FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Tree-Decomposition based Algorithms for Abstract Argumentation Frameworks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Günther Charwat
Matrikelnummer 0625026

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Dr. Techn. Stefan Woltran
Mitwirkung: Projektass. Dipl.-Ing. Wolfgang Dvořák

Wien, 12. Februar 2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Tree-Decomposition based Algorithms for Abstract Argumentation Frameworks

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Günther Charwat
Registration Number 0625026

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Privatdoz. Dipl.-Ing. Dr. Techn. Stefan Woltran
Assistance: Projektass. Dipl.-Ing. Wolfgang Dvořák

Vienna, 12. Februar 2012 _____ _____
(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Günther Charwat
Zemlinskygasse 88, 1230 Wien

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.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

First of all, I want to thank my advisor Stefan for introducing me to the field of abstract argumentation and for his invaluable support throughout this thesis. Furthermore I want to thank my co-advisor Wolfgang who always took the time for giving me tips, especially when I got stuck while working out the details of the correctness proofs.
Thanks also to Markus and my colleagues for providing a flexible working environment that allowed me to focus on my thesis.
I am grateful for the continuous support of my parents Christine and Erich who made it possible to study at the Vienna University of Technology. Furthermore I want to thank my siblings Silvia and Verena for the fun we have together and for them being always there for me.
Many thanks to my friends, especially to Stefan, Clemens and Stefan for the free time we spend together, providing a great balance to my studies. I also want to thank Paulus for whom I did not have a lot of time in the last months but who, nevertheless, always cheered me up.

Günther

Abstract

In recent years abstract argumentation frameworks (AFs) have emerged as an important research field in artificial intelligence. AFs are defined as directed graphs consisting of arguments (nodes in the graph) and attack relations (edges in the graph).

We are interested in the selection of 'appropriate' arguments in an AF. The sets of appropriate arguments are then called the extensions of the AF. In abstract argumentation this 'appropriateness' can be defined by a wide variety of different semantics. For most semantics the computation of extensions for an AF is in general computationally hard. But by binding a certain problem parameter to a fixed constant many of those intractable problems become tractable. One important parameter for graph problems is the tree-width of a graph which represents the tree-likeliness of the AF. The tree-width is defined on tree decompositions of an AF. A tree decomposition is a mapping of a graph to a tree where the latter can be used to evaluate a certain problem efficiently.

In this thesis we introduce novel algorithms for stable and complete semantics that are defined on so-called normalized tree decompositions. Furthermore we present a novel algorithm for admissible semantics based on semi-normalized tree decompositions that generalizes an existing algorithm on normalized tree decompositions. The advantage of semi-normalized (compared to normalized) tree decompositions is that they consist of less nodes and thus the overall depth of a semi-normalized tree decomposition is lower.

Besides the algorithms and the correctness proofs thereof we provide an implementation on basis of a purpose-built framework for algorithms on tree decompositions. This allows us to compare our novel algorithm for admissible semantics on semi-normalized tree decompositions to the existing algorithm on normalized tree decompositions. Our experimental results show that our novel implementation outperforms the existing one.

Kurzfassung

In den letzten Jahren gewannen *Abstract Argumentation Frameworks* (AFs) im Forschungsbereich der künstlichen Intelligenz immer mehr an Bedeutung. AFs sind als gerichtete Graphen definiert, welche aus Argumenten (Knoten im Graphen) und Angriffsbeziehungen (Kanten im Graphen) bestehen.

Wir beschäftigen uns nun mit der Auswahl von „passenden“ Argumenten aus dem AF. Diese werden auch als *Extensions* des AFs bezeichnet. Es existiert eine Vielzahl von Semantiken, die definieren, welche Argumente als „passend“ angesehen werden. Allerdings ist die Selektion der Argumente auf Basis einer Semantik in den meisten Fällen rechnerisch extrem aufwendig. Eine Idee aus dem Bereich der Komplexitätstheorie besteht nun darin, einen Problemparameter zu fixieren, wodurch das Problem im Allgemeinen leichter lösbar wird. Für Probleme auf Graphen ist der Parameter der *Tree-Width* relevant, welcher, informell gesagt, angibt, wie sehr der Graph einem Baum ähnelt. Die Tree-Width ist auf sogenannten *Tree Decompositions* definiert welche dazu verwendet werden können, ein Problem effizient zu lösen.

In dieser Diplomarbeit werden neue Algorithmen für die Berechnung von Extensions für die Semantiken *Stable* und *Complete* auf Basis von normalisierten Tree Decompositions präsentiert. Weiters wird ein neuer Algorithmus für die Semantik *Admissible* auf semi-normalisierten Tree Decompositions entwickelt. Der Unterschied zwischen normalisierten und semi-normalisierten Tree Decompositions besteht darin, dass letztere weniger Knoten enthalten und damit die gesamte Baumtiefe geringer ist.

Zusätzlich zu den Korrektheitsbeweisen der Algorithmen werden diese mithilfe eines bereits existierenden Frameworks implementiert, welches speziell dafür entwickelt wurde, auf Tree Decompositions basierende Algorithmen zu entwerfen. Der Algorithmus für die Semantik *Admissible* auf semi-normalisierten Tree Decompositions wird mit dem Algorithmus auf normalisierten Tree Decompositions verglichen. Die experimentellen Resultate zeigen, dass die semi-normalisierte Implementierung durchwegs schneller als die bereits existierende ist.

Contents

CHAPTER 1

Introduction

Problem Overview

Argumentation plays an important role in our everyday lives: In order to persuade someone we use arguments that support our own opinion or rebut our opponent's arguments. In debates we discuss a certain point of view with the ultimate goal of convincing our audience or compromising about a certain decision.

The research field of *argumentation* in *artificial intelligence* deals with the computer-based evaluation of arguments with the ultimate goal of drawing conclusions from a set of arguments. This can be described as a process that consists of the following tasks:

(1) Formalize the natural-language arguments.

(2) Identify conflicts between the arguments.

(3) Abstract away from the internal structure of arguments.

(4) Evaluate relations between arguments.

(5) Finally, draw conclusions.

The first step focuses on the translation of natural-language arguments into a computer-understandable representation of the arguments. Next, the arguments have to be analyzed in order to obtain information about the relations between arguments. In the third step the internal information about the arguments is abstracted away. What remains are the relations between arguments. Then, in the fourth step, it is possible to evaluate the arguments solely based on their relations and conflicts in-between them. Finally, we can draw our conclusions.

In this thesis we focus on step (4), the evaluation of relations between arguments. We do not have to deal with domain-specific characteristics of arguments (as this information is abstracted away in the previous step). The main advantage is that it is possible to solely work on relations between arguments. It is therefore possible to define a general-purpose strategy for the resolution of conflicts that is not restricted to any domain. In artificial intelligence this field of research is called *abstract argumentation*.

The main idea of abstracting away from concrete arguments and focusing on the relations between arguments dates back to the work of Dung [1995]. Dung introduced *abstract argumentation frameworks* (or AFs for short). An abstract argumentation framework is defined as a directed graph that consists of *arguments* (nodes in the graph) and *attack relations* (directed edges in the graph).

Based on the relations between arguments we are now interested in the selection of 'appropriate' arguments of an AF. The sets of 'appropriate' arguments are called the *extensions* of the AF. There exists a wide variety of different semantics that define which arguments are considered to be 'appropriate'. As an example consider an argument a that attacks an argument b. Intuitively, we can not select both arguments as there would be a conflict within our selected arguments. Baroni and Giacomin [2009] give a good overview of proposed semantics for abstract argumentation.

Different work (e.g. [Coste-Marquis et al., 2005; Dunne and Wooldridge, 2009; Dvořák and Woltran, 2010]) showed that the computation of extensions for most semantics is, in general, computationally hard. Now, the idea is to identify problem fragments for which the computation of extensions becomes tractable. One approach is based on the observation that by binding some problem parameter to a fixed constant many of the intractable problems can become tractable [Courcelle, 1990]. An important parameter for graph problems is the *tree-width* which represents the *tree-likeliness* of the graph (or, in our case, the argumentation framework). Dunne [2007] showed that many argumentation problems can be solved in linear time for AFs of bounded tree-width. The tree-width is defined on *tree decompositions* (originally introduced by Robertson and Seymour [1984]) of an AF. A tree decomposition is a mapping of a graph to a tree where the latter can be used to evaluate a certain problem efficiently. Informally, in a tree decomposition the nodes of the tree contain vertices of the original graph. Furthermore, vertices that are connected in the original graph have to appear together in at least one node of the tree. Finally, nodes that contain the same vertex of the graph have to be connected.

It is then possible to define algorithms on tree decompositions for a certain semantics. The tree is traversed in bottom-up order and the overall result is returned in the root node.

Main Contributions

In this thesis we introduce three novel algorithms for different semantics that are defined on tree decompositions. The algorithms for *stable* and *complete* semantics are defined on so-called normalized tree decompositions. The novel algorithm for *admissible* semantics is based on semi-normalized tree decompositions. It generalizes the existing algorithm on normalized tree decompositions as introduced by Dvořák et al. [2010a]. The advantage of semi-normalized tree decompositions, compared to normalized tree decompositions, is that they consist of less nodes and thus the overall depth of a semi-normalized tree decomposition is lower.

The algorithms are implemented on basis of the already-existing SHARP framework (Smart Hypertree decomposition-based Algorithm fRamework for Parameterized problems) [Morak, 2012]. The framework provides the necessary interfaces for the implementation of algorithms that are based on tree decompositions. It takes care of the overall workflow of the algorithm and the generation of tree decompositions. We provide definitions of the data structures and

implementations for the nodes of the tree decomposition. The node implementations reflect the algorithm definitions for the computation of stable, complete and admissible extensions.

The implementation of our algorithms is included in the dynPARTIX (Dynamic Programming Argumentation Reasoning Tool) project and is publicly available[1].

To sum it up, we provide

- novel algorithms for stable and complete semantics that are based on normalized tree decompositions,
- a novel algorithm for admissible semantics that is based on semi-normalized tree decompositions,
- correctness proofs thereof,
- an implementation of the algorithms that is based on an already-existing framework for algorithms on tree decompositions and
- experimental results that show that the the semi-normalized implementation for admissible semantics outperforms the normalized implementation.

Related Work

Approaches based on fixed-parameter tractability are for example presented in [Ordyniak and Szeider, 2011; Dvořák et al., 2010b,a]. Ordyniak and Szeider [2011] analyze special types of argumentation frameworks, such as *acyclic* and *noeven* frameworks. Acyclic AFs do not contain any directed cycles of attack relations whereas noeven AFs do not contain any cycles of even length. For certain decision problems (namely skeptical and credulous acceptance which we will introduce later on in this thesis) they show that the problems become tractable in case the 'distance' of a given AF to such classes is bounded. Dvořák et al. [2010b] present algorithms for argumentation frameworks that are not bound to tree-width but another parameter, namely *clique-width*. The approach in [Dvořák et al., 2010a] provides the basis for our novel algorithms. In there the complexity of *admissible* and *preferred* semantics for certain decision problems is analyzed. All algorithms are defined on normalized tree decompositions.

The computation of extensions on basis of *direct* algorithms is, for example, elaborated in [Modgil and Caminada, 2009; Verheij, 2007]. In there the presented algorithms are defined directly on the underlying argumentation frameworks (instead of tree decompositions as in our case). Furthermore, *reduction-based* algorithms are, for example, presented in [Egly and Woltran, 2006; Amgoud and Devred, 2011; Egly et al., 2010]. Reduction-based algorithms define some kind of mapping between argumentation frameworks (or properties thereof) and other languages.

[1] `http://www.dbai.tuwien.ac.at/research/project/argumentation/dynpartix/`

Organization

In Chapter 2 we introduce argumentation frameworks. Furthermore we define semantics for argumentation frameworks and especially focus on admissible, stable and complete semantics. Furthermore we present important decision problems, namely the questions for the existence of an extension, if an argument is contained in any extension (*credulous acceptance*) and if an argument is contained in every extension (*skeptical acceptance*).

In Chapter 3 we present our novel algorithms. We first explain the general ideas behind the algorithms on basis of admissible semantics. Then, we introduce our algorithms for stable and complete semantics on normalized tree decompositions and our algorithm for admissible semantics on semi-normalized tree decompositions.

Chapter 4 deals with the implementation of our algorithms. Thus, we first give a brief overview on the SHARP framework that provides the basis for our algorithms. We furthermore describe the implementation for the nodes in the tree decomposition. Finally, we give a system description of the dynPARTIX software that includes the implementations of our algorithms.

In Chapter 5 we evaluate our implementation of admissible semantics for semi-normalized tree decompositions and compare its run-time to that of the implementation for normalized tree decompositions.

Finally, in Chapter 6, we discuss the obtained results and present ideas for future work on the topics covered in this thesis.

CHAPTER 2

Background

Section 2.1 gives an overview of argumentation in artificial intelligence (AI). It provides the necessary theoretical basis for algorithms on argumentation frameworks. Furthermore we outline the steps that are needed to evaluate natural-language arguments, analyze them and draw conclusions.

In Section 2.2 we introduce abstract argumentation frameworks (AFs) which represent the relations between arguments, abstracting away the concrete contents of arguments.

An important part of argumentation is the selection of 'appropriate' arguments: This appropriateness can be defined by a wide range of different semantics. The selection of 'appropriate' arguments for most semantics is computationally hard. In Section 2.3 we present different proposed semantics for argumentation frameworks. We describe admissible, stable, complete and preferred semantics in detail.

In Section 2.4 we present several decision problems, namely skeptical and credulous acceptance as well as the question for the existence of an extension. Furthermore we give a brief overview of current complexity-theoretic results for decision problems.

In Section 2.5 we introduce the concept of tree decompositions and define normalized as well as semi-normalized tree decompositions. The algorithms for the semantics presented in this thesis are defined on these types of decompositions. Furthermore we introduce the concept of fixed parameter tractability. By binding some problem parameter to a fixed constant it is possible to reduce the overall complexity of the (decision) problem.

2.1 Argumentation

Overview

Argumentation is of significant relevance in our daily lives: We have to make decisions based on incomplete and contradicting information. In debates we have to convince our audience by providing arguments that support our own goal or rebut our opponent's arguments. Arguments may be based on personal feelings or opinions which leads to an important observation:

It can be seen that the goal of arguments and argumentation (as opposed to mathematical reasoning) lies on the persuasion of others (as opposed to mathematical proofs). Bench-Capon and Dunne [2007] give a good comparison of the nature of argumentation and mathematical reasoning:

- In mathematical reasoning, premises are *consistent*. They consist of closed concepts. In argumentation, premises may rely on background *assumptions*.
- In contrast to mathematical reasoning, arguments may be incomplete or are subject to change.
- A (correct) mathematical proof is *final*, the conclusion always remains valid. Arguments, on the other hand, are *defeasible*. It may be the case that new or changed information alters the output.
- As stated in [Bench-Capon and Dunne, 2007], "Proof is demonstration whereas argument is persuasion". Reasoning and conclusions are entirely objective whereas arguments are based on personal opinions of feelings.

Let us consider the following example:

Example 2.1. After a hard day of work Peter wants to go out for food. A nice restaurant nearby offers several dishes but Peter is unsure what he wants to eat: He likes both meat and fish but does not want to eat them at the same time. Vegetarian food is ok for him as well but only if the restaurant is out of the other dishes. A friend once told him that it is unhealthy to eat meat. But Peter has not eaten any meat for weeks and he is sure that a big steak once in a while will keep him healthy. He is convinced that vegetarian food would not help him staying healthy.

Let us analyze this example: It is solely Peter's *preference* not to combine fish and meat in one dish. By no means this represents a *fact*. Furthermore, his friend states that is is 'unhealthy to eat meat'. According to that (and assuming that Peter wants to stay healthy) he should not eat meat. We see that the argument 'meat is unhealthy' *attacks* Peter's choice for meat. He *defeats* his friend's argument by stating that a 'steak once in a while will keep him healthy'. 'Once in a while' is very *vague*. On could *interpret* that the way he wants and it also depends on the overall *context* of the argumentation.

We can now introduce a new argument that *attacks* Peter's choice for fish. This shows that arguments may be incomplete and can change:

Example 2.2. (Example 2.1 extended) Peter is also worried about the environment: The oceans are overfished and he feels that he should not order fish.

The new argument attacks Peter's choice for fish. If we take it into account Peter may should not eat fish. Furthermore it is possible to have *inconsistent* arguments:

Example 2.3. (Example 2.2 extended) Another friend of Peter always eats meat and thinks that he is only healthy because of the positive correlation between meat and health. But his friend does not feel healthy at all.

The premise of his friend's argument (he is healthy) is immediately contradicted by 'him not looking healthy at all', i.e. the premises of the argument contradict themselves.

Reasoning and argumentation are of particular importance in many different fields. In medicine, doctors have to decide which medical treatment is the best for their patients. The symptoms may indicate a certain disease and the doctors have to decide for or against a treatment. Some drugs may have side effects or are not compatible with other drugs. In law, judges have to make decisions based on laws, evidence and statements. Oftentimes, laws are ambiguously defined and statements from defendant and prosecutor contradict each other.

From Natural Language to Argumentation in Artificial Intelligence

We gave a brief introduction into the wide variety of arguments in real life and its comparison to mathematical reasoning. In order to work on arguments we have to formalize them.

One approach is proposed by Besnard and Hunter [2001]. They use classical (propositional) logic to represent argumentation.

What all arguments have in common is that they consist of premises and conclusions that can be drawn from the supporting premises. The premises Φ are defined in a knowledge base Δ that consists of propositional formalæ. These formulæ represent translations of natural-language arguments. An argument is represented by a pair $A = \langle \Phi, \alpha \rangle$ such that $\Phi \nvdash \bot$, $\Phi \vdash \alpha$ and Φ is a minimal subset of a knowledge base Δ satisfying $\Phi \vdash \alpha$. α represents the the conclusions (or *claim*) drawn from Φ whereas Φ is also called the *support* for an argument.

Example 2.4. Let $\Delta = \{m, f, m \to \neg f, f \to \neg m\}$ be a knowledge base resulting from the first part of Example 2.1. Informally, m represents that Peter eats meat, f stands for his choice for fish and the formulæ $m \to \neg f$ and $f \to \neg m$ represent that if he chooses meat he does not eat fish and vice versa. Possible formalized arguments could be

$$\langle \{m, m \to \neg f\}, \neg f \rangle \tag{2.1}$$

$$\langle \{f, f \to \neg m\}, \neg m \rangle \tag{2.2}$$

We refer the interested reader to [Besnard and Hunter, 2001] for details on this translation technique. In the paper, the authors define *undercuts* and *rebuttals* that describe the type of attack on arguments. An argument is an undercut for another argument if its claim directly attacks the support for another argument . In Example 2.4 we have an undercut because the claim $\neg f$ in (2.1) contradicts the support f of (2.2). A conflict between arguments is called rebuttal if the claims of two arguments contradict each other.

Other approaches rely on defeasible logic programming (DeLP): García and Simari [2004] propose the use of *facts* (ground atoms or negated ground atoms) and *strict* as well as *defeasible rules*. Strict rules correspond to basic rules introduced by Lifschitz [1996]. In comparison to strict rules, defeasible rules represent defeasible knowledge, i.e. tentative information that may

be used if nothing could be posed against it. Every rule consists of a literal, the head, that represents the claim and a body which is a non-empty set of literals. The body represents the support for the argument. The symbols $Head \leftarrow Body$ and $Head \prec Body$ syntactically denote strict and defeasible rules.

Example 2.5. Let us consider the follow rules that may be derived from example 2.1:

$$\sim f \leftarrow m$$
$$u \prec m$$

In natural language, $\sim f \leftarrow m$ would translate to 'If Peter eats meat he does not eat fish.' Here, $\sim$ denotes the (strong) negation. $u \prec m$ says that 'usually, meat is unhealthy'.

This approach allows applications to deal with incomplete or contradicting information in a natural way: By stating weak rules we can formally represent information that may be out-ruled. In order to derive conclusions we have to look for counter arguments that defeat our arguments.

2.2 Abstract Argumentation Frameworks

In the last section we described two different approaches that can be used for the translation of natural-language arguments into formal arguments and we described possible formal definitions of arguments. In abstract argumentation we are not interested in the internal structure of arguments. We abstract away the premises and conclusions and focus on the relations between arguments. The arguments and their relations can be represented in an *argumentation framework* (or AF for short). The most popular formalization for AFs was introduced by Dung [1995]:

Definition 2.1. *An* argumentation framework *is a pair $AF = \langle A, R \rangle$ where A is a set of arguments and $R \subseteq A \times A$ is the attack relation, representing attacks among arguments.*

Remark 2.2. *In the following we often write $a \rightarrowtail b$ in order to denote an attack $(a, b) \in R$. We say that a attacks b if $(a, b) \in R$.*

Furthermore we write $S \rightarrowtail a$ to denote that at least one argument s of the set of arguments S attacks a, i.e. there exists an $s \in S$ where $s \rightarrowtail a$. Due to symmetry, $a \rightarrowtail S$ means that a attacks at least one argument of S.

Finally, we write $S \rightarrowtail T$ to denote that there exists a $s \in S$ and a $t \in T$ such that $s \rightarrowtail t$.

We see that the internal structure or meaning of arguments is of no relevance in abstract argumentation frameworks. Thus, a single AF could represent many different situations: An argument a could represent a politician's statement or, at the same time, the weather forecast for a region. What remains are the abstract arguments and the relations between them. The advantage lies in the ability to analyze arguments independent from any specific context or situation. This allows for more general definitions of semantics (as we will see in the next section) and algorithms (see Chapter 3). On the other hand, we sometimes loose crucial information about the concrete problem that could help in the evaluation step of arguments and counter arguments.

Based on Definition 2.1 we can construct one possible AF that could result from our example of Peter's choice for food (see Example 2.3):

Example 2.6. (Example 2.3 continued) Let $F = \langle A, R \rangle$ be an AF such that:

$$A = \{a, b, c, d, e, f, g\}$$
$$R = \{(a, b), (b, a), (a, c), (b, c), (c, d), (d, e), (e, b), (f, a), (e, g), (g, g)\}$$

Informally, we assign the following meaning to the labels used in Example 2.6:

a	Peter orders fish.
b	Peter orders meat.
c	Peter orders vegetarian food.
d	Eating meat once in a while is healthy.
e	Meat is unhealthy.
f	The oceans are overfished.
g	Meat is healthy.

Table 2.1: Possible meaning for argument labels of Example 2.6

Note that, as already mentioned, labels in an AF generally do not have a meaning. They could represent any arguments as long as the attack relations 'fit' the natural-language arguments. In the following we will refer to the assignment of labels to nodes as it helps understanding the idea behind different semantics on AFs (see Section 2.3).

A nice feature of argumentation frameworks is that they can be represented as graphs where the nodes represent the arguments and the edges represent the attack relations. Hence, we can continue Example 2.6 and construct a graph as depicted in figure 2.1.

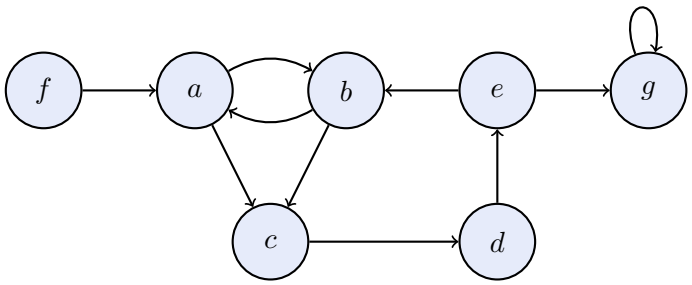


Figure 2.1: Example Argumentation Framework, represented as Graph

2.3 Semantics of Argumentation Frameworks

Overview

The representation of argumentation frameworks as graphs is extremely helpful when it comes to the analysis of relations between arguments and the selection of 'appropriate' arguments. Consider our example AF from Figure 2.1 and the argument g. Intuitively, it will never be possible to select g because of the self-attack $g \rightarrowtail g$: The argument somehow contradicts itself (Note that we stated that g represents that 'meat is healthy' but as stated in Example 2.3 'Peter's friend does not look healthy at all').

Another interesting observation can be gained from the attacks $a \rightarrowtail b$ and $b \rightarrowtail a$. Obviously, there exists some kind of conflict between those two arguments: If we choose a we can not choose b and vice-versa. If we follow our example this could represent that Peter does not want to eat fish and meat at the same time. But what happens if we additionally consider argument f? As $f \rightarrowtail a$, one could argue that if f is selected as an 'appropriate' argument, a can not be selected. Furthermore, consider the subgraph $F' = \langle\{a, b, f\}, \{(a, b), (b, a), (f, a)\}\rangle$. If we select f, a is attacked. Then, it could be possible to select b because f defends b against the attack from a.

Semantics on argumentation frameworks formally define which selection of arguments is 'appropriate'. Therefore we will now analyze our observations and give formal definitions of several proposed semantics. Baroni and Giacomin [2009] state that we are interested in the *justification state* of selected arguments. 'Intuitively, an argument is regarded as justified if it has some way to survive the attacks it receives, as not justified (or rejected) otherwise' [Baroni and Giacomin, 2009].

We already identified several *conflicts* in our example AF. A main property of all semantics presented in this thesis is the *conflict-freeness* of selected arguments. Intuitively, a set of arguments is conflict-free if no argument in the set attacks another one from the set (or itself).

Definition 2.3. *Let $F = \langle A, R\rangle$ be an AF. A set $S \subseteq A$ is* conflict-free (in F)*, iff there are no $a, b \in S$, such that $(a, b) \in R$. We denote the collection of sets which are conflict-free (in F) by $cf(F)$.*

Example 2.7. The conflict-free set of our example graph F (see figure 2.1) is

$$cf(F) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{a, d\}, \{a, e\}, \{b, d\}, \{b, f\}, \{b, d, f\}, \{c, e\}, \{c, e, f\}, \{d, f\}\}$$

Another important property is the notion of *acceptable* or *defended* arguments. Note that in literature *acceptable* and *defended* are equivalently used. In our example AF b would be defended if we select the arguments d and f, i.e. every argument that attacks b is attacked by another argument from our selected set of arguments. Additionally, f is always defended as it is not attacked by any other argument. Formally, this is defined as follows:

Definition 2.4. *Let $F = \langle A, R\rangle$ be an AF. An argument a is* defended *by S in F iff for each $b \in A$ with $b \rightarrowtail a$ there exists a $c \in S$ such that $c \rightarrowtail b$.*

Notions of Argumentation Semantics

In literature two different styles for the notion of argumentation semantics exist.

Extension-based semantics describe how to obtain a set of *extensions* from an AF based on certain criteria. An extension is a subset of arguments from an AF that are, based on the definition of the semantics, 'acceptable' or 'appropriate'. In [Baroni and Giacomin, 2007] the evaluation of extension-based semantics is analyzed.

In *labelling-based* semantics a predefined set of labels is used. Each argument gets assigned a label based on certain criteria specified by the semantics. Depending on the semantics one or more labels can be assigned to an argument. Given an AF $F = \langle A, R \rangle$ a labeling is a function $L : A \to \mathbb{L}$ where $\mathbb{L}$ is a set of labels. A set of labels can, for example, be of the form $\mathbb{L} = \{in, def, out\}$. In most cases the label in denotes that an argument is in the resulting set, i.e. the arguments that are labeled with in in a labelling-based semantics correspond to the arguments of an extension in extension-based semantics. Arguments labelled with out are normally not included in the resulting set. Hence, all extension-based semantics can be represented equivalently with labelling-based semantics consisting of the labels $\{in, out\}$. An in-depth analysis of labelling-based semantics is given in [Wu et al., 2010].

Semantics

Most of the semantics presented here were introduced by Dung [1995]. In this section we give a brief overview of the semantics and define them formally. We focus on the definition of *admissible*, *complete*, *preferred* and *stable* semantics as they provide the basis for the algorithms presented in Chapter 3. For other semantics like *semi-stable*, *grounded* or *ideal* we only give a general overview.

Admissible Semantics

The notion of *admissible* semantics was introduced by Dung [1995]. A set of arguments S is an admissible extension of an AF if no argument from S attacks another argument from the set (conflict-free) and all arguments that attack the set are themselves attacked by the set (the arguments in the set are defended). Formally, admissible semantics is defined as follows:

Definition 2.5. *Let $F = \langle A, R \rangle$ be an AF. A set $S \subseteq A$ is* admissible *if it is* conflict-free *in F and each $a \in S$ is* defended *by S in F. We denote the set of admissible extensions by $adm(F)$.*

Example 2.8. The set of admissible extensions for our example graph F (see Figure 2.1) is

$$adm(F) = \{\emptyset, \{f\}, \{b, d\}, \{b, d, f\}, \{c, e, f\}\}$$

Let us analyze this example: We see that the empty set is an admissible extension of our AF. Admissible semantics does not say anything about the maximality of arguments in the set. Therefore, every AF has at least one admissible extension, namely the empty set. Furthermore, f is not attacked by any other argument but it attacks a. What follows is that, based on the definition, a can never be in an admissible extension of F or, if we follow the assignments from Table 2.1, Peter will never order fish because the oceans are overfished and no counter-argument against this argument exists.

Complete Semantics

In his seminal paper, Dung [1995] defined *complete* semantics. A set of arguments is a complete extension if it is admissible in F and it contains every argument that is defended by the set. What follows is that every complete extension of an AF is also an admissible extension of that AF.

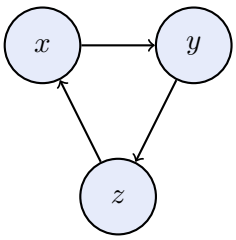
Definition 2.6. *Let $F = \langle A, R\rangle$ be an AF. A set $S \subseteq A$ is a* complete *extension of F if it is* admissible *in F and each $a \in A$ that is* defended *by S in F is* contained *in S. We denote the set of complete extensions by $comp(F)$.*

Example 2.9. The set of complete extensions for our example graph F (see Figure 2.1) is

$$comp(F) = \{\{f\}, \{b, d, f\}, \{c, e, f\}\}$$

We see that (set-inclusion based) maximality is not necessary for complete extensions. $\{f\}$ is a subset of $\{c, e, f\}$ but still a valid extension. Furthermore, the empty set is only a complete extension of an AF if there exists no argument that is not attacked. In our example this is not the case. The idea behind complete extensions is that we do not want to 'waste' defended arguments: If an argument is not attacked at all or defended it is included in the extension.

Example 2.10. As a further example consider the AF $F' = \langle\{x, y, z\}, \{(x, y), (y, z), (z, x)\}\rangle$.



Here we have that $comp(F') = \{\emptyset\}$. As $S = \emptyset$ is conflict-free and there is no argument that is defended by S (or not attacked at all) the properties for complete extensions are satisfied and $S = \emptyset$ is a complete extension for F'.

Preferred Semantics

Preferred extensions aim at the selection of a maximal number of arguments. As introduced by Dung [1995], a set of arguments is a preferred extension if the set is an admissible extension and there exists no other admissible extension that is a superset of the extension. Formally, Dung defined preferred extensions as follows:

Definition 2.7. *Let $F = \langle A, R\rangle$ be an AF. A set $S \subseteq A$ is a* preferred *extension of F if it is* admissible *in F and there exists no other admissible extension S' of F such that $S \subset S'$. We denote the set of preferred extensions by $pref(F)$.*

Example 2.11. The set of preferred extensions for our example graph F (see Figure 2.1) is

$$pref(F) = \{\{b, d, f\}, \{c, e, f\}\}$$

Here, we are interested in the 'largest' admissible sets of the AF, i.e. we do not want to 'waste' any argument. $\{b, d\}$ is an admissible extension that could represent that Peter orders meat and that eating meat once in a while is healthy. As the argument f can also be selected (considering admissible semantics) we want to include it in our result, e.g. our result, if looking back to the possible meaning of the labels, additionally contains that the oceans are overfished.

Stable Semantics

Dung [1995] defined *stable* extensions as the sets of arguments where the arguments are conflict-free and every argument that is not contained in the extension is attacked by the extension. Again, we give the formal definition for this semantics:

Definition 2.8. *Let $F = \langle A, R\rangle$ be an AF. A set $S \subseteq A$ is a* stable *extension of F if it is* conflict-free *in F and for each $a \in A \backslash S$, there exists a $b \in S$, such that $b \rightarrowtail a$. We denote the set of stable extensions by $stable(F)$.*

Example 2.12. The set of stable extensions for our example graph F (see Figure 2.1) is

$$stable(F) = \{\{c, e, f\}\}$$

The set of arguments $\{b, d, f\}$ is not a stable extension because g is not attacked by the arguments from the extension. Stable semantics make sure that every argument that is not selected has a counter-argument that is in the set of the extension. If we consider our example of natural language arguments this means that Peter wants to make sure that he 'takes all arguments into account', i.e. that at least one of his selected arguments conflicts with the argument(s) he does not choose. In $\{b, d, f\}$ he has no argument that invalidates his friend's statement of 'meat being healthy'.

Further Semantics

In the following we present further semantics that are not directly related to the contributions of this thesis. Hence, we refer the interested reader to the references of the corresponding semantics.

Semi-stable semantics were introduced by Caminada [2006]. Every stable extension is also a semi-stable extension. In contrast to stable extensions every AF has at least one semi-stable extension. All semi-stable extensions are admissible extensions. Furthermore, the set of arguments of a semi-stable extension combined with all arguments that are attacked by the extension must be maximal (wrt. set-inclusion), i.e. no other admissible set, combined with the arguments attacked by this set, is larger. The definition of *stage* extensions, as proposed in [Verheij, 1996], is similar to *semi-stable* extensions with the difference that the arguments are conflict-free (instead of admissible). Still, the property of maximality, as defined by semi-stable semantics, must hold.

Dung [1995] defined *grounded* extensions as the minimal set (wrt. set-inclusion) that includes all arguments that are not attacked at all or defended by the set. *Naive* semantics [Bondarenko et al., 1997] simply asks for the maximal (wrt. set-inclusion) conflict-free sets in an

AF. A set of argument is *ideal* if it is admissible and it is contained in every preferred set of arguments [Dung et al., 2007].

Many other semantics like *eager* [Caminada, 2007], *Resolution-based grounded* [Baroni and Giacomin, 2008] and *cf2* [Baroni and Giacomin, 2003] are proposed.

Relations between semantics

Based on the definition of the semantics, Wu et al. [2010] presented an overview of the relations between them. In Figure 2.2 we extend the original figure by further semantics that are mentioned in this thesis. An arrow represents an *is-a* relationship between the semantics. We see that all semantics are based on the conflict-freeness of arguments. In Figure 2.2 admissible, complete, stable and conflict-free semantics are highlighted as they are especially relevant in this thesis.

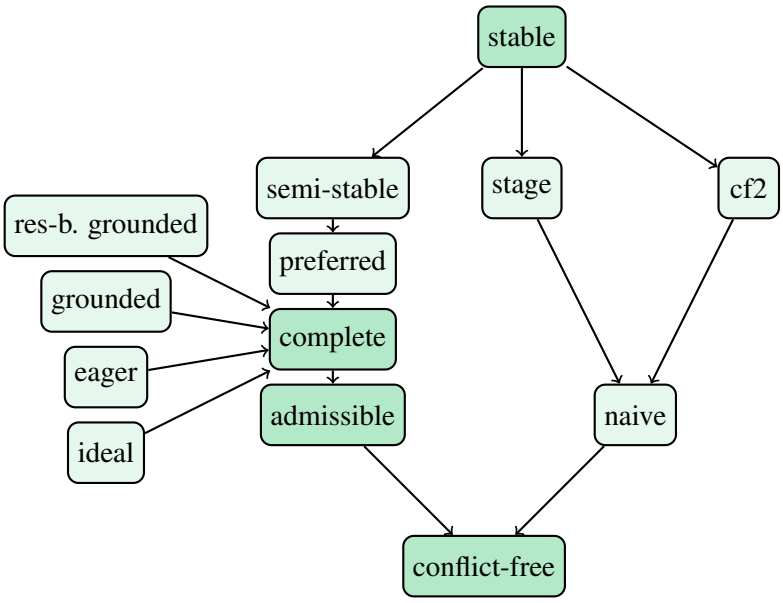


Figure 2.2: Relations between Semantics

2.4 Decision Problems on Argumentation Frameworks

Given an AF and a semantics σ we can define several decision problems that are of significant relevance in abstract argumentation. A decision problem takes certain input parameters and answers with *yes* or *no* for a given question. In this section we present important decision problems like *credulous* and *skeptical* acceptance as well as the question for the *existence* of an extension. Furthermore, we recall some of the relevant complexity classes from the field of complexity theory and give an overview of complexity-theoretic results for the decision problems.

