Die approbierte Originalversion dieser Dissertation ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Fixed Parameter Algorithms for Answer Set Programming

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

im Rahmen des Studiums

Dr.-Studium der technischen Wissenschaften

eingereicht von

Michael Jakl
Matrikelnummer 0226072
Fasangasse 10/10; A-1030 Wien

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: Prof. Dr. Reinhard Pichler
Betreuer: Priv.-Doz. Dr. Stefan Woltran

Wien, Mai 2010

(Unterschrift Verfasser)	(Unterschrift Betreuer)	(Unterschrift Betreuer)

FIXED PARAMETER ALGORITHMS FOR ANSWER SET PROGRAMMING

MICHAEL JAKL

Database and Artificial Intelligence Group, 184/2
Faculty of Informatics
Vienna Technical University

Mai 2010

Fixed Parameter Algorithms for Answer Set Programming,
Michael Jakl, Wien, Mai 2010

SUPERVISORS:
Prof. Dr. Reinhard Pichler
Priv.-Doz. Dr. Stefan Woltran

ABSTRACT

Answer Set Programming (ASP) has gained much interest in the last decade and is an increasingly acknowledged paradigm for solving combinatorial problems. In general, determining whether an ASP program has at least one solution, that is, the ASP consistency problem, is already on the second level of the polynomial hierarchy, Σ_2^P.

Parameterized complexity is a promising approach to deal with such intractable problems. It considers the problem-complexity not only as a function of the problem size, but also an additional problem-parameter. Hard problems may become tractable, if this problem-parameter is bounded by a fixed constant, they are then called *fixed parameter tractable*.

One such parameter is *treewidth*, it measures how close a graph resembles a tree. ASP programs can be encoded as graphs and have a treewidth. Using this approach, ASP consistency has been shown to be fixed parameter tractable via monadic second order logic and Courcelle's Theorem. Since the evaluation of monadic second order logic formulae (via tree automata) proved unpractical, this result was only of theoretical value.

In this thesis, we propose practical algorithms to efficiently solve the ASP consistency problem. Additionally two more algorithms for problems of ASP, namely the counting problem (compute the number of solutions) and enumeration problem (compute each solution) are shown.

For programs of bounded treewidth, our dynamic programming algorithms solve the ASP consistency problem in linear time. Further, assuming unit cost for arithmetic and set operations, the number of solutions can be computed in linear time, and the enumeration problem is shown to have linear delay. That is, the time between two solutions is linearly bounded by the size of the problem instance.

Finally, our experiments with a prototype implementation show unprecedented performance for the counting problem. For the enumeration problem, our implementation is comparable to state of the art ASP solvers for a treewidth up to six; only for the ASP consistency problem, we lie behind standard solvers.

KURZFASSUNG DER DISSERTATION

Answer Set Programming (ASP) hat sich in den letzten Jahren als geeignete Methode etabliert, um kombinatorische Probleme zu lösen. ASP Programme lassen sich jedoch nur mit großem Aufwand berechnen. Zu ermitteln ob ein ASP Programm widerspruchsfrei ist, ist auf der zweiten Stufe der polynomiellen Hierarchie, Σ_2^P.

Die parameterisierte Komplexitätstheorie ist ein vielversprechender Ansatz, allgemein schwer berechenbare Probleme zu behandeln. Hierbei wird nicht nur die Größe des Problems, sondern auch ein weiterer Wert, der Problemparameter, betrachtet. Einige Probleme sind leicht berechenbar wenn man den Problemparameter beschränkt, diese werden *parametrisierbar* genannt.

Ein solcher Parameter ist die Baumbreite. Sie gibt an wie stark ein Graph einem Baum entspricht. ASP Programme können als Graphen dargestellt werden und besitzen daher eine Baumbreite. Mit Hilfe von monadischer Logik zweiter Ordnung und Courcelle's Theorem wurde bewiesen, dass der Test auf Widerspruchsfreiheit eines ASP Programms parametrisierbar ist. Die Auswertung von monadischer Logik zweiter Ordnung (via Baumautomaten) ist jedoch nicht praktikabel, daher war dieses Ergebnis bisher nur von theoretischem Interesse.

Wir stellen einen praktischen Algorithmus vor, der effizient ermittelt ob ein parametrisierbares ASP Programm widerspruchsfrei ist. Weiters geben wir Algorithmen an, die die Anzahl der Lösungen eines ASP Programms berechnen und alle Lösungen aufzählen.

Mit Hilfe dynamischer Programmierung stellen wir in linearer Zeit fest, ob ein parametrisierbares ASP Programm widerspruchsfrei ist. Unter der Annahme, dass arithmetische Operationen und Mengenoperationen jeweils nur eine Zeiteinheit benötigen, berechnen wir in linearer Zeit wieviele Lösungen es gibt, und, dass beim Aufzählen aller Lösungen die Wartezeit zwischen zwei Lösungen linear in der Größe der Probleminstanz beschränkt ist.

Anhand einer prototypischen Implementierung können wir experimentell bestätigen, dass wir für Baumbreiten bis zu sechs, sehr gute Laufzeiten beim Zählen der Lösungen haben. Der direkte Vergleich mit einem aktuellen ASP System zeigt, dass die Laufzeit beim Aufzählen aller Lösungen kompetitiv ist. Lediglich der Test auf Widerspruchsfreiheit liegt hinter aktuellen Systemen.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

- Fast Counting with Bounded Treewidth; Jakl, Pichler, Rümmele, and Woltran; 2008; LPAR (International Conference on Logic for Programming, Artificial Intelligence and Reasoning) [47]
- Answer-Set Programming with Bounded Treewidth; Jakl, Pichler, and Woltran; 2009; IJCAI (International Joint Conference on Artificial Intelligence) [48]

ACKNOWLEDGMENTS

I would like to thank to my supervisors, Reinhard Pichler and Stefan Woltran, for fruitful discussions, challenging tasks, guidance, and valuable lessons. Thank you!

This work was supported by the Austrian Science Fund (FWF), project P20704-N18.

CONTENTS

1 INTRODUCTION 17
2 PRELIMINARIES 21
  2.1 Graph Structures 21
  2.2 Propositional Logic 22
    2.2.1 Syntax 23
    2.2.2 Semantics 23
    2.2.3 Conjunctive Normal Form 24
    2.2.4 Partial Interpretation and Model 24
    2.2.5 Satisfiability 25
  2.3 Answer Set Programming 26
    2.3.1 Stable Model Semantics 29
    2.3.2 Architecture of ASP solvers 30
  2.4 Finite Structures 31
  2.5 Monadic Second Order Logic 33
    2.5.1 Syntax 33
    2.5.2 Semantics 34
  2.6 Finite (Tree) Automata 35
3 TREEWIDTH AND COURCELLE'S THEOREM 37
  3.1 Tree Decompositions 37
    3.1.1 Tree Decompositions of Finite Structures 38
    3.1.2 Special Tree Decompositions 40
  3.2 Normalization Procedure 41
  3.3 Path Decompositions 45
  3.4 Algorithms for Finding Tree Decompositions 46
  3.5 Parameterized Complexity 50
  3.6 Courcelle's Theorem 51
4 BASIC STRUCTURE OF FIXED-PARAMETER ALGORITHMS ON TREE DECOMPOSITIONS 55
  4.1 Overview 55
  4.2 Graph Encoding 56
  4.3 Node-Types 57
  4.4 Tree Traversals 58
    4.4.1 General Structure 59
    4.4.2 Counting and Enumerating Models for Propositional Logic 61
5 ANSWER SET PROGRAMMING WITH BOUNDED TREEWIDTH 65
  5.1 Tree Decompositions of ASP Programs. 65
  5.2 The Dynamic Programming Approach for ASP 68

5.2.1 Tree Interpretations 68
5.2.2 Tree Models 71
5.2.3 ASP Consistency 105
5.2.4 Counting Answer Sets 107
5.2.5 Enumerating Answer Sets 110
6 IMPLEMENTATION AND EXPERIMENTAL RESULTS 115
6.1 Answer Set Programming System in Haskell 116
6.1.1 Data Structures 117
6.2 Performance Tests and Benchmarks 122
7 CONCLUSION 129
BIBLIOGRAPHY 133

LIST OF FIGURES

Figure 2.1 Example graph G defined in Table 1. 22
Figure 2.2 Example graph G with a vertex cover V_C and a minimal vertex cover V'_C. 22

Figure 3.1 Example graph G for decomposition purposes. 37
Figure 3.2 Tree decomposition $\mathcal{T}_1$ with width 3 of graph G. 38
Figure 3.3 Optimal tree decomposition $\mathcal{T}_4$ with width 2 of graph G with treewidth 2. 39
Figure 3.4 Primal graph $G_{\mathcal{A}}$ of τ-structure $\mathcal{A}$ shown in Example 3.1.9. 40
Figure 3.5 Normalized tree decomposition $\mathcal{T}_2$ with width 3 of graph G. 43
Figure 3.6 Normalized tree decomposition $\mathcal{T}_3$ with width 3 of graph G. 45
Figure 3.7 Optimal path decomposition $\mathcal{P}$ with pathwidth 2 of graph G with pathwidth 2. 46
Figure 3.8 Graph after the first step of the vertex elimination algorithm. 49
Figure 3.9 Resulting tree decomposition of the vertex elimination algorithm. 49

Figure 4.1 General structure of algorithms on tree decompositions. 55
Figure 4.2 Incidence graph G of formula φ. 56
Figure 4.3 Tree decomposition $\mathcal{T}$ with a width of 2 of graph G. 57
Figure 4.4 Annotated tree decomposition $\mathcal{T}$ with explicit node types. 59
Figure 4.5 Conditions for $(n', M') \prec_{\mathcal{T}} (n, M)$ for the various node-types of n. 62

Figure 5.1 Incidence graph G_P of example program P. 66
Figure 5.2 The *extended normalized tree decomposition* $\mathcal{T}$ of G_P (Figure 5.1). 67
Figure 5.3 Conditions for $(n', M', \mathcal{C}') \prec_{\mathcal{T}} (n, M, \mathcal{C})$ for the various node-types of n. 70

Figure 5.4 The tree decomposition with all $\mathcal{T}$-models and their relations, see Figure 5.5 and Figure 5.6 for the left and right branches of node n_7. 112
Figure 5.5 Left branch of branch node n_7. 113
Figure 5.6 Right branch of branch node n_7. 114
Figure 6.1 Compilation of a disjunctive logic program into an executable, as performed by LAPS. 116
Figure 6.2 Plot showing the number of models against the treewidth (1628 instances). 124
Figure 6.3 Time required by LAPS (upper graph) and DLV (lower graph) for counting of all answer sets (305 instances). 126
Figure 6.4 Time required by LAPS (upper graph) and DLV (lower graph) for enumeration of all answer sets—normalized to time per 100 answer sets (305 instances). 127
Figure 6.5 Comparison of the runtime behavior for counting (upper graph) and enumeration of 100 answer sets (lower graph) plotted against the treewidth (1628 instances). 128

LIST OF TABLES

Table 1 Table showing the set $V(G)$ and $E(G)$ of the graph G in Figure 2.1. 21

LISTINGS

Listing 2.1 ASP program encoding the coin tossing game. 26
Listing 2.2 Ground ASP program R. 26
Listing 2.3 Reduct R^{I_1} of program R in Listing 2.2. 29
Listing 2.4 Reduct R^{I_2} of program R in Listing 2.2. 30

Listing 2.5 Program P. 32

Listing 3.1 Normalization of a binary tree decomposition in pseudo code. 44
Listing 3.2 Vertex Elimination Algorithm [68] 48

Listing 5.1 Program P. 65

Listing 6.1 Example of lazy evaluation. 115
Listing 6.2 Rules in Haskell. 117
Listing 6.3 Basic types used in the program. 117
Listing 6.4 Data structure for the tree decomposition. 118
Listing 6.5 Some nodes of tree decomposition $\mathcal{T}_I$ encoded as Haskell program. 119
Listing 6.6 Haskell lists. 120
Listing 6.7 Declarative structure of the algorithm in Haskell. 120
Listing 6.8 Call of the `asp` algorithm with the root node as parameter. 121
Listing 6.9 Example for a leaf node in Haskell. 121
Listing 6.10 Example for a rule removal node in Haskell. 122

1 INTRODUCTION

In this work, we propose practical algorithms to efficiently solve important problems of Answer Set Programming (ASP) with the help of parameterized complexity.

