technik auszunutzen – und damit auch co-NP-Tests durchführen zu können – ist es oft erforderlich, ein sogenanntes Saturierungsverfahren zu verwenden. Damit sich Benutzerinnen und Benutzer nicht immer mit Saturierung beschäftigen müssen, sind in diesem Zusammenhang viele Ansätze entwickelt worden, um sie und ihn zu entlasten.

Unglücklicherweise gibt es diese „Bequemlichkeit“ bei der DP noch nicht. Daher wird in dieser Arbeit eine neue Methode vorgestellt, um Algorithmen basierend auf DP für Baumzerlegungen zu vereinfachen; im Speziellen wird dadurch Teilmengen-Minimierung (und -Maximierung) automatisch durchgeführt. Um es anhand eines Beispiels zu beschreiben, sei ein SAT-Algorithmus (d.h. ein Algorithmus, der entscheidet, ob es für eine aussagenlogische Formel eine Wahrheitsbelegung der Variablen gibt, die die Formel erfüllt) basierend auf DP für Baumzerlegungen anzunehmen. Die hier vorgeschlagene Methode macht es nun möglich, diesen Algorithmus gemeinsam mit einer einfachen Angabe, worüber optimiert wird, in einen Algorithmus für das Aufzählen von Teilmengen-minimalen Modellen zu verwandeln. Für den Programmierer ist es also nicht mehr notwendig, sich explizit um die Optimierung zu kümmern, da diese durch den neuen Ansatz implizit passiert.

Weiters ist es oft der Fall, dass Problemlösungen via DP-Algorithmen für Baumzerlegungen wegen der Teilmengenoptimalität unnötig viel Zeit- und Speicherressourcen verbrauchen. Mit der vorgeschlagenen Methode wird dieses Problem dadurch umgangen, dass die Berechnung nun in zwei Phasen (statt einer) erfolgt. In Phase eins werden zuerst Lösungskandidaten ohne Berücksichtigung des Optimierungskriteriums berechnet; erst danach wird in der zweiten Phase versucht, diese Kandidaten durch Finden von Gegenbeispielen zu invalidieren, was durch den Zwei-Phasen-Ansatz sehr effizient durchgeführt werden kann.

Um die Bedeutung dieser Arbeit näher darzulegen, werden hier neben einer Implementierung dieses Zwei-Phasen-Ansatzes – genannt D-FLAT^2 – praktische Ergebnisse

gezeigt. Die Einfachheit und Eleganz von D-FLATˆ2 wird anhand einiger Programme für häufig vorkommende AI-Probleme präsentiert. Ferner wird ein neuer Algorithmus für die Berechnung von „semi-stable“ Mengen aus der Abstrakten Argumentation entwickelt. Praktische Ergebnisse zeigen schließlich, dass D-FLATˆ2 einen großen Performancevorteil – im Vergleich zu D-FLAT, das nur einphasig arbeitet – mit sich bringt und weiters die theoretischen Fixed-Paramter-Tractability (FPT) Resultate bezüglich Baumweite mit diesen Erkenntnissen kompatibel sind. In diesem Zusammenhang ergibt sich für unsere getesteten Probleme, dass die Optimierung annähernd kostenlos auftritt. In anderen Worten, die korrespondierenden Probleme ohne Optimierung brauchen ungefähr die gleichen Zeit- und Speicherressourcen.

Abstract

Many problems from the area of AI have been shown tractable for bounded tree-width. In order to put such results into practice, quite involved Dynamic Programming (DP) algorithms on Tree Decompositions (TDs) have to be designed and implemented. These algorithms typically show recurring patterns that call for tasks like subset-minimization. Especially in the world of Answer-Set Programming (ASP), we can witness such a phenomenon: In order to exploit the full expressive power of this paradigm, a particular saturation programming technique is required in order to express co-NP tests. Several approaches for relieving the user from this task have been proposed.

Unfortunately, easy-to-use facilities had no analogue in the area of DP so far. In this thesis, we provide a new method for obtaining DP algorithms on TDs from simpler principles, where subset-minimization is performed automatically. For example, given a DP algorithm on TDs for SAT, i.e. the problem whether a propositional formula is satisfiable, our approach makes it possible to use this algorithm, together with simple statements on what to minimize, for finding only subset-minimal models. Making optimization implicit in this way makes the programmer's life considerably easier.

Furthermore, standard DP algorithms on TDs for such problems often suffer from a naive check for subset optimality that requires unnecessarily much space and time. Our method avoids this issue by implicitly proceeding in two stages (instead of one stage in the standard case): First we compute solution candidates without regard to optimization and then we rule out invalid candidates by trying to find counterexamples. Because of its two-phased nature, our approach can do so in an efficient way. We are not aware of any work so far that introduces similar two-phased DP algorithms on TDs along with the appropriate data structures.

To underline the practical relevance of our work, we present an implementation of this two-stage algorithm called D-FLAT^2. We illustrate the simplicity of our approach by providing D-FLAT^2 encodings for several AI problems including a new algorithm for semi-stable semantics of Abstract Argumentation. Empirical results indicate a huge performance advantage of D-FLAT^2 compared to using only one stage as in D-FLAT and that the theoretical Fixed-Parameter Tractability (FPT) results w.r.t. tree-width are consistent with our experiments. Furthermore, we gathered that for our tested problems, which are FPT w.r.t. tree-width, it turns out that the optimization is almost for free, i.e. corresponding problems without optimization require about the same time and memory resources.

Contents

Kurzfassung xi

Abstract xiii

Contents xv

List of Figures xvii

List of Tables xviii

List of Algorithms xix

List of Listings xix

1 Introduction 1

2 Background 7
- 2.1 Fixed-Parameter Tractability . . . 7
- 2.2 Tree Decompositions . . . 8
- 2.3 Dynamic Programming on Tree Decompositions . . . 9
- 2.4 Computational Complexity . . . 12
- 2.5 Abstract Argumentation . . . 14
- 2.6 Answer-Set Programming . . . 17

3 DP algorithms on TDs for Abstract Argumentation 21
- 3.1 Modified algorithm for Admissible Semantics . . . 21
- 3.2 New algorithm for Semi-stable Semantics . . . 32
- 3.3 An Adaption for Preferred Semantics . . . 43

4 Towards Optimization of DP algorithms on TDs 49
- 4.1 D-FLAT: DP on TDs . . . 49
  - 4.1.1 System Overview . . . 50
  - 4.1.2 Technical Details . . . 51
  - 4.1.3 D-FLAT Encodings for Abstract Argumentation . . . 61
- 4.2 D-FLAT^2: Optimizing DP on TDs . . . 70

4.2.1 Technical Details . . . 70
4.2.2 Further Optimizations . . . 74
4.2.3 System Overview . . . 75
4.2.4 Application to Common AI Problems . . . 77

5 Benchmarks **85**
5.1 Test Environment . . . 85
5.1.1 Compared Tools . . . 85
5.1.2 Test Framework . . . 86
5.1.3 Test Conditions . . . 86
5.1.4 Problem Instances . . . 87
5.2 Monolithic Encodings . . . 87
5.3 Results . . . 91
5.3.1 System Comparison . . . 91
5.3.2 Problem Comparison . . . 91
5.3.3 Basis Semantics Comparison . . . 93

6 Conclusion **95**
6.1 Summary . . . 95
6.2 Further Work . . . 95

Bibliography **97**

List of Figures

2.1 Incidence graph G_{Ex} and a semi-normalized TD $\mathcal{T}_{Ex}$ of ϕ_{Ex}. 9
2.2 Tables (`length`(1)-item trees) for SAT of ϕ_{Ex} in $\mathcal{T}_{Ex}$. 11
2.3 Item trees for $\subseteq$-MINIMAL SAT of ϕ_{Ex} in $\mathcal{T}_{Ex}$. 13
2.4 Argumentation framework $F_{Ex} = (A_{F_{Ex}}, R_{F_{Ex}})$ with $A_{F_{Ex}} = \{v, w, x, y, z\}$ and $R_{F_{Ex}} = \{(w, x), (x, w), (w, y), (z, z), (z, x)\}$. 15

3.1 Instance $F_{Ex} = (A_{F_{Ex}}, R_{F_{Ex}})$ and a normalized TD $\mathcal{T}_{Ex}$. 45
3.2 Computation of vcolorings for $F_{Ex} = (A_{F_{Ex}}, R_{F_{Ex}})$ w.r.t. $\mathcal{T}_{Ex}$ (see Figure 3.1). 46
3.3 Computation of vpairs for $F_{Ex} = (A_{F_{Ex}}, R_{F_{Ex}})$ w.r.t. $\mathcal{T}_{Ex}$ (see Figure 3.1). . . 47
3.4 Computation of vprefpairs for $F_{Ex} = (A_{F_{Ex}}, R_{F_{Ex}})$ w.r.t. T_{Ex} (see Figure 3.1). 48

4.1 Flowchart that shows how D-FLAT and its components work [Bli12, ABC+14a]. 51
4.2 Data flow while processing a node with n children [Bli12, ABC+14a]. 51
4.3 Instance ϕ_{Ex}, ASP representation, incidence graph G_{Ex} and a semi-normalized TD $\mathcal{T}_{Ex}$. 52
4.4 Instance ϕ_{Ex}, ASP representation, primal graph G^{prim}_{Ex} and a semi-normalized TD $\mathcal{T}^{\text{prim}}_{Ex}$. 56
4.5 Item tree computation for exchange node n_3 of $\mathcal{T}_{Ex}$. 72
4.6 Item tree computation for join node n_8 of $\mathcal{T}_{Ex}$. 74
4.7 D-FLATˆ2 updated flowchart, which shows how the system and its components work. 75

5.1 8-Grid instance, $(n = 6 \times m)$, *tree-width* $= 7$ for every $m \geq n$ [Cha12]. 88
5.2 Clique instance, *tree-width* $= 5$ [Cha12]. 89
5.3 System comparison: Average CPU time (left) and maximum resident set (right). 92
5.4 Problem comparison: Average CPU time (left) and maximum resident set (right). 92
5.5 Basis semantics comparison: Average CPU time (left) and maximum resident set (right). 93

List of Tables

2.1 Complexity results for Abstract Argumentation. 17

4.1 Input predicates describing the Tree Decomposition. 52
4.2 Input predicates describing tables of decomposition child nodes. 54
4.3 Output predicates for constructing the table of the current decomposition node. 55
4.4 Input predicates describing item trees of child nodes in the decomposition. . . 58
4.5 Output predicates for constructing the item tree of the current decomposition node. 58

5.1 Restrictions for the benchmarks. 86

List of Algorithms

4.1 The procedure `computeLv2`. 71
4.2 The function `handleExchange`. 72
4.3 The function `handleJoin`. 74

List of Listings

2.1 $\Pi_{\texttt{ex}}$: ASP program for solving QBF ψ. 19
4.1 $\Pi_{\textsc{Sat}}$: D-FLAT encoding for solving Sat. 54
4.2 $\Pi^{\text{prim}}_{\textsc{Sat}}$: D-FLAT encoding for solving Sat using primal graphs. 55
4.3 $\Pi_{\subseteq\text{-}\textsc{Minimal Sat}}$: D-FLAT encoding for solving ⊆-Minimal Sat. 59
4.4 $\Pi^{\text{prim}}_{\subseteq\text{-}\textsc{Minimal Sat}}$: D-FLAT encoding for solving ⊆-Minimal Sat using primal graphs. 59
4.5 $\Pi_{\texttt{stable}}$: D-FLAT encoding for stable extensions. 62
4.6 $\Pi_{\texttt{admissible}}$: D-FLAT encoding for admissible sets. 63
4.7 $\Pi'_{\texttt{admissible}}$: Alternative D-FLAT encoding for admissible extensions. 64
4.8 $\Pi_{\texttt{complete}}$: D-FLAT encoding for complete extensions. 65
4.9 $\Pi_{\texttt{preferred}}$: D-FLAT encoding for directly computing preferred extensions. 66
4.10 $\Pi'_{\texttt{preferred}}$: Alternative D-FLAT encoding for computing preferred extensions. 67
4.11 $\Pi_{\texttt{semiStable}}$: D-FLAT encoding for directly computing semi-stable extensions. 68
4.12 $\Pi_{\texttt{optAllItems}}$: used for solving conceptually simple problems (e.g., ⊆-Minimal Sat). 76
4.13 $\Pi_{\texttt{optForSemiStable}}$: used for computing semi-stable sets via $\Pi^2_{\texttt{semiStable}} = \Pi_{\texttt{optForSemiStable}} \cup \Pi_{\texttt{admissible}}$. 78
4.14 $\Pi_{\texttt{optForCirc}}$: used for solving Circumscription via $\Pi^2_{\textsc{Circ}} = \Pi_{\texttt{optForCirc}} \cup \Pi_{\textsc{Sat}}$. 78
4.15 $\Pi_{\texttt{pseudoForASP}}$: used for solving disjunctive ASP via $\Pi^2_{\texttt{ASP}} = \Pi_{\texttt{pseudoForASP}} \cup \Pi^2_{\textsc{Circ}}$. 79
4.16 $\Pi'_{\texttt{ex}}$: ASP program for solving QBF φ. 80

4.17 Input instance $\Pi_{\mathtt{input}_{\mathtt{ex}}}$ for solving QBF φ using $\Pi^2_{\mathtt{ASP}}$. 80
4.18 $\Pi_{\mathtt{ASP}}$: D-FLAT encoding for solving disjunctive ASP. 81
4.19 $\Pi^{2'}_{\mathtt{ASP}}$: D-FLAT^2 encoding for solving disjunctive ASP directly. 83
5.1 Monolithic encoding for stable semantics. 87
5.2 Monolithic encoding for admissible semantics. 88
5.3 Monolithic encoding for complete semantics. 89
5.4 Monolithic encoding for preferred semantics. 90
5.5 Monolithic encoding for semi-stable semantics. 90

CHAPTER 1

Introduction

Motivation

The problem class NP contains problems that can be solved in polynomial time using non-deterministic Turing machines. Since it is still assumed that the class P of problems (which can be solved in polynomial time with a deterministic Turing machine) does not equal NP, one needs exponential time to cover this non-determinism. NP-hard problems are problems s.t. any problem of the class NP like the canonical propositional SAT problem can be transformed to them in polynomial time and space. Some problems are believed to be even harder than NP(like propositional Circumscription or disjunctive Answer-Set Programming[1]). These problems can be organized in the so-called polynomial hierarchy. Many prominent such problems in the area of AI, however, have been shown tractable for bounded tree-width. Tree-width measures the "tree-likeness" of a given graph s.t. the smaller the tree-width, the more tree-like the graph. Luckily, several real-world applications have input graphs of small tree-width. Since we believe that P is not equal to NP and also assume that the polynomial hierarchy does not collapse, those NP-hard problems can be located in the class NP or somewhere beyond.

Thanks to Courcelle's meta-theorem [Cou90], it is sufficient to encode a problem as an MSO sentence in order to obtain a Fixed-Parameter Tractability (FPT) result. The literature contains several FPT proofs assuming fixed (i.e., bounded) tree-width (which measures – as already mentioned – how tree-like a given graph is) for problems like SAT [GS08], CIRC, i.e. propositional Circumscription, resp. disjunctive Answer-Set Programming (ASP) [GPW10a] or several problems of Abstract Argumentation [Dun07, DSW12]. We can exploit this circumstance by designing Dynamic Programming (DP) algorithms on Tree Decompositions (TDs) for these tractable fragments of NP.

The actual design of DP algorithms on TDs, however, can be quite tedious, in particular for problems located at the second level of the polynomial hierarchy like Cir-

[1] *Answer-Set Programming* (ASP) deals with finding the so-called Answer-Sets, which are stable models of a given program.

cumscription, Abduction, Answer-Set Programming [BET11] or Abstract Argumentation (see [DPW12, JPW09, JPRW08, GPW10b]). In many cases, the increased complexity of such problems is caused by subset minimization or maximization subproblems (e.g., minimality of models in Circumscription). It is exactly the handling of these subproblems that makes the design of the DP algorithms on TDs difficult.

In this thesis, we discuss a solution to this issue by providing a method for automatically obtaining DP algorithms on TDs for problems requiring minimization, given only an algorithm for a problem variant without minimization. For example, given a DP algorithm on TDs for SAT (like in [SS10]), the approach of this thesis makes it possible to use this algorithm, together with simple statements on what to minimize, for finding only subset-minimal models. Making minimization implicit in this way makes the programmer's life considerably easier. Furthermore, standard DP algorithms on such problems often suffer from a naive check for subset minimality that requires unnecessarily much space and time. The main contribution of this thesis is providing a method that avoids this issue by proceeding in two stages (instead of one stage in the naive case). Although we explicitly only consider minimization, all the results – including promising performance experiments on AI problems – of course also apply to maximization.

State of the Art

To put Courcelle's meta-theorem [Cou90] into practice, tailored systems for MSO logic are required. While there has been remarkable progress in this direction [KLR11] there is still evidence that designing DP algorithms on TDs for the considered problems from scratch results in more efficient software solutions (cf. [Nie06]).

To facilitate the development of such algorithms, the *D-FLAT* system[2] [ABC$^+$14b] has been introduced. It allows for rapid prototyping by automatically generating a Tree Decomposition upon which it subsequently executes DP steps according to a specification given by the user. The crucial feature of D-FLAT is that the user can encode these problem-specific DP steps in Answer-Set Programming (ASP, see [BET11]), and such an encoding is all that is required from the user. It therefore enables implementation of DP algorithms on TDs in a rapid-prototyping style, which makes it an appealing general-purpose tool for designing such algorithms. It allows for running DP-based algorithms on TDs in one bottom-up traversal on automatically created TDs within Fixed-Parameter polynomial time (if the algorithm is FPT). Creating optimal tree representations (minimal tree-width) for graphs is NP-*hard*. Since D-FLAT exploits Fixed-Parameter Tractability w.r.t. tree-width, the width of the TD has a huge impact on the performance of this tool. Therefore, it uses heuristics in order to get rather good tree representations and to finally achieve tractability for a large class of NP-*complete* problems or even ones beyond NP.

In order to find minimal propositional models via ASP encoding, we simply need to express the SAT problem together with a special minimize statement (recognized by

[2]D-FLAT is an acronym for *Dynamic Programming Framework with Local Execution of ASP on Tree Decompositions* and freely available at `http://www.dbai.tuwien.ac.at/research/project/dflat/`.

systems like *metasp* [GKS11]). In this way, we easily obtain a program computing minimal models. Unfortunately, easy-to-use facilities like such minimize statements had no analogue in the area of DP so far; this circumstance directly leads to the aim of this thesis.

Main Contributions

One of the goals is to benefit in form of the new system[3] D-FLAT^2, based on D-FLAT, which shall extend the existing system by an automated mechanism for subset-optimization tailored for problems located on the second level of the polynomial hierarchy; thus, it greatly simplifies certain existing DP algorithms on TDs and thereby also improves its overall runtime performance.

As mentioned above, we will follow a two-stage approach. In the first stage we compute solution candidates without regard to minimization and then in the second stage we rule out invalid candidates by trying to find counterexamples while reusing already materialized solution candidates of the first stage. Because of efficient reusing of solution candidates and its two-phased nature (number of potential counterexamples is greatly reduced since counterexamples are added in the second phase by using already the end result of the first phase), our approach can do so in an efficient way. We are not aware of any work so far that introduces similar two-phased DP algorithms on TDs along with the appropriate data structures.

Moreover, this thesis shall provide several selected DP algorithms on TDs for AI-related problems; in particular simplified implementations of DP algorithms (for certain semantics of Abstract Argumentation [Dun95]) on TDs for D-FLAT^2 will be introduced. The design and implementation of these algorithms requires defining and maintaining suitable data structures, which heavily depend on the given problem.

Benchmarks shall be provided, by comparing optimized implementations of algorithms for D-FLAT^2 with existing versions for D-FLAT and state-of-the-art Answer-Set Solvers [GKK+11] with the help of monolithic encodings of *ASPARTIX*[4]. Thus, the strengths of D-FLAT^2 will be shown, which brings us one step closer towards tractability for common intractable graph problems.

To sum up, in this thesis, we

- provide a new easy-to-use mechanism for automatically performing subset optimization in two-layered DP algorithms on TDs;
- show applications thereof and its simple solution in practice;
- develop a new algorithm for semi-stable semantics and include a correctness proof;
- provide preliminary results, which show performance of D-FLAT^2;
- give hints concerning a common methodology on developing encodings for D-FLAT^2.

[3]The system D-FLAT^2 is available at `https://github.com/hmarkus/dflat-2`.

[4]More information about *Answer-Set Programming Argumentation Reasoning Tool* (ASPARTIX) can be found at `http://www.dbai.tuwien.ac.at/proj/argumentation/systempage/`.

Case Study: Dynamic Programming for Abstract Argumentation

The Dung framework basically consists of a relation R and arguments such that whenever for two arguments $(a, b) \in R$ holds, a directed attack from a to b is modeled. The goal now is to find certain subsets of these arguments, according to some semantics that fulfill certain conditions. The considered semantics of this thesis are as already mentioned *admissible*, *complete*, *preferred* and *stable*.

Abstract Argumentation has gained popularity in recent years. A number of different frameworks exist, focusing not only on attack relations, but for instance also implementing combined attacks or support relations. Therefore plenty of topics in this field are active research. In [DSW12] several semantics and reasoning modes are discussed and FPT results w.r.t. bounded tree-width are proven by reduction to MSO.

This thesis includes a new algorithm for semi-stable extensions of Abstract Argumentation and a correctness proof for it. Previous work [DPW12] discusses a full correctness proof for admissible semantics and then extends it to preferred extensions. On top of this, we modified the algorithm for admissible sets in order to obtain one for computing semi-stable extensions. In order to prove correctness of certain DP algorithms on TDs, formal methods are required; especially for problems located on the second level of the polynomial hierarchy (for instance for computing preferred or semi-stable extensions) [DB02, DW10, Dvo12] this quickly gets tedious.

Encodings for D-FLATˆ2 will be presented in the light of readability, reusability and maintainability, since D-FLATˆ2 elegantly allows reusing common parts of encodings. Moreover, a methodology hint on how to write such encodings shall be presented. In order to show D-FLATˆ2's efficiency and prove practical applicability, we present results using common AI problems (located on the second level of the polynomial hierarchy) and compare its memory and time performance with existing tools. In case of small tree-width, the subset-minimization resp. -maximization part of a given optimization problem – which is FPT w.r.t. tree-width – seems to be almost for free. In other words, if we are given an algorithm A for a basis problem P with FP tractability w.r.t. tree-width and an extended optimization problem P' involving subset-minimization resp. maximization on top of P, the algorithm A' (extended by D-FLATˆ2) for P' appears to require similar time and memory requirements than A in case of small tree-width. Since all of the problems used in this thesis are actually Fixed-Parameter Linear (FPL) w.r.t. tree-width – as already mentioned, this can be shown by reduction to MSO and using the meta-theorem by Courcelle [Cou90] – D-FLATˆ2 can be fruitfully applied. For FPT results concerning Abstract Argumentation, we refer to [DSW12]; for Circumscription, disjunctive ASP (and Abduction), our desired results are shown in [GPW10a]; results for the propositional SAT problem appear in [GS08].

Publications

This thesis complements and extends both the description of D-FLATˆ2 [BCHW15b], which focuses more on the technical realization of the system, and our paper [BCHW15a] concerning D-FLATˆ2 in practice.

Overview

First of all, necessary theoretical background on the topics is provided in Chapter 2, which includes in particular Abstract Argumentation, Answer-Set Programming and Dynamic Programming on Tree Decompositions. Next, Chapter 3 modifies existing algorithms of admissible extensions, then extends them further and provides a new algorithm for semi-stable semantics where correctness is proved. D-FLAT and encodings for selected problems (including ASP encodings) will be discussed in Chapter 4, followed by the implementation of our approach – D-FLAT^2 – and its underlying technical details. The benchmark set-up, the results and a short discussion are available in Chapter 5. Finally, the last chapter of this thesis is a conclusion and hints at further work.

CHAPTER 2

Background

In this chapter we outline Dynamic Programming on Tree Decompositions. The ideas underlying this concept stem from the field of parameterized complexity. Many computationally hard problems become tractable in case a certain problem parameter is bound to a fixed constant. This property is referred to as *Fixed-Parameter Tractability* (FPT) [DF99], and the complexity class FPT consists of problems that are solvable in $f(k) \cdot n^{\mathcal{O}(1)}$, where f is a function that only depends on the parameter k, and n is the input size.

2.1 Fixed-Parameter Tractability

As a matter of fact, D-FLAT^2, the software developed in the focus of this thesis, is an application of Fixed-Parameter Tractability. This means that the goal is to get tractability by taking advantage of some characteristic parameter being bounded. This is done here by creating Tree Decompositions and using tree-width as the desired parameter. The fact that a tree-decomposed problem with k-bound tree-width can be solved in linear time for such graphs is often shown with the help of Courcelle's meta-theorem using monadic second order logic (MSO) [Cou90]. This meta-theorem does not lead to practically efficient algorithms, but shows that there is an algorithm with runtime bounded by a function of the form $f(k) \cdot |I|^{(O(1))}$, where I is the problem instance and $n = |I|$ is its size. So the runtime basically has an upper-bound of a polynomial of the input size multiplied by a function of k. Considering any fixed k, this formula is polynomial as $f(k)$ is constant. Therefore the fixed k is responsible for the tractability, the algorithm obviously is in P under these limitations.

2.2 Tree Decompositions

For problems whose input can be represented as a graph, one important parameter is *tree-width*, which, roughly speaking, measures the "tree-likeness" of a graph. It is defined by means of Tree Decompositions (TDs), originally introduced in [RS84]. The intuition behind TDs 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 $G = (V, E)$ is a pair $\mathcal{T} = (T, \chi)$ where $T = (N, F)$ is a (rooted) tree and $\chi : N \to 2^V$ assigns to each node a set of vertices (called the node's* bag*), such that the following conditions are met: 1. For every vertex $v \in V$, there exists a node $n \in N$ such that $v \in \chi(n)$. 2. For every edge $e \in E$, there exists a node $n \in N$ such that $e \subseteq \chi(n)$. 3. For every $v \in V$, the subtree of T induced by $\{n \in N \mid v \in \chi(n)\}$ is connected.*

We call $\max_{n \in N} |\chi(n)| - 1$ *the* width *of the decomposition. The* tree-width *of a graph is the minimum width over all its Tree Decompositions.*

Note that for constructing TDs, the input graph G can be directed as well. In this case we only consider the *shadow* of G, which is the graph G' that is obtained by not considering the directions of the edges of G.
In general, constructing a TD with minimum width is intractable [ACP87]. However, there are heuristics that give "good" TDs in polynomial time [Dec03, DGG$^+$08, BK10].

Tree Decompositions can be defined in a generalized way for hypergraphs, which allow edges between an arbitrary number of vertices [GLS02]. D-FLAT^2, in fact, automatically tries to generate hypergraph tree decompositions of minimum width, which in general are *NP-hard* to compute, but it uses efficient and randomized heuristics to overcome this restriction. More about Tree Decompositions and structural decomposition techniques and its uses in general can be found at [ADG$^+$11].

In this thesis we will mainly consider so-called *semi-normalized* TDs:

Definition 2.2. *A Tree Decomposition $\mathcal{T} = (T, \chi)$ with $T = (N, F)$ is* semi-normalized *if each non-leaf node $n \in N$ is an* exchange node *(n has exactly one child), or a* join node *(n has exactly two children n', n'' with $\chi(n) = \chi(n') = \chi(n'')$).*

Furthermore, we assume that for root node r of T, $\chi(r) = \emptyset$ holds. Especially for Chapter 3, we additionally require *normalized* TDs.

Definition 2.3. *A Tree Decomposition $(\mathcal{T}, \mathcal{X})$ of a graph G is called normalized if $\mathcal{T}$ is a rooted tree and if each node* [1] *$t \in \mathcal{T}$ is of one of the following types.*

1. *LEAF: t is a leaf of $\mathcal{T}$;*
2. *FORGET: t has only one child t' and $X_t = X_{t'} \setminus \{v\}$ for some $v \in X_{t'}$;*
3. *INSERT: t has only one child t' and $X_t = X_{t'} \cup \{v\}$ for some $v \notin X_{t'}$;*
4. *JOIN: t has two children t', t'' and $X_t = X_{t'} = X_{t''}$.*

[1] For $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$ we often write $t \in \mathcal{T}$ instead of $t \in V_\mathcal{T}$

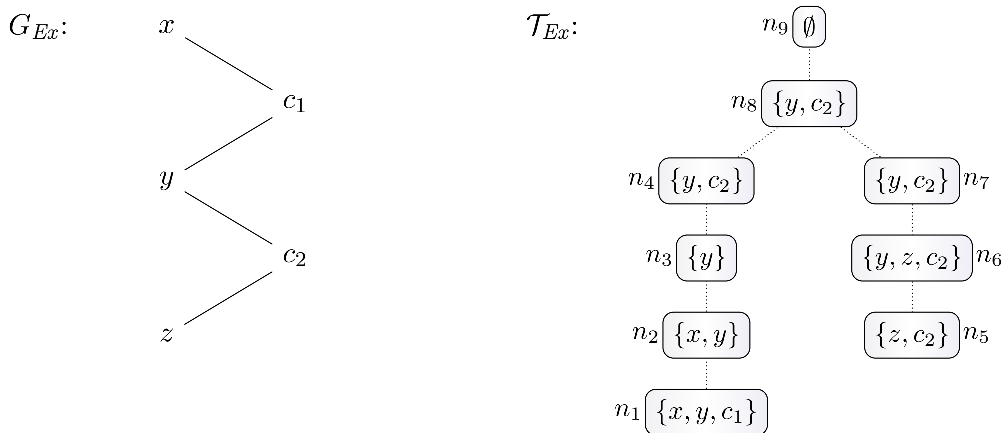

Figure 2.1: Incidence graph G_{Ex} and a semi-normalized TD $\mathcal{T}_{Ex}$ of ϕ_{Ex}.

A TD can be transformed into a semi-normalized resp. normalized one in linear time without increasing the width [Klo94].

The enumeration variants of the SAT and ⊆-MINIMAL SAT problems ("Given a propositional formula ϕ in CNF, what are the (subset-minimal) models of ϕ?") will serve as running examples throughout this section. These problems are well suited since the DP algorithms incorporate concepts that often reappear in other AI-related problem domains. To obtain a TD of a CNF-formula ϕ, we first have to construct an appropriate graph representation. Let $\mathcal{C}$ denote the set of clauses and $\mathcal{A}$ the atoms in ϕ. Furthermore, let $at(c)$ denote the atoms occurring in $c \in \mathcal{C}$. Then, the *incidence graph* $G = (V, E)$ of ϕ is given as $V = \mathcal{C} \cup \mathcal{A}$ and $E = \bigcup_{c \in \mathcal{C}} \{\{a, c\} \mid a \in at(c)\}$.

Example 2.4. *Let $\phi_{Ex} = (x \vee y) \wedge (\neg y \vee z)$. We have $\mathcal{C} = \{c_1, c_2\}$ with $c_1 = x \vee y$, $c_2 = \neg y \vee z$, $at(c_1) = \{x, y\}$ and $at(c_2) = \{y, z\}$. The corresponding incidence graph G_{Ex} and a possible (semi-normalized) TD $\mathcal{T}_{Ex}$ are depicted in Figure 2.1. The width of $\mathcal{T}_{Ex}$ is 2.*

2.3 Dynamic Programming on Tree Decompositions

Algorithms for DP on TDs generally traverse the TD in bottom-up order. At each node, partial solutions for the subgraph induced by the vertices encountered so far are computed and stored in a data structure associated with the node. The size of the data structure is typically bounded by the width of the TD and the number of TD nodes is linear in the input size. Hence, if the width is bounded by a constant, the search space for the subproblem is constant as well, and the number of subproblems only grows by a linear factor for larger instances. The most challenging task when designing DP algorithms on TDs is to identify the data structure on how to represent partial solution candidates at each node.

In the following, so-called *item trees* [ABC$^+$14b] will serve as the data structure for storing partial solutions. In Section 4.2.1 we will present modifications that allow us to solve several problems more efficiently that are hard for the second level of the polynomial hierarchy. Each item tree node contains an *item set* whose elements are called *items*. Usually, the information stored in the items is restricted to (or dependent on) the bag elements of the respective decomposition node. Each item tree node additionally has a set of *extension pointer tuples* that represents its origin. With this, one can reconstruct solutions for the complete problem instance by starting at the root of the TD and following the extension pointer tuples while combining the contents of the respective item sets. These concepts are formalized as follows:

Definition 2.5. *Let $\mathcal{T} = (T, \chi)$ with $T = (N, F)$ be a Tree Decomposition, and let $n \in N$. The* item tree *of n is a triple (S_n, X_n, Y_n) where $S_n = (T_n, E_n)$ is a rooted tree. Furthermore, X_n is a function that assigns to each* item tree node *$t_n \in T_n$ an* item set*, where each item is some arbitrary string. Let $n_1, \ldots, n_x \in N$ be the child nodes of n in T, let t_n be an item tree node in S_n, and let, for $1 \leq i \leq x$, T_{n_i} be the item tree nodes in S_{n_i}. Then, Y_n is a function that assigns to each t_n a non-empty set of* extension pointer tuples*, where each tuple is of the form $(t_1, \ldots, t_x)$, such that $t_i \in T_{n_i}$ for $1 \leq i \leq x$. Finally, we inductively define the set of* extensions *of t_n as $Z(t_n) = \{X_n(t_n) \cup A \mid A \in \bigcup_{(p_1,\ldots,p_x) \in Y_n(t_n)} \{e_1 \cup \cdots \cup e_x \mid e_i \in Z(p_i)\}\}$.*

For a DP algorithm solving the SAT problem on a TD of the input formula's incidence graph, we store partial solutions in the item sets at depth 1 of the item trees. For a TD node n, these item sets are subsets of $\chi(n)$. Intuitively, each item set represents a partial interpretation for ϕ, together with the clauses satisfied by the interpretation, restricted to $\chi(n)$. At each decomposition node n, we obtain an item tree node by extending one node from the item tree of each child of n. If n is an exchange node with child n', we additionally guess for every introduced atom (i.e., from $\chi(n) \setminus \chi(n')$), whether it is true and if so, we add it to the item set. If a clause from $\chi(n)$ is satisfied by the interpretation represented by this item set, we additionally store that clause in the set. Whenever an atom a is removed from the bag (i.e., in $\chi(n') \setminus \chi(n)$), a is not put into an item set at n. For a clause c that is removed from the bag, we only extend item sets that contained c, as other item sets represent partial interpretations that do not satisfy c. In join nodes, we extend pairs of item tree nodes that coincide on the partial interpretations, and compute the union of the already satisfied clauses. Intuitively, the partial interpretations have to agree on the truth assignment for common atoms in order to be a solution for the complete problem instance.

Example 2.6. *Figure 2.2 illustrates the item trees for $\mathcal{T}_{Ex}$ of ϕ_{Ex}. For instance, in n_1 we store the partial interpretations $\emptyset, \{x\}, \{y\}, \{x, y\}$. The latter three satisfy clause c_1, which is additionally stored in the respective item sets. In n_2, c_1 is removed. Interpretation $\emptyset$ is not extended, since it does not satisfy c_1. On the other hand, the item tree node in n_2 containing $\{x\}$, for example, extends the item tree node $\{x, c_1\}$ in n_1. In the figure, extension pointer tuples are marked with dashed lines. In n_8, the item tree node with item set $\{y, c_2\}$ extends $\{y\}$ in n_4 and $\{y, c_2\}$ in n_7, since the latter two both contain the*

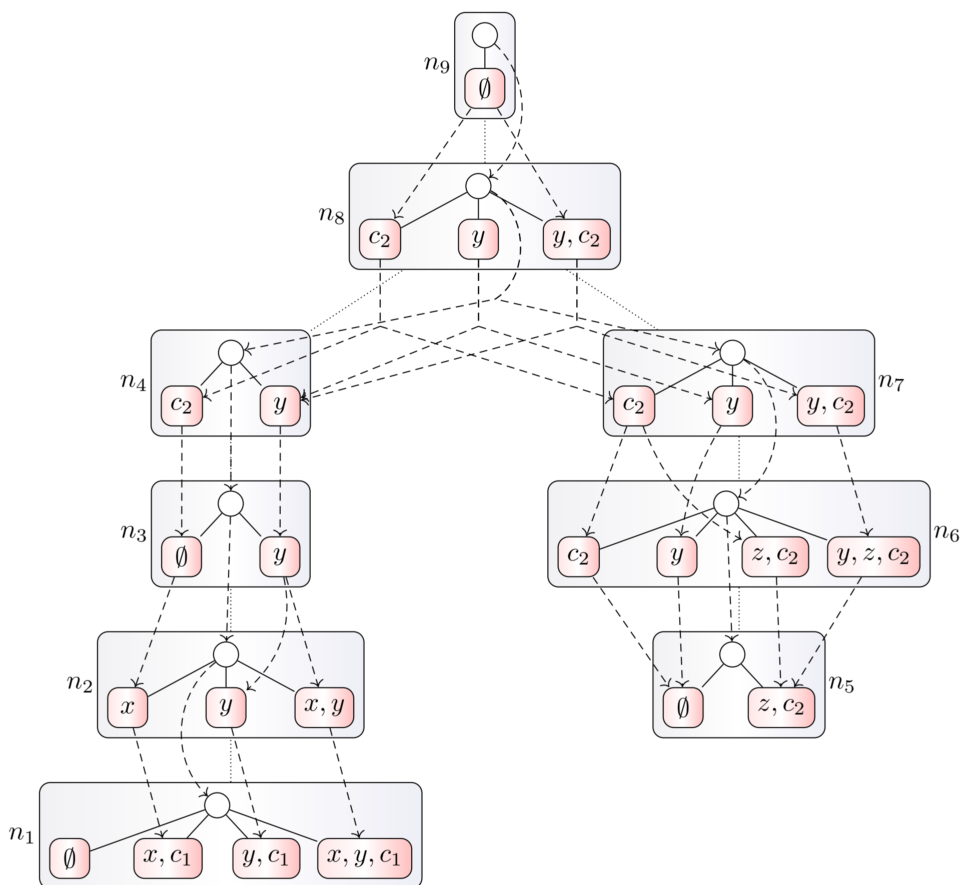

Figure 2.2: Tables (`length`(1)-item trees) for SAT of ϕ_{Ex} in $\mathcal{T}_{Ex}$.

same partial interpretation. Here, the extension pointer tuple is of arity 2 and contains references to both extended item tree nodes. For enumerating the solutions, we follow the extension pointer tuples, starting at the root of the decomposition, and build the union over the item sets, resulting in $\{\{x, c_1, c_2\}, \{x, z, c_1, c_2\}, \{y, z, c_1, c_2\}, \{x, y, z, c_1, c_2\}\}$. *The models of* ϕ_{Ex} *are* $\{\{x\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$.