Decision Problems

Credulous Acceptance for a Semantics σ

Sometimes we want to know if an argument a is contained in any σ-extension of a given AF. This is described by the decision problem of *credulous acceptance*, denoted by $Cred_\sigma$.

Input:	An AF $F = \langle A, R \rangle$ and an argument $a \in A$.
Question:	Is a contained in at least one σ-extension of F?

Let us consider Example 2.8 where we compute the set of admissible extensions $adm(F) = \{\emptyset, \{f\}, \{b, d\}, \{b, d, f\}, \{c, e, f\}\}$ for our example graph. The arguments a and g are not contained in any admissible extension, hence $Cred_{adm}(F, a)$ and $Cred_{adm}(F, g)$ returns *false*. b, on the other hand, is contained in two admissible extensions and therefore it is *credulously accepted* in F wrt. admissible semantics. But b is not contained in any stable extension $stable(F) = \{\{c, e, f\}\}$ of F and is therefore not credulously accepted in F (see Example 2.12) wrt. stable semantics.

Skeptical Acceptance for a Semantics σ

The decision problem of *skeptical acceptance* asks if an argument a is contained in every σ-extension of F, denoted by $Skept_\sigma$.

Input:	An AF $F = \langle A, R \rangle$ and an argument $a \in A$.
Question:	Is a contained in every σ-extension of F?

Let us again recall Example 2.8: The set of admissible extensions always contains the empty set $\emptyset$. Hence, the computation for credulous acceptance for admissible semantics always returns *false*. In Example 2.9 we computed the set of complete extensions $comp(F) = \{\{f\}, \{b, d, f\}, \{c, e, f\}\}$. Here, f is *skeptically accepted* wrt. complete semantics because it is contained in every complete extension, i.e. $Skept_{comp}(F, f)$ returns *true*. The other arguments in this example are not skeptically accepted because they are either in no or only in some complete extensions of our example AF F.

Existence for a Semantics σ

Another interesting problem is the question for the existence of an extension for a semantics σ. We denote this decision problem by $Exists_\sigma$.

Input:	An AF $F = \langle A, R \rangle$.
Question:	Does there exist a σ-extension for F?

In our examples for admissible (see Ex. 2.8), complete (see Ex. 2.9), preferred (see Ex. 2.11) and stable (see Ex. 2.12) semantics σ no set of extensions is empty. Therefore, $Exists_\sigma(F, x)$ returns *true* for our example graph F and any argument $x \in A$. In fact, the decision problem of existence for admissible, complete and preferred semantics is trivial. We already stated that every AF has at least one admissible extension, namely the empty set. Furthermore, preferred semantics asks for the maximal (wrt. set inclusion) set of admissible extensions. What follows is that every AF must have at least one maximal admissible, or preferred, extension. For complete semantics we can distinguish two cases: Either, some argument(s) are not attacked at all. Then, by Definition 2.6, these arguments are contained in the complete extension(s). On the other hand, if all arguments are attacked the empty set is always a complete extension of the AF.

Other Decision Problems

In literature, other decisions problems like the existence of a non-empty extension or the verification of an extension for a semantics σ have been considered. As we do not address these problems in the following chapters directly we only give a short overview of the ideas behind them here.

As shown before the answer to $Exists_\sigma$ is trivial for semantics whose extensions always contain the empty set. We can address this by asking for the *existence of a non-empty extension* for a semantics σ: This is denoted by $Exists_\sigma^{\neg\emptyset}$. This problem can not directly be answered by the definition of the respective semantics; for all semantics presented here it is necessary to compute the extensions. Hence, the question of $Exists_\sigma^{\neg\emptyset}$ is not trivial anymore.

Another interesting problem is the *verification* of a given set of arguments with regard to a given AF and a semantics σ. We want to know if the set of arguments S is a σ-extension of F. This is denoted by Ver_σ.

Complexity Theory

In this section we give an overview of the complexity classes P, NP, co-NP and Π_2^p. Furthermore we recall the definition of NP-complete problems. For further details we refer to [Papadimitriou, 2003] where a brief overview on complexity theory is given. For an in-depth insight into the field of complexity theory we refer the interested reader to the book *Complexity Theory* [Papadimitriou, 1994].

One of the most important complexity classes is P, that is, the collection of all problems that can be solved in polynomial time in the size of the input instance. P may be defined as follows:

Definition 2.9. *The* complexity class P *consists of all problems P that satisfy the following conditions:*

1. *There exists a program Π that decides the problem P.*
2. *For all instances I of P the runtime of Π on I is polynomial in $|I|$.*

In other words, a decision problem P is in the complexity class P if there exists a program Π that solves the problem in $O(|I|^k)$ for all instances I, where k is a constant.

Another important *complexity class* is *NP*: Positive instances I of a problem P have solutions (or *certificates*) whose size is at most polynomial in the size of the instance, i.e. they are *polynomially balanced*. Given a possible certificate C for an I and P it is possible to check if C is a positive instance of P in polynomial time, i.e. it is *polynomially decidable*.

Definition 2.10. *The* complexity class NP *consists of all problems P that satisfy the following conditions:*

1. *There exists a* polynomially balanced *certificate relation for P.*
2. *There exists a* polynomially decidable *certificate relation for P.*

Remark 2.11. co-NP *is the class to problems P such that the complement $\overline{P}$ of P is in NP.*

Furthermore, we recall the definition of *NP-complete* decision problems. Besides satisfying the conditions for problems that are in NP (polynomially balanced and decidable), for NP-complete problems there exists a polynomial time algorithm that can transform (reduce) other problems from NP to the problem P.

Definition 2.12. *A decision problem P is* NP-complete *if it satisfies the following conditions:*

1. *P is in NP.*
2. *every problem that is in NP is reducible to P in polynomial time.*

Besides the complexity classes P, NP and co-NP we introduce the class Π_2^p. It resides on the second level of the polynomial hierarchy. The *polynomial hierarchy* can be defined inductively where $\Pi_0^P = P$ and $\Pi_{i+1}^P = \textit{co-NP}^{\Pi_i^P}$ for $i \geq 0$. It is a group of classes where some part of the problem is defined to be computed by an *oracle*. The intuition behind oracles is that we can neglect the cost of the computation for a subroutine carried out by the oracle, i.e. we assume that the cost is 1 (the call to the oracle). We can therefore study the complexity of a problem where some part of the computation 'comes for free'. This is useful because we can then identify independent sources of complexity that are computed by the oracle and can ask for the complexity of the remaining parts of the problem. In the following we define the Π_i^P classes that are relevant in this thesis:

Definition 2.13. *The complexity classes Π_i^P for $i \in \{0, 1, 2\}$ are defined by the* polynomial hierarchy *where*

- $\Pi_0^P = P$

- $\Pi_1^P = co\text{-}NP$ *and*
- $\Pi_2^P = co\text{-}NP^{NP}$.

For $co\text{-}NP^{NP}$ the exponent NP of represents an oracle that answers a decision problem that is in NP in constant time. The problem that asks the oracle is in co-NP. In other words, the class Π_2^p contains problems that are in co-NP and whose co-NP routine asks an oracle that is in NP arbitrarily many times. The oracle answers the question in constant time.

Decision problems that are in the complexity class P are considered to be *tractable*, i.e. there exist efficient algorithms for the computation of solutions. Note that tractable does not necessarily mean that there exists a practicable algorithm to compute the solution. If the exponent k is very large the algorithm may be slow in practice. Problems that are NP-complete, co-NP-complete or Π_2^p-complete are considered to be computationally hard, i.e. they are *intractable*.

Complexity Results for Decision Problems in AA

We already stated the the computation of $Skept_{adm}$ as well as $Exists_{adm}$, $Exists_{comp}$ and $Exists_{pref}$ is trivial. In the following we present further complexity-theoretic results for the decision problems and semantics that are defined in this thesis.

The complexity of the credulous and skeptical acceptance problems has been studied for example in [Doutre and Mengin, 2004; Dimopoulos and Torres, 1996; Coste-Marquis et al., 2005; Dunne and Bench-Capon, 2002]. A general overview is given by Dunne and Wooldridge [2009] where the results for preferred and stable semantics are summarized.

For the semantics presented in this thesis is is shown that the credulous acceptance problem is NP-complete. This can, for example, be done by a reduction from 3-SAT to $Cred_\sigma$. In Dimopoulos and Magirou [1994] credulous acceptance for stable as well as preferred semantics is analyzed. Although Dimopoulos et al. define their complexity proofs on graphs and use other notions (They use the terminology of 'semi-kernel', 'maximal semi-kernel' and 'kernel' which corresponds to 'admissible set', 'preferred extension' and 'stable extension' in this thesis) their results are directly applicable for our AFs and our defined semantics and decision problems. Dunne and Bench-Capon [2002] showed that $Skept_{pref}$ is Π_2^P-complete. Finally, the skeptical acceptance problem for stable semantics is shown to be co-NP-complete while $Exists_{skept}$ is NP-complete.

Table 2.2 summarizes the complexity results obtained from [Dimopoulos and Torres, 1996; Coste-Marquis et al., 2005; Dunne and Bench-Capon, 2002].

σ	$Cred_\sigma$	$Skept_\sigma$	$Exists_\sigma$
$adm(F)$	$NP\text{-}c$	trivial	trivial
$comp(F)$	$NP\text{-}c$	$P\text{-}c$	trivial
$pref(F)$	$NP\text{-}c$	$\Pi_2^P\text{-}c$	trivial
$stable(F)$	$NP\text{-}c$	$co\text{-}NP\text{-}c$	$NP\text{-}c$

Table 2.2: Overview: Complexity Results

Algorithms for Abstract Argumentation Frameworks

In literature, many different approaches for the computation of extensions and the evaluation of decision problems have been proposed. In here, we give an overview of *direct* as well as *reduction-based* algorithms. Both approaches result in algorithms that reflect the complexity-theoretic results from Table 2.2. In Section 2.5 we present a *dynamic programming* approach based on tree decompositions that aims at the identification of certain fragments of AFs. It is then possible to bind the complexity of the problem to a fixed parameter and therefore to reduce the overall run-time. The dynamic programming approach provides the basis for our algorithms in Chapter 3.

Direct Algorithms

The direct algorithm approaches we present here all rely on labelling-based notions (see Section 2.3). They are defined *directly* on the underlying argumentation frameworks.

The approach presented by Modgil and Caminada [2009] can be described as an *argument game*. The approach not only aims at the computation of a correct solution but also shows (and hence proves) that the gained solution is indeed a correct one (i.e. the decision for or against arguments in an extension is well defined and it can be shown that the selection of these arguments is indeed a valid extension for a given semantics). In the argument game two opponents play against each other. The proponent (PRO) selects an initial argument x and trys to defend it. The opponent (OPP) selects an argument y that attacks x. What follows is that again PRO selects an argument z that attacks y, thus defending his original argument x. The game is played as long as arguments can be selected by PRO and OPP. If a player can not respond to a move from his opponent (i.e. the last move in the game) then the player of the last move wins. If such a game is won by PRO over an argument x this is called the *line of defense* for x. Because every player answers on the last choice of his opponent, this argument game approach leads to a depth-first consideration of the attacks. Figure 2.3 shows an example *dispute tree* where PRO selects the argument a from the AF of the left side as the initial argument. The opponent can then either choose b or c as both attack a.

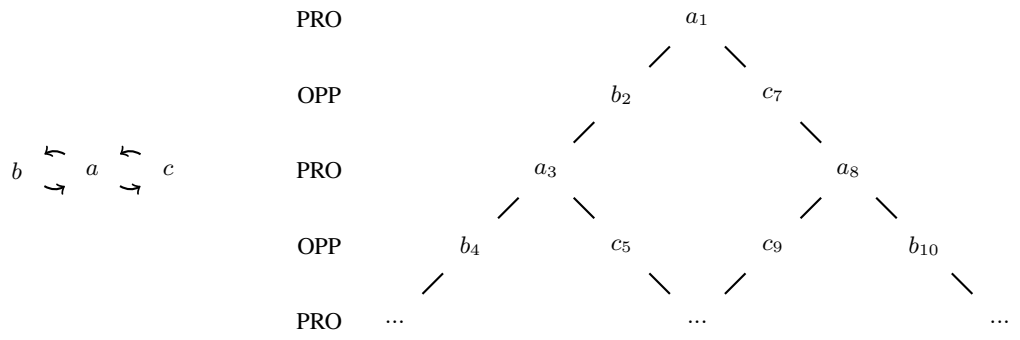


Figure 2.3: AF and Dispute Tree [Modgil and Caminada, 2009]

In contrast, Verheij [2007] proposes a breadth-first approach: It relies on the computation

of partial proofs and partial refutations of an underlying AF where partial refers to a sub-graph of the original AF. An argument is *partially proved* if all attackers are again attacked by at least one argument. *Partial refutation* describes that an argument that is not selected is attacked by at least one selected argument. Depending on the depth of the partial proof selected arguments and attackers take turns. For a complete proof the original AF is taken into account.

Reduction-Based Algorithms

Reduction-based algorithms define some kind of mapping between AFs (or properties thereof) and other languages, i.e. the decision problem can be *reduced* (or translated) to another (logic) language.

Caminada and Gabbay [2009] propose an interesting approach that combines direct computation with an reduction-based approach. First, they define labelings for nodes and show their one-to-one relationship to extensions defined by a semantics, i.e. every extension corresponds to one labeling and vice-versa. They then use a meta level language that describes the labelings. The meta level language can be classical logic or modal logic. By proving that the labeling corresponds to expressions of the meta level language they show that the semantics can be expressed in this logic. It is therefore possible to

1. define an equivalent labeling for a semantics,
2. translate the labeling into another (logic) language and
3. solve the problem within this language.

Depending on the language existing tools may be used (such as SAT-solvers).

In [Egly and Woltran, 2006] the problems are translated to *Quantified Boolean Formalæ* (QBF). A QBF is based on standard propositional formulæ but additionally allows to define quantifiers on the propositional variables. The language contains the unary operators $\forall x$ (universal quantifier) and $\exists x$ (existential quantifier) where x is an atom that is bound by the quantifier. The advantage is that there exist efficient QBF solvers that can be seen as a *black box* during the computation of a decision problem.

Amgoud and Devred [2011] propose a reduction to *Constraint Satisfaction Problems* (CSP). A CSP is defined by variables, a domain and a set of constraints. The constraints consist of variables that define which values can be assigned to the variables. We are interested in a variable assignment where all constraints are satisfied. Depending on the decision problem it is either possible to compute all combinations of variable assignments (i.e. we enlist all solutions) or we are only interested in the question of the existence of a solution for a CSP (recall, for example, the decision problem $Exists_\sigma$). Such problems are, in general, NP-complete but there is ongoing work in the optimization of solvers for CSPs.

Other work focuses on the reduction of argumentation problems to *anser-set programming* (ASP). One approach is proposed by Egly et al. [2010]. Answer-set programs follow a declarative approach where the program consists of constants and rules. The rules can be used to derive solutions (or answers) from a knowledge base for a given question.

Besnard and Doutre [2004] analyze the decision problem of Ver_σ, i.e. if a given set of arguments is a σ-extension of an AF. They propose three different approaches. The first one

analyzes if a set of arguments satisfies given *equations* where the equations represent attack and defense relations between arguments. Another approach is based on *model checking*. In here, all extensions of a semantics are characterized by a model that is defined in propositional logic. 'S is an extension if and only if S corresponds to a model of the formula' [Besnard and Doutre, 2004]. The third approach is similar to [Caminada and Gabbay, 2009] where it is checked if a set of arguments *satisfies* a given propositional formula. It is only a valid extension if the formula is satisfied. To sum it up Besnard and Doutre [2004] elaborate equalities of formulæ and sets of arguments.

2.5 Dynamic Programming and Tree Decompositions

As presented in Table 2.2 many decision problems like acceptance and the existence of extensions for a given AF are computationally hard. In this section we present a dynamic programming approach that is based on *fixed parameter tractability* (FPT). We introduce *tree decompositions* for graphs that allow us to bind the complexity of the algorithms to a constant, i.e. the *tree-width*, which represents the *tree-likeliness* of an AF. By binding some problem parameter to a fixed constant many of the intractable decision problems become tractable.

Furthermore, we introduce different variants of tree decompositions, namely *normalized* and *semi-normalized* tree decompositions. Based on the tree decomposition, the tree contains more or less nodes. This can affect the run-time of algorithms that are defined on the decomposition. We will make use of the different decomposed trees in our algorithms that we present in Chapter 3.

Tree Decompositions

The intractability results from Table 2.2 lead to the question if we can reduce the complexity of the decision problems. One idea is based on the fact that some hard problem on graphs can become tractable if we restrict ourselves to trees. The notion of *tree decompositions* is one possible approach that aims at the translation of graphs to trees. It was proposed by Robertson and Seymour [1984] and since then it has been well-studied by many authors (see e.g. [Bodlaender, 1993, 1997; Kloks, 1994]).

Definition 2.14. *A* tree decomposition *of an undirected graph $G = (V, E)$ is a pair $(\mathcal{T}, \mathcal{X})$ where $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$. $V_\mathcal{T}$ are the vertices in the tree and $E_\mathcal{T}$ are the edges of the tree. $\mathcal{X} : V_\mathcal{T} \to 2^V$ is a so-called* labelling function *that assigns to every vertex $V_\mathcal{T}$ of the tree a set of vertices V from the original graph. The sets of vertices $\mathcal{X} = (X_t)_{t \in V_\mathcal{T}}$ have to satisfy the following conditions:*

(i) $\bigcup_{t \in V_\mathcal{T}} X_t = V$

(ii) $(v_i, v_j) \in E \Rightarrow \exists t \in V_t : \{v_i, v_j\} \subseteq X_t$

(iii) $v \in X_{t_1} \wedge v \in X_{t_2} \wedge t_3 \in path(t_1, t_2) \Rightarrow v \in X_{t_3}$

Remark 2.15. *X_t is also called the* bag *for the vertex $t \in V_\mathcal{T}$.*

Property (i) ensures that every vertex $v \in V$ of the original graph is contained in at least one bag X_t of the tree decomposition. This ensures that no vertex is 'lost' when we decompose the graph G. Furthermore, if the vertices v_i, v_j from the original graph are connected via an edge, they have to appear together in at least one bag X_t of $\mathcal{T}$. This is defined by condition (ii). An algorithm that traverses the tree decomposition can therefore analyze the relations between vertices of the bags (i.e. it can check if there exist edges between vertices) and it is guaranteed that no edge of the original graph is 'lost'. The third property (iii) finally ensures that no vertex can 'reappear' on a path from the root to the leaf nodes, i.e. bags X_t that contain a vertex v are connected: If a vertex v is removed somewhere on path we know that this vertex is completely processed by the algorithm.

In Figure 2.4 one possible tree decomposition of our example graph from Figure 2.1 is given.

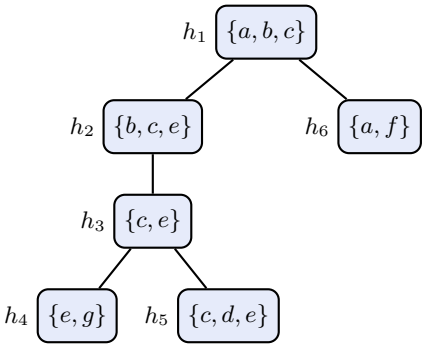


Figure 2.4: Possible Tree-Decomposition for the Graph in Figure 2.1

The example tree is rooted in h_1 and its bag X_{h_1} contains the vertices $\{a, b, c\}$ from the original graph. The vertex e is, for example, contained in the bags X_{h_2} and X_{h_4}. Hence, due to property (iii) of Definition 2.14, it must also be contained in X_{h_3}. Note that tree decompositions in general are not binary tree, i.e. nodes can have arbitrarily many children.

Definition 2.16. *The* width *of a tree decomposition* $(\mathcal{T}, \mathcal{X})$ *is defined as*

$$max(|X_{t \in V_t}|) - 1$$

Definition 2.17. *The* tree-width *of a graph* G *is the minimum width of all possible tree decompositions of* G*.*

Hence, the *width* of our example tree is 2, i.e. the size of the largest bags X_{h_1} and X_{h_5} is 3, minus 1. An interesting approach that describes the intuition behind tree-width is the *cops-and-robber game* [Seymour and Thomas, 1993]: The game is played on an finite undirected graph. The robber stands on a vertex and can move along the edges of the graph to any other vertex as long as the path between the vertices is not occupied by a cop. There are k cops in the game that try to catch the robber. Cops can move from vertex to vertex arbitrarily, i.e. their moves are not bound to the edges of the graph. The robber sees the cops approaching and can move to another vertex before the cops arrive. The cops try to corner the robber, i.e. they block all adjacent vertices of the robber's current position and an additional cop catches the robber. [Seymour and Thomas, 1993] shows that the minimal number of cops needed to catch the robber minus one corresponds to the tree-width of the graph.

The computation of an optimal tree decomposition (wrt. width) is known to be an NP-complete problem [Arnborg et al., 1987]. Hence, there exist several algorithms that provide 'good' tree decompositions in polynomial time. A general approach for the computation of tree decompositions is as follows: First, an ordering (called the *elimination ordering*) of the vertices from the original graph is defined. Afterwards, the vertices are processed via *bucket elimination*[1] [Dechter, 2003]: (1) For each vertex v_i from the ordering create a bucket B_{v_i}. (2) For

[1] *Buckets* correspond to the *bags* we defined for tree decompositions.

every edge (v_i, v_j) in the graph, add the vertex with lower elimination ordering to the bucket of the other vertex from the edge. (3) Traverse the buckets B_{v_i} as given by the elimination ordering and copy all vertices $v \in B_{v_i} \backslash \{v_i\}$ to the bucket B_{v_j} where v_j is the vertex with the highest ordering. (4) Finally, connect the buckets B_{v_i} and B_{v_j}.

Furthermore there exist several heuristics that are based on bucket elimination. They especially try to improve the initial elimination ordering (see e.g. [Bodlaender and Koster, 2010; Dermaku et al., 2008]).

Normalization of Tree Decompositions

As we want to develop algorithms that are based on tree decompositions we introduce *normalized* as well as *semi-normalized* tree decompositions. The conditions for tree decompositions as given in Definition 2.14 may result in trees that are not comfortable when it comes to the definition of algorithms on them. Normalized and semi-normalized tree decompositions introduce additional nodes in the tree that simplify the task of developing dynamic-programming algorithms on them. The nodes can be introduced by a single traversal of the tree and hence the additional effort is negligible, i.e. a tree decomposition with k width and n nodes of a graph G can be transformed to a normalized or semi-normalized tree decomposition of $O(n)$ nodes in $O(n)$ time [Niedermeier, 2006].

Normalized Tree Decompositions

Normalized tree decompositions comply with Definition 2.14 but they consist of four different node types. Normalized tree decompositions are defined as follows [Kloks, 1994]:

Definition 2.18. *A tree decomposition $(\mathcal{T}, \mathcal{X})$ of a graph G is called* normalized *(or* nice*) if $\mathcal{T}$ is a rooted tree and the following conditions are satisfied:*

1. *Every node t of $\mathcal{T}$ has at most two children.*
2. *If a node t has two children t_1 and t_2, then $X_t = X_{t_1} = X_{t_2}$. Then, t is called a* JOIN *or* BRANCH *node.*
3. *If a node t has one child t_1 one of the following conditions must hold:*
   a) $|X_t| = |X_{t_1}| + 1$ *and* $X_{t_1} \subset X_t$. *Here, t is called an* INTRODUCTION *node.*
   b) $|X_t| = |X_{t_1}| - 1$ *and* $X_t \subset X_{t_1}$. *Here, t is called a* FORGET *or* REMOVAL *node.*
4. *If t has no child nodes, it is called a* LEAF *node.*

Remark 2.19. *Note that we we call such tree decompositions* normalized*. In literature, they are sometimes called* nice *tree decompositions. Furthermore, in the following we will call nodes that satisfy condition 2 branch nodes and nodes that satisfy condition 3b removal nodes.*

Our example tree from Figure 2.4 can easily be transformed to a normalized tree decomposition as depicted in Figure 2.5. The tree is rooted in the branch node n_1. The bags of nodes n_2 and n_{11} contain the same nodes (from the original graph) as the branch node, i.e. $X_{n_1} = X_{n_2} = X_{n_{11}} = \{a, b, c\}$. Furthermore, n_2 is an example for an introduction node where $X_{n_3} \subset X_{n_3}$. One argument (a) is introduced. n_3, on the other hand, is a removal node.

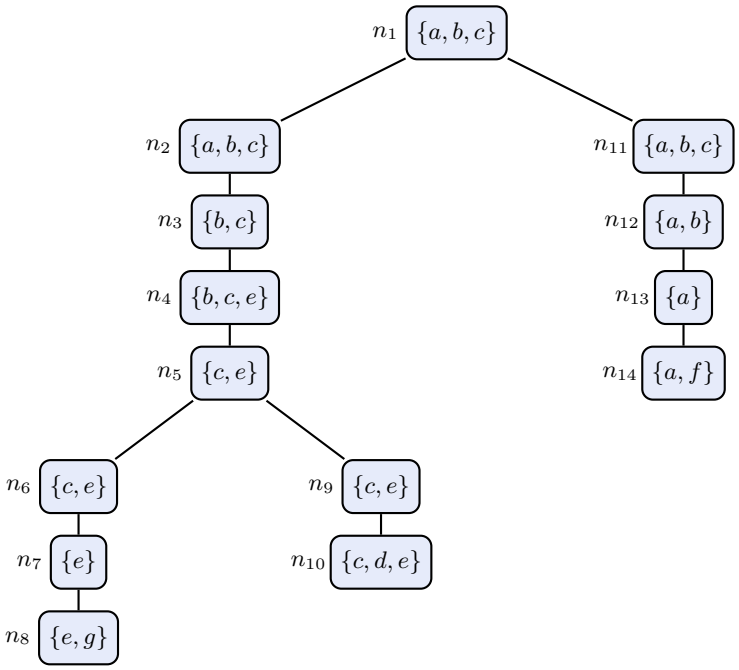


Figure 2.5: Normalized Tree-Decomposition

Semi-Normalized Tree Decompositions

For our purposes in this thesis we introduce a further kind of normalization. Semi-normalized tree decompositions are something in between tree decompositions and normalized tree decompositions. They consist of two different types of nodes. We define semi-normalized decompositions as follows:

Definition 2.20. *A tree decomposition $(\mathcal{T}, \mathcal{X})$ of a graph G is called* semi-normalized *if $\mathcal{T}$ is a rooted tree and the following conditions are satisfied:*

1. *Every node t of $\mathcal{T}$ has at most two children.*
2. *If a node t has two children t_1 and t_2, then $X_t = X_{t_1} = X_{t_2}$. Then, t is called a* JOIN *or* BRANCH *node.*
3. *Otherwise, one of the following conditions must hold for a node t:*
   a) *t has no child nodes.*
   b) *t has exactly one child t_1 and $X_t = (X_{t_1} \setminus S') \cup S''$ where $S' \subseteq X_{t_1}$ and $S' \cap S'' = \{\}$ hold.*

   Then, t is called an EXCHANGE *node.*

Remark 2.21. *In literature the term semi-normalized tree decomposition is ambiguously defined. Dorn and Telle [2009], for example, define three different node types, namely introduce,*

forget and join nodes. Introduce and forget nodes correspond to the nodes of Definition 2.18 but a join node t has two children t_1 and t_2 where only $X_t = X_{t_1} \cup X_{t_2}$ must hold.

The definition of branch nodes for semi-normalized tree decompositions corresponds to that of normalized tree decompositions. Leaf, introduction and removal nodes, however, are combined in exchange nodes. Exchange nodes allow us to introduce or remove arbitrarily many vertices v from the original graph G. This is captured by condition 3b of Definition 2.20: S' is the set of vertices that are removed from the bag of the child X_{t_1} and S'' contains all vertices that are introduced. In Figure 2.6 we depict a possible semi-normalized tree that could result from the original tree decomposition in Figure 2.4.

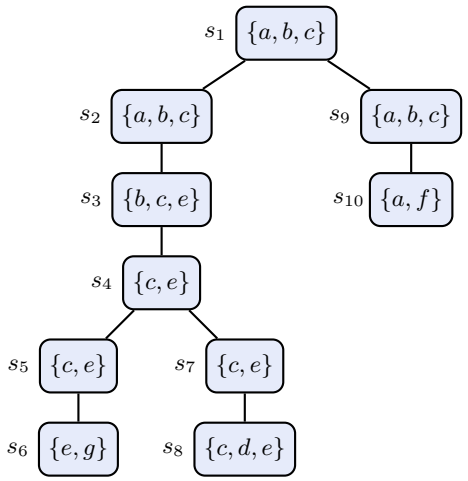


Figure 2.6: Semi-normalized Tree-Decomposition

The node s_9 of the semi-normalized tree decomposition represented in Figure 2.6 is a good example for an exchange node: The set S' of removed vertices contains f whereas the set of introduced vertices $S'' = \{b, c\}$.

Semi-normalized tree decompositions have several advantages compared to general tree decompositions and normalized tree decompositions: They simplify the definition of algorithms because we only have to deal with at most two child nodes at a time. Furthermore, in branch nodes, the bags of the children contain the same vertices. This restriction allows us to apply certain optimizations during the evaluation of branch nodes (as we will see in Chapter 4). Permutation nodes, on the other hand, reduce the overall size of the tree (compared to normalized tree decompositions) which leads to a better run-time of algorithms (see Chapter 5).

Fixed Parameter Tractability

In classical complexity theory problems are oftentimes analyzed solely based on the size of the input instance. Although many problems are intractable in general it is sometimes possible to identify instances for which the problem is tractable. We want to formally define these tractable instances. One idea is based on the definition of an additional parameter that serves as a bound

for the problem. We already identified *tree-width* as an important parameter of tree decompositions. In this section we introduce *fixed parameter tractability* (FPT) and outline how intractable problems can become tractable when bound to a constant.

A brief overview for FPT is, for example, given in Bodlaender [1997]. For further details we recommend the book *Invitation to Fixed-Parameter Algorithms* written by Niedermeier [2006]. First, we give the theoretical background to FPT that is based on the ideas by Downey and Fellows [1995]:

Definition 2.22. *A* parameterized problem *is a language $L \subseteq \Sigma^* \times \Sigma^*$. The first component Σ of $\Sigma^* \times \Sigma^*$ is a finite alphabet. The second component is called the* parameter *of the problem.*

In almost all cases the parameter is a nonnegative integer or a set thereof. In the case of tree decompositions we will use the *tree-width* as parameter. The complexity class FPT is then defined as follows:

Definition 2.23. *The complexity class FPT consists of problems that can be computed in $f(k) \cdot n^{O(1)}$ time where f is a function that depends on the fixed parameter k and n is the input size. The problem L is then called* fixed-parameter tractable.

The run-time of fixed-parameter tractable problems heavily depends on k. It may be the case that FPT problems are not solvable efficiently enough in practice. Hence, parameterized problems are generally useful if the constant k is a low value. Recall our definition of *width* (see Definition 2.16) and *tree-width* (see Definition 2.17). If we find a width of a tree decomposition that is equal to the tree-width of the original graph the fixed-parameter algorithm performs relatively good. For graphs with high tree-width, however, the tree decomposition based approach is oftentimes of little use.

Courcelle's Theorem

The fact that many NP-hard problems are tractable for graphs of bounded tree-with was shown by Courcelle. Essentially, Courcelle's Theorem states that every problem defined in Monadic Second Order (MSO) logic can be solved in linear time on graphs of a bounded tree-width [Courcelle, 1990].

Monadic Second Order logic has high expressiveness. It is an extension of propositional and first order logic. Propositional logic consists of variables and logic operators such as $\wedge$, $\vee$ and $\neg$. First order logic additionally introduces predicates and quantifiers. Predicates can, informally, be interpreted as functions that return true or false based on their variables. Quantifiers 'bind' the variables where for the universal quantifier $\forall x P$ all x have to satisfy the formula P and for $\exists x P$ at least one x has to satisfy P. x is a variable that can have values from the domain. In Monadic Second Order we additionally have 'set variables' that can range over sets of elements from the domain. It is therefore not only possible to quantify over single objects but also over sets of objects.

Courcelle's famous theorem then reads as follows [Courcelle, 1990]:

Theorem 2.24. *Let Φ be an MSO formula and $k \geq 1$. Given a graph G and a tree decomposition of width at most k, there is a linear-time algorithm that decides wether G satisfies the MSO formula.*

This result does not necessarily mean that we can find efficient algorithms for our decision problems. MSO logic has very large expressiveness and it is possible to define rather short MSO formulæ to express some NP-hard problems. But the complexity may be hidden in the big-O notation and algorithms may be slow in practice.

Algorithms based on Fixed Parameter Tractability

Before we introduce our own algorithms we give an overview of other fixed-parameter tractability based algorithms that can be found in literature.

Ordyniak and Szeider [2011] analyze *acyclic* as well as *noeven* argumentation frameworks. Acyclic argumentation frameworks do not contain any directed cycles whereas noeven frameworks do not contain directed cycles of even length. They analyze skeptical as well as credulous acceptance and show that these fragments can be solved in polynomial time for AFs that are bound to the distance to the respective fragments. Furthermore they present negative results for *bipartite* and *symmetric* argumentation frameworks, i.e. they show that even with distance 1 from these fragments the problems $Skept_\sigma$ and $Cred_\sigma$ do not become tractable. Symmetric frameworks consist only of symmetric attacks, i.e. every argument a that attacks an argument b is itself attacked by b. Bipartite AFs are frameworks that can be partitioned into two independent conflict-free sets.

Dvořák et al. [2010b] present algorithms for argumentation frameworks of bounded *clique-width*. Clique-with is a measurement of the complexity of a graph. It is defined via a construction process of the graph where only a limited number of vertex labels is available. If some vertices share the same labels somewhere during construction they can be treated uniformly in the following steps of the construction process. They analyze the problem of acceptance with respect to admissible and preferred semantics.

The tree-decomposition based approaches presented in [Dvořák et al., 2010a] and [Dvořák et al., 2011] are directly related to the algorithms presented here. Dvořák et al. [2010a] define algorithms for the computation of admissible as well as preferred extensions of argumentation frameworks. Furthermore they show a possible way to answer the decision problems of credulous as well as skeptical acceptance for these semantics. Their approach is based on normalized tree decompositions. In the following chapter we will introduce their main ideas and will extend them to stable and complete semantics. Furthermore we will elaborate an algorithm for admissible semantics on semi-normalized tree decompositions. In [Dvořák et al., 2011] a software framework is presented that allows us to directly work on tree decompositions, i.e. the framework handles the tree decomposition step and we can focus on the implementation of the algorithms. A software, *dynPARTIX* is presented that efficiently computes admissible and preferred extensions for an AF. As a result of this thesis we extend this framework by further semantics and improve the overall performance (in particular by using semi-normalized tree decompositions).

CHAPTER 3

Tree-Decomposition based Algorithms

In this chapter we present three novel algorithms that are based on tree decompositions. The algorithms compute admissible, stable and complete extensions for argumentation frameworks. Then we can either enumerate all extensions for a semantics or count the overall number of extensions. Furthermore, the definition of the algorithms supports the evaluation of decision problems, namely credulous and skeptical acceptance.

In Section 3.1 we introduce general definitions that are shared by all algorithms and give an introduction to the general concepts behind the algorithms. We furthermore outline the execution steps of our algorithms. The algorithms traverse the tree decompositions in bottom-up order. In every step we analyze the arguments in the bag of the current node based on the arguments in the bags below in the tree decomposition. The extensions for the input instance can be obtained by the final computation step in the root node. In order to be able to represent the intermediate results in every node we introduce the concept of colorings as well as labelings (for complete semantics). Then we can encode the relations between arguments. We present the definitions and the general idea of our algorithms on basis of admissible semantics for normalized tree decompositions as presented by Dvořák et al. [2010a].

We continue with the definition of algorithms for stable (see Section 3.2) and complete semantics (see Section 3.3). These algorithms are defined on normalized hypertrees.

In Section 3.4 we propose a novel algorithm on semi-normalized tree decompositions for admissible semantics. This allows us to compare it with an implementation on normalized tree decompositions as developed by Dvořák et al. [2010a].

Furthermore in Section 3.5 we outline how the decision problems of credulous and skeptical acceptance can be answered on basis of our defined algorithms.