ASP [56, 63], also known as A-Prolog [4], has gained much interest in the last decade and is an increasingly acknowledged paradigm for solving combinatorial problems. Due to its high expressiveness it enjoys a large collection of successful applications in the areas of artificial intelligence (AI) and knowledge representation and reasoning (KR).

In complexity theory, only problems lying in the class P are considered to be tractable. Many practically relevant problems are NP complete and thus considered intractable. The decision problem of disjunctive logic programs, the underlying concept of ASP, has been shown to be on the second level of the polynomial hierarchy, Σ_2^P, and is thus highly intractable in general, see Eiter and Gottlob (1995) [19].

A promising approach to deal with such intractable problems is parameterized complexity proposed by Downey and Fellows (1999) [17]. Hard problems can become tractable, if some problem parameter is bounded by a fixed constant. These problems are then called fixed parameter tractable, or "in FPT".

One important parameter is *treewidth*, it measures the "tree-likeness" of a graph. Courcelle (1990) [12] proposed a general method to prove FPT results with treewidth as parameter using a restricted form of second order logic, *monadic* second order logic, or MSO logic for short. Courcelle's Theorem, as it is called, was extended by Arnborg, Lagergren, and Seese (1991) [3] and later by Flum, Frick, and Grohe (2002) [30], to capture not only decision problems but also counting and enumeration problems.

By using these results, Gottlob, Pichler, and Wei (2006) [40] have shown that the decision problem for ASP, that is ASP consistency, is indeed fixed parameter tractable when the treewidth of the ASP program is bounded by a fixed constant.

However, an FPT result itself does not immediately lead to an efficient algorithm. The MSO logic used by Courcelle et al. is an elegant way for proving FPT results, but the evaluation of a sentence in MSO logic (via tree automata) suffers from very

high multiplicative constants, rendering the whole approach unpractical even for smallest formulae, see Grohe (1999) [42] and Frick and Grohe (2004) [31].

The goal of this work is, to turn the theoretical tractability results into a practical way of deciding the ASP consistency problem, to identify possible obstacles by doing experiments with a prototype implementation, and to compare our performance to a state of the art ASP solver.

In addition to determining whether a disjunctive logic program has a solution (at least one answer set) or not, we propose two additional algorithms. One to count all answer sets, which is #NP complete in general, and one to enumerate all answer sets.

We generalize the dynamic programming approach for #SAT due to Samer and Szeider (2007) [70] to the world of ASP in order to count and enumerate all answer sets of a given program. In doing so, we provide a novel approach for computing answer sets, which significantly differs from standard ASP systems. These currently do not exploit fixed parameter properties, see Gebser et al. (2007) [33] for an overview of current ASP systems.

MAIN RESULTS To summarize, we provide the following main contributions.

- An FPT algorithm for ASP consistency. The most basic problems are the decision problems, that is, to determine if there is at least one possible solution or none at all. In the world of ASP, this is called ASP consistency checking.

- An FPT algorithm for counting the number of answer sets. To solve the problems at hand, we sometimes need to know how many solutions we can expect. Our algorithms give a precise count of the solutions in linear time (assuming unit cost for arithmetic operations).

- A new method for enumerating *all* answer sets with *linear delay*. Knowing whether there is an answer or not, is often not enough. Our algorithm computes one solution after the other with a guaranteed time bound between two solutions. The time between two solutions is only linear in the size of the problem instance.

- A presentation of our prototype implementation and experimental results. For our experiments we built a prototype in Haskell, a pure functional programming language. The

existence of a working solver using our algorithms ensures the feasibility of our approach.

- A comparison with a state of the art ASP solver. Even though our implementation has only prototype character, the results on problems with low treewidth are very good. We thus provide a head to head comparison with a leading ASP solver, DLV, and identify the limits of our implementation.

STRUCTURE OF THIS WORK In Chapter 2 we introduce the basic notions used throughout the thesis, including Answer Set Programming. In Chapter 3 we show how to transform graphs into tree-structures with an associated treewidth, here we also recall Courcelle's Theorem and why many algorithms based on this approach are not feasible in practice. The basic structure of the dynamic programming algorithms with an example using propositional logic is discussed in Chapter 4. Chapter 5 is the central chapter and presents the algorithms for the three problems mentioned above together with formal proofs of the correctness and a complexity analysis. Chapter 6 gives details on the prototype and benchmarks against a leading ASP system. Finally Chapter 7 gives an outlook and a discussion of related and future work.

2 PRELIMINARIES

In this chapter, we present the basic definitions used throughout the text. We introduce basic undirected graphs, propositional logic and monadic second order logic, Answer Set Programming, and finite structures.

2.1 GRAPH STRUCTURES

We start with the most basic definition of a graph, and introduce some further concepts on top of it.

Definition 2.1.1. *A graph is a pair of sets* $G = (V, E)$ *such that* $E \subseteq V \times V$. *The set* V *is the set of vertices, and the set* E *is the set of edges.*

The set of vertices of a graph G can be accessed by $V(G)$, and likewise, the set of edges by $E(G)$.

Definition 2.1.2. *A graph* G *is called* undirected *if* $(a, b) \in E(G) \iff (b, a) \in E(G)$ *holds for any two vertices* $a, b \in V(G)$.

If not specified, we assume undirected graphs throughout the text.

Example 2.1.3. *The graph* G *in Figure 2.1 is defined by the set* $V(G)$, *and* $E(G)$ *shown in Table 1.*

$V(G)$	$E(G)$
a	(a, c), (c, a)
b	(a, b), (b, a)
c	(b, c), (c, b)
d	(c, d), (d, c)

Table 1: Table showing the set $V(G)$ and $E(G)$ of the graph G in Figure 2.1.

For tree decompositions, cliques are important.

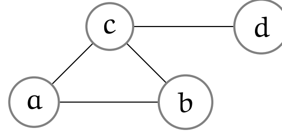


Figure 2.1: Example graph G defined in Table 1.

Definition 2.1.4. *A* clique *in a graph* G *is a set of vertices* $V_c \subseteq V(G)$ *such that for every two vertices* $u, v \in V_c$ *an edge,* $(u, v) \in E(G)$*, exists.*

Example 2.1.5. *In the graph* G *in Figure 2.2, the vertices* a*,* b*, and* c *form a clique. The set of vertices* $\{b, c, d\}$ *does not form a clique because there exists no edge between the vertices* b *and* d*.*

The VERTEX COVER problem is well studied, especially in parameterized complexity, and we will occasionally refer to it.

Definition 2.1.6. *A set of vertices* $V_C \subseteq V(G)$ *is called* vertex cover *if each edge in* $E(G)$ *has at least one endpoint in* V_C*.*

Definition 2.1.7. *A vertex cover* V_C *is called* minimal *if there exists no vertex cover* $V'_C \subset V_C$*.*

In general, it is NP complete to find a minimal vertex cover for a given graph G, see Garey and Johnson (1979) [32].

Example 2.1.8. *In our example graph* G*, the set* $V_C = \{a, b, d\}$ *forms a vertex cover, but it is not minimal. The set* $V'_C = \{b, c\}$ *(also shown in Figure 2.2) is a minimal vertex cover.*

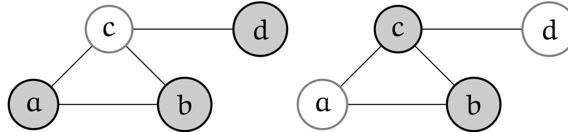


Figure 2.2: Example graph G with a vertex cover V_C and a minimal vertex cover V'_C.

2.2 PROPOSITIONAL LOGIC

Propositional logic is the fundamental formal system used throughout this work. To have a common understanding of the terms and concepts, we introduce the most important definitions here.

2.2.1 *Syntax*

Formulae of propositional logic are constructed from propositional atoms and operators. We consider only well-formed formulae, which can be constructed by the following rules.

atom Every propositional atom is a well-formed formula. Atoms are written as lower case letters $a, \dots, z$ or with a subscript $a_1, \dots, a_n$.

negation If φ is a well-formed formula, then $(\neg\varphi)$ is also a well-formed formula.

conjunction If φ and ψ are well-formed formulae, then $(\varphi \wedge \psi)$ is also a well-formed formula.

disjunction If φ and ψ are well-formed formulae, then $(\varphi \vee \psi)$ is also a well-formed formula.

implication If φ and ψ are well-formed formulae, then $(\varphi \rightarrow \psi)$ is also a well-formed formula.

The above connectives are ordered by their associativity, for example the negation ($\neg$) binds "tighter" than the non exclusive or ($\vee$). It is convenient to omit the unnecessary parentheses, thus the formula $\neg a \vee b$ is a shorthand for $((\neg a) \vee b)$.

We assume only well-formed formulae from now on.

2.2.2 *Semantics*

Until now, we have only established how propositional formulae look like, but we have not fixed a notion of truth values for the formulae.

An interpretation I assigns truth values to propositional atoms.

If a formula consists of a single atom, $\varphi = a$, the formula φ evaluates to the same truth value as the atom a.

The truth values of formulae with connectives are defined in a similar fashion as before.

negation The formula $(\neg\varphi)$ evaluates to true if φ evaluates to false, or $(\neg\varphi)$ evaluates to false if φ evaluates to true.

conjunction The formula $(\varphi \wedge \psi)$ evaluates to true if φ and ψ evaluate to true, otherwise $(\varphi \wedge \psi)$ evaluates to false.

disjunction The formula $(\varphi \vee \psi)$ evaluates to true if φ or ψ, or both evaluate to true, otherwise $(\varphi \vee \psi)$ evaluates to false.

implication The formula $(\varphi \rightarrow \psi)$ evaluates to false if φ evaluates to true and ψ evaluates to false, otherwise $(\varphi \rightarrow \psi)$ evaluates to true.