Next, let us consider the $\subseteq$-MINIMAL SAT problem. Here, solution candidates are again stored at depth 1 of item trees. Furthermore, we store so-called *counter candidates* at depth 2. A counter candidate is a potential witness for the solution candidate (its parent) not being subset-minimal. This concept of storing witnesses (also called *certificates*) is commonly used for problems that are hard for the second level of the polynomial hierarchy [JPW09]. Item sets for solution and counter candidates are computed as for the SAT problem. Additionally, partial interpretations represented by counter candidates are strict subsets of partial interpretations represented by solution candidates. In the following, we give details on how and when these counter candidates are constructed (for $\subseteq$-MINIMAL SAT). For an introduced atom a in decomposition node n, whenever we have

that a is in the item set of some solution candidate t_n, we add a new child whose item set contains a and the atoms and clauses as the extended item set of t_n Furthermore, a new child for t_n is added whose item set contains the same atoms and clauses as the extended item set of t_n. This child represents a new potential witness for the partial interpretation associated with t_n being not subset-minimal. In join nodes, solution candidates are extended as in the SAT problem. Furthermore, we construct associated counter candidates by combining two counter candidates or a counter candidate of one item tree with the solution candidate of the other item tree, whenever they coincide on the partial interpretations. Intuitively, a counter candidate at the join node either represents a smaller interpretation in both child item trees, or in one of them together with the solution candidate stored in the other item set. For removed atoms, and introduced and removed clauses, the item sets are updated as described for SAT. At the root node, item tree nodes at depth 1 without associated counterexamples represent subset-minimal models of the overall problem instance, and the models are obtained by following the respective extension pointer tuples.

Example 2.7. *Consider again $\mathcal{T}_{Ex}$ of ϕ_{Ex}. Figure 2.3 contains the computed item trees for TD nodes n_1, n_2 and n_3. In n_1, we construct the item sets at depth 1 as described before. The item sets at depth 2 represent counter candidates that are strict subsets of the interpretations at depth 1. In n_2, clause c_1 is removed. Hence, we remove all item sets representing interpretations that do not satisfy c_1. In n_3, atom x is removed. Observe that this results in two item sets at depth 1 that both solely contain y, but differ in the counter candidates stored at depth 2.*

2.4 Computational Complexity

Since this thesis is not mainly directed towards complexity theory, we introduce here just some required complexity classes. A given problem P is C-*complete* if P is in the class C (membership) and P is C-*hard*, i.e. any problem of C can be reduced to P in polynomial time and space.

Problems in P can be solved in polynomial time with a deterministic Turing machine. The problem class NP contains problems that can be solved in polynomial time using non-deterministic Turing machines. For any class C, the class co-C is the set of problems, whose complement is in C, i.e., a problem P is in co-C if and only if the complement co-C of P is in C. Σ_2^{P} is the class $\mathsf{NP}^{\mathsf{NP}} = \mathsf{NP}^{\text{co-}\mathsf{NP}}$, i.e. its members can be solved by a non-deterministic Turing machine, which has access to an NP oracle in every step, which itself can use the power of a non-deterministic Turing machine; Π_2^{P} is the class co-$\mathsf{NP}^{\mathsf{NP}}$ = co-$\mathsf{NP}^{\text{co-}\mathsf{NP}}$. One can generalize further (and show some classes of the *polynomial hierarchy* PH):

- $\Sigma_0^{\mathsf{P}} = \mathsf{NP}, \Pi_0^{\mathsf{P}} = \text{co-}\mathsf{NP}$
- $\Sigma_i^{\mathsf{P}} = \mathsf{NP}^{\Sigma_{i-1}^{\mathsf{P}}}, \Pi_i^{\mathsf{P}} = \text{co-}\mathsf{NP}^{\Pi_{i-1}^{\mathsf{P}}}$ for $i \geq 1$

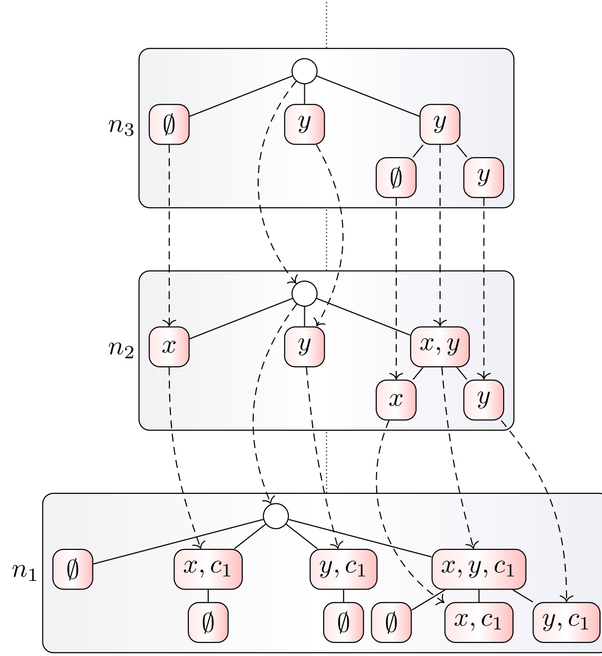

Figure 2.3: Item trees for ⊆-Minimal Sat of ϕ_{Ex} in $\mathcal{T}_{Ex}$.

Quantified Boolean Formulas Literature showed that *Quantified Boolean Formulas* (QBFs) are useful for showing complexity results due to one of its unique properties explained in Proposition 2.1 below. The set of well-formed QBFs Q is the smallest set that obeys the following rules (assuming *atoms* resp. *variables* is the set of atoms resp. variables):

- If $c \in \{\top, \bot\}$, then $c \in Q$
- If $a \in$ atoms, then $a \in Q$
- If $x \in$ variables, then $x \in Q$
- If $\varphi \in Q$, then $(\neg\varphi) \in Q$
- If $\varphi, \psi \in Q$ and $\circ \in \{\vee, \wedge\}$, then $(\varphi \circ \psi) \in Q$
- If $\varphi \in Q$, $Y \in \{\exists, \forall\}$ and $x \in$ variables, then $(Yx(\varphi)) \in Q$

It is common practice to forget about parentheses in case precedence is clear assuming the following typical order $\neg < \wedge < \vee$.

We call $\varphi \in Q$ *satisfiable* if there is some $\mathcal{M}$ with $\mathcal{M} \models \varphi$ where $\mathcal{M}$: (variables $\cup$ atoms) $\to \{\top, \bot\}$. $\varphi \in Q$ is *valid* if for any $\mathcal{M}$ defined as above $\mathcal{M} \models \varphi$ holds.

The semantic $\models$-relation is defined below. For this we need the notion $\mathcal{M} =_{\%x} \mathcal{M}'$, which means that $\mathcal{M}$ is equal to $\mathcal{M}'$ modulo variable x, i.e. $\mathcal{M}(y) = \mathcal{M}'(y)$ for any $y \in$ (variables $\cup$ atoms)$\setminus\{x\}$.

- $\mathcal{M} \models \top$, $\mathcal{M} \not\models \bot$
- If $a \in$ atoms and $\mathcal{M}(a) = \top$, then $\mathcal{M} \models a$
- If $x \in$ variables and $\mathcal{M}(x) = \top$, then $\mathcal{M} \models x$
- If $\varphi \in Q$ and $\mathcal{M} \not\models \varphi$, then $\mathcal{M} \models \neg\varphi$
- If $\varphi \wedge \psi \in Q$ and $\mathcal{M} \models \varphi, \psi$, then $\mathcal{M} \models \varphi \wedge \psi$
- If $\varphi \vee \psi \in Q$ and ($\mathcal{M} \models \varphi$ or $\mathcal{M} \models \psi$), then $\mathcal{M} \models \varphi \vee \psi$
- If $x \in$ variables, $\exists x\ \varphi \in Q$, $\mathcal{M}'(x) = \top$, $\mathcal{M}''(x) = \bot$, $\mathcal{M} =_{\%x} \mathcal{M}' =_{\%x} \mathcal{M}''$ and ($\mathcal{M}' \models \varphi$ or $\mathcal{M}'' \models \varphi$) then $\mathcal{M} \models \exists x\ \varphi$
- If $x \in$ variables, $\forall x\ \varphi \in Q$, $\mathcal{M}'(x) = \top$, $\mathcal{M}''(x) = \bot$, $\mathcal{M} =_{\%x} \mathcal{M}' =_{\%x} \mathcal{M}''$, $\mathcal{M}' \models \varphi$ and $\mathcal{M}'' \models \varphi$ then $\mathcal{M} \models \forall x\ \varphi$

In the following, we assume closed QBFs, which are in *prenex normal form* (pnf), i.e., they are of the form $Y_1 X_1\ Y_2 X_2\ \cdots\ Y_n X_n\ \varphi$ where $\varphi \in Q, Y \in \{\exists, \forall\}, X_i \subseteq$ variables, there is no occurrence of Y in φ and every variable of φ occurs in some variable set X_i. Note that closed QBFs can only be valid or invalid.

Proposition 2.1. *Assume a closed QBF ψ of the form $Y_1 X_1\ Y_2 X_2\ \cdots\ Y_n X_n\ \varphi$ (pnf) and some odd i with $1 \leq i \leq n-1$. If $Y_i = \exists, Y_{i+1} = \forall$, then the question whether ψ is valid (QSAT$_n$) is Σ_n^{P}-complete. If $Y_i = \forall, Y_{i+1} = \exists$, then QSAT$_n$ is Π_n^{P}-complete.*

2.5 Abstract Argumentation

Argumentation frameworks have gained increasing popularity in the world of artificial intelligence in recent years and therefore they and their connections to other related topics are currently active research. In this thesis, the Dung Argumentation Framework [Dun95] is considered, which is defined as follows.

Definition 2.8. *A Dung argumentation framework is a tuple (A, R), where A is a set of arguments and $R \subseteq A \times A$ models the set of attacks.*

Here we are not interested in instantiation of argumentation frameworks such that we get a specific resulting framework. Hence, the graph instances are considered to be given and the goal is to select sets $S \subseteq A$, which meet certain properties according to the desired semantics. In the following section, a short review of the semantics that are used in Chapter 4.1.3, namely *admissible*, *complete*, *preferred*, *semi-stable* and *stable*, is provided. For this, we assume an argumentation framework $F = (A, R)$ given as shown in Figure 2.4. For SEM $\in$ {conflict-free, admissible, complete, preferred, semi-stable}, the set SEM(F) denotes the set of SEM-extensions in F. Next, we need some basic definitions concerning defended and conflict-free sets.

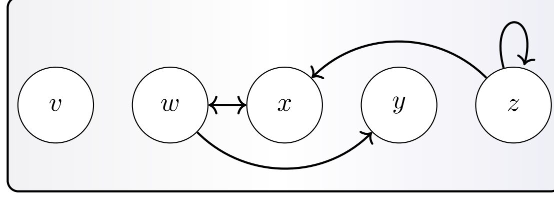

Figure 2.4: Argumentation framework $F_{Ex} = (A_{F_{Ex}}, R_{F_{Ex}})$ with $A_{F_{Ex}} = \{v, w, x, y, z\}$ and $R_{F_{Ex}} = \{(w, x), (x, w), (w, y), (z, z), (z, x)\}$.

Definition 2.9. *Given an argumentation framework $F = (A, R)$. Any $s \in S$, such that $S \subseteq A$ is called* defended by S in F *if for every $(s', s) \in R$, there exists $s'' \in S$ such that $(s'', s') \in R$. The set $def_F(S)$ denotes $\bigcup_{s \in A:\ s \text{ is defended by } S \text{ in } F} s$, any $S' \subseteq def_F(S)$ is called* defended by S in F. *Any $S \subseteq A$ without existing $s, s' \in S$ such that $(s, s') \in R$ is called* conflict-free in F.

Example 2.10. *Consider the set conflict-free(F_{Ex}) w.r.t. F_{Ex} of Figure 2.4. We observe that $\emptyset \in$ conflict-free(F_{Ex}). Clearly, for every $a \in A_{F_{Ex}}$ s.t. $a \neq z$ it holds that $\{a\} \in$ conflict-free(F_{Ex}); since v is isolated, also $\{v, a\} \in$ conflict-free(F_{Ex}) for every $a \in A_{F_{Ex}}$ with $a \neq z$. Argument z is not contained in any $S \in$ conflict-free(F_{Ex}), since it attacks itself. Finally, $\{x, y\} \in$ conflict-free(F_{Ex}) and of course, $\{v, x, y\} \in$ conflict-free(F_{Ex}). We have conflict-free(F_{Ex}) $= \{\emptyset, \{v\}, \{w\}, \{x\}, \{y\}, \{v, w\}, \{v, x\}, \{v, y\}, \{x, y\}, \{v, x, y\}\}$.*

This basic notion leads to further definitions of semantics in the following.

Definition 2.11. *Given an argumentation framework $F = (A, R)$. Any $S \subseteq A$ which fulfills the following conditions is called* admissible*: 1.) S is* conflict-free in F. *2.) every $s \in S$ is* defended by S in F. *Any $S \subseteq A$ which fulfills the following conditions is called* stable*: 1.) S is* conflict-free in F. *2.) every $s \in A \setminus S$ is* attacked *by any $s' \in S$.*

Example 2.12. *Consider the set admissible(F_{Ex}) (w.r.t. F_{Ex} of Figure 2.4). By Definition 2.11, we only have to consider elements of conflict-free(F_{Ex}). The argument x can never be part of any admissible extension, therefore also y is not in any extension. The remaining sets of conflict-free(F_{Ex}) are admissible; thus, we have that admissible(F_{Ex}) $= \{\emptyset, \{v\}, \{w\}, \{v, w\}\}$. We consider now the set stable(F_{Ex}) w.r.t. framework F_{Ex} of Figure 2.4. By Definition 2.11, we only have to consider elements of conflict-free(F_{Ex}). Since z is not contained in any extension $S \in$ conflict-free(F_{Ex}) and it is not attacked by any $a \in S$ (z only attacks itself), there cannot be any stable extension. Thus, stable(F_{Ex}) $= \emptyset$.*

Using the notion of admissible extensions, one can define complete sets.

Definition 2.13. *Given an argumentation framework $F = (A, R)$. Any $S \subseteq A$ which fulfills the following conditions is called* complete in F*: 1.) S is* admissible in F. *2.) $def_F(S) = S$.*

Example 2.14. *We consider the set complete*(F_{Ex}) *of framework* F_{Ex} *of Figure 2.4. By Definition 2.13, we only have to consider elements of admissible*(F_{Ex}). *The set* $\emptyset$ *is not complete since* $def_{F_{Ex}}(\emptyset) = \{v\}$, $\{w\} \notin$ *complete*($\{w\}$), *since* $def_{F_{Ex}}(\{w\}) = \{v, w\}$. *The remaining sets of admissible*(F_{Ex}) *are complete, i.e. complete*(F_{Ex}) $= \{\{v\}, \{v, w\}\}$.

One can also define extensions based on some optimization criteria; the following two definitions maximize the elements in an extension resp. the range of an extension.

Definition 2.15. *Given an argumentation framework* $F = (A, R)$. *Any* $S \subseteq A$ *which fulfills the following conditions is called* preferred in F: *1.)* S *is* admissible in F. *2.) every* $S' \supset S$ *is not* admissible in F. *Any* $S \subseteq A$ *s.t.* S *is* admissible in F *and every* $S' \subseteq A$ *s.t.* S' *is* admissible in F *with* $S^+_R \not\subset S'^+_R$ *is called* semi-stable in F *where for* $S \subset A$, $S^+_R := S \cup \{a \mid \exists b \in S \text{ s.t. } (b, a) \in R\}$ *(called the* range of S in F*)*.

Example 2.16. *We consider the set preferred*(F_{Ex}) *of framework* F_{Ex} *of Figure 2.4. By Definition 2.15, we only have to consider subset-maximal elements of admissible*(F_{Ex}), *i.e. preferred*(F_{Ex}) $= \{\{v, w\}\}$. *Now we want to determine semi-stable*(F_{Ex}) *w.r.t.* F_{Ex} *of Figure 2.4. By Definition 2.15, we only have to consider range-maximal elements of semi-stable*(F_{Ex}), *i.e. semi-stable*(F_{Ex}) $= \{\{v, w\}\}$.

Relation between semantics Recall that the set SEM(F) denotes the set of SEM-extensions in F for SEM $\in$ {conflict-free, admissible, complete, preferred, semi-stable}. Given this – combined with the definitions above – for every framework $F = (A, R)$ the following holds: stable(F) $\subseteq$ semi-stable(F) $\subseteq$ preferred(F) $\subseteq$ complete(F) $\subseteq$ admissible(F) $\subseteq$ conflict-free(F).

One can also provide an alternative to Definition 2.15 by using complete semantics as basis in order to obtain a different, but equivalent (can be shown easily by the subset-inclusions) characterization.

Definition 2.17. *Given an argumentation framework* $F = (A, R)$. *Any* $S \subseteq A$ *which fulfills the following conditions is called* preferred$_{\text{complete}}$ in F: *1.)* S *is* complete in F. *2.) every* $S' \supset S$ *is not* complete in F. *Any* $S \subseteq A$ *s.t.* S *is* complete in F *and every* $S' \subset S$ *s.t.* S' *is* complete in F *with* $S^+_R \not\subset S'^+_R$ *is called* semi-stable$_{\text{complete}}$ in F *where for* $S \subset A$, S^+_R *is called the* range of S in F *(as defined above). Also for SEM* $\in$ *{preferred$_{complete}$, semi-stable$_{complete}$}, the set SEM*(F) *denotes the set of SEM-extensions in* F.

Example 2.18. *Note that semi-stable$_{complete}$*(F_{Ex}) *= semi-stable*(F_{Ex}) *and furthermore preferred$_{complete}$*(F_{Ex}) *= preferred*(F_{Ex}) *since* $\{v, w\}$ *is already a complete extension.*

Concerning notation, note that in this thesis – if the framework F is clear from the context – we will omit framework F, for instance instead of writing that S is admissible in F, we will denote for short that S is admissible.

Computational Complexity Remember the short introduction concerning computational complexity of Section 2.4. In Table 2.1, we show complexity results [DB02,

Semantics	Credulous Acceptance	Skeptical Acceptance
stable	NP-complete	co-NP-complete
admissible	NP-complete	trivial
complete	NP-complete	P-complete
preferred	NP-complete	Π_2^P-complete
semi-stable	Σ_2^P-complete	Π_2^P-complete

Table 2.1: Complexity results for Abstract Argumentation.

DW10, Dvo12] for the presented semantics. We define the terms *credulous* and *skeptical* acceptance as follows.

Definition 2.19. *Assume AF $F = (A, R)$ and $a \in A$; $Cred_\sigma$ is the problem whether a is contained in at least one σ-extension of F.*

Definition 2.20. *Assume AF $F = (A, R)$ and $a \in A$; $Skept_\sigma$ is the problem whether a is contained in every σ-extension of F.*

It was shown in [DSW12] by reduction to MSO using Courcelle's meta-theorem [Cou90] that the discussed semantics are FPT (actually Fixed-Parameter Linear) w.r.t. k-bounded tree-width for both credulous acceptance and skeptical acceptance. Actually in [DSW12] FPT results are shown for several other semantics and different reasoning modes apart from credulous and sceptical reasoning.

2.6 Answer-Set Programming

Answer-Set Programming[2], which is often also referred to as logic programming under the stable-model semantics, deals with finding the so-called Answer-Sets of a given set of rules. The following brief definition of logic programs and Answer-Sets is based on [BET11, BMW12].

Definition 2.21. *A disjunctive logic program Π is a finite set of rules of the form $a_1 \mid \cdots \mid a_k \leftarrow b_1, \cdots, b_m, not\ b_{m+1}, ..., not\ b_n$, where $a_1, \cdots, a_k, b_1, \cdots b_n$ are propositional atoms. For an arbitrary rule $r \in \Pi$, the following sets of atoms are defined: Head $h(r) = \{a_1, ..., a_k\}$; Positive body $b^+(r) = \{b_1, ..., b_m\}$; Negative body $b^-(r) = \{b_{m+1}, ..., b_n\}$; Body $b(r) = b^+(r) \cup b^-(r)$. If for a given rule r, $h(r) = \emptyset$, r is called constraint, whereas if $b(r) = \emptyset$, r is called a fact and $\leftarrow$ can be left out.*

Definition 2.22. *A rule $r \in \Pi$ is satisfied by an interpretation I, which is a set of atoms, if $I \cap h(r) \neq \emptyset$ or $b^-(r) \cap I \neq \emptyset$ or $b^+(r) \setminus I \neq \emptyset$. An interpretation I is an Answer-Set of a program Π (denoted by $I \in \mathtt{Answers}(\Pi)$) if it satisfies every rule r of*

[2]Answer-Set Programming is often abbreviated to ASP; its connection to classical logic can be found in [LPV01]

the Gelfond-Lifschitz reduct $\Pi^I = \{h(r) \leftarrow b^+(r) \mid r \in \Pi, b^-(r) \cap I = \emptyset\}$ *and every proper subset of* I *does not satisfy* Π^I*.*

Typically, programs are designed according to the Guess & Check [EP06] paradigm. This is done by generating (guessing) potential solution candidates and then eliminating invalid ones by checking certain conditions.

The Gringo system [GKK+11] supports more advanced expressions and classical negation, but all of its mechanisms have in common an ability to be reduced to logic programs as defined above. Gringo's atoms are predicates, whose arguments are variables or ground terms. These programs are abbreviations for variable-free logic programs, where the variables are replaced by ground terms during the process of *grounding*.

Gringo allows the appearance of cardinality constraints of the form $l\{L_1, ..., L_n\}u$, which are satisfied by an interpretation I if at least l and at most u of the literals $L_1, ..., L_n$ are true in I. A literal is either an atom or a negated atom (using *not*). The default value of l is 0 and of u it is infinity, i.e., every subset of the specified set is valid. More details of Gringo and its syntactic sugar can be found in [GKK+11].

Besides Circumscription, the paper [GPW10a] presents FPT (actually FPL) results for disjunctive ASP by reduction to MSO and applying Courcelle's meta-theorem [Cou90].

Enhancing ASP Besides several extensions of disjunctive ASP – including weak constraints, disjunction in the body of a rule or adding quantifiers – there are tricks to unleash the full expressive power, but not necessarily without paying the price of losing simplicity.

The saturation technique (as in [EG95, EGM97, LRS00]) is a common practice to encode problems in disjunctive ASP and unleash its full expressive power (Σ_2^P, see Section 2.4) of it. This technique requires restricted usage of default negation and disjunction; an introduction can be found in [ABC+15]. Saturation complements the Guess & Check [EP06] paradigm, where usage of default negation for the guess typically gets replaced by disjunctive heads. This enables the possibility for an Answer-Set to contain all the atoms that are subject to the guess.

The pattern often works like this and uses disjunction in guesses: First of all, we guess a solution candidate S for which we want to know if there is no possibility for S being no solution. To do so, we also guess a potential counter candidate C for invalidating S being a solution. If C is not a counter candidate, we derive a designated atom A. This atom A then causes the atoms subject to the second guess being set to true. Thus, all models not leading to a counter candidate are saturated (hence "saturation") with all the atoms of the disjunction (of the second guess). If on the other hand C is a counter candidate, S is discarded by a constraint (involving default negated A). If S is in fact an invalid solution candidate, i.e., there exists some counter candidate C' ($C \neq C'$), it is discarded by the minimal model semantics, because each model not encoding such a C' is saturated and is therefore not a minimal model of its reduct. In fact, the Answer-Sets just encode solutions S without existence of any counter candidate C'.

Applying Saturation In the following, we present Example 2.23, which uses disjunctive ASP (see the two definitions at the beginning of this chapter); due to the structure of the

Quantified Boolean Formula (QBF) ψ of this example, it requires Σ_2^{P} (see Section 2.4) computational power. In order to solve problems, which require Σ_2^{P} computational power, with ASP, typically the saturation technique is applied [EG95, EGM97, LRS00, ABC+15].

Example 2.23. *Assume the following simple QBF* $\psi := \exists a \forall b(\neg a \vee b)$*. In order to solve this problem, we can use the propositional disjunctive program* Π_{ex} *given in Listing 2.1. It uses the saturation technique and atoms* tV *resp.* fV *to model that variable* V *of* ψ *(*$V \in \{a, b\}$*) is set to true resp. false. Observe that* ψ *is evaluated to true, the witnesses are* M_1, M_2 *with* $M_1(a) = M_2(a) =$ *false and* $M_1(b) =$ *true and* $M_2(b) =$ *false. Assume now* Π_{ex} *and a model* M *with a guessed to false, i.e.* $M(fa) =$ *true and* $M(ta) =$ *false. Clearly, due to Lines 3, 6 and 7,* $M(sat) =$ *true,* $M(tb) =$ *true and* $M(fb) =$ *true. Observe now* Π_{ex}^M*, which is the same as* Π_{ex} *but without the rule in Line 10. Assume that there is* $M' \subseteq M$ *with* $M' \models \Pi_{\mathsf{ex}}^M$*. Note that* $M'(ta)$ *cannot be true, since we require* $M' \subseteq M$*. By Line 13 we get that* $M'(fa) =$ *true. But then* $M'(sat) =$ *true,* $M'(tb) =$ *true and* $M'(fb) =$ *true has to hold as well; so* $M' = M$*. Therefore* M *is an Answer-Set of* Π_{ex}*. We observe that there cannot exist any model* M'' *of* Π_{ex} *that is exactly as* M*, but with* $M''(fa) =$ *true instead of* $M''(fa) =$ *false (since* $M'' \supset M$*). To conclude that* M *is the only Answer-Set of* Π_{ex}*, assume a model* M^* *of* Π *with* $M^*(ta) =$ *true. Due to Line 10, we are forced to set* $M^*(tb) =$ *true – leading to* $M^*(sat) =$ *true – because otherwise* M^* *would not be a model of* Π_{ex}*. Actually, we observe that there exists model* $N^* \subset M^*$ *(with* $N^*(fb) =$ *true,* $N^*(tb) =$ *false,* $N^*(fa) =$ *false,* $N^*(ta) =$ *true and* $N^*(sat) =$ *false) of* $\Pi_{\mathsf{ex}}^{M^*}$ *invalidating* M^* *as an Answer-Set of* Π_{ex}*.*

```
1  % Model the cases where ψ evaluates to true
2  sat ← tb.
3  sat ← fa.

5  % Saturize over the ∀−quantified variables, if ψ evaluates to true
6  tb ← sat.
7  fb ← sat.

9  % Ensure satisfiability
10 ← not sat.

12 % Guess truth values of variables a and b
13 ta v fa.
14 tb v fb.
```

Listing 2.1: Π_{ex}: ASP program for solving QBF ψ.

Metasp [GKS11] was designed to simplify several ASP encodings by extending disjunctive ASP via several statements designed for enabling optimization without explicitly using the (tedious) saturation technique as explained in the previous subsection. Instead, it provides a way to perform complex optimizations with little additional effort.

CHAPTER 3

DP algorithms on TDs for Abstract Argumentation

This chapter first gives an introduction on DP algorithms on TDs for Abstract Argumentation by discussing and proving a DP algorithm on TDs for admissible semantics similar to [DPW12] (see Section 3.1). We basically modified the given algorithm [DPW12] for admissible semantics in order to be able to further adapt it to the new semi-stable algorithm of Section 3.2. Moreover, we will briefly show how to further adapt the (more advanced) proof of the DP algorithm on TDs for semi-stable semantics, in order to reach a DP algorithm on TDs for preferred semantics in Section 3.3. Note that this chapter will consider normalized TDs as defined in Chapter 2.

3.1 Modified algorithm for Admissible Semantics

First of all, we need some basic definitions for induced subframeworks.

Definition 3.1. *For a Tree Decomposition $(\mathcal{T}, \mathcal{X})$ of an AF F and $t \in \mathcal{T}$, let $X_{\geq t}$ be the union of all bags $X_s \in \mathcal{X}$ such that s occurs in the subtree of $\mathcal{T}$ rooted at t. Moreover, $X_{>t}$ denotes $X_{\geq t} \backslash X_t$. We also use the following terminology:*

- *$F_t = F|_{X_t}$ is the subframework in t;*
- *$F_{\geq t} = F|_{X_{\geq t}}$ is the subframework induced by (the subtree rooted at) t.*

Note that the subframework induced by the root of such a decomposition of an AF F is F itself.

From now on we restrict ourselves to normalized Tree Decompositions where the bag of the root is empty. Unless stated otherwise, we thus assume below that $(\mathcal{T}, \mathcal{X})$ always

denotes a normalized Tree Decomposition (with empty root bag) for some given AF F. Note that TDs for directed graphs (and therefore also for AFs F) are defined without concerning about directions, i.e. we take the shadow of the attack graph of F (see Section 2).

Definition 3.2. *Let $F = (A, R)$ be an AF and B a set of arguments. A tuple (S, D) s.t. $S, D \subseteq A$ and $S \cap D = \emptyset$ is a B-restricted admissible tuple for F, if*

1. *S is conflict-free in F and S defends itself in F against all $a \in A \cap B$ and*
2. *For each $a \in A$: $((S \rightarrowtail_{attacks} a$ or $a \rightarrowtail_{attacks} S) \implies a \in D)$, i.e. if $a \in A$ is already or still requires to be defeated (by S), $a \in D$ is ensured.*

S is called a B-restricted admissible set for F if (S, D) is a B-restricted admissible tuple for F.

Concerning notation we will denote both (S, D) to be B-restricted admissible and S to be B-restricted admissible if the context makes it clear that (S, D) is a tuple and S is a set.

Note that for $A \subseteq B$, B-restricted admissible sets of AF (A, R) are just admissible sets for F. For $A \cap B = \emptyset$, B-restricted admissible sets are just the conflict-free sets for F.

Example 3.3. *Let us again consider the example framework F_{Ex} given in Figure 2.4 (see Chapter 2). Figure 3.1 shows both F_{Ex} and the decomposition of this framework and also includes induced subframeworks (by also including the dashed arguments within a node of a Tree Decomposition).*

Consider the AF $F = (A = \{w, x, y\}, R = \{(w, x), (x, w), (w, y)\})$, which is a subframework induced by node n_3 of the TD $\mathcal{T}_{Ex}$ (see Figure 3.1). The $\{x, y\}$-restricted admissible sets are $\emptyset$, $\{w\}$, $\{x\}$, $\{y\}$ and $\{x, y\}$. The set $\{y\}$ however is not $\{w\}$-restricted admissible, since $w \rightarrowtail_{attacks} y$ and y does not defend itself against w. We turn the stated $\{x, y\}$-restricted admissible sets into $\{x, y\}$-restricted admissible tuples by adding a second component, a superset of defeated (including arguments requiring defeating) arguments w.r.t. F. Since A always satisfies condition (2) of Definition 3.2, in the following we only state the smallest set, which satisfies condition (2). That is, the $\{x, y\}$-restricted admissible tuples with smallest (w.r.t. the order induced by $\subseteq$) second component are $(\emptyset, \emptyset)$, $(\{w\}, \{x, y\})$, $(\{x\}, \{w\})$, $(\{y\}, \{w\})$ and $(\{x, y\}, \{w\})$. Of course, we can always extend the second component up to A, therefore the remaining $\{x, y\}$-restricted admissible tuples are $(\emptyset, \{w\})$, $(\emptyset, \{x\})$, $(\emptyset, \{y\})$, $(\emptyset, \{w, x\})$, $(\emptyset, \{w, y\})$, $(\emptyset, \{x, y\})$, $(\emptyset, \{w, x, y\})$, $(\{w\}, \{w, x, y\})$, $(\{x\}, \{w, x\})$, $(\{x\}, \{w, y\})$, $(\{x\}, \{w, x, y\})$, $(\{y\}, \{w, x\})$, $(\{y\}, \{w, y\})$, $(\{y\}, \{w, x, y\})$, $(\{x, y\}, \{w, x\})$, $(\{x, y\}, \{w, y\})$, $(\{x, y\}, \{w, x, y\})$.

We require basic definitions about the semantical concept of valid colorings and the syntactical vcolorings with its allowed operations in order to prove the correctness of the algorithm for admissible semantics in the following (by showing equivalence between valid colorings and vcolorings).

Definition 3.4. *Let $F = (A, R)$ be an AF $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition and $t \in \mathcal{T}$. We call $C : X_t \rightarrow \{in, attc, def, out\}$ a coloring and define for such a coloring C, $[C] = \{a \mid C(a) = in\}$ and $[[C]] = \{a \mid C(a) = def \text{ or } C(a) = attc\}$.*

Definition 3.5. *Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of an AF F and $t \in \mathcal{T}$. Given a coloring C for t, we define $e_t(C)$ as the collection of $X_{>t}$-restricted admissible tuples (S, D) for $F_{\geq t}$ which satisfy the following for each $a \in X_t$.*

(i) $C(a) = in \iff a \in S$
(ii) $C(a) = def \iff S \rightarrowtail_{attacks} a$
(iii) $C(a) = attc \iff S \not\rightarrowtail_{attacks} a$ and $a \rightarrowtail_{attacks} S$
(iv) $C(a) = out \implies S \not\rightarrowtail_{attacks} a$ and $a \not\rightarrowtail_{attacks} S$
(v) $C(a) \in \{def, attc\} \iff a \in D$

If $e_t(C) \neq \emptyset$, C is called a valid coloring for t; $\mathcal{C}_t$ denotes the set of valid colorings for t.

Definition 3.6. *Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of an AF F and $t \in \mathcal{T}$. Given a coloring C for t, we define $e'_t(C)$ for convenience as follows: $e'_t(C) := \{S \mid (S, D) \in e_t(C)\}$.*

Lemma 3.1. *Assuming that r is the root of a Tree Decomposition of an AF F and ϵ the (empty) coloring for r, we have that $e'_r(\epsilon) = admissible(F)$.*

Proof. Note that for $A \subseteq B$, B-restricted admissible sets of AF (A, R) are just admissible sets for F, therefore $e'_r(\epsilon) = \text{admissible}(F)$ follows immediately from Definitions 3.2 and 3.5, assuming r is the root of the Tree Decomposition and ϵ the coloring for r. □

The key observation that $e'_r(\epsilon) = \text{admissible}(F)$ is crucial and justifies why it is required to prove equivalence between colorings and vcolorings. Therefore, it suffices to show equivalence between the concept of valid colorings and the vcolorings (in other words the operations of our algorithm for admissible semantics). For this we need Definition 3.5 and require knowledge about allowed syntactical operations concerning vcolorings (see the forthcoming definition).

Example 3.7. *Consider the node $t = n_3$ of our Tree Decomposition $X_t = \{w, x\}$ (see Figure 3.1) and the coloring C with $C(w) = in$, $C(x) = def$. It holds that $F_{\geq t} = (\{w, x, y\}, \{(w, x), (x, w), (w, y))\}$ and $X_{>t} = \{y\}$. The only tuple in $e_t(C)$ which is $X_{>t}$-restricted admissible for $F_{\geq t}$ and satisfies the conditions from Definition 3.5 is $(\{w\}, \{x, y\})$, i.e. $e_t(C) = \{(\{w\}, \{x, y\})\}$.*

The operations of the forthcoming Definition 3.8 intuitively correspond to allowed operations of our algorithm for admissible semantics. The $-$-operation is for any FORGET node, the different $+$-operations are the potential possibilities for any INSERT node, whereas the $\bowtie$-operator is allowed for JOIN nodes. The diverse restrictions of using vcolorings are then formalized in Definition 3.9.

Intuitively, there are three possibilities (and this is in fact the crucial observation, which is required for computing semi-stable extensions as we will see in the next section) for the +-operations. $C +_{\text{attc}} a$ guesses the newly introduced atom a to be an attacking candidate (attc), i.e. a requires defeating (def) by the resulting extension, let us call it S, that is $S \rightarrowtail_{\text{attacks}} a$. $C \dot{+}_{\text{in}} a$ assumes the new atom a to be in the resulting extension, and finally $C \hat{+}_{\text{out}} a$ results in the assumption that a is neither in the extension, nor is it an attacking candidate of it.