3.1 Overview

Basic Definitions

First, we introduce some basic definitions and notions that are needed for all algorithms developed in the context of this thesis. Until now we defined argumentation frameworks (see Definition 2.1) and defined tree decompositions on graphs (see Definition 2.14). As tree decompositions are defined on graphs it remains to formally define the relation between argumentation frameworks and graphs:

Definition 3.1. *Let $F = \langle A, R\rangle$ be an AF. A* tree decomposition of an AF *F is a tree decomposition of the undirected graph $G = (A, R')$ where$A$ are the arguments of the AF and R' are the edges of R without orientation.*

Remark 3.2. *Analogous to the definition of* tree-width *for graphs we can define the tree-width for an AF F as the is the minimum width of all possible tree decompositions for F.*

In order to work on the tree decompositions we have to introduce several notions for parts of the decomposition that are similar to the work of Dvořák et al. [2010a].

Definition 3.3. *Let $(\mathcal{T}, \mathcal{X})$ be a tree decomposition of an AF F and let $t \in \mathcal{T}$. For a subtree of $\mathcal{T}$ that is rooted in t we define $X_{\geq t}$ as the union of all bags within this subtree, e.g. $X_{\geq t}$ contains all arguments of this subtree.*

Furthermore, $X_{>t}$ denotes $X_{\geq t} \backslash X_t$, i.e. all arguments from the bags in the subtree without the arguments from the bag of t.

In Figure 3.1 we present a semi-normalized tree decomposition of our original AF from Figure 2.1. $X_{\geq s_3}$, for example, contains the arguments $\{b, c, d, e, g\}$. $X_{>s_3}$ contains the arguments $\{d, g\}$. Furthermore, we define sub-frameworks within the decomposition as follows:

Definition 3.4. *For a tree decomposition $(\mathcal{T}, \mathcal{X})$ of an AF $F = \langle A, R\rangle$ let $t \in \mathcal{T}$ be a node of the tree. Then, the* sub-framework in *t, denoted by $F|_{X_t}$ or F_t, consists of all arguments $x \in X_t$ and the attack relations(x_1, x_2) where $x_1 \in X_t$, $x_2 \in X_t$ and $(x_1, x_2) \in R$.*

Furthermore, the sub-framework induced by the subtree rooted in *t, denoted by $F|_{X_{\geq t}}$ or $F_{\geq t}$, consists of all arguments $x \in X_{\geq t}$ and the attack relations (x_1, x_2) where $x_1 \in X_{\geq t}$, $x_2 \in X_{\geq t}$ and $(x_1, x_2) \in R$.*

Let us again consider the example tree depicted in Figure 3.1. For each node t, the arguments that are contained in bag X_t are marked with solid cycles. F_t, the sub-framework in t, consists of the arguments in solid cycles and all solid attack arrows. In combination with the dashed parts we obtain the induced sub-frameworks $F_{\geq t}$.

The dashed parts in a node can be considered as processed parts of the original AF, i.e. the corresponding arguments where already removed from the tree decomposition and they will not reappear in parent nodes of $\mathcal{T}$ (remember the connected property of tree decompositions, see (iii) in Definition 2.14). The solid parts (arguments) have to be analyzed when the algorithm traverses the respective node.

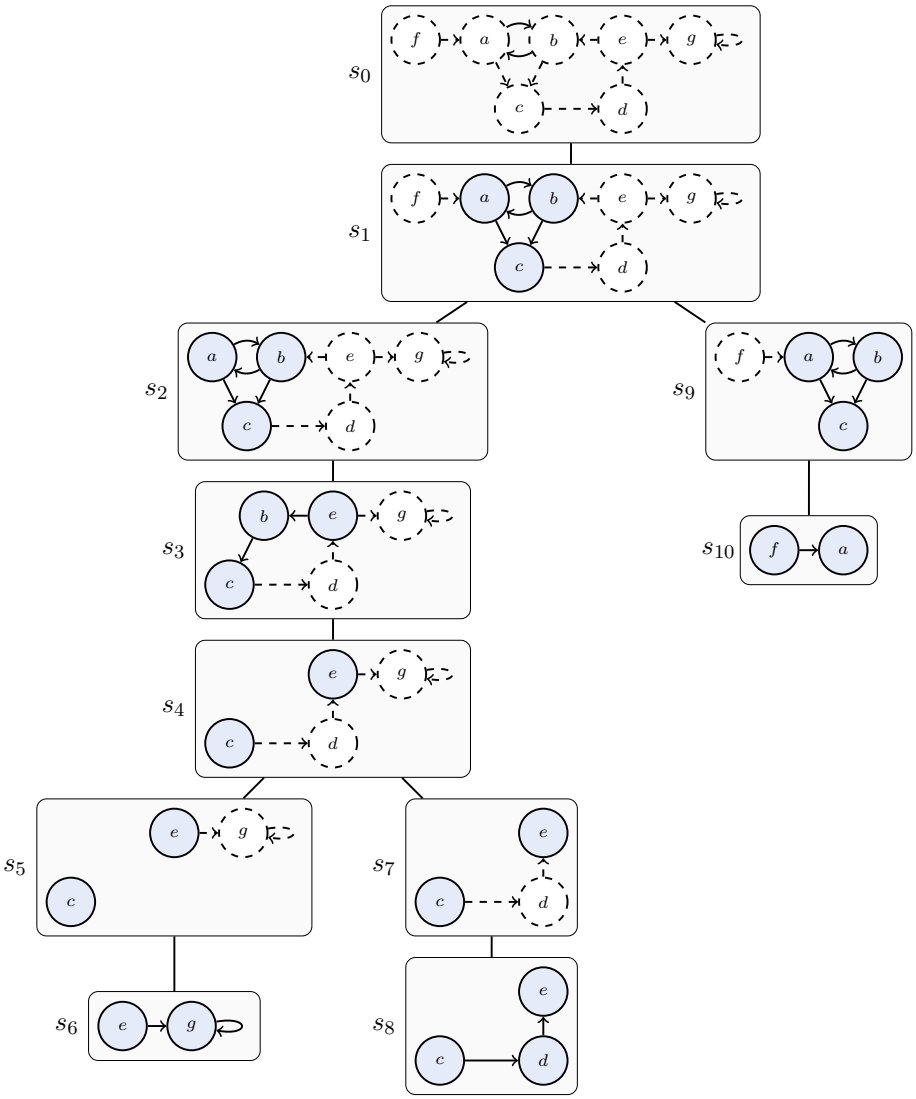


Figure 3.1: Semi-normalized Tree Decomposition with Sub-Frameworks

Note that we introduce an additional node s_0 as the root node of the tree decomposition. This node has an empty bag $X_{s_0} = \{\}$ of arguments. Hence, in the final computation step for a semantics all arguments are removed. This allows a more comprehensive definition of algorithms because the final removal step yields towards equivalence of the computation for $F_{\geq s_0}$ and extensions of the original argumentation framework.

Working on Tree-Decompositions

In here we online the general steps that are taken by our algorithms. As all our algorithms follow an approach similar to [Dvořák et al., 2010a] we recall their definitions for admissible semantics and explain the ideas behind them in detail. A nice feature of tree decompositions

is that it is possible to only work on local information that is available in the respective nodes. The *dynamic programming* approach allows us to discard information as early as possible (i.e. we know that arguments that are removed from a bag will never reappear in a bag above in the tree). We can then obtain extensions or answers to decision problems by completely traversing the tree decomposition in bottom-up order and due to the definitions of the algorithms and their correspondence to extensions we obtain the solution in the root node.

Algorithm Definitions

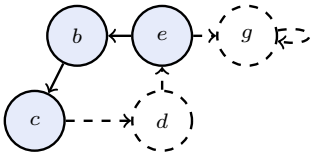
In this section we outline the definitions of our algorithms. As an example, we give the definitions for admissible semantics that can be found in Dvořák et al. [2010a]. Our definitions follow a uniform approach:

Restricted Sets: In every node t of the tree decomposition we can analyze the (sub)-framework $F_{\geq t}$. $X_{>t}$ denotes all arguments that were already completely processed by the algorithm (within the sub-tree rooted at t). Hence, in every node we can define $X_{>t}$-restricted σ sets of arguments that fulfill the conditions of the respective semantics σ. As an example, let us consider admissible semantics: A $X_{>t}$-restricted admissible set S for a sub-framework $F_{\geq t}$ has to be conflict free and it has to defend itself against the arguments in $X_{>t} \backslash S$. These conditions have to be satisfied by all arguments in $X_{>t}$. Arguments in $X_t \cap S$ have to be conflict-free but they can be attacked by arguments in $X_t \backslash S$ as they can still be defended somewhere above in the tree decomposition. Formally, this is defined as follows:

Definition 3.5. *Let $F = \langle A, R \rangle$ be an AF and $B \subseteq A$ a set of arguments from A. A set $S \subseteq A$ is a* B-restricted admissible set for F, *if S is conflict-free in F and S defends itself against all $a \in B \backslash S$.*

Note that if $B = X_{>t}$ we have that S is conflict-free and that S defends itself against all $a \in X_{>t} \backslash S$, i.e. there is no attack between completely processed arguments in $X_{>t} \backslash S$ and arguments in S.

Example 3.1. Let us consider the the sub-framework $F_{\geq s_3}$ from our example tree decomposition (see Figure 3.1) as shown below.



The $X_{>s_3}$ (or $\{d, g\}$)-restricted admissible sets for $F_{\geq s_3}$ are $\{\emptyset, \{b\}, \{c\}, \{d\}, \{b, d\}, \{c, e\}\}$.

Note that selected arguments only have to be defended against d and g in the example sketched above. As e is attacked by d, $\{e\}$ is not a $X_{>s_3}$-restricted admissible set. Furthermore, the $X_{>s_0}$-restricted admissible sets for $F_{\geq s_0}$ (of our example tree decomposition) correspond to the admissible sets of our example AF.

Colorings: Every bag X_t of a node t in the tree decomposition contains the arguments whose attack relations have to be considered during the computation of this node. As we traverse the decomposition in bottom-up order and due to the properties of tree decompositions, the arguments of $X_{>t}$ have already been completely considered in at least one sub-node of t. On basis of the arguments that were already considered we want to analyze the arguments of the current bag X_t: We introduce *colorings* that allow us to specify selected arguments from X_t and their relationship to other arguments in $X_{\geq t}$. In other words the concept of colorings allows us to store the information of relationships between arguments in $X_{\geq}$ solely by assigning colors to arguments in X_t. For admissible semantics we define the colorings as follows:

Definition 3.6. *Let t be a node of a tree decomposition for an AF F and X_t the bag of arguments in t. The* colorings for t *(for admissible semantics) are defined as functions $C : X_t \to \{in_a, def_a, att_a, out_a\}$.*

Remark 3.7. *Given a coloring C for a node t, we denote the set $[C]_{i_a} = \{a \mid C(a) = in_a\}$, i.e. $[C]_{i_a}$ contains all arguments that are marked with in_a. The colorings def_a, att_a and out_a describe the relationship between between $[C]_{i_a}$ and the other arguments of $X \backslash [C]_{i_a}$.*

If an argument is marked with in_a it is contained in the set of selected arguments S. Furthermore, all arguments from $X_{>t}$ that were colored with in_a are also contained in S. In a coloring for arguments, $[C]_{i_a}$ contains all selected arguments from X_t. The coloring def_a for an argument a denotes that it is attacked by $[C]_{i_a}$. Furthermore, an argument is colored with att_a if it attacks the set $[C]_{i_a}$ but is not attacked by $[C]_{i_a}$. Finally, out_a describes that the respective argument is neither attacked by $[C]_{i_a}$ nor attacks $[C]_{i_a}$.

Valid Colorings: It remains to formally define sets of colorings that we consider to be valid within a node t for the respective semantics.

Definition 3.8. *[Dvořák et al., 2010a] Let t be the node of a tree decomposition for an AF F. Given a coloring C for t, we define the* extensions of C, $e_t(C)$, *as the collection of $X_{>t}$-restricted admissible sets S for $F_{\geq t}$ which satisfy the following conditions for each $a \in X_t$:*

$$C(a) = in_a \quad \textit{iff } a \in S$$
$$C(a) = def_a \textit{ iff } S \rightarrowtail a$$
$$C(a) = att_a \textit{ iff } S \not\rightarrowtail a \textit{ and } a \rightarrowtail S$$
$$C(a) = out_a \textit{ iff } S \not\rightarrowtail a \textit{ and } a \not\rightarrowtail S$$

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

Example 3.2. Let us again consider the node s_3 of our example tree decomposition. Furthermore, assume the colorings $C(b) = in_a$, $C(c) = def_a$ and $C(e) = def_a$. We already identified the $X_{>s_3}$ (or $\{d, g\}$)-restricted admissible sets for $F_{\geq s_3}$ as $\emptyset$, $\{b\}$, $\{c\}$, $\{d\}$, $\{b, d\}$ and $\{c, e\}$. $\{b, d\}$ is the only set S that additionally fulfills the conditions from Definition 3.8. It is therefore a valid coloring. On the other hand, consider for example $S = \{b\}$. Then, condition 2 of Definiton 3.8 is violated as $C(e) = def_a$ but $\{b\}$ does not attack e.

Goal: We want to compute the extensions of an AF $F = \langle A, R \rangle$ for a given semantics σ. The tree decomposition $(\mathcal{T}, \mathcal{X})$ is traversed in bottom-up order where we can compute the valid colorings $\mathcal{C}_t$ for every node $t \in \mathcal{T}$. Furthermore, our tree decompositions have a root node r with an empty bag $X_r = \emptyset$ of arguments. Hence, $X_{>r} = A$ holds. If we now compute the valid colorings $\mathcal{C}_r$ we obtain the $X_{>r}$-restricted admissible sets for $F_{\geq r}$ which correspond to the admissible sets of arguments for F (as shown by Dvořák et al. [2010a]).

V-Colorings: The computation and definition steps outlined above demand that $e_t(\cdot)$ is computed explicitly in every node of the decomposition. This is computationally expensive (we have to analyze the sub-framework $F_{\geq t}$ in every node) and fixed-parameter tractability with respect to tree-width is no longer guaranteed. Therefore we introduce the concept of *v-colorings*: The v-colorings are efficiently computed in every node t of the tree decomposition for all arguments in X_t. This computation is solely based on the v-colorings of the successor node(s) and the arguments in X_t. Hence it is not necessary to explicitly compute $e_t(C)$. Obvioulsy, we have to prove that the defined $v - colorings$ are equivalent to valid colorings.

Algorithms on Tree Decompositions

Our algorithms consist of three main parts, namely preparation, computation and result delivery.

In the preparation step the problem instance is read and the tree decomposition is computed:

1. First, read in the argumentation framework $F = \langle A, R \rangle$ in a predefined format. We can read arguments as well as attack relations between arguments.

2. From the argumentation framework, that is internally represented as a graph, obtain a tree decomposition $(\mathcal{T}, \mathcal{X})$ where $\mathcal{T} = (A_\mathcal{T}, R_\mathcal{T})$. In this thesis, we do not directly focus on how to gain 'good' decompositions (See Section 2.5 for a brief overview).

3. Based on the definition of the algorithm, obtain a normalized or semi-normalized tree decomposition. This can be achieved by simply traversing the tree in top-down order. If a branch node has several children, introduce new branch nodes until all of them are binary. For normalized tree decompositions, add insert nodes until only at most one argument is introduced in these nodes. Additionally, make sure that at most one argument is removed. For semi-normalized tree decompositions this step can be skipped.

The computation is defined on the tree decomposition where the tree is traversed in bottom-up order. Based on the type of node different actions are defined. We outline the intuition behind the computation of v-colorings within the different node types on basis of admissible semantics. The idea originates from Dvořák et al. [2010a]. They additionally proved that v-colorings and valid colorings coincide in every node of the tree decomposition.

Leaf Node: At each leaf node t compute all combinations of the arguments from the bag X_t. Discard any combination where two adjacent arguments are selected. This corresponds to the conflict-freeness of arguments: Two arguments that are connected can never be contained in a final solution. Note that this holds for all semantics that are defined in this thesis. Furthermore,

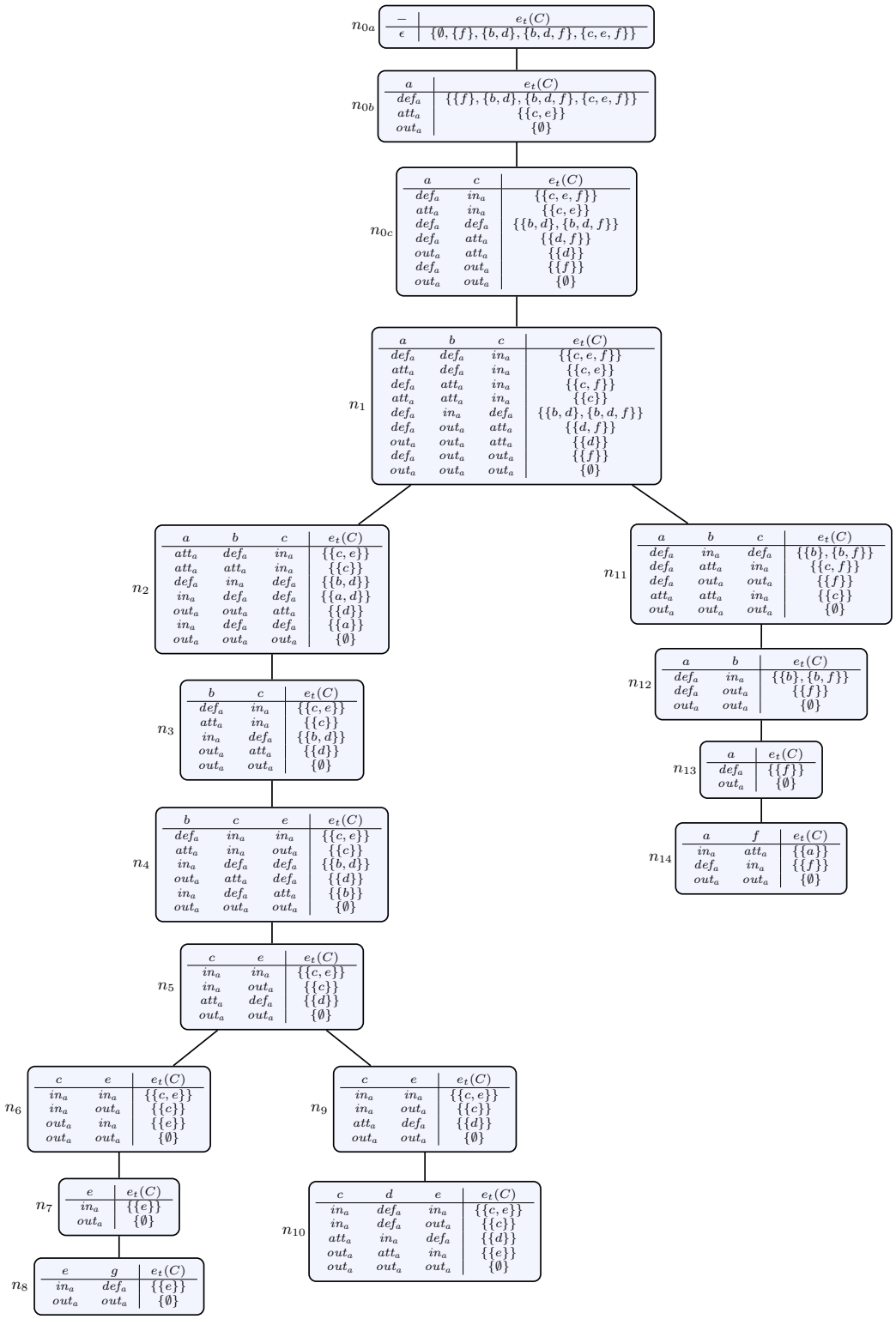


Figure 3.2: Normalized Tree Decomposition with V-Colorings for Admissible Semantics

assign v-colorings to the arguments: The colorings are defined by the respective algorithms (Depending on the semantics different colorings are assigned). For admissible semantics it is sufficient that arguments that are colored with in_a are conflict-free as $X_{>t} = \emptyset$ in the leaf node. The v-colorings are defined as follows:

Definition 3.9. *[Dvořák et al., 2010a] Let t be a* leaf node *of the tree decomposition and consider the colorings $X_t \to \{in_a, def_a, att_a, out_a\}$. If*

$$C(x) = in_a \Rightarrow C(y) \in \{att_a, def_a\} \text{ for all } y \rightarrowtail x$$
$$C(x) = def_a \Leftrightarrow \exists y : C(y) = in_a \text{ and } y \rightarrowtail x$$
$$C(x) = att_a \Rightarrow \exists y : C(y) = in_a \text{ and } x \rightarrowtail y$$

holds for all $x \in X_t$, the coloring is a v-coloring for t.

Figure 3.2 shows a normalized tree decomposition of our example AF. Consider, for example, the leaf node n_{10}: The bag $X_{n_{10}}$ contains the arguments for the sub-framework $F_{n_{10}} = \langle\{c, d, e\}, \{(c, d), (d, e)\}\rangle$. Every row represents a v-coloring for n_{10}. The conflict-free sets consist of all arguments colored with in_a. The sets of valid colorings are given on the right side. Now, consider for example a coloring $C(c) = def_a$, $C(d) = in_a$ and $C(e) = def_a$. $\{d\}$ is a conflict-free set but $C(c) = def_a$ violates the second condition for v-colorings in leaf nodes: It is not attacked but attacks the conflict-free set. Hence, a valid v-coloring would be $C(c) = att_a$, $C(d) = in_a$ and $C(e) = def_a$.

Introduction Node: For each introduction node t with child node t_1 (in normalized tree decompositions), combine the new argument with the arguments from X_{t_1}. Based on the definition of the algorithm colors of old arguments may change. Only colors for arguments of the sub-framework F_t have to be considered, e.g. one has to check if there exists an attack relation between the new argument and the other arguments in X_t. Hence, we can obtain the v-colorings for X_t using the v-colorings from X_{t_1}. Note that we do not explicitly compute the extensions $e_t(C)$ for the sub-framework $F_{\geq t}$.

Definition 3.10. *[Dvořák et al., 2010a] Let t be an* introduction node *of a tree decomposition, t_1 be the child node of t and let a be the argument that is introduced in X_t. If C is a v-coloring for t_1 then $C + a$ is a v-coloring for t. If also $a \not\rightarrowtail a$, $[C]_{i_a} \not\rightarrowtail a$ and $a \not\rightarrowtail [C]_{i_a}$ then $C \dot{+} a$ is a v-coloring for t as well. For $C : A \to \{in_a, out_a att_a, def_a\}$, $C + a$ and $C \dot{+} a$ are defined as follows:*

$$(C + a)(b) = \begin{cases} C(b) & \text{if } b \in A \\ def_a & \text{if } b = a \text{ and } [C]_{i_a} \rightarrowtail a \\ att_a & \text{if } b = a \text{ and } [C]_{i_a} \not\rightarrowtail a \text{ and } a \rightarrowtail [C]_{i_a} \\ out_a & \text{otherwise} \end{cases}$$

$$(C \dot{+} a)(b) = \begin{cases} in_a & \text{if } b = a \text{ or } C(b) = in_a \\ def_a & \text{if } a \neq b \text{ and } ((a,b) \in F_t \text{ or } C(b) = def_a) \\ out_a & \text{if } a \neq b \text{ and } C(b) = out_a \text{ and } (a,b) \notin F_t \text{ and } (b,a) \notin F_t \\ att_a & \text{otherwise} \end{cases}$$

Consider the introduction node n_{12} of the tree decomposition in Figure 3.2 for admissible semantics where the argument b is introduced. Then, b can be colored with one of att_a, def_a or out_a and, if $[C]_{i_a} \not\rightarrow a$ and $a \not\rightarrow [C]_{i_a}$ it can be colored with in_a. In the child node n_{13} a is colored with def_a (in the coloring of the first row). For $C + b$ we can assign the coloring out_a to the introduced argument b. Furthermore, for $C \dot{+} b$ we can color b with in_a as it is not attacked by $[C]_{i_a} = \emptyset$ in n_{13} or attacks $[C]_{i_a} = \emptyset$ in n_{13} (of the coloring in the first row).

Removal Node: For each removal node t (in normalized tree decompositions), delete the coloring of the removed argument from all sets of colorings. Depending on the algorithm, it may be possible to delete a complete set of colorings. This is due to the fact that we know that the removed argument is completely processed: Because of the properties of tree decompositions an argument that is removed can not reappear in a bag of another node upwards the tree. Furthermore, all arguments that that are connected via attack relations in the original AF have to appear together somewhere in a bag of the tree. For admissible semantics, this is defined as follows:

Definition 3.11. *[Dvořák et al., 2010a] Let t be a* removal node *of a tree decomposition, t_1 be the child node of t and let a be the argument that is removed in X_t. If C is a v-coloring for t_1 and $C(a) \neq att_a$ then $C - a$ is a v-coloring for t. For $C : A \rightarrow \{in_a, out_a, att_a, def_a\}$, $C - a$ is defined as follows:*

$$(C - a)(b) = C(b) \text{ for each } b \in A \setminus \{a\}$$

Now, consider the removal node n_{13} of Figure 3.2 where argument f is removed. In the first coloring of the child node n_{14} a is colored with in_a. As f attacks a it is colored with att_a. As f is removed in n_{13} we know that it will never reappear above in the tree and can never be attacked by another argument, i.e. a will never be defended against f. This contradicts the definition of admissible semantics. Hence, we can remove the complete coloring.

Branch Node: For each branch node t combine the v-colorings of the child nodes t_1 and t_2. How to combine the v-colorings heavily depends on the respective algorithm. A nice feature of normalized and semi-normalized tree decompositions is that the bags of the child nodes contain the same arguments. This property simplifies the computation of v-colorings and certain optimizations can be applied in branch nodes. A detailed analysis of optimization strategies is given in Chapter 4. For admissible semantics, the computation is defined as follows:

Definition 3.12. *[Dvořák et al., 2010a] Let t be* branch node *of a tree decomposition, and let C be the v-coloring for the child node t_1 and D be the v-coloring for the child node t_2. If*

$[C]_{i_a} = [D]_{i_a}$, *then* $C \bowtie D$ *is a v-coloring for* t. *For* $C, D : A \to \{in_a, out_a, att_a, def_a\}$, $C \bowtie D$ *is defined as follows:*

$$(C \bowtie D)(b) = \begin{cases} in_a & \text{if } C(b) = D(b) = in_a \\ out_a & \text{if } C(b) = D(b) = out_a \\ def_a & \text{if } C(b) = def_a \text{ or } D(b) = def_a \\ att_a & \text{otherwise} \end{cases}$$

Let us again consider our example tree decomposition (see Figure 3.2). n_5 is a branch node where the v-colorings of the child nodes are combined. Based on the definition we combine all v-colorings C (from n_6) and D (from n_9) where $[C]_{i_a} = [D]_{i_a}$, i.e. where the same arguments are colored with in_a. Then, we compute the colorings of the remaining arguments: If an argument is colored with def_a in one child node it is defended in the the current sub-framework F_{n_5}. If it is only attacked (in at least one child node) but not defended in the other it attacks the set of selected arguments $[C]_{i_a}$ in node n_6. If it is neither defended (colored with def_a) or attacks the set (colored with att_a) in the child nodes (and hence colored with out_a in both nodes) it must be colored with out_a in n_5.

Result Delivery: After the *root node* r, only a v-coloring C of the empty set remains (Note that in every node t the v-colorings for the current arguments in X_t are (re)-computed. As the bag X_r for the root node is always empty, the v-colorings just color the empty set). The extensions of C, $e_r(C)$ are the A-restricted σ sets for the sub-framework induced by r, i.e. $F_{\geq r}$. This sub-framework corresponds to the original argumentation framework F as well as the A-restricted σ sets correspond to the σ extensions of F. Hence, the σ extensions of F are given by $e_r(C)$.

For admissible semantics we have that $e_r(C)$ contains the A-restricted admissible sets for $F_{\geq r} = F$. This, in turn, is equivalent to all admissible extensions of F as for each $S \in e_r(C)$ we have that S must be conflict-free and S attacks all arguments $A \backslash S$ of F.

In the following we will introduce novel algorithms for stable and complete semantics on normalized tree decompositions and an algorithm for admissible semantics on semi-normalized tree decompositions. We will define the computation of v-colorings for sub-frameworks within the tree decomposition and prove that the solutions $e_r(C)$ in the root node r correspond to the extensions of the respective semantics. Furthermore, we outline how credulous as well as skeptical acceptance can be computed within the algorithms.

3.2 Algorithm for Stable Semantics (Normalized)

Restricted Sets:

Definition 3.13. *Let $F = \langle A, R\rangle$ be an AF and $B \subseteq A$ a set of arguments from A. A set $S \subseteq A$ is a* B-restricted stable set for F*, if S is conflict-free in F and S attacks all $a \in B \backslash S$.*

Colorings:

Definition 3.14. *Let t be a node of a tree decomposition for an AF F and X_t the bag of arguments in t. The* colorings for t *(for stable semantics) are defined as functions*

$$C_t : X_t \to \{in_s, def_s, out_s\}.$$

Furthermore, in a coloring C, we define the set of arguments a that are colored with in_s as

$$[C]_{i_s} = \{a \mid C(a) = in_s\}.$$

Definition 3.15. *Let t be the node of a tree decomposition for an AF F. Given a coloring C for t, we define the* extensions of C, $e_t(C)$*, as the collection of $X_{>t}$-restricted stable sets S for $F_{\geq t}$ which satisfy the following conditions for each $a \in X_t$:*

$$\begin{aligned} &C(a) = in_s \quad \textit{iff } a \in S \\ &C(a) = def_s \textit{ iff } S \rightarrowtail a \\ &C(a) = out_s \textit{ iff } S \not\rightarrowtail a \textit{ and } a \notin S \end{aligned}$$

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

By definition, all extensions of C, $e_t(C)$, are $X_{>t}$-restricted stable sets S for $F_{\geq t}$. It remains to show that also the other direction holds:

Lemma 3.16. *Let t be a node of a tree decomposition for an AF F and S be an $X_{>t}$-restricted stable set for $F_{\geq t}$. Then, there is a coloring $C \in \mathcal{C}_t$ such that $S \in e_t(C)$.*

Proof. By assumption, S is an $X_{>t}$-restricted stable set for $F_{\geq t}$. It remains to show that we can define a coloring C for S such that $S \in e_t(C)$. For an argument $a \in X_t$ we can distinguish three different cases and can assign the following colorings:

$$\begin{aligned} &\text{if } a \in S : && C(a) = in_s \\ &\text{if } S \rightarrowtail a : && C(a) = def_s \\ &\text{if } a \notin S \text{ and } S \not\rightarrowtail a : && C(a) = out_s \end{aligned}$$

As S is an $X_{>t}$-restricted stable set and due to the construction of C, S satisfies the conditions of Definition 3.15. Hence, $S \in e_t(C)$, i.e. for every $X_{>t}$-restricted stable set S for $F_{\geq t}$ there exists an extension for C. □

Next, we want to show that two different colorings C and C' for a node t represent different $X_{>t}$-restricted stable sets.

Lemma 3.17. *Let t be a node of a tree decomposition for an AF F and C and C' be two different colorings for the bag X_t. Then, the extensions of C and C' are disjoint, i.e. $e_t(C) \cap e_t(C') = \emptyset$.*

Proof. Suppose that there exist two distinct colorings C and C' for a bag X_t of a node t in a tree decomposition and that there exists a set S such that $S \in e_t(C) \cap e_t(C')$. Then, given an argument $a \in X_t$, at least one color $C(a)$ must be different from $C'(a)$. Towards a contradiction, we can distinguish three cases:

1. Suppose that $C(a) = in_s$ and $C'(a) = def_s$. By Definition 3.15, $C(a) = in_s$ implies that $a \in S$. $C'(a) = def_s$, on the other hand, implies that $S \rightarrowtail a$. Due to the conflict-freeness of $X_{>t}$-restricted stable sets, $S \rightarrowtail a$ implies that $a \notin S$, a contradiction.

2. Suppose that $C(a) = in_s$ and $C'(a) = out_s$. Again, $C(a) = in_s$ implies that $a \in S$ but due to Definition 3.15, for $C'(a) = out_s$ $a \notin S$ must hold, we have a contradiction.

3. Suppose that $C(a) = def_s$ and $C'(a) = out_s$. Then, due to Definition 3.15, $C(a) = def_s$ implies that $S \rightarrowtail a$ but $C'(a) = out_s$ implies that $S \not\rightarrowtail a$. Again, we have a contradiction.

The other cases, where the colors for a are exchanged in C and C', follow by symmetry. Hence, $e_t(C) \cap e_t(C') = \emptyset$. □

We defined that all extensions of C, $e_t(C)$, are $X_{>t}$-restricted stable sets S for $F_{\geq t}$ (by Definition 3.15). Furthermore we proved that for all $X_{>t}$-restricted stable sets S for $F_{\geq t}$ there exists a valid coloring C (see Lemma 3.16). Finally, we proved that different valid colorings C and C' do not represent any S where $S \in e_t(C)$ and $S \in e_t(C')$. For every $X_{>t}$-restricted stable set we thus have a unique coloring.

Leaf Node:

Definition 3.18. *Let t be a* leaf node *of the tree decomposition and consider the colorings $X_t \rightarrow \{in_s, def_s, out_s\}$. If*

$$
\begin{aligned}
&C(x) = in_s \;\; \Rightarrow C(y) \in \{def_s, out_s\} \textit{ for all } y \rightarrowtail x \\
&C(x) = def_s \Leftrightarrow [C]_{i_s} \rightarrowtail x \\
&C(x) = out_s \Rightarrow [C]_{i_s} \not\rightarrowtail x
\end{aligned}
$$

holds for all $x \in X_t$, the coloring is a stable v-coloring for t.

Lemma 3.19. *For any* leaf node t *in a tree-decomposition of an AF, valid colorings and v-colorings for stable semantics coincide.*

Proof. In every leaf node t the set of arguments $X_{>t}$ is the empty set. Hence, the $X_{>t}$ (or $\emptyset$)-restricted stable sets for $F_{\geq t}$ correspond to the conflict-free sets.

$\Rightarrow$: Given a valid coloring C we have to prove that C is also a v-coloring for t. If C is a valid coloring it satisfies the conditions of Definition 3.15. Hence, there exists a conflict-free set $S \in e_t(C)$. Now, consider an argument $a \in S$. Then, due to the definition it is colored with in_s. Furthermore, all attackers b of a, where $b \rightarrowtail a$ can not be colored with in_s, i.e. they must be colored with def_s or out_s. Hence, the first implication of Definition 3.18 for v-colorings is satisfied. Now, consider an argument a that satisfies $S \rightarrowtail a$. Due to the second condition of Definition 3.15 it must be colored with def_s. As all arguments $a \in S$ are colored with in_s, S is exactly the set $[C]_{i_s}$ (as defined in Def. 3.14). Hence, the second condition for v-colorings is satisfied. Due to the third condition of Definition 3.15 for valid colorings arguments colored with out_s are neither in S nor are attacked by S. As $S = [C]_{i_s}$, the third condition for v-colorings is satisfied.

$\Leftarrow$: Now, consider a v-coloring C for t. We have to prove that C is also a valid coloring for t. We claim that $[C]_{i_s} \in e_t(C)$. Suppose the opposite, i.e. that C is not a valid coloring for t. Then, either the set $[C]_{i_s}$ is not conflict-free or one of the other conditions of Definition 3.15 is not satisfied.

- If $[C]_{i_s}$ is not conflict-free, there exist two arguments $C(a) = in_s$, $C(b) = in_s$ where $a \rightarrowtail b$. But, by Definition 3.18 for v-colorings, all arguments a that attack an argument b with $C(b) = in_s$ are either colored with def_s or out_s.

- Now, assume that condition 2 for valid colorings is not satisfied, i.e. $C(a) = def_s$ but $S \not\rightarrowtail a$ or vice versa. But as $S = [C]_{i_s}$ and due to condition 2 of Definition 3.18 we have a contradiction.

- Finally, we show that condition 3 of Definition 3.15 is always satisfied: If $C(a) = out_s$ then $a \notin [C]_{i_s}$ and by condition 3 of Definition 3.18 also $[C]_{i_s} \not\rightarrowtail a$ holds. As $a \notin [C]_{i_s}$ $C(a) \neq in_s$ must hold. Furthermore, as $[C]_{i_s} \not\rightarrowtail a$, condition 2 of Definition 3.18 can not be satisfied. Hence, $C(a) \neq def_s$. Hence, condition 3 of Definition 3.15 is always satisfied.