Note that the formula $(\varphi \rightarrow \psi)$ has the same truth values as the formula $(\neg\varphi \vee \psi)$ for any truth values of φ and ψ, that is, they are *semantically equivalent*.

2.2.3 *Conjunctive Normal Form*

Throughout this work, we consider propositional formulae in a special form, the so called *conjunctive normal form*, CNF for short.

Definition 2.2.1. *A literal* l *is either an atom* a *or the negation of an atom,* $\neg a$*. A clause* c *is a disjunction of literals. A formula* φ *is in* conjunctive normal form *if it is a conjunction of clauses.*

Example 2.2.2. *The formula* $\varphi = (a \vee \neg b) \wedge c$ *is in* CNF, *whereas the formula* $\varphi' = a \vee \neg b \wedge c$ *is not. The case becomes clear if we make the associativity explicit, i.e.,* $\varphi' = a \vee (\neg b \wedge c)$.

Any formula φ can be transformed into an equivalent formula φ' in CNF but with an exponential blowup in the worst case, see for example Huth and Ryan (2004) [46].

2.2.4 *Partial Interpretation and Model*

It is convenient to use a set-representation of formulae in the following. When we use a set notation, we will refer to negated atoms as $\bar{a}$ instead of $\neg a$. In formulae $\neg a$ will be used as described above.

Throughout this work, we assume a universe U of propositional atoms. For a set A of atoms, $\overline{A} = \{\bar{a} \mid a \in A\}$ denotes the set of negated atoms.

A formula is represented as a set of clauses, and clauses are represented as sets of literals. Implicitly we know, that the literals in a set representing a clause are connected with disjunctions, and the clauses in the set of a formula are connected with conjunctions. See Example 2.2.5 for a formula written using standard connectives and a set representation of it.

An interpretation I is a set of atoms.

Definition 2.2.3. *A (partial) interpretation* I *satisfies the clause* c *under the set* $O \subseteq U$*, written as* $I \models_O c$*, if the clause* c *evaluates to true under the following conditions.*

- *All atoms in* I *are considered to be true.*
- *All atoms* a *in* $O \setminus I$ *are considered to be false (or the negation,* $\overline{a}$*, is considered to be true).*

That is,

$$I \models_O c \iff ((I \cap O) \cup \overline{(O \setminus I)}) \cap c \neq \emptyset$$

holds.

For a set C of clauses, $I \models_O C$ holds iff $I \models_O c$, for each $c \in C$. If we consider all atoms in the universe U, that is $O = U$, we usually write $\models$ instead of $\models_O$.

Definition 2.2.4. *An interpretation* I *is called* model *for* φ *if the formula* φ *evaluates to true under* I.

Example 2.2.5. *Consider the following formula* φ.

$$\varphi = a \wedge (b \vee c) \wedge \neg d$$

Let us rewrite φ *in set notation. The corresponding clause set* C_φ *thus looks like this:*

$$C_\varphi = \{\{a\}, \{b, c\}, \{\overline{d}\}\}.$$

Let $O = U = \{a, b, c, d\}$*. Now consider the interpretation* $I_1 = \{a, b, c\}$*. The interpretation is a model for* φ *(that is,* $I_1 \models \varphi$ *holds) because all clauses evaluate to true. The clauses* a *and* $b \vee c$ *are true (*$I_1 \cap \{a\}$ *and* $I_1 \cap \{b, c\}$ *are nonempty). Since* $d \notin I_1$*, the atom* d *is considered to be false (*$\overline{U \setminus I_1} \cap \{\overline{d}\}$ *is nonempty), thus* $\neg d$ *is also satisfied.*

The interpretation $I_2 = \{a, b, c, d\}$ *is not a model of* φ *since* d *is considered to be true, and thus* $\overline{U \setminus I_2} \cap \{\overline{d}\}$ *is empty.*

2.2.5 *Satisfiability*

Satisfiability checking is the problem of determining if a given propositional formula φ evaluates to true for at least one truth assignment, that is, to check if there is at least one model for φ.

To determine whether a given formula φ has a model is NP complete for formulae in conjunctive normal form (CNF), shown by Cook (1971) [11] and Levin (1973) [52].

The satisfiability problem is also called *consistency checking*.

2.3 ANSWER SET PROGRAMMING

ASP [56, 63] is a declarative approach for solving combinatorial problems. The basic idea is to encode models of a problem into the solutions of a (non-monotonic) disjunctive logic program. These models are written as rules and constraints. ASP came up in the nineties, and has its roots in logic programming [78], knowledge representation, and non-monotonic reasoning [59]. Since ASP has common concepts with more classical logic programming, that is Prolog, it is also known as AnsProlog or A-Prolog, see Baral (2003) [4].

Example 2.3.1. *Consider a very simple game: coin tossing. Listing 2.1 shows a program encoding the game rules. The program is read as follows.* `euro1` *and* `euro2` *are coins, you win if the coin's face is up, and you lose if the coin's face is down. The last rule is a disjunction, it encodes that a coin's face is either up or down, but only if we have no evidence that the coin has been forged.*

Uppercase letters, or words starting with an uppercase letter, are considered to be variables, lower case letters, or words starting with lower case letters, are considered to be constants.

Listing 2.1: ASP program encoding the coin tossing game.

```
1 coin(euro1).
2 coin(euro2).
3 win(X) :- coin(X), up(X).
4 lose(X) :- coin(X), down(X).
5 up(X) v down(X) :- coin(X), not forged(X).
```

In this work, we consider propositional disjunctive logic programs, or *ground* programs with a disjunction in its head, see Minker (1994) [60]. ASP in general allows variables to be used in the programs, which requires the program to be "grounded" before the models can be computed. The result of the grounding is a propositional (disjunctive) logic program.

Example 2.3.2. *Listing 2.1 is a simple non-ground program, with Listing 2.2 as the grounded version.*

Listing 2.2: Ground ASP program R.

```
1 coin(euro1).
2 win(euro1) :- coin(euro1), up(euro1).
3 lose(euro1) :- coin(euro1), down(euro1).
```

```
 4 up(euro1) v down(euro1) :- coin(euro1),
 5     not forged(euro1).
 6 coin(euro2).
 7 win(euro2) :- coin(euro2), up(euro2).
 8 lose(euro2) :- coin(euro2), down(euro2).
 9 up(euro2) v down(euro2) :- coin(euro2),
10     not forged(euro2).
```

There are many variants and extensions to ASP programs. We will review the basic ones before we dive into the disjunctive case. For an extensive treatment of possible extensions, see Niemelä (2005) [64]. Each of these extensions has its own expressiveness and time or space complexity to compute an answer, see Eiter and Gottlob (1995) [19] for some complexity results.

The most basic logic program has neither negation nor a disjunction in its head, and is called *Horn*.

Definition 2.3.3. *A Horn logic program is a set of rules of the form*

$$h \leftarrow a_1, \ldots, a_m$$

where h *and* $a_1, \ldots, a_m$ *are propositional atoms from a universe* U*. The atom* h *is called* head*, and the set of atoms* $a_1, \ldots, a_m$ *is called* body*.*

The consistency problem for Horn logic programs is P complete, see Dantsin et al. (2001) [15].

Definition 2.3.4. *A normal program is a set of rules of the form*

$$h \leftarrow a_1, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_{m+n}$$

where h *and* $a_1, \ldots, a_{m+n}$ *are propositional atoms from a universe* U*. The atom* h *is called* head*, the set of atoms* $a_1, \ldots, a_m$ *is called* positive body*, and the set of atoms* $a_{m+1}, \ldots, a_{m+n}$ *is called* negative body*.*

A normal program may have no solution since the negation may invalidate all solutions. The program

$$p \leftarrow \texttt{not}\ p$$

is such an example. To deal with negation, we need additional concepts like stratification [1], well-founded semantics [35], or—the one we will use—stable models semantics [37] defined below. The ASP consistency problem for normal programs is NP complete, see Dantsin et al. (2001) [15].

Definition 2.3.5. *A disjunctive program is a set of rules of the form*

$$h_1 \vee \ldots \vee h_j \leftarrow a_1, \ldots, a_m, \texttt{not}\ a_{m+1}, \ldots, \texttt{not}\ a_{m+n}$$

where $h_1, \ldots, h_j$ *and* $a_1, \ldots, a_{m+n}$ *are propositional atoms from a universe* U. *The set of atoms* $h_1, \ldots, h_j$ *is called* head*, the set of atoms* $a_1, \ldots, a_m$ *is called* positive body*, and the set of atoms* $a_{m+1}, \ldots, a_{m+n}$ *is called* negative body*.*

Adding disjunction to normal programs increases its expressiveness (and complexity) so that it is able to express problems on the second level of the polynomial hierarchy, Σ_2^P. Again, to deal with negation we have various semantics at our disposal. We will use disjunctive stable model semantics, which is an adapted form of the stable model semantics [20].

A program with no negation is called *positive*. Since there can be no contradictions, a positive program has always a solution. In the following, we will use positive programs to define the notion of a "reduct".

We define shortcuts for the head and the body of a rule r as given in Definition 2.3.5, such that the set of atoms in the head $\text{H}(r) = \{h_1, \ldots, h_j\}$, the set of atoms in the positive body $\text{B}^+(r) = \{a_1, \ldots, a_m\}$, and the negative body $\text{B}^-(r) = \{a_{m+1}, \ldots, a_{m+n}\}$.

By $At(R)$ we denote the set of atoms occurring in program R.

We identify a program R with the clause set $\{\text{H}(r) \cup \text{B}^-(r) \cup \overline{\text{B}^+(r)} \mid r \in R\}$.

Example 2.3.6. *The rule in line 9 of Listing 2.2 is thus represented by the set*

$$\{up(euro2), down(euro2), forged(euro2), \overline{coin(euro2)}\},$$

and the whole program Listing 2.2 by the set

$$\begin{aligned}
\{\\
&\{coin(euro1)\},\\
&\{coin(euro2)\},\\
&\{win(euro1), \overline{up(euro1)}, \overline{coin(euro1)}\},\\
&\{lose(euro1), \overline{down(euro1)}, \overline{coin(euro1)}\},\\
&\{up(euro1), down(euro1), forged(euro1), \overline{coin(euro1)}\},\\
&\{win(euro2), \overline{up(euro2)}, \overline{coin(euro2)}\},\\
&\{lose(euro2), \overline{down(euro2)}, \overline{coin(euro2)}\},\\
&\{up(euro2), down(euro2), forged(euro2), \overline{coin(euro2)}\}\\
\}.
\end{aligned}$$

Listing 2.3: Reduct R^{I_1} of program R in Listing 2.2.

```
1 coin(euro1).
2 coin(euro2).
3 win(euro1) :- coin(euro1), up(euro1).
4 lose(euro1) :- coin(euro1), down(euro1).
5 up(euro1) v down(euro1) :- coin(euro1).
6 win(euro2) :- coin(euro2), up(euro2).
7 lose(euro2) :- coin(euro2), down(euro2).
8 up(euro2) v down(euro2) :- coin(euro2).
```

2.3.1 *Stable Model Semantics*

There are several possible semantics, see Minker (1996) [61], the most prominent being the *stable model semantics* introduced by Gelfond and Lifschitz (1988) [36] and covered by many others [56, 63, 75], which we will concentrate on. The solutions of ASP programs are thus called stable models, we will occasionally refer to them simply as answer sets.