Definition 3.8. *Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of an AF F. Moreover, assume C resp. D to be a coloring for node t' resp. $t'' \in \mathcal{T}$. We define the following operations.*

$(C - a)(b) = C(b)$ *for each* $b \in X_{t'} \setminus \{a\}$.

$$(C +_{attc} a)(b) = \begin{cases} C(b) & \text{if } b \in X_{t'}, \\ def & \text{if } a = b \text{ and } [C] \rightarrowtail_{attacks} a, \\ attc & \text{otherwise}, \end{cases}$$

$$(C \hat{+}_{out} a)(b) = \begin{cases} C(b) & \text{if } b \in X_{t'}, \\ out & \text{otherwise}, \end{cases}$$

$$(C \dot{+}_{in} a)(b) = \begin{cases} in & \text{if } a = b \text{ or } C(b) = in, \\ out & \text{if } a \neq b, (a,b) \notin F, (b,a) \notin F, C(b) = out, \\ def & \text{if } a \neq b \text{ and } ((C(b) = attc \text{ and } (a,b) \in F) \text{ or } C(b) = def), \\ attc & \text{otherwise}, \end{cases}$$

$$(C \bowtie D)(b) = \begin{cases} in & \text{if } C(b) = D(b) = in, \\ out & \text{if } C(b) = D(b) = out, \\ def & \text{if } C(b) = def \text{ or } D(b) = def, \\ attc & \text{otherwise}, \end{cases}$$

Definition 3.9. *Let $t \in \mathcal{T}$ be a node in a normalized Tree Decomposition $(\mathcal{T}, \mathcal{X})$ of an AF F and t', t'' be the possible children of t. The operations are taken as defined in Definition 3.8. A vcoloring is defined as following.*

- *FORGET: If C is a vcoloring for t', $X_t = X_{t'} \setminus \{a\}$ and $C(a) \neq attc$ then $C - a$ is a vcoloring for t.*
- *INSERT:*
  - *(i) $C +_{attc} a$ is a vcoloring for t if C is a vcoloring for t' and $X_t = X_{t'} \cup \{a\}$;*
  - *(ii) $C \hat{+}_{out} a$ is a vcoloring for t if additionally to (i), $[C] \not\rightarrowtail_{attacks} a$ and $a \not\rightarrowtail_{attacks} [C]$;*
  - *(iii) $C \dot{+}_{in} a$ is a vcoloring for t if additionally to (i), $a \not\rightarrowtail_{attacks} a, [C] \not\rightarrowtail_{attacks} a$, $a \not\rightarrowtail_{attacks} [C]$ and $[[C]] = [[C \dot{+}_{in} a]]$*
- *JOIN: If C is a vcoloring for t', D is a vcoloring for t'', $[C] = [D]$, and $[[C]] = [[D]]$, then $C \bowtie D$ is a vcoloring for t.*

- *LEAF: Each coloring* $X_t \to \{in, out, def, attc\}$ *s.t.*
  $C(x) = in \Rightarrow C(y) \in \{def, attc\}$ *for all* $y \rightarrowtail_{attacks} x$
  $C(x) = def \iff \exists y : C(y) = in$ *and* $y \rightarrowtail_{attacks} x$
  holds for all $x \in X_t$*, is a vcoloring.*

Example 3.10. *Figure 3.2 contains the bottom-up computation of the vcolorings for* F_{Ex} *of Figure 3.1 w.r.t. the Tree Decomposition shown in Figure 3.1. The representation is a little bit different (more compact and in table form, cf. Section 4.1.2.1; each row represents one vcoloring) than for instance in Figure 2.2. In the forthcoming examples, we will discuss some transitions from children to parent nodes (as defined in Definition 3.9).*

The last column of the computed table of every Tree Decomposition node contains the extension pointers as described in Chapter 2. Observe that by following these extension pointers, we get that in total admissible$(A_{F_{Ex}}) = \{\emptyset, \{v\}, \{w\}, \{v, w\}\}$ *(compare to Example 2.12).*

Now we show for every node n (i.e. all the different node types) equivalence between vcolorings and colorings assuming equivalence between child nodes of n. This helps us later to reach our goal of showing correctness of the algorithm for admissible semantics (specified by Definitions 3.8 and 3.9) by structural induction.

Lemma 3.2. *For any LEAF node in a Tree Decomposition of an AF* F*, valid colorings and vcolorings coincide.*

Proof. Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of F and t a leaf in $\mathcal{T}$. We have $X_{>t} = \emptyset$; therefore, the $X_{>t}$-restricted admissible sets for $F_{\geq t}$ coincide with the conflict-free sets. First, let C be a vcoloring for t. We have to show that then C is a valid coloring for t. Suppose to the contrary that it is not, i.e., either $[C]$ is not conflict-free in $F_t = F_{\geq t}$ or C violates one of the conditions in Definition 3.5. It is easy to check that, in both cases, one of the conditions for C being a vcoloring is violated. For instance, if there is a conflict in $[C]$, then there exist arguments $x, y \in X_t$ with $X \rightarrowtail_{\text{attacks}} y$ and $C(x) = C(y) = \text{in}$. Hence, the first condition in Definition 3.9 for vcolorings at a LEAF node is violated, a contradiction.
Now suppose that C is a valid coloring for t, i.e., C satisfies the conditions of colorings (see Definition 3.5) and $[C]$ is conflict-free in $F_{\geq t}$. Then C satisfies the condition of a vcoloring for a LEAF node according to Definition 3.9. For instance, let $x, y \in X_t$ with $C(x) = \text{in}$ and $y \rightarrowtail_{\text{attacks}} x$. Then, since C is a valid coloring, either case (ii) or case (iii) of Definition 3.5 applies and, thus, $C(y) \in \{\text{attc}, \text{def}\}$ holds. □

Example 3.11. *Consider for instance node* $t = n_1$ *of Figure 3.1 with bag* $\{w, y\}$*. We have six different vcolorings (see Figure 3.2), which correspond to conflict-free sets (actually* $\emptyset$*-restricted admissible tuples) for* $F_{\geq t} = (\{w, y\}, \{(w, y)\})$*:* $(\emptyset, \emptyset)$*,* $(\emptyset, \{w\})$*,* $(\emptyset, \{y\})$*,* $(\emptyset, \{w, y\}$*,* $(\{w\}, \{y\})$*,* $(\{y\}, \{w\})$*.*

For proving equivalence for FORGET nodes, we need an additional lemma as follows.

Lemma 3.3. *For any FORGET node t in a Tree Decomposition of an AF F with child node t' such that $X_t = X_{t'} \backslash \{a\}$, and every $S \subseteq A$, the following relationships hold.*

1. *If (S, D) is an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$, then (S, D) is also an $X_{>t'}$-restricted admissible tuple for $F_{\geq t'}$.*
2. *If (S, D) is an $X_{>t'}$-restricted admissible tuple for $F_{\geq t'}$ and $a \in S$, then (S, D) is an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$.*
3. *If (S, D) is an $X_{>t'}$-restricted admissible tuple for $F_{\geq t'}$, $a \notin S$ and S defends itself against a (including the case that a does not attack S at all), then (S, D) is an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$.*

Proof. First, since $X_t \subseteq X_{t'}$, we have $F_{\geq t} = F_{\geq t'}$ and $X_{>t} \supseteq X_{>t'}$. Let (S, D) be an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$ and hence for $F_{\geq t'}$, i.e., S is conflict-free in F, S defends itself against all $b \in X_{>t}$ and D contains the arguments requiring defeating s.t. $S \cap D = \emptyset$. By $X_{>t} \supseteq X_{>t'}$, S thus also defends itself against all $b \in X_{>t'}$. Hence (S, D) is an $X_{>t'}$-restricted admissible tuple for $F_{\geq t'}$, D still contains remaining arguments requiring defeating and assertion (1) follows.
Now assume that (S, D) is an $X_{>t'}$-restricted admissible tuple for $F_{\geq t'}$ and $a \in S$. Then S is conflict-free in F, S defends itself against all $b \in X_{>t'}$ and D contains the arguments requiring defeating s.t. $S \cap D = \emptyset$. Strictly speaking, S defends itself against all $b \in X_{>t'} \backslash S$. By $X_{>t'} \backslash S = X_{>t} \backslash S$ (recall we are assuming that $a \in S$), therefore S is an $X_{>t}$-restricted admissible set for $F_{\geq t}$. Since $a \in S$, (S, D) is an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$. This proves assertion (2).
Finally assume that (S, D) is an $X_{>t'}$-restricted admissible tuple for $F_{\geq t'}$, $a \notin S$ and S defends itself against a (i.e. $a \in D$). Since S is $X_{>t'}$-restricted admissible, it is conflict-free in F and defends itself against all $b \in X_{>t'}$. Moreover, $X_{>t} = X_{>t'} \cup \{a\}$ and, by assumption, S defends itself against a. Hence, S defends itself against all $b \in X_{>t}$. Thus, (S, D) is $X_{>t}$-restricted admissible and assertion (3) follows. □

Lemma 3.4. *For any FORGET node t in a Tree Decomposition of an AF F, valid colorings and vcolorings coincide, if they coincide in the child node t' of t.*

Proof. We assume that valid colorings and vcolorings coincide at t'. Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of $F = (A, R)$, t a FORGET node in $\mathcal{T}$, and t' the child node of t. By definition, $X_t = X_{t'} \backslash \{a\}$ for some $a \in A$. Moreover, we get $X_{\geq t} = X_{\geq t'}$ and $X_{>t} = X_{>t'} \cup \{a\}$.
Let C be a valid coloring for t. We show that there exists a valid coloring C' for t' with $C'(a) \neq \text{attc}$ and $C = C' - a$. We define C' as follows. For all $b \in X_t = X_{t'} \backslash \{a\}$, we set $C'(b) = C(b)$. Hence, no matter which value $\{\text{in}, \text{def}, \text{out}\}$ we assign to $C'(a)$, we have $C = C' - a$. In order to define $C'(a)$, we consider an arbitrary $(S, D) \in e_t(C)$ and distinguish two cases:

1. If $a \in S$, then we set $C'(a) = \text{in}$. Since (S, D) is $X_{>t}$-restricted admissible for $F_{\geq t}$, it is also $X_{>t'}$-restricted admissible for $F_{\geq t'} = F_{\geq t}$, by Lemma 3.3. Moreover,

$(S, D) \in e_{t'}(C')$, i.e., C' is a valid coloring for t' (this can be seen by just using the chosen S in the conditions in Definition 3.5). Hence, by assumption, C' is a vcoloring for t' and, therefore, also $C = C' - a$ is a vcoloring for t, by definition.

2. Now let $a \notin S$. If $S \rightarrowtail_{\text{attacks}} a$, we set $C'(a) = \text{def}$. If $S \not\rightarrowtail_{\text{attacks}} a$ and $a \not\rightarrowtail_{\text{attacks}} S$, we set $C'(a) = \text{out}$. In both cases, $(S, D) \in e_{t'}(C')$. Note that the case $S \not\rightarrowtail_{\text{attacks}} a$ and $a \rightarrowtail_{\text{attacks}} S$ cannot occur since, by assumption, S is $X_{>t}$-restricted admissible for $F_{\geq t}$. By the same reasoning as above, C' is a vcoloring for t'; thus C is also a vcoloring for t.

Now let C be a vcoloring for t, i.e., there exists a vcoloring C' for t' such that $C'(a) \neq \text{attc}$ and $C = C' - a$. By assumption, C' is a valid coloring for t'. Hence, there exists $(S, D) \in e_{t'}(C')$, i.e., (S, D) is $X_{>t'}$-restricted admissible for $F_{\geq t'} = F_{\geq t}$. Since $C'(a) \neq \text{attc}$, it cannot happen that both $a \rightarrowtail_{\text{attacks}} S$ and $S \not\rightarrowtail_{\text{attacks}} a$ hold. But then (S, D) is also $X_{>t}$-restricted admissible for $F_{\geq t}$ by Lemma 3.3 and $(S, D) \in e_t(C)$. Thus, $C \in \mathcal{C}_t$. □

Example 3.12. *Our running example (Figure 3.2) proceeds with node $t = n_1$ of Figure 3.1. The next node n_2 above removes argument y and thus is of type FORGET; $X_{>n_2} = \{y\}$. Vcolorings for n_2 are obtained from vcolorings for n_1 except for C with $C(y) = attc$. Intuitively, such colorings are not extended further, because y is still an undefeated attacking candidate (i.e. y requires defeating). By properties (2) and (3) of Tree Decompositions, y is not attacked by any argument outside $X_{\geq n_2}$, i.e. there is no chance for y becoming defeated. The colorings for n_2 are now in accordance with the $X_{>n_2}$-restricted admissible tuples for $F_{\geq n_2} = F_{\geq n_1}$.*

We consider now nodes of type INSERT, but also here an additional lemma is required.

Lemma 3.5. *For any INSERT node t in a Tree Decomposition of an AF F with child node t' such that $X_t = X_{t'} \cup \{a\}$, and every $S \subseteq A$, the following relationships hold.*

1. *If (S, D) is an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$, then $(S\backslash\{a\}, D\backslash\{a\})$ is an $X_{>t'}$-restricted admissible tuple for $F_{\geq t'}$.*
2. *If (S, D) is an $X_{>t'}$-restricted admissible tuple for $F_{\geq t'}$, then $(S, D \cup \{a\})$ is also an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$.*
3. *If (S, D) is an $X_{>t'}$-restricted admissible tuple for $F_{\geq t'}$ and $S \cup \{a\}$ is conflict-free in $F_{\geq t}$ then $(S \cup \{a\}, D)$ is an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$.*
4. *If (S, D) is an $X_{>t'}$-restricted admissible tuple for $F_{\geq t'}$, $S \not\rightarrowtail_{attacks} a$ and $a \not\rightarrowtail_{attacks} S$, then (S, D) is also an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$.*

Proof. By assumption, we have $X_t = X_{t'} \cup \{a\}$ and $a \notin X_{t'}$. Thus, also $X_{\geq t} = X_{\geq t'} \cup \{a\}$ and $X_{>t} = X_{>t'}$ hold. By properties (2) and (3) of Tree Decompositions, we know that there are no attacks between the new argument a and arguments in $X_{>t}$.
First, let (S, D) be an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$. By $X_{>t} = X_{>t'}$, (S, D) is also $X_{>t'}$-restricted admissible for $F_{\geq t}$. Moreover, since a cannot attack any argument in $X_{>t'}$, also $(S\backslash\{a\}, D\backslash\{a\})$ is $X_{>t'}$-restricted admissible for $F_{\geq t'}$ (of course if $a \notin S$, then $S\backslash\{a\} = S$ and the latter admissibility property is trivial). This proves assertion (1).

Now consider an $X_{>t'}$-restricted admissible tuple (S, D) for $F_{\geq t'}$. Then S is conflict-free in F. Moreover, as explained above, there are no attacks between the new argument a and arguments in $X_{>t}$. Hence, the argument a does not affect the second condition for being an $X_{>t}$-restricted admissible set. Thus $(S, D \cup \{a\})$ and $(S \cup \{a\}, D)$ (in case $S \cup \{a\}$ is conflict-free) are $X_{>t}$-restricted admissible tuples of $F_{\geq t}$. This proves assertions (2) and (3). Moreover, if $a \not\rightarrowtail_{\text{attacks}} S$ and $S \not\rightarrowtail_{\text{attacks}} a$, it is not required that $a \in D$, i.e. (S, D) is already $X_{>t}$-restricted admissible for $F_{\geq t}$, which proves assertion (4). □

Lemma 3.6. *For any INSERT node t in a Tree Decomposition of an AF F, valid colorings and vcolorings coincide, if they coincide in the child node t' of t.*

Proof. Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of $F = (A, R)$, t an INSERT node in $\mathcal{T}$, and t' the child node of t. Moreover, let $X_t = X_{t'} \cup \{a\}$; observe that $a \notin X_{t'}$. Let C be a valid coloring for t, i.e., there exists an $X_{>t}$-restricted admissible tuple $(S, D) \in e_t(C)$ for $F_{\geq t}$. Then, by Lemma 3.5, $(S \backslash \{a\}, D \backslash \{a\})$ is $X_{>t'}$-restricted admissible for $F_{\geq t'}$. We construct a coloring C' for t' with $(S \backslash \{a\}, D \backslash \{a\}) \in e_{t'}(C')$ as follows. For arbitrary $b \in X_{t'}$, we define:

$$
\begin{aligned}
C'(b) &= \text{in} && \text{if } b \in S \backslash \{a\}, \\
C'(b) &= \text{def} && \text{if } b \notin S \text{ and } S \backslash \{a\} \rightarrowtail_{\text{attacks}} b, \\
C'(b) &= \text{attc} && \text{if } b \notin S, b \rightarrowtail_{\text{attacks}} S \backslash \{a\}, \text{ and } S \backslash \{a\} \not\rightarrowtail_{\text{attacks}} b, \\
C'(b) &= \text{attc} && \text{if } b \notin S, b \not\rightarrowtail_{\text{attacks}} S \backslash \{a\}, C(b) \neq \text{out and } S \backslash \{a\} \not\rightarrowtail_{\text{attacks}} b, \\
C'(b) &= \text{out} && \text{if } b \notin S, b \not\rightarrowtail_{\text{attacks}} S \backslash \{a\}, C(b) = \text{out and } S \backslash \{a\} \not\rightarrowtail_{\text{attacks}} b,
\end{aligned}
$$

Thus, $C' \in \mathcal{C}_{t'}$, and by assumption, is a vcoloring for t'. Moreover, it is easy to check that either $C = C' +_{\text{attc}} a$ or $C = C' \hat{+}_{\text{out}} a$ (if $a \notin S$) holds, or $C = C' \dot{+}_{\text{in}} a$ holds (if $a \in S$). Hence, C is a vcoloring for t.
Now let C be a vcoloring for t, i.e., there exists a vcoloring C' for t' with either $C = C' +_{\text{attc}} a$, $C = C' \dot{+}_{\text{in}} a$ or $C = C' \hat{+}_{\text{out}} a$. By assumption, C' is a valid coloring for t, i.e., there exists an $X_{>t'}$-restricted (and, hence $X_{>t}$-restricted) admissible tuple $(S, D) \in e_{t'}(C')$ of $F_{\geq t'}$. It is easy to check that then (using Lemma 3.5) $(S, D \cup \{a\}) \in e_t(C' +_{\text{attc}} a)$ and $(S, D) \in e_t(C' \hat{+}_{\text{out}} a)$ (if $S \not\rightarrowtail_{\text{attacks}} a$ and $a \not\rightarrowtail_{\text{attacks}} S$). Moreover, if the set $S \cup \{a\}$ is conflict-free in $F_{\geq t}$, then $(S \cup \{a\}, D) \in e_t(C \dot{+}_{\text{in}} a)$ as well. Thus, C (which is either $C' +_{\text{attc}} a$, $C' \dot{+}_{\text{in}} a$ or $C' \hat{+}_{\text{out}} a$) is a valid coloring for t. □

Example 3.13. *We continue our running example (Figure 3.2) of computing vcolorings w.r.t. $\mathcal{T}_{Ex}$ of Figure 3.1. Node n_3 adds argument x. Consider coloring C' of n_2 with $C'(w) = attc$. We have now three possibilities to add argument x (corresponding to the three different $+$-operations of vcolorings).*

- *$C = C' +_{attc} x$: This results in $C(x) = attc$ and $C(w) = attc$.*
- *$C = C' \dot{+}_{in} x$: If we set $C(x) = in$, this leads to $C(w) = def$ since $x \rightarrowtail_{attacks} w$.*

- $C = C' \hat{+}_{out} x$: *This leads to* $C(x) = out$ *and* $C(w) = attc$.

For JOIN nodes we need two additional helping lemmas as following.

Lemma 3.7. *Assume any JOIN node t in a Tree Decomposition of an AF F with child nodes t' and t''. Now let $S_1, D_1 \subseteq X_{\geq t'}$ and $S_2, D_2 \subseteq X_{\geq t''}$, such that*

1. *(S_1, D_1) is $X_{>t'}$-restricted admissible for $F_{\geq t'}$;*
2. *(S_2, D_2) is $X_{>t''}$-restricted admissible for $F_{\geq t''}$;*
3. *$S_1 \cap X_t = S_2 \cap X_t$.*

Then $(S = S_1 \cup S_2, D = D_1 \cup D_2)$ is an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$.

Proof. By properties (2) and (3) of Tree Decompositions, there are no attacks between the argument sets $X_{>t'}$ and $X_{>t''}$. In order to show that $S = S_1 \cup S_2$ is $X_{>t}$-restricted admissible, we have to prove that (a) S is conflict-free in the AF $F_{\geq t}$; (b) S defends itself against all attacks from arguments in $X_{>t} = X_{>t'} \cup X_{>t''}$ in $F_{\geq t}$ and (c) For every $a \in X_{\geq t} : ((S \rightarrowtail_{\text{attacks}} a \text{ or } a \rightarrowtail_{\text{attacks}} S) \implies a \in D)$ and $S \cap D = \emptyset$.
(a) Suppose to the contrary that there is a conflict $a \rightarrowtail_{\text{attacks}} b$ with $a, b \in S$. Then $a, b \in X_{\geq t'}$ (resp. $a, b \in X_{\geq t''}$) or $a \in X_{\geq t'}$ while $b \in X_{\geq t''}$ (or vice versa). In the case $a, b \in X_{\geq t'}$, we get $a, b \in S_1$ and, therefore, S_1 is not conflict-free in $F_{\geq t'}$, a contradiction to assumption 1 (the same argument applies to $a, b \in X_{\geq t''}$). Thus, assume $a \in X_{\geq t'}$ while $b \in X_{\geq t''}$ (or vice versa). Since there are no attacks between an argument from $X_{>t'}$ and an argument from $X_{>t''}$, it must hold that $a \in X_t$ or $b \in X_t$. Hence, $\{a, b\} \subseteq X_{\geq t'}$ or $\{a, b\} \subseteq X_{\geq t''}$ holds. Assuming $S_1 \cap X_t = S_2 \cap X_t$, this means that there is a conflict in S_1 or S_2, yielding a contradiction to assumption 1 or 2.
(b) We show that all arguments in S_1 are defended by S against arguments from $X_{>t}$ in $F_{\geq t}$. The analogous result for S_2 then follows by symmetry. In total, every argument in S is defended by S against arguments from $X_{>t}$ in $F_{\geq t}$. Together with the result from (a), we thus derive the desired result, i.e. that S is an $X_{>t}$-restricted admissible set for $F_{\geq t}$.
By assumption, S_1 defends itself against $X_{>t'}$ in $F_{\geq t'}$ and thus against $X_{>t'}$ in $F_{\geq t}$. Moreover, there are no attacks from $X_{>t''}$ against $X_{>t'}$ in $F_{\geq t}$ by the properties of Tree Decompositions. So $X_{>t''}$ can only attack arguments in $S_1 \cap X_t$. Thus, S_2 defends S_1 against $X_{>t''}$ since, $S_1 \cap X_t = S_2 \cap X_t$ and by assumption, S_2 defends itself against all attacks from $X_{>t''}$ in $F_{\geq t'}$ and thus also in $F_{\geq t}$. Putting this together, we have $S = S_1 \cup S_2$ defends S_1 against $X_{>t}$ in $F_{\geq t}$.
(c) Since both for every $a \in X_{\geq t'} : ((S_1 \rightarrowtail_{\text{attacks}} a \text{ or } a \rightarrowtail_{\text{attacks}} S_1) \implies a \in D_1)$ and $\forall a \in X_{\geq t''} : ((S_2 \rightarrowtail_{\text{attacks}} a \text{ or } a \rightarrowtail_{\text{attacks}} S_2) \implies a \in D_2)$ holds, $\forall a \in X_{\geq t} : ((S \rightarrowtail_{\text{attacks}} a \text{ or } a \rightarrowtail_{\text{attacks}} S) \implies a \in D)$ follows; $S \cap D = \emptyset$ holds since $S_1 \cap D_1 = S_2 \cap D_2 = \emptyset$ and $S_2 \cap D_1 = \emptyset$ holds since there are no attacks between the argument sets $X_{>t'}$ and $X_{>t''}$ and $X_t = X_{t'} = X_{t''}$ and $S_1 \cap X_t = S_2 \cap X_t$ ($S_1 \cap D_2 = \emptyset$ is by symmetry). □

Lemma 3.8. *Let (S, D) be an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$, $S_1 = S \cap X_{\geq t'}$, $S_2 = S \cap X_{\geq t''}$, $D_1 = D \cap X_{\geq t'}$ and $D_2 = D \cap X_{\geq t''}$. Then,*

1. *(S_1, D_1) is $X_{>t'}$-restricted admissible for $F_{\geq t'}$;*
2. *(S_2, D_2) is $X_{>t''}$-restricted admissible for $F_{\geq t''}$;*
3. *$S_1 \cap X_t = S_2 \cap X_t$.*
4. *$D_1 \cap X_t = D_2 \cap X_t$.*

Proof. Let (S, D) be an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$. Assumptions 3 and 4 are immediate by the fact that $X_{\geq t'} \cap X_{\geq t''} = X_t$. Moreover, since S is conflict-free in $F_{\geq t}$, each subset of S is conflict-free in any subframework of $F_{\geq t}$, in particular $S_1 = S \cap X_{\geq t'}$ is conflict-free in $F_{\geq t'}$ and $S_2 = S \cap X_{\geq t''}$ is conflict-free in $F_{\geq t''}$. It remains to show that S_1 (resp. S_2) defends itself against all attacks from $X_{>t'}$ (resp. from $X_{>t''}$) in $F_{\geq t'}$ (resp. in $F_{\geq t''}$). Suppose to the contrary that there exists $a \in X_{>t'}$ such that $a \rightarrowtail_{\text{attacks}} S_1$ and $S_1 \not\rightarrowtail_{\text{attacks}} a$ in $F_{\geq t'}$. Since S is $X_{>t}$-restricted admissible in $F_{\geq t}$, we know that $S \rightarrowtail_{\text{attacks}} a$ in $F_{\geq t}$. Hence, there has to exist an argument $b \in S \backslash S_1 = S \cap X_{>t''}$ such that $b \rightarrowtail_{\text{attacks}} a$ in $F_{\geq t}$. But, as already observed earlier, there are no attacks between $X_{>t'}$ and $X_{>t''}$, a contradiction. Since $X_{>t'} \subseteq X_{>t}$, (S_1, D_1) is $X_{>t'}$-restricted admissible for $F_{\geq t'}$. By symmetry, also (S_2, D_2) is $X_{>t''}$-restricted admissible for $F_{\geq t''}$. $\square$

Lemma 3.9. *For any JOIN node t in a Tree Decomposition of an AF F, valid colorings and vcolorings coincide, if they coincide also for both child nodes t' and t'' of t.*

Proof. Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of $F = (A, R)$ and t a JOIN node in $\mathcal{T}$ with successors t' and t''. Then $X_t = X_{t'} = X_{t''}$ and $X_{\geq t'} \cap X_{\geq t''} = X_t$ and $X_{\geq t} = X_{\geq t'} \cup X_{\geq t''}$. So we can partition $X_{\geq t}$ into three disjoint sets $X_{>t'}$, $X_{>t''}$ and X_t. Thus every set $S \subseteq X_{\geq t}$ can be seen as the union of two sets $S_1 \subseteq X_{\geq t'}$ and $S_2 \subseteq X_{\geq t''}$ with $S_1 \cap X_t = S_2 \cap X_t$. The Lemmas 3.7 and 3.8 identify important properties of these sets S_1 and S_2.
We now show that valid colorings and vcolorings for a JOIN node t coincide. First, let C be a vcoloring for t, i.e., $C = C' \bowtie C''$, where C' (resp. C'') is a vcoloring for t' (resp. t''), $[C'] = [C'']$ and $[[C']] = [[C'']]$. By assumption, C' and C'' are valid colorings for the respective nodes t' and t''. Hence, there exist $(S_1, D_1) \in e_{t'}(C')$ and $(S_2, D_2) \in e_{t''}(C'')$. Moreover, by $[C'] = [C'']$, we have $S_1 \cap X_t = S_2 \cap X_t$. Thus, by Lemma 3.7, $(S = S_1 \cup S_2, D = D_1 \cup D_2)$ is $X_{>t}$-restricted admissible. It remains to show that $(S, D) \in e_t(C)$. To this end, we check that conditions in Definition 3.5 are satisfied for every $a \in X_t$.

(i) By the definition of the $\bowtie$-operator in Definition 3.8, we have $C(a) = \text{in} \iff C'(a) = \text{in}$ and $C''(a) = \text{in}$. This, in turn, is equivalent to $a \in S_1$ and $a \in S_2$. In total we have $C(a) = \text{in} \iff a \in S$.

(ii) $C(a) = \text{def} \iff C'(a) = \text{def}$ or $C''(a) = \text{def}$ (see Definition 3.8) $\iff S_1 \rightarrowtail_{\text{attacks}} a$ or $S_2 \rightarrowtail_{\text{attacks}} a \iff S \rightarrowtail_{\text{attacks}} a$

(iii) $S \not\rightarrowtail_{\text{attacks}} a$ and $a \rightarrowtail_{\text{attacks}} S \iff S_1 \not\rightarrowtail_{\text{attacks}} a, S_2 \not\rightarrowtail_{\text{attacks}} a, a \rightarrowtail_{\text{attacks}} S_1$ and $a \rightarrowtail_{\text{attacks}} S_2 \implies C'(a) = \text{attc}$ and $C''(a) = \text{attc} \iff C(a) = \text{attc}$

(iv) $C(a) = \text{out} \iff C'(a) = \text{out}$ and $C''(a) = \text{out}$ (see Definition 3.8) $\implies S_1 \not\rightarrowtail_{\text{attacks}} a, S_2 \not\rightarrowtail_{\text{attacks}} a, a \not\rightarrowtail_{\text{attacks}} S_1$ and $a \not\rightarrowtail_{\text{attacks}} S_2 \implies S \not\rightarrowtail_{\text{attacks}} a$ and $a \not\rightarrowtail_{\text{attacks}} S$

(v) $C(a) \in \{\text{def}, \text{attc}\} \iff C'(a) \in \{\text{def}, \text{attc}\}$ or $C''(a) \in \{\text{def}, \text{attc}\}$ (see Definition 3.8) $\iff a \in D_1$ or $a \in D_2 \iff a \in D$.

Now assume that C is a valid coloring for t, i.e., there exists $(S, D) \in e_t(C)$. We define $S_1 = S \cap X_{\geq t'}, S_2 = S \cap X_{\geq t''}, D_1 = D \cap X_{\geq t'}$ and $D_2 = D \cap X_{\geq t''}$. Then, by Lemma 3.8, (S_1, D_1) is $X_{>t'}$-restricted admissible for $F_{\geq t'}$. (S_2, D_2) is $X_{>t''}$-restricted admissible for $F_{\geq t''}$, and $S_1 \cap X_t = S_2 \cap X_t$. We define a coloring C' at t' and a coloring C'' at t'', such that $(S_1, D_1) \in e_{t'}(C')$ and $(S_2, D_2) \in e_{t''}(C'')$. Then C' and C'' are valid colorings for the respective nodes t' and t'', and, therefore, by assumption they are also vcolorings for their node. Now define the vcoloring $C^* = C' \bowtie C''$ for node t. We claim that $C^* = C$ holds. To prove this claim, we have to show that $C^*(a) = C(a)$ for every $a \in X_t$. This equality is shown by distinguishing the four possible values $\{\text{in}, \text{def}, \text{attc}, \text{out}\}$ and by exploiting the conditions in Definition 3.5 as well as the definition of the $\bowtie$-operator in Definition 3.8. We only work out the case of in-nodes here; the remaining cases are treated analogously. Inspecting the $\bowtie$-definition in Definition 3.8, shows that $C^*(a) = \text{in} \iff C'(a) = \text{in}$ and $C''(a) = \text{in} \iff a \in S_1$ and $a \in S_2 \iff a \in S \iff C(a) = \text{in}$. □

Example 3.14. *The only JOIN node of our running example is $t = n_7$, which combines subframeworks $F_{\geq 3}$ and $F_{\geq 6}$ (see Figure 3.2 w.r.t. $\mathcal{T}_{Ex}$ of Figure 3.1). Assume C' resp. C'' to be a coloring for n_3 resp. n_6. Moreover let us agree on $C'(w) = def = C''(w)$ and $C'(x) = in = C''(x)$. Since $[C'] = [C'']$ and $[[C']] = [[C'']]$, the colorings coincide on $X_{\geq 3} \cap X_{\geq 6}$ and we can join these colorings without any conflict leading to $C = C' \bowtie C''$ with $C(x) = C'(x) = C''(x)$ and $C(w) = C'(w) = C''(w)$ for node n_7. Now consider coloring C^* with $C^*(w) = in$ and $C^*(x) = def$. It holds that $[C''] \neq [D']$ and $[C''] \cup [D'] = \{w, x\}$ has a conflict leading to the fact that C'' and D' does not result in a vcoloring for node $t = n_7$.*

Now we have everything to complete our correctness proof of admissible semantics.

Theorem 3.1. *Let $(\mathcal{T}, \mathcal{X})$ be a normalized Tree Decomposition of an AF $F = (A, R)$. Then, for each coloring C for a node $t \in \mathcal{T}$, it holds that C is a valid coloring for t iff C is a vcoloring for t.*

Proof. Structural induction and Lemmas 3.2 through 3.8. □

Since we have shown equivalence, recall now our Lemma 3.1 and that the A-restricted admissible sets for an AF $F = (A, R)$ are just the admissible sets for F. We are able to construct our valid coloring for the root r of any Tree Decomposition $\mathcal{T}$ via computing vcolorings in a bottom-up manner and therefore finally are for instance capable of computing (enumerating) our admissible sets via $e'_r(\epsilon)$ (using Lemma 3.1 assuming that

ϵ is the coloring for r). Observe that $\emptyset$ is by definition always an admissible extension, so ϵ trivially exists, but for enumerating (computing $e'_r(\epsilon)$) the vcoloring results for all the nodes of $\mathcal{T}$ are required. In an efficient implementation, this can be done via connecting vcolorings for the diverse nodes of $\mathcal{T}$ appropriately (during the bottom-up procedure; compare with Section 2.3 and [ABC$^+$14a]).

3.2 New algorithm for Semi-stable Semantics

In the previous section, we formally introduced both an algorithm for admissible semantics and concepts required for proving its correctness. The algorithm was based on [DPW12], but modified in such a way that we can now easily adapt it in order to achieve an algorithm for semi-stable semantics. For this, we need to recall Definition 2.15, where we defined the range of an admissible set $S \subseteq A$ of an AF $F = (A, R)$ as $S_R^+ = S \cup \{a \in A \mid S \rightarrowtail_{\text{attacks}} a\}$. So, our goal in this section now is to obtain an algorithm for semi-stable semantics, that is we want to compute admissible extensions with a subset-maximal range.

We require some helping lemmas for simplifications as follows.

Lemma 3.10. *Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of an AF F, $t \in \mathcal{T}$, and (S, D) an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$. Then, there is a coloring $C \in \mathcal{C}_t$ s.t. $(S, D) \in e_t(C)$.*

Proof. Since (S, D) is an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$, each argument $a \in X_t$ satisfies one of the following conditions: (i) $a \in S$ and $a \notin D$, (ii) $S \rightarrowtail_{\text{attacks}} a$ and $a \in D$, (iii) $S \not\rightarrowtail_{\text{attacks}} a$,$a \rightarrowtail_{\text{attacks}} S$ and $a \in D$, or (iv) $S \not\rightarrowtail_{\text{attacks}} a$ and $a \not\rightarrowtail_{\text{attacks}} S$. For these four cases, we define C as follows:

for case $(i) : C(a) = \text{in}$
for case $(ii) : C(a) = \text{def}$
for case $(iii) : C(a) = \text{attc}$
for case $(iv) : C(a) = \text{out}$

By the construction of C, S and D satisfy the conditions of Definition 3.5 and, since (S, D) is $X_{>t}$-restricted admissible for $F_{\geq t}$, it holds that $(S, D) \in e_t(C)$.

□

Lemma 3.11. *Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of an AF $F = (A, R)$ and let C, C' be different colorings for a node $t \in \mathcal{T}$. Then, $e_t(C) \cap e_t(C') = \emptyset$.*

Proof. Suppose to the contrary that there is a tuple $(S, D) \in e_t(C) \cap e_t(C')$, where C and C' are different colorings for t. Then there exists an argument $a \in X_t$ such that $C(a) \neq C'(a)$. It remains to inspect all possible pairs of values of $C(a)$ and $C'(a)$ and to derive a contradiction in each case. First let us consider the case where $C(a) = \text{in}$ and $C'(a) \in \{\text{def}, \text{attc}, \text{out}\}$. By Definition 3.5, $C(a) = \text{in}$ implies $a \in S$ and further $C'(a) \in \{\text{def}, \text{attc}, \text{out}\}$ implies $a \notin S$, a contradiction. We continue with the case where $C(a) = \text{def}$ and $C'(a) \in \{\text{attc}, \text{out}\}$. By Definition 3.5, $C(a) = \text{def}$ implies $S \rightarrowtail_{\text{attacks}} a$.

On the other hand, $C'(a) \in \{\text{attc}, \text{out}\}$ implies $S \not\rightarrowtail_{\text{attacks}} a$, a contradiction. Finally, the case $C(a) = \text{attc}$ and $C'(a) = \text{out}$. By Definition 3.5, $C(a) = \text{attc}$ implies $a \in D$, but $a \notin D$ since $C'(a) = \text{out}$. □

Similar to before, we define the semantical part of proving our semi-stable algorithm in form of valid pairs (Definition 3.15) and the vpairs in Definition 3.18, which require adjusted operations formalized in Definition 3.17 (i.e. the syntactical parts; in other words – and as before – our specification of the DP algorithm on TDs).

Definition 3.15. *Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of an AF F, $t \in \mathcal{T}$, and (C, Γ) a pair with C being a coloring for t and Γ being a set of colorings for t. We call (C, Γ) simply a pair for t and define $e_t(C, \Gamma)$ as the collection of tuples (S, D) which satisfy the following conditions.*

(i) $(S, D) \in e_t(C)$;
(ii) *For all $C' \in \Gamma$, there is an $(E, U) \in e_t(C')$ such that $S \cup D \subset E \cup U$;*
(iii) *For all $X_{>t}$-restricted admissible (for $F_{\geq t}$) tuples (E, U) with $S \cup D \subset E \cup U$, there exists some $C' \in \Gamma$ with $(E, U) \in e_t(C')$.*

If $e_t(C, \Gamma) \neq \emptyset$, (C, Γ) is a valid pair for t.