□

Introduction Node:

Definition 3.20. *Let t be an* introduction node *of a tree decomposition, t_1 be the child node of t and let a be the argument that is introduced in X_t. If C is a v-coloring for t_1 then $C + a$ is a v-coloring for t. If also $a \not\rightarrowtail a$, $[C]_{i_s} \not\rightarrowtail a$ and $a \not\rightarrowtail [C]_{i_s}$ then $C \dot{+} a$ is a v-coloring for t as well. For $C : A \to \{in_s, def_s, out_s\}$ we define $C + a$ and $C \dot{+} a$ as follows:*

$$(C + a)(b) = \begin{cases} C(b) & \text{if } b \in A \\ def_s & \text{if } b = a \text{ and } [C]_{i_s} \rightarrowtail a \\ out_s & \text{if } b = a \text{ and } [C]_{i_s} \not\rightarrowtail a \end{cases}$$

$$(C \dot{+} a)(b) = \begin{cases} in_s & \text{if } b = a \text{ or } C(b) = in_s \\ def_s & \text{if } a \neq b \text{ and } ((a,b) \in F_t \text{ or } C(b) = def_s) \\ out_s & \text{if } a \neq b,\ C(b) = out_s,\ (a,b) \notin F_t \end{cases}$$

Lemma 3.21. *For any* introduction node t *in a tree-decomposition of an AF, valid colorings and v-colorings for stable semantics coincide if they coincide in the child node* t_1 *of* t.

Proof. Let t be an introduction node of a tree decomposition and let t_1 be the child node of t. Furthermore, assume that $X_t = X_{t_1} \cup \{a\}$ where $a \notin X_{t_1}$, i.e. the argument a is the introduced argument. Then, $X_{\geq t} = X_{\geq t_1} \cup \{a\}$. Furthermore, by Definition 2.18 of normalized tree decompositions, we know that a does not appear in the sub-framework $F_{>t} = F_{>t_1}$. Hence, $X_{>t} = X_{>t_1}$ and there do not exist any attack relations between $X_{>t}$ and the new argument a.

$\Rightarrow$: Suppose there exists a valid coloring C for t. Furthermore, by assumption, there exists a valid coloring (and v-coloring) C_1 for the child node t_1. We want to prove that C is a v-coloring for t. Then, there exists a set $S \in e_t(C)$ that is an $X_{>t}$-restricted stable set for $F_{\geq t}$. As $X_{>t} = X_{>t_1}$, S is also an $X_{>t_1}$-restricted stable set for $F_{\geq t}$. Furthermore, as $a \notin X_{t_1}$, a can not attack any argument in X_{t_1}. Then, $S \backslash \{a\}$ must also be an $X_{>t_1}$-restricted stable set for $F_{\geq t_1}$. Now, the approach is as follows: We define a coloring C_1 that fulfills the properties for valid colorings as defined in Definition 3.15. If we can define this coloring, based on S, we furthermore know that is a v-coloring for t_1. Finally, we simply have to check if $C = C_1 + a$ (for $a \notin S$) and $C = C_{t_1} \dot{+} a$ (for $a \in S$) hold. Then, C is a v-coloring for t. For an argument $b \in X_{t_1}$ we define the coloring for C_1 as follows:

$$\begin{array}{ll} \text{if } b \in S \backslash \{a\} : & C_1(b) = in_s \\ \text{if } b \notin S \text{ and } S \backslash \{a\} \rightarrowtail b : & C_1(b) = def_s \\ \text{if } b \notin S \text{ and } S \backslash \{a\} \not\rightarrowtail b : & C_1(b) = out_s \end{array}$$

The conditions for our defined colorings C_1 correspond to the conditions of Definition 3.15 for colorings. Furthermore, as $S \backslash \{a\}$ is an $X_{>t_1}$-restricted stable set, all properties for valid colorings are satisfied and we have that C_1 is a valid coloring and a v-coloring for t_1. Finally, we have to distinguish two cases:

- Consider the case $a \in S$. We have to show that $C = C_1 \dot{+} a$ is a v-coloring for t. By assumption, C is a valid coloring. Therefore, C is conflict-free which means that $a \not\rightarrowtail a$, $S \not\rightarrowtail a$ and $a \not\rightarrowtail S$ must hold. Furthermore, by Definition 3.15, if $a \in S$, $C(a) = in_s$ and by Definition 3.14, $[C]_{in_s} \subseteq S$. Hence, the conditions of Definition 3.20 for $C_1 \dot{+} a$ are met and we have that $(C_1 \dot{+} a)(a) = in_s$. But then, $(C_1 \dot{+} a)(a) = in_s = C(a)$ holds. It is easy to see that $(C_1 \dot{+} a)(b) = C(b)$ also holds for any other argument $b \in S$.

- Consider the case $a \notin S$. We have to show that $C = C_{t_1} + a$ is a v-coloring for t. We know that C_1 is conflict-free by assumption. Furthermore, as $a \notin S$, a can not be colored with in_s. Then, $S = [C_1 + a]$ must also be conflict-free. In here, we outline the equivalence of $C = C_1 + a$ for the introduced argument a where $S \rightarrowtail a$. Then, by Definition 3.15, $C(a) = def_s$. Furthermore also $[C]_{in_s} \rightarrowtail a$ holds, i.e. , as defined in Definition 3.20, also

$(C_1 + a)(a) = def_s$. It's easy to see that, by similar argumentation, $C = C_1 + a$ holds for the colors in_s and out_s as well.

$\Leftarrow$: Suppose there exists a v-coloring C for t, i.e. there exists a v-coloring C_1 for the child node t_1 where either $C = C_1 + a$ or $C = C_1 \dot{+} a$ holds. By assumption, C_1 is also a valid coloring. We want to prove that C is a valid coloring for t. As C_1 is a valid coloring, there exists an $X_{>t_1}$-restricted stable set $S \in e_{t_1}(C_1)$ for $F_{\geq t_1}$. As $X_{>t_1} = X_{>t}$ it is also an $X_{>t}$-restricted stable set.

- For $C_1 + a$, we have that by Definition 3.20 the colors for each argument $b \neq a$ remain the same. Furthermore, as $a \notin e_t(C)$ (because it is not colored with in_s) S must be conflict-free and is either colored with def_s or out_s (if $S \rightarrowtail a$ or $S \not\rightarrow a$). Then, S must be an extension for $C_1 + a$, i.e. $S \in e_t(C_1 + a)$, in other words $C_1 + a$ is a valid coloring.

- For $C_1 \dot{+} a$, by Definition 3.20 we have that $a \not\rightarrow a$, $[C_1]_{i_s} \not\rightarrow a$ and $a \not\rightarrow [C_1]_{i_s}$ and hence $[C_1]_{i_s} \cup \{a\}$ is conflict-free. Furthermore, due to the connectedness condition of tree decompositions (see Definition 2.14) there is no attack between $X_{>t}$ and a. Hence, it is easy to see that $S \cup \{a\}$ is conflict-free and we have that $S \cup \{a\} \in e_t(C_1 \dot{+} a)$.

Finally, we obtain that C is a valid coloring for t. □

Removal Node:

Definition 3.22. *Let t be a* removal node *of a tree decomposition, t_1 be the child node of t and let a be the argument that is removed in X_t. If C is a v-coloring for t_1 and $C(a) \neq out_s$ then $C - a$ is a v-coloring for t. For $C : A \to \{in_s, def_s, out_s\}$ we define $C - a$ as follows:*

$$(C - a)(b) = C(b) \text{ for each } b \in A \setminus \{a\}$$

Lemma 3.23. *For any* removal node *t in a tree-decomposition of an AF, valid colorings and v-colorings for stable semantics coincide if they coincide in the child node t_1 of t.*

Proof. $\Rightarrow$: Suppose there exists a valid coloring C for t. Furthermore, by assumption, there exists a valid coloring C_1 that is also a v-coloring for t_1. We want to prove that C is a v-coloring for t. If C is a valid coloring for t then, by Definition 3.15, there exists an $X_{>t}$-restricted stable set S for $F_{\geq t}$ and the conditions of Definition 3.15 for C are satisfied, i.e. $S \in e_t(C)$. Furthermore, as t is a removal node, we know that exactly one argument a is removed in X_t and hence $X_t \cup \{a\} = X_{t_1}$. Furthermore, $X_{\geq t} = X_{\geq t_1}$ and $F_{\geq t} = F_{\geq t_1}$ must hold (because removed arguments remain in $X_{\geq t}$ and $F_{\geq t}$).

First, we show that we can define a valid coloring C_1 for t_1 where $C = C_1 - a$ and $C_1(a) \neq out_s$. The first condition is easily met by setting each argument, except a (the removed one), $b \in X_{t_1} \setminus \{a\}$ to $C_1(b) = C(b)$. Hence, $C = C_1 - a$ is always satisfied. It remains to define the color of a, $C_1(a)$:

- First, assume that $a \in S$. Then, we set $C_1(a) = in_s$. By assumption, S is an $X_{>t}$-restricted stable set S for $F_{\geq t}$. As $X_{>t} = X_{>t_1} \cup \{a\}$, S is also an $X_{>t_1}$-restricted stable set for $F_{\geq t} = F_{\geq t_1}$ (By removing a the condition must still be satisfied). For the same reason, as $S \in e_t(C)$, $S \in e_{t_1}(C_1)$ must hold (The conditions of Definition 3.15 are satisfied for $S \in e_t(C)$ and hence for $S \in e_{t_1}(C_1)$). By assumption, C_1 is a valid coloring and a v-coloring, and hence, due to Definition 3.22 and $C_1(a) \neq out_s$, $C = C_1 - a$ is a v-coloring for t.

- Now, assume that $S \rightarrowtail a$. Then, we set $C_1(a) = def_s$. As S is an $X_{>t}$-restricted stable set for $F_{\geq t}$ (by assumption) it must also be a $X_{>t_1}$-restricted stable set for $F_{\geq t_1}$ (for the same reason as above). Furthermore, by assumption, $S \in e_t(C)$. As $S \rightarrowtail a$ and $C_1(a) = def_s$ the conditions of Definition 3.15 are satisfied and it follows that $S \in e_t(C_1)$. As C_1 is a valid coloring and v-coloring (by assumption) for t_1, we have that C s a v-coloring for t.

$\Leftarrow$: Suppose that there exists a v-coloring C for t. Furthermore, by assumption, there exists a v-coloring C_1 for t_1 such that $C = C_1 - a$ and $C_1(a) \neq out_s$. Then, C_1 is also a valid coloring for t_1, i.e. there exists an $X_{>t_1}$-restricted stable set $S \in e_t(C_1)$. As $C_1(a) \neq out_s$, we know that $S \rightarrowtail a$ or $a \in S$ (by Definition 3.15). But then, as defined in 3.13, S must attack all $(X_{>t_1} \cup \{a\}) \backslash S$. As $X_{t_1} \cup \{a\} = X_{>t}$, S is also an $X_{>t}$-restricted stable set for $F_{\geq t}$. Then,by Definition 3.15, $S \in e_t(C)$, i.e. S is a valid coloring for t. □

Branch Node:

Definition 3.24. *Let t be* branch node *of a tree decomposition, and let C be the v-coloring for the child node t_1 and D be the v-coloring for the child node t_2. If $[C]_{i_s} = [D]_{i_s}$, then $C \bowtie D$ is a v-coloring for t. For $C, D : A \to \{in_s, def_s, out_s\}$ we define $C \bowtie D$ as follows:*

$$(C \bowtie D)(b) = \begin{cases} in_s & \text{if } C(b) = D(b) = in_s \\ def_s & \text{if } C(b) = def_s \text{ or } D(b) = def_s \\ out_s & \text{if } C(b) = D(b) = out_s \end{cases}$$

Lemma 3.25. *For any* branch node t *in a tree-decomposition of an AF, valid colorings and v-colorings for stable semantics coincide if they coincide in the child nodes t_1 and t_2 of t.*

Proof. Let t be a branch node of a normalized tree decomposition and t_1 and t_2 be the child nodes of t. By Definition 2.18 of branch nodes (for normalized tree decompositions) we have that $X_t = X_{t_1} = X_{t_2}$, $X_t = X_{\geq t_1} \cap X_{\geq t_2}$ and $X_{\geq t} = X_{\geq t_1} \cup X_{\geq t_2}$. Then, we can partition $X_{\geq t}$ into three disjoint sets $X_{>t_1}$, $X_{>t_2}$ and X_t. Furthermore, we can define a set $S \subseteq X_{\geq t}$ as the union of two sets S_1 and S_2, $S = S_1 \cup S_2$, where $S_1 \subseteq X_{\geq t_1}$, $S_2 \subseteq X_{\geq t_2}$ and $S_1 \cap X_t = S_2 \cap X_t$. In the following we prove that S is an $X_{>t}$-restricted stable set for $F_{\geq t}$ iff S_1 is an $X_{>t_1}$-restricted stable set for the sub-framework $F_{\geq t_1}$, S_2 is an $X_{>t_2}$-restricted stable set for the sub-framework $F_{\geq t_2}$ and $S_1 \cap X_t = S_2 \cap X_t$:

Lemma 3.26. *Let $S_1 \subseteq X_{\geq t_1}$ and $S_2 \subseteq X_{\geq t_2}$, such that*

- *S_1 is an $X_{>t_1}$-restricted stable set for $F_{\geq t_1}$*

- *S_2 is an $X_{>t_2}$-restricted stable set for $F_{\geq t_2}$*
- *$S_1 \cap X_t = S_2 \cap X_t$.*

Then, $S = S_1 \cup S_2$ is an $X_{>t}$-restricted stable set for $F_{\geq t}$.

Proof. As arguments that attack one another must be contained together in one bag and due to the connectedness condition of tree decompositions (see Definition 2.14) we know that there is no attack between arguments in $X_{>t_1}$ and $X_{>t_2}$. S is an $X_{>t}$-restricted stable set for $F_{\geq t}$ if it is (1) conflict-free in $F_{\geq t}$ and (2) S attacks all arguments $a \in X_{>t} \backslash S$ (see Definition 3.13).

(1) Suppose there exists a conflict where $a, b \in S$ and $a \rightarrowtail b$. As $S \subseteq X_{\geq t_1} \cup X_{\geq t_2}$ we can distinguish between two cases:

   (a) The arguments a and b are contained in $a, b \in X_{\geq t_1}$ (or $a, b \in X_{\geq t_2}$). Then, we get that also $a, b \in S_1$ (or $a, b \in S_2$) and therefore S_1 (or S_2) is not conflict-free in $F_{\geq t_1}$ (or $F_{\geq t_2}$), i.e. we have a contradiction to assumption 1 (or 2) of Lemma 3.26. (Note that this also holds if $a, b \in X_{\geq t_1} \cap X_{\geq t_2}$).

   (b) The argument a is contained in $a \in X_{\geq t_1}$ whereas the argument b is contained in $b \in X_{\geq t_2}$ (or vice-versa). As there is an attack $a \rightarrowtail b$ (by assumption) and due to Condition (ii) of Defintion 2.14 of tree decompositions we have that a and b have to appear together somewhere in a bag of the decomposition. Furthermore, due to (iii), the connectedness condition, we have that a or b must appear in X_t, i.e. $\{a\} \subseteq X_t$ or $\{b\} \subseteq X_t$. Hence, we have that $\{a, b\} \subseteq X_{\geq t_1}$ or $\{a, b\} \subseteq X_{\geq t_2}$, and assuming that $S_1 \cap X_t = S_2 \cap X_t$ we have that $\{a, b\} \subseteq S_1$ (or $\{a, b\} \subseteq S_2$), contradicting our assumption of S_1 (resp. S_2) being an $X_{>t_1}$ (resp. $X_{>t_2}$)-restricted stable set for $F_{\geq t_1}$ (resp. $F_{\geq t_2}$).

(2) Now we show that all arguments $a \in X_{>t} \backslash S$ are attacked by S.

   By assumption, S_1 attacks all arguments $X_{>t_1} \backslash S_1$ in $F_{\geq t_1}$ and hence also in $F_{\geq t}$. Furthermore, S_2 attacks all arguments $X_{>t_2} \backslash S_2$ in $F_{\geq t}$. Due to the connectedness condition of tree decompositions we have that $X_{>t_1} \cap X_{>t_2} = \emptyset$. Hence, as $S = S_1 \cup S_2$, we have that S attacks all arguments $X_{>t_1} \backslash S$ in $F_{\geq t}$. Due to symmetry, S also attacks all arguments $X_{>t_2} \backslash S$ in $F_{\geq t}$. As S attacks all arguments $X_{>t_1} \backslash S$ and $X_{>t_2} \backslash S$ it also attacks all arguments $(X_{>t_1} \cup X_{>t_2}) \backslash S$ in $F_{\geq t}$. But, as $X_{>t_1} \cup X_{>t_2} = X_{>t}$, we have that S is an $X_{>t}$-restricted stable set for $F_{\geq t}$.

Hence, based on our assumptions, we have that $S = S_1 \cup S_2$ is an $X_{>t}$-restricted stable set for $F_{\geq t}$. □

Lemma 3.27. *Let S be an $X_{>t}$-restricted stable set for $F_{\geq t}$, $S_1 = S \cap X_{\geq t_1}$ and $S_2 = S \cap X_{\geq t_2}$. Then,*

(1) S_1 is an $X_{>t_1}$-restricted stable set for $F_{\geq t_1}$

(2) S_2 is an $X_{>t_2}$-restricted stable set for $F_{\geq t_2}$

(3) $S_1 \cap X_t = S_2 \cap X_t$.

Proof. Let S be an $X_{>t}$-restricted stable set for $F_{\geq t}$.

(1, 2) As S is conflict-free in $F_{\geq t}$ we have that the subset $S_1 = S \cap X_{\geq t_1}$ (respectively $S_2 = S \cap X_{\geq t_2}$) must be conflict-free in $F_{\geq t_1}$ ($F_{\geq t_2}$). It remains to show that S_1 attacks all arguments $X_{>t_1} \backslash S_1$. Then, by symmetry, it also holds that S_2 attacks all arguments $X_{>t_2} \backslash S_2$. Suppose to the contrary that there exists an argument $a \in X_{>t_1}$ with $a \notin S_1$ and $S_1 \not\rightarrowtail a$ in $F_{\geq t_1}$. Since S is $X_{>t}$-restricted stable we know that $S \rightarrowtail a$ in $F_{\geq t}$ or, in other words, there must exist an argument $b \in S \backslash S_1$ that attacks a. As $a \notin X_{t_1}$ ($= X_t$), a and b can not appear together in a bag (due to the connectedness condition of tree decompositions). But by Definition 2.14 a and b must appear together in a bag of the decomposition, i.e. we have a contradiction.

(3) Immediate by $X_t = X_{\geq t_1} \cap X_{\geq t_2}$ and the definition of S_1 and S_2 it follows that also $S_1 \cap X_t = S_2 \cap X_t$.

□

For our proof of Lemma 3.25 it remains to show that valid colorings and v-colorings for stable semantics of a branch node t coincide if they coincide in the child nodes t_1 and t_2.

$\Leftarrow$: Suppose that we have v-coloring C for t where $C = C_1 \bowtie C_2$ and C_1 is a v-coloring for t_1 and C_2 is a v-coloring for t_2. Furthermore, by assumption, we have $[C_1]_{i_s} = [C_2]_{i_s}$. As C_1 and C_2 are valid colorings we have that there exists an extension $S_1 \in e_{t_1}(C_1)$ and $S_2 \in e_{t_1}(C_1)$. Furthermore, as $[C_1]_{i_s} = [C_2]_{i_s}$, we have that $S_1 \cap X_t = S_2 \cap X_t$. Hence, by Lemma 3.26 we have that $S = S_1 \cup S_2$ is an $X_{>t}$-restricted stable set for $F_{\geq t}$. It remains to show that the properties for valid colorings (as defined in 3.15) are satisfied, i.e. we prove that $S \in e_t(C)$:

- By Definition 3.24 we have that $C(a) = in_s$ iff $C_1(a) = in_s$ and $C_2(a) = in_s$. As C_1 and C_2 are valid colorings we have (by Definition 3.15) that $a \in S_1$ and $a \in S_2$. As $S = S_1 \cup S_2$ we have that $C(a) = in_s$ iff $a \in S$.

- By Definition 3.24 we have that $C(a) = def_s$ iff $C_1(a) = def_s$ or $C_2(a) = def_s$. As C_1 and C_2 are valid colorings this is equivalent to $S_1 \rightarrowtail a$ or $S_2 \rightarrowtail a$ (by Definition 3.15). As $S = S_1 \cup S_2$ we obtain $C(a) = def_s$ iff $S \rightarrowtail a$.

- By Definition 3.24 we have that $C(a) = out_s$ iff $C_1(a) = out_s$ and $C_2(a) = out_s$. As C_1 and C_2 are valid colorings we have (by Definition 3.15) that $a \notin S_1$, $S_1 \not\rightarrowtail a$, $a \notin S_2$ and $S_2 \not\rightarrowtail a$. Hence, we have that $C(a) = out_s$ iff $a \notin S$ and $S \not\rightarrowtail a$.

$\Rightarrow$: Suppose we have a valid coloring C for t. Then, there exists an extension $S \in e_t(C)$. Furthermore, we define $S_1 = S \cap X_{\geq t_1}$ and $S_2 = S \cap X_{\geq t_2}$. Then, by Lemma 3.27, we have that S_1 is an $X_{>t_1}$-restricted stable set for $F_{\geq t_1}$, S_2 is an $X_{>t_2}$-restricted stable set for $F_{\geq t_2}$ and $S_1 \cap X_t = S_2 \cap X_t$.

By Lemma 3.16 we have colorings C_1 and C_2 such that $S_1 \in e_{t_1}(C_1)$ and $S_2 \in e_{t_2}(C_2)$. Furthermore, as $S_1 \cap X_t = S_2 \cap X_t$, we have that $[C_1]_{i_s} = [C_2]_{i_s}$. Hence, $C^* = C_1 \bowtie C_2$ is a v-coloring. In the following we prove that $C^* = C$ and thus C is also a v-coloring.

- First, consider $C(a) = in_s$. Then, by Definition 3.15, $a \in S$. Then, $a \in X_t$ and as $X_t = X_{t_1} = X_{t_2}$ we have that $a \in S_1$ and $a \in S_2$. Furthermore, due to Definition 3.15, we have that $C_1(a) = C_2(a) = in_s$. By Definition 3.24 of v-colorings we finally have that $C^*(a) = (C_1 \bowtie C_2)(a) = in_s$.

- Next, consider $C(a) = def_s$. Then, by Definition 3.15, $S \rightarrowtail a$. As $S = S_1 \cup S_2$, either $S_1 \rightarrowtail a$ or $S_2 \rightarrowtail a$. We consider the first case, $S_1 \rightarrowtail a$. Due to Definition 3.15, we have that $C_1(a) = def_s$. But then, due to Definition 3.24 of v-colorings we finally have that $C^*(a) = (C_1 \bowtie C_2)(a) = def_s$.

- Finally, consider $C(a) = out_s$. Then, by Definition 3.15, $S \not\rightarrowtail a$ and $a \notin S$. As $S = S_1 \cup S_2$, $S_1 \not\rightarrowtail a$, $a \notin S_1$, $S_2 \not\rightarrowtail a$ and $a \notin S_2$. Due to Definition 3.15, we have that $C_1(a) = C_2(a) = out_s$. But then, due to Definition 3.24 of v-colorings we finally have that $C^*(a) = (C_1 \bowtie C_2)(a) = out_s$.

□

Theorem 3.28. *Let $(\mathcal{T}, \mathcal{X})$ be a normalized tree decomposition of an AF $F = \langle A, R \rangle$. 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 v-coloring for t.*

Proof. Lemma 3.19 to 3.25 state that valid colorings and v-colorings for stable semantics coincide in the four different node types of normalized tree decompositions. By induction over the tree decomposition we have that they coincide in every node of a normalized tree decomposition. □

A normalized tree decomposition of our running example (see Example 2.6) with v-colorings for stable semantics is given in Figure 3.3.

Example 3.3. Consider the leaf node n_8: g can not be colored with in_s as it attacks itself and thus the first condition of Definition 3.18 is not satisfied. Then, we have the colorings C'_{n_8} where $C'_{n_8}(e) = in_s$ and $C'_{n_8}(g) = def_s$, and respectively, C''_{n_8} where $C''_{n_8}(e) = out_s$ and $C''_{n_8}(g) = out_s$. In the removal node n_7 where g is removed we only have the coloring C'_{n_7} with $C'_{n_7}(e) = in_s$. C''_{n_8} does not satisfy the condition $C''_{n_8}(g) \neq out_s$. In n_6 where c is introduced we have that $X_t = \{c, e\}$. As there do not exist any attack relations between the two arguments c can either be colored with in_s or out_s. In the branch node n_5 only the colorings from the child nodes n_6 and n_9 are combined if $[C_{n_6}]_{i_s} = [C_{n_9}]_{i_s}$.

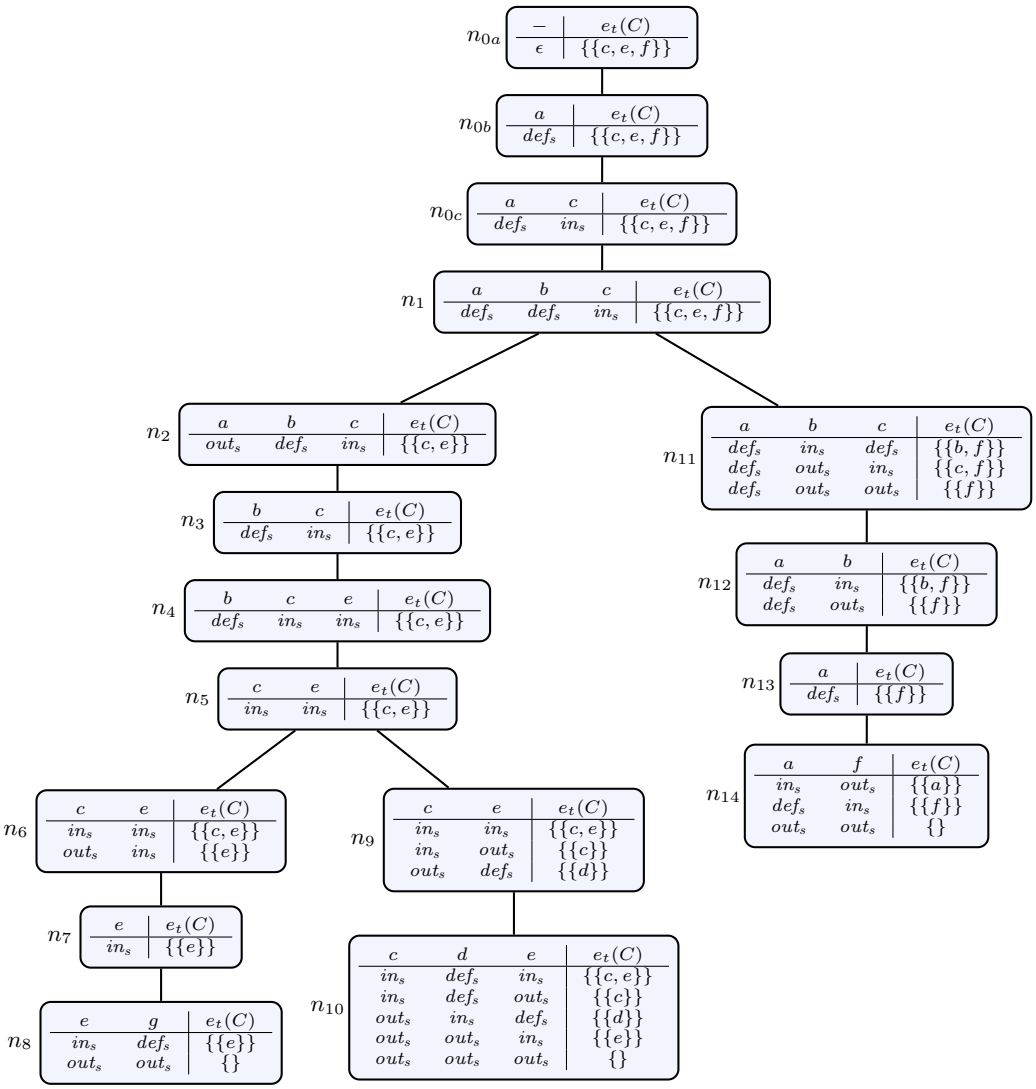


Figure 3.3: Normalized Tree Decomposition with V-Colorings for Stable Semantics

3.3 Algorithm for Complete Semantics (Normalized)

Complete Labelings: For the definition and proof of our algorithm for complete semantics we borrow the concept of labelings from Caminada and Gabbay [2009].

Definition 3.29. *Given an AF $F = \langle A, R\rangle$ a labeling $\mathcal{L}$ is a function $A \to \{in, def, out\}$. We denote such a function by a triple $\mathcal{L} = \langle \mathcal{L}_{in}, \mathcal{L}_{def}, \mathcal{L}_{out}\rangle$ where $\mathcal{L}_{in}$ is the set of arguments labeled with in, $\mathcal{L}_{def}$ is the set of arguments labeled with def and $\mathcal{L}_{out}$ is the set of arguments labeled with out.*

A triple $\mathcal{L} = \langle \mathcal{L}_{in}, \mathcal{L}_{def}, \mathcal{L}_{out}\rangle$ is a complete labeling *if it satisfies the following conditions:*

i $a \in \mathcal{L}_{in} \Leftrightarrow \{b \mid (b, a) \in R\} \subseteq \mathcal{L}_{def}$

ii $a \in \mathcal{L}_{def} \Leftrightarrow \mathcal{L}_{in} \rightarrowtail a$

iii $a \in \mathcal{L}_{out} \Leftrightarrow \mathcal{L}_{in} \not\rightarrowtail a \wedge \mathcal{L}_{out} \rightarrowtail a$

The following theorem follows from results presented in [Caminada and Gabbay, 2009]:

Theorem 3.30. *Let $F = \langle A, R\rangle$ be an argumentation framework. There is a one-to-one mapping between the complete labelings of F and the complete extensions of F, such that a complete labeling $\mathcal{L} = \langle \mathcal{L}_{in}, \mathcal{L}_{def}, \mathcal{L}_{out}\rangle$ corresponds to a complete extension $\mathcal{L}_{in}$.*

Restricted Labelings:

Definition 3.31. *Let $F = \langle A, R\rangle$ be an AF and $B \subseteq A$ a set of arguments from A. A labeling $\mathcal{L} = \langle \mathcal{L}_{in}, \mathcal{L}_{def}, \mathcal{L}_{out}\rangle$ for F is a* B-restricted complete labeling for F, *if $\mathcal{L}_{in}$ is conflict-free in F, $\mathcal{L}_{in} \not\rightarrowtail \mathcal{L}_{out}$, $\mathcal{L}_{out} \not\rightarrowtail \mathcal{L}_{in}$ and for each $a \in B$*

i $a \in \mathcal{L}_{in} \Leftrightarrow \{b \mid (b, a) \in R\} \subseteq \mathcal{L}_{def}$,

ii $a \in \mathcal{L}_{def} \Leftrightarrow \mathcal{L}_{in} \rightarrowtail a$,

iii $a \in \mathcal{L}_{out} \Leftrightarrow \mathcal{L}_{in} \not\rightarrowtail a \wedge \mathcal{L}_{out} \rightarrowtail a$.

Colorings:

Definition 3.32. *Let t be a node of a tree decomposition for an AF F and X_t the bag of arguments in t. The* colorings for t *(for complete semantics) are defined as functions*

$$C_t : X_t \to \{in_c, def_c, defp_c, out_c, outp_c\}.$$

Furthermore, in a coloring C, we denote the set of arguments, based on their color, as follows:

$$[C]_{i_c} = \{a \mid C(a) = in_c\}$$
$$[C]_{d_c} = \{a \mid C(a) = def_c \text{ or } C(a) = defp_c\}$$
$$[C]_{o_c} = \{a \mid C(a) = out_c \text{ or } C(a) = outp_c\}$$

Definition 3.33. *Let t be the node of a tree decomposition for an AF F. Given a coloring C for t, we define the* labelings of C, $l_t(C)$, *as the collection of $X_{>t}$-restricted complete labelings $\mathcal{L}$ for $F_{\geq t}$ which satisfy the following conditions for each $a \in X_t$:*

$$
\begin{aligned}
&C(a) = in_c \quad \textit{iff } a \in \mathcal{L}_{in} \\
&C(a) = def_c \quad \textit{iff } a \in \mathcal{L}_{def} \textit{ and } \mathcal{L}_{in} \rightarrowtail a \\
&C(a) = defp_c \textit{ iff } a \in \mathcal{L}_{def} \textit{ and } \mathcal{L}_{in} \not\rightarrowtail a \\
&C(a) = out_c \quad \textit{iff } a \in \mathcal{L}_{out} \textit{ and } \mathcal{L}_{in} \not\rightarrowtail a \textit{ and } a \not\rightarrowtail \mathcal{L}_{in} \textit{ and } \mathcal{L}_{out} \rightarrowtail a \\
&C(a) = outp_c \textit{ iff } a \in \mathcal{L}_{out} \textit{ and } \mathcal{L}_{in} \not\rightarrowtail a \textit{ and } a \not\rightarrowtail \mathcal{L}_{in} \textit{ and } \mathcal{L}_{out} \not\rightarrowtail a
\end{aligned}
$$

If $l_t(C) \neq \emptyset$, C is called a valid coloring *for t. We denote the set of valid colorings by $\mathcal{C}_t$.*

By definition, all labelings of C, $l_t(C)$, are $X_{>t}$-restricted complete labelings $\mathcal{L}$ for $F_{\geq t}$. It remains to show that also the other direction holds:

Lemma 3.34. *Let t be a node of a tree decomposition for an AF F and $\mathcal{L}$ be an $X_{>t}$-restricted complete labeling for $F_{\geq t}$. Then, there is a coloring $C \in \mathcal{C}_t$ such that $\mathcal{L} \in l_t(C)$.*

Proof. By assumption, $\mathcal{L}$ is an $X_{>t}$-restricted complete labeling for $F_{\geq t}$. It remains to show that we can define a coloring C for $\mathcal{L}$ such that $\mathcal{L} \in l_t(C)$. For an argument $a \in X_t$ we can distinguish different cases and can assign the following colors:

if $a \in \mathcal{L}_{in}$:	$C(a) = in_c$
if $a \in \mathcal{L}_{def}$ and $\mathcal{L}_{in} \rightarrowtail a$:	$C(a) = def_c$
if $a \in \mathcal{L}_{def}$ and $\mathcal{L}_{in} \not\rightarrowtail a$:	$C(a) = defp_c$
if $a \in \mathcal{L}_{out}$, $\mathcal{L}_{in} \not\rightarrowtail a$, $a \not\rightarrowtail \mathcal{L}_{in}$ and $\mathcal{L}_{out} \rightarrowtail a$:	$C(a) = out_c$
if $a \in \mathcal{L}_{out}$, $\mathcal{L}_{in} \not\rightarrowtail a$, $a \not\rightarrowtail \mathcal{L}_{in}$ and $\mathcal{L}_{out} \not\rightarrowtail a$:	$C(a) = outp_c$

As $\mathcal{L}$ is an $X_{>t}$-restricted complete labeling and due to the construction of C, $\mathcal{L}$ satisfies the conditions of Definition 3.33. Hence, $\mathcal{L} \in l_t(C)$, i.e. for every $X_{>t}$-restricted complete labeling $\mathcal{L}$ for $F_{\geq t}$ there exists a labeling for C.

□

Next, we want to show that two different colorings C and C' for a node t represent different $X_{>t}$-restricted complete labelings.

Lemma 3.35. *Let t be a node of a tree decomposition for an AF F and C and C' be two different colorings for the bag X_t. Then, the labelings of C and C' are disjoint, i.e. $l_t(C) \cap l_t(C') = \emptyset$.*

Proof. Suppose there exist two different colorings C and C' for a bag X_t of a node t in a tree decomposition and there exists a labeling $\mathcal{L}$ such that $\mathcal{L} \in l_t(C) \cap l_t(C')$. Then, there must exist an argument $a \in X_t$ where $C(a) \neq C'(a)$. Towards a contradiction, we analyze the following cases where $C(a) \neq C'(a)$:

- Suppose that $C(a) = in_c$ whereas $C'(a) = def_c$. By Definition 3.33, we have that $C(a) = in_c$ implies $a \in \mathcal{L}_{in}$. $C'(a) = def_c$, on the other hand, implies that $a \in \mathcal{L}_{def}$, a contradiction. The same argument holds for $C(a) = in_c$ and $C'(a) \in \{defp_c, out_c, outp_c\}$. It is easy to see that for $C(a) \in \{def_c, defp_c\}$ and $C'(a) \in \{out_c, outp_c\}$ a similar argument holds.

- Now, suppose that $C(a) = def_c$ whereas $C'(a) = defp_c$. By Definition 3.33, we have that $C(a) = def_c$ implies $\mathcal{L}_{in} \rightarrowtail a$. $C'(a) = defp_c$, on the other hand, implies that $\mathcal{L}_{in} \not\rightarrowtail a$, a contradiction.