Answer sets are defined around the so called *reduct*, which was introduced to deal with negation in logic programs. The reduct is a positive program constructed from the disjunctive program R and an interpretation I, or more formally:

Definition 2.3.7. *We define the reduct* R^I *of a program* R *with respect to an interpretation* I *as* $\{H(r) \cup \overline{B^+(r)} \mid r \in R, B^-(r) \cap I = \emptyset\}$.

The program R^{I_1} shown in Listing 2.3 is the reduct of the program R shown in Listing 2.2 together with the interpretation

$$I_1 = \{\mathrm{coin}(\mathrm{euro1}), \mathrm{coin}(\mathrm{euro2}), \mathrm{lose}(\mathrm{euro1}), \mathrm{down}(\mathrm{euro1}), \mathrm{lose}(\mathrm{euro2}), \mathrm{down}(\mathrm{euro2})\}.$$

In this case, only the atoms of the negative body are removed from R. In the case of the following interpretation,

$$I_2 = \{\mathrm{coin}(\mathrm{euro1}), \mathrm{coin}(\mathrm{euro2}), \mathrm{lose}(\mathrm{euro1}), \mathrm{down}(\mathrm{euro1}), \mathrm{lose}(\mathrm{euro2}), \mathrm{down}(\mathrm{euro2}), \mathrm{forged}(\mathrm{euro1})\},$$

the whole rule containing $\mathrm{forged}(\mathrm{euro1})$ in its negative body is removed, see Listing 2.4.

We are now ready to define the stable models, or answer sets, of a program.

Listing 2.4: Reduct R^{I_2} of program R in Listing 2.2.

```
1 coin(euro1).
2 coin(euro2).
3 win(euro1) :- coin(euro1), up(euro1).
4 lose(euro1) :- coin(euro1), down(euro1).
5 win(euro2) :- coin(euro2), up(euro2).
6 lose(euro2) :- coin(euro2), down(euro2).
7 up(euro2) v down(euro2) :- coin(euro2).
```

Definition 2.3.8. *An interpretation* I *is an* answer set *of a program* R *iff* $I \models R$ *and there exists no* $J \subset I$ *such that* $J \models R^I$*. The set of all answer sets of a program* R *is denoted by* $\mathcal{AS}(R)$*.*

Example 2.3.9. *Let us now inspect the two interpretations* $I_1 \models R$ *and* $I_2 \models R$ *we have shown above.*

For the interpretation I_1*, there is no* $J \subset I_1$ *such that* $J \models R^{I_1}$*, thus* I_1 *is indeed an answer set of* R*.*

For the interpretation I_2 *and the reduct* R^{I_2} *shown in Listing 2.4, we have two* $J \subset I_2$ *such that* $J \models R^{I_2}$ *holds. For example the set*

$$J = \{coin(euro1), coin(euro2), lose(euro2), down(euro2)\}.$$

Thus, we conclude that I_2 *is not a stable model (or answer set) for* R*, and hence not included in* $\mathcal{AS}(R)$*.*

Our program R *shown in Listing 2.1 (or equivalently Listing 2.2) has four answer sets (we omit the* `coin`*,* `up`*, and* `down` *atoms for clarity):*

$$\begin{aligned}\mathcal{AS}(R) = \{ & \\ & \{lose(euro1), lose(euro2)\}, \\ & \{win(euro1), lose(euro2)\}, \\ & \{lose(euro1), win(euro2)\}, \\ & \{win(euro1), win(euro2)\} \\ \}. & \end{aligned}$$

2.3.2 *Architecture of* ASP *solvers*

A typical ASP solver follows a two phase architecture, first the non-ground program is grounded, then the answer sets are computed for the propositional (or *ground*) program.

1. Grounding step: the grounding step could lead to an exponential blowup of the program, thus several grounding

procedures have been developed to optimize the ground program, see for example Eiter et al. (2007) [23].

2. Model search: for the grounded program a solver computes the stable models, which are also stable models for the non-ground programs. For this model search, many different algorithms have been proposed, see Eiter, Ianni, and Krennwallner (2009) [24] for a survey.

Recently, some research into a more interwoven ASP solver has been conducted. For example, techniques for lazy or partial grounding have been shown (see [14, 34]).

Our algorithms presented in Chapter 5 are a new way for solving step 2, the model search, which has been shown to be Σ_2^P complete for disjunctive logic programs by Eiter and Gottlob (1995) [19].

2.4 FINITE STRUCTURES

Definition 2.4.1. *Let τ be a set of predicate symbols $\tau = \{R_1, \ldots, R_n\}$. A* finite structure *$\mathcal{A}$ over τ (a τ-structure for short) is given by a finite domain, that is, a set of constants A, and relations $R_i^{\mathcal{A}} \subseteq A^{r_i}$, where r_i is the arity of $R_i \in \tau$.*

We will show how to encode propositional formulae as finite structures in Example 2.4.2, and how to encode ASP programs in Example 2.4.3.

Example 2.4.2. *To encode propositional formulae as a finite structure, we introduce several predicate symbols $\tau = \{var, cl, pos, neg\}$ with the following intended meaning.*

- *$var(a)$ denotes that a is a variable.*
- *$cl(C)$ denotes that C is a clause.*
- *$pos(C, a)$ denotes that a occurs positively in C.*
- *$neg(C, a)$ denotes that a occurs negatively in C.*

The propositional formula given in Example 2.2.5,

$$\varphi = a \wedge (b \vee c) \wedge \neg d$$

can now be encoded using finite structures (we refer to the clauses as $C_1 = a$, $C_2 = a \vee c$, and $C_3 = \neg d$ respectively).

$$\mathcal{A} = (A, var^{\mathcal{A}}, cl^{\mathcal{A}}, pos^{\mathcal{A}}, neg^{\mathcal{A}})$$

The finite domain is defined as $A = \{a, b, c, d, C_1, C_2, C_3\}$*, and the relations are defined as*

$$\begin{aligned}
var^{\mathcal{A}} &= \{var(a), var(b), var(c), var(d)\} \\
cl^{\mathcal{A}} &= \{cl(C_1), cl(C_2), cl(C_3)\} \\
pos^{\mathcal{A}} &= \{pos(C_1, a), pos(C_2, b), pos(C_2, c)\} \\
neg^{\mathcal{A}} &= \{neg(C_3, d)\}
\end{aligned}$$

Example 2.4.3. *To encode an* ASP *program as a finite structure, we introduce several predicate symbols* $\tau = \{at, rl, pB, nB, H\}$ *with the following intended meaning.*

- $at(a)$ *denotes that* a *is an atom.*
- $rl(R)$ *denotes that* R *is a rule.*
- $pB(R, a)$ *denotes that* a *occurs positively in the body of* R.
- $nB(R, a)$ *denotes that* a *occurs negatively in the body of* R.
- $H(R, a)$ *denotes that* a *occurs in the head of* R.

Consider the following program (Listing 2.5).

Listing 2.5: Program P.

```
1 a.
2 b :- a, c.
3 e :- a, d.
4 c ∨ d :- a, not f.
```

Program P *shown in Listing 2.5 can now be encoded using finite structures (we refer to the rules by their line number, that is the rule in line 1 is* R_1*)*

$$\mathcal{A} = (A, at^{\mathcal{A}}, rl^{\mathcal{A}}, pB^{\mathcal{A}}, nB^{\mathcal{A}}, H^{\mathcal{A}})$$

The finite domain is defined as $A = \{a, b, c, d, e, f, R_1, R_2, R_3, R_4\}$*, and the relations are defined as*

$$\begin{aligned}
at^{\mathcal{A}} &= \{at(a), at(b), at(c), at(d), at(e), at(f)\} \\
rl^{\mathcal{A}} &= \{rl(R_1), rl(R_2), rl(R_3), rl(R_4)\} \\
pB^{\mathcal{A}} &= \{pB(R_2, a), pB(R_2, c), pB(R_3, a), pB(R_3, d), pB(R_4, a)\} \\
nB^{\mathcal{A}} &= \{nB(R_4, f)\} \\
H^{\mathcal{A}} &= \{H(R_1, a), H(R_2, b), H(R_3, e), H(R_4, c), H(R_4, d)\}.
\end{aligned}$$

2.5 MONADIC SECOND ORDER LOGIC

Monadic Second Order logic (MSO logic for short), see Ebbinghause and Flum (1999) [18], is a very expressive language. It can be shown that for every level of the polynomial hierarchy there exists a problem expressible in MSO logic. But it is not possible to capture the polynomial hierarchy itself, see Matz and Thomas (1997) [58] and Libkin (2004) [53].

2.5.1 *Syntax*

MSO logic is defined over τ-structures $\mathcal{A}$ with domain A, that is, we have constants a standing for objects from the domain $a \in A$. Additionally, we have

- an infinite supply of individual variables (written as lower case letters) ranging over the domain A, and
- an infinite supply of set variables (written as upper case letters) ranging over subsets of the domain A.

Further, an MSO logic formula is defined inductively over the variables and constants and the set of predicate symbols, τ.

- If P is a predicate symbol, $P \in \tau$, with arity r, and $t_1, \ldots, t_r$ are constants or variables, then $P(t_1, \ldots, t_r)$ is a formula.
- If X is a set variable and t is a constant, then $X(t)$ is a formula. This may also be written as $t \in X$.
- If t_1 and t_2 are constants, then $t_1 = t_2$ is a formula.
- If φ is a formula, then $(\neg\varphi)$ is a formula.
- If φ is a formula and x is a variable, then $((\forall x)\varphi)$ and $((\exists x)\varphi)$ are formulae.
- If φ is a formula and X is a set variable, then $((\forall X)\varphi)$ and $((\exists X)\varphi)$ are formulae.
- If φ and ψ are formulae, then $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, and $(\varphi \rightarrow \psi)$ are formulae.

A (set-)variable is called *bound* in φ if it is quantified, that is, $\varphi = (\forall x)\psi$ or $\varphi = (\exists x)\psi$, otherwise, x is called *free*. If φ has free variables x and X, we write $\varphi(x, X)$. If φ has no free variables, it is called a *sentence*.

2.5.2 *Semantics*

The truth value of a formula is defined be the following rules ($\varphi[x/a]$ is used as a shortcut of the formula φ where each *free* occurrence of x is replaced with a).