Definition 3.16. *Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of an AF F and $t \in \mathcal{T}$. Given a pair (C, Γ) for t, we define $e'_t(C, \Gamma)$ for convenience as follows: $e'_t(C, \Gamma) := \{S \mid (S, D) \in e_t(C, \Gamma)\}$.*

Definition 3.17. *Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of an AF F. Moreover, assume Γ resp. Δ to be a set of colorings for node t' resp. $t'' \in \mathcal{T}$. We define the following operations (compare with Definition 3.8).*

$$
\begin{aligned}
\Gamma - a &= \{C - a \mid C \in \Gamma, C(a) \neq attc\} \\
\Gamma +_{attc} a &= \{C +_{attc} a \mid C \in \Gamma, [C] \not\rightarrowtail_{attacks} a, a \not\rightarrowtail_{attacks} [C]\} \\
\Gamma \dot{+}_{in} a &= \{C \dot{+}_{in} a \mid C \in \Gamma, [C] \not\rightarrowtail_{attacks} a, a \not\rightarrowtail_{attacks} [C], a \not\rightarrowtail_{attacks} a \text{ and} \\
&\quad [[C]] = [[C \dot{+}_{in} a]]\} \\
\Gamma \hat{+}_{out} a &= \{C \hat{+}_{out} a \mid C \in \Gamma\} \\
\Gamma \bowtie \Delta &= \{C \bowtie D \mid C \in \Gamma, D \in \Delta, [C] = [D] \text{ and } [[C]] = [[D]]\}
\end{aligned}
$$

Definition 3.18. *Let $(\mathcal{T}, \mathcal{X})$ be a normalized Tree Decomposition of an AF F and let $t \in \mathcal{T}$ be a node with t', t'' its possible children. Depending on the node type of t we define a vpair for t as follows.*

- *LEAF: Each (C, Γ) where $C \in \mathcal{C}_t$ and $\Gamma = \{C' \in \mathcal{C}_t \mid [C] \cup [[C]] \subset [C'] \cup [[C']]\}$ is a vpair for t.*
- *FORGET: If (C', Γ') is a vpair for t', $X_t = X_t \backslash \{a\}$, and $C'(a) \neq attc$, then $(C' - a, \Gamma' - a)$ is a vpair for t.*

- *INSERT: If (C', Γ') is a vpair for t' and $X_t = X_{t'} \cup \{a\}$, and if $C' \dot{+}_{in} a$ is a vcoloring for t then $(C' \dot{+}_{in} a, (\Gamma' +_{attc} a) \cup (\Gamma' \dot{+}_{in} a))$ is a vpair for t; if moreover $C' \hat{+}_{out} a$ is a vcoloring for t, then $(C' \hat{+}_{out} a, (\{C'\} +_{attc} a) \cup (\{C'\} \dot{+}_{in} a) \cup (\Gamma' +_{attc} a) \cup (\Gamma' \dot{+}_{in} a) \cup (\Gamma' \hat{+}_{out} a))$ is a vpair for t; $(C' +_{attc} a, (\Gamma' +_{attc} a) \cup (\Gamma' \dot{+}_{in} a))$ is a vpair for t.*

- *JOIN: If (C', Γ') is a vpair for t', (C'', Γ'') is a vpair for t'', $[C'] = [C'']$ and $[[C']] = [[C'']]$, then $(C' \bowtie C'', (\Gamma' \bowtie \Gamma'') \cup (\{C'\} \bowtie \Gamma'') \cup (\Gamma' \bowtie \{C''\}))$ is a vpair for t.*

Example 3.19. *Figure 3.3 illustrates the computation of the vpairs for F_{Ex} of Figure 3.1 w.r.t. the Tree Decomposition shown in Figure 3.1.*

In addition to Figure 3.2, the additional column Γ of the table of every Tree Decomposition node uses column ID and represents (strict) counter candidates. Observe that by following the extension pointers, we get that semi-stable$(A_{F_{Ex}}) = \{\{v, w\}\}$ (compare to Example 2.16).

Observe that indeed there exists coloring C s.t. (C, Γ) and (C, Γ') being pairs for the same node with $\Gamma \neq \Gamma'$.

Similar to the begin of this subsection, we need two additional lemmas adjusted to pairs as follows.

Lemma 3.12. *Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of an AF F, $t \in \mathcal{T}$, and (S, D) an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$. Then, there is a pair (C, Γ) for t s.t. $(S, D) \in e_t(C, \Gamma)$.*

Proof. Let (S, D) be an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$. By Lemma 3.10, there exists a coloring C with $(S, D) \in e_t(C)$. Moreover, let $\mathcal{E} = \{(E, U) \mid (E, U)$ is $X_{>t}$-restricted admissible for $F_{\geq t}$ s.t. $S \cup D \subset E \cup U\}$. Moreover, let $\Gamma = \{C' \mid \exists (E, U) \in \mathcal{E},$ s.t. $(E, U) \in e_t(C')\}$. We claim that $(S, D) \in e_t(C, \Gamma)$. To prove this, we check conditions (i)-(iii) from Definition 3.15: $(i)(S, D) \in e_t(C)$ by the selection of C. (ii) For all $C' \in \Gamma$, there exists $(E, U) \in e_t(C')$ with $S \cup D \subset E \cup U$; this follows by the construction of Γ from $\mathcal{E}$. (iii) For all $X_{>t}$-restricted admissible tuples (E, U) (in $F_{\geq t}$) with $S \cup D \subset E \cup U$, there exists $C' \in \Gamma$ with $(E, U) \in e_t(C')$; again this follows by the construction of Γ from $\mathcal{E}$. □

Lemma 3.13. *Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of an AF $F = (A, R)$, $t \in \mathcal{T}$, and let $(C, \Gamma), (C', \Gamma')$ be different pairs for t (but not necessarily $C \neq C'$). Then, $e_t(C, \Gamma) \cap e_t(C', \Gamma') = \emptyset$.*

Proof. If $C \neq C'$ then, by Lemma 3.11, $e_t(C) \cap e_t(C') = \emptyset$ and our claim follows. Thus, it remains to consider pairs $(C, \Gamma), (C', \Gamma')$ with $C = C'$ and $\Gamma \neq \Gamma'$. W.l.o.g., we assume that there exists a coloring $\bar{C}$ for t such that $\bar{C} \in \Gamma$ but $\bar{C} \notin \Gamma'$. In order to show that $e_t(C, \Gamma) \cap e_t(C', \Gamma') = \emptyset$, we prove that none of the tuples $(S, D) \in e_t(C, \Gamma)$ is contained in $e_t(C', \Gamma')$.
Let (S, D) be an arbitrary tuple in $e_t(C, \Gamma)$. Suppose to the contrary that (S, D) is also contained in $e_t(C, \Gamma')$. By Definition 3.15 (applied to $e_t(C, \Gamma)$), there exists an

$X_{>t}$-restricted admissible tuple $(E, U) \in e_t(\bar{C})$ for $F_{\geq t}$ such that $S \cup D \subset E \cup U$. By Definition 3.15 (applied to $e_t(C', \Gamma')$), there exists a coloring $C^* \in \Gamma'$ such that $(E, U) \in e_t(C^*)$. By Lemma 3.11, the colorings $\bar{C}$ and C^* coincide. Thus, $\bar{C} \in \Gamma'$, a contradiction. □

The following proposition is the key ingredient in our proof and justifies why it is enough to just show equivalence between valid pairs and vpairs.

Proposition 3.1. *Let r be the root of a normalized Tree Decomposition $(\mathcal{T}, \mathcal{X})$ of an AF F. Then, $e'_r(\epsilon, \emptyset) =$ semi-stable(F).*

Proof. Recall that $e'_r(\epsilon) =$ admissible(F) (see Lemma 3.1, Definitions 3.2 and 3.5). To show $e'_r(\epsilon, \emptyset) \subseteq$ semi-stable(F), let (S, D) be an arbitrary tuple s.t. $(S, D) \in e_r(\epsilon, \emptyset)$. By Definition 3.15(i) we obtain that S is admissible for $F_{\geq r} = F$. Further by (iii) and the fact that $\Gamma = \emptyset$ we conclude that there is no admissible tuple (E, U) for F with $E \cup U$ being a proper superset of $S \cup D$, i.e. S is a semi-stable extension of F. It remains to show that $e'_r(\epsilon, \emptyset) \supseteq$ semi-stable(F). Thus, let $S \in$ semi-stable(F) be an arbitrary semi-stable extension with range R of F. We set $D = R \backslash S$ to get the arguments requiring defeating. By Lemma 3.12 we get that there exists a pair (C, Γ) such that $(S, D) \in e_r(C, \Gamma)$. Since the root node has an empty bag, $C = \epsilon$ and further, by Definition 3.15(ii) and the fact that $S \cup D$ is maximal (w.r.t. $\subset$-inclusion) in F, we conclude that $\Gamma = \emptyset$ has to hold as well. □

In the following, the different node types handle equivalence between vpairs and valid pairs (similar to the equivalence between vcolorings and valid colorings of the previous section).

Lemma 3.14. *For any LEAF node t in a Tree Decomposition of an AF F, its vpairs coincide with its valid pairs.*

Proof. Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of F and t a leaf node in $\mathcal{T}$. The $X_{>t}$-restricted admissible tuples for $F_{\geq t}$ coincide with the tuples $([C], [[C]])$ for the valid colorings $C \in \mathcal{C}_t$. Moreover, the valid colorings and vcolorings for t coincide by Lemma 3.2. Now let (C, Γ) be a valid pair for t. Then, by Definition 3.15, $([C], [[C]]) \in e_t(C, \Gamma)$. Hence, by Definition 3.18, (C, Γ) is a vpair for t.
Conversely, let (C, Γ) be a vpair for t and let $S = [C], D = [[C]]$. By Definition 3.9, (S, D) is $X_{>t}$-restricted admissible for $F_{\geq t}$. Hence, by Definitions 3.15 and 3.18, $(S, D) \in e_t(C, \Gamma)$. (C, Γ) is thus a valid pair for node t. □

Lemma 3.15. *For any FORGET node t in a Tree Decomposition of an AF F, vpairs and valid pairs coincide, if they coincide in the child node t' of t.*

Proof. Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of $F = (A, R)$, t a FORGET node in $\mathcal{T}$, and t' the child node of t. We have that $X_t = X_{t'} \backslash \{a\}$ for some argument $a \in X_{t'}$.

First we show that every valid pair for t is also a vpair for t. Thus, let (C, Γ) be a valid pair for t. Then there exists a tuple $(S, D) \in e_t(C, \Gamma)$. In particular, (S, D) is $X_{>t}$-restricted admissible for $F_{\geq t}$, and hence, also $X_{>t'}$-restricted admissible for $F_{\geq t'} = F_{\geq t}$.

Thus, by Lemmas 3.12 and 3.13, there exists one unique, valid pair (C', Γ') for t' with $(S, D) \in e_{t'}(C', \Gamma')$. By assumption (C', Γ') is a vpair for t'. Since (S, D) is $X_{>t}$-restricted admissible for $F_{\geq t}$ and $(S, D) \in e_{t'}(C')$, we have $C'(a) \neq \text{attc}$. Then $(C' - a, \Gamma' - a)$ is a vpair for t. We claim that $(C' - a, \Gamma' - a) = (C, \Gamma)$ holds.

For $C' - a = C$, recall the construction from the proof of Lemma 3.4, where we constructed a coloring, which we denote here as C^*, such that $C^* - a = C$ and $(S, D) \in e_{t'}(C^*)$. As also $(S, D) \in e_{t'}(C')$ holds, by Lemma 3.11, we have that $C' = C^*$ and thus $C' - a = C$.

To show $\Gamma' - a = \Gamma$, we first consider the inclusion $\Gamma' - a \subseteq \Gamma$: Let $T' \in \Gamma'$ with $T'(a) \neq \text{attc}$. By condition (ii) of Definition 3.15, there exists an $X_{>t'}$-restricted admissible tuple (E, U) for $F_{\geq t'}$ with $S \cup D \subset E \cup U$ and $(E, U) \in e_{t'}(T')$. By $T'(a) \neq \text{attc}$, we know that (E, U) is also $X_{>t}$-restricted admissible. Hence, by condition (iii) of Definition 3.15, there exists $T \in \Gamma$ with $(E, U) \in e_t(T)$. As before, one can use the construction of Lemma 3.4 and Lemma 3.11, to obtain $T = T' - a$. Hence, $\Gamma' - a \subseteq \Gamma$.

Now consider an arbitrary coloring $T \in \Gamma$. By condition (ii) of Definition 3.15, there exists $X_{>t}$-restricted admissible tuple (E, U) for $F_{\geq t}$ with $S \cup D \subseteq E \cup U$ and $(E, U) \in e_t(T)$. By condition (iii) of Definition 3.15 and since (E, U) is also $X_{>t'}$-restricted admissible for $F_{\geq t'}$, there exists $T' \in \Gamma'$ with $(E, U) \in e_{t'}(T')$. Again, by the construction of the proof of Lemma 3.4 and Lemma 3.11 we have that $T = T' - a$. Hence, $\Gamma \subseteq \Gamma' - a$.

We now show that every vpair of the FORGET node t is also valid pair for t. Let (C, Γ) be a vpair for t, i.e., there exists a vpair (C', Γ') for node t' with $C'(a) \neq \text{attc}$ and $(C, \Gamma) = (C' - a, \Gamma' - a)$. By assumption, (C', Γ') is a valid pair for t'. Hence, there exists $(S, D) \in e_{t'}(C', \Gamma')$. We claim that also $(S, D) \in e_t(C, \Gamma)$ holds. As in the proof of Lemma 3.4, $(S, D) \in e_t(C)$ holds since $C = C' - a$. It remains to show that also conditions (ii) and (iii) of Definition 3.15 are fulfilled.

To show condition (ii), let $T \in \Gamma$, i.e. T is of the form $T = T' - a$ for some $T' \in \Gamma'$ with $T'(a) \neq \text{attc}$. Since $(S, D) \in e_{t'}(C', \Gamma')$, there exists $(E, U) \in e_{t'}(T')$ with $S \cup D \subset E \cup U$. As in the proof of Lemma 3.4, then also $(E, U) \in e_t(T' - a)$. To show condition (iii), let (E, U) be $X_{>t}$-restricted admissible for $F_{\geq t}$ with $S \cup D \subset E \cup U$. Then (E, U) is also $X_{>t'}$-restricted admissible for $F_{\geq t'}$, and therefore, there exists $T' \in \Gamma'$ with $(E, U) \in e_{t'}(T')$. Since (E, U) is $X_{>t}$-restricted admissible, we have $T'(a) \neq \text{attc}$. But then, as in the proof of Lemma 3.4, also $(E, U) \in e_t(T' - a)$.

□

Lemma 3.16. *For any INSERT node t in a Tree Decomposition of an AF F, vpairs and valid pairs coincide, if they coincide in the child node t' of t.*

Proof. Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of $F = (A, R)$, t an INSERT node in $\mathcal{T}$, and t' the child node of t. Hence we have that $X_t = X_{t'} \cup \{a\}$ for some argument $a \in A$.

First we show that every valid pair for t is also a vpair for t. Thus let (C, Γ) be a valid pair for t. Then there exists $(S, D) \in e_t(C, \Gamma)$, which is $X_{>t}$-restricted admissible for $F_{\geq t}$ and further the set $(S' = S \setminus \{a\}, D' = D \setminus \{a\})$ is $X_{>t'}$-restricted admissible for $F_{\geq t'}$. Thus, by Lemmas 3.12 and 3.13, there exists a unique valid pair (C', Γ') for t' with $(S', D') \in e_{t'}(C', \Gamma')$. By assumption, (C', Γ') is a vpair for t'. Then, if $[C'] \not\rightarrow_{\text{attacks}} a$

and $a \not\rightarrowtail_{\text{attacks}} [C']$, $(C' \hat{+}_{\text{out}}\ a, \Gamma_1)$ with $\Gamma_1 = \{(\{C'\} +_{\text{attc}}\ a) \cup (\{C'\}\ \dot{+}_{\text{in}}\ a) \cup (\Gamma' +_{\text{attc}}\ a) \cup (\Gamma'\ \dot{+}_{\text{in}}\ a) \cup (\Gamma'\ \hat{+}_{\text{out}}\ a)\}$, and further, if $[C'] \cup \{a\}$ is conflict-free in $F_{t'}$ and $[[C']] = [[C'\ \dot{+}_{\text{in}}\ a]]$, $(C'\ \dot{+}_{\text{in}}\ a, (\Gamma' +_{\text{attc}} a) \cup (\Gamma'\ \dot{+}_{\text{in}}\ a))$ are both vpairs. Moreover, $(C' +_{\text{attc}} a, (\Gamma' +_{\text{attc}} a) \cup (\Gamma'\ \dot{+}_{\text{in}}\ a))$ is a vpair. We claim that either $(C'\ \hat{+}_{\text{out}}\ a, \Gamma_1) = (C, \Gamma)$, $(C' +_{\text{attc}} a, (\Gamma' +_{\text{attc}} a) \cup (\Gamma'\ \dot{+}_{\text{in}}\ a)) = (C, \Gamma)$ or $(C'\ \dot{+}_{\text{in}}\ a, (\Gamma' +_{\text{attc}} a) \cup (\Gamma'\ \dot{+}_{\text{in}}\ a)) = (C, \Gamma)$ holds.

To show that either $C = C' +_{\text{attc}} a$, $C = C'\ \hat{+}_{\text{out}}\ a$ (if $a \notin S$) or $C = C'\ \dot{+}_{\text{in}}\ a$ holds, recall the proof of Lemma 3.6, where we constructed a coloring, which we denote here as C^*, such that $C = C^* +_{\text{attc}} a$, $C = C^*\ \hat{+}_{\text{out}}\ a$ or $C = C^*\ \dot{+}_{\text{in}}\ a$ and $(S \backslash \{a\}, D \backslash \{a\}) \in e_{t'}(C^*)$. As also $(S \backslash \{a\}, D \backslash \{a\}) \in e_{t'}(C')$ holds, by Lemma 3.11 we have that $C' = C^*$ and the assertion follows.

In the following we show that also the respective sets of certificates coincide. To this end we distinguish the two mentioned cases $a \notin S$ and $a \in S$, respectively:

(1a) Assume $a \notin S, a \notin D$: To derive $\Gamma_1 = \Gamma$, we first show the relation $\Gamma_1 \subseteq \Gamma$; this can be split up into the following statements:
- (α) $\Gamma' +_{\text{attc}} a \subseteq \Gamma$,
- (β) $\Gamma'\ \dot{+}_{\text{in}}\ a \subseteq \Gamma$,
- (γ) $C'\ \dot{+}_{\text{in}}\ a \subseteq \Gamma$,
- (δ) $C' +_{\text{attc}}\ a \subseteq \Gamma$,
- (ϵ) $\Gamma'\ \hat{+}_{\text{out}}\ a \subseteq \Gamma$.

To show (α), (β) and (ϵ), consider $T' \in \Gamma'$. By condition (ii) of Definition 3.15, there exists an $X_{>t'}$-restricted admissible tuple (E', U') for $F_{\geq t'}$ with $S' \cup D' \subset E' \cup U'$ and $(E', U') \in e_{t'}(T')$. As we have here $S = S', D = D'$, we obtain $S \cup D \subset (E = E') \cup (U = U' \cup \{a\})$ for (α), $S \cup D \subset (E = E' \cup \{a\}) \cup (U = U')$ for (β) and $S \cup D \subset (E = E') \cup (U = U')$ for (ϵ). In the first case we have that E is conflict-free in $F_{\geq t}$ by definition, and further as $X_{>t} = X_{>t'}$ and $a \notin X_{>t}$, (E, U) is also an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$. In the second case E is conflict-free in $F_{\geq t}$ iff the set $[T'] \cup \{a\}$ is conflict-free. This is due to the definition of Tree Decompositions which ensures that there are no attacks between the set $X_{>t}$ and the new argument a. Using that a is not attacked by $X_{>t}$ or the third case, we get that if E is conflict-free in $F_{\geq t}$ then (E, U) is also an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$.
Now by condition (iii) of Definition 3.15 there exists $T \in \Gamma$ such that $(E, U) \in e_t(T)$. As before, using the construction from the proof of Lemma 3.6 together with Lemma 3.11, we obtain that, in the case (α) it holds that $T = T' +_{\text{attc}} a$, in the case (β) it holds that $T = T'\ \dot{+}_{\text{in}}\ a$, and in case ($\epsilon$), $T = T'\ \hat{+}_{\text{out}}\ a$ holds. Next, we prove statement (γ). To do this let us consider the set $(C'\ \dot{+}_{\text{in}}\ a)$. If $(C'\ \dot{+}_{\text{in}}\ a) = \emptyset$ statement (γ) is trivially true. Otherwise we have that $(S \cup \{a\}, D) \in e_t(C'\ \dot{+}_{\text{in}}\ a)$ and as $S \cup D \subset S \cup \{a\} \cup D$ that $C'\ \dot{+}_{\text{in}}\ a \in \Gamma$. Hence, $C'\ \dot{+}_{\text{in}}\ a \subseteq \Gamma$. Let us consider $C' +_{\text{attc}} a$ of (δ); we have that $(S, D \cup \{a\}) \in e_t(C' +_{\text{attc}} a)$ and as $S \cup D \subset S \cup D \cup \{a\}$, $C' +_{\text{attc}} a \in \Gamma$ holds. Finally, $\Gamma_1 \subseteq \Gamma$.
To prove $\Gamma \subseteq \Gamma_1$, consider an arbitrary $T \in \Gamma$. By condition (ii) of Definition 3.15, there exists an $X_{>t}$-restricted admissible tuple (E, U) for $F_{\geq t}$ with $S \cup D \subset E \cup U$ and $(E, U) \in e_t(T)$. By the assumption $a \notin S, a \notin D$, i.e. $S = S', D = D'$, we have

that for $E' = E\backslash\{a\}, U' = U\backslash\{a\}$ either $S' \cup D' \subset E' \cup U'$, $E = S \cup \{a\}, U = D'$ (i.e. $E' = S$) or $E = S, U = D \cup \{a\}$ (i.e. $U' = D$) holds. In both cases we have that (E', U') is $X_{>t'}$-restricted admissible for $F_{\geq t'}$ and thus there exists $T' \in \Gamma'$ with $(E', U') \in e_{t'}(T')$. Now we can use the proof of Lemma 3.6 together with Lemma 3.11 to show that in the case $S' \cup D' \subset E' \cup U'$, $T = T' \dot{+}_{\text{in}} a$, $T = T' +_{\text{attc}} a$ or $T = T' \hat{+}_{\text{out}} a$ holds; or $T = C' +_{\text{attc}} a$ or $T = C' \dot{+}_{\text{in}} a$ otherwise. Hence, $\Gamma \subseteq \Gamma_1$.

(1b) Assume $a \notin S, a \in D$: To show $(\Gamma' +_{\text{attc}} a) \cup (\Gamma' \dot{+}_{\text{in}} a) = \Gamma$, we first show the relation $(\Gamma' +_{\text{attc}} a) \cup (\Gamma' \dot{+}_{\text{in}} a) \subseteq \Gamma$
Consider $T' \in \Gamma'$. By condition (ii) of Definition 3.15, there exists an $X_{>t'}$-restricted admissible tuple (E', U') for $F_{\geq t'}$ with $S' \cup D' \subset E' \cup U'$ and $(E', U') \in e_{t'}(T')$. As we have here $S = S', D = D' \cup \{a\}$, we obtain $S \cup D \subset (E = E') \cup (U = U' \cup \{a\})$ for showing $((\Gamma' +_{\text{attc}} a) \subseteq \Gamma)$ and $S \cup D \subset (E = E' \cup \{a\}) \cup (U = U')$ for $((\Gamma' \dot{+}_{\text{in}} a) \subseteq \Gamma)$. In the first case we have that E is conflict-free in $F_{\geq t}$ by definition, and further as $X_{>t} = X_{>t'}$ and $a \notin X_{>t}$, (E, U) is also an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$. In the latter case E is conflict-free in $F_{\geq t}$ iff the set $[T'] \cup \{a\}$ is conflict-free. This is due to the definition of Tree Decompositions which ensures that there are no attacks between the set $X_{>t}$ and the new argument a. Using that a is not attacked by $X_{>t}$, we get that if E is conflict-free in $F_{\geq t}$ then (E, U) is also an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$.
Now by condition (iii) of Definition 3.15 there exists $T \in \Gamma$ such that $(E, U) \in e_t(T)$. As before, using the construction from the proof of Lemma 3.6 together with Lemma 3.11, we obtain that, in the first case it holds that $T = T' +_{\text{attc}} a$, and in the second case it holds that $T = T' \dot{+}_{\text{in}} a$. Finally $((\Gamma' +_{\text{attc}} a) \cup (\Gamma' \dot{+}_{\text{in}} a)) \subseteq \Gamma$.
To prove $\Gamma \subseteq ((\Gamma' +_{\text{attc}} a) \cup (\Gamma' \dot{+}_{\text{in}} a))$, consider an arbitrary $T \in \Gamma$. By condition (ii) of Definition 3.15, there exists an $X_{>t}$-restricted admissible tuple (E, U) for $F_{\geq t}$ with $S \cup D \subset E \cup U$ and $(E, U) \in e_t(T)$. By the assumption $a \notin S, a \in D$, i.e. $S = S', D = D' \cup \{a\}$, we have that for $E' = E\backslash\{a\}, U' = U\backslash\{a\}$ $S' \cup D' \subset E' \cup U'$ holds. We have that (E', U') is $X_{>t'}$-restricted admissible for $F_{\geq t'}$ and thus there exists $T' \in \Gamma'$ with $(E', U') \in e_{t'}(T')$. Now we can use the proof of Lemma 3.6 together with Lemma 3.11 to show that since $S' \cup D' \subset E' \cup U'$, $T = T' \dot{+}_{\text{in}} a$ or $T = T' +_{\text{attc}} a$ otherwise. Hence, $\Gamma \subseteq ((\Gamma' +_{\text{attc}} a) \cup (\Gamma' \dot{+}_{\text{in}} a))$.

(2) Assume $a \in S$: To show $(\Gamma' +_{\text{attc}} a) \cup (\Gamma' \dot{+}_{\text{in}} a) = \Gamma$, we first consider the inclusion $(\Gamma' +_{\text{attc}} a) \cup (\Gamma' \dot{+}_{\text{in}} a) \subseteq \Gamma$: Consider $T' \in \Gamma'$. By condition (ii) Definition 3.15, there exists an $X_{>t'}$-restricted admissible tuple (E', U') for $F_{\geq t'}$ with $S' \cup D' \subset E' \cup U'$ and $(E', U') \in e_{t'}(T')$. As by assumption $S = S' \cup \{a\}$ we have that on the one hand $S \cup D \subset E' \cup (U = U' \cup \{a\})$ and (E, U) is $X_{>t}$-restricted admissible for $F_{\geq t}$. On the other hand, $S \cup D \subset (E = E' \cup \{a\}) \cup U$ and as in case (1) if $[T'] \cup \{a\}$ is conflict-free then (E, U) is $X_{>t}$-restricted admissible for $F_{\geq t}$. In both cases we get by Definition 3.15 that there exists $T \in \Gamma$ such that $(E, U) \in e_t(T)$. By the construction from the proof of Lemma 3.6 and Lemma 3.11, it holds that $T = T' +_{\text{attc}} a$ or $T = T' \dot{+}_{\text{in}} a$. Hence $((\Gamma' +_{\text{attc}} a) \cup (\Gamma' \dot{+}_{\text{in}} a)) \subseteq \Gamma$.
To prove $\Gamma \subseteq ((\Gamma' +_{\text{attc}} a) \cup (\Gamma' \dot{+}_{\text{in}} a))$, consider an arbitrary $T \in \Gamma$. By condition (ii) of Definition 3.15, there exists an $X_{>t}$-restricted admissible tuple (E, U) for

$F_{\geq t}$ with $S \cup D \subset E \cup U$ and $(E,U) \in e_t(T)$. We have that both $S' \cup D' \subset (E' = E \backslash \{a\}) \cup (U' = U)$ and $S' \cup D' \subset (E' = E) \cup (U' = U \backslash \{a\})$ holds; further that (E',U') in both cases is $X_{>t'}$-restricted admissible for $F_{\geq t'}$. Thus there exists $T' \in \Gamma'$ with $(E',U') \in e_{t'}(T')$. By the construction from the proof of Lemma 3.6 and Lemma 3.11, we get that $T = T' +_{\text{attc}} a$ or $T = T' \dot{+}_{\text{in}} a$. Hence $\Gamma \subseteq ((\Gamma' +_{\text{attc}} a) \cup (\Gamma' \dot{+}_{\text{in}} a))$ holds.

It remains to show that every vpair for an INSERT node is also a valid pair. Thus let (C,Γ) be a vpair for t, i.e., there exists a vpair (C',Γ') for node t' such that either (1) $(C,\Gamma) = (C' \hat{+}_{\text{out}} a, \Gamma_1)$ (with $[C'] \not\rightarrowtail_{\text{attacks}} a, a \not\rightarrowtail_{\text{attacks}} [C']$; Γ_1 defined as above, recall that $\Gamma_1 = \{(\{C'\} +_{\text{attc}} a) \cup (\{C'\} \dot{+}_{\text{in}} a) \cup (\Gamma' +_{\text{attc}} a) \cup (\Gamma' \dot{+}_{\text{in}} a) \cup (\Gamma' \hat{+}_{\text{out}} a)\})$, (2) $(C,\Gamma) = (C' \dot{+}_{\text{in}} a, (\Gamma' +_{\text{attc}} a) \cup (\Gamma' \dot{+}_{\text{in}} a))$ with $[C] \cup \{a\}$ being conflict-free in F_t and $[[C]] = [[C \dot{+}_{\text{in}} a]]$ or (3) $(C,\Gamma) = (C' +_{\text{attc}} a, (\Gamma' +_{\text{attc}} a) \cup (\Gamma' \dot{+}_{\text{in}} a))$. By assumption, (C',Γ') is a valid pair for t' and thus there exists $(S',D') \in e_{t'}(C',\Gamma')$. To show that (C,Γ) is a valid pair for t we distinguish the cases (1), (2) and (3) as follows:

(1) As in the proof of Lemma 3.6, $(S = S', D = D') \in e_t(C)$ holds since $C = C' \hat{+}_{\text{out}} a$. It remains to show that also conditions (ii) and (iii) of Definition 3.15 are fulfilled. To show condition (ii), consider an arbitrary $T \in \Gamma$, i.e., T is either of the form
   (a) $T = T' +_{\text{attc}} a$
   (b) $T = T' \dot{+}_{\text{in}} a$ with $[T] \cup \{a\}$ conflict-free in F_t and $[[T']] = [[T' \dot{+}_{\text{in}} a]]$,
   (c) $T = T' \hat{+}_{\text{out}} a$ with $[T'] \not\rightarrowtail_{\text{attacks}} a$ and $a \not\rightarrowtail_{\text{attacks}} [T']$,
   (d) $T = C' \dot{+}_{\text{in}} a$ with $[C'] \cup \{a\}$ conflict-free in F_t and $[[C']] = [[C' \dot{+}_{\text{in}} a]]$ or
   (e) $T = C' +_{\text{attc}} a$.

   Since $(S, D') \in e_{t'}(C',\Gamma')$, there exists $(E',U') \in e_{t'}(T')$ with $S \cup D' \subset E' \cup U'$. In case (a), we follow the proof of Lemma 3.6, and obtain $(E', U' \cup \{a\}) \in e_t(T' +_{\text{attc}} a)$. For case (b), we get by the construction of T that $E = E' \cup \{a\}$ is conflict-free in $F_{\geq t}$. Once more we can use the fact that $X_t \not\rightarrowtail_{\text{attacks}} a$ to obtain that $(E, U = U')$ is an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$. Further $S \cup D \subset E \cup U$ and as in the proof of Lemma 3.6, then also $(E,U) \in e_t(T' \dot{+}_{\text{in}} a)$ holds. For case (c), we get by the construction of T that $[T'] \not\rightarrowtail_{\text{attacks}} a$ and $a \not\rightarrowtail_{\text{attacks}} [T']$ to obtain that $(E = E', U = U')$ is an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$. Further $S \cup D \subset E \cup U$ and as in the proof of Lemma 3.6, then also $(E,U) \in e_t(T' \hat{+}_{\text{out}} a)$ holds. Finally, for (d) resp. (e) the construction of $T = C' \dot{+}_{\text{in}} a$ resp. $T = C' +_{\text{attc}} a$ yields that $(E = (S \cup \{a\}), U = D')$ resp. $(E = S, U = (D \cup \{a\}))$ is conflict-free in $F_{\geq t}$ and thus as before (E,U) is an $X_{>t}$-restricted admissible set for $F_{\geq t}$. Hence, as in the proof of Lemma 3.6, also $(E,U) \in e_t(C' \dot{+}_{\text{in}} a)$ holds. Further, as $a \notin S$, we have that $S \cup D \subset E \cup U$.

   To show condition (iii), consider an arbitrary $X_{>t}$-restricted admissible tuple (E,U) for $F_{\geq t}$ such that $S \cup D \subset E \cup U$. Then $(E' = E \backslash \{a\}, U' = U \backslash \{a\})$ is $X_{>t'}$-restricted admissible for $F_{\geq t'}$. If $E' = S, U' = D'$ then C' is the unique vcoloring such that $(E',U') \in e_{t'}(C')$. Otherwise if $E' \neq S$ or $U' \neq D'$ it holds that $S' \cup D' \subset E' \cup U'$ and thus, there exists $T' \in \Gamma$ with $(E',U') \in e_{t'}(T')$. Since (E,U) is $X_{>t}$-restricted admissible for $F_{\geq t}$, we have that there is a unique vcoloring T such that $(E,U) \in e_t(T)$.

But then, as in the proof of Lemma 3.6, either $T = C' \dot{+}_{\text{in}} a$, $T = T' +_{\text{attc}} a$, $T = T' \dot{+}_{\text{in}} a$, $T = T' \hat{+}_{\text{out}} a$ or $T = C' +_{\text{attc}} a$ holds.

(2) By the assumption $C = C' \dot{+}_{\text{in}} a$ we have that $(S = S' \cup \{a\}, D = D' \backslash \{a\}) \in e_t(C)$. It remains to show that the vpair (C, Γ) also satisfies conditions (ii) and (iii) of Definition 3.15. To show condition (ii), consider $T \in \Gamma$, i.e., T is of the form $T = T' \dot{+}_{\text{in}} a$ with $[T'] \cup \{a\}$ conflict-free in F_t and $[[T']] = [[T' \dot{+}_{\text{in}} a]]$. Since $(S', D') \in e_{t'}(C', \Gamma')$, there exists $(E', U') \in e_{t'}(T')$ with $S' \cup D' \subset E' \cup U'$. By the construction of T we have that $E = E' \cup \{a\}$ is conflict-free in F_t and thus that $(E, U = U')$ is an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$. By definition of E it holds that $S \cup D \subset E \cup U$ and further, as in the proof of Lemma 3.6, we get that $(E, U) \in e_t(T' \dot{+}_{\text{in}} a)$. To show condition (iii) of Definition 3.15, let (E, U) be an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$ such that $S \cup D \subset E \cup U$. Then $(E' = E \backslash \{a\}, U' = U \backslash \{a\})$ is $X_{>t'}$-restricted admissible for $F_{\geq t'}$ and $(S' = S \backslash \{a\}) \cup (D' = D \backslash \{a\}) \subset E' \cup U'$. Thus, there exists $T' \in \Gamma'$ with $(E', U') \in e_{t'}(T')$. Since (E, U) is $X_{>t}$-restricted admissible, we have that there is a unique vcoloring T such that $(E, U) \in e_t(T)$. But then, as in the proof of Lemma 3.6, $T = T' \dot{+}_{\text{in}} a$ (if $a \in E$) or $T = T' +_{\text{attc}} a$ (if $a \in U$) holds.

(3) As in the proof of Lemma 3.6, $(S = S', D = D' \cup \{a\}) \in e_t(C)$ holds by assumption $(C = C' +_{\text{attc}} a)$. It remains to show that also conditions (ii) and (iii) of Definition 3.15 are fulfilled. To show condition (ii), consider an arbitrary $T \in \Gamma$, i.e. (a) $T = T' +_{\text{attc}} a$ or (b) $T = T' \dot{+}_{\text{in}} a$. Since $(S, D') \in e_{t'}(C', \Gamma')$, there exists $(E', U') \in e_{t'}(T')$ with $S \cup D' \subset E' \cup U'$. In case (a), we follow the proof of Lemma 3.6, and obtain $(E', U' \cup \{a\}) \in e_t(T' +_{\text{attc}} a)$. For case (b), we get by the construction of T that $E = E' \cup \{a\}$ is conflict-free in $F_{\geq t}$. Once more we can use the fact that $X_t \not\rightarrowtail_{\text{attacks}} a$ to obtain that $(E, U = U')$ is an $X_{>t}$-restricted admissible tuple for $F_{\geq t}$. Further $S \cup D \subset E \cup U$ and as in the proof of Lemma 3.6, then also $(E, U) \in e_t(T' \dot{+}_{\text{in}} a)$ holds.

To show condition (iii), consider an arbitrary $X_{>t}$-restricted admissible tuple (E, U) for $F_{\geq t}$ such that $S \cup D \subset E \cup U$. Then $(E' = E \backslash \{a\}, U' = U \backslash \{a\})$ is $X_{>t'}$-restricted admissible for $F_{\geq t'}$. Since $E' \neq S$ or $U' \neq D'$ it holds that $S' \cup D' \subset E' \cup U'$ and thus, there exists $T' \in \Gamma$ with $(E', U') \in e_{t'}(T')$. Since (E, U) is $X_{>t}$-restricted admissible for $F_{\geq t}$, we have that there is a unique vcoloring T such that $(E, U) \in e_t(T)$. But then, as in the proof of Lemma 3.6, either $T = T' \dot{+}_{\text{in}} a$ (if $a \in E$), or $T = T' +_{\text{attc}} a$ (if $a \in U$) holds.

□

Lemma 3.17. *For any JOIN node t in a Tree Decomposition of an AF F, vpairs and valid pairs coincide, if they coincide on the successors t' and t'' of t.*

Proof. Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of $F = (A, R)$ and t a JOIN node in $\mathcal{T}$ with successors t' and t''.

First consider an arbitrary valid pair (C, Γ) for t. We show that (C, Γ) is also a vpair. As (C, Γ) is valid, there exists an $X_{>t}$-restricted admissible tuple (S, D) for $F_{\geq t}$ such that $(S, D) \in e_t(C, \Gamma)$. As in the proof of Lemma 3.9 we have that there exist unique sets

$S_1 \subseteq X_{\geq t'}$ and $S_2 \subseteq X_{\geq t''}$, such that $S_1 \cap X_t = S_2 \cap X_t$ and $S = S_1 \cup S_2$; moreover, there exist $D_1 \subseteq X_{\geq t'}$ and $D_2 \subseteq X_{\geq t''}$ with $D = D_1 \cup D_2$. Further, there exist valid colorings C', C'' such that $(S_1, D_1) \in e_{t'}(C'), (S_2, D_2) \in e_{t''}(C'')$ and $C = C' \bowtie C''$. Thus, by Lemma 3.13 there are valid pairs (C', Γ') and (C'', Γ'') such that $(S_1, D_1) \in e_{t'}(C', \Gamma')$ and $(S_2, D_2) \in e_{t''}(C'', \Gamma'')$. (Note that by Lemma 3.11 there cannot be a pair (C^*, Γ^*) with $(S_1, D_1) \in e_{t'}(C^*, \Gamma^*)$ and $C' \neq C^*$. Analogously, C'' with the above properties is unique.) By assumption these valid pairs are also vpairs.