- Finally, suppose that $C(a) = out_c$ whereas $C'(a) = outp_c$. By Definition 3.33, we have that $C(a) = out_c$ implies $\mathcal{L}_{out} \rightarrowtail a$. $C'(a) = outp_c$, on the other hand, implies that $\mathcal{L}_{out} \not\rightarrowtail a$, a contradiction.

The other cases, where the colors for a are exchanged in C and C', follow by symmetry. Hence, $l_t(C) \cap l_t(C') = \emptyset$. □

We defined that all $\mathcal{L} \in l_t(C)$ are X_t-restricted complete labelings for $F_{\geq t}$ (by Definition 3.33). Furthermore, there exists a one-to-one mapping between labelings and complete extensions. We proved that there exists a valid coloring C for every $\mathcal{L}$ (see Lemma 3.34). Finally, we proved that different valid colorings C and C' do not represent any $\mathcal{L}$ where $\mathcal{L} \in l_t(C)$ and $\mathcal{L} \in l_t(C')$. For every X_t-restricted complete labeling we have a unique coloring.

Leaf Node:

Definition 3.36. *Let t be a* leaf node *of the tree decomposition and consider the colorings $X_t \rightarrow \{in_c, def_c, defp_c, out_c, outp_c\}$. If*

$$
\begin{aligned}
&C(x) = in_c && \Rightarrow y \in [C]_{d_c} \textit{ for all } y \rightarrowtail x \\
&C(x) = def_c && \Leftrightarrow [C]_{i_c} \rightarrowtail x \\
&C(x) = defp_c && \Rightarrow [C]_{i_c} \not\rightarrowtail x \\
&C(x) = out_c && \Leftrightarrow [C]_{i_c} \not\rightarrowtail x \textit{ and } x \not\rightarrowtail [C]_{i_c} \textit{ and } [C]_{o_c} \rightarrowtail x \\
&C(x) = outp_c && \Rightarrow [C]_{i_c} \not\rightarrowtail x \textit{ and } x \not\rightarrowtail [C]_{i_c} \textit{ and } [C]_{o_c} \not\rightarrowtail x
\end{aligned}
$$

holds for all $x \in X_t$, the coloring is a complete v-coloring for t.

Lemma 3.37. *For any* leaf node t *in a tree-decomposition of an AF, valid colorings and v-colorings for complete semantics coincide.*

Proof. For any leaf node t, $X_{>t} = \emptyset$. Hence, the $X_{>t}$-restricted complete labelings $\mathcal{L}$ for $F_{\geq t}$ correspond to the labelings that are conflict-free in $\mathcal{L}_{in}$ and satisfy $\mathcal{L}_{in} \not\rightarrowtail \mathcal{L}_{out}$ and $\mathcal{L}_{out} \not\rightarrowtail \mathcal{L}_{in}$.

$\Rightarrow$: Suppose that we have a valid coloring C in the leaf node t. We have to prove that C is also a v-coloring in t. As C is a valid coloring, by Definition 3.33 we know that there exists a labeling $\mathcal{L} \in l_t(C)$ where $\mathcal{L}_{in}$ of $\mathcal{L}$ is conflict-free, $\mathcal{L}_{in} \not\rightarrowtail \mathcal{L}_{out}$ and $\mathcal{L}_{out} \not\rightarrowtail \mathcal{L}_{in}$. For each $a \in X_t$ we can distinguish the following cases:

- Consider $C(a) = in_c$: Then, by Definition 3.33, $a \in \mathcal{L}_{in}$. Due to the conflict-freeness of $\mathcal{L}_{in}$, we have that an attacker $b \rightarrowtail a$ can not be colored with $\mathcal{L}_{in}$. Furthermore, as $\mathcal{L}_{in} \not\rightarrowtail \mathcal{L}_{out}$, we have that $C(b) \neq out_c$ and $C(b) \neq outp_c$. Hence, $C(b) \in \{def_c, defp_c\}$, which is equivalent to $b \in [C]_{d_c}$, must hold. This exactly corresponds to the first condition in Definition 3.36 of v-colorings.

- Now consider the case $C(a) = def_c$: By Definition 3.33 we have that this is equivalent to $a \in \mathcal{L}_{def}$ and $\mathcal{L}_{in} \rightarrowtail a$. This, on the other hand, means that there must exist an argument $b \in \mathcal{L}_{in}$ that attacks a. As $b \in \mathcal{L}_{in}$, it follows that $C(b) = in_c$. In total we have that $[C]_{i_c} \rightarrowtail a$ which is equivalent to $C(a) = def_c$, corresponding to the second condition of Definition 3.36 of v-colorings.

- In case $C(a) = defp_c$ we have that $a \in \mathcal{L}_{def}$ and $\mathcal{L}_{in} \not\rightarrowtail a$. It is easy to see that then there can not be any any argument $b \in \mathcal{L}_{in}$, hence $C(b) = in_c$, that attacks a. In total we have that $C(a) = defp_c \Rightarrow [C]_{i_c} \not\rightarrowtail a$, the third condition of v-colorings.

- Next, we analyze the case $C(a) = out_c$: By Definition 3.33, this is equivalent to $a \in \mathcal{L}_{out}$, $\mathcal{L}_{in} \not\rightarrowtail a$ (and therefore $[C]_{i_c} \not\rightarrowtail a$), $a \not\rightarrowtail \mathcal{L}_{in}$ (hence $a \not\rightarrowtail [C]_{i_c}$) and $\mathcal{L}_{out} \rightarrowtail a$. Hence, there must exist an argument $b \in \mathcal{L}_{out}$ where $b \rightarrowtail a$. By Definition 3.33 we have that an argument $b \in \mathcal{L}_{out}$ must be colored with either out_c or $outp_c$, in other words $[C]_{o_c} \rightarrowtail a$. Hence, condition four of v-colorings is satisfied.

- Finally, consider the case $C(a) = outp_c$: Then, we have that $a \in \mathcal{L}_{out}$, $\mathcal{L}_{in} \not\rightarrowtail a$, $a \not\rightarrowtail \mathcal{L}_{in}$ and $\mathcal{L}_{out} \not\rightarrowtail a$. Now, towards a contradiction, consider an argument b where $C(b) = out_c$ and $b \rightarrowtail a$. Then, by Definition 3.33 we have that $b \in \mathcal{L}_{out}$ and $\mathcal{L}_{out} \rightarrowtail a$. But this contradicts $\mathcal{L}_{out} \not\rightarrowtail a$.

$\Leftarrow$: Now, suppose that we have a v-coloring C for t. We have to prove that C is also a valid coloring for t. Hence, we claim that there exists a $\mathcal{L} = \langle \mathcal{L}_{in}, \mathcal{L}_{def}, \mathcal{L}_{out} \rangle$ where $\mathcal{L} \in l_t(C)$. Towards a contradiction, suppose that C is not a valid coloring. Then, (1) $\mathcal{L}_{in}$ is not conflict-free, (2) $\mathcal{L}_{in} \rightarrowtail \mathcal{L}_{out}$, (3) $\mathcal{L}_{out} \rightarrowtail \mathcal{L}_{in}$ or (4) one of the conditions of Definition 3.33 is not satisfied.

(1) Suppose there exists a conflict in $\mathcal{L}_{in}$. Then, there must exist two arguments a and b where $a \rightarrowtail b$ and $C(a) = C(b) = in_c$. In other words, $a, b \in \mathcal{L}_{in}$. But this contradicts the first condition of Definition 3.36 where $a \in \mathcal{L}_{def}$must hold, i.e. C would not be a v-coloring.

(2) Now, suppose that $\mathcal{L}_{in} \rightarrowtail \mathcal{L}_{out}$: Then there exist two arguments a and b where $a \in \mathcal{L}_{in}$ (or, in other words, $C(a) = in_c$), $b \in \mathcal{L}_{out}$ (or, in other words, $C(b) \in \{out_c, outp_c\}$) and $a \rightarrowtail b$. For leaf nodes, this corresponds to $a \in [C]_{i_c}$ and $b \in [C]_{o_c}$. We can distinguish two cases: First, let $C(b) = out_c$. Then, by Definition 3.36 of v-colorings we have that $[C]_{i_c} \not\rightarrowtail a$, a contradiction. Otherwise, if $C(b) = outp_c$, we also have that $[C]_{i_c} \not\rightarrowtail a$, again a contradiction to our assumption of C being a v-coloring.

(3) Suppose that $\mathcal{L}_{out} \rightarrowtail \mathcal{L}_{in}$: Then there exist two arguments a and b where $a \in \mathcal{L}_{out}$ (or, in other words, $C(a) \in \{outp_c, out_c\}$), $b \in \mathcal{L}_{in}$ (or, in other words, $C(b) = \mathcal{L}_{in}$) and

$a \rightarrowtail b$. For leaf nodes, this corresponds to $a \in [C]_{o_c}$ and $b \in [C]_{i_c}$. We can distinguish two cases: First, let $C(a) = out_c$. Then, by Definition 3.36 of v-colorings we have that $a \not\rightarrowtail [C]_{i_c}$, a contradiction. Otherwise, if $C(a) = outp_c$, we also have that $a \not\rightarrowtail [C]_{i_c}$, again a contradiction to our assumption of C being a v-coloring.

(4) Finally, we analyze the conditions for valid colorings of Definition 3.33. As an example, we outline the case $C(a) = def_c$. Then, by Definition 3.36, $[C]_{i_c} \rightarrowtail a$. As $[C]_{i_c} = \mathcal{L}_{in}$ for leaf nodes and as $a \in \mathcal{L}_{def}$, this exactly correspond to condition 2 of valid colorings. It is easy to see that a similar argument yields the equivalence of the other colors that can be assigned to a.

□

Introduction Node:

Definition 3.38. *Let t be an* introduction node *of a tree decomposition, t_1 be the child node of t and let a be the argument that is introduced in X_t. If C is a v-coloring for t_1 then $C + a$ is a v-coloring for t. Furthermore, if $[C]_{i_c} \not\rightarrowtail a$ and $a \not\rightarrowtail [C]_{i_c}$, then $C \dot{+} a$ is a v-coloring for t. Finally, if $a \not\rightarrowtail a$, $[C]_{i_c} \not\rightarrowtail a$, $a \not\rightarrowtail [C]_{i_c}$, $[C]_{o_c} \not\rightarrowtail a$, $a \not\rightarrowtail [C]_{o_c}$, then $C \ddot{+} a$ is a v-coloring for t as well. For $C : A \to \{in_c, def_c, defp_c, out_c, outp_c\}$ we define $C + a$, $C \dot{+} a$ and $C \ddot{+} a$ as follows:*

$$(C + a)(b) = \begin{cases} C(b) & \text{if } b \in A \\ def_c & \text{if } b = a \text{ and } [C]_{i_c} \rightarrowtail a \\ defp_c & \text{if } b = a \text{ and } [C]_{i_c} \not\rightarrowtail a \end{cases}$$

$$(C \dot{+} a)(b) = \begin{cases} C(b) & \text{if } b \in A \text{ and } C(b) \in \{in_c, def_c, defp_c, out_c\} \\ out_c & \text{if } (b = a \text{ and } [C]_{o_c} \rightarrowtail a) \text{ or } (a \neq b \text{ and } a \rightarrowtail b \text{ and } C(b) = outp_c) \\ outp_c & \text{otherwise} \end{cases}$$

$$(C \ddot{+} a)(b) = \begin{cases} in_c & \text{if } b = a \\ C(b) & \text{if } b \in A \text{ and } C(b) \in \{in_c, def_c, out_c, outp_c\} \\ def_c & \text{if } b \neq a \text{ and } (a, b) \in F_t \text{ and } C(b) = defp_c \\ defp_c & \text{if } b \neq a \text{ and } (a, b) \notin F_t \text{ and } C(b) = defp_c \end{cases}$$

Lemma 3.39. *For any* introduction node *t in a tree-decomposition of an AF, valid colorings and v-colorings for complete semantics coincide if they coincide in the child node t_1 of t.*

Proof. Let t be an introduction node of a tree decomposition and let t_1 be the child node of t. Furthermore, let a be the argument that is introduced in t, i.e. $X_t = X_{t_1} \cup \{a\}$ where $a \notin X_{t_1}$. By the properties of tree decompositions (see Definition 2.18) we know that $a \notin F_{>t} = F_{>t_1}$. Furthermore, $X_{>t} = X_{>t_1}$ and there is no attack relation between a and $X_{>t}$.

$\Rightarrow$: Suppose that we have a valid coloring C for t. Then there exists a labeling $\mathcal{L} = \langle \mathcal{L}_{in}, \mathcal{L}_{def}, \mathcal{L}_{out} \rangle \in l_t(C)$ that is an $X_{>t}$-restricted complete labeling for $F_{\geq t}$. Furthermore, as $a \notin X_{t_1}$, $\mathcal{L} \backslash \{a\} = \langle \mathcal{L}_{in} \backslash \{a\}, \mathcal{L}_{def} \backslash \{a\}, \mathcal{L}_{out} \backslash \{a\} \rangle$ is an $X_{>t_1}$ restricted complete labeling

for $F_{\geq t_1}$. Based on $\mathcal{L}$ we now define a coloring C_1 for every argument $b \in X_{t_1}$ that fulfills the properties for valid colorings as defined in Definition 3.33:

$$\begin{array}{ll}
\text{if } b \in \mathcal{L}_{in}\backslash\{a\}: & C_1(b) = in_c \\
\text{if } b \in \mathcal{L}_{def}\backslash\{a\} \text{ and } \mathcal{L}_{in}\backslash\{a\} \rightarrowtail b: & C_1(b) = def_c \\
\text{if } b \in \mathcal{L}_{def}\backslash\{a\} \text{ and } \mathcal{L}_{in}\backslash\{a\} \not\rightarrowtail b: & C_1(b) = defp_c \\
\text{if } b \in \mathcal{L}_{out}\backslash\{a\} \text{ and } \mathcal{L}_{in}\backslash\{a\} \not\rightarrowtail b \text{ and } \mathcal{L}_{out}\backslash\{a\} \rightarrowtail b: & C_1(b) = out_c \\
\text{if } b \in \mathcal{L}_{out}\backslash\{a\} \text{ and } \mathcal{L}_{in}\backslash\{a\} \not\rightarrowtail b \text{ and } \mathcal{L}_{out}\backslash\{a\} \not\rightarrowtail b: & C_1(b) = outp_c
\end{array}$$

As the definition of C_1 exactly corresponds to Definition 3.33 of valid colorings we have that, by assumption, C_1 is also a v-coloring for t_1. It remains to prove the equivalence of C (being a valid coloring by assumption) and C^* which is a v-coloring by construction of $C^* = C_1 + a$, $C^* = C_1 \dot{+} a$ (in case of $[C]_{i_c} \not\rightarrowtail a$ and $a \not\rightarrowtail [C]_{i_c}$) and $C_1 \ddot{+} a$ (in case $a \not\rightarrowtail a$, $[C]_{i_c} \not\rightarrowtail a$, $a \not\rightarrowtail [C]_{i_c}$, $[C]_{o_c} \not\rightarrowtail a$ and $a \not\rightarrowtail [C]_{o_c}$ hold). We can distinguish three cases:

- Consider the case $C(a) = in_c$. By Definition 3.33 we have that $a \in \mathcal{L}_{in}$. Furthermore, as C is a valid coloring and by Definition 3.31, we have that $\mathcal{L}_{in}$ is conflict-free. Hence, $a \not\rightarrowtail a$, $\mathcal{L}_{in} \not\rightarrowtail a$ and $a \not\rightarrowtail \mathcal{L}_{in}$. As $[C]_{i_c} \subseteq \mathcal{L}_{in}$, we have that $[C]_{i_c} \not\rightarrowtail a$ and $a \not\rightarrowtail [C]_{i_c}$. Furthermore, $\mathcal{L}_{in} \not\rightarrowtail \mathcal{L}_{out}$ and $\mathcal{L}_{out} \not\rightarrowtail \mathcal{L}_{in}$ and as $[C]_{o_c} \subseteq \mathcal{L}_{out}$ therefore $a \not\rightarrowtail [C]_{o_c}$ resp. $[C]_{o_c} \not\rightarrowtail a$. Hence, the conditions for $C^* = C_1 \ddot{+} a$ being a v-coloring are satisfied. For $C^*(a) = (C_1 \ddot{+} a)(a)$ we have $C^*(a) = in_c = C(a)$. It remains to analyze the colors for the arguments $b \in X_t \backslash \{a\}$: For $C(b) \in \{in_c, def_c, out_c, outp_c\}$ we have that $C^*(b)$ maps to the same color. Now, consider the case where $C(b) = def_c$. Then, $\mathcal{L}_{in} \rightarrowtail b$. We can distinguish two cases: Either, $C_1(b) = def_c$, i.e. there exists an attack relation in $X_{\geq t_1}$. Then, also $C^*(b) = def_c$. In the other case, $C_1(b) = defp_c$. Then, there exists an attack $a \rightarrowtail b$, i.e. $(a, b) \in F_t$. But then, $C^*(b) = def_c$. Finally, for $C(b) = defp_c$, we have $\mathcal{L}_{in} \not\rightarrowtail b$. But then, $a \not\rightarrowtail b$ in F_t. Hence, $C^*(b) = defp_c$.

- Now, consider the case $C(a) \in \{def_c, defp_c\}$: By assumption, C_1 is conflict-free and as $a \notin \mathcal{L}_{in}$, also $[C_1 + a]_{i_c}$ is conflict-free. Furthermore, as $a \notin \mathcal{L}_{out}$, and as C_1 is a valid coloring, we also have that $[C_1 + a]_{i_c} \not\rightarrowtail [C_1 + a]_{o_c}$ and $[C_1 + a]_{o_c} \not\rightarrowtail [C_1 + a]_{i_c}$. It remains to prove the equivalence of $C = C^*$, where $C^* = C_1 + a$. Consider the case $C(a) = def_c$. Then, $a \in \mathcal{L}_{def}$ and $\mathcal{L}_{in} \rightarrowtail a$. But then, due to the properties of tree decompositions, also $[C_1]_{i_c} \rightarrowtail a$ must hold, i.e. $C^*(a) = (C_1 + a)(a) = def_c$. It is easy to see that also $C(a) = C^*(a)$ for $C(a) = defp_c$ holds. The colors of the other arguments remain the same within $C_1 + a$, hence, the equivalence of the other colors is trivial.

- Finally, consider the case $C(a) \in \{out_c, outp_c\}$: By assumption, C_1 is conflict-free and as $a \notin \mathcal{L}_{in}$, also $[C_1 \dot{+} a]_{i_c}$ is conflict-free. Furthermore we have that $a \in \mathcal{L}_{out}$. By assumption, C is a valid coloring and hence we have that $\mathcal{L}_{in} \not\rightarrowtail \mathcal{L}_{out}$ and $\mathcal{L}_{out} \not\rightarrowtail \mathcal{L}_{in}$. As $[C]_{o_c} \subseteq \mathcal{L}_{out}$ it follows that $[C]_{i_c} \not\rightarrowtail a$ and $a \not\rightarrowtail [C]_{i_c}$. Now, we outline the the case $C(a) = out_c$. Then, $\mathcal{L}_{out} \rightarrowtail a$. This corresponds to $[C_1]_{o_c} \rightarrowtail a$ and hence $C^*(a) = (C_1 \dot{+} a)(a) = out_c$.

$\Leftarrow$: Suppose that we have a v-coloring C for t, i.e. there exists a v-coloring for C_1 for the child node t_1 such that $C = C_1 + a$, $C = C_1 \dot{+} a$ or $C = C_1 \ddot{+} a$. As C_1 is a valid coloring there

exists an $X_{>t_1}$-restricted complete labeling $\mathcal{L} \in l_{t_1}(C_1)$ for $F_{\geq t_1}$. Furthermore, as $X_{>t_1} = X_{>t}$, $\mathcal{L}$ is also an $X_{>t}$-restricted complete labeling for $F_{\geq t}$.

We want to prove that C is a valid coloring for t.

- For $C_1 + a$ the colors of each argument $b \neq a$ remain the same. As the introduced argument a is either colored with def_c or $defp_c$, we have that $\mathcal{L}_{in}$ remains conflict-free, $\mathcal{L}_{in} \not\mapsto \mathcal{L}_{out}$ and $\mathcal{L}_{out} \not\mapsto \mathcal{L}_{in}$. In case $[C_1]_{i_c} \rightarrowtail a$ we have that b is colored with def_c. This corresponds to condition 2 of Definition 3.33 where $\mathcal{L}_{in} \rightarrowtail a$. Furthermore, if $[C_1]_{i_c} \not\mapsto a$ we have that $(C_1 + a)(b) = defp_c = C(b)$. Hence, $\langle \mathcal{L}_{in}, \mathcal{L}_{def} \cup \{a\}, \mathcal{L}_{out} \rangle$ must be a labeling for $l_t(C_1 + a)$ and we have that $C_1 + a$ is a valid coloring.

- For $C_1 \dot{+} a$, if $C_1(b) \in \{in_c, def_c, defp_c, out_c\}$, the colors remain the same for $C \dot{+} a$. Therefore, $\mathcal{L}_{in}$ remains the same and hence, remains conflict-free. Furthermore, by Definition 3.38, we have that $[C_1]_{i_c} \not\mapsto a$ and $a \not\mapsto [C_1]_{i_c}$. As C_1 is a valid coloring and every argument that is colored with out_c or $outp_c$ in C_1 is also colored with out_c or $outp_c$ in $C_1 \dot{+} a$ we have that $\mathcal{L}_{out} \cup \{a\} \not\mapsto \mathcal{L}_{in}$ and $\mathcal{L}_{in} \not\mapsto \mathcal{L}_{out} \cup \{a\}$. It is easy to check that $\langle \mathcal{L}_{in}, \mathcal{L}_{def}, \mathcal{L}_{out} \cup \{a\} \rangle$ is a labeling for $C_1 \dot{+} a$ and we have that $C_1 \dot{+} a$ is a valid coloring.

- For $C_1 \ddot{+} a$ we have that $a \not\mapsto a$, $[C]_{i_c} \not\mapsto a$ and $a \not\mapsto [C]_{i_c}$. Hence, $[C_1]_{i_c} \cup \{a\}$ is conflict-free. Furthermore, due to the connectedness condition of tree decompositions it is easy to see that also $\mathcal{L}_{in} \cup \{a\}$ is conflict-free. Furthermore, as $[C]_{o_c} \not\mapsto a$, $a \not\mapsto [C]_{o_c}$ and due to the fact that $C_1 \ddot{+} a$ does not change the colors of arguments where $C_1(b) \in \{out_c, outp_c\}$, it is obvious that $\mathcal{L}_{in} \not\mapsto \mathcal{L}_{out}$ and $\mathcal{L}_{out} \not\mapsto \mathcal{L}_{in}$ and hence$\langle \mathcal{L}_{in} \cup \{a\}, \mathcal{L}_{def}, \mathcal{L}_{out} \rangle$ is a labeling for $l_t(C_1 \ddot{+} a)$.

Finally, we obtain that valid colorings and v-colorings for complete semantics of a node t coincide if they coincide in the child node t_1. □

Removal Node:

Definition 3.40. *Let t be a* removal node *of a tree decomposition, t_1 be the child node of t and let a be the argument that is removed in X_t. If C is a v-coloring for t_1 and $C(a) \notin \{defp_c, outp_c\}$ then $C - a$ is a v-coloring for t. For $C : A \to \{in_c, def_c, defp_c, out_c, outp_c\}$ we define $C - a$ as follows:*

$$(C - a)(b) = C(b) \text{ for each } b \in A \setminus \{a\}$$

Lemma 3.41. *For any* removal node *t in a tree-decomposition of an AF, valid colorings and v-colorings for complete semantics coincide if they coincide in the child node t_1 of t.*

Proof. As t is a removal node there exists exactly one argument a that is removed in t. Hence, $X_t \cup \{a\} = X_{t_1}$. As a remains in $X_{\geq t}$, we have that $X_{\geq t} = X_{\geq t_1}$ and $F_{\geq t} = F_{\geq t_1}$.

$\Rightarrow$: Suppose that there exists a valid coloring C for t. We show that there exists a valid coloring C_1 for t_1 with $C_1(a) \notin \{defp_c, outp_c\}$ and $C = C_1 - a$. We define C_1 as follows:

- For each argument $b \in X_{t_1} \setminus \{a\}$ we set $C_1(b) = C(b)$. Hence, $C = C_1 - a$ is always satisfied.

It remains to define the color for $C_1(a)$. For this, consider a labeling $\mathcal{L} \in l_t(C)$:

- Suppose that $a \in \mathcal{L}_{in}$: Then we set $C_1(a) = in_c$. By assumption, $\mathcal{L}$ is an $X_{>t}$-restricted complete labeling for $F_{\geq t}$. As $X_{>t} = X_{>t_1} \cup \{a\}$ we have that $\mathcal{L}$ is also an $X_{>t_1}$-restricted complete labeling for $F_{\geq t_1}$. Hence, $\mathcal{L} \in l_{t_1}(C_1)$. Furthermore, by assumption, C_1 is a v-coloring for t_1 and therefore, $C = C_1 - a$ is a v-coloring for t, by definition.

- Suppose that $a \notin \mathcal{L}_{in}$: If $\mathcal{L}_{in} \rightarrowtail a$ we set $C_1(a) = def_c$. In this case, $\mathcal{L} \in l_{t_1}(C_1)$. As $\mathcal{L}$ is an $X_{>t}$ restricted complete labeling for $F_{\geq t}$ it can never occur that we set $C_1(a) = defp_c$ in case $\mathcal{L}_{in} \not\rightarrowtail a$. If $\mathcal{L}_{in} \not\rightarrowtail a$, $a \not\rightarrowtail \mathcal{L}_{in}$ and $\mathcal{L}_{out} \rightarrowtail a$ we set $C_1(a) = out_c$. Again, $\mathcal{L} \in l_{t_1}(C_1)$. As $\mathcal{L}$ is an $X_{>t}$ restricted complete labeling for $F_{\geq t}$ it can never happen that $\mathcal{L}_{out} \rightarrowtail a$, hence $C_1(a)$ is never set to $outp_c$. As C_1 is a valid coloring and it is also a v-coloring for t_1 we have that $C = C - a$ is a v-coloring for t.

$\Leftarrow$: Suppose that there exists a v-coloring C for t. Furthermore, by assumption, there exists a v-coloring C_1 for t_1 such that $C = C_1 - a$ and $C(a) \notin \{defp_c, outp_c\}$. As $C(a) \notin \{defp_c, outp_c\}$ there exists a labeling $\mathcal{L}$ such that (1) $a \in \mathcal{L}_{in}$, (2) $a \in \mathcal{L}_{def}$ and $\mathcal{L}_{in} \rightarrowtail a$ or (3) $a \in \mathcal{L}_{out}$, $\mathcal{L}_{in} \not\rightarrowtail a$, $a \not\rightarrowtail \mathcal{L}_{in}$ and $\mathcal{L}_{out} \rightarrowtail a$. This corresponds to Definition 3.31 of restricted labelings. As $X_{t_1} \cup \{a\} = X_t$, $\mathcal{L}$ is an $X_{>t}$-restricted complete labeling for $F_{\geq t}$. Then, as $\mathcal{L} \in l_t(C)$, i.e. $\mathcal{L}$ is a valid coloring for t. □

Branch Node:

Definition 3.42. *Let t be* branch node *of a tree decomposition, and let C be the v-coloring for the child node t_1 and D be the v-coloring for the child node t_2. If $[C]_{i_c} = [D]_{i_c}$, $[C]_{d_c} = [D]_{d_c}$ and $[C]_{o_c} = [D]_{o_c}$, then $C \bowtie D$ is a v-coloring for t. For $C, D : A \to \{in_c, def_c, defp_c, out_c, outp_c\}$ we define $C \bowtie D$ as follows:*

$$(C \bowtie D)(b) = \begin{cases} in_c & \text{if } C(b) = D(b) = in_c \\ def_c & \text{if } C(b) = def_c \text{ or } D(b) = def_c \\ defp_c & \text{if } C(b) = D(b) = defp_c \\ out_c & \text{if } C(b) = out_c \text{ or } D(b) = out_c \\ outp_c & \text{if } C(b) = D(b) = outp_c \end{cases}$$

Lemma 3.43. *For any* branch node t *in a tree-decomposition of an AF, valid colorings and v-colorings for complete semantics coincide if they coincide in the child nodes t_1 and t_2 of t.*

Proof. Let t be a branch node of a normalized tree decomposition and t_1 and t_2 be the child nodes of t. Then, we have that $X_t = X_{t_1} = X_{t_2}$, $X_t = X_{\geq t_1} \cap X_{\geq t_2}$ and $X_{\geq t} = X_{\geq t_1} \cup X_{\geq t_2}$ (see Definiton 2.18). As in the proof for stable semantics we can then partition $X_{\geq t}$ into three disjoint sets X_t, $X_{>t_1}$ and $X_{>t_2}$. We can then define a labeling $\mathcal{L} \subseteq X_{\geq t}$ where $\mathcal{L} \subseteq X_{\geq t} \Leftrightarrow \mathcal{L}_{in}, \mathcal{L}_{def}, \mathcal{L}_{out} \subseteq X_{\geq t}$ and $\mathcal{L}_{in} \cap \mathcal{L}_{def} \cap \mathcal{L}_{out} = \emptyset$.

Additionally, we define $\mathcal{L} = \mathcal{L}^{'} \cup \mathcal{L}^{''}$ as $\mathcal{L}_{in} = \mathcal{L}^{'}_{in} \cup \mathcal{L}^{''}_{in}$, $\mathcal{L}_{def} = \mathcal{L}^{'}_{def} \cup \mathcal{L}^{''}_{def}$ and $\mathcal{L}_{out} = \mathcal{L}^{'}_{out} \cup \mathcal{L}^{''}_{out}$.

Furthermore, $\mathcal{L}^{'} \subseteq X_{\geq t_1}$ and $\mathcal{L}^{''} \subseteq X_{\geq t_2}$ where $\mathcal{L}^{'} \cap X_t = \mathcal{L}^{''} \cap X_t$ where $\mathcal{L}^{'} \cap X_t = \langle \mathcal{L}^{'}_{in} \cap X_t, \mathcal{L}^{'}_{def} \cap X_t, \mathcal{L}^{'}_{out} \cap X_t \rangle$.

Now we prove that $\mathcal{L}$ is an $X_{>t}$-restricted complete labeling for $F_{\geq t}$ iff $\mathcal{L}^{'}$ is an $X_{>t_1}$-restricted complete labeling for the sub-framework $F_{\geq t_1}$, $\mathcal{L}^{''}$ is an $X_{>t_2}$-restricted complete labeling for the sub-framework $F_{\geq t_2}$, $\mathcal{L}^{'}_{in} \cap X_t = \mathcal{L}^{''}_{in} \cap X_t$, $\mathcal{L}^{'}_{def} \cap X_t = \mathcal{L}^{''}_{def} \cap X_t$ and $\mathcal{L}^{'}_{out} \cap X_t = \mathcal{L}^{''}_{out} \cap X_t$.

Lemma 3.44. *Let $\mathcal{L}^{'}_{in}, \mathcal{L}^{'}_{def}, \mathcal{L}^{'}_{out} \subseteq X_{\geq t_1}$ and $\mathcal{L}^{''}_{in}, \mathcal{L}^{''}_{def}, \mathcal{L}^{''}_{out} \subseteq X_{\geq t_2}$, such that*

- *$\mathcal{L}^{'}$ is an $X_{>t_1}$-restricted complete labeling for $F_{\geq t_1}$,*
- *$\mathcal{L}^{''}$ is an $X_{>t_2}$-restricted complete labeling for $F_{\geq t_2}$,*
- *$\mathcal{L}^{'}_{in} \cap X_t = \mathcal{L}^{''}_{in} \cap X_t$,*
- *$\mathcal{L}^{'}_{def} \cap X_t = \mathcal{L}^{''}_{def} \cap X_t$,*
- *$\mathcal{L}^{'}_{out} \cap X_t = \mathcal{L}^{''}_{out} \cap X_t$.*

Then, $\mathcal{L} = \mathcal{L}^{'} \cup \mathcal{L}^{''}$ is an $X_{>t}$-restricted complete labeling for $F_{\geq t}$.

Proof. $\mathcal{L}$ is an $X_{>t}$-restricted complete labeling for $F_{\geq t}$ if $\mathcal{L}_{in}$ is (1) conflict-free in $F_{\geq t}$, (2) $\mathcal{L}_{in} \not\rightarrowtail \mathcal{L}_{out}$, (3) $\mathcal{L}_{out} \not\rightarrowtail \mathcal{L}_{in}$ and (4) for each argument $a \in X_{>t}$ (i) $a \in \mathcal{L}_{in} \Leftrightarrow \{b \mid (b, a) \in R\} \subseteq \mathcal{L}_{def}$, (ii) $a \in \mathcal{L}_{def} \Leftrightarrow \mathcal{L}_{in} \rightarrowtail a$ and (iii) $a \in \mathcal{L}_{out} \Leftrightarrow \mathcal{L}_{in} \not\rightarrowtail a \wedge \mathcal{L}_{out} \rightarrowtail a$.

(1) Suppose that there exists a conflict where $a, b \in \mathcal{L}_{in}$ and $a \rightarrowtail b$. As $\mathcal{L}_{in} = \mathcal{L}^{'}_{in} \cup \mathcal{L}^{''}_{in}$ we can distinguish two cases:

   (a) Consider the case where $a, b \in X_{\geq t_1}$. Then we have that $a, b \in \mathcal{L}^{'}_{in}$. But then, $F_{\geq t_1}$ is not conflict-free and we have a contradiction to the first assumption of Lemma 3.44. By symmetry, the same argument holds for the case $a, b \in X_{\geq t_2}$ where we have a conflict in $F_{\geq t_2}$, contradicting assumption 2 of Lemma 3.44.

   (b) Now, consider the case where $a \in X_{\geq t_1}$, $b \in X_{\geq t_2}$ and there exists an attack $a \rightarrowtail b$. Due to the properties of tree decompositions, a and b have to appear together in at least one bag. Furthermore, due to the connectedness condition we have that a or b must appear in the bag X_t. But as $\{a\} \subseteq \mathcal{L}^{'}_{in}$ and $\{b\} \subseteq \mathcal{L}^{''}_{in}$ and by the assumption that $\mathcal{L}^{'}_{in} \cap X_t = \mathcal{L}^{''}_{in} \cap X_t$ we have that $\{a, b\} \subseteq \mathcal{L}^{'}_{in}$ (or $\{a, b\} \subseteq \mathcal{L}^{''}_{in}$) and hence, a conflict in $\mathcal{L}^{'}_{in}$ or $\mathcal{L}^{''}_{in}$ which contradicts assumption 1 and 2 of Lemma 3.44. It is obvious that the case where $b \in X_{\geq t_1}$, $a \in X_{\geq t_2}$ also results in a contradiction of our assumptions.

(2) Suppose that there exists an attack between two arguments a and b where $a \in \mathcal{L}_{in}$ and $b \in \mathcal{L}_{out}$. We have that $\mathcal{L}_{in} = \mathcal{L}^{'}_{in} \cup \mathcal{L}^{''}_{in}$ and $\mathcal{L}_{out} = \mathcal{L}^{'}_{out} \cup \mathcal{L}^{''}_{out}$. Hence, we can distinguish the following cases:

(a) Consider the case where $a \in \mathcal{L}_{in}^{'}$ and $b \in \mathcal{L}_{out}^{'}$ where $a \rightarrowtail b$. Then, $\mathcal{L}_{in}^{'} \rightarrowtail \mathcal{L}_{out}^{'}$ in the sub-framework $F_{\geq t_1}$, i.e. we have a contradiction to assumption 1. Furthermore, in the case where $a \in \mathcal{L}_{in}^{''}$ and $b \in \mathcal{L}_{out}^{''}$ we again have a contradiction (to assumption 2).

(b) Now consider $a \in \mathcal{L}_{in}^{'}$ and $b \in \mathcal{L}_{out}^{''}$ where $a \rightarrowtail b$. Hence, we know that $a \in X_{\geq t_1}$ and $b \in X_{\geq t_2}$. Furthermore, due to the properties of tree decompositions, $\{a, b\} \subseteq X_t$ must hold. As $\{a\} \subseteq \mathcal{L}_{in}^{'}$ and assuming that $\mathcal{L}_{in}^{'} \cap X_t = \mathcal{L}_{in}^{''} \cap X_t$ we have that $\{a\} \subseteq \mathcal{L}_{in}^{''}$. But as also $b \in \mathcal{L}_{out}^{''}$, we have that $\mathcal{L}_{in}^{''} \rightarrowtail \mathcal{L}_{out}^{''}$ which contradicts assumption (2) of Lemma 3.44. The case where $a \in \mathcal{L}_{in}^{''}$ and $b \in \mathcal{L}_{out}^{'}$ follows by symmetry.