- A formula $\varphi = (\forall x)\psi$ evaluates to true if, for all constants $a \in A$ the formula $\psi[x/a]$ evaluates to true, otherwise it evaluates to false.
- A formula $\varphi = (\forall X)\psi$ evaluates to true if, for all subsets $B \subseteq A$ the formula $\psi[X/B]$ evaluates to true, otherwise it evaluates to false.
- A formula $\varphi = (\exists x)\psi$ evaluates to true if there exists an atom $a \in A$ such that the formula $\psi[x/a]$ evaluates to true, otherwise it evaluates to false.
- A formula $\varphi = (\exists X)\psi$ evaluates to true if there exists a subset $B \subseteq A$ such that $\psi[X/B]$ evaluates to true, otherwise it evaluates to false.
- A formula $\varphi = \psi \wedge \xi$ evaluates to true if the formulae ψ and ξ evaluate to true, otherwise it evaluates to false.
- A formula $\varphi = \psi \vee \xi$ evaluates to true if at least one of the formulae ψ or ξ evaluate to true, otherwise it evaluates to false.
- A formula $\varphi = \psi \to \xi$ evaluates to false if the formula ξ evaluates to false and the formulae ψ evaluates to true, otherwise it evaluates to true
- If $\varphi = X(t)$, respectively $\varphi = (t \in X)$, is a sentence, t is a constant, and $X \subseteq A$ is a set, then φ evaluates to true if the t is contained in X.
- If $\varphi = P(t_1, \ldots, t_r)$ is a sentence and P is a predicate symbol, $P \in \tau$, with arity r, and $t_1, \ldots, t_r$ are constants, then φ evaluates to true if there exists a tuple $(t_1, \ldots, t_r) \in P^{\mathcal{A}}$, otherwise it evaluates to false.

A formula with free variables has no truth value.

Definition 2.5.1. *Any property of a finite structure expressible in* MSO *logic is called* MSO *logic* definable*.*

Example 2.5.2. *We encode the well-known* 3-COLORABILITY *problem in* MSO *logic defined over a* τ*-structure* $\mathcal{A}$.

Consider the τ*-structure* $\mathcal{A}$*, encoding a graph* G *where the domain* V *are vertices of the graph* G*, and the only predicate symbol,* E*, represents edges in the graph* G*,* $\tau = \{E\}$*. That is, the tuple* (a, b) *where* $a, b \in V$ *is in* E *if there is an edge between the vertices* a *and* b.

First we define a formula encoding a partition of the universe into three non-overlapping subsets.

$$\begin{aligned} \mathtt{partition}(X_1, X_2, X_3) = &(\forall x)((x \in X_1 \wedge x \notin X_2 \wedge x \notin X_3) \vee \\ &(x \notin X_1 \wedge x \in X_2 \wedge x \notin X_3) \vee \\ &(x \notin X_1 \wedge x \notin X_2 \wedge x \in X_3)) \end{aligned}$$

Finally, we define the formula φ *encoding the* 3-COLORABILITY *problem.*

$$\begin{aligned} \varphi = &(\exists R, G, B)\mathtt{partition}(R, G, B) \rightarrow \\ &((\forall v_1, v_2)(v_1, v_2) \in E \rightarrow \\ &(v_1 \in R \wedge v_2 \in G) \vee (v_1 \in R \wedge v_2 \in B) \vee (v_1 \in B \wedge v_2 \in G) \vee \\ &(v_1 \in G \wedge v_2 \in R) \vee (v_1 \in B \wedge v_2 \in R) \vee (v_1 \in G \wedge v_2 \in B)) \end{aligned}$$

That is, if φ *evaluates to true over the* τ*-structure* $\mathcal{A}$*, then there exists a valid partition of the vertices (domain) into three disjunct sets, such that no two adjacent vertices are in the same set.*

2.6 FINITE (TREE) AUTOMATA

Definition 2.6.1. *Let* Σ *be a finite nonempty set of symbols, the set of finite strings over* Σ *is denoted as* Σ^**. A subset* $L \subseteq \Sigma^*$ *is called* language.

Definition 2.6.2. *A* finite automaton *is a tuple*

$$A = (Q, \Sigma, q_0, F, \delta)$$

where Q *is a finite set of* states*,* Σ *is a finite set of symbols,* $q_0 \in Q$ *is the initial state,* $F \subseteq Q$ *is the set of final states, and* δ *is a function from* $Q \times \Sigma$ *to* 2^Q *called* transition function*. An automaton is called* deterministic *if* $|\delta(q, a)| = 1$ *for every* $q \in Q$ *and* $a \in \Sigma$*, otherwise it is called* nondeterministic.

Every nondeterministic finite automaton can be transformed into a deterministic finite automaton, see Glabbeek and Ploeger (2008) [38] for an overview of determinisation algorithms. We will hence assume deterministic finite automata if not noted otherwise.

A *run* of a finite automaton A on $l \in L$, where $l = a_1, \ldots, a_n$, is a mapping $r : \{1, \ldots, n\} \to Q$ where

$$r(1) = \delta(q_0, a_1)$$
$$r(i+1) = \delta(r(i), a_i)$$

Such a run is called *accepting* if $r(n) \in F$.

We say, an automaton A recognizes a language L if for every $l \in L$, a run of the automaton A accepts l, and for every $l \notin L$ there is no accepting run of the automaton A for l.

There is a well-known correspondence between formulae in MSO logic and finite automata, see Büchi (1960) [9] and Elgot (1961) [25]. That is, a language is recognizable by a finite state automaton if and only if it is MSO logic definable.

Definition 2.6.3. *A Σ-tree is a pair (T, β) where T is a tree and β is a function from T to Σ called* labelling function*.*

The set TREE(Σ) *is the set of all Σ-trees, a subset $L_T \subseteq$ TREE(Σ) is called tree language.*

Definition 2.6.4. *A* finite tree automaton *is a tuple*

$$A = (Q, \Sigma, q_0, F, \delta)$$

where Q is a finite set of states*, Σ is a finite set of symbols, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and δ is a function from $Q \times Q \times \Sigma$ to 2^Q called* transition function*. An automaton is called* deterministic *if $|\delta(q_1, q_2, a)| = 1$ for every $q_1, q_2 \in Q$ and $a \in \Sigma$, otherwise it is called* nondeterministic*.*

Similarly to finite automata, nondeterministic finite tree automata can be transformed into deterministic finite tree automata, hence we will assume only deterministic finite tree automata.

A *run* of a finite tree automaton A on a Σ-tree (T, β) is a mapping $r : T \to Q$ where

$$r(s) = \delta(q_0, q_0, \beta(s))$$

if $s \in T$ is a leaf node, and

$$r(p) = \delta(r(s_1), r(s_2), \beta(b))$$

where $s_1, s_2, p \in T$ and s_1 and s_2 are the children of p.

Such a run is called *accepting* if $r(n) \in F$, where $n \in T$ is the root node of T.

Thatcher and Wright (1968) [76] have shown that a tree language is recognizable by a finite tree automaton if and only if it is MSO logic definable. Furthermore, they show how to construct a finite tree automaton from an MSO logic formula.

3 TREEWIDTH AND COURCELLE'S THEOREM

A tree decomposition is a way of transforming a graph into a tree structure. This new structure has certain properties, and with them, it preserves certain properties of the original graph.

Many algorithms run considerably faster on these structures than on the original graph. In general, it is NP complete to decide if there exists a tree decomposition or a path decomposition of a graph with width k, see Arnborg, Corneil, and Proskurowski (1987) [2] for details. Fortunately, there exist FPT algorithms and heuristics to speed up the decomposition process.

Example 3.0.5. *We will use the following graph to illustrate the various definitions.*

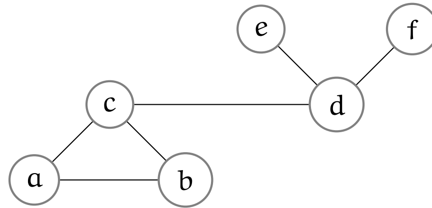


Figure 3.1: Example graph G for decomposition purposes.

3.1 TREE DECOMPOSITIONS

The notion of tree decompositions was coined by Robertson and Seymour in 1984 [67]. Since then, tree decompositions have been widely studied, and successfully applied to solve otherwise hard problems, see Bodlaender (1993) [5] for a survey of successful applications within 10 years of the original publication.

The treewidth of a graph is a measure how close the graph resembles a tree.

Definition 3.1.1. *A* tree decomposition $\mathcal{T}$ *of a graph* $G = (V, E)$ *is a pair* $\mathcal{T} = (T, \beta)$, *where* T *is a rooted tree and* β *maps each node* $n \in T$ *to a* bag $\beta(n) \subseteq 2^V$, *such that the following conditions are met.*

- *For each* $v \in V$, *there exists an* $n \in T$ *such that* $v \in \beta(n)$.

- *For each* $(v, w) \in E$*, there exists an* $n \in T$*, such that* $v, w \in \beta(n)$.
- *For any three nodes* $n_1, n_2, n_3 \in T$*, if* n_2 *lies on the path from* n_1 *to* n_3*, then* $\beta(n_1) \cap \beta(n_3) \subseteq \beta(n_2)$ *holds (this is often called* connectedness condition*).*

Definition 3.1.2. *The* width of a tree decomposition *is defined as the cardinality of its largest bag* $|\beta(n)|$ *minus* 1.

Example 3.1.3. *One possible tree decomposition of the graph* G *is shown in Figure 3.2.*

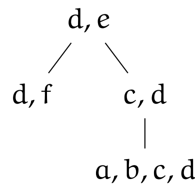


Figure 3.2: Tree decomposition $\mathcal{T}_1$ with width 3 of graph G.

Definition 3.1.4. *The* treewidth of a graph G*, denoted as* $tw(G)$*, is the minimum width over all tree decompositions of* G.

There exists a tree decomposition of a tree with two elements per bag, and thus trees have treewidth of 1.

Definition 3.1.5. *A tree decomposition is called* optimal *if the width of the tree decomposition equals the treewidth of the graph.*

Example 3.1.6. *Note that the tree decomposition in Figure 3.2 is not optimal. It has width* 3*, but the graph has a treewidth of* 2 *(the element* d *in the node* a, b, c, d *is redundant). See Figure 3.3 for an optimal tree decomposition.*

The treewidth of the example graph is 2*. All vertices that form a clique, have to occur at least once together in a bag. The vertices* a, b*, and* c *form a clique, hence we have a node containing* a, b*, and* c *leading to a treewidth of at least* 2*. Our tree decomposition* $\mathcal{T}_4$ *(shown in Figure 3.3) has width* 2*, so we can conclude that we have found an optimal tree decomposition.*

3.1.1 *Tree Decompositions of Finite Structures*

In this section we will generalize the notion of tree decompositions of graphs to tree decompositions of arbitrary finite structures.

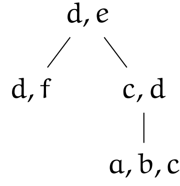


Figure 3.3: Optimal tree decomposition $\mathcal{T}_4$ with width 2 of graph G with treewidth 2.