Now we turn our attention to the set Γ. We first have to show $\Gamma \subseteq \Gamma^*$ with $\Gamma^* = (\Gamma' \bowtie \Gamma'') \cup (\{C'\} \bowtie \Gamma'') \cup (\Gamma' \bowtie \{C''\})$. For every $T \in \Gamma$ there exists an $X_{>t}$-restricted admissible tuple $(E, U) \in e_t(T)$ such that $S \cup D \subset E \cup U$. We define $E = E_1 \cup E_2, U = U_1 \cup U_2$ analogously to S_1, S_2, D_1, D_2. Now we have that $S \cup D \subset E \cup U$ holds iff either

(i) $S_1 \cup D_1 \subset E_1 \cup U_1$ and $S_2 \cup D_2 \subset E_2 \cup U_2$,
(ii) $S_1 = E_1, D_1 = U_1$ and $S_2 \cup D_2 \subset E_2 \cup U_2$ or
(iii) $S_2 = E_2, D_2 = U_2$ and $S_1 \cup D_1 \subset E_1 \cup U_1$

holds. (Observe that $S_i \cup D_i \not\subseteq E_i \cup U_i$ is not possible, since $S_1 \cap X_t = S_2 \cap X_t, D_1 \cap X_t = D_2 \cap X_t, E_1 \cap X_t = E_2 \cap X_t, U_1 \cap X_t = U_2 \cap X_t, S_i \cap D_i = \emptyset, E_i \cap U_i = \emptyset$ and by properties (2) and (3) of Tree Decompositions, i.e. there cannot exist $x \in X_{\geq t'}, X_{\geq t''}$, but $x \notin X_t$.) We discuss these cases separately:

(i) Consider arbitrary $T' \in \Gamma', T'' \in \Gamma'$ with $[T'] = [T''], [[T']] = [[T'']], (E_1, U_1) \in e_{t'}(T')$ and $(E_2, U_2) \in e_{t''}(T'')$. By Definition 3.15 we have that $S_i \cup D_i \subset E_i \cup U_i$. We conclude that $S \cup D \subset E \cup U$ and by the proof of Lemma 3.9 and Lemma 3.10 that $T = T' \bowtie T''$ is the unique coloring such that $(E, U) \in e_t(T)$. Therefore $T' \bowtie T'' \in \Gamma$ and thus $\Gamma' \bowtie \Gamma'' \subseteq \Gamma$.
(ii) Consider an arbitrary $T'' \in \Gamma''$ with $[T'] = [T''], [[T']] = [[T'']]$ and $(E_2, U_2) \in e_{t''}(D'')$. We have that $S \cup D \subset (E = S_1 \cup E_2) \cup (U = D_1 \cup U_2)$ and that $T = \{C'\} \bowtie T''$ is the unique coloring such that $(E, U) \in e_t(T)$. Thus $\{C'\} \bowtie \Gamma'' \subseteq \Gamma$.
(iii) By symmetry to (ii).

Thus we have that $\Gamma \subseteq \Gamma^*$. It remains to show that $\Gamma^* \subseteq \Gamma$ which is equivalent to showing each of the following inclusions:

(i) $\Gamma^* \bowtie \Gamma'' \subseteq \Gamma$,
(ii) $\{C'\} \bowtie \Gamma'' \subseteq \Gamma$ and
(iii) $\Gamma' \bowtie \{C''\} \subseteq \Gamma$.

This can be done as follows:

(i) Consider arbitrary $T' \in \Gamma'$ and $T'' \in \Gamma''$ with $[T'] = [T''], [[T']] = [[T'']], (E_1, U_1) \in e_{t'}(T')$ and $(E_2, U_2) \in e_{t''}(T'')$. By Definition 3.15 we have that $S_1 \cup D_1 \subset E_1 \cup U_1$ and $S_2 \cup D_2 \subset E_2 \cup U_2$. We conclude that $S \cup D \subset E \cup U$ and by the proof of Lemma 3.9 and Lemma 3.11 that $T = T' \bowtie T''$ is the unique coloring such that $(E, U) \in e_t(T)$. Therefore $T' \bowtie T'' \in \Gamma$ and thus $\Gamma' \bowtie \Gamma'' \subseteq \Gamma$.

(ii) Consider an arbitrary $T'' \in \Gamma''$ with $[C'] = [T'']$, $[[C']] = [[T'']]$ and $(E_2, U_2) \in e_{t''}(T'')$. We have that $S \cup D \subset (E = S_1 \cup E_2) \cup U$ and that $T = \{C'\} \bowtie T''$ is the unique coloring such that $(E, U) \in e_t(T)$. Thus $\{C'\} \bowtie \Gamma'' \subseteq \Gamma$.

(iii) By symmetry to (ii).

This shows $\Gamma = \Gamma^*$ and thus every valid pair (C, Γ) is also a vpair.

Now we show that every vpair for t is also a valid pair for t. Thus, let (C, Γ) be a vpair for t, i.e., there exists a vpair (C', Γ') for node t' and a vpair (C'', Γ'') for node t'' with $[C'] = [C'']$ and $[[C']] = [[C'']]$ such that $(C, \Gamma) = (C' \bowtie C'', \Gamma^*)$ (Γ^* defined as above). By assumption (C', Γ') and (C'', Γ'') are valid pairs. Hence, there exist tuples $(S_1, D_1) \in e_{t'}(C', \Gamma')$ and $(S_2, D_2) \in e_{t''}(C'', \Gamma'')$. As in the proof of Lemma 3.9, $(S = S_1 \cup S_2, D = D_1 \cup D_2) \in e_t(C)$ holds since $[C''] = [C']$.

It remains to show that (C, Γ) also fulfills conditions (ii) and (iii) of Definition 3.15. To show condition (ii), consider $T \in \Gamma$, i.e., T is one of the following forms:

(a) $T = T' \bowtie T''$ for some $T' \in \Gamma', T'' \in \Gamma''$ with $[D'] = [D''], [[D']] = [[D'']]$;

(b) $T = C' \bowtie T''$ for some $T'' \in \Gamma''$ with $[C'] = [D''], [[C']] = [[D'']]$;

(c) $T = T' \bowtie C''$ for some $T' \in \Gamma'$ with $[T'] = [C''], [[T']] = [[C'']]$.

We only discuss case (a) here as the cases (b) and (c) are similar: Since $(S_1, D_1) \in e_{t'}(C', \Gamma')$ and $(S_2, D_2) \in e_{t''}(C'', \Gamma'')$, there exist $(E_1, U_1) \in e_{t'}(T')$ and $(E_2, U_2) \in e_{t''}(T'')$ with $S \cup D \subset E_1 \cup U_1, S \cup D \subset E_2 \cup U_2$ and $E_1 \cap X_t = E_2 \cap X_t$. As in the proof of Lemma 3.9, then also $(E = E_1 \cup E_2, U = U_1 \cup U_2) \in e_t(T' \bowtie T'')$ and $S \cup D \subset E \cup U$.

To show condition (iii), let (E, U) be $X_{>t}$-restricted admissible for $F_{\geq t}$ with $S \cup D \subset E \cup U$. Then (E_1, U_1) is $X_{>t'}$-restricted admissible for $F_{\geq t'}$ and (E_2, U_2) is $X_{>t''}$-restricted admissible for $F_{\geq t''}$. Hence there exist sets T' and T'' with $(E_1, U_1) \in e_{t'}(T'), (E_2, U_2) \in e_{t''}(T'')$, $E_1 \cap X_t = E_2 \cap X_t$, and either

(A) $T' \in \Gamma', T'' \in \Gamma''$,

(B) $T' = C', T'' \in \Gamma''$ or

(C) $T' \in \Gamma', T'' = C''$

holds. But then, as in the proof of Lemma 3.9, also $(E = E_1 \cup E_2, U = U_1 \cup U_2) \in e_t(D' \bowtie D'')$. □

With this knowledge, we can show the main theorem of this section in the following (again the structural induction is just sketched).

Theorem 3.2. *Let $(\mathcal{T}, \mathcal{X})$ be a normalized Tree Decomposition of an AF $F = (A, R)$. Then, for each pair (C, Γ) for a node t, it holds that (C, Γ) is a valid pair for t iff (C, Γ) is a vpair for t.*

Proof. As in Theorem 3.1, the proof proceeds by structural induction. For the induction base, we have to show that vpairs and valid pairs coincide on LEAF nodes, which is the case due to Lemma 3.14. For the induction step, we have to show this property for the remaining nodes. Indeed, this is captured by Lemmas 3.15, 3.16 and 3.17. □

The previous theorem and Proposition 3.1 now allow us to compute semi-stable extensions via vpairs. For the root r of some Tree Decomposition $\mathcal{T}$ (w.r.t. some framework $F = (A, R)$), we can now compute our validpairs via vpairs starting from the leaves towards the root of $\mathcal{T}$ in a bottom-up manner. For enumerating $e'_r(\epsilon, \emptyset)$ (i.e. the semi-stable extensions w.r.t. $F_{\geq r} = F$), the determined vpairs of all the nodes of $\mathcal{T}$ are required (compare with Section 2.3 and [ABC+14a]).

3.3 An Adaption for Preferred Semantics

The previous section deals with computing semi-stable sets (as defined in Definition 2.15), that is maximizing (w.r.t. $\subseteq$) the *range*, i.e. the elements in the set and the ones attacked by it, of a given admissible extension S. Using this knowledge, the simpler task of computing preferred extensions (see Definition 2.15), i.e. maximizing only the elements in the set S, can now be obtained easily. For adapting the previous algorithm of semi-stable semantics in order to present an alternative (compared to [DPW12]) algorithm for preferred semantics, we basically need to modify (actually simplify) two definitions, which are Definitions 3.15 and 3.18. The different proofs go through similarly, several cases even collapse.

Definition 3.20. *Let $(\mathcal{T}, \mathcal{X})$ be a Tree Decomposition of an AF F, $t \in \mathcal{T}$, and $(C, \mathcal{T})$ a pair with C being a coloring for t and Γ being a set of colorings for t. We call (C, Γ) simply a prefpair for t and define $e_t^{\texttt{pref}}(C, \Gamma)$ as the collection of tuples (S, D) which satisfy the following conditions (see Definitions 3.15).*

(i) $(S, D) \in e_t(C)$;
(ii) For all $C' \in \Gamma$, there is an $(E, U) \in e_t(C')$ such that $S \subset E$;
(iii) For all $X_{>t}$-restricted admissible (for $F_{\geq t}$) tuples (E, U) with $S \subset E$, there exists some $C' \in \Gamma$ with $(E, U) \in e_t(C')$.

If $e_t^{\texttt{pref}}(C, \Gamma) \neq \emptyset$, (C, Γ) is a valid prefpair for t.

Definition 3.21. *Let $(\mathcal{T}, \mathcal{X})$ be a normalized Tree Decomposition of an AF F and let $t \in \mathcal{T}$ be a node with t', t'' its possible children. Depending on the node type of t we define a vprefpair for t as follows (see Definitions 3.18).*

- *LEAF: Each (C, Γ) where $C \in \mathcal{C}_t$ and $\Gamma = \{C' \in \mathcal{C}_t \mid [C] \subset [C']\}$ is a vprefpair for t.*
- *FORGET: If (C', Γ') is a vprefpair for t', $X_t = X_t \backslash \{a\}$, and $C'(a) \neq attc$, then $(C' - a, \Gamma' - a)$ is a vprefpair for t.*
- *INSERT: If (C', Γ') is a vprefpair for t' and $X_t = X_{t'} \cup \{a\}$, and if $C' \dot{+}_{in} a$ is a vcoloring then $(C' \dot{+}_{in} a, \Gamma' \dot{+}_{in} a)$ is a vprefpair for t; if moreover $C' \hat{+}_{out} a$ is a vcoloring, then $(C' \hat{+}_{out} a, (\{C'\} \dot{+}_{in} a) \cup (\Gamma' +_{attc} a) \cup (\Gamma' \dot{+}_{in} a) \cup (\Gamma' \hat{+}_{out} a))$ is*

a vprefpair for t; $(C' +_{attc} a, (\Gamma' +_{attc} a) \cup (\Gamma' \dot{+}_{in} a) \cup (\Gamma' \hat{+}_{out} a))$ *is a vprefpair for* t*.*

- *JOIN: If* (C', Γ') *is a vprefpair for* t'*,* (C'', Γ'') *is a vprefpair for* t''*,* $[C'] = [C'']$ *and* $[[C']] = [[C'']]$*, then* $(C' \bowtie C'', (\Gamma' \bowtie \Gamma'') \cup (\{C'\} \bowtie \Gamma'') \cup (\Gamma' \bowtie \{C''\}))$ *is a vprefpair for* t*.*

Example 3.22. *Figure 3.4 shows the computation of the vprefpairs for* F_{Ex} *of Figure 3.1 w.r.t. Tree Decomposition shown in Figure 3.1.*

In addition to Figure 3.2, the additional column Γ *of the table of every Tree Decomposition node uses column ID and represents (strict) counter candidates. Observe that by following the extension pointers, we get that preferred*$(A_{F_{Ex}}) = \{\{v, w\}\}$ *(compare to Example 2.16).*

If one compares now Figure 3.4 with Figure 3.3, where the goal is to compute admissible sets of maximum range (more involved optimization criteria), one can – due to the overall potentially smaller cardinalities of Γ *and reduced number of lines per table – observe that computing preferred extensions seems to be simpler than computing semi-stable ones.*

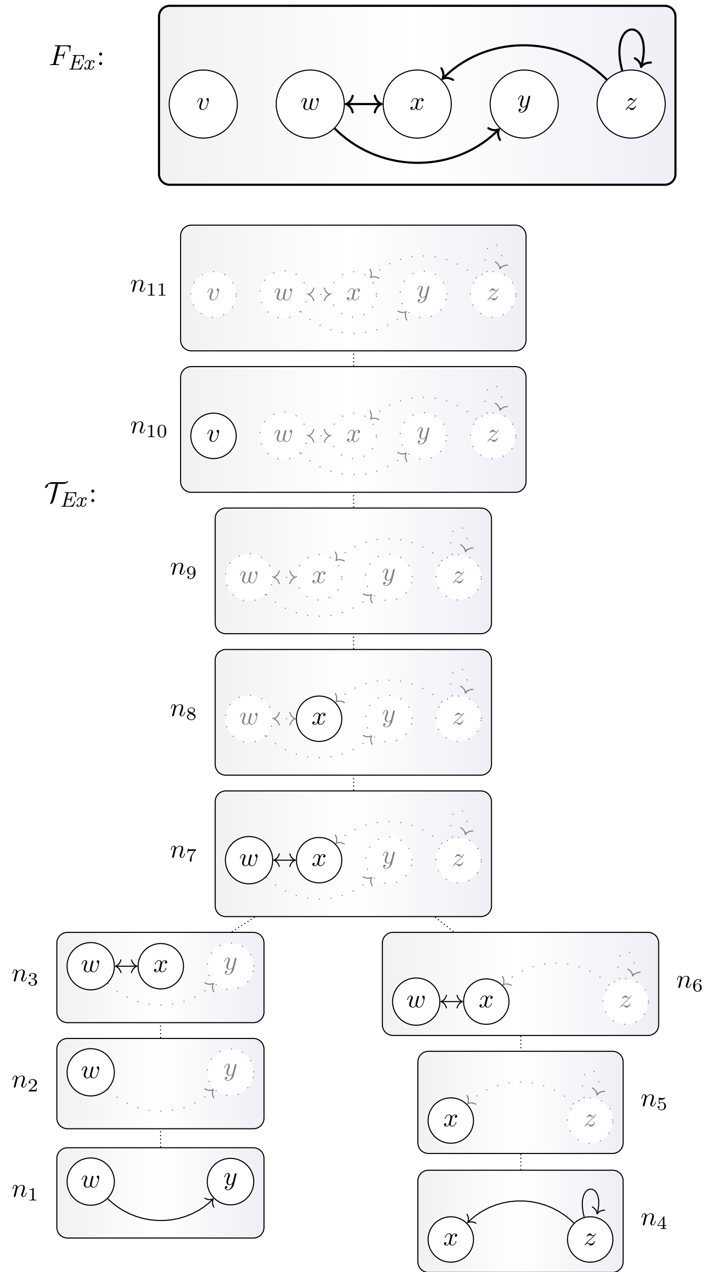

Figure 3.1: Instance $F_{Ex} = (A_{F_{Ex}}, R_{F_{Ex}})$ and a normalized TD $\mathcal{T}_{Ex}$.

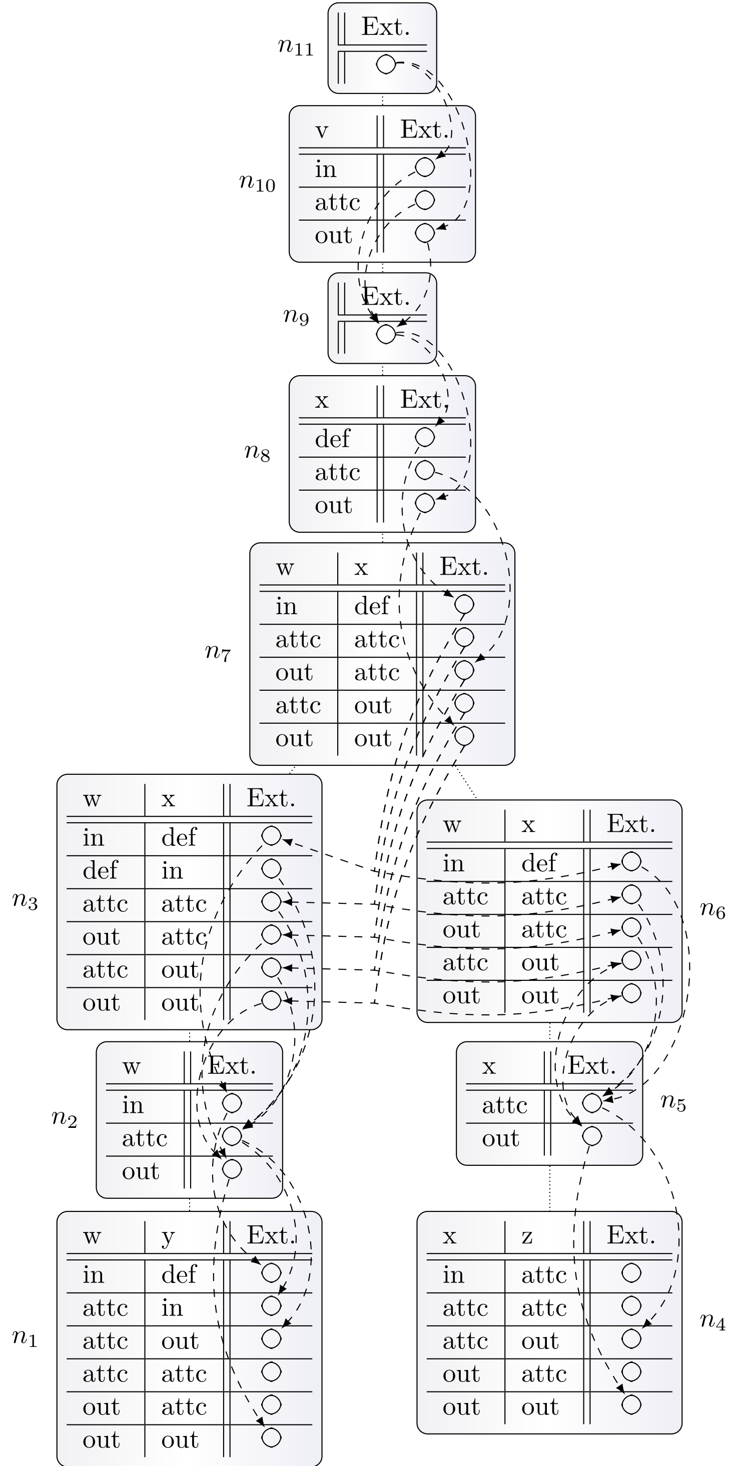

Figure 3.2: Computation of vcolorings for $F_{Ex} = (A_{F_{Ex}}, R_{F_{Ex}})$ w.r.t. $\mathcal{T}_{Ex}$ (see Figure 3.1).

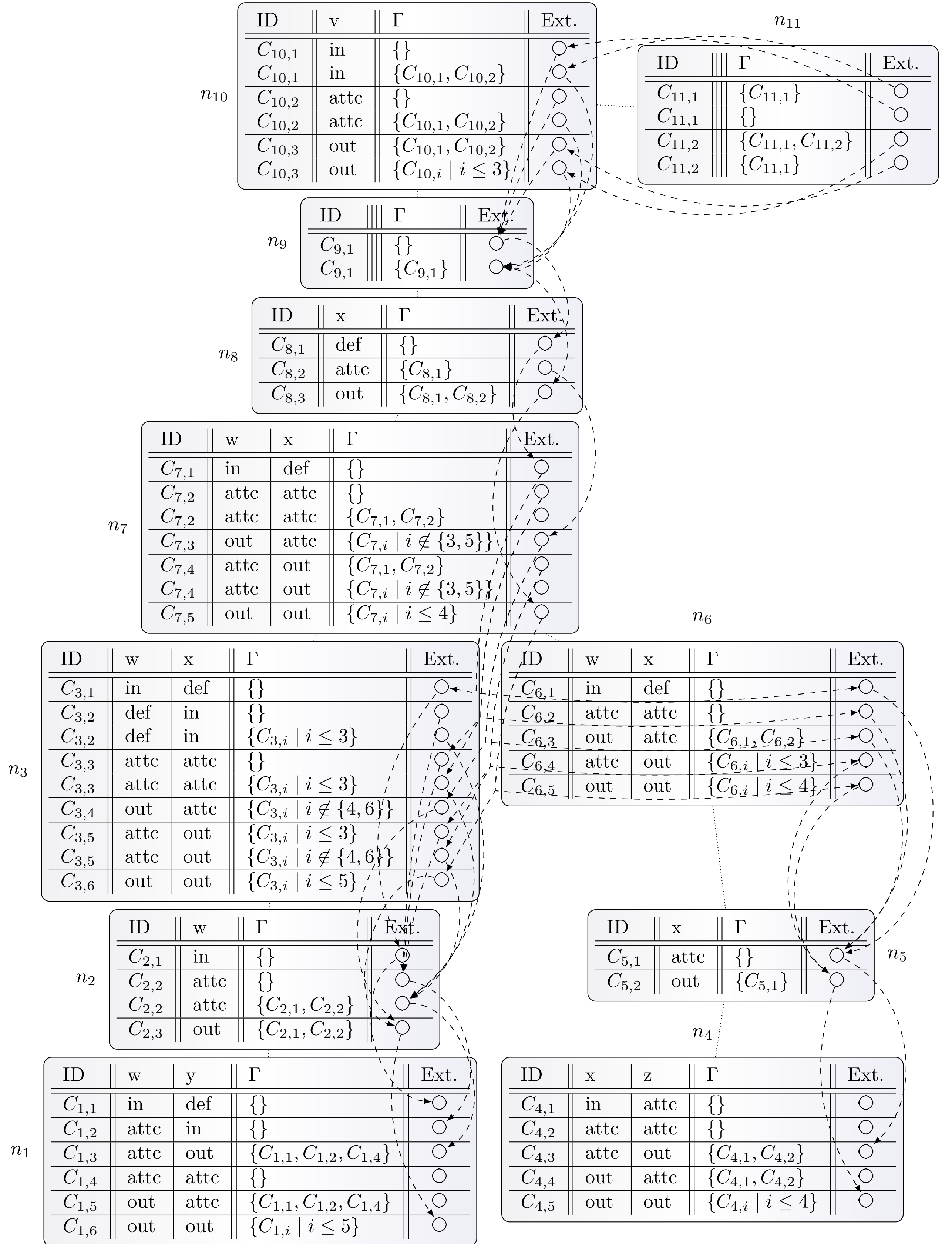

Figure 3.3: Computation of vpairs for $F_{Ex} = (A_{F_{Ex}}, R_{F_{Ex}})$ w.r.t. $\mathcal{T}_{Ex}$ (see Figure 3.1).

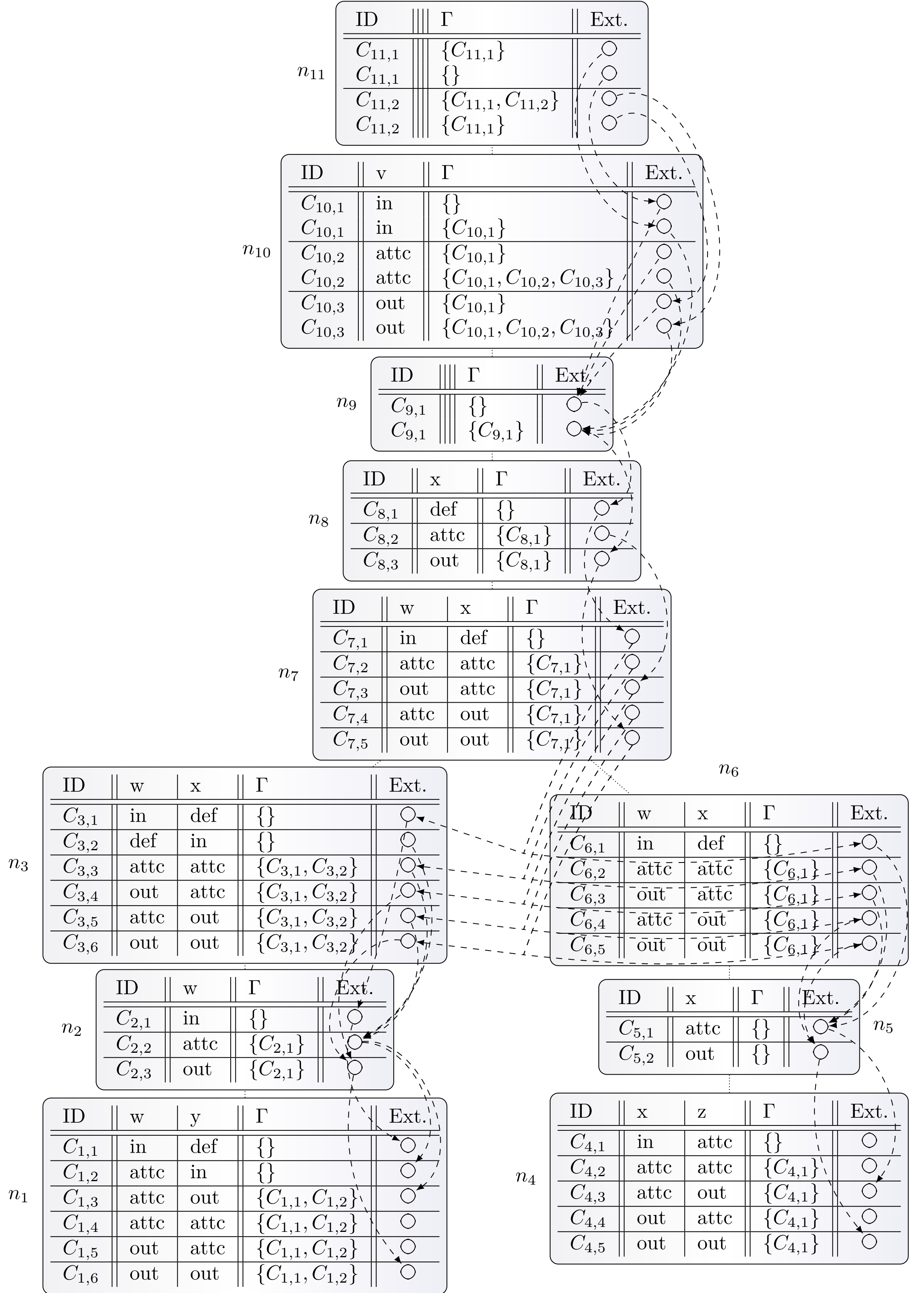

Figure 3.4: Computation of vprefpairs for $F_{Ex} = (A_{F_{Ex}}, R_{F_{Ex}})$ w.r.t. T_{Ex} (see Figure 3.1).

CHAPTER 4

Towards Optimization of DP algorithms on TDs

4.1 D-FLAT: DP on TDs

D-FLAT is a system for rapid prototyping of DP algorithms specified in ASP, where the user only has to develop a problem-specific ASP encoding that is executed at each node of the TD.

In this section we explain the concept of DP on Tree Decompositions (TDs) as realized in the D-FLAT system. We highlight the concepts on basis of enumeration variants for the SAT and ⊆-MINIMAL SAT problems. These problems are well suited since the DP algorithms incorporate concepts that often reappear in other AI-related problem domains. In general, given some problem that is tractable for bounded tree-width (such as SAT) and an input instance (e.g., some formula ϕ), DP on TDs consists of the following steps:

1. The input instance is decomposed, thereby obtaining a TD. Each node in the TD represents parts of the original instance (e.g., some atoms and clauses of ϕ).
2. The TD is traversed in post-order. At each TD node, partial solutions are computed (e.g., partial interpretations of ϕ).
3. In order to enumerate the solutions for the whole problem instance (e.g., the models of ϕ), the TD is traversed a second time where the partial solutions are combined.

Usually, an algorithm designer has to implement these three steps for every problem from scratch. D-FLAT is a system for rapid prototyping of DP algorithms. Here, the user only has to develop a problem-specific ASP encoding that defines how the partial solutions are constructed. This encoding will be invoked once for every node during a post-order traversal of the TD, and its models specify the partial solutions at the respective node. Communication between D-FLAT and the encoding is implemented via an interface

consisting of pre-defined predicates. Overall, when D-FLAT is called together with the encoding and an input instance, it internally executes the steps described above and returns the solution.

4.1.1 System Overview

D-FLAT[1] uses Dynamic Programming on TDs, which are generated by the Hypertree Decomposition library *Htdecomp* [DGG+08]. As it is an application of Fixed-Parameter Tractability with tree-width as the parameter, its performance highly depends on the generated decomposition; to be more precise, on its width. It is therefore vital to generate decompositions with a rather small width, which comes very close to the tree-width of the input graph.

The figures and the content basis of this subsection are taken from [Bli12, ABC+14a]. The control flow of D-FLAT is as seen in Figure 4.1 and in the following section explained briefly. For more details we refer to [Bli12, ABC+14a]. First of all, the input instance is parsed and prepared (graph representation is created) for Htdecomp, which tries to generate a Tree Decomposition of small width. Then a bottom-up traversal is performed to compute the tables of the TD nodes. A table is a data representation containing rows and is realized as an item tree of height 1 (see Section 4.1). Each row contains for each node of the bag the desired mapped information and extension information of its child nodes. Depending on the problem type, it is not always trivial to figure out, what information should be mapped to a particular node in order to get the complete solutions out of the partial solution candidates during the tree traversal.

The data representation can also be done in a way using more than one level, which was designed particularly for problems in some complexity class beyond *NP* of the PH. For this D-FLAT uses so-called item trees (see Section 2.3) of higher depth than 1; in particular for problems on the 2nd level of PH, item trees of depth 2 are used. Second-Level encodings allow each item set (often a solution candidate) at level one to manage its own children (items) at level two, which typically are used to store counter candidates. As the number of levels increases, it is even possible to define algorithms with i levels – D-FLAT internally uses item trees of depth i to solve this i^{th}-Level algorithm.

At the root node of the decomposition, i.e., at the end of the bottom-up traversal, the remaining rows of its table represent the solutions to the given problem instance – every row represents many solutions in general. By traversing the stored inheritance information from any row at the root node back to the bottom of the decomposition, D-FLAT is able to combine the mapped information in order to show all solutions.

Within a node during the bottom-up traversal, D-FLAT flattens child tables, which means it builds a set of facts describing the content of the table of each child node. In order to compute the new table, D-FLAT invokes the integrated Answer-Set Solver with the problem instance, the Dynamic Programming algorithm (the user program), a set of

[1] D-FLAT is an acronym for *Dynamic Programming Framework with Local Execution of ASP on Tree Decompositions* and it is publicly available for free at `http://www.dbai.tuwien.ac.at/proj/dynasp/dflat/`

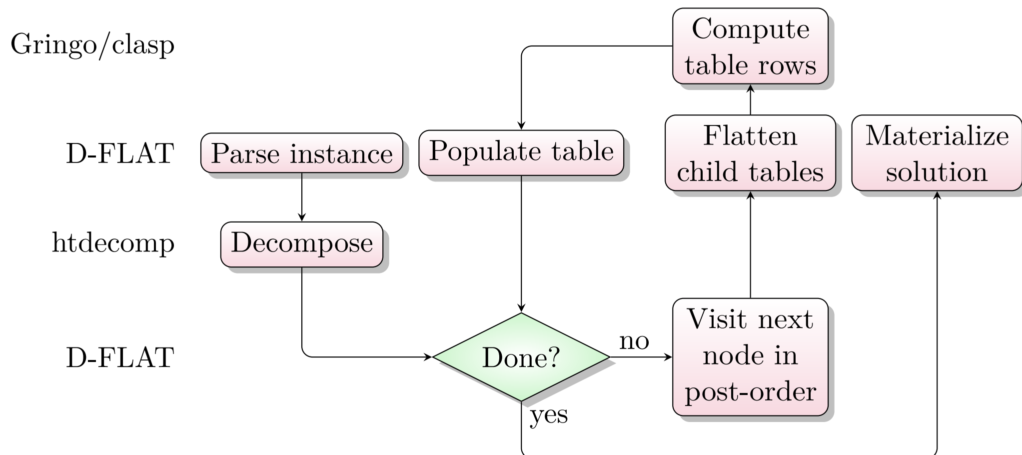

Figure 4.1: Flowchart that shows how D-FLAT and its components work [Bli12, ABC+14a].

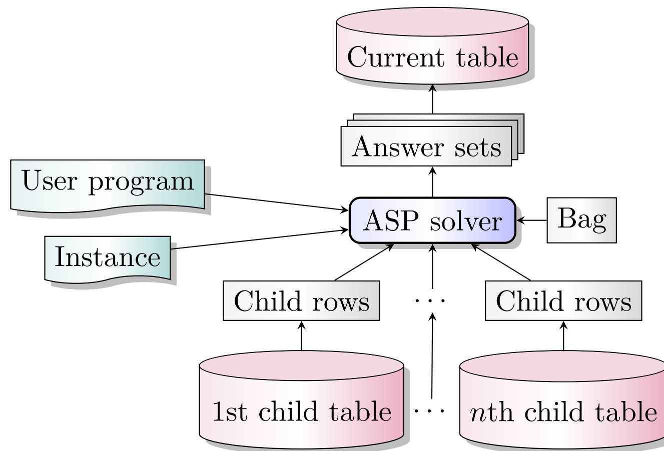

Figure 4.2: Data flow while processing a node with n children [Bli12, ABC+14a].

facts for each child node and a description of its bag contents and a description of the current bag as a set of facts, stating which vertices are present, as shown in Figure 4.2.

4.1.2 Technical Details

In this section we explain the concept of Dynamic Programming (DP) on Tree Decompositions (TDs) as realized in the D-FLAT system. In the following we explain the individual steps of D-FLAT based on the enumeration variants of the Sat and ⊆-Minimal Sat problems. We first describe how the input for our running example (Sat) can be represented in D-FLAT. Remember that in Section 2.2 we introduced TDs formally, Section 4.1.2.1 describes how partial solutions are represented and how the

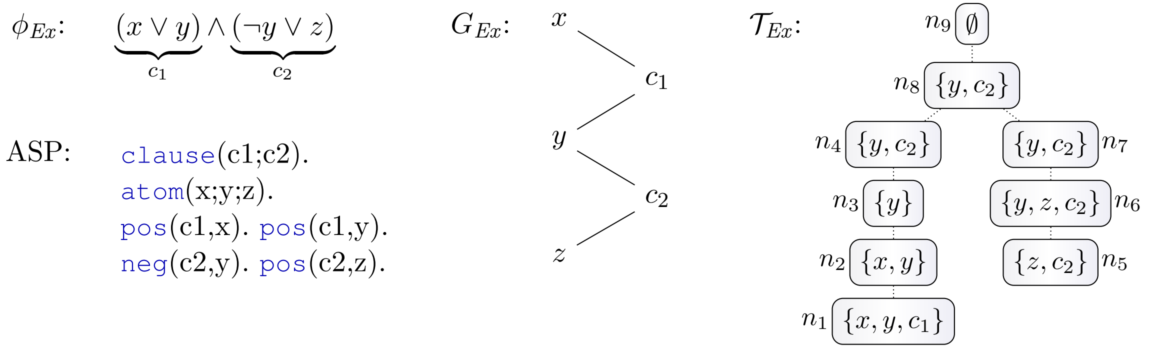

Figure 4.3: Instance ϕ_{Ex}, ASP representation, incidence graph G_{Ex} and a semi-normalized TD $\mathcal{T}_{Ex}$.

Input predicate	Meaning
final	The current Tree Decomposition node is the root.
childNode(N)	N is a child of the current decomposition node.
bag(N, V)	V is contained in the bag of the decomposition node N.
current(V)	V is an element of the current bag.
introduced(V)	V is a current vertex but was in no child node's bag.
removed(V)	V was in a child node's bag but is not in the current one.

Table 4.1: Input predicates describing the Tree Decomposition.

problem-specific encoding can be written. Finally, in Section 4.1.2.2 we outline how the partial solutions are combined.

Input representation In order to construct a TD, the input has to be specified in form of a graph. For SAT, we consider the *incidence graph* $G = (V, E)$ of ϕ, where V is the set of clauses and atoms occurring in ϕ, given as ASP facts clause($\cdot$) and atom($\cdot$) respectively. E is given as facts pos(c, a) (neg(c, a)) that denote that some atom a occurs positively (negatively) in clause c. An example is given in Figure 4.3.