(3) As the case $\mathcal{L}_{out} \not\rightarrowtail \mathcal{L}_{in}$ follows the same line of argument as (2) we do not work it out in detail here.

(4) Finally, we have to prove that for each argument $a \in X_{>t}$ (i) $a \in \mathcal{L}_{in} \Leftrightarrow \{b \mid (b, a) \in R\} \subseteq \mathcal{L}_{def}$, (ii) $a \in \mathcal{L}_{def} \Leftrightarrow \mathcal{L}_{in} \rightarrowtail a$ and (iii) $a \in \mathcal{L}_{out} \Leftrightarrow \mathcal{L}_{in} \not\rightarrowtail a \wedge \mathcal{L}_{out} \rightarrowtail a$ holds.

(i) For each $a \in \mathcal{L}_{in}^{'}$ we have that $\{b \mid (b, a) \in R\} \subseteq \mathcal{L}_{def}^{'}$. In other words, for each argument b that attacks a in $X_{>t_1}$ we know that $b \in \mathcal{L}_{def}^{'}$. The same holds for arguments in $X_{>t_2}$ that are in $\mathcal{L}_{in}^{''}$. Due to the connectedness condition of tree decompositions we know that there is no $c \in X_{>t_2}$ where $c \rightarrowtail a$ (and vice-versa). Hence, as $\mathcal{L}_{in}^{'} \cup \mathcal{L}_{in}^{''} = \mathcal{L}_{in}$, $\mathcal{L}_{def}^{'} \cup \mathcal{L}_{def}^{''} = \mathcal{L}_{def}$ and $X_{>t_1} \cup X_{>t_2} = X_{>t}$ but $X_{>t_1} \cap X_{>t_2} = \emptyset$ we have that each argument in $X_{>t}$ that is in $\mathcal{L}_{in}$ is only attacked by arguments of $\mathcal{L}_{def}$ in $F_{\geq t}$. Hence, condition (i) is satisfied.

(ii) Now, consider an argument $a \in \mathcal{L}_{def}^{'}$ for $X_{>t_1}$ in $F_{\geq t_1}$. Then, we know that also $\mathcal{L}_{in}^{'} \rightarrowtail a$ holds. Furthermore, each for each argument $b \in \mathcal{L}_{def}^{''}$ for $X_{>t_2}$ in $F_{\geq t_2}$ we know that also $\mathcal{L}_{in}^{''} \rightarrowtail b$ holds. As $\mathcal{L}_{in}^{'} \cup \mathcal{L}_{in}^{''} = \mathcal{L}_{in}$, $\mathcal{L}_{def}^{'} \cup \mathcal{L}_{def}^{''} = \mathcal{L}_{def}$ and $X_{>t_1} \cup X_{>t_2} = X_{>t}$ we have that each argument in $\mathcal{L}_{def}$ is attacked by $\mathcal{L}_{in}$ (and vice-versa) in $X_{>t}$ for $F_{\geq t}$. Hence, condition (ii) is satisfied.

(ii) Finally, consider an argument $a \in \mathcal{L}_{out}^{'}$ for $X_{>t_1}$ in $F_{\geq t_1}$. Then, we know that $\mathcal{L}_{in}^{'} \not\rightarrowtail a$. Furthermore, due to the connectedness condition, we know that no argument $b \in \mathcal{L}_{in}^{''}$ attacks a. Hence, $\mathcal{L}_{in}^{'} \cup \mathcal{L}_{in}^{''} = \mathcal{L}_{in}$ does not attack a. Furthermore, as $\mathcal{L}_{out}^{'} \rightarrowtail a$, we know that also $\mathcal{L}_{out}^{'} \cup \mathcal{L}_{out}^{''} = \mathcal{L}_{out}$ attacks a. For arguments in $\mathcal{L}_{out}^{''}$ the same follows by symmetry. Hence, (iii) is satisfied.

Hence, based on our assumptions, we have that $\mathcal{L} = \mathcal{L}^{'} \cup \mathcal{L}^{''}$ is an $X_{>t}$-restricted complete labeling for $F_{\geq t}$. □

Lemma 3.45. *Let $\mathcal{L} = \mathcal{L}^{'} \cup \mathcal{L}^{''}$ be an $X_{>t}$-restricted complete labeling for $F_{\geq t}$ with $\mathcal{L}_{in}^{'} = \mathcal{L}_{in} \cap X_{\geq t_1}$, $\mathcal{L}_{in}^{''} = \mathcal{L}_{in} \cap X_{\geq t_2}$, $\mathcal{L}_{def}^{'} = \mathcal{L}_{def} \cap X_{\geq t_1}$, $\mathcal{L}_{def}^{''} = \mathcal{L}_{def} \cap X_{\geq t_2}$, $\mathcal{L}_{out}^{'} = \mathcal{L}_{out} \cap X_{\geq t_1}$ and $\mathcal{L}_{out}^{''} = \mathcal{L}_{out} \cap X_{\geq t_2}$. Then*

(1) $\mathcal{L}^{'}$ is an $X_{>t_1}$-restricted complete labeling for $F_{\geq t_1}$,

(2) $\mathcal{L}^{''}$ is an $X_{>t_2}$-restricted complete labeling for $F_{\geq t_2}$,

(3) $\mathcal{L}_{in}^{'} \cap X_t = \mathcal{L}_{in}^{''} \cap X_t$,

(4) $\mathcal{L}^{'}_{def} \cap X_t = \mathcal{L}^{''}_{def} \cap X_t$,

(5) $\mathcal{L}^{'}_{out} \cap X_t = \mathcal{L}^{''}_{out} \cap X_t$.

Proof. Let $\mathcal{L} = \mathcal{L}^{'} \cup \mathcal{L}^{''}$ be an $X_{>t}$-restricted complete labeling for $F_{\geq t}$.

(1,2) As $\mathcal{L}_{in}$ is conflict-free in $F_{\geq t}$ and $\mathcal{L}^{'}_{in} = \mathcal{L}_{in} \cap X_{\geq t_1}$, we have that $\mathcal{L}^{'}_{in}$ must be conflict-free in $F_{\geq t_1}$. Furthermore, as $\mathcal{L}_{in} \not\rightarrowtail \mathcal{L}_{out}$ and $\mathcal{L}^{'}_{out} = \mathcal{L}_{out} \cap X_{\geq t_1}$ we have that also $\mathcal{L}^{'}_{in} \not\rightarrowtail \mathcal{L}^{'}_{out}$ must hold. Furthermore, $\mathcal{L}^{'}_{out} \not\rightarrowtail \mathcal{L}^{'}_{in}$ must hold. By the same line of argument these properties are also satisfied in $\mathcal{L}^{''}$.

It remains to show that for an argument $a \in X_{>t_1}$ (i) $a \in \mathcal{L}^{'}_{in} \Leftrightarrow \{b \mid (b,a) \in R\} \subseteq \mathcal{L}^{'}_{def}$, (ii) $a \in \mathcal{L}^{'}_{def} \Leftrightarrow \mathcal{L}^{'}_{in} \rightarrowtail a$ and (iii) $a \in \mathcal{L}^{'}_{out} \Leftrightarrow \mathcal{L}^{'}_{in} \not\rightarrowtail a \wedge \mathcal{L}^{'}_{out} \rightarrowtail a$ are satisfied. Then, by symmetry, this also holds for arguments in $X_{>t_2}$ and $\mathcal{L}^{''}$.

(i) Towards a contradiction, suppose that there exists an argument $a \in \mathcal{L}^{'}_{in}$ and an argument $b \notin \mathcal{L}^{'}_{def}$ where $b \rightarrowtail a$ in $X_{>t_1}$. Due to the properties of tree decompositions there can not be any argument $x \in X_{t_1}$ (or anywhere else above in the tree decomposition) where $x \rightarrowtail b$ and hence, $b \notin \mathcal{L}_{def}$. But, due to the construction of $\mathcal{L}_{in}$, we have that $a \in \mathcal{L}_{in}$. Then, we have that $a \in \mathcal{L}_{in}$ but $b \notin \mathcal{L}_{def}$ with $b \rightarrowtail a$, a contradiction.

Now, suppose that $a \notin \mathcal{L}^{'}_{in}$ and $\{b \mid (b,a) \in R\} \subseteq \mathcal{L}^{'}_{def}$, i.e. there exists an argument $a \notin \mathcal{L}^{'}_{in}$ that is only attacked by arguments in $\mathcal{L}^{'}_{def}$ for $X_{>t_1}$. As there can not be an argument $x \in X_t$ where $x \rightarrowtail a$ and since $\mathcal{L}^{'}_{in} \subseteq \mathcal{L}_{in}$ we have that $a \notin \mathcal{L}_{in}$ but all arguments that attack a are in $\mathcal{L}_{def}$. This contradicts our assumption of $\mathcal{L}$ being an $X_{>t}$-restricted labeling for $F_{\geq t}$.

(ii) Towards a contradiction, suppose that there exists an argument $a \in \mathcal{L}^{'}_{def}$ and there is no argument $b \in \mathcal{L}^{'}_{in}$ such that $b \rightarrowtail a$ in $X_{>t_1}$. As there can not be an attack between arguments in $X_{>t_2}$ and a there can not be any argument $x \in \mathcal{L}^{''}_{in}$ where $x \rightarrowtail a$. But then, also $\mathcal{L}^{'}_{in} \cup \mathcal{L}^{''}_{in} = \mathcal{L}_{in}$ does not contain any argument that attacks $a \in \mathcal{L}^{'}_{def} \cup \mathcal{L}^{''}_{def} = \mathcal{L}_{def}$.

Now, suppose that $a \notin \mathcal{L}^{'}_{def}$ and there is an argument $b \in \mathcal{L}^{'}_{in}$ such that $b \rightarrowtail a$ in $X_{>t_1}$. But then, due to the construction of $\mathcal{L}_{in}$, $b \in \mathcal{L}_{in}$ and $b \rightarrowtail a$ in $X_{>t}$. But then, $a \in \mathcal{L}_{def} = \mathcal{L}^{'}_{def} \cup \mathcal{L}^{''}_{def}$. Due to the properties of tree decompositions, $a \notin \mathcal{L}^{''}_{def}$. Hence, $a \in \mathcal{L}^{'}_{def}$, i.e. we have a contradiction.

(iii) Towards a contradiction, suppose that $a \in \mathcal{L}^{'}_{out}$, $\mathcal{L}^{'}_{in} \not\rightarrowtail a$ and $\mathcal{L}^{'}_{out} \not\rightarrowtail a$. Following the same approach as above (taking the connectedness condition and the condition of two arguments that attack each other appearing in at least one bag together) we achieve that $\mathcal{L}^{''}_{out} \not\rightarrowtail a$. But then, $a \in \mathcal{L}_{out}$ but $\mathcal{L}_{out} \not\rightarrowtail a$ which contradicts our assumption of $\mathcal{L}$ being an $X_{>t}$-restricted labeling for $F_{\geq t}$ as $\mathcal{L}_{out} \not\rightarrowtail a$ must hold. It is easy to see that by introducing other contradictions in $a \in \mathcal{L}^{'}_{out} \Leftrightarrow \mathcal{L}^{'}_{in} \not\rightarrowtail a \wedge \mathcal{L}^{'}_{out} \rightarrowtail a$ we have that $a \in \mathcal{L}_{out} \Leftrightarrow \mathcal{L}_{in} \not\rightarrowtail a \wedge \mathcal{L}_{out} \rightarrowtail a$ can not be satisfied.

(3,4,5) In here, we prove that (3) $\mathcal{L}^{'}_{in} \cap X_t = \mathcal{L}^{''}_{in} \cap X_t$ holds for $\mathcal{L} = \mathcal{L}^{'} \cup \mathcal{L}^{''}$ being an $X_{>t}$-restricted complete labeling for $F_{\geq t}$. We have that $X_t = X_{\geq t_1} \cap X_{\geq t_2}$ and $\mathcal{L}^{'}_{in} = \mathcal{L}_{in} \cap X_{\geq t_1}$ as well as $\mathcal{L}^{''}_{in} = \mathcal{L}_{in} \cap X_{\geq t_2}$. But then, we have that $\mathcal{L}^{'}_{in} \cap X_t = \mathcal{L}^{''}_{in} \cap X_t \Leftrightarrow \mathcal{L}_{in} \cap X_{\geq t_1} \cap X_t = \mathcal{L}_{in} \cap X_{\geq t_2} \cap X_t \Leftrightarrow \mathcal{L}_{in} \cap X_{\geq t_1} \cap X_{\geq t_1} \cap X_{\geq t_2} = \mathcal{L}_{in} \cap X_{\geq t_2} \cap X_{\geq t_1} \cap X_{\geq t_2} \Leftrightarrow \mathcal{L}_{in} \cap X_{\geq t_1} \cap X_{\geq t_2} = \mathcal{L}_{in} \cap X_{\geq t_1} \cap X_{\geq t_2}$.

The other cases (4) $\mathcal{L}^{'}_{def} \cap X_t = \mathcal{L}^{''}_{def} \cap X_t$ and (5) $\mathcal{L}^{'}_{out} \cap X_t = \mathcal{L}^{''}_{out} \cap X_t$ follow the same line of argument.

□

For our proof of Lemma 3.43 it remains to show that valid colorings and v-colorings for complete semantics of a branch node t coincide if they coincide in the child nodes t_1 and t_2 of t.

$\Leftarrow$: Suppose that we have a v-coloring C for t, a v-coloring C_1 for t_1 and a v-coloring C_2 for t_2. Furthermore, let $C = C_1 \bowtie C_2$. By assumption, we have $[C_1]_{i_c} = [C_2]_{i_c}$, $[C_1]_{d_c} = [C_2]_{d_c}$ and $[C_1]_{o_c} = [C_2]_{o_c}$. Furthermore, by assumption, C_1 and C_2 are also valid coloring and hence there exists a labeling $\mathcal{L}^{'} \in l_{t_1}(C_1)$ and $\mathcal{L}^{''} \in l_{t_2}(C_2)$. Furthermore, we have that $\mathcal{L}^{'}_{in} \cap X_t = \mathcal{L}^{''}_{in} \cap X_t$, $\mathcal{L}^{'}_{def} \cap X_t = \mathcal{L}^{''}_{def} \cap X_t$ and $\mathcal{L}^{'}_{out} \cap X_t = \mathcal{L}^{''}_{out} \cap X_t$.

Hence, by Lemma 3.44, we have that $\mathcal{L} = \mathcal{L}^{'} \cup \mathcal{L}^{''}$ is an $X_{>t}$-restricted complete labeling for $F_{\geq t}$. It remains to show that the properties of valid colorings (see Definition 3.33) for $\mathcal{L} = \mathcal{L}^{'} \cup \mathcal{L}^{''}$ are satisfied, i.e. $\mathcal{L} \in l_t(C)$. For an argument $a \in X_t$ we analyze the following cases:

- By Definition 3.42 we have that $C(a) = in_c$ iff $C_1(a) = in_c$ and $C_2(a) = in_c$. As C_1 and C_2 are valid colorings we have (by Definition 3.33) that $a \in \mathcal{L}^{'}_{in}$ and $a \in \mathcal{L}^{''}_{in}$. Furthermore, as $\mathcal{L}_{in} = \mathcal{L}^{'}_{in} \cup \mathcal{L}^{''}_{in}$ we have that $C(a) = in_c$ iff $a \in \mathcal{L}_{in}$.

- By Definition 3.42 we have that $C(a) = def_c$ iff $C_1(a) = def_c$ or $C_2(a) = def_c$. As C_1 and C_2 are valid colorings we have (by Definition 3.33) that $a \in \mathcal{L}^{'}_{def}$ and $\mathcal{L}^{'}_{in} \rightarrowtail a$ or $a \in \mathcal{L}^{''}_{def}$ and $\mathcal{L}^{''}_{in} \rightarrowtail a$. As $\mathcal{L}_{in} = \mathcal{L}^{'}_{in} \cup \mathcal{L}^{''}_{in}$ we have that $\mathcal{L}_{in} \rightarrowtail a$ and as $\mathcal{L}_{def} = \mathcal{L}^{'}_{def} \cup \mathcal{L}^{''}_{def}$ it follows that $C(a) = def_c$ iff $a \in \mathcal{L}_{def}$ and $\mathcal{L}_{in} \rightarrowtail a$.

- By Definition 3.42 we have that $C(a) = defp_c$ iff $C_1(a) = defp_c$ and $C_2(a) = defp_c$. As C_1 and C_2 are valid colorings we have (by Definition 3.33) that $a \in \mathcal{L}^{'}_{def}$ and $\mathcal{L}^{'}_{in} \not\rightarrow a$ and $a \in \mathcal{L}^{''}_{def}$ and $\mathcal{L}^{''}_{in} \not\rightarrow a$. As $\mathcal{L}_{in} = \mathcal{L}^{'}_{in} \cup \mathcal{L}^{''}_{in}$ we have that $\mathcal{L}_{in} \not\rightarrow a$ and as $\mathcal{L}_{def} = \mathcal{L}^{'}_{def} \cup \mathcal{L}^{''}_{def}$ it follows that $C(a) = def_c$ iff $a \in \mathcal{L}_{def}$ and $\mathcal{L}_{in} \not\rightarrow a$.

- By Definition 3.42 we have that $C(a) = out_c$ iff $C_1(a) = out_c$ or $C_2(a) = out_c$. As C_1 and C_2 are valid colorings we have (by Definition 3.33) that $a \in \mathcal{L}^{'}_{out}$, $\mathcal{L}^{'}_{in} \not\rightarrow a$, $a \not\rightarrow \mathcal{L}^{'}_{in}$, $\mathcal{L}^{'}_{out} \rightarrowtail a$ or $a \in \mathcal{L}^{''}_{out}$, $\mathcal{L}^{''}_{in} \not\rightarrow a$, $a \not\rightarrow \mathcal{L}^{''}_{in}$, $\mathcal{L}^{''}_{out} \rightarrowtail a$. As $\mathcal{L}_{in} = \mathcal{L}^{'}_{in} \cup \mathcal{L}^{''}_{in}$ and $\mathcal{L}_{out} = \mathcal{L}^{'}_{out} \cup \mathcal{L}^{''}_{out}$ we have that $C(a) = out_c$ iff $a \in \mathcal{L}_{out}$, $\mathcal{L}_{in} \not\rightarrow a$, $a \not\rightarrow \mathcal{L}_{in}$ and $\mathcal{L}_{out} \rightarrowtail a$.

- It is easy to check that by the same line of argument as above we have that $C(a) = outp_c$ iff $a \in \mathcal{L}_{out}$, $\mathcal{L}_{in} \not\rightarrow a$, $a \not\rightarrow \mathcal{L}_{in}$ and $\mathcal{L}_{out} \not\rightarrow a$.

$\Rightarrow$: Now, suppose that we have a valid coloring C for t. Then there exists a labeling $\mathcal{L} \in l_t(C)$. Furthermore, we define $\mathcal{L}^{'} = \langle \mathcal{L}^{'}_{in}, \mathcal{L}^{'}_{def}, \mathcal{L}^{'}_{out} \rangle$ and $\mathcal{L}^{''} = \langle \mathcal{L}^{''}_{in}, \mathcal{L}^{''}_{def}, \mathcal{L}^{''}_{out} \rangle$ where $\mathcal{L}^{'}_{in} =$

$\mathcal{L}_{in} \cap X_{\geq t_1}$, $\mathcal{L}^{''}_{in} = \mathcal{L}_{in} \cap X_{\geq t_2}$, $\mathcal{L}^{'}_{def} = \mathcal{L}_{def} \cap X_{\geq t_1}$, $\mathcal{L}^{''}_{def} = \mathcal{L}_{def} \cap X_{\geq t_2}$, $\mathcal{L}^{'}_{out} = \mathcal{L}_{out} \cap X_{\geq t_1}$ and $\mathcal{L}^{''}_{out} = \mathcal{L}_{out} \cap X_{\geq t_2}$. Then, by Lemma 3.45, we have that $\mathcal{L}^{'}$ is an $X_{>t_1}$-restricted complete labeling for $F_{\geq t_1}$, $\mathcal{L}^{''}$ is an $X_{>t_2}$-restricted complete labeling for $F_{\geq t_2}$, $\mathcal{L}^{'}_{in} \cap X_t = \mathcal{L}^{''}_{in} \cap X_t$, $\mathcal{L}^{'}_{def} \cap X_t = \mathcal{L}^{''}_{def} \cap X_t$ and $\mathcal{L}^{'}_{out} \cap X_t = \mathcal{L}^{''}_{out} \cap X_t$.

By Lemma 3.34 we have two colorings C_1 and C_2 such that $\mathcal{L}^{'} \in l_{t_1}(C_1)$ and $\mathcal{L}^{''} \in l_{t_2}(C_2)$. Furthermore, due to Lemma 3.45, we have that $[C_1]_{i_c} = [C_2]_{i_c}$, $[C_1]_{d_c} = [C_2]_{d_c}$ and $[C_1]_{o_c} = [C_2]_{o_c}$. Hence, $C^* = C_1 \bowtie C_2$ is a v-coloring. It remains to show that $C = C^*$, i.e. that the valid coloring C is also a v-coloring. Due to brevity, we only outline the case $C(a) = in_c$:

- Consider the case $C(a) = in_c$. Then, by Definition 3.33, we know that $a \in \mathcal{L}_{in}$. Furthermore, as $X_t = X_{t_1} = X_{t_2}$ and $\mathcal{L}^{'}_{in} = \mathcal{L}_{in} \cap X_{\geq t_1}$ as well as $\mathcal{L}^{''}_{in} = \mathcal{L}_{in} \cap X_{\geq t_2}$ we have that $a \in \mathcal{L}^{'}_{in}$ and $a \in \mathcal{L}^{''}_{in}$. But then, due to Definition 3.33, we know that $C_1(a) = C_2(a) = in_c$. This, in turn, means that $C^*(a) = (C_1 \bowtie C_2)(a) = in_c$ and hence $C(a) = C^*(a)$.

□

Theorem 3.46. *Let $(\mathcal{T}, \mathcal{X})$ be a normalized tree decomposition of an AF $F = \langle A, R \rangle$. Then, for each complete coloring C for a node $t \in \mathcal{T}$, it holds that C is a valid coloring for t iff C is a v-coloring for t.*

Proof. We showed that valid colorings and v-colorings for complete semantics coincide in every node type of normalized tree decompositions. Hence, by structural induction over the tree decomposition, they coincide in every node of a normalized tree decomposition. □

A normalized tree decomposition of our running example (see Example 2.6) with v-colorings for complete semantics is given in Figure 3.4. For all $\mathcal{L} \in l_t(C)$ where $\mathcal{L} = \langle \mathcal{L}_{in}, \mathcal{L}_{def}, \mathcal{L}_{out} \rangle$ we denote the sets of arguments that are in $\mathcal{L}_{in}$ by $l^i_t(C)$.

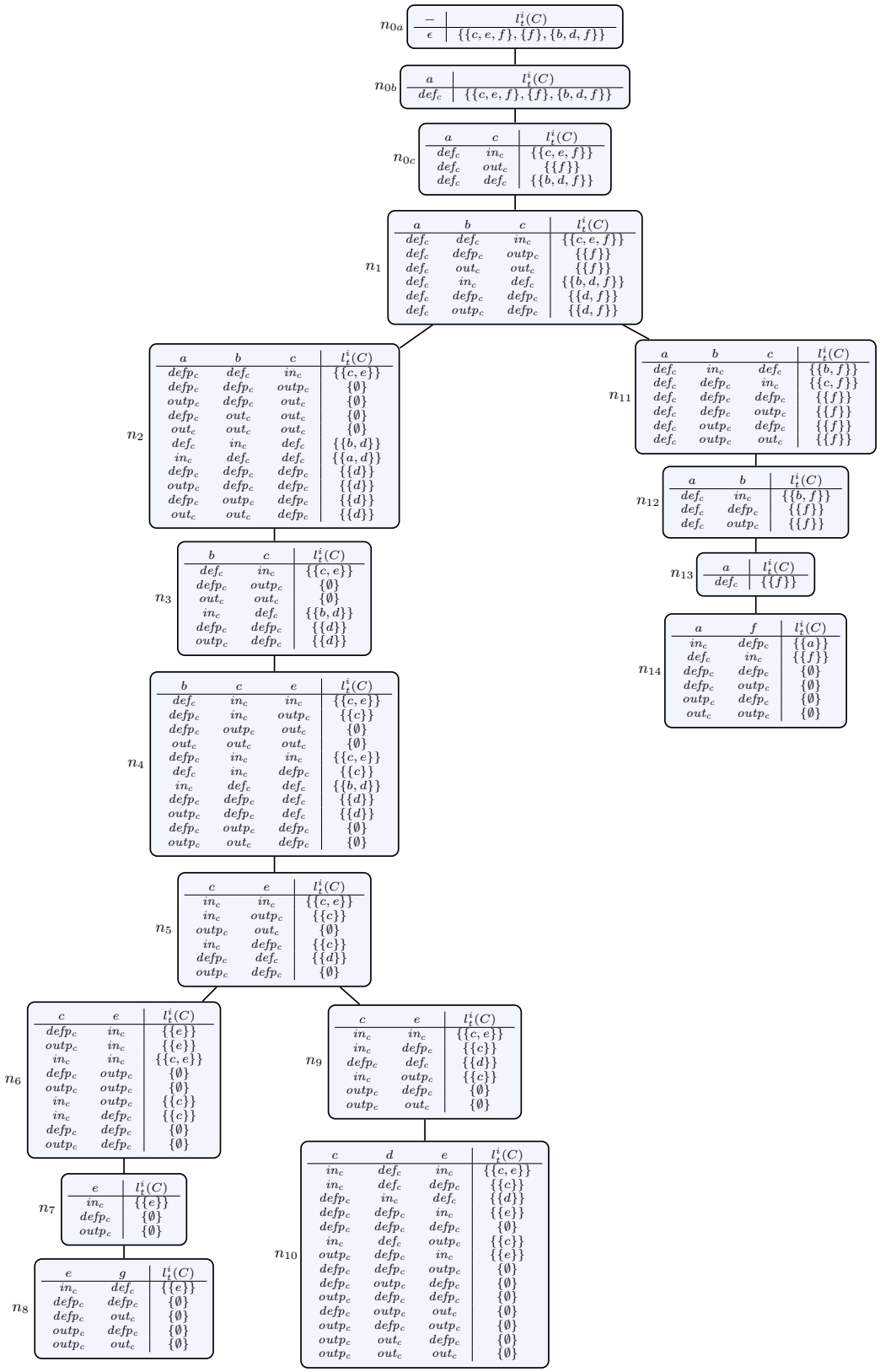


Figure 3.4: Normalized Tree Decomposition with V-Colorings for Complete Semantics

3.4 Algorithm for Admissible Semantics (Semi-Normalized)

In this section we present a novel algorithm for admissible semantics on semi-normalized tree decompositions. Dvořák et al. [2010a] already presented an algorithm for normalized tree decompositions. Thus, we have to define v-colorings for exchange nodes where, compared to introduction and removal nodes of normalized tree decompositions, several arguments can be introduced or removed.

In Section 3.1 we already defined *B-restricted admissible sets* (see Definition 3.5) and *valid colorings* for admissible semantics (see Definition 3.8). Furthermore, Dvořák et al. [2010a] proved that there exists a one-to-one mapping between the extensions of the colorings C, $e_t(C)$, in a node t and the $X_{>t}$-restricted admissible sets S for $F_{\geq t}$. In other words, the valid colorings for a node t represent the $X_{>t}$-restricted admissible sets S for $F_{\geq t}$ and different colorings represent different $X_{>t}$-restricted admissible sets.

As we do not want to compute $e_t(C)$ explicitly in every node t it remains to define v-colorings for the different node types in semi-normalized tree decompositions and to prove that they correspond to valid colorings.

Branch Node: In this thesis the definition of v-colorings for branch nodes is given in Definition 3.12. As shown by Dvořák et al. [2010a] the following lemma holds for branch nodes:

Lemma 3.47. *[Dvořák et al., 2010a] For any* branch node t *in a tree-decomposition of an AF, valid colorings and v-colorings for admissible semantics coincide if they coincide in the child nodes t_1 and t_2 of t.*

It remains to define v-colorings for exchange nodes.

Exchange Node:

Definition 3.48. *Let t be an* exchange node *of a tree decomposition and t_1 be the child node of t. Furthermore, let $S' \subseteq X_{t_1}$, $S'' \subseteq A$ and $S' \cap S'' = \emptyset$ such that $X_t = (X_{t_1} \setminus S') \cup S''$. Finally, let $T \in cf(F|_{S''})$ (i.e. T is a conflict-free in the sub-framework of F induced by S'').*

If

- *C is a v-coloring for t_1,*
- *$\forall a \in S' : C(a) \neq att$ and*
- *$[C]_{i_a} \cup T$ is conflict-free.*

then $(C - S') + T$ is a v-coloring for t.

For $C : X_{t_1} \rightarrow \{in_a, def_a, att_a, out_a\}$ and $D : U \rightarrow \{in_a, def_a, att_a, out_a\}$ where $U = X_{t_1} \setminus S'$ we define $C - S'$ over U and $D + T$ over X_t:

$$(C - S')(b) = C(b) \textit{ for each } b \in X_{t_1} \backslash S'$$

$$(D + T)(b) = \begin{cases} in_a & \textit{if } C(b) = in_a \vee b \in T \\ def_a & \textit{if } C(b) = def_a \vee T \rightarrowtail b \\ att_a & \textit{if } b \in U\textit{: } T \not\rightarrow b \wedge ((C(b) = att_a) \vee (C(b) = out_a \wedge b \rightarrowtail T)) \\ & \textit{if } b \in S''\textit{: } ([C]_{i_a} \cup T \not\rightarrow b) \wedge (b \rightarrowtail [C]_{i_a} \cup T) \\ out_a & \textit{if } b \in U\textit{: } C(b) = out_a \wedge T \not\rightarrow b \wedge b \not\rightarrow T \\ & \textit{if } b \in S''\textit{: } ([C]_{i_a} \cup T \not\rightarrow b) \wedge (b \not\rightarrow [C]_{i_a} \cup T) \end{cases}$$

Lemma 3.49. *For any* exchange node t *in a tree-decomposition of an AF, valid colorings and v-colorings for admissible semantics coincide if they coincide in the child node* t_1 *of* t.

Proof sketch. $S' = \{a_1, a_2, ..., a_i\}$ are the removed arguments in X_t and $S'' = \{b_1, b_2, ..., b_j\}$ are the introduced arguments in X_t. We can define intermediate nodes $r_0, r_1, ..., r_i$ and $i_0, i_1, ..., i_j$ between t_1 and t where:

- $X_{r_0} = X_{t_1}$,
- $X_{r_k} = X_{r_{k-1}} \backslash \{a_k\}$ for $0 < k \leq i$,
- $X_{i_0} = X_{r_i}$,
- $X_{i_l} = X_{i_{l-1}} \cup \{b_l\}$ for $0 < l \leq j$ and
- $X_{i_j} = X_t$.

Due to our construction of the intermediate nodes we have that $r_0, r_1, ..., r_i$ correspond to removal nodes of normalized tree decompositions and $i_0, i_1, ..., i_j$ correspond to introduction nodes of tree decompositions. Dvořák et al. [2010a] showed that v-colorings and valid colorings for nodes of type introduction and removal coincide. Hence, in particular they coincide in X_{i_j} for $F_{\geq i_j}$. Due to the construction of the bags we have that the $X_{>r_0}$-restricted admissible sets for $F_{\geq r_0}$ correspond to the $X_{>t_1}$-restricted admissible sets for $F_{\geq t_1}$ and the $X_{>i_j}$-restricted admissible sets for $F_{\geq i_j}$ correspond to the $X_{>t}$-restricted admissible sets for $F_{\geq t}$. Note that $F_{\geq t}$ for semi-normalized tree decompositions corresponds to $F_{\geq i_j}$ of our construction for normalized tree decompositions. Furthermore, $X_{i_j} = X_t$. Hence we have that valid colorings for X_{i_j} and X_t coincide. As already showed the v-colorings for X_{i_j} in $F_{\geq i_j}$ coincide. To prove Lemma 3.49 it remains to show that v-colorings for X_{i_j} correspond to v-colorings for X_t.

$\Rightarrow$: Suppose that C_{i_j} is a v-coloring the node i_j. Then we know that there exists a v-coloring $C_{t_1} = C_{r_0}$ such that $C_{i_j} = (...(((...((C_{r_0} - a_1) - a_2)... - a_i) \oplus_1 b_1) \oplus_2 b_2)... \oplus_j b_j)$ where $\oplus_l \in \{+, \dot{+}\}$. Consider $T = \{b_l : \oplus_l = \dot{+}\}$. We show that $C_{i_j} = (C_{t_1} - S') + T$.

- Within our construction of C_{i_j} we have that for each node r_k with $0 < k \leq i$ where $X_{r_k} = X_{r_{k-1}} \backslash \{a_k\}$ and $C_{r_{k-1}}(a_k) \neq att_a$ as we know that $C_{r_k} = C_{r_{k-1}} - a_k$ (see Definition 3.11) is a v-coloring for r_k (This is proved in [Dvořák et al., 2010a]). Moreover, we have $C_{r_{k-1}}(a_k) = C_{r_0}(a_k)$. Then, in node t, we have that for all $a \in S' : C(a) \neq att_a$. Especially note that $C_{r_i} = C_{i_0}$ is a v-coloring for i_0.

- Furthermore, in the nodes i_l with $0 < l \leq j$ exactly one argument b_l is introduced. We define $T_l = \{b_y : \oplus_y = \dot{+} \text{ for } 0 < y \leq l\}$, i.e. T_l for a node i_l contains the arguments b_y that are introduced with the operation $\dot{+}$. Now, towards a contradiction, suppose that $[C_{i_0}]_{i_a} \cup T_l$ is not conflict-free where l is minimal, i.e. $[C_{i_0}]_{i_a} \cup T_{l-1}$ is conflict-free but $[C_{i_0}]_{i_a} \cup T_l$ is not conflict-free. If $[C_{i_0}]_{i_a}$ contains a conflict this contradicts our assumption of C_{i_0} being a v-coloring. Furthermore, it can be the case that either $b_y \rightarrowtail b_y$, $[C_{i_{l-1}}]_{i_a} \rightarrowtail b_y$ or $b_y \rightarrowtail [C_{i_{l-1}}]_{i_a}$. But due to the definition of $C_{i_{l-1}} \dot{+} b_y$ we again have a contradiction. Hence, $[C_{i_0}]_{i_a} \cup T_l$ is conflict-free and we have that also $[C]_{i_a} \cup T$ is conflict-free for t.

Now we show that the computation of v-colorings over C_{r_k} for $0 < k \leq i$ and C_{i_l} for $0 < l \leq j$ where $X_{r_0} = X_{t_1}$, $X_{i_j} = X_t$ and $C_{r_k} = C_{t_1}$ is equivalent to the computation of v-colorings for $C_t = (C_{t_1} - S') + T$, i.e. we show that $C_{i_j} = C_t$.

1. First, consider $C - S'$: $C - S'$ is a coloring over $X_{t_1} \setminus S'$. C_{r_i} is a coloring over $X_{t_1} \setminus S'$. By definition they both coincide with $C_{r_0} = C_{t_1}$ on $X_{t_1 \setminus S'}$. Hence, we have that $C - S' = C_{r_i}$.

2. Now, for an argument $a \in X_t$ we want to show that $C_{i_j}(a) = C_t(a)$ where $C_{i_j}(a) = (C_{r_i} \oplus_1 ... \oplus_j b_j)(a)$ and $C_t(a) = (C_{r_i} + T)(a)$. We exemplify this for the case $C_{i_j}(a) = in_a$. Then, either $a \in [C_{r_0}]_{i_a}$ (respectively $a \in [C_{r_i}]$) or a is added with the $C_{i_l} \dot{+} a$ operation somewhere along the nodes i_l where $0 < l \leq j$. But then, either $C_{t_1}(a) = in_a$ or $a \in T_t$ and therefore $C_t(a) = in_a$.