Definition 3.1.7. *A* tree decomposition $\mathcal{T}$ *of a* τ*-structure* $\mathcal{A}$ *with domain A is a pair* $\mathcal{T} = (T, \beta)$*, where* T *is a rooted tree and* β *maps each node* $n \in T$ *to a* bag $\beta(n) \subseteq A$*, such that the following conditions are met.*

- *For each* $a \in A$*, there exists an* $n \in T$ *such that* $a \in \beta(n)$*.*
- *For every* $R_i \in \tau$ *and every tuple* $(a_1, \dots, a_r) \in R_i^{\mathcal{A}}$*, there exists an* $n \in T$ *such that* $\{a_1, \dots, a_r\} \subseteq \beta(n)$*.*
- *For any three nodes* $n_1, n_2, n_3 \in T$*, if* n_2 *lies on the path from* n_1 *to* n_3*, then* $\beta(n_1) \cap \beta(n_3) \subseteq \beta(n_2)$ *holds (this is often called* connectedness condition*).*

We can use the definition of tree decompositions of arbitrary τ-structures to define tree decompositions of graphs by encoding graphs as τ-structures.

Suppose the graph G has vertices V and edges E, that is, $G = (V, E)$. To encode this graph as a τ-structure $\mathcal{A}$, we define the set of predicate symbols as $\tau = \{\texttt{Edge}\}$, and the domain of $\mathcal{A}$ as V.

$\texttt{Edge}^{\mathcal{A}}$ contains all edges, that is, for each $(v_1, v_2) \in E$ there exists a $(v_1, v_2) \in \texttt{Edge}^{\mathcal{A}}$.

It is also possible to encode an arbitrary τ-structure $\mathcal{A}$ as a graph $G_{\mathcal{A}}$ (more specifically, a *primal graph*, see Definition 3.1.8) and build equivalent tree decompositions from the graph $G_{\mathcal{A}}$. It can be shown that the tree decompositions of the τ-structure $\mathcal{A}$ are exactly the same as the tree decompositions of the corresponding primal graph G of $\mathcal{A}$.

Definition 3.1.8. *A primal graph (also known as Gaifman graph)* $G_{\mathcal{A}} = (V, E)$ *of a* τ*-structure* $\mathcal{A}$ *with domain* A *is defined as follows. The domain elements of* $\mathcal{A}$ *are the vertices of the graph* V*, that is* $V = A$*. Two vertices* v_1 *and* v_2 *are adjacent in* $G_{\mathcal{A}}$*, that is* $(v_1, v_2) \in E$*, if there exists an* $R_i \in \tau$ *such that* $(a_1, \dots, a_r) \in R_i^{\mathcal{A}}$ *and* $v_1, v_2 \in \{a_1, \dots, a_r\}$*.*

Example 3.1.9. *In this example, we will illustrate the equivalence of the tree decompositions of the graph* G *in Figure 3.1, and the tree decompositions of the corresponding* τ*-structure* $\mathcal{A}$.

The domain elements A *of* $\mathcal{A}$ *are the vertices of* G*, that is,* $A = \{a, b, c, d, e, f\}$*. The* $\mathsf{Edge}^{\mathcal{A}}$*-relation is defined as*

$$\mathsf{Edge}^{\mathcal{A}} = \{(a, b), (a, c), (b, c), (c, d), (d, e), (d, f)\}.$$

The primal graph $G_{\mathcal{A}}$ *of* $\mathcal{A}$ *is shown in Figure 3.4.*

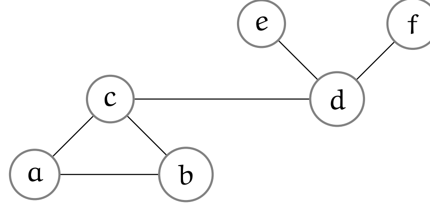


Figure 3.4: Primal graph $G_{\mathcal{A}}$ of τ-structure $\mathcal{A}$ shown in Example 3.1.9.

Since the graph G *and the primal graph* $G_{\mathcal{A}}$ *of the corresponding* τ*-structure* $\mathcal{A}$ *are identical,* G *and* $G_{\mathcal{A}}$ *have the same tree decompositions.*

3.1.2 *Special Tree Decompositions*

In this section, we introduce the notion of normalized tree decomposition and extended normalized tree decompositions. Arbitrary tree decompositions can be transformed into (extended) normalized tree decompositions in linear time. (Extended) normalized tree decompositions are convenient to work with.

Definition 3.1.10. *A tree decomposition* $\mathcal{T} = (T, \beta)$ *is called* normalized *if the following conditions hold.*

- *Each node in* T *has at most two children.*
- *For each node* n *with two children* n_1 *and* n_2 *the bags are equal, i.e.,* $\beta(n) = \beta(n_1) = \beta(n_2)$.
- *For each node* n *with one child* n'*, the corresponding bags,* $\beta(n)$ *and* $\beta(n')$*, differ in exactly one element, i.e.,*

$$|\beta(n') \setminus \beta(n)| + |\beta(n) \setminus \beta(n')| = 1.$$

A normalized tree decomposition is called nice tree decomposition by Kloks (1994) [50].

3.2 NORMALIZATION PROCEDURE

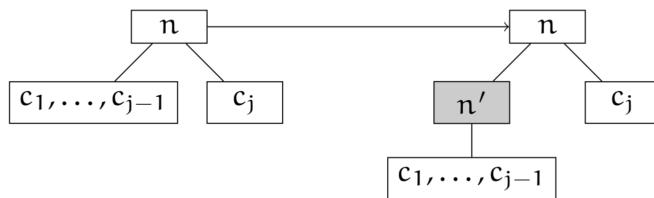
In the following, we will explain the normalization procedure in detail. The algorithm takes a node of a tree decomposition as input and transforms the tree rooted at that node into a normalized tree decomposition. That is, if we apply the algorithm to the root node of a tree decomposition, the result is a normalized tree decomposition. Listing 3.1 contains the normalization algorithm explained below in pseudo code.

Let n be the node the algorithm is called with.

STEP 1: BINARY TREE DECOMPOSITION The tree decomposition is first transformed into a *binary* tree decomposition. This is done by a recursive approach.

Let $c_1, \cdots, c_j$ be the children of n.

- If $j > 2$:
  - Remove the children $c_1, \ldots, c_{j-1}$ from n.
  - Insert a new node n' where $\beta(n') = \beta(n)$ as the first child of n, and set $c_1, \ldots, c_{j-1}$ as the children of n'.
- Transform n' and c_j recursively.



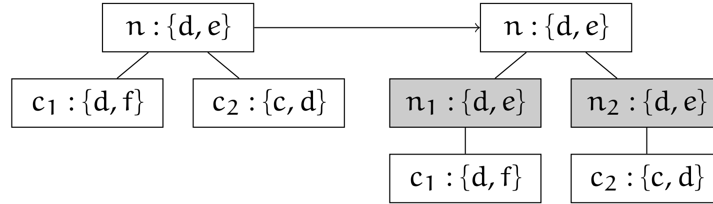
STEP 2: ENSURING CONDITIONS After the transformation into a binary tree decomposition, we ensure the remaining conditions given in Definition 3.1.10. That is, the following exposition assumes a binary tree decomposition as input.

We have three cases: the node n is either a leaf node, branch node, or a node with exactly one child.

- Leaf Node: If n is a leaf node, we are done.

- Branch Node: For a branch node, we duplicate the node n two times, n_1 and n_2, and insert the new nodes as children of n. For the original children c_1 and c_2 of n, we set c_1 as the only child of n_1 and c_2 as the only child of n_2.

Finally, we call the normalization algorithm once with n_1 and n_2 as parameter to normalize the left and right branches of the branch node.



- For all other nodes, that is, nodes n with exactly one child c, we compute the difference of the bags, $S_1 = \beta(n) \setminus \beta(c)$ and $S_2 = \beta(c) \setminus \beta(n)$.

  It is necessary to leave out exactly one element of either S_1 or S_2, otherwise we would create a node identical to c. Identical bags are only permitted for branch nodes, so we have to be careful not to introduce duplicate bags.

  The set S_1 contains those elements that are only in the bag of n, and hence have to be removed one by one. The set S_2 contains those elements that are only in the bag of c and have hence to be added one by one.

  It is crucial to start with the elimination step, otherwise we would increase the width of the tree decomposition.

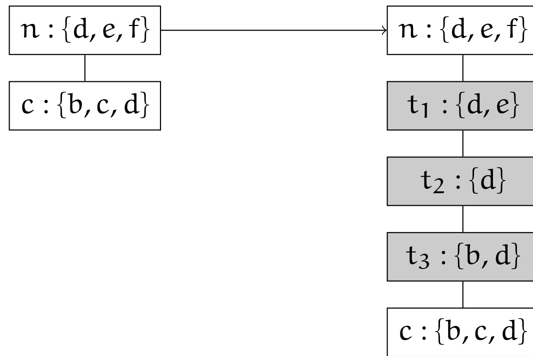
  For each element in $s \in S_1$ we copy the node n to t, remove the element s from $\beta(t)$ and set t as the child of n. Finally t is the new n for the next iteration.

  After all elements of S_1 have been processed, we add the elements of S_2 one by one.

  For each element in $s \in S_2$ we copy the node n to t, add the element s to $\beta(t)$ and set t as the child of n. Finally t is the new n for the next iteration.

  Finally, we set c as the new child of n, and call the normalization algorithm with c as parameter.

  In our example tree decomposition shown below, the sets for eliminating and adding elements look like $S_1 = \{e, f\}$ and $S_2 = \{b, c\}$, leaving out one element of S_2, we have $S_2' = \{b\}$.



Example 3.2.1. *In Figure 3.5, we normalize the tree decomposition $\mathcal{T}_1$ of the graph* G *(Figure 3.2). The nodes inserted by the normalization step are highlighted.*

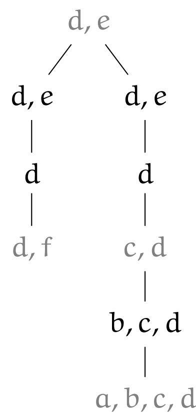


Figure 3.5: Normalized tree decomposition $\mathcal{T}_2$ with width 3 of graph G.

We further extend the notion of a normalized tree decomposition to allow only empty leaf nodes and an empty root node.

Definition 3.2.2. *A tree decomposition is called* extended normalized, *if it is a normalized tree decomposition, and the bag* $\beta(n)$ *of a node* n *is empty for all leaf nodes and the root node.*

The normalization algorithm shown in Listing 3.1 runs in time $O(k^2 \cdot n)$, where k is the width and n the number of nodes in the tree decomposition. By adding empty nodes as root and leaf nodes, it can immediately be used to generate extended normalized tree decompositions. Hence we can conclude that the size of the (extended) normalized tree decomposition is linearly bounded in the number of nodes of an arbitrary tree decomposition.