Since constructing a TD with minimum width is intractable, D-FLAT relies on polynomial-time heuristics.

D-FLAT transforms a TD into a semi-normalized one in linear time without increasing the width. A possible TD $\mathcal{T}_{Ex}$ of our example formula ϕ_{Ex} is depicted in Figure 4.3. The width of $\mathcal{T}_{Ex}$ is 2. When traversing the Tree Decomposition, D-FLAT provides information about the current TD node and its child node(s) via the predicates described in Table 4.1.

4.1.2.1 Data Representation and Algorithm Execution

D-FLAT traverses the decomposition in post-order. At each DP node the partial solutions are computed. Here, an ASP solver (Gringo 4.4.0, Clasp 3.1.1) is called with the following input:

1. the user-specified, problem-specific ASP encoding,
2. the input instance,
3. information about the current and child TD node(s) (see above), and
4. the partial solutions computed in the child TD node(s) (described in this section).

The models returned by the solver represent partial solutions of the current node.[2] A partial solution is a solution to the problem restricted to the subgraph induced by the vertices encountered so far. In D-FLAT each node is associated with a data structure that stores the partial solutions. For problems in NP *tables* are employed, harder problems can be solved using *item trees* [ABC$^+$14a] as described in Section 2.3.

In the following, encodings for D-FLAT covering SAT and ⊆-MINIMAL SAT are given. Later on in Chapter 4.1.3, we will see encodings for certain semantics of Abstract Argumentation. These encodings are based on the work on Tree Decomposition algorithms in the thesis [Cha12] and are all based on the Guess & Check [EP06] paradigm. A basic approach on how to start writing encodings for D-FLAT is presented in Chapter 3.2 of [Bli12], whereas Chapter 4 of the same document provides good documentation of D-FLAT and a set of selected case studies. More details about encodings for the current version of D-FLAT can be found in [ABC$^+$14a], in particular there is an updated and extended version of case studies for the current version of D-FLAT in Chapter 4 of [ABC$^+$14a]. Chapter 5 of [ABC$^+$14a] introduces a debugger for D-FLAT designed for support in finding flaws in complicated encodings. The given programs make use of some language features of the grounder Gringo. More details can be found in the guide [GKK$^+$11].

Data Representation (tables) Each row in a table represents many partial solutions to the problem. When traversing the TD, tables for the already-visited child nodes are given to the user-specified encoding via predicates as listed in Table 4.2. Then, the partial solutions for the current TD node are computed via the user-specified encoding, and returned via the output predicates listed in Table 4.3. Each row consists of a set of *items* that are arbitrary ground ASP terms. While items are typically used to store information that has to coincide in the two children of a join node (e.g., the truth assignment of atoms in the SAT problem), auxiliary items store information that does not need to coincide (e.g., whether a clause becomes satisfied). Following this rule, D-FLAT in particular provides a default implementation for join nodes (called default join) in case of semi-normalized TDs (and several other ones, see [ABC$^+$14a]). We will explain this distinction in detail

[2] We use colors to highlight input (red), output (orange) and input instance (blue) predicates.

Input predicate	Meaning
`childRow`(R, N)	R is a table row belonging to decomposition node N.
`childItem`(R, I)	The item set of table row R contains I.
`childAuxItem`(R, I)	The auxiliary item set of table row R contains I.

Table 4.2: Input predicates describing tables of decomposition child nodes.

throughout the following example for SAT. The output predicate `extend`/1 specifies the child row(s) that give rise to the partial solution encoded by the respective model.

Listing 4.1 gives an example of a user-specified ASP encoding Π_{SAT} for the SAT problem. The encoding makes use of D-FLAT's input interface (see Tables 4.1 and 4.2) in the bodies of the rules, and the output interface (see Table 4.3) in the heads of the rules. The computed partial solutions for our running example are depicted in Figure 2.2. We will now go through Listing 4.1 and our example in detail. Let us first consider leaf node n_1 of $\mathcal{T}_{Ex}$. Since n_1 has no children, only Lines 9, 22 and 23 are of interest to us. For atoms x and y in $\chi(n_1) = \{x, y, c_1\}$ we guess their truth assignment (Line 9). In case an atom gets assigned true, it is added to the `item` set of the computed row. Lines 22 and 23 denote that a clause is added to the `auxItem` set in case it is satisfied by the current truth assignment. In node n_1 of Figure 2.2, we thus have four partial solutions ("rows"), namely $\{\emptyset, \{x, c_1\}, \{y, c_1\}, \{x, y, c_1\}\}$. In n_2, c_1 is removed from the bag. Whenever an atom was assigned false in a child row, Line 2 makes this explicit, and Line 3 identifies clauses that have not been satisfied yet. We now `extend` each partial solution of the child node (Line 6). Partial solution $\emptyset$ is not extended, since it does not satisfy c_1 (Line 15). The other partial solutions are extended and their information, restricted to the current bag in case one requires FPT, is kept via Lines 18 and 19. In Figure 2.2, this extension is marked with dashed arrows. In join nodes, we extend exactly one row per child table at a time (Line 6). Extended partial solutions have to agree on the truth assignment of atoms in the current bag (Line 12). Consider join node n_8. Here, the row containing partial solution $\{y, c_2\}$ extends $\{y\}$ in n_4 and $\{y, c_2\}$ in n_7, since the latter two both contain the same truth assignment. This also highlights the difference between `item` and `auxItem`: In the former, common items have to be contained in both extended partial solutions (they have to agree on the truth assignment of atoms) and a union over the items is built (required for non-common items); in the latter only a union over the auxiliary items is built (i.e., a clause that is satisfied in one subgraph is also satisfied in the combined subgraph).

```
1 % Define false atoms and unsatisfied clauses
2 f(R,X)      ← childRow(R,N), bag(N,X), not childItem(R,X).
3 unsat(R,C) ← childRow(R,N), bag(N,C), not childAuxItem(R,C).

5 % Guess partial solutions to be extended
6 1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
```

Output predicate	Meaning
item(I)	The item set of the current table row shall contain the item I.
auxItem(I)	The auxiliary item set of the current table row shall contain the item I.
extend(R)	The current table row shall extend the child table row R.

Table 4.3: Output predicates for constructing the table of the current decomposition node.

```
8  % Guess truth value of introduced atoms
9  { item(A) : atom(A), introduced(A) }.

11 % Only join rows coinciding on truth values of atoms
12 ← extend(X;Y), atom(A), childItem(X,A), f(Y,A).

14 % Rows with unsatisfied, removed clauses are not extended
15 ← clause(C), removed(C), extend(R), unsat(R,C).

17 % True atoms and satisfied clauses are kept
18 item(X)    ← extend(R), childItem(R,X),    current(X).
19 auxItem(C) ← extend(R), childAuxItem(R,C), current(C).

21 % Through guess clauses may become satisfied
22 auxItem(C) ← current(C;A), pos(C,A),     item(A).
23 auxItem(C) ← current(C;A), neg(C,A), not item(A).
```

Listing 4.1: Π_{SAT}: D-FLAT encoding for solving SAT.

Note that instead of the incidence graph (see Figure 4.3), we could also for instance specify a different form of input graph, the so-called *primal graph* (see Figure 4.4), where every atom of some clause c is connected to every other atom of c (i.e. the atoms of a clause form a clique). Observe that, by the properties of TDs (see Chapter 2), it is thus required that there has to occur some bag containing all the atoms of c. So, when the node whose bag contains all the atoms of some clause c is processed, one can decide about the satisfiability of c (in this case we say that c is implicitly contained in the current bag). Listing 4.2 should look familiar if compared to Listing 4.1, in fact the main difference has to do with predicate curr/1, which marks whether a clause is implicitly in the current bag (and one can therefore decide about its satisfiability). Lines 17 and 18 deal with marking the current clause, while Line 25 of Listing 4.2 handles unsatisfied clauses. As a final note, one has to keep in mind here that the only edge type of the input graph is occurTogether/1 and that the vertices are just the atoms of the input formula.

```
1 % Define false atoms
2 f(R,X) ← childRow(R,N), bag(N,X), not childItem(R,X).
```

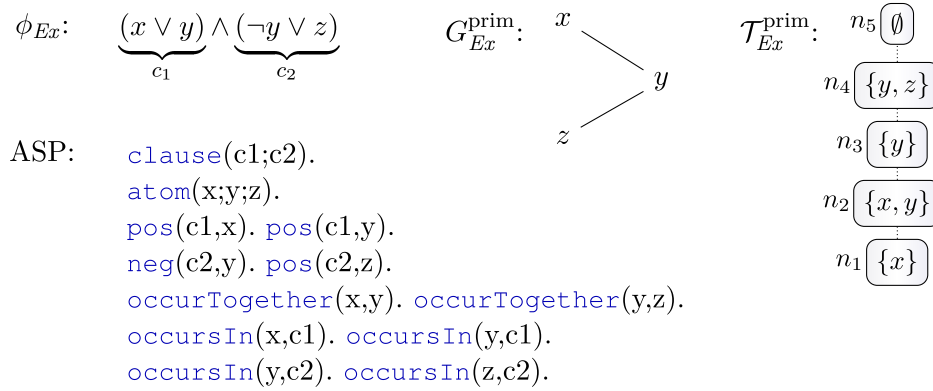

Figure 4.4: Instance ϕ_{Ex}, ASP representation, primal graph G^{prim}_{Ex} and a semi-normalized TD $\mathcal{T}^{\text{prim}}_{Ex}$.

```
4  % Guess partial solutions to be extended
5  1 { extend(R) : childRow(R,N) } 1 ← childNode(N).

7  % Guess truth value of introduced atoms
8  { item(A) : introduced(A) }.

10 % Only join rows coinciding on truth values of atoms
11 ← extend(X;Y), childItem(X,A), f(Y,A).

13 % True atoms are kept
14 item(X) ← extend(R), childItem(R,X), current(X).

16 % Current clauses
17 notCurr(C) ← occursIn(A,C), not current(A).
18 curr(C)    ← clause(C),     not notCurr(C).

20 % Define satisfied clauses
21 sat(C) ← curr(C), pos(C,A),     item(A).
22 sat(C) ← curr(C), neg(C,A), not item(A).

24 % Current clauses need to be satisfied
25 ← curr(C), not sat(C).
```

Listing 4.2: $\Pi^{\text{prim}}_{\text{SAT}}$: D-FLAT encoding for solving SAT using primal graphs.

Data Representation (item trees) For problems that are harder than NP, D-FLAT provides *item trees* to store partial solutions. A brief overview is given in this section, for details we refer to [ABC+14a]. The predicates specifying item trees computed in the child nodes are given in Table 4.4, and the output predicates are listed in Table 4.5.

Similar to before, each item tree node contains an `item`/2 and an `auxItem`/2 set. Again, the information stored in the items is restricted to (or dependent on) the bag elements of the respective decomposition node in case one requires FPT. Each item tree node additionally has a set of *extension pointer tuples* that represents its origin, denoted by `extend`/2. (Note that these are now binary predicates.) Each root-to-leaf path has a particular `length`/1, and the *level* of an item tree node is its depth on the path. At the TD's root, each item tree node must be labeled with either `accept` or `reject` if it is a leaf, otherwise with `or`/1 or `and`/1. D-FLAT uses this to filter out solution candidates for which "counterexamples" exist: Only so-called *accepting* nodes are kept at the TD root, where a node is accepting if

1. its label is `accept`, or
2. its label is `or` and at least one child is accepting, or
3. its label is `and` and all children are accepting.

One can view the table-based data structure as a special case of item trees, where the `length` of each root-to-leaf path is one, and the root node is of type `or`. Furthermore, contrary to tables, where each model returned by the ASP solver represents a row, for item trees a model represents a single root-to-leaf path in the item tree.

Now we will explain item trees on basis of the ⊆-MINIMAL SAT problem. Conceptually, we store solution candidates at depth 1 of the item trees (similar to partial solutions stored in the rows for SAT). Furthermore, we store so-called *counter candidates* at depth 2. A counter candidate is a potential witness for the solution candidate (its parent) not being subset-minimal (cf., e.g., [JPW09]). This concept of storing witnesses (also called *certificates*) is commonly used for problems that are hard for the second level of the polynomial hierarchy [JPW09]. In D-FLAT, it is again only required to specify a single ASP encoding, depicted in Listing 4.3. The encoding defines that item sets for both solution and counter candidates are computed as in the SAT problem. Additionally, this encoding ensures that partial interpretations represented by counter candidates are strict subsets of partial interpretations represented by solution candidates. Line 1 states that we have an item tree of depth 2. Furthermore, the encoding is designed to only return a solution in case there exists some (`or`(0)) solution candidate at level 1, such that no (`and`(1) in combination with Line 36) smaller counter candidate at level 2 exists. Similar to Π_{SAT}, Lines 3-7 make explicit if an atom is false or a clause is unsatisfied in an item tree node. Exactly one root-to-leaf path of each child item tree is extended (Lines 10-11). Then, for each level it is guessed whether an introduced atom is contained in the interpretation represented by the item set (Line 14), root-to-leaf paths are only joined in case the respective item sets coincide on their truth assignments for current atoms (Line 17), item sets with unsatisfied removed clauses are not extended (Line 20), truth assignments are propagated (Lines 23-24) and the set of satisfied clauses is updated (Lines 27-28). Line 31 guarantees that the interpretation in a counter candidate is a subset of (or equal to) that of a solution candidate. Then, flag `smaller` denotes that the counter candidate represents a proper subset of the solution candidate (Lines 34-35). In

Input predicate	Meaning
`atNode`(S, N)	S is an item tree node belonging to decomposition node N.
`rootOf`(S, N)	S is the root of the item tree at decomposition node N.
`sub`(R, S)	R is an item tree node with child S.
`childItem`(S, I)	The item set of item tree node S contains item I.
`childAuxItem`(S, I)	The auxiliary item set of item tree node S contains item I.

Table 4.4: Input predicates describing item trees of child nodes in the decomposition.

Output predicate	Meaning
`item`(L, I)	I is in the item set of the node at level L in the current root-to-leaf path.
`auxItem`(L, I)	I is in the auxiliary item set at level L in the current root-to-leaf path.
`extend`(L, S)	Node at level L in current root-to-leaf path extends child item tree node S.
`length`(L)	The current root-to-leaf path has length L.
`or`(L)/`and`(L)	The node at level L in the current root-to-leaf path has type “or”/“and”.
`accept`/`reject`	The leaf in the current root-to-leaf path has type “accept”/“reject”.

Table 4.5: Output predicates for constructing the item tree of the current decomposition node.

the final (i.e., root) node of the TD, solution candidates are rejected that still have a smaller counter candidate, and accepted otherwise (Lines 36-37). Figure 2.3 contains the computed item trees for TD nodes n_1, n_2 and n_3 of our running example. In n_1, we construct the item sets at depth 1 as described for SAT. The item sets at depth 2 represent counter candidates that are strict subsets of the interpretations at depth 1 (note that we omit counter candidates without `smaller` here). In n_2, clause c_1 is removed. Hence, we remove all item sets representing interpretations that do not satisfy c_1. In n_3, atom x is removed. This results in two item sets at depth 1 that both solely contain y. However, they differ in the counter candidates stored at depth 2.

Observe that Listing 4.3 is quite similar to Listing 4.1. However, in Listing 4.3 the counter candidates at the second level have to be handled, which, in turn, is done similarly to the solution candidates at the first level. In D-FLAT^2, we address this repetition of code by abstracting away the minimization task from the user. As we will see, this allows one to reuse Listing 4.1 for solving $\subseteq$-MINIMAL SAT by solely specifying the set of items to be minimized upon (see Section 4.2.1).

```
1 length(2). or(0). and(1).

3 childBag(S,X) ← atNode(S,N), sub(_,S), childNode(N), bag(N,X).

5 % Define false atoms and unsatisfied clauses
6 f(S,X)      ← childBag(S,X), not childItem(S,X).
7 unsat(S,C) ← childBag(S,C), not childAuxItem(S,C).

9 % Guess root-to-leaf paths in item trees to be extended
10 extend(0,R) ← root(R).
11 1 { extend(L+1,S) :  sub(R,S) } 1 ← extend(L,R), L<2.

13 % Guess truth value of introduced atoms
14 { item(2,A;1,A) : atom(A), introduced(A) }.

16 % Only join root-to-leaf paths coinciding on atom truth values
17 ← extend(L,X;L,Y), atom(A), childItem(X,A), f(Y,A), L=1..2.

19 % Paths with unsatisfied, removed clauses are not extended
20 ← extend(L,R), clause(C), removed(C), unsat(R,C), L=1..2.

22 % True atoms and satisfied clauses are kept
23 item(L,X)    ← extend(L,R), childItem(R,X),    current(X), L=1..2.
24 auxItem(L,C) ← extend(L,R), childAuxItem(R,C), current(C), L=1..2.

26 % Through guess, clauses may become satisfied
27 auxItem(L,C) ← current(C;A), pos(C,A),     item(L,A), L=1..2.
28 auxItem(L,C) ← current(C;A), neg(C,A), not item(L,A), L=1..2.

30 % Interpretation at level 2 must be subset of that at level 1
31 ← atom(A), item(2,A), not item(1,A).

33 % Update subset information; reject larger models at root
34 auxItem(2,smaller) ← extend(2,S), childAuxItem(S,smaller).
35 auxItem(2,smaller) ← atom(A), item(1,A), not item(2,A).
36 reject ← final, auxItem(2,smaller).
37 accept ← final, not reject.
```

Listing 4.3: $\Pi_{\subseteq\text{-MINIMAL SAT}}$: D-FLAT encoding for solving ⊆-MINIMAL SAT.

Similar to the previous data representation (tables), one can also state a different encoding for ⊆-MINIMAL SAT using primal graphs (see Listing 4.4).

```
1 length(2). or(0). and(1).

3 % Define false atoms
4 f(S,X) ← atNode(S,N), sub(_,S), childNode(N), bag(N,X), not
     childItem(S,X).
```

```
 6 % Guess root-to-leaf paths in item trees to be extended
 7 extend(0,R) ← root(R).
 8 1 { extend(L+1,S) :  sub(R,S) } 1 ← extend(L,R), L<2.

10 % Guess truth value of introduced atoms
11 { item(2,A;1,A) : introduced(A) }.

13 % Only join root-to-leaf paths coinciding on atom truth values
14 ← extend(L,X;L,Y), childItem(X,A), f(Y,A), L=1..2.

16 % Current clauses
17 notCurr(C) ← occursIn(A,C), not current(A).
18 curr(C)    ← clause(C),      not notCurr(C).

20 % Define satisfied clauses
21 sat(L,C) ← curr(C), pos(C,A),     item(L,A), L=1..2.
22 sat(L,C) ← curr(C), neg(C,A), not item(L,A), L=1..2.

24 % Current clauses need to be satisfied
25 ← curr(C), not sat(L,C), L=1..2.

27 % True atoms are kept
28 item(L,X) ← extend(L,R), childItem(R,X), current(X), L=1..2.

30 % Interpretation at level 2 must be subset of that at level 1
31 ← item(2,A), not item(1,A).

33 % Update subset information; reject larger models at root
34 auxItem(2,smaller) ← extend(2,S), childAuxItem(S,smaller).
35 auxItem(2,smaller) ← item(1,A), not item(2,A).
36 reject ← final, auxItem(2,smaller).
37 accept ← final, not reject.
```

Listing 4.4: $\Pi^{\text{prim}}_{\subseteq\text{-Minimal Sat}}$: D-FLAT encoding for solving $\subseteq$-Minimal Sat using primal graphs.

4.1.2.2 Obtaining Solutions

At the TD's root, from the properties of Tree Decompositions we know that the whole instance has been taken into account. Typically, for decision problems (e.g., satisfiability, credulous, or skeptical reasoning) the result is directly available at the root node. For enumeration tasks the tree is traversed a second time (now in pre-order) and the partial solutions associated with each Tree Decomposition node are combined in order to obtain the complete solutions (see Definition 2.5 for formal means). Here, we follow the extension pointer tuples while combining the contents of the respective item sets. For our running example, considering Sat, we have $\{\{x, c_1, c_2\}, \{x, z, c_1, c_2\}, \{y, z, c_1, c_2\}, \{x, y, z, c_1, c_2\}\}$.

The models of ϕ_{Ex} are $\{\{x\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$, and the subset-minimal models are $\{\{x\}, \{y, z\}\}$.

4.1.3 D-FLAT Encodings for Abstract Argumentation

The following section contains encodings for Abstract Argumentation using D-FLAT, in particular we will show encodings for stable, admissible, complete, preferred and semi-stable extensions.

For the presented encodings for Abstract Argumentation, it is assumed that the input instances model the attack relation by using `att`/2.

Further, the encodings have in common that in order to deal with the problem of selecting sets $S \subseteq A$ with the desired properties of a given argumentation framework $F = (A, R)$, they map colors to nodes $n \in A$ in order to remember information during the traversal of the generated Tree Decomposition. The color can be `in` if and only if a node is inside the target set S, `out` to state that the node is not included in the set, `outc` to mark `out` *candidates* (used for encoding complete semantics), `attc` to mark attacking *candidate* nodes (they are not allowed to stay undefeated), `att` to mark attackers, and finally `def`, which is used to remember defeated nodes. It is important to note that, in general, the mapping of a node $n \in A$ to any of these colors is not fixed and therefore may change while the bottom-up traversal of the generated Tree Decomposition. This observation is due to the fact that, in general, information concerning the attack relation may appear some time later during the traversal of the decomposition.

Due to the already discussed process of how D-FLAT represents a solution in Chapter 4.1, i.e. how it generates the output of a valid solution, the following rules are introduced in order to transfer the output of D-FLAT into the correct solution for the given graph input.

1. The color of every node is either `in` or one from $\{\texttt{outc}, \texttt{out}, \texttt{attc}, \texttt{att}, \texttt{def}\}$.

2. If the color is one from $\{\texttt{outc}, \texttt{out}, \texttt{attc}, \texttt{att}, \texttt{def}\}$, the produced result needs to be viewed with respect to the partial ordering $<_C$, i.e. one needs to take the minimum thereof, formalized in Lines 4.1 and 4.2 as follows.

$$\forall col \in \{\texttt{outc}, \texttt{out}, \texttt{attc}, \texttt{att}, \texttt{def}\} : col \leq_C col \tag{4.1}$$

$$\texttt{def} <_C \texttt{att} <_C \texttt{attc} <_C \texttt{out} <_C \texttt{outc} \tag{4.2}$$

3. If one node does not explicitly assign a color, it is considered `out`, i.e. `out` is the default color.

Stable semantics This is the shortest and easiest of the encodings of this section. Its implementation uses the colors `in` and `def`, uncolored nodes are considered `out`.

Similar to the encodings of the previous section, it guesses the row, which is going to be extended in Line 2. Line 5 guesses, whether an introduced atom is going to be in the resulting extension or not included at all. It is required to remember which argument is

defeated (see Line 19). Recall that this is done similarly in incidence SAT above (see Listing 4.1), where we stored the already satisfied clauses. Note that arguments which are guessed to be in an extension can not be defeated, because extensions are required to be conflict-free (see Line 22). Lines 12 and 13 so to say inherit information, that is whether an argument is in an extension resp. defeated by it, to the next node towards the root of the TD. Finally, Line 16 makes sure that there are no undefeated arguments outside an extension (due to the connectedness property (3.) of the TD; see Definition 2.1).

```
1 % Guess root-to-leaf paths in item trees to be extended
2 1 { extend(R)  : childRow(R,N) } 1 ← childNode(N).

4 % Guess whether an element is in or out
5 0 { item(in(A)) } 1 ←           introduced(A).

7 % Join only arguments with the same color
8 nIn(S,A)  ← childRow(S,N), bag(N,A), not childItem(S,in(A)).
9 ← extend(S1), extend(S2), childItem(S1,in(A)),     nIn(S2,A).

11 % Inherit arguments that are in or def
12 item(in(A))     ← extend(S), childItem(S,in(A)),        current(A).
13 auxItem(def(A)) ← extend(S), childAuxItem(S,def(A)),    current(A).

15 % Discard sets if any attacking or out-argument is removed.
16 ← removed(A), not childItem(S,in(A)), not childAuxItem(S,def(A)),
   extend(S), childRow(S,N), bag(N,A).

18 % Set defeated arguments
19 auxItem(def(A))  ← current(A;B), att(B,A), item(in(B)).

21 % Assure that the set is conflict-free
22 ← item(in(A)), item(in(B)), att(A,B).
```

Listing 4.5: $\Pi_{\texttt{stable}}$: D-FLAT encoding for stable extensions.

Admissible semantics This program explicitly uses the colors in, def and attc (for marking attacking *candidates*), whereas uncolored nodes are considered out as mentioned above.

Note that in Chapter 3 we specified an algorithm for admissible semantics and proved its correctness. Listing 4.6 now shows an implementation of this algorithm for the D-FLAT framework.

$\Pi_{\texttt{admissible}}$ (see Listing 4.6) contains several parts, which are similar to the previous Listing 4.5. Lines 1 and 31 are in fact identical as in Listing 4.5. Lines 9, 10 and 11 ensure that only rows which color same arguments with the same color are joined together. Observe that in this sense argument a with colors def (defeated) of node, say n_1, and attc (attacking candidate) (which still require defeating) of node, say n_2 is allowed when

joining n_1 and n_2 together in node n_3, because this implies that a is already defeated in the subtree rooted at n_1 (see Section 3.1). Lines 15 and 20 also show this circumstance; only if a certain attacking candidate argument a is not defeated in any branch, it gets again color attc as seen in Line 23. The remaining parts are quite easy, however it is not allowed that an argument, which is guessed to be not in the extension (and also not an attacking candidate), is defeated (see Lines 27 resp. 19). Undefeated attacking candidates are not allowed, as Line 34 suggests.

```
1 1 { extend(R)  : childRow(R,N) } 1 ← childNode(N).

3 % Guess whether an element is in, out or attacking (attc)
4 0 { item(in(A)); attc(A) } 1        ← introduced(A).

6 % Join only arguments with the same color
7 nIn(S,A)  ← childRow(S,N), bag(N,A), not childItem(S,in(A)).
8 nDef(S,A) ← childRow(S,N), bag(N,A), not childAuxItem(S,def(A)),
      not childAuxItem(S,attc(A)).
9 ← extend(S1), extend(S2), childItem(S1,in(A)),       nIn(S2,A).
10 ← extend(S1), extend(S2), childAuxItem(S1,def(A)),  nDef(S2,A).
11 ← extend(S1), extend(S2), childAuxItem(S1,attc(A)), nDef(S2,A).

13 % Inherit arguments that are in, defeated or attackers
14 item(in(A)) ← extend(S), childItem(S,in(A)),      current(A).
15 chdef(A)    ← extend(S), childAuxItem(S,def(A)),  current(A).
16 attc(A)     ← extend(S), childAuxItem(S,attc(A)), current(A).

18 % Set defeated arguments
19 auxItem(def(A))  ← current(A;B), att(B,A), item(in(B)).
20 auxItem(def(A))  ← chdef(A).

22 % Still remaining (undefeated) attackers
23 auxItem(attc(A)) ← attc(A), not auxItem(def(A)).

25 % Out-arguments are not allowed to be defeated/attackers
26 out(A) ← not attc(A), not chdef(A), current(A).
27 ← auxItem(def(A)), out(A).
28 ← out(A), current(A), item(in(B)), att(A,B).

30 % Assure that the set is conflict-free
31 ← item(in(A)), item(in(B)), att(A,B).

33 % Remove candidates that leave attackers undefeated
34 ← extend(S), childAuxItem(S,attc(A)), removed(A).
```

Listing 4.6: $\Pi_{\texttt{admissible}}$: D-FLAT encoding for admissible sets.

The alternative program of Listing 4.7 explicitly uses the colors `in`, `def` and `att` (for marking attacking arguments), whereas uncolored nodes are considered `out` as mentioned

above.

Listing 4.7 contains on principal several parts, which are similar to the previous Listing 4.6, but simplified. Lines 1 and 20 is in fact identical as in Listing 4.6, the main difference lies in Line 4 (simplified guess). Line 8 ensures that only rows which color same arguments are joined together. Observe that in this sense also (in addition to the case of the previous encoding) arguments a with colors att (attacked) of node, say n_1, and out (no attacker until now) of node, say n_2 is allowed when joining n_1 and n_2 together in node n_3, because this implies that a is an attacker (in the subtree rooted at n_1; see Section 3.1). The remaining parts are quite easy. Undefeated attacking candidates are not allowed, as Line 23 suggests.

```
1 1 { extend(R)  : childRow(R,N) } 1 ← childNode(N).

3 % Guess whether an element is in or out
4 0 { item(in(A)) } 1       ← introduced(A).

6 % Join only arguments with the same color
7 nIn(S,A)  ← childRow(S,N), bag(N,A), not childItem(S,in(A)).
8 ← extend(S1), extend(S2), childItem(S1,in(A)), nIn(S2,A).

10 % Inherit arguments that are in, defeated or attackers
11 item(in(A))       ← extend(S), childItem(S,in(A)),     current(A).
12 auxItem(def(A))   ← extend(S), childAuxItem(S,def(A)), current(A).
13 auxItem(att(A))   ← extend(S), childAuxItem(S,att(A)), current(A),
     not auxItem(def(A)).

15 % Set defeated arguments
16 auxItem(def(A))  ← current(A;B), att(B,A), item(in(B)).
17 auxItem(att(A))  ← current(A;B), att(A,B), item(in(B)), not
     auxItem(def(A)).

19 % Assure that the set is conflict-free
20 ← item(in(A)), item(in(B)), att(A,B).

22 % Remove candidates that leave attackers undefeated
23 ← extend(S), childAuxItem(S,att(A)), removed(A).
```

Listing 4.7: $\Pi'_{\texttt{admissible}}$: Alternative D-FLAT encoding for admissible extensions.

Complete semantics In addition to the previous encoding, this one uses the additional color `outc` (and also explicitly colors nodes `out` this time), which marks `out` *candidates*.

This leads to the additional Line 15, where the out arguments are inherited. Trough the encoding we use `outc`/1 to actually mark both out candidates and out-arguments as seen in Line 29. An out candidate a becomes out in Line 22 if an other out candidate or out argument b attacks a, (i.e. it is not defeated due to an attack by an argument which is colored in). If an outc argument a does not become truely out, it still keeps

the out candidate status as formalized in Line 26. Similar to the previous Listing 4.6 it is not allowed to keep so to say unproven (i.e. candidate) arguments; in particular out candidates are not tolerated when the argument gets removed (see Line 40).

```
1 1 { extend(R)  : childRow(R,N) } 1 ← childNode(N).

3 % Guess whether an element is in, out or attacking (attc)
4 0 { item(in(A)); attc(A) } 1       ← introduced(A).

6 % Join only arguments with the same color
7 nIn(S,A)  ← childRow(S,N), bag(N,A), not childItem(S,in(A)).
8 nDef(S,A) ← childRow(S,N), bag(N,A), not childAuxItem(S,def(A)),
     not childAuxItem(S,attc(A)).
9 ← extend(S1), extend(S2), childItem(S1,in(A)),      nIn(S2,A).
10 ← extend(S1), extend(S2), childAuxItem(S1,def(A)),  nDef(S2,A).
11 ← extend(S1), extend(S2), childAuxItem(S1,attc(A)), nDef(S2,A).

13 % Inherit arguments that are in, defeated or attackers
14 item(in(A))       ← extend(S), childItem(S,in(A)),       current(A).
15 auxItem(out(A))   ← extend(S), childAuxItem(S,out(A)),   current(A).
16 chdef(A)          ← extend(S), childAuxItem(S,def(A)),   current(A).
17 attc(A)           ← extend(S), childAuxItem(S,attc(A)), current(A).

19 % Set defeated arguments
20 auxItem(def(A))  ← current(A;B), att(B,A), item(in(B)).
21 auxItem(def(A))  ← chdef(A).
22 auxItem(out(A))  ← current(A;B), att(B,A), outc(B), outc(A).

24 % Still remaining (undefeated) attackers
25 auxItem(attc(A))  ← attc(A), not auxItem(def(A)).
26 auxItem(outc(A))  ← outc(A), not auxItem(out(A)).

28 % Out−arguments are not allowed to be defeated/attackers
29 outc(A) ← not attc(A), not chdef(A), not item(in(A)), current(A).
30 ← auxItem(def(A)), outc(A).
31 ← outc(A), current(A), item(in(B)), att(A,B).

33 % Assure that the set is conflict−free
34 ← item(in(A)), item(in(B)), att(A,B).

36 % Remove candidates that leave attackers undefeated
37 ← extend(S), childAuxItem(S,attc(A)), removed(A).

39 % Remove candidates that leave out candidates undefeated
40 ← extend(S), childAuxItem(S,outc(A)), removed(A).
```

Listing 4.8: $\Pi_{\texttt{complete}}$: D-FLAT encoding for complete extensions.

Preferred semantics The goal is to select admissible sets $S \subseteq A$ of a framework (A, R), such that every $S' \supset S$ is not admissible. The intended way to deal with problems beyond *NP* is to use multiple levels [Bli12], so the additional level one is used to try to invalidate the condition above, i.e. to find for any guessed set S a set S', which is admissible. If no such S' exists, the guessed set actually is a preferred extension, otherwise it is not. Therefore colorings at level one represent counter candidates (compare with Chapter 4.2.1). If any of these counter-example candidates remain valid (at level one) for a row at level zero of the root node (of the decomposition), i.e. the according row at level one has not already been discarded during the tree traversal, the given row at level zero cannot represent a preferred extension.

This program explicitly uses the colors `in`, `def` and `attc`, whereas uncolored nodes are considered `out`.

Observe that there are quite redundant tasks, similar work has to be done at the different levels. Moreover, one can detect recurring patterns (compare to Listing 4.3).

In particular, we need additional Line 41 to prohibit a smaller (w.r.t. $\subseteq$) extension at level two. Lines 45 and 44 maintain the bigger flag to indicate true counter candidates (up to the current node). Finally, Line 46 rejects extensions with counter candidates at the root of the TD; otherwise they are accepted in Line 47.

```
1 length(2). or(0). and(1).

3 % Guess root−to−leaf paths in item trees to be extended
4 extend(0,R) ← root(R).
5 1 { extend(L+1,S) :  sub(R,S) } 1 ← extend(L,R), L<2.

7 % Guess truth value of introduced atoms
8 0 { item(L,in(A)); attc(L,A) } 1 ← introduced(A), L=1..2.

10 % Only join root−to−leaf paths coinciding on colors
11 nIn(S,A)  ← childNode(N), atNode(S,N), bag(N,A), not
     childItem(S,in(A)).
12 nDef(S,A) ← childNode(N), atNode(S,N), bag(N,A), not
     childAuxItem(S,def(A)), not childAuxItem(S,defc(A)).
13 ← extend(L,S1;L,S2), childItem(S1,in(A)),       nIn(S2,A).
14 ← extend(L,S1;L,S2), childAuxItem(S1,def(A)),   nDef(S2,A).
15 ← extend(L,S1;L,S2), childAuxItem(S1,defc(A)), nDef(S2,A).

17 % Inherit arguments that are in, defeated or attackers
18 item(L,in(A))   ← extend(L,S), childItem(S,in(A)),       current(A).
19 chdef(L,A)      ← extend(L,S), childAuxItem(S,def(A)),   current(A).
20 attc(L,A)       ← extend(L,S), childAuxItem(S,attc(A)), current(A).

22 % Set defeated arguments
23 auxItem(L,def(A))  ← current(A;B), att(B,A), item(L,in(B)).
24 auxItem(L,def(A))  ← chdef(L,A).
```

```
26 % Still remaining (undefeated) attackers
27 auxItem(L,attc(A)) ← attc(L,A), not auxItem(L,def(A)).

29 % Out−arguments are not allowed to be defeated/attackers
30 out(L,A) ← not attc(L,A), not chdef(L,A), current(A), L=1..2.
31 ← auxItem(L,def(A)), out(L,A).
32 ← out(L,A), current(A), item(L,in(B)), att(A,B).

34 % Assure that the set is conflict−free
35 ← item(L,in(A)), item(L,in(B)), att(A,B).

37 % Remove candidates that leave attackers undefeated
38 ← extend(L,S), childAuxItem(S,attc(A)), removed(A).

40 % Items at level 2 must be subset of that at level 1
41 ← item(1,A), not item(2,A).

43 % Update subset information; reject models at root with larger s
44 auxItem(2,bigger) ← extend(2,S), childAuxItem(S,bigger).
45 auxItem(2,bigger) ← item(2,A), not item(1,A).  %bigger S
46 reject ← final, auxItem(2,bigger).
47 accept ← final, not reject.
```

Listing 4.9: $\Pi_{\texttt{preferred}}$: D-FLAT encoding for directly computing preferred extensions.