$\Leftarrow$: Now suppose that C_t is a v-coloring for X_t. Then, there exists a C_{r_0}, T such that $(C_{r_0} - S') + T = C_t$. We have to show that $C_{i_j} = C_t$ where $C_{i_j} = (...(((...((C_{r_0} - a_1) - a_2)... - a_i) \oplus_1 b_1) \oplus_2 b_2)... \oplus_j b_j)$ with $\oplus_l = \dot{+}$ if $b_i \in T$ and $\oplus_l = +$ if $b_i \notin T$. We first show that C_{i_j} is a v-coloring:

- We know that $\forall a_k \in S' : C(a_k) \neq att$. But then it it easy to see that for each $C_{r_{k-1}}$ we have that $C_{r_{k-1}}(a_k) \neq att$ (as $C_{r_{k-1}} = C_{r_0}(a)$). Then we have that all $C_{r_k} = C_{r_{k-1}} - a_k$ for $0 \leq k \leq i$ are v-colorings.

- $[C]_{i_a} \cup T$ is conflict-free by assumption. But then, every subset $[C_{i_k - 1}] \cup \{b_y\}$ for $b_y \in T$ is conflict free, i.e. we have for each $C_{i_{l-1}} \dot{+} b_y$ that $b_y \not\rightarrowtail b_y$, $C_{i_{l-1}} \not\rightarrowtail b_y$ and $b_y \not\rightarrowtail C_{i_{l-1}}$. Then we have that all C_{i_l} for $0 \leq l \leq j$ are v-colorings. In particular, C_{i_j} is a v-coloring.

It remains to show that $C_t = C_{i_j}$. We exemplify this for an argument $a \in X_t$ where $C_t(a) = ((C_{r_0} - S') + T)(a) = in_a$. Then, either $C_{r_0}(a) = in_a$ or $a \in T$. But then, either $a \in [C_{r_0}]_{i_a}$ and we have that a is not removed in any node and maintains its color over our construction of C_{i_j}. Or, if $a \in T$ then a is colored with in_a in one of $C_{i_{l-1}} \dot{+} a$. □

Then we have that the following theorem holds:

Theorem 3.50. *Let $(\mathcal{T}, \mathcal{X})$ be a semi-normalized tree decomposition of an AF $F = \langle A, R \rangle$. Then, for each admissible coloring C for a node $t \in \mathcal{T}$, it holds that C is a valid coloring for t iff C is a v-coloring for t.*

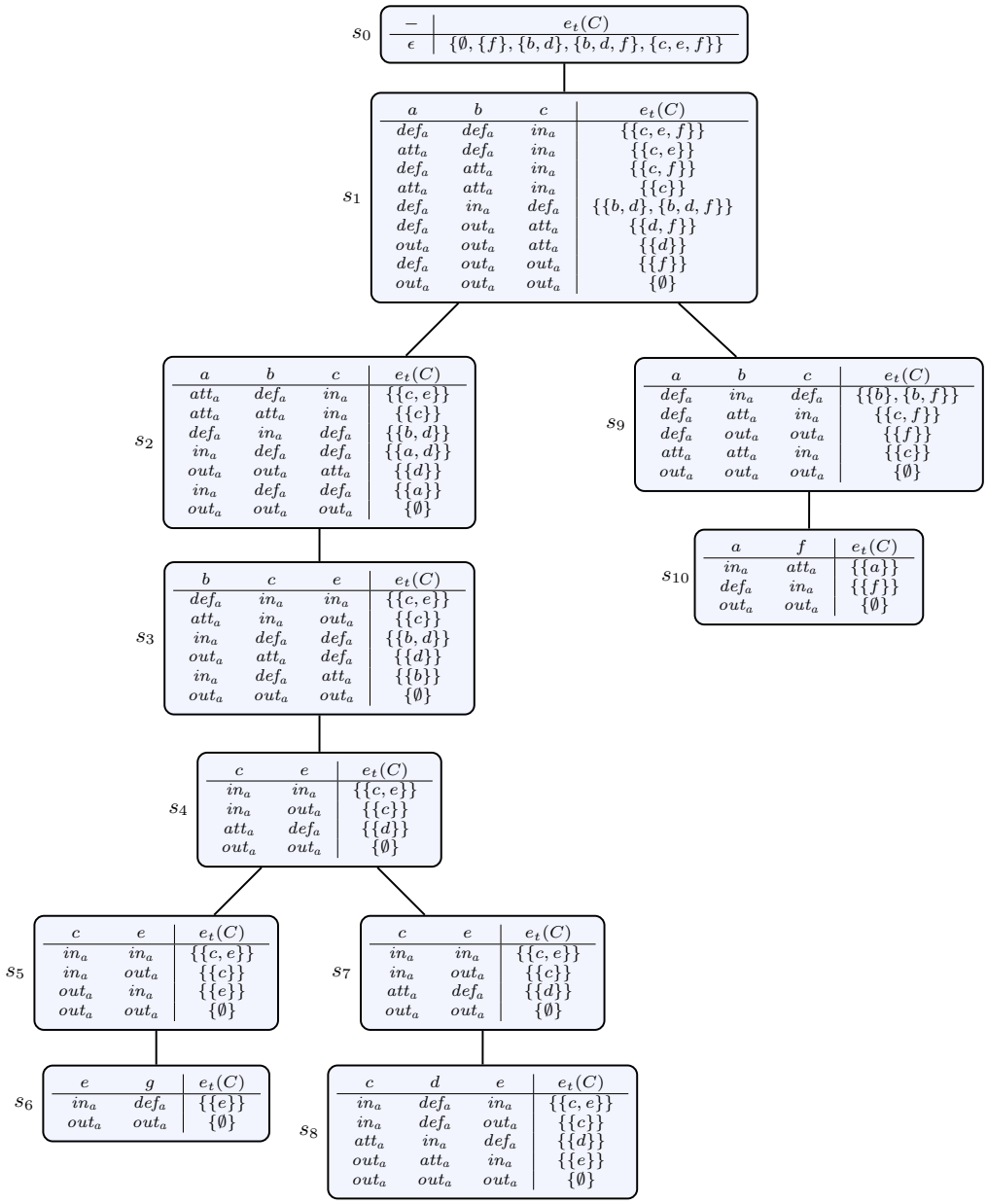


Figure 3.5: Semi-normalized Tree Decomposition with V-Colorings for Admissible Semantics

A semi-normalized tree decomposition of our running example with v-colorings for each node is given in Figure 3.5.

Example 3.4. Consider the exchange node s_9 where the argument f is removed and the arguments b and c are introduced. The coloring $C'_{s_{10}}$ where $C'_{s_{10}}(a) = in_a$ and $C'_{s_{10}}(f) = att_a$ is removed. Furthermore consider the coloring $C''_{s_{10}}$ where $C''_{s_{10}}(a) = def_a$ and $C''_{s_{10}}(f) = in_a$. In s_9, the conflict-free sets of the introduced arguments b and c are $\{\emptyset, \{b\}, \{c\}\}$. Thus, for $C''_{s_{10}}(a) = def_a$ we have three colorings in s_9 as shown in Figure 3.5.

3.5 Evaluation of Decision Problems

In the previous sections we presented the algorithm definitions for admissible, stable and complete semantics. These algorithms define how the σ-extensions for a given problem instance can be obtained. Additionally, the algorithm definitions support the evaluation of decision problems. In here we outline the ideas for the evaluation of $Cred_\sigma$ and $Skept_\sigma$ (as described in Section 2.4). These ideas are based on the work of Dvořák et al. [2010a].

As the approach for the evaluation of $Cred_\sigma$ and $Skept_\sigma$ is almost identical for all of our semantics $\sigma \in \{adm, stable, comp\}$ we present the overall idea and give for each problem an example for one of the semantics.

Credulous Acceptance

The decision problem of credulous acceptance for an argument a can be answered with help of the computation of v-colorings: Each v-coloring where $C(a)$ is set to in (which corresponds to either in_a, in_s or in_c for the respective semantics) is marked. Additionally, we pass the information of marked v-colorings up to the root: If a v-coloring somewhere above in the tree is constructed on basis of a marked v-coloring it is marked as well. Finally, if the v-coloring in the root node is marked we know that a is credulously accepted (wrt. to a certain semantics and a problem instance).

Example 3.5. Consider the argumentation framework F of our running example (see Example 2.6) and the normalized tree decomposition with v-colorings for complete semantics as depicted in Figure 3.4. We want to know if d is credulously accepted wrt. to complete semantics, i.e. if $Cred_{comp}(F, d)$ holds.

Consider node n_{10} of the tree decomposition and consider the coloring where $C_{n_{10}}(c) = defp_c$, $C_{n_{10}}(d) = in_c$ and $C_{n_{10}}(e) = def_c$. This coloring is marked because $C(d) = in_c$. Now, consider the parent node n_9 which is a removal node. The coloring where $C_{n_9}(c) = defp_c$ and $C_{n_9}(e) = def_c$ is marked as well because it results from the marked coloring of the child node.

Skeptical Acceptance

Again, we make use of the computed v-colorings in the nodes of the tree decomposition. Skeptical acceptance asks if an argument a is contained in every σ-extension of an AF. In this case we mark all v-colorings where $C(a) \neq in$ (for nodes where $a \in X_t$). Again, we pass this information upwards along the tree decomposition. If a coloring is constructed on basis of a marked coloring we mark this coloring as well. If the v-coloring in the root node is marked we know that there exists an extension that does not contain a. Otherwise, a is skeptically accepted.

For admissible semantics this decision problem is trivial as the empty set is always an extension of the AF and thus a can not be skeptically accepted.

Example 3.6. Again, consider the argumentation framework F of our running example (see Example 2.6) and the normalized tree decomposition with v-colorings for stable semantics as depicted in Figure 3.3. We want to know if b is skeptically accepted wrt. to stable semantics, i.e. if $Skept_{stable}(F, b)$ holds.

In node n_{12} b is introduced. Hence, we mark the coloring where $C(b) \neq in_s$, that is, the coloring $C'_{n_{12}}$ where $C'_{n_{12}}(a) = def_s$ and $C'_{n_{12}}(b) = out_s$. The coloring $C''_{n_{12}}$ where $C''_{n_{12}}(a) = def_s$ and $C''_{n_{12}}(b) = in_s$ is not marked.

In n_{11} we then mark the colorings that are constructed on basis of $C'_{n_{12}}$, that is, the coloring $C'_{n_{11}}$ with $C'_{n_{11}}(a) = def_s$, $C'_{n_{11}}(b) = out_s$ and $C'_{n_{11}}(c) = in_s$ and the coloring $C''_{n_{11}}$ with $C''_{n_{11}}(a) = def_s$, $C''_{n_{11}}(b) = out_s$ and $C''_{n_{11}}(c) = out_s$. The coloring $C'''_{n_{11}}$ with $C'''_{n_{11}}(a) = def_s$, $C'''_{n_{11}}(b) = in_s$ and $C'''_{n_{11}}(c) = def_s$ is not marked.

In n_4, b is introduced as well: The only coloring is marked and hence the only colorings in n_3 and n_2 are marked as well.

Now, in the branch node n_2, we have that only marked colorings can be joined (due to the definition of branch nodes) and hence the only resulting is marked as well. Finally, we obtain that b is not skeptically accepted.

CHAPTER 4

Implementation

The algorithms that were presented in the last chapter are implemented on basis of the already-existing SHARP framework (Smart Hypertree decomposition-based Algorithm fRamework for Parameterized problems). SHARP provides the necessary interfaces for the implementation of tree decomposition based algorithms. For a problem instance the framework builds the tree decomposition (in our case a normalized or semi-normalized tree decomposition) and executes the user-defined implementation for each node of the decomposition. The framework is described in Section 4.1.

In Section 4.2 we present the implementation of the algorithms for admissible semantics on normalized as well as semi-normalized tree decompositions. We focus on the implementation of the different node types (leaf, introduction, removal, exchange and branch node) and present strategies for the optimization of the different node types. We present ideas for the optimization of exchange nodes that can not be applied to leaf, introduction and removal nodes of normalized tree decompositions. Furthermore we focus on branch nodes as this node type is the most complex one.

In the course of this thesis the implementation of the novel algorithms for admissible semantics on normalized as well as semi-normalized tree decompositions and the implementation of stable and complete semantics on normalized tree decompositions are included in the dynPARTIX (Dynamic Programming Argumentation Reasoning Tool) project. In Section 4.3 we give a system description of dynPARTIX. Furthermore we describe the input format for argumentation problems and present the currently-available command line options for the dynPARTIX software.

4.1 The SHARP Framework

The SHARP[1] (Smart Hypertree decomposition-based Algorithm fRamework for Parameterized problems) framework provides the basis for the implementation of our algorithms. It was originally implemented by Michael Morak and was first used in his project on a dynamic programming-based Answer Set Programming solver (dynASP[2]). In here we outline the most relevant aspects of the SHARP framework [Morak, 2011]:

The SHARP framework is based in the observation that most algorithms for fixed-parameter problems where the problem is parameterized by tree width follow a uniform approach:

1. Read in an input instance of a fixed tree width and obtain a tree decomposition of a certain width.
2. Traverse the tree decomposition in bottom-up order and compute the intermediate results for each node based on the intermediate results of the sub-node(s).
3. Do a second traversal in top-down order and compute the relevant solution(s). In our case the solutions may be an enumeration of all extensions for a given semantics, the overall number of extensions (counting) or a *yes* or *no* answer to a decision problem.

The SHARP framework is implemented in C++. It handles the program flow and provides the tree decompositions of input instances. In the following we present the relevant parts of the framework and the interfaces that have to be implemented for our algorithms.

The `Problem` Class

The `Problem` class serves as the main interface between the user-specified algorithm and the overall workflow of the program. A simplified version of the header file for the `Problem` class is depicted in Listing 4.1. In order to maintain readability the listing only contains the most relevant methods and variables. The header file includes the following methods that have to be implemented by the user:

- `parse()`: This method is responsible for reading in the problem instance. The framework does not restrict the user to any input format. It then should save the data in an user-defined internal format.
- `preprocess()`: This method is intended for optimizations on the input and may alter the data that was read in by the `parse()` method.
- `buildHypergraphRepresentation()`: This method should convert the previously stored data into a format on which the SHARP framework can work on, namely an instance of the `Hypergraph` class.

Furthermore, the most important methods that are implemented within the SHARP framework are:

[1] `http://www.dbai.tuwien.ac.at/research/project/sharp/`
[2] `http://www.dbai.tuwien.ac.at/research/project/dynasp/`

- `calculateTreeWidth()`: This method calls the methods `parse()`, `preprocess()` and `buildHypergraphRepresentation()`. Then, it generates a tree decomposition and the width of the tree decomposition is returned.

- `calculateSolution()`: This method calls the methods `parse()`, `preprocess()` and `buildHypergraphRepresentation()`. It generates a tree decomposition and executes the user-specified `AbstractHTDAlgorithm` implementation. Finally, it returns a `Solution` instance for the problem.

```
1  class Problem
2  {
3  public:
4    Problem(bool collectBenchmarkInformation = false);
5    virtual ~Problem();
6
7  public:
8    // computes the width of the tree decomposition
9    int calculateTreeWidth();
10   //  computes the solution for a problem with a given algorithm
11   Solution *calculateSolution(AbstractHTDAlgorithm *algorithm);
12
13 protected:
14   // reads in the problem instance
15   virtual void parse() = 0;
16   // called after parsing, optional optimizations
17   virtual void preprocess() = 0;
18   // creates the internal SHARP representation of the problem instance
19   virtual Hypergraph *buildHypergraphRepresentation() = 0;
20
21 private:
22   // omitted here
23 };
```

Listing 4.1: `Problem` header file (simplified) [Morak, 2012]

A user then has to derive his own `*Problem` class and provide implementations of the `parse()`, `preprocess()` and `buildHypergraphRepresentation()` methods. In our case we implemented the methods within the `ArgumentationProblem` class.

The `Abstract*HTDAlgorithm` Classes

The `Abstract*HTDAlgorithm` classes provide the necessary interface for the algorithm implementation. Based on the normalization type of the tree decomposition the SHARP framework provides different class definitions. In our case, the `AbstractHTDAlgorithm`, the `AbstractSemiNormalizedHTDAlgorithm` and the `AbstractNormalizedHTDAlgorithm` are relevant. A simplified version of the `AbstractHTDAlgorithm` header file is depicted in Listing 4.2.

In the following we analyze the most relevant methods that are declared in the header file of Listing 4.2 and have to be implemented in the `Abstract*HTDAlgorithm` classes.

```
// For algorithms on non-normalized tree decompositions
class AbstractHTDAlgorithm
{
public:
  AbstractHTDAlgorithm(Problem *problem);
  virtual ~AbstractHTDAlgorithm();

  // evaluates a problem based on a tree decomposition and a problem type
  Solution *evaluate(const ExtendedHypertree *root, Instantiator *instantiator =
      NULL);

protected:
  // returns the tree decomposition (as is)
  virtual const ExtendedHypertree *prepareHypertreeDecomposition(const
      ExtendedHypertree *root);
  // evaluate a node of the tree decomposition
  virtual TupleSet *evaluateNode(const ExtendedHypertree *node) = 0;
  // called after all nodes in tree are evaluated
  virtual Solution *selectSolution(TupleSet *tuples, const ExtendedHypertree *root
      ) = 0;

private: // omitted here
};

// For algorithms on semi-normalized tree decompositions
class AbstractSemiNormalizedHTDAlgorithm : public AbstractHTDAlgorithm
{
public:
  AbstractSemiNormalizedHTDAlgorithm(Problem *problem);
  virtual ~AbstractSemiNormalizedHTDAlgorithm();

protected:
  // returns a semi-normalized tree decomposition
  virtual const ExtendedHypertree *prepareHypertreeDecomposition(const
      ExtendedHypertree *root);
  // calls the evaluate*Node() methods
  virtual TupleSet *evaluateNode(const ExtendedHypertree *node);

  virtual TupleSet *evaluateBranchNode(const ExtendedHypertree *node) = 0;
  virtual TupleSet *evaluatePermutationNode(const ExtendedHypertree *node) = 0;
};

// For algorithms on normalized tree decompositions
class AbstractNormalizedHTDAlgorithm : public AbstractSemiNormalizedHTDAlgorithm
{
public:
  AbstractNormalizedHTDAlgorithm(Problem *problem);
  virtual ~AbstractNormalizedHTDAlgorithm();

protected:
  // returns a normalized tree decomposition
  virtual const ExtendedHypertree *prepareHypertreeDecomposition(const
      ExtendedHypertree *root);
  // calls the other evaluate*Node methods
  virtual TupleSet *evaluatePermutationNode(const ExtendedHypertree *node);

  virtual TupleSet *evaluateIntroductionNode(const ExtendedHypertree *node) = 0;
  virtual TupleSet *evaluateRemovalNode(const ExtendedHypertree *node) = 0;
  virtual TupleSet *evaluateLeafNode(const ExtendedHypertree *node) = 0;
};
```

Listing 4.2: `AbstractHTDAlgorithm` header file (simplified)

- `prepareHypertreeDecomposition()`: Each class contains a method `prepareHypertreeDecomposition()`. It is responsible for the normalization of the tree decomposition. In case of the `AbstractHTDAlgorithm` class it simply returns the `ExtendedHypertree` instance.

  The `AbstractSemiNormalizedHTDAlgorithm` class implementation returns a hyper tree that consists of binary branch nodes that contain the same elements as their respective child nodes.

  In the `AbstractNormalizedHTDAlgorithm` implementation additional nodes are added such that either one element is introduced or removed in each node (except for the leaf nodes).

- `evaluate*Node()`: Depending on the `Abstract*HTDAlgorithm` class there exist different `evaluate*Node()` methods. If, for example, a user wants to develop an algorithm based on the `AbstractNormalizedHTDAlgorithm` class, he has to implement the `evaluateBranchNode()`, `evaluateIntroductionNode()`, `evaluateRemovalNode()` and `evaluateLeafNode()` methods. In this case the class contains an implemented version of the `evaluatePermuationNode()` method: This method simply calls the other node implementations based on their node type.

  Furthermore note that exchange nodes as defined in the previous chapters exactly correspond to permutation nodes within the SHARP framework.

- `selectSolution()`: This method is used by all algorithm classes and is called after the evaluation of the tree decomposition. It allows to implement special code that has to be applied to the tuples within the root node.

- `evaluate()`: The `evaluate()` method simply prepares the tree decomposition based on the normalization type (e.g. `prepareHypertreeDecomposition()` is called), calls `evaluate*Node()` for the root node and finally returns the value of the `selectSolution()` method.

The main task for an algorithm developer is to implement `evaluate*Node()` of the respective `Abstract*HTDAlgorithm` class. Note that the bottom-up traversal of the tree decomposition has to implemented by the algorithm developer: In each `evaluate*Node()` method the `evaluate*Node()` method(s) of the child node(s) are called first. Then, the `evaluate*Node()` method can work on the intermediate results of its child node(s).

Data Representation

Intermediate results of each computation within the `evaluate*Node()` methods are stored in instances of the `Tuple` class. As these results heavily depend on the algorithm definition an algorithm developer has to define his own `*Tuple` class. In each node the currently 'possible' tuples are computed on basis of the tuples that were returned form a call to the `evaluate*Node()` method(s) of the child node(s). New tuples may be created or old ones removed (this reflects the dynamic programming approach that all algorithms on tree decompositions have in common).

Each tuple can represent one or more partial solutions to the overall problem. It is not feasible to compute and store all associated (partial) solutions directly as this would result in exponential run-time and we would lose the property of fixed-parameter tractability. Hence, the framework contains the `Solution` and `SolutionContent` classes. Each tuple is associated with an instance of the `SolutionContent` class. This instance represents the partial solution(s) for the respective tuple. The framework provides implementations for different types of `SolutionContent` classes, e.g. for enumeration or counting problems. The main purpose of the `SolutionContent` class is that it does not store the partial solution directly but contains information about how to compute a solution. The `SolutionContent` class must provide three different method implementations that define how a solution is created:

- `calculateUnion()`: This method is called when two tuples within a node coincide after the evaluation of this node.
- `calculateCrossJoin()`: This method defines how solutions are combined within a branch node.
- `calculateAdd()`: This method is called if a value is added to a solution.

After the computation of the tuples and their respective `SolutionContent` instances the tree is traversed in top-down order and the information within the `SolutionContent` instances is evaluated. This results in the computation of all solutions. This approach allows a lazy computation of all solutions and it furthermore guarantees that no partial solutions are computed that are not part of the overall solution.

4.2 Algorithm Implementation

In this section we describe the implementation of our algorithms for admissible semantics. The algorithms for stable and complete semantics follow a similar approach.

Parsing

The input is parsed with help of a lexer (Flex[3]) and a parser (Bison[4]). The lexer is responsible for breaking down the input into a list of tokens that can be handled by the parser. The syntax is as follows:

`arg(a)`	defines the argument `a`
`att(a,b)`	defines an attack relation between the arguments `a` and `b`
`%`	Everything after `%` is a comment and hence ignored

The parser then reads the tokens and calls methods that add the arguments and attack relations to the internal representation of the the argumentation framework. This representation is defined within the `ArgumentationProblem` class. When the SHARP framework calls the `parse()` method the parser and the lexer are executed.

The `preprocess()` method removes any arguments that do not appear in any attack relation. Furthermore, it warns about any argument that is not explicitly defined within the input. Finally, the `buildHypergraphRepresentation()` method converts the arguments and attack relations into the internal data format of the SHARP framework, i.e. a `Hypergraph` instance.

Coloring Representation

Each computed coloring within a node of the tree decomposition is represented as an instance of the `AdmissibleArgumentationTuple`. The header file is depicted in Listing 4.3.

```
1  class AdmissibleArgumentationTuple : public Tuple
2  {
3  public:
4    AdmissibleArgumentationTuple();
5    virtual ~AdmissibleArgumentationTuple();
6
7    ArgumentSet inArguments;
8    ArgumentSet outArguments;
9    ArgumentSet attArguments;
10   ArgumentSet defArguments;
11
12   virtual bool operator<(const Tuple &other) const;
13   virtual bool operator==(const Tuple &other) const;
14   virtual int hash() const;
15 };
```

Listing 4.3: `ArgumentationTuple` header file

[3] `http://dinosaur.compilertools.net/flex/index.html`
[4] `http://dinosaur.compilertools.net/bison/index.html`

The arguments are stored within four disjoint sets, based on their assigned color. An instance of `AdmissibleArgumentationTuple` then represents a valid coloring (and v-coloring) for the current node in the tree decomposition.

`ArgumentSet` is defined as a set of unsigned integers. The `std::set` implementation of C++ allows fast access to the arguments (e.g. finding an argument takes logarithmic time). Furthermore, we override the operations < and =: Within a set of tuples we have that they are primarily ordered by their arguments in `inArguments`. This allows us to apply certain optimizations, for example within the branch node. The decision for using sets within the `AdmissibleArgumentationTuple` class yields towards better performance but it has to be noted that this data structure needs more memory than simple arrays of arguments.

Another advantage is that the choice for sets allows us to use the efficient Standard Template Library (STL) algorithm implementations for the set difference (`set_difference`), set intersection (`set_intersection`) and set union (`set_union`). These algorithms perform at most $2 * (|A| + |B|) - 1$ comparisons of arguments where A and B are the respective sets.

Node Implementation

Each node implementation first calls the `evaluate*Node()` method for its child node(s) (if there are any child nodes). It then computes its set of tuples (or colorings) based on the set(s) of tuples from the child nodes and the introduced or removed arguments. In the following we analyze the implementations of the different node types.

Conflict-Free Sets

The computation of the conflict-free sets of arguments is needed within the leaf and exchange nodes. It is implemented as a recursive call:

```
1 vector<ArgumentSet> conflictFreeSets(ArgumentSet args)
2 {
3   // initialize empty set of arguments that are in the conflict-free set
4   ArgumentSet inArgs = ArgumentSet();
5   // initialize empty set of arguments that are not in the conflict-free set
6   ArgumentSet outArgs = ArgumentSet();
7   return recCFS(inArgs, outArgs, args);
8 }
```

Listing 4.4: Computation of conflict-free sets `conflictFreeSets()` (simplified)

In each call of `recCFS()` one argument `currentArg` of the set `openArgs` is added to the conflict-free set in case it does not attack any argument in `inArgs`, is attacked by `inArgs` or attacks itself (see line 16-23 of Listing 4.5). `recCSF()` is not called if adding `currentArg` to `inArgs` would result in a conflict. Furthermore, `currentArg` can always be added to `outArgs` as `inArgs` remains conflict-free (see line 25-29).

```
vector<ArgumentSet> recCFS(ArgumentSet inArgs, ArgumentSet outArgs, ArgumentSet
    openArgs)
{
  vector<ArgumentSet> result;

  if (open.empty())
    result.insert(inArguments);
  else
  {
    Argument currentArg = openArgs.first();
    ArgumentSet  currentOpen = openArgs.remove(currentArg);

    // check if currentArg and inArgs are conflict-free
    if (!argumentAttacksSet(currentArg, inArgs) && !argumentIsAttackedBySet(
        currentArg, inArgs) && !argumentAttacksArgument(currentArg, currentArg))
    {
      // Argument is added to inArgs
      ArgumentSet newIn = inArgs.insert(currentArg);
      vector<ArgumentSet> newCFS = recCSF(newIn, outArgs, currentOpen);
      result.insert(newCFS);
    }

    // Argument is added to outArgs
    ArgumentSet outNew = outArgs.insert(currentArg);
    vector<ArgumentSet> newCFS = recCSF(inArgs, outNew, currentOpen);
    result.insert(newCFS);
  }
  return result;
}
```

Listing 4.5: Recursive call `recCFS()` (simplified)

Leaf Node

The leaf node is defined for normalized tree decompositions. It has no child nodes.

1. Call `conflictFreeSets()` for the arguments within the leaf node.
2. For each conflict-free set , `cfset`, do the following:
   1) Create a new tuple and set the `inArguments` to `cfset`.
   2) All arguments that are attacked by `inArguments` are added to `defArguments`.
   3) All arguments that are not attacked by `inArguments` but attack `inArguments` are added to `attArguments`.
   4) The other arguments are added to `outArguments`.
   5) Create a new leaf solution that contains the `inArguments`.
3. Return the set of tuples for the leaf node.

Introduction Node

It is defined for normalized tree decompositions where exactly one argument is introduced in an introduction node.

1. Call the `evaluateNode()` method for the child node. This returns the set of tuples from the child node, `childTuples`.
2. For each `childTuple` in `childTuples` do the following:
   1) Assume that the introduced argument, `introducedArg`, is not added to the arguments `inArguments` of the new tuple. Copy the sets of arguments of `childTuple` to the sets of `newTuple`. Add `introducedArg` to the set `defArguments`, `attArguments` or `outArguments` of `newTuple`, based on its attack relations to `inArguments` (see Definition 3.10, $C+a$). Use the `calculateUnion()` method to compute the solution of the new tuple based on the solution of the child tuple.
   2) If `introducedArg` and `inArguments` is conflict-free, create an additional tuple, `additionalTuple`. Compute the sets of arguments for `additionalTuple` based on Definition 3.10, $C\dot{+}a$. Use the `calculateAdd()` method to add the introduced argument to the solution of the child tuple.
3. Delete the tuples of the the child node.
4. Return the set of tuples for the introduction node.

Removal Node

The removal node is defined for normalized decompositions. Exactly one argument is removed.

1. Call the `evaluateNode()` method for the child node. This returns a set of all tuples from the child node, `childTuples`.
2. For each `childTuple` in `childTuples` do the following:
   1) If the removed argument, `removedArg`, is in `attArguments` of `childTuple`, do nothing.
   2) Otherwise, create a new tuple for the current node and copy the sets of arguments for the child node, without the `removedArg`. Use the `calculateUnion()` method to compute the solution of the new tuple based on the solution of the child tuple.
3. Delete the tuples of the the child node.
4. Return the set of tuples for the removal node.

Exchange Node

Compared to the leaf, introduction and removal nodes of normalized tree decompositions the exchange node is defined for semi-normalized tree decompositions where distinct sets of arguments are removed and introduced.

1. Call the `evaluateNode()` method for the child node. This returns a set of all tuples from the child node, `childTuples`. It the exchange node does not have a child node, create an empty tuple and add it to `childTuples`.

2. Compute the conflict-free sets of all introduced arguments, `cfsets`.
3. For each `childTuple` in `childTuples` do the following:
   1) If any of the removed arguments, `removedArgs`, is in `attArguments` of the child tuple, delete the complete tuple.
   2) Otherwise, remove the `removedArgs` from the sets of arguments of `childTuple` and do the following for each conflict-free set, `cfset`:
      (1) If the union of `inArguments` in `childTuple` and `cfset` is conflict-free, create a new tuple where `inArguments` is the union of both sets. Furthermore, compute sets of arguments in the new tuple based on Definition 3.48.
      (2) Use the `calculateAdd()` method to add the introduced arguments to the solution of the child tuple and the `calculateUnion()` method if no argument was added to `inArguments`.
4. Delete the tuples of the the child node.
5. Return the set of tuples for the exchange node.

Within the exchange node, several arguments can be removed within one loop over the `childTuples`. If at least one removed argument is contained in `attArguments` of a child tuple the complete tuple can be deleted immediately. Furthermore, the conflict-free sets of introduced arguments only have to be computed once. Then, the `childTuples` can be joined with the conflict-free sets if their union if `inArguments` and `cfset` is again conflict-free. As the exchange node is defined on sets of arguments the efficient C++ implementations for set union, difference and intersection can be used.

Branch Node

The branch node is defined for both normalized and semi-normalized tree decompositions. Hence, the two implementations for admissible semantics share the same code that executes the following steps:

1. Call the `evaluateNode()` method for the two child nodes. This returns two tuple sets, `leftTuples` and `rightTuples`.
2. Initialize two iterators, `leftIt` and `rightIt` with the first tuples, `leftTuple` and `rightTuple`, from the tuple sets.
3. Until the end of one set is reached, do the following:
   1) Compare the set `inArguments` of `leftTuple` with the set `inArguments` of `rightTuple`.
   2) If one set is of `inArguments` is smaller than the other one, increase the iterator `leftIt` or `rightIt` of the smaller tuple and assign the new value to `leftTuple` or `rightTuple`.

3) If `inArguments` of both tuples is equal we have to compute a new tuple and have to join the (partial) solutions of the two tuples. The `inArguments` of the new tuple are the same as the `inArguments` of the child tuples. If an argument is in `defArguments` of one child tuple it is in `defArguments` of the new tuple. If an argument is in `outArguments` of both child tuples it is in `outArguments` of the new tuple. Otherwise the argument is in `attArguments` of the new tuple. This corresponds to Definition 3.12 for the computation of v-colorings in a branch node for admissible semantics.

4) Finally, combine the (partial) solutions of the child nodes with the `crossJoin` operation and add the new tuple to the tuple set of the current node.

5) Increase the iterator `rightIt`.

4. Delete the tuples of the the child nodes.

5. Return the set of tuples for the branch node.

This algorithm takes advantage of the fact that the tuple sets are ordered primarily by their `inArguments`. Then, it is not necessary to compare each tuple of `leftTuples` with each tuple of `rightTuples` (which would result in $|\texttt{leftTuples}| * |\texttt{rightTuples}|$ comparisons in total).

4.3 The dynPARTIX Project

The implementation of the algorithms for admissible, stable and complete semantics as defined in the previous chapter and described in this chapter is included in the dynPARTIX[5] project. dynPARTIX (Dynamic Programming Argumentation Reasoning Tool) is based on the SHARP framework. The software can currently be called with the following parameters:

```
dynpartix [-b] [-t] [-d] [-r <seed>] [-f <file>]
  [-n <normalize>] [-s <semantics>]
  [--enum[=number] | --count | --cred <arg> | --skept <arg>]
```

`-b`	Prints benchmark information.
`-t`	Only perform the tree decomposition step. The width of the tree decomposition is printed.
`-d`	Prints a comma-separated list consisting of filename, seed, time and solution.
`-r <seed>`	Initialize the random number generator with `<seed>`.
`-f <file>`	Specify an input file `<file>` containing the AF.
`-n <normalize>`	Specify the tree decomposition normalization type, one of {norm (default), semi}.
`-s <semantics>`	Specify the semantics, one of {admissible (default), stable, complete}.
`--enum[=number]`	Prints an enumeration of all solutions. Optionally, `number` limits the number of printed solutions.
`--count`	Prints the number of solutions.
`--cred <arg>`	Checks if `<arg>` is credulously accepted or not.
`--skept <arg>`	Checks if `<arg>` is skeptically accepted or not.

Table 4.1: dynPARTIX Call Options

The listing below shows the input for our example AF (see Figure 2.1).

```
1  % define arguments
2  arg(a).
3  arg(b).
4  arg(c).
5  arg(d).
6  arg(e).
7  arg(f).
8  arg(g).
9
10
11
12
13
14 % define attack relations
15 att(a,b).
16 att(b,a).
17 att(a,c).
18 att(b,c).
19 att(c,d).
20 att(d,e).
21 att(e,b).
22 att(f,a).
23 att(e,g).
24 att(g,g).
```

[5] http://www.dbai.tuwien.ac.at/research/project/argumentation/dynpartix/

In here we now present some examples for program calls. The examples are based on the input file `example.graph` that contains the argumentation framework of our running example. A simple program call for the computation of admissible extensions (on normalized tree decompositions) is as follows:

```
dynpartix -f example.graph
 Solutions: 5
 {{},{e,f,c},{f},{f,b,d},{b,d}}
```

Next, if we want to benchmark the computation of admissible extensions on semi-normalized tree decompositions we can call dynPARTIX outline in the next listing. Besides the extensions this gives us the used seed for the tree decomposition (for a given seed, specified with the option `-r <seed>` we always obtain the same tree decomposition. Furthermore the time for the evaluation of the solutions and the overall run-time is given.

```
dynpartix -f example.graph -s admissible -n semi -b
 Using seed: 1328877763
 Calculating solution content... done! (took 0 seconds)
 ----------------------------------------
 Overall time: 0 seconds
 Solutions: 5
 {{},{e,f,c},{f},{f,b,d},{b,d}}
```

In order to obtain the width of the tree decomposition we can use the parameter `-t`. Then, only the tree decomposition is generated and the width is returned. Thus we know that the largest bag of the decomposition contains $width + 1$ arguments.

```
dynpartix -f example.graph -t
 Width: 2
```

Finally, we give examples for program calls that compute the overall number of extensions and check for credulous as well as skeptical acceptance.

```
dynpartix -f example.graph -s stable --count
 Solutions: 1
dynpartix -f example.graph -s stable --skept b
 NO
dynpartix -f example.graph -s stable --cred c
 YES
dynpartix -f example.graph -s admissible --skept c
 NO
dynpartix -f example.graph -s admissible --cred c
 YES
```

Note that for admissible semantics the skeptical acceptance of an argument is always answered with `NO`.

CHAPTER 5

Experimental Results

In this chapter we compare the run-time of algorithms on normalized tree decompositions and semi-normalized tree decompositions. We run the algorithms for admissible semantics and count the overall number of admissible extensions for our test instances. Besides the overall run-time we are interested in a direct comparison between leaf, introduction and removal nodes of normalized decompositions and exchange nodes of semi-normalized decompositions.