Listing 3.1: Normalization of a binary tree decomposition in pseudo code.

```
1  normalize(Node n):
2    if n is a leaf node:
3      // do nothing
4    if n is a branch node
5      // save the child nodes
6      c1 = n.child1
7      c2 = n.child2
8      n.child1 = copy of n
9      n.child2 = copy of n
10     // set c1 and c2 as sole children
11     n.child1.child1 = c1
12     remove n.child1.child2
13     n.child2.child1 = c2
14     remove n.child2.child2
15     // recursion
16     normalize(n.child1)
17     normalize(n.child2)
18   else:
19     c = n.child1
20     // compute the differences
21     s1 = β(n) \ β(c)
22     s2 = β(c) \ β(n)
23     if s2 is not empty:
24       remove an arbitrary element of s2
25     else:
26       remove an arbitrary element of s1
27     // remove elements
28     foreach s in s1:
29       t = copy of n
30       β(t) = β(t) \ {s}
31       n.child1 = t
32       n = t
33     // add elements
34     foreach s in s2:
35       t = copy of n
36       β(t) = β(t) ∪ {s}
37       n.child1 = t
38       n = t
39     // set the new child
40     n.child = c
41     // recursion
42     normalize(c)
```

Example 3.2.3. *In the example shown in Figure 3.6, we extend the normalized tree decomposition $\mathcal{T}_2$ of the graph* G*. The nodes inserted by the extension step are highlighted (compared to the normalized tree decomposition $\mathcal{T}_2$).*

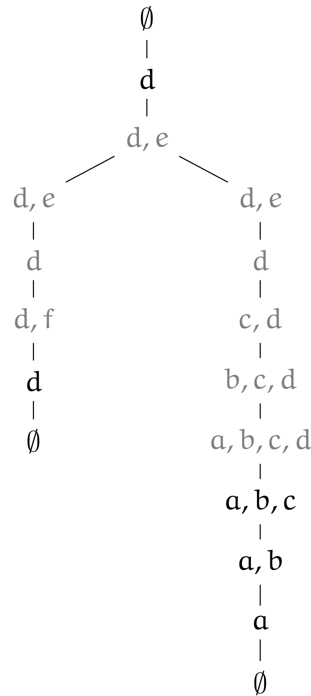


Figure 3.6: Normalized tree decomposition $\mathcal{T}_3$ with width 3 of graph G.

Bodlaender (1996) published an FPT algorithm to compute the tree decomposition of a graph with width k if it exists. That is, for arbitrary but fixed $k \geqslant 1$, it is feasible in linear time to decide if a graph has a treewidth of at most k and, if so, to compute a tree decomposition of width k [6]. From this follows that the size of a tree decomposition is linear in the size of the graph.

3.3 PATH DECOMPOSITIONS

A path decomposition is similar to the notion of tree decompositions. Instead of transforming the graph into a tree, the graph is transformed into a path, that is, a sequence of nodes without branches.

Naturally, the pathwidth is closely related to the treewidth, but cannot be smaller than the treewidth of the same graph.

Since path decompositions have no branches like tree decompositions, algorithms using path decompositions are much simpler to design. A branch-node is often the most complex node to deal with, as we will see in our FPT algorithms in Chapter 5.

A path decomposition has all characteristics of a tree decomposition, it only lacks branch nodes. Hence, a path decomposition is a special case of a tree decomposition: any path decomposition is also a tree decomposition. Thus, algorithms written for tree decompositions work directly on path decompositions but suffer from the potentially larger pathwidth. We make no distinction between tree decompositions and path decompositions in the following.

Example 3.3.1. *The extended normalized pathwidth of the example graph is 2, and the corresponding path decomposition is shown in $\mathcal{P}$ (Figure 3.7).*

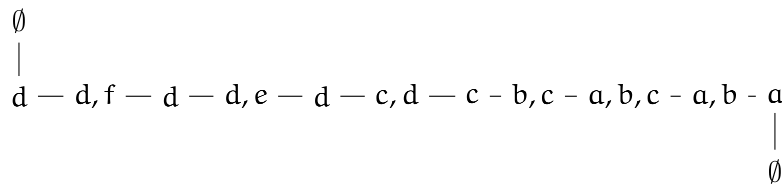


Figure 3.7: Optimal path decomposition $\mathcal{P}$ with pathwidth 2 of graph G with pathwidth 2.

3.4 ALGORITHMS FOR FINDING TREE DECOMPOSITIONS

Algorithms to decompose a graph into a tree or a path have been found over the last decades, we will explain the principles of the important algorithms here.

Exact algorithms like branch and bound algorithms are unpractical for almost all instances. We have to rely on other methods to build the tree decomposition of a graph. In the following we will mention some important algorithms for theoretical considerations (fixed parameter algorithms) and practical considerations (heuristics).

FPT algorithms have good theoretical time bounds, unfortunately the multiplicative constants are very high. So high, that they are currently not considered practical. Heuristics on the other hand may not produce optimal tree decompositions, but compute "good enough" decompositions in a very short time.

FIXED PARAMETER ALGORITHMS Many FPT algorithms are known for several years, their runtime ranges from $O(f(k) \cdot n \cdot \log n)$ (Reed 1992 [66]) to linear runtime for treewidths of 1 to 4 [57, 71]. Bodlaender (1996) [6] improved on an algorithm proposed by Bodlaender and Kloks (1991) [7] and proposed a general linear time FPT algorithm. Yet many of these algorithms are considered unpractical for graphs with a treewidth higher than 2. The algorithm proposed by Bodlaender, for example, has an upper time bound of $O(2^{35k^3} \cdot n)$ [26] where k is the treewidth and n the size of the graph.

HEURISTICS Even though there exists a linear time FPT algorithm, the algorithm has very high multiplicative constants rendering it unpractical for now.

Our FPT algorithms do not rely on the optimal tree decomposition to give the correct answer, so it is natural to use heuristics.

VERTEX ELIMINATION The basic principle behind many tree decomposition heuristics is *vertex elimination*.

The vertex elimination algorithm takes the graph G and an elimination ordering O as input. The elimination ordering is a sequence of nodes of the graph G and determines the tree decomposition. Since there are many possible elimination orderings, it is hard to find the best elimination ordering which then yields the optimal tree decomposition with width of the tree decomposition equal to the treewidth of the graph. This method is an adaption of the vertex elimination algorithm for sparse matrix factorization by Rose and Tarjan (1975) [68]. Vertex elimination for approximating tree decompositions was introduced by Bodlaender et al. (1991) [8].

The algorithm takes a graph $G = (V, E)$ and an elimination ordering $O = (v_1, \dots, v_n)$ of all vertices in V as input, and returns a tree decomposition $\mathcal{T} = (T, \beta)$ of G where $T = (V_T, E_T)$.

The vertex elimination algorithm creates a bag for each node of the graph $G = (V, E)$, that is, the resulting tree decomposition has exactly $|V|$ bags. These bags are initialized in the first step and for each vertex $v \in V$, a set $N(v)$ containing all neighbors (vertices connected with an edge) of v is computed.

After the initialization, the vertices $v \in V$ are considered in the ordering determined by the elimination ordering O. For each vertex v, the vertex itself and all neighbors are added to the bag $\beta(v)$ for the vertex v. Here we see that all vertices in a clique are contained together in at least one bag. Then the vertex v and all

Listing 3.2: Vertex Elimination Algorithm [68]

```
1  E_T = ∅
2  for each v ∈ V:
3    // initialize the bags
4    β(v) = ∅
5    // compute the neighbors of v
6    N(v) = ⋃_{(v,x)∈E}{x}
7
8  for each v ∈ O:
9    // compute the bag of v
10   β(v) = {v} ∪ N(v)
11   // introduce edges between all adjacent nodes of v
12   E = E ∪ (N(v) × N(v))
13   Let w be the next vertex in O such that w ∈ N(v).
14   // connect the nodes
15   E_T = E_T ∪ {(v,w)}
16   // remove v from the graph G
17   G = G \ {v}
18
19 // Choose root node in (V,E_T)
20 return ((V,E_T),β)
```

edges adjacent to v are removed from the graph. All neighbors are then connected with edges in the graph G. The new bag for the vertex v is now connected to the bag of the vertex in its neighborhood which is to be eliminated next. Note that this is not necessarily the next node in the elimination ordering.

After all vertices have been processed, a root node is arbitrarily chosen and the tree decomposition $\mathcal{T}$ is returned.

We will show how the pseudo-code algorithm shown in Listing 3.2 works using an example shown in Example 3.4.1.

Example 3.4.1. *Assume as input the graph* G *of Figure 3.1, and the elimination ordering* $O = (d, e, f, c, b, a)$.

We begin by initializing the bags $\beta(a), \ldots, \beta(f) = \emptyset$*, and by setting the edge relation for the tree* $E_T = \emptyset$.

The first vertex to be eliminated is d*, thus after the first iteration of the main loop, our new graphs is shown in Figure 3.8, note that some edges have been introduced! The bag of* d *is thus defined as* $\beta(d) = \{c, d, e, f\}$ *and an edge between the nodes* d *and* e *in* T *of* $\mathcal{T}$ *is inserted because* e *is the next vertex in the elimination ordering which also occurs in the neighborhood of* d*, that is* $E_T = \{(d, e)\}$.

The next vertex to be eliminated is e*. The bag is now fixed* $\beta(e) = \{c, e, f\}$*, and an edge between* e *and* f *is inserted because* f *is the next vertex in the elimination ordering which also occurs in the neighborhood of* e*, thus* $E_T = \{(d, e), (e, f)\}$*.*

After six iterations the algorithm is done and outputs the tree decomposition shown in Figure 3.9, note that the root of the tree decomposition can be arbitrarily chosen. Further, modern tree decomposition algorithms also collapse, or prune, redundant nodes.

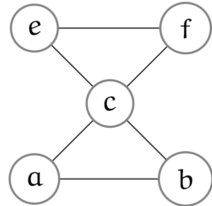


Figure 3.8: Graph after the first step of the vertex elimination algorithm.

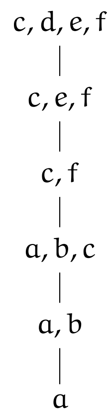


Figure 3.9: Resulting tree decomposition of the vertex elimination algorithm.

To find an elimination ordering that produces (near) optimal tree decompositions is thus the goal of various heuristics, see Schafhauser (2006), Dermaku et al. (2008), and Hammerl (2009) for surveys, new algorithms, and benchmarks [72, 16, 43].

For our implementation (see Chapter 6), we used a tree decomposition library based on a combination of various heuristics, see Dermaku et al. (2008) [16].

3.5 PARAMETERIZED COMPLEXITY

Parameterized complexity is a branch of classical complexity theory. Classical complexity theory considers the time and space behavior of problems under growing problem sizes. Parameterized complexity theory on the other hand determines the time and space requirements for problems with growing problem size but explicitly taking additional parameters into account.

Classical complexity theory is often called "one-dimensional" in the context of parameterized complexity theory, which is then referred to as "two-dimensional".