Similar to the above alternative encoding covering admissible semantics (see Listing 4.7), we also state here a simpler encoding for preferred semantics for the sake of completeness. $\Pi_{\texttt{preferred}}$ (see Listing 4.9) is an implementation of the algorithm for preferred semantics defined in Chapter 3; this approach, however, can be simplified. In fact, Listing 4.10 evolved from the more efficient $\Pi'_{\texttt{admissible}}$ (compare with Listing 4.7). It explicitly uses the colors `in`, `def` and `att` (for marking attackers), whereas uncolored nodes are considered `out` as mentioned above. Similar to $\Pi_{\texttt{admissible}}$ vs. $\Pi'_{\texttt{admissible}}$, most of the parts are as in the previous Listing 4.9, but simplified.

```
1 length(2). or(0). and(1).

3 % Guess root−to−leaf paths in item trees to be extended
4 extend(0,R) ← root(R).
5 1 { extend(L+1,S) :  sub(R,S) } 1 ← extend(L,R), L<2.

7 % Guess truth value of introduced atoms
8 0 { item(L,in(A)) } 1 ← introduced(A), L=1..2.

10 % Items at level 2 must be subset of that at level 1
11 item(2,A) ← item(1,A).

13 % Only join root−to−leaf paths coinciding on colors
14 nIn(S,A)  ← childNode(N), atNode(S,N), bag(N,A), not
```

```
    childItem(S,in(A)).
15 ← extend(L,S1;L,S2), childItem(S1,in(A)),      nIn(S2,A).

17 % Inherit arguments that are in, defeated or attackers
18 item(L,in(A))      ← extend(L,S), childItem(S,in(A)),     current(A).
19 auxItem(L,def(A)) ← extend(L,S), childAuxItem(S,def(A)), current(A).
20 % Set defeated arguments
21 auxItem(L,def(A)) ← current(A;B), att(B,A), item(L,in(B)).

23 % Still remaining (undefeated) attackers
24 auxItem(L,att(A)) ← extend(L,S), childAuxItem(S,att(A)),
    current(A), not auxItem(L,def(A)).
25 auxItem(L,att(A)) ← current(A;B), att(A,B), item(L,in(B)), not
    auxItem(L,def(A)).

27 % Assure that the set is conflict-free
28 ← item(L,in(A)), item(L,in(B)), att(A,B).

30 % Remove candidates that leave attackers undefeated
31 ← extend(L,S), childAuxItem(S,att(A)), removed(A).

33 % Update subset information; reject models at root with larger s
34 auxItem(2,bigger) ← extend(2,S), childAuxItem(S,bigger).
35 auxItem(2,bigger) ← item(2,A), not item(1,A).  %bigger S
36 reject ← final, auxItem(2,bigger).
37 accept ← final, not reject.
```

Listing 4.10: $\Pi'_{\texttt{preferred}}$: Alternative D-FLAT encoding for computing preferred extensions.

Semi-stable semantics This program behaves quite similar to the encoding before, but has to somehow capture the fact that our goal is to maximize the range S_R^+ of any given admissible set S. Remember (compare with Chapter 2) that the range is defined as follows: $S_R^+ := S \cup \{a \mid \exists b \in S \ s.t. \ (b,a) \in R\}$.

Note that Listing 4.11 complies with a D-FLAT encoding of the algorithm for semi-stable semantics specified (and shown to be correct) in Chapter 3.

Listing 4.11 is quite similar to the previous Listing 4.9, but maximizes the range. For this, we need Lines 41, 42 and 43, which set up the range. Line 48 finally is the desired line for setting the bigger flag similar to the previous encoding. Note that this is the implementation of the according theoretical Section 3.2, in other words Listing 4.11 forms the implementation of the proven algorithm.

```
1 length(2). or(0). and(1).

3 % Guess root-to-leaf paths in item trees to be extended
4 extend(0,R) ← root(R).
```

```
5 1 { extend(L+1,S) :  sub(R,S) } 1 ← extend(L,R), L<2.

7 % Guess truth value of introduced atoms
8 0 { item(L,in(A)); attc(L,A) } 1 ← introduced(A), L=1..2.

10 % Only join root−to−leaf paths coinciding on colors
11 nIn(S,A)  ← childNode(N), atNode(S,N), bag(N,A), not
     childItem(S,in(A)).
12 nDef(S,A) ← childNode(N), atNode(S,N), bag(N,A), not
     childAuxItem(S,def(A)), not childAuxItem(S,defc(A)).
13 ← extend(L,S1;L,S2), childItem(S1,in(A)),      nIn(S2,A).
14 ← extend(L,S1;L,S2), childAuxItem(S1,def(A)),  nDef(S2,A).
15 ← extend(L,S1;L,S2), childAuxItem(S1,defc(A)), nDef(S2,A).

17 % Inherit arguments that are in, defeated or attackers
18 item(L,in(A))  ← extend(L,S), childItem(S,in(A)),      current(A).
19 chdef(L,A)     ← extend(L,S), childAuxItem(S,def(A)),  current(A).
20 attc(L,A)      ← extend(L,S), childAuxItem(S,attc(A)), current(A).

22 % Set defeated arguments
23 auxItem(L,def(A))  ← current(A;B), att(B,A), item(L,in(B)).
24 auxItem(L,def(A))  ← chdef(L,A).

26 % Still remaining (undefeated) attackers
27 auxItem(L,attc(A)) ← attc(L,A), not auxItem(L,def(A)).

29 % Out−arguments are not allowed to be defeated/attackers
30 out(L,A) ← not attc(L,A), not chdef(L,A), current(A), L=1..2.
31 ← auxItem(L,def(A)), out(L,A).
32 ← out(L,A), current(A), item(L,in(B)), att(A,B).

34 % Assure that the set is conflict−free
35 ← item(L,in(A)), item(L,in(B)), att(A,B).

37 % Remove candidates that leave attackers undefeated
38 ← extend(L,S), childAuxItem(S,attc(A)), removed(A).

40 % s at level 2 must be subset of that at level 1
41 s(L,A) ← item(L,in(A)).
42 s(L,A) ← auxItem(L,attc(A)).
43 s(L,A) ← auxItem(L,def(A)).
44 ← s(1,A), not s(2,A).

46 % Update subset information; reject models at root with larger s
47 auxItem(2,bigger) ← extend(2,S), childAuxItem(S,bigger).
48 auxItem(2,bigger) ← s(2,A), not s(1,A).    %bigger S
49 reject ← final, auxItem(2,bigger).
```

```
50 accept ← final, not reject.
```
Listing 4.11: $\Pi_{\texttt{semiStable}}$: D-FLAT encoding for directly computing semi-stable extensions.

4.2 D-FLAT^2: Optimizing DP on TDs

4.2.1 Technical Details

We give a general algorithm of how the task of subset minimization in DP on TDs can be implemented efficiently. There are some issues with DP algorithm specifications involving subset optimization. In particular, the development of our algorithm is motivated by the following observations.

1) Counter candidates are often constructed similarly to solution candidates.
2) Typically, counter candidates are also stored as solution candidates.
3) Counter candidates that are also solution candidates can be omitted if the respective solution candidate turns out to be no solution.

Our approach avoids redundant computations of solution and counter candidates. It supports minimization and maximization on user-specified items (e.g., for ⊆-Minimal Sat, on atoms but not on clauses).

In our approach so-called *reduced item trees* serve as the main data structure. A reduced item tree is an item tree of depth 1 that can store additional information at each node n: Besides extension pointer tuples, n is associated with a set of *back pointers*. Node n has a back pointer to a node n' iff some element of some extension pointer tuple in n' references n. Furthermore, n has an *optimization item set*, which contains the items on which subset minimization is performed. In contrast to the straightforward way of using (non-reduced) item trees whose depth-2 nodes represent counter candidates, n contains a set of *counter candidate pointers*, which are hidden from the user. A counter candidate pointer is a pair (c, s), where c is a reference to a sibling of n and s is a Boolean flag that is set to true iff there is an extension of c whose optimization items form a proper subset (or superset for maximization problems) of those of each extension of n (similar to the `smaller` predicate in Listing 4.3). Note that every node n has at least the counter candidate pointers $(n, \top)$ or $(n, \bot)$.

Opposed to classical implementations such as the algorithm for ⊆-Minimal Sat described in Section 2, we perform two bottom-up traversals of the TD: In the first traversal, we compute all reduced item trees as in classical implementations, but only up to depth 1. After this step, the back pointers are added to the reduced item tree nodes. In the second traversal, we add counter candidates to nodes at depth 1 appropriately, but instead of creating children that are copies of other item sets from depth 1, we store only counter candidate pointers to already existing item sets.

Our main contribution is outlined in Algorithms 4.1, 4.2 and 4.3 (assuming semi-normalized TDs as defined in Definition 2.2 whose leaves have empty bags): The second

Algorithm 4.1: The procedure `computeLv2`.

```
   Input: Item tree rooted at a node r
 1 foreach tuple ∈ r.extPtrs, e ∈ tuple do
 2 │ computeLv2(e);
 3 end
 4 foreach c ∈ r.children do
 5 │ if r belongs to a leaf node then
 6 │ │ c.counterC ← c.counterC ∪ {(c, ⊥)};
 7 │ end
 8 │ else
 9 │ │ foreach tuple ∈ c.extPtrs do
10 │ │ │ if r belongs to an exchange node then
11 │ │ │ │ cCs ← handleExchange(c, tuple);
12 │ │ │ end
13 │ │ │ else
14 │ │ │ │ cCs ← handleJoin(c, tuple);
15 │ │ │ end
16 │ │ │ insertCompress(c, tuple, cCs);
17 │ │ end
18 │ end
19 end
```

TD traversal is initiated by applying `computeLv2` to the root of the reduced item tree at the TD's root. This results in all counter candidates being appended to the reduced item trees.

To begin with, we describe the notation employed in our pseudocode. Let n be a reduced item tree node. By n.extPtrs we denote its set of extension pointer tuples, n.children refers to its set of children, n.backPtrs is its set of back pointers, n.items is its set of items, and n.optItems is a set consisting of those items at n among which minimization is performed. Finally, if n is at depth 1 of the reduced item tree, n.counterC is a set of counter candidates of the form (c, s). At the TD's root, these are used to delete any node n at depth 1 of the reduced item tree that has among n.counterC a tuple $(n', \top)$, which witnesses that n' represents a better solution than n.

The omitted procedure `insertCompress`$(c, \textit{tuple}, \textit{cCs})$ sets c.extPtrs to c.extPtrs $\setminus \{\textit{tuple}\}$, duplicates node c resulting in c' and adds c' to the reduced item tree (as a sibling of c) with c'.counterC set to $\textit{cCs}$ and c'.extPtrs set to $\{\textit{tuple}\}$; moreover, c.backPtrs and c'.backPtrs get recomputed. We will discuss the reason for this duplication below (and it will turn out that in some cases we can avoid duplication).

Algorithm 4.2: The function `handleExchange`.

```
   Input: Item tree node c, extension pointer tuple (e)
1  cCs ← ∅;
2  foreach (cc, strict) ∈ e.counterC do
3  |  foreach b ∈ cc.backPtrs do
4  |  |  if b.optItems ⊆ c.optItems then
5  |  |  |  s ← strict ∨ (b.optItems ⊂ c.optItems);
6  |  |  |  cCs ← cCs ∪ {(b, s)};
7  |  |  end
8  |  end
9  end
10 return cCs;
```

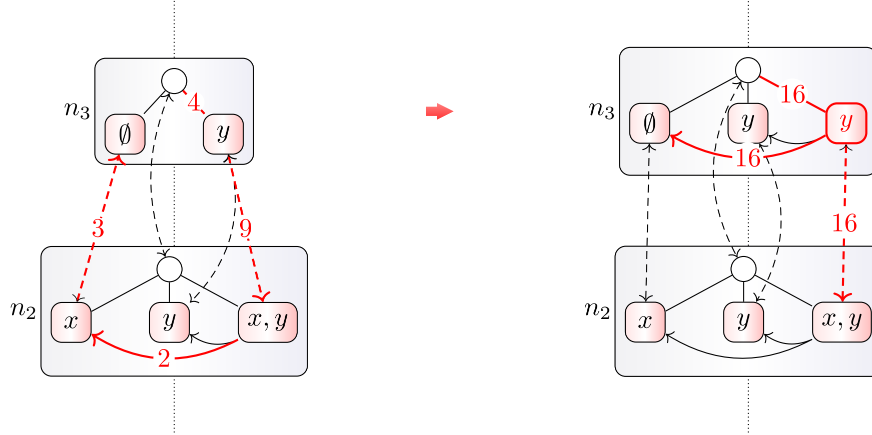

Figure 4.5: Item tree computation for exchange node n_3 of $\mathcal{T}_{Ex}$.

4.2.1.1 Exchange nodes

We show how Algorithm 4.2 proceeds at exchange nodes by means of an example illustrated in Figure 4.5, which consists of two parts: The left hand side depicts two reduced item trees belonging to an exchange node n_3 and its child n_2, respectively, just before the counter candidates at node n_3 are computed. The result of this computation is depicted on the right hand side. Straight solid lines signify parent-child relationships in an reduced item tree; dashed arrows represent extension pointers and back pointers; a curved solid arrow denotes that the target node represents a counter candidate to the source node. (All such arrows depicted in our figures correspond to counter candidates whose subset relation is proper. Note that we omitted self loops via such arrows, which are in fact present at every node because any solution candidate is a counter candidate to itself.) In the following, we explain the figure in detail.

Assuming that all solution candidates have already been obtained, the idea for

computing the counter candidates at n_3 is as follows: For each solution candidate C at n_3, we look at the solution candidates at n_2 reachable by extension pointers. For these we already know the counter candidates, as we are doing a bottom-up traversal. We iterate over all of these counter candidates and check if the current one has some back pointer referencing a sibling of C called C' whose set of optimization items is a subset of the respective item set of C. If so, we conclude that C' is a counter candidate to C.

To see an example for such a situation, let $n_{i:S}$ in the following denote the reduced item tree node at n_i whose item set is S in the left hand side of Figure 4.5. Assume that our invocation of `computeLv2` is currently in a state where the variable c in Line 4 is $n_{3:\{y\}}$ and the variable *tuple* in Line 9 has the value $(n_{2:\{x,y\}})$. We moreover assume that the tuple of variables (cc, strict) in Line 2 is set to $(n_{2:\{x\}}, \top)$, and the variable b in Line 3 is $n_{3:\emptyset}$. This state of our invocation of `computeLv2` is indicated in the left part of Figure 4.5 by red arrows whose attached numbers indicate the line number of the corresponding loop. Since $n_{3:\emptyset}.\mathrm{optItems} \subset n_{3:\{y\}}.\mathrm{optItems}$, the set cCs in Line 6 grows by $(n_{3:\emptyset}, \top)$ in the current iteration of the loop. Similarly, the next iteration of the loop in Line 2 causes cCs to grow by $(n_{3:\{y\}}, \top)$ and the current call to `handleExchange` returns $\{(n_{3:\emptyset}, \top), (n_{3:\{y\}}, \top)\}$.

As shown on the right hand side of Figure 4.5, the subsequent call to `insertCompress` splits node $n_{3:\{y\}}$ into two nodes having the same item set: One of these copies extends $n_{2:\{y\}}$ while the other one extends $n_{2:\{x,y\}}$. In the former case, no counter candidates exist (as $n_{2:\{y\}}$ has none), whereas in the latter we can obtain counter candidates just as the preceding call to `handleExchange` indicated. Splitting nodes like this has the reason that the counter candidates of a solution candidate C depend on which extension pointer of C we choose.

In some cases we can avoid this duplication – in fact, the case of $n_{3:\{y\}}$ is one of them: The rightmost copy of $n_{3:\{y\}}$ on the right hand side of Figure 4.5 has among its counter candidates a node with the same item set. So if that copy turns out to lead to a solution, then that counter candidate will lead to a proper counterexample: As soon as two nodes have the same item set, they are indistinguishable by DP algorithms.

4.2.1.2 Join nodes

At join nodes, extension pointer tuples have arity 2, so back pointers are not just inverted extension pointers. This complicates the algorithm compared to the case of exchange nodes. Algorithm 4.3 shows how we handle join nodes.

To explain the algorithm, we follow an example illustrated in Figure 4.6. Let $n_{i:S}$ in the following denote the reduced item tree node at n_i whose item set is S in the left hand side of Figure 4.6. Assume that our invocation of `computeLv2` is currently in a state as depicted in Figure 4.6, where the variable c in Line 4 is $n_{8:\{y,c_2\}}$ and the variable *tuple* in Line 9 has the value $(n_{4:\{y\}}, n_{7:\{y,c_2\}})$. We moreover assume that the tuple of variables (i_1, i_2) in Line 5 is set to $(n_{4:\{y\}}, n_{7:\{y\}})$; note that in Figure 4.6 we have omitted the arrow for the first component as it is one of the self-loops that we have hidden. In Line 6, procedure `occurExtPtrs`(tuple, c) provides a set of nodes such that for each b in the set we have $\mathit{tuple} \in b.\mathrm{extPtrs}$ and $b.\mathrm{optItems} \subseteq c.\mathrm{optItems}$. Intuitively, `occurExtPtrs` uses

Algorithm 4.3: The function `handleJoin`.

Input: Item tree node c, extension pointer tuple (e_1, e_2)

1 $(cCs, e'_1, e'_2) \leftarrow (\emptyset, \emptyset, \emptyset)$;
2 **foreach** $e_i \in \{e_1, e_2\}, (cc, \mathit{strict}) \in e_i$.counterC **do**
3 | $e'_i \leftarrow e'_i \cup \{cc\}$;
4 **end**
5 **foreach** $(i_1, i_2) \in e'_1 \times e'_2$ **do**
6 | **foreach** $b \in$ `occurExtPtrs`$((i_1, i_2), c)$ **do**
7 | | $\mathit{strict}_1 \leftarrow (i_1, \top) \in e_1$.counterC;
8 | | $\mathit{strict}_2 \leftarrow (i_2, \top) \in e_2$.counterC;
9 | | $cCs \leftarrow cCs \cup \{(b, \mathit{strict}_1 \vee \mathit{strict}_2)\}$;
10 | **end**
11 **end**
12 **return** cCs;

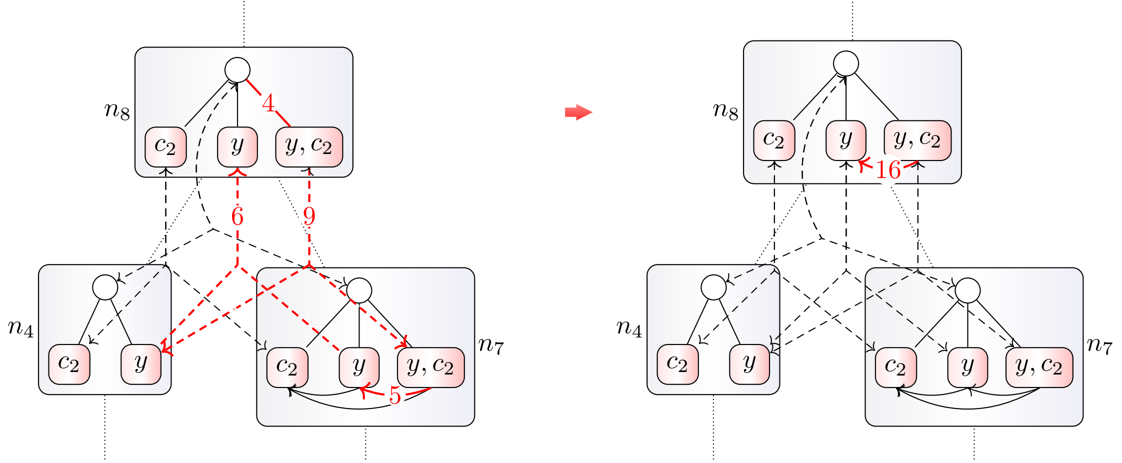

Figure 4.6: Item tree computation for join node n_8 of $\mathcal{T}_{Ex}$.

the backward pointers to compute all the extension pointer occurrences, i.e., nodes that have the given 2-tuple among their extension pointer tuples. In our example, for tuple $(n_{4:\{y\}}, n_{7:\{y\}})$, variable b is $n_{8:\{y\}}$. Line 16 inserts the found node $n_{8:\{y\}}$ to the counter candidates of $n_{8:\{y,c_2\}}$, as depicted on the right hand side of Figure 4.6. The computation of the strict flag in Line 9 is not depicted; it is the disjunction of the flags from the two counter candidates that are extended by the new counter candidate.

4.2.2 Further Optimizations

After the counter candidates at a TD node have been computed, all counter candidate pointers at children in the TD can be discarded, which allows for all reduced item tree

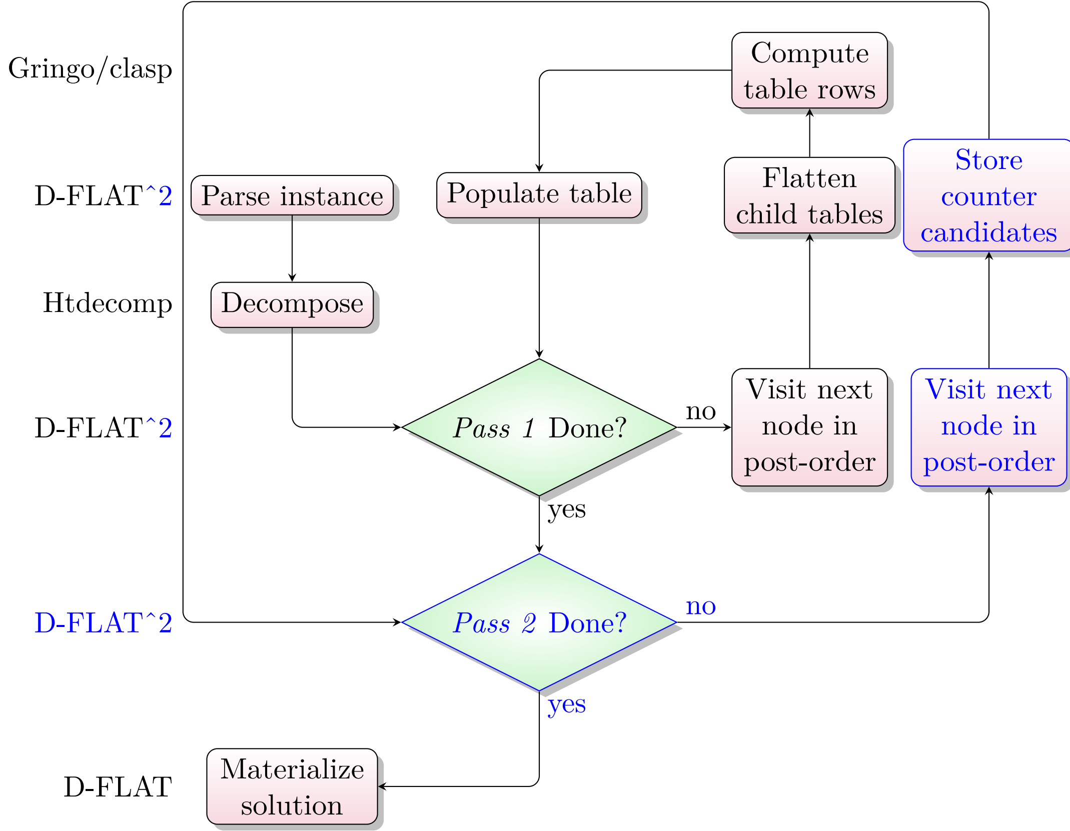

Figure 4.7: D-FLAT^2 updated flowchart, which shows how the system and its components work.

nodes to be deleted that are no longer referred to by an extension pointer. A further optimization is to remove reduced item tree nodes n with $(n', \top) \in n.\text{counterC}$ such that $n.\text{items} = n'.\text{items}$.

As for some problems the counter candidates are not necessarily solution candidates, we propose to mark them with a special flag as "pseudo solution candidates" that only serve to be referenced as counter candidates. To avoid unnecessary memory consumption, such candidates can be deleted as soon as the parent node in the TD has been fully processed.

4.2.3 System Overview

We have implemented our approach (as described in the previous section) in a system called D-FLAT^2 by extending the publicly available D-FLAT system [ABC+14b]. The updated flowchart compared to D-FLAT (see Figure 4.1 of Chapter 4.1) can be seen in Figure 4.7.

Both systems allow the problem-specific steps of a DP algorithm working on a TD to be specified in ASP; i.e., similar to the traditional D-FLAT methodology, the only thing required from the user of D-FLAT^2 is an ASP encoding where each model corresponds to a single solution or counter candidate. In D-FLAT^2, the user can specify the computation of solution candidates and the system takes care of the second TD traversal that computes counter candidate pointers and performs subset minimization. For the introductory example $\subseteq$-MINIMAL SAT, it suffices to provide an algorithm encoding that computes models like for SAT and to specify that items corresponding to atoms are optimization items. Similarly, for preferred sets in Abstract Argumentation, we can start with an encoding for admissible sets and state that all arguments that are in an extension also are optimization items, as we will see later on in Section 4.2.4.

As already mentioned, besides extension pointer tuples, every node n has an *optimization item set*, which contains the items on which subset optimization is performed. D-FLAT^2 thus defines the new output predicate `optItem`/1, where `optItem`(S) means that item S is subject to optimization. Instead of using item trees whose nodes at depth 2 represent counter candidates, n contains a set of *counter candidate pointers*, which are hidden from the user.

```
optItem(X) ← item(X).
```

Listing 4.12: $\Pi_{\texttt{optAllItems}}$: used for solving conceptually simple problems (e.g., $\subseteq$-MINIMAL SAT).

Program $\Pi_{\texttt{optAllItems}}$ (Listing 4.12) illustrates a trivial usage of the `optItem`/1 predicate. With this, we can obtain the D-FLAT^2 encoding $\Pi^2_{\subseteq\text{-MINIMAL SAT}} = \Pi_{\texttt{optAllItems}} \cup \Pi_{\text{SAT}}$, which allows us to solve $\subseteq$-MINIMAL SAT by simply using the existing encoding for SAT and adding information on what to optimize. In this case, the basic specification $\Pi_{\texttt{optAllItems}}$ suffices, as in $\subseteq$-MINIMAL SAT we consider all items for minimization. D-FLAT^2 encoding $\Pi^2_{\subseteq\text{-MINIMAL SAT}}$ gives several advantages over the traditional D-FLAT encoding (see Listing 4.3): It allows us to again use the simplified table-based interface and the overall length of the encoding is greatly reduced.

Remember the encoding $\Pi^{\text{prim}}_{\text{SAT}}$ (see Listing 4.2) for solving SAT using primal graphs instead of incidence graphs. Recall that the primal graph of a given formula consists of all the atoms of the formula and every atom of some clause c is connected to every other one of c. Adding now $\Pi_{\texttt{optAllItems}}$, it is possible to obtain $\Pi^{2\ \text{prim}}_{\subseteq\text{-MINIMAL SAT}}$ for solving $\subseteq$-MINIMAL SAT using primal graphs, i.e. $\Pi^{2\ \text{prim}}_{\subseteq\text{-MINIMAL SAT}} = \Pi_{\texttt{optAllItems}} \cup \Pi^{\text{prim}}_{\text{SAT}}$.

For some problems, we require counter candidates that are not solution candidates at the same time. An example will be given in Section 4.2.4, where we consider disjunctive ASP. There, counter candidates are models of a program reduct and not the program itself. In such cases, the item tree nodes representing such counter candidates can be marked with a special flag as "pseudo solution candidates" by means of the atom `auxItem`(`pseudo`), which is taken into account by D-FLAT^2.

4.2.3.1 Command-line Interface

In the following, we shortly describe the command-line interface for D-FLAT^2 in addition to its ASP interface as seen in the previous chapter (Section 4.1). More detailed descriptions can be found in [ABC+14a] and [ABC+14b]. The examples (as for instance the previous section) only work for semi-normalized Tree Decomposition. Thus, one needs to start D-FLAT^2 with `-n semi`. Minimization resp. maximization (i.e. enabling the recognition of the `optItem` predicate) can be switched on by adding `--tables-min` resp. `--tables-max`.

The rest of the options required for D-FLAT^2 are the same as for D-FLAT. An edge type is specified with `-e` and the program filename with `-p` (containing the ASP program and occurrences of the `optItem` predicate).

4.2.4 Application to Common AI Problems

4.2.4.1 Abstract Argumentation

Problems from Abstract Argumentation [Dun95] are further examples where our approach is reasonable. Recall from Chapter 2, that, given an object $F = (A, R)$, where A is a set of arguments and $R \subseteq A \times A$, we call a set $S \subseteq A$ *admissible in* F if (1) $(a, b) \notin R$ for all $a, b \in S$ and (2) for each $s \in S$ and $r \in A$, $(r, s) \in R$ implies that there is some $q \in S$ with $(q, r) \in R$. S is *preferred in* F if it is a subset-maximal admissible set in F. For any $C \subseteq A$, we call $C^+_R = C \cup \{a \mid \exists b \in C \text{ s.t. } (b, a) \in R\}$ the *range of* C *in* F. A set S is *semi-stable in* F if it is admissible in F and for every admissible set S' in F, $S^+_R \not\subset S'^+_R$ holds.

Recall Listing 4.6, which shows an encoding $\Pi_{\texttt{admissible}}$ for computing admissible sets. $\Pi_{\texttt{admissible}}$ complies with the algorithm for admissible extensions specified in Chapter 3 and is also used in Chapter 3 to form an algorithm for computing semi-stable semantics. At first glance, it seems to be overcomplicated compared to the algorithm in [DPW12] (compare with $\Pi'_{\texttt{admissible}}$ of Listing 4.7): For computing admissible sets it actually suffices to guess which introduced arguments are in S, whereas we guess in $\Pi_{\texttt{admissible}}$ which arguments are in the set, attackers or neither. In order for a guessed set to be a solution, every attacker has to be defeated by the time it is removed from the bag. Once it can be determined that an attacker is defeated, its status changes from “`attc`” to “`def`”. This additional complexity allows us to reuse $\Pi_{\texttt{admissible}}$ for computing semi-stable sets later on.

Given $\Pi_{\texttt{admissible}}$, we can compute preferred sets by simple subset maximization via $\Pi^2_{\texttt{preferred}} = \Pi_{\texttt{optAllItems}} \cup \Pi_{\texttt{admissible}}$. Moreover, for computing preferred extension, we can even use $\Pi'_{\texttt{admissible}}$ (see Listing 4.7), i.e. we have that $\Pi^{2'}_{\texttt{preferred}} = \Pi_{\texttt{optAllItems}} \cup \Pi'_{\texttt{admissible}}$ with $\Pi^{2'}_{\texttt{preferred}} \equiv \Pi^2_{\texttt{preferred}}$. Furthermore, it is now also easy to compute semi-stable sets by means of $\Pi_{\texttt{optForSemiStable}}$ (Listing 4.13), which gives us $\Pi^2_{\texttt{semiStable}} = \Pi_{\texttt{optForSemiStable}} \cup \Pi_{\texttt{admissible}}$. Although the guess from $\Pi_{\texttt{admissible}}$ could have been simplified for the previous examples, here it is indeed required because we need to find those admissible sets that have maximal range.

```
1 optItem(A) ← item(in(A)).
2 optItem(A) ← auxItem(attc(A)).
3 optItem(A) ← auxItem(def(A)).
```

Listing 4.13: $\Pi_{\texttt{optForSemiStable}}$: used for computing semi-stable sets via $\Pi^2_{\texttt{semiStable}} = \Pi_{\texttt{optForSemiStable}} \cup \Pi_{\texttt{admissible}}$.

Note that with D-FLAT^2 it is even possible to take complete semantics (see Chapter 4.1.3) and compute some modification of preferred semantics, where complete semantics serves as basis (see Definition 2.17), i.e. $\Pi^2_{\texttt{preferred}_{\texttt{complete}}} = \Pi_{\texttt{optAllItems}} \cup \Pi_{\texttt{complete}}$. Similarly, one can solve semi-stable semantics with complete as basis semantics (see Definition 2.17), in signs $\Pi^2_{\texttt{semiStable}_{\texttt{complete}}} = \Pi_{\texttt{optForSemiStable}} \cup \Pi_{\texttt{complete}}$.

4.2.4.2 Circumscription

In the propositional case of Circumscription [McC80], we are given a theory T and sets of atoms P and Z, and we are interested in models M of T such that there is no model M' with $M' \cap P \subset M \cap P$ and $M \cap Z = M' \cap Z$ (see Definition 4.1).

Definition 4.1. *The problem* CIRC *[McC80] is defined as follows: Given a theory T and sets of atoms P and Z. The goal is the set of models defined as* $\text{CIRC}_{\texttt{Mod}}(T; P; Z) := \{M \mid M \models T, \nexists M^* : (M^* \models T, M^* \cap P \subset M \cap P, M \cap Z = M^* \cap Z)\}$. *The formula whose classical models correspond to exactly those solutions is denoted by* $\text{CIRC}(T; P; Z)$.

We can model Circumscription in our approach by a slight modification of our $\subseteq$-MINIMAL SAT algorithm: We only put an atom x in an optimization item set if $x \in P$; and for any $x \in Z$ we add optimization items $t(x)$ or $f(x)$ if the item set contains x or not, respectively (thus making solution candidates with different interpretations of Z incomparable). By applying this technique we have that for any sibling nodes n and n' in an item tree, the optimization item set of n is a subset of the one of n' iff $M_n \cap P \subseteq M_{n'} \cap P$ and $M_n \cap Z = M_{n'} \cap Z$, where M_n is the partial interpretation for node n following extension pointers. The program $\Pi_{\texttt{optForCirc}}$ (Listing 4.14) takes these considerations into account; moreover, $\Pi^2_{\text{CIRC}} = \Pi_{\texttt{optForCirc}} \cup \Pi_{\text{SAT}}$ is a D-FLAT^2 implementation of the DP algorithm for Circumscription. As input it expects the theory T to be given as a CNF formula like for Π_{SAT} and the sets P and Z to be given using the unary predicates `p` and `z`, respectively.

```
1 optItem(X)    ←     item(X), p(X).
2 optItem(t(X)) ←     item(X), z(X).
3 optItem(f(X)) ← not item(X), z(X), current(X).
```

Listing 4.14: $\Pi_{\texttt{optForCirc}}$: used for solving Circumscription via $\Pi^2_{\text{CIRC}} = \Pi_{\texttt{optForCirc}} \cup \Pi_{\text{SAT}}$.

Similar to $\Pi^{\text{prim}}_{\text{SAT}}$ (see Listing 4.2) one can also solve Circumscription using primal graphs. With D-FLAT^2, we can reuse $\Pi_{\texttt{optForCirc}}$ to obtain the encoding $\Pi^{2\ \text{prim}}_{\text{CIRC}} = \Pi_{\texttt{optForCirc}} \cup \Pi^{\text{prim}}_{\text{SAT}}$.

4.2.4.3 Disjunctive ASP

While a traditional TD-based DP algorithm for solving disjunctive ASP can be found in [JPW09], here we solve the problem with D-FLATˆ2. We first do so via reduction to Circumscription. In the following, for any interpretation, rule or set of atoms X, we write X' to denote the result of replacing each atom a in X with a new atom a' (see Proposition 4.1). Given a disjunctive logic program Π consisting of rules, we use the notations $H(r), B^+(r)$ and $B^-(r)$ (cf. Definition 4.2).
Definition 4.1 can now be used to define a reduction from disjunctive ASP to Circumscription. For this, we need to add a new definition concerning ASP.

Definition 4.2. *Given a disjunctive logic program Π consisting of rules of the form $a_1 \vee a_2 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n$, we denote $H(r) := \bigvee_{1 \leq i \leq k} a_i$, $B^+(r) := \bigwedge_{1 \leq i \leq m} b_i$ and $B^-(r) := \bigwedge_{m+1 \leq i \leq n} \neg b_i$. $HB(\Pi)$ denotes the Herbrand base of Π, that is, it contains all the (ground) atoms that can possibly occur within Π.*

Proposition 4.1. *For any interpretation, rule or set of atoms X, we write X' to denote the result of replacing each atom a in X with a new atom a'. As shown in [PTW09], $I \in \texttt{Answers}(\Pi)$ iff $I \cup I' \models \psi$ with*

$$\psi := \bigwedge_{p \in HB(\Pi)} (p \equiv p') \wedge \textsc{Circ}(\bigcup_{r \in \Pi} \{(B^+(r) \wedge B^-(r')) \rightarrow H(r)\}; HB(\Pi); HB(\Pi)').$$

As shown in [PTW09], an interpretation I is an Answer-Set of Π iff $I \cup I'$ is a model of ψ, assuming $HB(\Pi)$ denotes the Herbrand base of Π (see Proposition 4.1). In order to compute models of this formula, we can first calculate models of the Circumscription part and then remove those models (using the "`pseudo`" item for marking pseudo solution candidates), where the truth value of some atom a is different from the one of a'. This amounts to $\Pi_{\texttt{pseudoForASP}}$ (Listing 4.15), which can be used for solving disjunctive ASP by means of the combined program $\Pi^2_{\texttt{ASP}} = \Pi_{\texttt{pseudoForASP}} \cup \Pi^2_{\textsc{Circ}}$. The predicate `cor`/2 in this encoding is assumed to be symmetric and occurring in input facts in order to associate each atom a with its corresponding primed variant a'. (We require that a and a' always occur together in some bag, which can be achieved by adding an edge (a, a') to the input graph.)

```
1 auxItem(pseudo) ← cor(A,B), current(A;B), item(A), not item(B).
2 auxItem(pseudo) ← extend(R), childAuxItem(R,pseudo).
```

Listing 4.15: $\Pi_{\texttt{pseudoForASP}}$: used for solving disjunctive ASP via $\Pi^2_{\texttt{ASP}} = \Pi_{\texttt{pseudoForASP}} \cup \Pi^2_{\textsc{Circ}}$.

Once more, this trick also works for primal graphs; since ASP is solved here via reduction to Circumscription, one can also state an encoding for D-FLATˆ2 using primal graphs: $\Pi^{2\ \text{prim}}_{\texttt{ASP}} = \Pi_{\texttt{pseudoForASP}} \cup \Pi^{2\ \text{prim}}_{\textsc{Circ}}$.

Example 4.3. *Assume the following simple QBF (see Section 2.4) $\varphi := \exists a \exists c \forall b((\neg a \wedge c) \vee (a \wedge c) \vee b)$. In order to solve this problem, we can use the propositional disjunctive*

grounded program $\Pi'_{\texttt{ex}}$ given in Listing 4.16. It uses the saturation technique and atoms tV resp. fV to model that variable V of φ ($V \in \{a, b, c\}$) is set to true resp. false. Now we can transform this problem into CIRC *as seen in the formula* $\varphi_{\text{CIRC}} := (sat \equiv sat') \wedge \text{CIRC}(\varphi_{\texttt{rules}}; \{sat, ta, fa, tb, fb, tc, fc\}; \{sat'\})$ *where* $\varphi_{\texttt{rules}} = \{fa \wedge tc \rightarrow sat, ta \wedge tc \rightarrow sat, tb \rightarrow sat, sat \rightarrow tb, sat \rightarrow fb, \neg sat' \rightarrow \bot, \top \rightarrow ta \vee fa, \top \rightarrow tb \vee fb, \top \rightarrow tc \vee fc\}$.