In Section 5.1 we describe the test environment. Furthermore we outline the methodological approach for the generation of reliable test data: Although we can generate test data of fixed tree width and a fixed number of nodes we do not necessarily obtain tree decompositions of fixed width (where, in the best case the tree width of the input instance graph corresponds to the width of the tree decomposition). This is due to the fact that it is computationally hard to generate optimal tree decompositions. The implementation of the tree decomposition makes use of a heuristics and therefore returns tree decomposition instances of variable width for a fixed tree width.

In Section 5.2 we describe two types of test instances that are used for the evaluation of our algorithms, namely grid-based instances and clique-based instances. *Grid*-based instances, where every argument in the graph can be connected to all of its neighbors, have an upper bound for the tree width. *Clique*-based instances consist of several cliques (every node is connected with every other node within the clique). They have a fixed tree-width which corresponds to the number of nodes within a clique, minus one.

In Section 5.3 we compare the normalized and semi-normalized implementation on basis of the two different test instance types. We vary the (theoretical, maximal) tree width of grid-based instances and compute the run-time for tree-decompositions of a certain width (or a range of width values). Furthermore we evaluate the algorithms on clique-based instances for a certain tree width (which corresponds to the with of the tree decomposition).

Finally, in Section 5.4, we summarize our experimental results. We analyze the benefits of algorithms based on semi-normalized tree decompositions compared to those defined on normalized tree decompositions. Furthermore we identify input instances where the benefit may not be significant.

5.1 Test Setup and Approach

The general approach for the generation of reliable test instances and the automatic generation of benchmark information is as follows:

1. Based on the instance type (grid or clique) initial test instances are generated. These instances have a fixed number of nodes (arguments) and have a fixed (maximal) tree width.
2. In a first run, several tree decompositions for every initial test instance are computed. The reason for this is that the tree decomposition step is based on a heuristics. Hence, the width of the decomposition may vary. All tree decompositions with a fixed width (or a small range of widths) are used later on as the basis for our benchmarks.
3. The normalized and semi-normalized implementation for admissible semantics is executed on the same tree decomposition instances. We compute the overall run-time, the preparation time, the time that is spent within the branch node and the time within the other nodes of the decomposition (either exchange or leaf, introduction and removal nodes). For every input instance size we guarantee that at least 20 different tree decompositions are executed.
4. We compute the average run-time for each instance type, width (or range of widths) and number of nodes.

We are interested in the execution time of the algorithms. As we want to keep side effects minimal during the execution of our benchmark tests we measure the CPU-time, not the elapsed real time. Although not perfect, the CPU-time supplies us with sufficiently accurate run-time information.

The following metrics are measured for each input instance:

- The *total time* gives the overall execution time of the algorithm.
- The *preparation time* includes the time that is needed for parsing the input data. Furthermore, it includes the computation of the normalized or semi-normalized tree decomposition.
- The *branch node time* is the time that is spent within all branch nodes.
- The *exchange node time* gives the overall time that is needed for the computation of exchange nodes in semi-normalized tree decompositions. The *leaf, intro, rem node time* gives the overall execution time within leaf, introduction and removal nodes of normalized tree decompositions.

The *exchange node time* and the *leaf, intro, rem node time* are the most relevant benchmark metrics as they allow a direct and reliable comparison of normalized and semi-normalized tree decompositions.

The benchmark tests are executed on an openSUSE 11.4 machine with two Intel Xeon CPUs (E5345, QuadCore, 2.33 GHz). The current implementation is single-threaded and thus only makes use of one core at a time.

5.2 Test Instances

Grid Structure

Grid-based test instances are defined on basis of grid graphs: A grid graph consists of a matrix of $n \times m$ vertices. Each vertex can be connected with its neighbors. In here, we use 8-grid graphs: Then, a vertex is connoted with all eight vertical and horizontal as well as diagonal neighbors. As we have directed edges that represent the attack relations each argument attacks all neighbors.

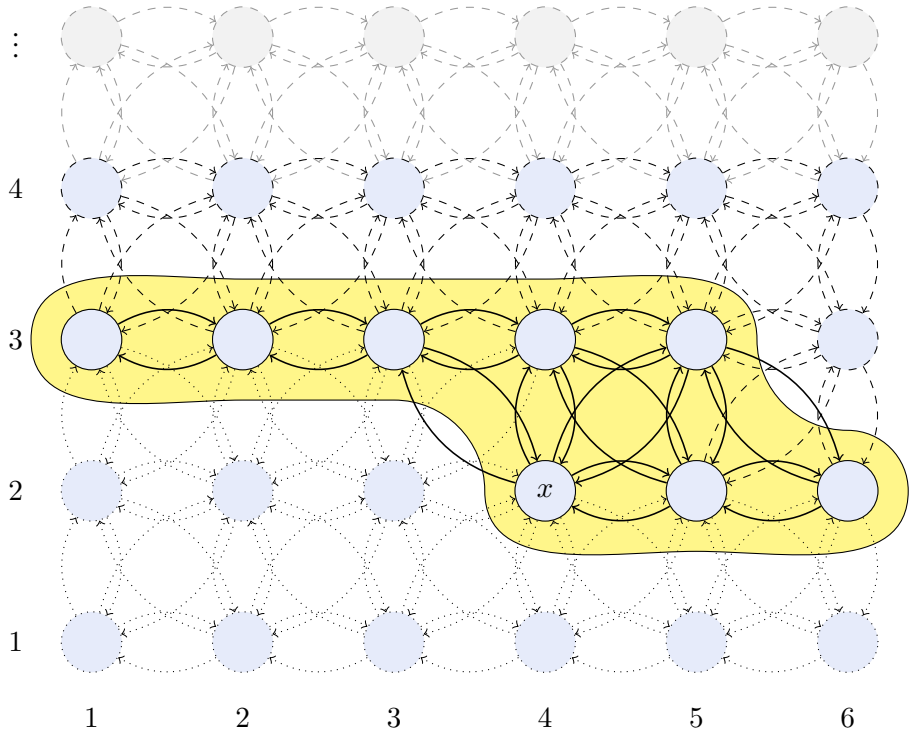


Figure 5.1: 8-Grid, $(6 \times m)$, Tree Width 7

An $(n \times m)$ grid where $n = 6$ is depicted in Figure 5.1. For any $m \geq n$, the tree width of the graph is $n+1$. In our case, the tree width for $m \geq 6$ is 7. This is due to the properties of tree decompositions. We have that arguments that attack each other have to be contained in a bag. Furthermore an argument that is removed can never be introduced anywhere above in the tree decomposition. Finally, every argument has to appear in at least one bag. The dotted parts of Figure 5.1 represent arguments that where already removed, $X_{>t}$. Arguments that are encircled with dashed lines were not yet introduced. Finally, the yellow part shows the arguments that are in the current bag X_t. In an optimal tree decomposition we have that at least $n+2$ arguments have to appear together in a node of the tree decomposition. Consider the argument x in Figure 5.1. It is the only argument that can be removed after X_t as it is the only argument where all attack relations where either considered in $X_{>t}$ or X_t. If we would introduce a new argument this

would result in a larger bag size. In other words, for an 8-grid $(n \times m)$ and $m \geq n$ we have that the minimal bag size is $n+2$. The tree width is $n+1$. If we obtain an optimal tree decomposition the width corresponds to the tree width.

For our benchmarks we generate the grid-based instances on basis of a certain edge probability p. p defines the probability of an attack relation between two nodes (arguments) appearing in the test instance. Then, $n + 1$ (for $m \geq n$) is an upper bound for the tree width.

Clique Structure

Clique-based test instances consist of one or more independent cliques. In general, a clique is an undirected graph where all vertices of the graph are connected. In our case, as we have directed attack relations, we have that each argument attacks all other arguments of the clique. The tree width of a clique with x vertices is $x - 1$ as all arguments attack each other. Hence they have to appear together in a bag of the tree decomposition.

Then, our test instances consist of n independent cliques (there is no attack relation between them) of size $tw + 1$. An example instance is given in Figure 5.2. Here, every clique consists of 6 arguments. hence, the tree width is 5. The yellow parts mark arguments that have to appear together in a bag of the tree decomposition.

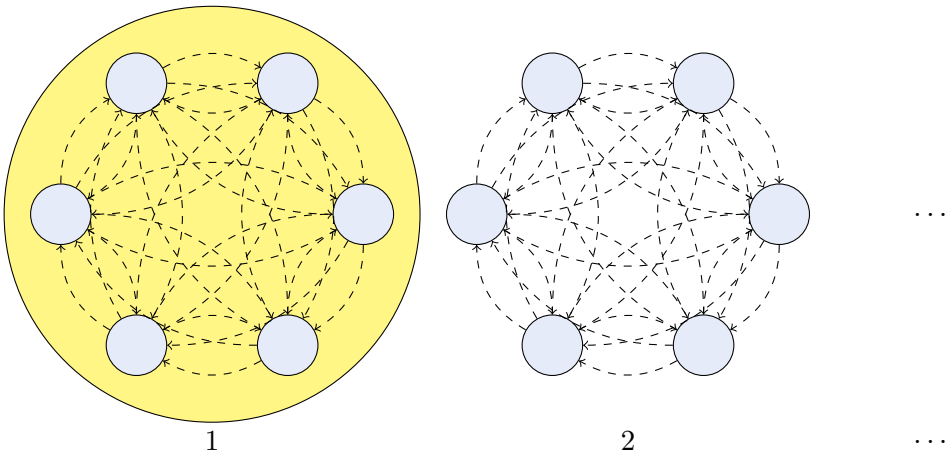


Figure 5.2: Clique Structure, Tree Width 5

5.3 Normalized vs. Semi-Normalized Algorithms

Grid-Based Instances

In this section we compare the algorithm for semi-normalized tree decompositions to that for normalized tree decompositions. We analyze the run-time performance on basis of instances with different width, size (number of arguments) and edge probability.

Benchmark for 8-Grid Instances, Width 4 and Edge Probability 1

In this case the width is relatively small. The tree width of all instances is 4 and we obtained a width of 4 for all input instances. In Figure 5.3 the results for instances with 600 to 9600 nodes are presented.

On average, the preparation of the tree decompositions took about 73 percent of the total run-time (for both algorithms). Therefore, with 3 percent of performance gain on average the semi-normalized implementation is only slightly faster than the normalized implementation. But when we analyze the average performance gain of the exchange node implementation to the leaf, introduction and removal node implementation we have that the exchange node needs about 12.5 percent less time. Another interesting result is that the branch node needs almost no time.

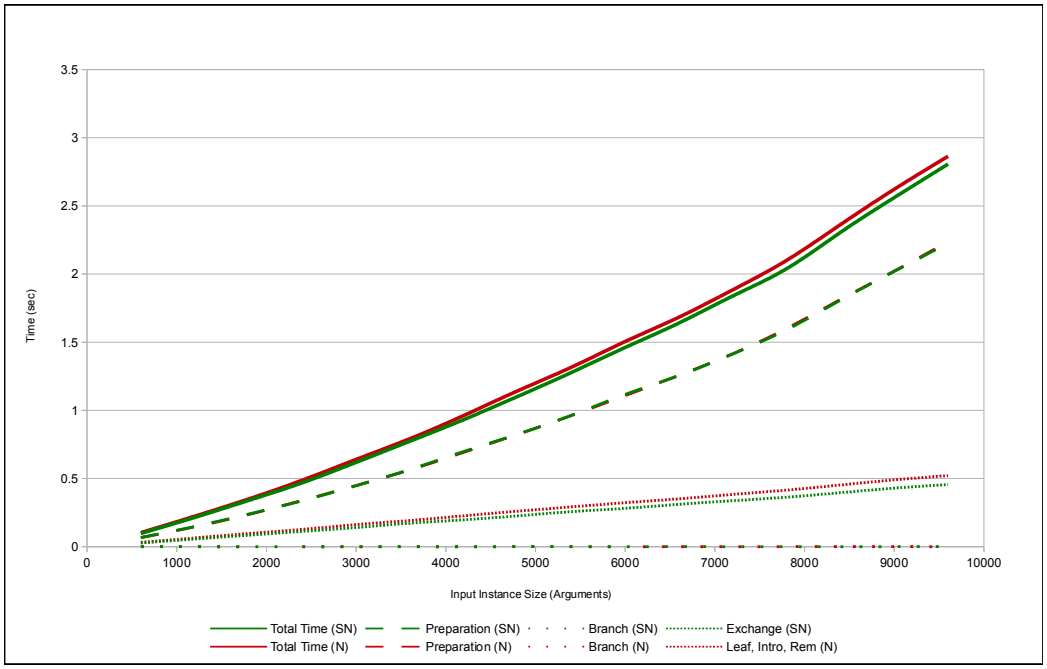


Figure 5.3: Benchmark Results for $(3, m)$ 8-Grid, Width 4, Probability 1

Benchmark for 8-Grid Instances, Width 13-15 and Edge Probability 0.6

The instances for this benchmark test where generated from graphs with a theoretical (maximal) tree width of 7. Hence, the graph consists of a $6 \times m$ matrix. For a range of 200 to 3200 arguments we compare the run-time performance. The results are depicted in Figure 5.4.

For these instances the preparation of the tree decompositions needed almost no time. Not surprisingly, time for the computation within branch nodes of normalized and semi-normalized tree decompositions is almost identical. This is due to the fact that both implementations share the same branch node code. Furthermore the branch node takes about 60 percent of the overall computation time.

This test case shows that a semi-normalized implementation can outperform the normalized implementation: On average, the semi-normalized implementation of exchange nodes is 51 percent faster than the implementation of the respective nodes for semi-normalization. Furthermore, the overall runtime increases by 20 percent.

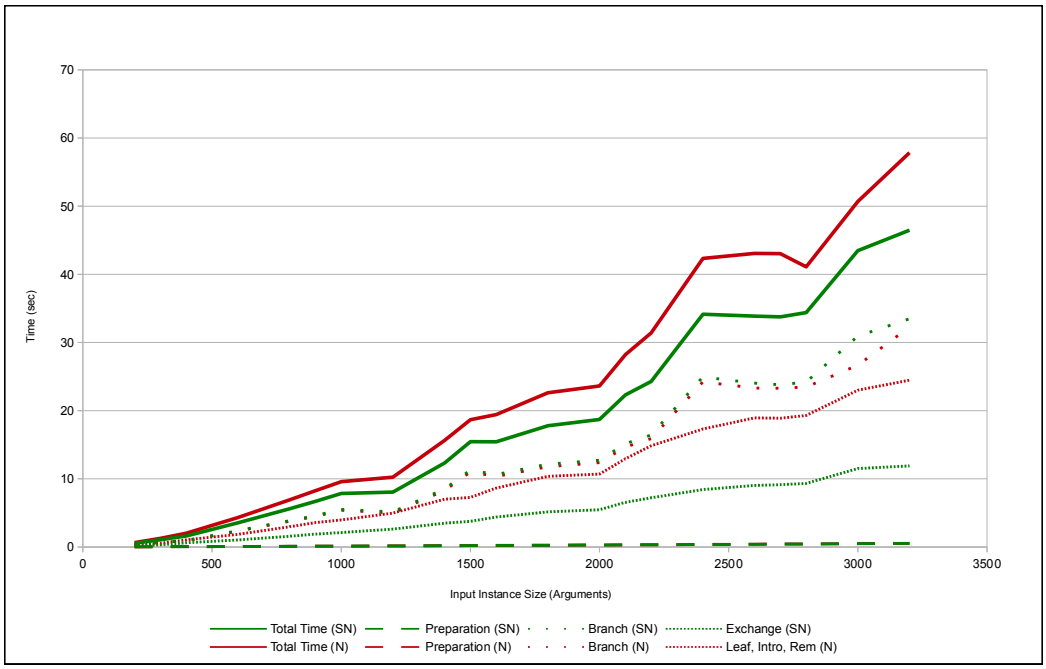


Figure 5.4: Benchmark Results for $(6, m)$ 8-Grid, Width 13-15, Probability 0.6

Benchmark for 8-Grid Instances, Width 10-12 and Edge Probability 0.3

For this benchmark test we generated instances of theoretical (maximal) tree width 9 and an edge probability of 30 percent. The tree decompositions we use inhere all have a width between 10 and 12. The number of arguments ranges from 600 to 6600. The results can be seen in Figure 5.5.

As in the previous benchmark the costs for the preparation of the tree decompositions is negligible. The branch node of both implementations is responsible for about 70 percent of the total run-time.

On average, the exchange nodes of the semi-normalized implementation need about 40 percent less time than the leaf, introduction and removal nodes of the normalized implementation. Furthermore, the semi-normalized implementation is about 11.5 percent faster than the normalized implementation.

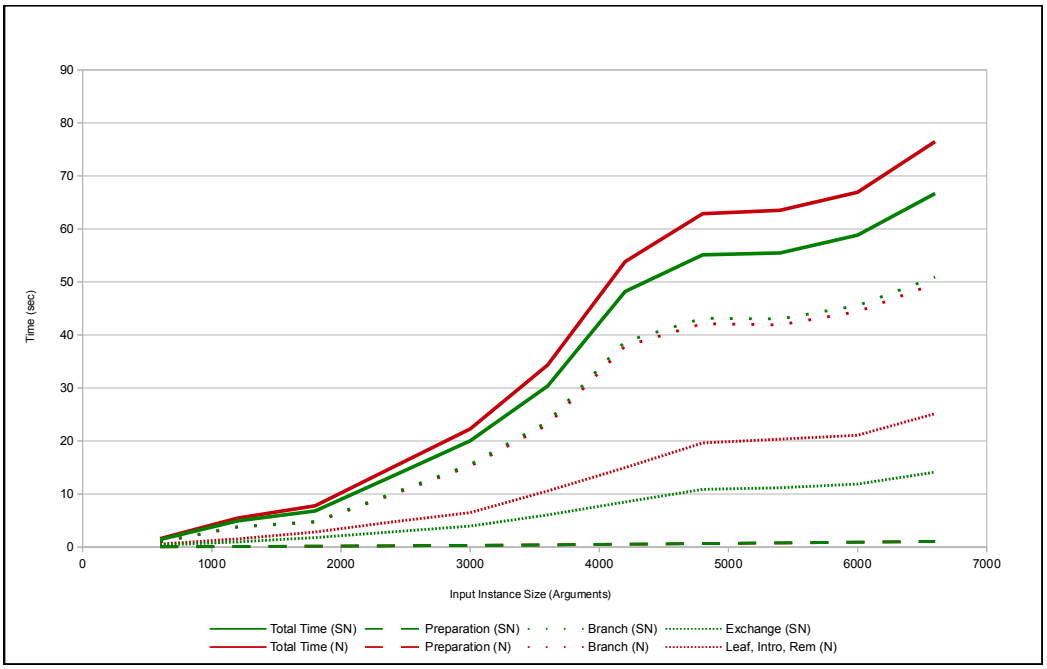


Figure 5.5: Benchmark Results for $(8, m)$ 8-Grid, Width 10-12, Probability 0.3

Clique-Based Instances

In this section we compare the implementation of the semi-normalized algorithm to that of the normalized algorithm for admissible semantics. For the clique based test instances we have that the tree decomposition computation always returns decompositions where the width is equal to the tree width of the original graph.

Benchmark for Clique Instances, Tree Width 100

The test instances all have a width of 100 (and a tree width of 100 for the original graph). We test the performance on instances where the number of cliques ranges from 1 to 10. As there are 101 arguments in each clique we have that this corresponds to an overall number of 101 to 1010 arguments.

The results are depicted in Figure 5.6. Most notably, the preparation needs by far the most of the run-time, i.e. about 90 percent of the run-time of the normalized implementation and about 93 percent of the run-time of the semi-normalized implementation.

Hence, although we have a performance gain of 24.5 percent when comparing the exchange node implementation to the implementation of leaf, introduction and removal node the overall run-time only decreases by about 2.5 percent.

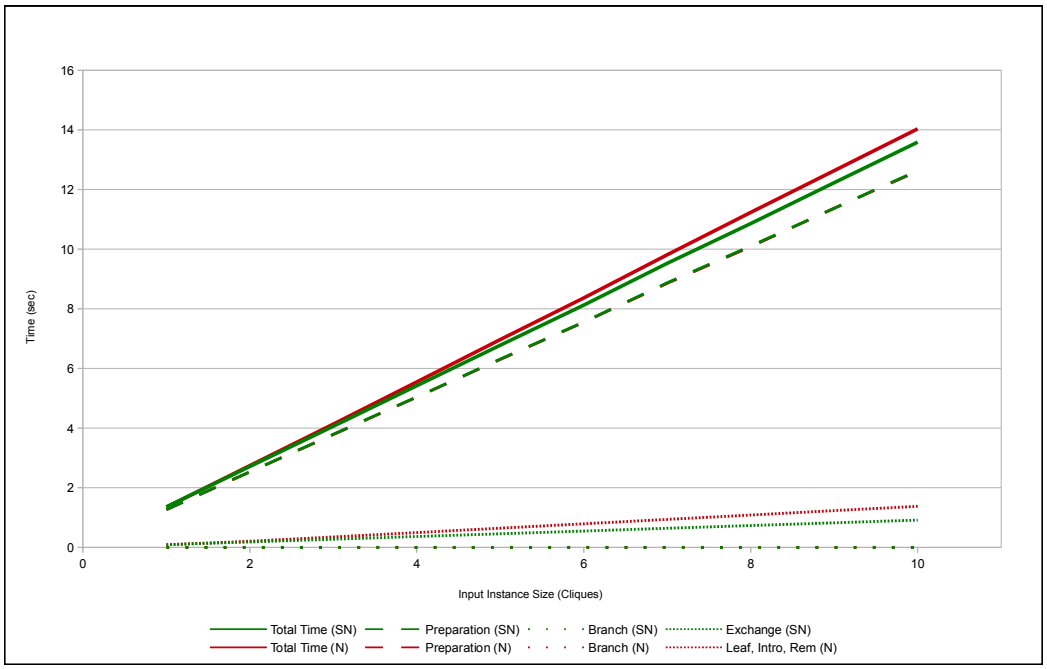


Figure 5.6: Benchmark Results for Cliques, Width 100, Tree Width 100

5.4 Analysis of Benchmarks

The benchmarks show that the implementation for admissible semantics on semi-normalized tree decompositions performs better than the implementation on normalized tree decompositions. A reason for this is that the semi-normalized tree decompositions contain less nodes. Furthermore it is possible to implement optimizations for removed or introduced sets of arguments within the exchange node.

The overall performance gain heavily depends on the input instance: For clique-based instances the preparation consumes most of the overall execution time. A reason for this is probably the extremely high number of edges: In a clique with x vertices every vertex is connected to all other $x - 1$ vertices. As we have directed edges it contains $x(x - 1)$ edges. This not only results in large input files but also the tree decomposition generation takes a long time.

Furthermore the computation within the branch node is extremely time consuming in case the number of partial results within the child nodes is large. All partial results of the two child nodes have to be combined.

On the other hand, if we compare the exchange node implementation directly to the implementation of leaf, introduction and removal node we have a huge performance gain. For our grid-based instances with width 13 to 15 and an edge probability of 0.6 the exchange node computation is 51 percent faster. For instances with width 10 to 12 and an edge probability of 0.3 the exchange node performs 40 percent better.

Another interesting observation is that the preparation of the tree decompositions needs almost the same time for both algorithms although the normalized tree decomposition consists of much more nodes. As nodes can be introduced in time $O(n)$ the performance difference is not notable.

CHAPTER 6

Conclusion and Future Work

Conclusion

In this thesis we presented three novel algorithms for abstract argumentation frameworks that are based on tree decompositions.

For our algorithms we made use of the properties of tree decompositions. Each argument of the original argumentation framework appears in at least one bag of the decomposition. Furthermore, if there exists an attack relation between arguments they are contained together in at least one bag. Additionally, bags containing the same argument are connected upwards the tree decomposition. Then, we defined B-restricted sets where B contains all arguments that were already completely considered in the sub-tree of the respective node and that fulfill the properties of the respective semantics. Furthermore we defined valid colorings on basis of the arguments in the current bag of a node and the arguments in the B-restricted sets. In the root node we obtained that the B-restricted sets correspond to the extensions of the AF for a given semantics. For complete semantics we additionally introduced the concept of labelings. This allowed us to characterize B-restricted labelings where we do not only have information about the arguments in the extensions of the (sub)-frameworks but also about arguments that are not in the extensions.

As the valid colorings are defined on basis of the B-restricted sets (or labelings) for a node we defined, towards fixed-parameter tractability, v-colorings that are defined solely on basis of the arguments in the current bag and the colorings of the child node(s). For the different node types (leaf, introduction, removal, branch and exchange node) we proved that the v-colorings correspond to valid colorings. Thus we did not have to compute the extensions of the sub-frameworks explicitly. With this approach we achieved that the computational costs for the computation of extensions is bound by the tree-with of the original argumentation framework. Hence, we can compute extensions in

$$f(k) \cdot n^{O(1)}$$

time where k is the tree-width and n is the size of the AF.

Furthermore we proved the correctness of our algorithm for admissible semantics on semi-normalized tree decompositions by showing that the computation of v-colorings in a semi-normalized tree decomposition corresponds to the computation over introduction and removal nodes in a normalized tree decomposition.

We implemented two algorithms for stable and complete semantics for normalized tree decompositions and the algorithm for admissible semantics on semi-normalized tree decompositions. The implementation is included in the dynPARTIX[1] project.

Furthermore we compared the already-existing algorithm for admissible semantics on normalized tree decompositions to our novel algorithm for semi-normalized tree decompositions. It turned out the semi-normalized decomposition performed better in all test cases. The overall run-time, however, only decreased significantly if the preparation time for the tree decomposition and the evaluation of the branch nodes did not take too much time. This was the case for our grid-based instances with width 13-15 (edge probability 0.6) and width 10-12 (edge probability 0.3). Our benchmark tests for clique-based instances showed that the preparation of the tree decomposition took a lot of time. One reason was probably the fact that these instances consist of many attack relations and the algorithm for the generation of tree decompositions has to consider all of those.

Future Work

Our benchmark results show that the implementation on semi-normalized tree decompositions with an exchange node where several arguments can be removed and introduced performs better than the normalized tree decomposition with separate nodes for each removed or introduced argument. Thus, it is expected that the development of algorithms for stable and complete semantics on semi-normalized tree decompositions results in better performance as well.

On the other hand, considering our benchmark results, we also have to improve the performance of the other components. In some cases the generation of the tree decomposition takes about 90 percent of the overall run-time. It is therefore necessary to investigate how the heuristics for the generation of tree decompositions can be improved. Furthermore the applied heuristics provide us with tree decompositions where the width is much higher than the theoretical tree-with of the problem instance. It is expected that a lower width (which, in the best case, corresponds to the tree-width of the input instance) results in a far better performance. Smarter heuristics could, of course, also need more run-time. This has to be investigated.

Furthermore, it may be possible to improve the overall performance of the branch nodes. In this thesis we restricted ourselves to normalized and semi-normalized tree decompositions. An evaluation of algorithms on tree decompositions without normalization would be interesting. This, of course, results in more complicated definitions and proofs of the respective algorithms.

In this thesis we did not give a detailed complexity analysis for the decision problems of credulous and skeptical acceptance. The complexity-theoretic results state the the problems are fixed-parameter tractable but as the real run-time of algorithms may be hidden by the big-O-notion it may be of great interest to narrow down the theoretical run-time of the algorithms.

Additionally, there exist many more semantics for abstract argumentation. It would be of great interest to develop algorithms for further semantics such as naive, stage or ideal semantics that are based on fixed-parameter tractability.

[1] `http://www.dbai.tuwien.ac.at/proj/argumentation/dynpartix/`

List of Figures

Bibliography

Amgoud, L. and Devred, C. (2011). Argumentation frameworks as constraint satisfaction problems. In *Proceedings of the 5th international conference on Scalable uncertainty management*, SUM'11, pages 110–122. Springer Berlin / Heidelberg.

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

Baroni, P. and Giacomin, M. (2003). Solving semantic problems with odd-length cycles in argumentation. In Nielsen, T. and Zhang, N., editors, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, volume 2711 of *Lecture Notes in Computer Science*, pages 440–451. Springer Berlin / Heidelberg.

Baroni, P. and Giacomin, M. (2007). On principle-based evaluation of extension-based argumentation semantics. *Artificial Intelligence*, 171:675–700.

Baroni, P. and Giacomin, M. (2008). Resolution-based argumentation semantics. In *Proceedings of the 2008 conference on Computational Models of Argument: Proceedings of COMMA 2008*, pages 25–36, Amsterdam, The Netherlands. IOS Press.

Baroni, P. and Giacomin, M. (2009). Semantics of abstract argument systems. In Simari, G. and Rahwan, I., editors, *Argumentation in Artificial Intelligence*, pages 25–44. Springer US.

Bench-Capon, T. and Dunne, P. E. (2007). Argumentation in artificial intelligence. *Artificial Intelligence*, 171(10-15):619 – 641.

Besnard, P. and Doutre, S. (2004). Checking the acceptability of a set of arguments. In Delgrande, J. P. and Schaub, T., editors, *10th International Workshop on Non-Monotonic Reasoning (NMR 2004)*, pages 59–64.

Besnard, P. and Hunter, A. (2001). A logic-based theory of deductive arguments. *Artificial Intelligence*, 128(1-2):203 – 235.

Bodlaender, H. (1997). Treewidth: Algorithmic techniques and results. In Prívara, I. and Ružicka, P., editors, *Mathematical Foundations of Computer Science 1997*, volume 1295 of *Lecture Notes in Computer Science*, pages 19–36. Springer Berlin / Heidelberg.

Bodlaender, H. L. (1993). A tourist guide through treewidth. *Acta Cybernetica*, 11:1–23.

Bodlaender, H. L. and Koster, A. M. (2010). Treewidth computations I. Upper bounds. *Information and Computation*, 208(3):259 – 275.

Bondarenko, A., Dung, P., Kowalski, R., and Toni, F. (1997). An abstract, argumentation-theoretic approach to default reasoning. *Artificial Intelligence*, 93(1-2):63 – 101.

Caminada, M. (2006). Semi-stable semantics. In *Proceedings of the 2006 conference on Computational Models of Argument: Proceedings of COMMA 2006*, pages 121–130, Amsterdam, The Netherlands. IOS Press.

Caminada, M. (2007). Comparing two unique extension semantics for formal argumentation: ideal and eager. In *Proceedings of the 19th Belgian-Dutch Conference on Artificial Intelligence*, pages 81–87. BNAIC 2007.

Caminada, M. and Gabbay, D. (2009). A logical account of formal argumentation. *Studia Logica*, 93:109–145.

Coste-Marquis, S., Devred, C., and Marquis, P. (2005). Symmetric argumentation frameworks. In Godo, L., editor, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, volume 3571 of *Lecture Notes in Computer Science*, pages 471–471. Springer Berlin / Heidelberg.

Courcelle, B. (1990). Graph rewriting: An algebraic and logic approach. In van Leeuwen, J., editor, *Handbook of theoretical computer science (vol. B)*, pages 193–242. MIT Press, Cambridge, MA, USA.

Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.

Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B., Musliu, N., and Samer, M. (2008). Heuristic methods for hypertree decomposition. In Gelbukh, A. and Morales, E., editors, *MICAI 2008: Advances in Artificial Intelligence*, volume 5317 of *Lecture Notes in Computer Science*, pages 1–11. Springer Berlin / Heidelberg.

Dimopoulos, Y. and Magirou, V. (1994). A graph-theoretic approach to default logic. *Information and Computation*, 112(2):239 – 256.

Dimopoulos, Y. and Torres, A. (1996). Graph theoretical structures in logic programs and default theories. *Theoretical Computer Science*, 170(1-2):209 – 244.

Dorn, F. and Telle, J. A. (2009). Semi-nice tree-decompositions: The best of branchwidth, treewidth and pathwidth with one algorithm. *Discrete Applied Mathematics*, 157(12):2737 – 2746.

Doutre, S. and Mengin, J. (2004). On sceptical versus credulous acceptance for abstract argument systems. In Alferes, J. and Leite, J., editors, *Logics in Artificial Intelligence*, volume 3229 of *Lecture Notes in Computer Science*, pages 462–473. Springer Berlin / Heidelberg.

Downey, R. G. and Fellows, M. R. (1995). Fixed-parameter tractability and completeness I: Basic results. *SIAM J. Comput.*, 24:873–921.

Dung, P., Mancarella, P., and Toni, F. (2007). Computing ideal sceptical argumentation. *Artificial Intelligence*, 171(10-15):642 – 674.

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

Dunne, P. E. (2007). Computational properties of argument systems satisfying graph-theoretic constraints. *Artificial Intelligence*, 171(10-15):701 – 729.

Dunne, P. E. and Bench-Capon, T. (2002). Coherence in finite argument systems. *Artificial Intelligence*, 141(1-2):187 – 203.

Dunne, P. E. and Wooldridge, M. (2009). Complexity of abstract argumentation. In Simari, G. and Rahwan, I., editors, *Argumentation in Artificial Intelligence*, pages 85–104. Springer US.

Dvořák, W., Pichler, R., and Woltran, S. (2010a). Towards fixed-parameter tractable algorithms for argumentation. In Lin, F., Sattler, U., and Truszczynski, M., editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010*, pages 112 – 122. AAAI Press.

Dvořák, W., Szeider, S., and Woltran, S. (2010b). Reasoning in argumentation frameworks of bounded clique-width. In Baroni, P., Cerutti, F., Giacomin, M., and Simari, G. R., editors, *Proceedings of the 2010 conference on Computational Models of Argument: Proceedings of COMMA 2010*, volume 216 of *FAIA*, pages 219–230. IOS Press.

Dvořák, W., Morak, M., Nopp, C., and Woltran, S. (2011). dynPARTIX - A dynamic programming reasoner for abstract argumentation. *CoRR*, abs/1108.4804.

Dvořák, W. and Woltran, S. (2010). Complexity of semi-stable and stage semantics in argumentation frameworks. *Information Processing Letters*, 110(11):425 – 430.

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

Egly, U. and Woltran, S. (2006). Reasoning in argumentation frameworks using quantified boolean formulas. In Dunne, P. E. and Bench-Capon, T. J. M., editors, *Proceedings of the 2006 conference on Computational Models of Argument: Proceedings of COMMA 2006*, volume 144 of *FAIA*, pages 133–144. IOS Press.

García, A. J. and Simari, G. R. (2004). Defeasible logic programming: an argumentative approach. *Theory Pract. Log. Program.*, 4:95–138.

Kloks, T. (1994). *Treewidth: computations and approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg.

Lifschitz, V. (1996). *Foundations of logic programming*, pages 69–127. Center for the Study of Language and Information, Stanford, CA, USA.

Modgil, S. and Caminada, M. (2009). Proof theories and algorithms for abstract argumentation frameworks. In Simari, G. and Rahwan, I., editors, *Argumentation in Artificial Intelligence*, pages 105–129. Springer US.

Morak, M. (2011). A Dynamic Programming-based Answer Set Programming Solver. Master's thesis, Vienna University of Techology.

Morak, M. (2012). SHARP - A Smart Hypertree decomposition-based Algorithm fRamework for Parameterized problems. `http://www.dbai.tuwien.ac.at/research/project/sharp/` (Accessed: Jan 12, 2012).

Niedermeier, R. (2006). *Invitation to Fixed-Parameter Algorithms*, volume 31. Oxford Lecture Series in Mathematics and its Applications.

Ordyniak, S. and Szeider, S. (2011). Augmenting tractable fragments of abstract argumentation. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 1033–1038.

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

Papadimitriou, C. H. (2003). Computational complexity. In *Encyclopedia of Computer Science*, pages 260–265. John Wiley and Sons Ltd., Chichester, UK.

Robertson, N. and Seymour, P. (1984). Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49 – 64.

Seymour, P. D. and Thomas, R. (1993). Graph searching and a min-max theorem for tree-width. *J. Comb. Theory Ser. B*, 58:22–33.

Verheij, B. (1996). Two approaches to dialectical argumentation: Admissible sets and argumentation stages. In *In Proceedings of the biannual International Conference on Formal and Applied Practical Reasoning (FAPR) workshop*, pages 357–368. Universiteit.

Verheij, B. (2007). A labeling approach to the computation of credulous acceptance in argumentation. In Veloso, M. M., editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 623–628.

Wu, Y., Caminada, M., and Podlaszewski, M. (2010). A labelling-based justification status of arguments. *Proceedings of the 13th International Workshop on Non-Monotonic Reasoning (NMR 2010)*, Studies in Logic, 3 (2010)(4):12–29.