Listing 4.17 shows a given valid input instance for D-FLAT^2 using program $\Pi^2_{\texttt{ASP}}$ *(and the option to perform minimization).*

```
1  % Model the cases where the matrix of φ evaluates to true
2  sat ← fa, tc.
3  sat ← ta, tc.
4  sat ← tb.

6  % Saturize over the ∀-quantified variables, if φ evaluates to true
7  tb ← sat.
8  fb ← sat.

10 % Ensure satisfiability
11 ← not sat.

13 % Guess truth values of variables a,b and c
14 ta v fa.
15 tb v fb.
16 tc v fc.
```

Listing 4.16: $\Pi'_{\texttt{ex}}$: ASP program for solving QBF φ.

```
1  % Define used clauses
2  clause(c1). clause(c2). clause (c3). clause(c4). clause(c5).
3  clause(c6). clause(c7). clause (c8). clause(c9).

5  % Define used atoms
6  atom(sat). atom(sat'). cor(sat, sat').
7  atom(ta). atom(fa).
8  atom(tb). atom(fb).
9  atom(tc). atom(fc).

11 % Define varying atoms
12 p(sat).
13 p(ta). p(fa).
14 p(tb). p(fb).
15 p(tc). p(fc).

17 % Define fixed atoms
18 z(sat').

20 % Model the cases where φ evaluates to true
```

```
21 % sat ← tb.
22 pos(c1, sat). neg(c1, tb).
23 % sat ← fa, tc.
24 pos(c2, sat). neg(c2, fa). neg(c2, tc).
25 % sat ← ta, tc.
26 pos(c3, sat). neg(c2, ta). neg(c2, tc).

28 % Saturize over the ∀-quantified variables, if φ evaluates to true
29 % tb ← sat.
30 pos(c4, tb). neg(c4, sat).
31 % fb ← sat.
32 pos(c5, fb). neg(c5, sat).

34 % Ensure satisfiability
35 % ← not sat.
36 pos(c6, sat').

38 % Guess truth values of variables a,b and c
39 % ta v fa.
40 pos(c7, ta). pos(c7, fa).
41 % tb v fb.
42 pos(c8, tb). pos(c8, fb).
43 % tc v fc.
44 pos(c9, tc). pos(c9, fc).
```

Listing 4.17: Input instance $\Pi_{\mathtt{input}_{\mathtt{ex}}}$ for solving QBF φ using $\Pi^2_{\mathtt{ASP}}$.

We are now going to present a direct encoding for disjunctive ASP for D-FLAT^2 without reducing to Circumscription. Before that, Listing 4.18 contains the implementation of a direct encoding of disjunctive ASP for D-FLAT. This encoding stores both satisfied rules and atoms in the item set (and does not use the default join implementation provided by D-FLAT). Parts at the beginning of it are relatively similar to Listing 4.3 and explained by comments, parts starting from Line 36 cover the optimization part and also occurs similarly to Listing 4.3. The different parts that require explanation are Line 30 and the block starting from Line 23. The lines of the block beginning in Line 23 cover satisfied rules among the two levels. In this sense, a rule is satisfied whenever any atom of its head is true w.r.t. the current item set, or the body of it is not. However, the second level is minorly different (see Line 30). This is due to the semantics of disjunctive ASP. Whenever we have that at the first level, some atom A is guessed to true (i.e. in the item set) and it appears in the negative part of the body of a rule, the rule is automatically satisfied (by the definition of the reduct). This exactly allows more potentially smaller models (minimal-model semantics) at the second level since in this case it is not required for A to be in the item set at the second level.

```
1 length(2). or(0). and(1).

3 % Make explicit when an atom is false or a rule is unsatisfied
```

```
4 false(S,X) ← sub(_,S), atNode(S,N), bag(N,X), not childItem(S,X).

6 % Guess a branch in the item tree for every child node
7 extend(0,R) ← root(R).
8 1 { extend(L+1,S) : sub(R,S) } 1 ← extend(L,R), L < 2.

10 % Only join child item sets that coincide on common atoms
11 ← extend(L,X), extend(L,Y), atom(A), childItem(X,A), false(Y,A).

13 % A child item set cannot be extended if a removed rule is
     unsatisfied by it
14 ← extend(_,S), rule(X), removed(X), false(S,X).

16 % True atoms and satisfied rules remain so unless removed
17 item(L,X) ← extend(L,S), childItem(S,X), current(X).

19 % Guess truth value of introduced atoms
20 { item(1,A;2,A) : atom(A), introduced(A) }.

22 % Through the guess, rules may become satisfied
23 item(1,R) ← current(R), current(A), head(R,A),     item(1,A).
24 item(1,R) ← current(R), current(A), pos(R,A),  not item(1,A).
25 item(1,R) ← current(R), current(A), neg(R,A),      item(1,A).
26 item(2,R) ← current(R), current(A), head(R,A),     item(2,A).
27 item(2,R) ← current(R), current(A), pos(R,A),  not item(2,A).

29 % If a negative body atom is true on the top level, the rule
     disappears from reduct (w.r.t. the top level)
30 item(2,R) ← current(R), current(A), neg(R,A),      item(1,A).

32 % Level 2 interpretation must not be bigger than level 1
33 ← atom(A), item(2,A), not item(1,A).

35 % Inherit (or extend) markers indicating that the level 2
     interpretation is smaller
36 item(2,smaller) ← extend(2,S), childItem(S,smaller).
37 item(2,smaller) ← atom(A), item(1,A), not item(2,A).

39 % Make sure that eventually only minimal models of the reduct
     survive
40 reject ← final, item(2,smaller).
41 accept ← final, not reject.
```

Listing 4.18: $\Pi_{\texttt{ASP}}$: D-FLAT encoding for solving disjunctive ASP.

In Listing 4.19, we present an alternative approach to solving disjunctive ASP via D-FLAT^2 (compare to Listing 4.18), which does not resort to Circumscription. In this encoding, $\Pi^{2'}_{\texttt{ASP}}$, we generate solution candidates for all interpretations that are candidates for being a classical model of the input program, which is specified by means

of the predicates `head`, `pos` and `neg`. A classical model M of a program P might be no Answer-Set because some $M' \subset M$ is a model of the reduct P^M. To check this, we generate additional item tree nodes that only serve as counter candidates (like M') to the nodes representing classical model candidates (like M). For any atom a from the current bag, if an item set contains a, then the corresponding interpretation sets a to true (otherwise to false). The item tree nodes representing counter candidates can additionally contain items of the form $\mathtt{r}(a)$. This signifies that the atom a is false in the respective counter candidate but true in the classical model candidates that reference this counter candidate (by means of their counter candidate pointers). In Lines 28 and 31, we make sure that any item tree node containing an item $\mathtt{r}(a)$ is marked with the "`pseudo`" item and will therefore not be considered as a solution but rather serves as a counter candidate only. Lines 36 and 37 are required to ensure that any counter candidate C of any solution candidate M only contains $r(a)$ for atoms a that are also contained in M.

```
1  1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
2  % Guess truth value/rule flag of introduced atoms
3  0 { item(A;r(A)) : atom(A), introduced(A) } 1.

5  % Make explicit when an atom is false or a rule is unsat
6  false(R,X)  ← childRow(R,N), bag(N,X), not childItem(R,X).
7  falser(R,X) ← childRow(R,N), bag(N,X), not childItem(R,r(X)).
8  unsat(R,X)  ← childRow(R,N), bag(N,X), not childAuxItem(R,X).

10 % Only join child item sets that coincide on common atoms
11 ← extend(X;Y), atom(A), childItem(X,A),    false(Y,A).
12 ← extend(X;Y), atom(A), childItem(X,r(A)), falser(Y,A).

14 % Only extend child item sets satisfying all removed rules
15 ← extend(S), rule(X), removed(X), unsat(S,X).

17 % True atoms and satisfied rules remain so unless removed
18 item(X)    ← extend(S), childItem(S,X),    current(X).
19 item(r(X)) ← extend(S), childItem(S,r(X)), current(X).

21 % Through the guess, rules may become satisfied
22 auxItem(R) ← current(R;A), head(R,A),     item(A).
23 auxItem(R) ← current(R;A), pos(R,A),  not item(A).
24 auxItem(R) ← current(R;A), neg(R,A),      item(A).

26 % Rule is not in reduct if a negative body atom is set to true
27 auxItem(R) ← current(R;A), neg(R,A),      item(r(A)).
28 auxItem(pseudo) ← item(r(X)), current(X).

30 % Inherit pseudo flag from child nodes
31 auxItem(pseudo) ← extend(R), childAuxItem(R,pseudo).

33 optItem(S)    ← atom(S), item(S).
```

```
35 % Prevents r(S) at level 2 (reduct) if S is not true at level 1
36 optItem(r(S)) ← atom(S), item(S), not auxItem(pseudo).
37 optItem(r(S)) ← atom(S), item(r(S)).
```

Listing 4.19: $\Pi^{2'}_{\texttt{ASP}}$: D-FLAT^2 encoding for solving disjunctive ASP directly.

CHAPTER 5

Benchmarks

5.1 Test Environment

In this chapter we focus on experiments for problems from the area of Abstract Argumentation.

5.1.1 Compared Tools

The benchmark results were gathered using D-FLAT, D-FLAT^2, ASPARTIX [EGW10] (for solving argumentation-related problems directly via ASP) and the popular Answer-Set Programming toolchain Potassco [GKK+11], in particular Gringo 4.4.0 and Clasp 3.1.1. D-FLAT and D-FLAT^2 internally also use ASP grounder Gringo 4.4.0 and solver Clasp 3.1.1. The results for ASPARTIX were produced with Gringo 3.0.5, as it is not fully compatible with newer versions of Gringo.

In order to assure a fair and comparable benchmark process, each tool only used one single core of the underlying CPU. D-FLAT^2 was used for benchmarks that use semi-normalized Tree Decompositions (using encodings of Section 4.2.4). D-FLAT also served for semi-normalized TDs (using encodings of Section 4.1.3) and the Potassco suite was chosen for benchmarks using the monolithic encodings of Section 5.2.

Since we actually discussed different D-FLAT and D-FLAT^2 encodings for both *admissible* and *preferred* semantics, we note here that this section contains results using the simplified $\Pi'_{\mathtt{admissible}}$ and therefore $\Pi^{2'}_{\mathtt{preferred}} = \Pi_{\mathtt{optAllItems}} \cup \Pi'_{\mathtt{admissible}}$ with $\Pi^{2'}_{\mathtt{preferred}} \equiv \Pi_{\mathtt{preferred}}$ (see Section 4.2.4). Moreover, we employ $\Pi'_{\mathtt{preferred}}$ instead of $\Pi_{\mathtt{preferred}}$ for our benchmarks. Of course for the more elaborate semi-stable semantics, we still use $\Pi^{2}_{\mathtt{semiStable}} = \Pi_{\mathtt{optForSemiStable}} \cup \Pi_{\mathtt{admissible}}$ here.

The implemented and presented encodings were tested in practice with many different seeds (and therefore different generated Tree Decompositions) and a huge number of nodes.

5.1.2 Test Framework

An important consideration for the design of the test framework was that it should be capable of running several benchmarks both concurrently and independently. This is due to the fact that D-FLAT uses *Htdecomp*, which is based on randomized heuristics and therefore produces different results depending on the given random seed. In order to generate reproducible results, the developed test framework uses different seeds for each of the ten *runs*. To be more concrete, if run number, instance size and semantics are the same, the tested toolkits will share the same generated seed. That means, for example, that every i^{th} run ($i \leq 5$) of the three compared tools of the problem type *counting* and the *semi-stable* semantics with an instance size of 55 shares the same seed.

The test framework was mainly designed for running different processes in parallel depending on the current requirements and capabilities of the CPU. This is achieved by monitoring whether an already started task has stopped working due to various reasons, so whenever any task terminates, the framework spawns new tasks until the maximum number of active tasks is reached, which can be dynamically changed.

5.1.3 Test Conditions

Every test was repeated five times and restricted to the conditions shown in Table 5.1. The plots in this chapter show the instance size in relation to either the time usage or the resident memory set. If time and memory resource usages did not differ significantly within the five runs of a given problem instance, this result is emphasized by showing the corresponding maximum usage, whereas for bigger deviations, i.e. instances with increasing tree-width, the arithmetical mean is plotted.

Restriction	**Violation Consequences**
A maximum of 3GB of virtual main memory was allowed.	Memory Error, Termination
A maximum of 3600 seconds of CPU time were permitted.	Timeout, Termination
Enumeration only: The main parts of the output (the particular extensions) were discarded.	

Table 5.1: Restrictions for the benchmarks.

Test Devices One x64-based PC was used to perform the benchmarks; it consisted of 16GB 1600Mhz, CL8 main memory and was powered by an Intel i5-3470 CPU and a Gigabyte GA-Z77-DS3H motherboard. For collecting data, a 64GB Samsung SSD 830

Series served as main disk. The underlying software was Xubuntu-14.04[1] with disabled swap to ensure that every process gets enough physical main memory and would not be slowed down due to swapping.

5.1.4 Problem Instances

The figures of this subsection are taken from [Cha12]. In order to make use of the gained Fixed-Parameter Tractability and to therefore show the advantages and disadvantages of D-FLAT, specific test cases were used. Although during the benchmarking process random instances also were used, it became apparent that it was not advisable to choose them, because of the fact that the performance of D-FLAT and D-FLAT^2 highly depends on the produced Tree Decomposition. Therefore in order to gain expressive results, the benchmarks concentrated primarily on the 8-Grid [Cha12] instances with different tree-widths. These "grid-based" instances, where vertices are arranged on an $n \times m$ matrix and edges connect horizontally, vertically and diagonally neighbouring vertices. Figure 5.1 contains an example of this instance type. The encircled part shows which nodes of the graph have to occur together in one node inside any Tree Decomposition. With these instances memory and time requirements are more stable when using D-FLAT resp. D-FLAT^2 since the design of them allows to set an upper-bound for the width of any potential Tree Decomposition. Clique instances of [Cha12] were also tried. An example of such an instance is seen in Figure 5.2. However, these instances only worked for semi-normalized decompositions; non-normalized ones caused timeouts even with small numbers of nodes in cliques and even smaller numbers of cliques.

5.2 Monolithic Encodings

The following encodings are taken from the *ASPARTIX*[2] project and are only included for completeness since they are used for result comparison purposes; these may be used for comparison with the corresponding D-FLAT resp. D-FLAT^2 encodings of Section 4.1.3 resp. Section 4.2. Descriptions are given as comments to the listings, some encodings require metasp (see Section 2.6).

```
1 %% an argument x defeats an argument y if x attacks y
2 defeat(X,Y) ← att(X,Y).

4 %% Guess a set S \subseteq A
5 in(X)  ← not out(X), arg(X).
6 out(X) ← not in(X), arg(X).
```

[1]To be more specific, the version available at http://cdimage.ubuntu.com/xubuntu/releases/trusty/release/xubuntu-14.04-desktop-amd64.iso.torrent was used.

[2]More information about ASPARTIX can be found at http://www.dbai.tuwien.ac.at/proj/argumentation/systempage/.

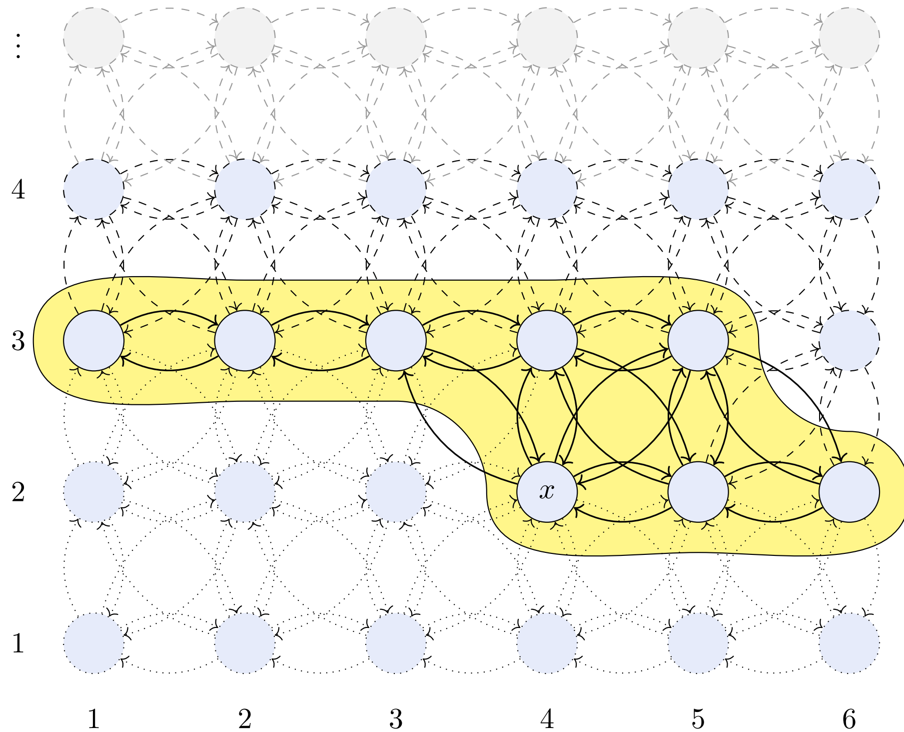

Figure 5.1: 8-Grid instance, ($n = 6 \times m$), *tree-width* = 7 for every $m \geq n$ [Cha12].

```
8 %% S has to be conflict-free
9 ← in(X), in(Y), defeat(X,Y).

11 %% The argument x is defeated by the set S
12 defeated(X) ← in(Y), defeat(Y,X).

14 %% S defeats all arguments which do not belong to S
15 ← out(X), not defeated(X).
```

Listing 5.1: Monolithic encoding for stable semantics.

```
1 %% an argument x defeats an argument y if x attacks y
2 defeat(X,Y) ← att(X,Y).

4 %% Guess a set S \subseteq A
5 in(X)  ← not out(X), arg(X).
6 out(X) ← not in(X), arg(X).

8 %% S has to be conflict-free
```

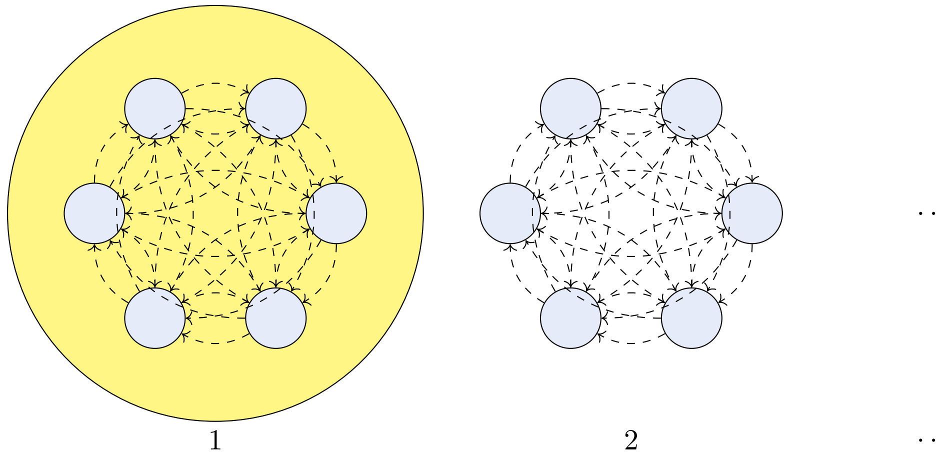

Figure 5.2: Clique instance, *tree-width* = 5 [Cha12].

```
9  ← in(X), in(Y), defeat(X,Y).

11 %% The argument x is defeated by the set S
12 defeated(X) ← in(Y), defeat(Y,X).

14 %% The argument x is not defended by S
15 not_defended(X) ← defeat(Y,X), not defeated(Y).

17 %% All arguments x \in S need to be defended by S
18 ← in(X), not_defended(X).
```

Listing 5.2: Monolithic encoding for admissible semantics.

```
1  %% an argument x defeats an argument y if x attacks y
2  defeat(X,Y) ← att(X,Y).

4  %% Guess a set S \subseteq A
5  in(X)  ← not out(X), arg(X).
6  out(X) ← not in(X), arg(X).

8  %% S has to be conflict-free
9  ← in(X), in(Y), defeat(X,Y).

11 %% The argument x is defeated by the set S
12 defeated(X) ← in(Y), defeat(Y,X).

14 %% The argument x is not defended by S
```

```
15 not_defended(X) ← defeat(Y,X), not defeated(Y).

17 %% admissible
18 ← in(X), not_defended(X).

20 %% Every argument which is defended by S belongs to S
21 ← out(X), not not_defended(X).
```

Listing 5.3: Monolithic encoding for complete semantics.

```
1 %% an argument x defeats an argument y if x attacks y
2 defeat(X,Y) ← att(X,Y).

4 %% Guess a set S \subseteq A
5 in(X)  ← not out(X), arg(X).
6 out(X) ← not in(X), arg(X).

8 %% S has to be conflict-free
9 ← in(X), in(Y), defeat(X,Y).

11 %% All arguments x \in S need to be defended by S (admissibility)
12 defeated(X) ← in(Y), defeat(Y,X).
13 ← in(X), defeat(Y,X), not defeated(Y).

15 %% Minimize out to get the subset-maximal admissible sets (needs
   metasp)
16 #minimize [ out(X) ].
```

Listing 5.4: Monolithic encoding for preferred semantics.

```
1 %% an argument x defeats an argument y if x attacks y
2 defeat(X,Y) ← att(X,Y).

4 %% Guess a set S \subseteq A
5 in(X)  ← not out(X), arg(X).
6 out(X) ← not in(X), arg(X).

8 %% S has to be conflict-free
9 ← in(X), in(Y), defeat(X,Y).

11 %% All arguments x \in S need to be defended by S (admissibility)
12 defeated(X) ← in(Y), defeat(Y,X).
```

```
13 ← in(X), defeat(Y,X), not defeated(Y).

15 %% Compute range of S
16 in_range(X) ← in(X).
17 in_range(X) ← in(Y), defeat(Y,X).
18 not_in_range(X) ← arg(X), not in_range(X).

20 %% Minimize not_in_range to get the subset−maximal sets wrt range
   (needs metasp)
21 #minimize [ not_in_range(X) ].
```

Listing 5.5: Monolithic encoding for semi-stable semantics.

5.3 Results

This section shortly presents and discusses the results of our selected benchmarks. First of all, D-FLAT^2 is compared with the Potassco toolkit, then information concerning performance differences between D-FLAT and D-FLAT^2 is gathered. Finally, we give an outlook concerning different basis semantics (admissible vs. complete semantics).

5.3.1 System Comparison

We considered the problem of enumerating all preferred extensions and compared the systems on grid-based instances with 40 to 65 nodes and tree-width 4. Figure 5.3 illustrates average runtimes and allocated memory together with the 95 % confidence interval. D-FLAT^2 showed the best performance, while D-FLAT is slightly slower and requires more memory. For ASPARTIX we observed timeouts for instances having more than 55 nodes.

5.3.2 Problem Comparison

As D-FLAT^2 is based on D-FLAT, we compared these systems on several problems. Moreover, we analyzed the cost of computing preferred and semi-stable sets compared to only obtaining admissible sets. As instances have much more admissible sets than preferred sets, which would bias a performance comparison when doing explicit enumeration, we considered the counting variants of these problems. Results are summarized in Figure 5.4, where again grid-based instances with tree-width 4 were tested.

When counting admissible sets, D-FLAT^2 requires slightly more time and memory than D-FLAT due to the overhead imposed by using reduced item trees instead of item trees. For preferred sets, the inefficiency of computing redundant counter candidates in D-FLAT becomes evident. On the contrary, in D-FLAT^2 the difference in runtime for

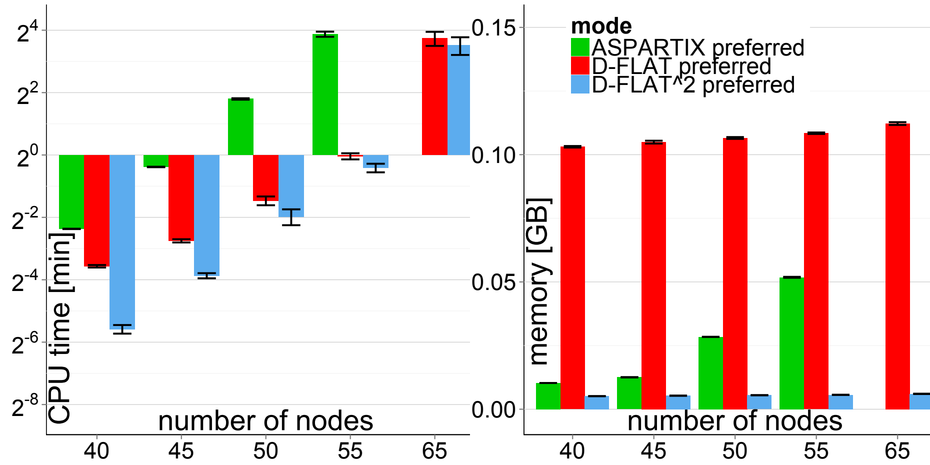

Figure 5.3: System comparison: Average CPU time (left) and maximum resident set (right).

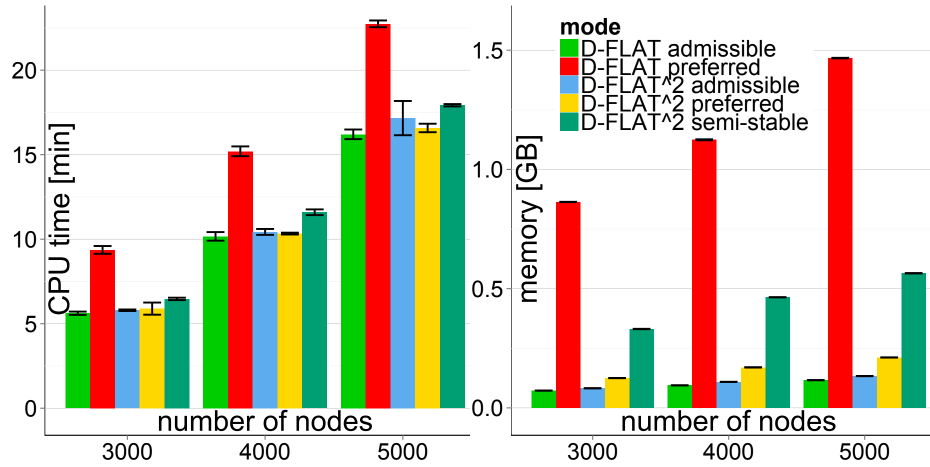

Figure 5.4: Problem comparison: Average CPU time (left) and maximum resident set (right).

counting preferred instead of admissible sets is barely measurable (i.e., within the 95% confidence interval). Here, we observed that subset maximization comes practically for free for instances of small tree-width. (We also observed this effect in a comparison of SAT with ⊆-MINIMAL SAT, where the overhead of D-FLAT was even larger.) Finally, for semi-stable sets, D-FLAT was not able solve instances with 500 vertices within the given memory limits. One reason is that for this problem many potential counter candidates have to be computed that turn out to be not even admissible. Thus, our two-phased approach of first computing (not necessarily maximal) solutions and then performing maximization obviously pays off in this case.

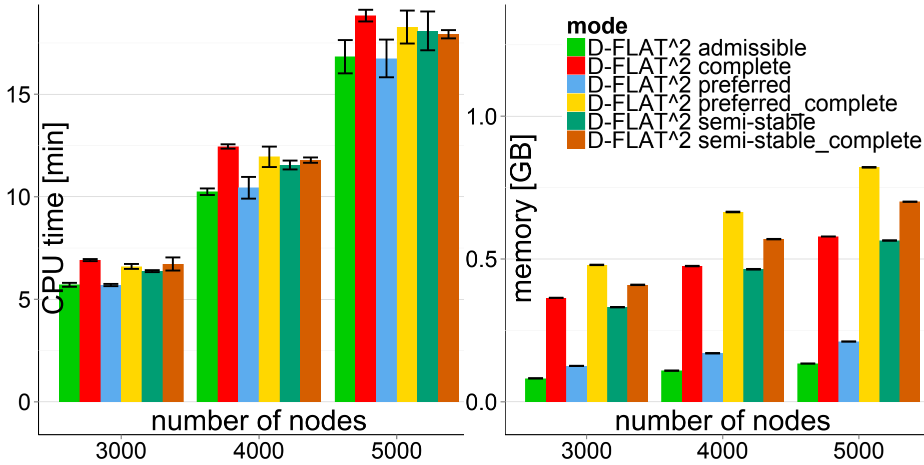

Figure 5.5: Basis semantics comparison: Average CPU time (left) and maximum resident set (right).

5.3.3 Basis Semantics Comparison

In the following, counting results of 8-Grid using D-FLAT^2 are discussed. As seen in Figure 5.5, there are only some statistically significant (non-overlapping confidence interval ranges) differences concerning runtime between the two basis semantics *admissible*, *complete* and the two more complex ones *semi-stable* and *preferred* w.r.t. the two basis semantics. However, *complete* semantics seem to be the most expensive one, that is, as in the previous section, we observe the more elaborate semantics cost only about the runtime its basis costs. Concerning memory usage, we obtain a different image (similar to before). Since there is a more involved guess in the encodings for complete-based semantics (and actually semi-stable since $\Pi^2_{\texttt{semiStable}}$ uses the more complex $\Pi_{\texttt{admissible}}$), we also observe the increasing memory requirements (compared to admissible-based ones).

However, it seems strange that with complete basis semantics $\Pi^2_{\texttt{preferred}_{\texttt{complete}}}$ needs more memory than $\Pi^2_{\texttt{semStable}_{\texttt{complete}}}$. An explanation might be that $\Pi^2_{\texttt{semiStable}_{\texttt{complete}}}$ sooner invalidates potential solution candidates during bottom-up traversal due to strict counter candidates involving nodes with the same item set (see Section 4.2.2).

To sum up, our general observation that the more involved encodings practically require resources of its basis semantics in case of small tree-width, can also be observed here.

CHAPTER 6

Conclusion

6.1 Summary

In this work we presented a method for solving problems involving subset minimization by means of DP on TDs. Given an algorithm for a version of the problem without minimization, our method allows us to perform the minimization tasks in an automatic and uniform way, thus making the development of such algorithms significantly easier. This thesis also presented an implementation called D-FLATˆ2 allowing for subset-minimization resp. maximization w.r.t. a user-specified optimization set in a simple way.

Users of D-FLATˆ2 are only required to provide an ASP program that specifies an algorithm for a version of the problem without optimization. Our method then models the optimization tasks in the DP algorithm in an automatic and uniform way, thus making the development of such algorithms significantly easier. We have outlined the details of this minimization procedure and shown its effectiveness and efficiency by experimenting with a prototype implementation.

Moreover, we applied D-FLATˆ2 to several problems by giving appropriate ASP encodings. Following our case study concerning Abstract Argumentation, this thesis even provides a new DP algorithm on TDs for semi-stable semantics along with a correctness proof.

Preliminary experiments indicate that the new approach brings significant advantages in terms of time and memory compared to previous solutions. This makes the method relevant especially for AI problems, as these often require some sort of subset optimization.

6.2 Further Work

There are still several tasks required in order to optimize the specified algorithm. This thesis presents a working version of D-FLATˆ2, but software like this requires permanent

maintenance and further research.

In the future, we also need to test our approach on more problems in order to further improve the D-FLATˆ2 system. Moreover, we plan to extend our approach to problems that are even higher in the polynomial hierarchy than the second level. That is, the algorithm presented in Chapter 4.2 shall be further generalized to arbitrary item trees and moreover to arbitrary, non-normalized TDs.

Ongoing work in particular includes a formal correctness proof concerning our approach of automatically generated parts of the DP.

Bibliography

[ABC$^+$14a] M. Abseher, B. Bliem, G. Charwat, F. Dusberger, M. Hecher, and S. Woltran. D-FLAT: Progress report. Technical Report DBAI-TR-2014-86, TU Wien, 2014.

[ABC$^+$14b] M. Abseher, B. Bliem, G. Charwat, F. Dusberger, M. Hecher, and S. Woltran. The D-FLAT system for dynamic programming on tree decompositions. In *JELIA*, volume 8761 of *LNCS*, pages 558–572. Springer, 2014.

[ABC$^+$15] M. Abseher, B. Bliem, G. Charwat, F. Dusberger, and S. Woltran. Computing secure sets in graphs using answer set programming. *J. of Logic and Computation, special issue of ASPOCP 2014*, 2015. Accepted for publication.

[ACP87] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987.

[ADG$^+$11] M. Aschinger, C. Drescher, G. Gottlob, P. Jeavons, and E. Thorstensen. Structural decomposition methods and what they are good for. In *STACS*, volume 9 of *LIPIcs*, pages 12–28. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[BCHW15a] B. Bliem, G. Charwat, M. Hecher, and S. Woltran. D-FLAT^2: Subset minimization in dynamic programming on tree decompositions made easy. In *ASPOCP*, 2015.

[BCHW15b] B. Bliem, G. Charwat, M. Hecher, and S. Woltran. Optimization of tree-decomposition-based dynamic programming for AI problems. Unpublished draft. Available at `http://dbai.tuwien.ac.at/proj/dflat/dflat-squared-draft.pdf`, 2015.

[BET11] G. Brewka, T. Eiter, and M. Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[BK10] H. J. Bodlaender and A. M. C. A. Koster. Treewidth computations I. Upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.

[Bli12] B. Bliem. Decompose, guess & check: declarative problem solving on tree decompositions. Master's thesis, Vienna University of Technology, 2012.

[BMW12] B. Bliem, M. Morak, and S. Woltran. D-FLAT: Declarative problem solving using tree decompositions and answer-set programming. *TPLP*, 12:445–464, 2012.

[Cha12] G. Charwat. Tree-decomposition based algorithms for abstract argumentation frameworks. Master's thesis, Vienna University of Technology, 2012.

[Cou90] B. Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

[DB02] Paul E. Dunne and Trevor J. M. Bench-Capon. Coherence in finite argument systems. *Artif. Intell.*, 141(1/2):187–203, 2002.

[Dec03] R. Dechter. *Constraint Processing.* Morgan Kaufmann, 2003.

[DF99] R. G. Downey and M. R. Fellows. *Parameterized Complexity.* Monographs in Computer Science. Springer, 1999.

[DGG+08] A. Dermaku, T. Ganzow, G. Gottlob, B. J. McMahan, N. Musliu, and M. Samer. Heuristic methods for hypertree decomposition. In *MICAI*, volume 5317 of *LNCS*, pages 1–11. Springer, 2008.

[DPW12] W. Dvořák, R. Pichler, and S. Woltran. Towards fixed-parameter tractable algorithms for abstract argumentation. *Artif. Intell.*, 186:1–37, 2012.

[DSW12] W. Dvořák, S. Szeider, and S. Woltran. Abstract argumentation via monadic second order logic. In *SCM*, volume 7520 of *LNCS*, pages 85–98. Springer, 2012.

[Dun95] P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–357, 1995.

[Dun07] P. E. Dunne. Computational properties of argument systems satisfying graph-theoretic constraints. *Artif. Intell.*, 171(10-15):701–729, 2007.

[Dvo12] Wolfgang Dvorák. Computationale aspekte der abstrakten argumentation. In *Ausgezeichnete Informatikdissertationen 2012*, pages 61–70, 2012.

[DW10] Wolfgang Dvorák and Stefan Woltran. Complexity of semi-stable and stage semantics in argumentation frameworks. *Inf. Process. Lett.*, 110(11):425–430, 2010.

[EG95] T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15(3-4):289–323, 1995.

[EGM97] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Trans. Database Syst.*, 22(3):364–418, September 1997.

[EGW10] U. Egly, S. A. Gaggl, and S. Woltran. Answer-set programming encodings for argumentation frameworks. *Argument and Computation*, 1(2):147–177, 2010.

[EP06] T. Eiter and A. Polleres. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *TPLP*, 6(1-2):23–60, 2006.

[GKK+11] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The potsdam answer set solving collection. *AI Communications*, 24:107–124, 2011.

[GKS11] M. Gebser, R. Kaminski, and T. Schaub. Complex optimization in answer set programming. *TPLP*, 11(4-5):821–839, 2011.

[GLS02] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *J. of Computer and System Sciences*, 64(3):579 – 627, 2002.

[GPW10a] G. Gottlob, R. Pichler, and F. Wei. Bounded treewidth as a key to tractability of knowledge representation and reasoning. *Artif. Intell.*, 174(1):105–132, 2010.

[GPW10b] G. Gottlob, R. Pichler, and F. Wei. Tractable database design and datalog abduction through bounded treewidth. *Inf. Syst.*, 35(3):278–298, 2010.

[GS08] G. Gottlob and S. Szeider. Fixed-parameter algorithms for artificial intelligence, constraint satisfaction and database problems. *Comput. J.*, 51(3):303–325, 2008.

[JPRW08] M. Jakl, R. Pichler, S. Rümmele, and S. Woltran. Fast counting with bounded treewidth. In *LPAR*, volume 5330 of *LNCS*, pages 436–450. Springer, 2008.

[JPW09] M. Jakl, R. Pichler, and S. Woltran. Answer-set programming with bounded treewidth. In *IJCAI*, pages 816–822. AAAI Press, 2009.

[Klo94] T. Kloks. *Treewidth: Computations and Approximations*, volume 842 of *LNCS*. Springer, 1994.

[KLR11] J. Kneis, A. Langer, and P. Rossmanith. Courcelle's theorem – a game-theoretic approach. *Discrete Optimization*, 8(4):568–594, 2011.

[LPV01] V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Trans. Comput. Log.*, 2:526–541, 2001.

[LRS00] N. Leone, R. Rosati, and F. Scarcello. Enhancing answer set planning. Technical Report DBAI-TR-2000-37, TU Wien, 2000.

[McC80] J. McCarthy. Circumscription – a form of non-monotonic reasoning. *Artif. Intell.*, 13(1-2):27–39, 1980.

[Nie06] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms.* Oxford Lecture Series in Mathematics and its Applications. OUP, 2006.

[PTW09] D. Pearce, H. Tompits, and S. Woltran. Characterising equilibrium logic and nested logic programs: Reductions and complexity. *TPLP*, 9(5):565–616, 2009.

[RS84] N. Robertson and P. D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

[SS10] M. Samer and S. Szeider. Algorithms for propositional model counting. *J. Discrete Algorithms*, 8(1):50–64, 2010.