Example 3.5.1. *Consider the* VERTEX COVER *problem discussed in Chapter 2. In general it is* NP *complete, but if we restrict the size of the cover, that is, the number of vertices in the vertex cover, it becomes (fixed parameter) tractable. Thus, the problem is transformed from "find a minimal vertex cover in graph* G*" to "find a minimal vertex cover in graph* G*, but consider at most* k *vertices".*

There are many possible parameters to choose from. Some popular parameters when dealing with graphs are branch-width, clique-width, and pathwidth (see Hlineny et al. (2007) [45] for a discussion), but treewidth is referred to as "universal parameter" by Fellows (2003) [26] due to its wide applicability.

There is no need for parameters to be single values, they can capture more values at once. A notable example combining three values into one compound parameter is described by Fellows, Gramm, and Niedermeier (2006) [27] for motif search problems.

For a broader overview of the field of parameterized complexity, we refer to the two special issue of "Computer Journal" (Volume 51, Issue 1 and 3) published in 2008, covering survey articles on the various fields where parameterized complexity is used.

Even though the term "fixed parameter algorithm" was coined in the late eighties, fixed parameter algorithms were quite common before that, for example the Integer Linear Programming of Lenstra (1983) uses the number of variables as parameter [51] essentially solving an NP complete problem with an FPT algorithm.

Another example for an FPT algorithm presented before the dawn of parameterized complexity theory is type-checking in ML, which is EXPTIME complete [44], but practically solvable with an FPT algorithm by restricting the nesting depth of type declarations, as shown by Lichtenstein and Pnueli (1985) [54].

Parameterized complexity theory came up in the late eighties and provides the foundations and mathematical tools to analyze and "talk about" such algorithms.

Let us now recast the definitions of parameterized problems and some important complexity classes in the context of parameterized complexity [26, 17].

Definition 3.5.2. *A* parameterized language L *is a subset* $L \subseteq \Sigma^* \times \Sigma^*$. *If* L *is a parameterized language, and* $(x, k) \in L$, *then we refer to* x *as the main part, and* k *as the parameter.*

Based on the definition of a parameterized language, we can now introduce two important complexity classes.

Definition 3.5.3. *A parameterized language is called* fixed parameter tractable, *or in* FPT, *if it can be decided in time* $O(f(k) \cdot n^{O(1)})$ *whether* $(x, k) \in L$, *where* f *is an arbitrary function.*

Definition 3.5.4. *A parameterized language is in* XP, *if it can be decided in time* $O(f(k) \cdot n^{g(k)})$ *whether* $(x, k) \in L$, *where* f *and* g *are arbitrary functions.*

Problems in FPT are considered to be tractable, even though the function $f(k)$ could be exceedingly large. Similarly, problems in XP are considered to be intractable. There are many more classes between FPT and XP, see Niedermeier (2006) [62] and Flum and Grohe (2003) [29]. For many of these classes the relationship between each other is not fully known, but we do know that there are strictly fewer problems in the class FPT than in the class XP, that is, FPT $\subset$ XP, see Downey and Fellows (1999) [17].

3.6 courcelle's theorem

An important result in the context of parameterized complexity is known as *Courcelle's Theorem* [12].

Theorem 3.6.1. *Let the treewidth* $k \geqslant 1$ *and let* φ *be an* MSO *sentence. Then there is a linear-time algorithm, given a graph* $G = (V, E)$ *with a tree decomposition of width at most* k *that decides whether* $G \models \varphi$.

The theorem was later extended to capture counting problems by Arnborg, Lagergren, and Seese (1991) [3] and finally to capture enumeration problems by Flum, Frick, and Grohe (2002) [30].

Example 3.6.2. *From the results shown by Courcelle, Makowsky, and Rotics (2001) [13] the following encoding of the satisfaction check* $X \models F$,

where F *is a propositional formula in* CNF *can be derived (as shown by Gottlob, Pichler, and Wei (2006) [40]). Let the formula* F *be encoded as a finite* τ*-structure* $\mathcal{A}$ *shown in Section 2.4, then we can derive a formula* F' *in* MSO *logic over* $\mathcal{A}$.

$$F' = (\forall c)\mathit{cl}(c) \rightarrow (\exists z)[(\mathit{pos}(z,c) \wedge z \in X) \vee (\mathit{neg}(z,c) \wedge z \notin X)]$$

Thus, we have a finite structure, $\mathcal{A}$*, from which we can derive a tree decomposition with some width* k*, and an encoding of the formula* F *in an* MSO *sentence. We can now conclude, using Courcelle's Theorem, that it is possible to check whether there exists an* X *such that* $X \models F$ *by evaluating the formula* F' *(via tree automata) in linear time.*

The proof that there exists an algorithm to solve the ASP consistency problem in linear time on structures of bounded treewidth is also given by Gottlob, Pichler, and Wei (2006) [40]. They give an encoding (reproduced in Example 3.6.2) of the ASP consistency problem in MSO logic, and hence we can conclude, using Courcelle's Theorem, that ASP consistency is indeed a problem in FPT.

Example 3.6.3. *Let* P *be the* ASP *program encoded as a finite* τ*-structure* $\mathcal{A}$ *(as shown in Section 2.4). Then we can derive a formula* Con *in* MSO *logic over* $\mathcal{A}$ *solving the* ASP *consistency problem for* P*. The formula* Con *is defined over a tree decomposition* $\mathcal{T}_{\mathcal{A}}$ *of the* τ*-structure* $\mathcal{A}$*, the formula* Red *is an encoding of the reduct as given in Definition 2.3.7 (see Gottlob, Pichler, and Wei (2007) [41] for the details).*

$$\begin{aligned} \mathit{Red}(X,Y) =& (\forall r)(\mathit{rl}(r) \rightarrow (\exists z)[(H(r,z) \wedge z \in X) \\ & \vee (\mathit{pB}(r,z) \wedge z \notin X) \vee (\mathit{nB}(r,z) \wedge z \in Y)] \\ \mathit{Con} =& (\exists X)(\mathit{Red}(X,X) \wedge ((\forall Z)[Z \subset X \rightarrow \neg \mathit{Red}(Z,X)])) \end{aligned}$$

Unfortunately, the space requirements to transform an MSO logic formula into a tree automaton are non-elementary in the number of alternations of quantifier blocks of the MSO logic formula, see Libkin (2004) [53]. We call the number of alternations of quantifier blocks *rank* of an formula, that is, the formula F of Example 3.6.2 has rank two.

To evaluate an MSO logic formula F over τ-structure $\mathcal{A}$ using tree decompositions, we need to transform F into an tree automaton, and $\mathcal{T}_{\mathcal{A}}$ into a Σ-tree. Following Flum, Frick, and Grohe (2002) [30], the tree decomposition $\mathcal{T}_{\mathcal{A}}$ has to be transformed into a Σ-tree, $\mathcal{T}^*_{\mathcal{A}}$, called *binary colored tree*. The formula F is then transformed into an MSO logic formula $F_{\mathcal{T}}$ over the Σ-tree $\mathcal{T}^*_{\mathcal{A}}$. Finally, $F_{\mathcal{T}}$ can be transformed into an tree automaton.

In their work, even atomic formulae, when rewritten into MSO logic formulae over binary colored trees, already have some quantifier block alternations. A predicate with two variables has rank four (for example, $\mathtt{pos}(\cdot,\cdot)$ in Example 3.6.2). Taken together, the transformation of the formula in Example 3.6.2 has rank six (the rank of the predicate and the two additional quantifier alternations). Thus the worst-case space requirement of the tree automaton representing the transformed formula, $F_{\mathcal{T}}$, is in the order of

$$O(2^{2^{2^{2^{2^{f(k)}}}}}).$$

The function $f(k)$ is the size of the automaton without quantifiers, which only depends on the width k of the tree decomposition $\mathcal{T}_{\mathcal{A}}$. See Frick and Grohe (2004) [31] for a thorough analysis.

The transformation of the formula in Example 3.6.3 is even more memory intensive, $\mathtt{Con}$ has rank seven.

The size of the tree automata easily exceeds current memory limits. We have to admit that Courcelle's Theorem is an easy way to proof the existence of an FPT algorithm, but it is not practical—even for a treewidth of 1.

4 BASIC STRUCTURE OF FIXED-PARAMETER ALGORITHMS ON TREE DECOMPOSITIONS

Many FPT algorithms on tree decompositions work on a similar basic concept. In this chapter we will explain these basic concepts underlying this work and some approaches previously shown [47, 40].

This basic structure is also the guiding principle behind our framework for implementing algorithms for structures of bounded treewidth in Chapter 6.

4.1 OVERVIEW

The computation is split into several parts.

1. Encode the problem P in a graph G.
2. Build the tree decomposition $\mathcal{T}$ of G.
3. Solve the problem on the tree decomposition.
4. Optionally transform the solution of the tree decomposition $\mathcal{T}$ to a solution of the original problem P.

The solution of the problem on $\mathcal{T}$ is often also a direct solution of the original problem P. There are cases where we need an additional translation step from a solution of the problem on $\mathcal{T}$ to a solution of the original problem. For example, the algorithm proposed by Flum, Frick, and Grohe (2002) [30] requires such a translation.

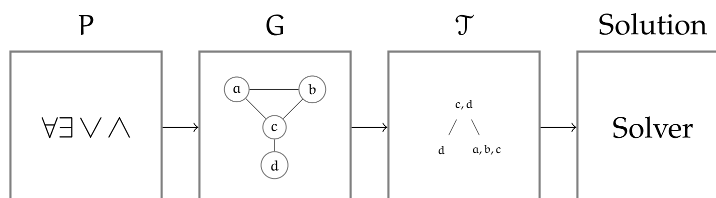


Figure 4.1: General structure of algorithms on tree decompositions.

4.2 graph encoding

There are various ways to encode some problem as a graph. In the context of rules or formulae, there is one dominant method, namely by using *Incidence Graphs*. There are other ways, for example using *Primal Graphs* shown in Chapter 3. See Samer and Szeider (2007) [70] for a discussion of various graph types in a #SAT setting.

To illustrate the graph encoding, we will use the following formula,

$$\varphi = (a \lor b \lor c) \land (\neg b \lor c \lor d) \land (\neg c \lor e) \tag{4.1}$$

and refer to the clauses as

- $C_1 = (a \lor b \lor c)$,
- $C_2 = (\neg b \lor c \lor d)$, and
- $C_3 = (\neg c \lor e)$.

Definition 4.2.1. *An* incidence graph $G = (V, E)$ *of a formula, is a bipartite graph with atoms and clauses as vertices* V*, and an edge between the vertex of an atom,* $v_a \in V$*, and the vertex of a clause,* $v_c \in V$*, if the atom occurs in the clause* $(v_a, v_c) \in E$.

Example 4.2.2. *The example formula* φ *is encoded as incidence graph in Figure 4.2.*

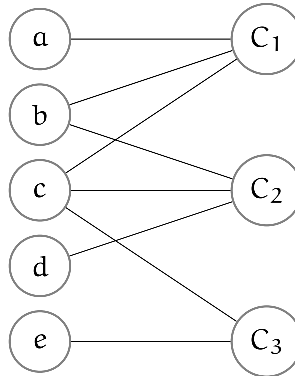


Figure 4.2: Incidence graph G of formula φ.

Note that the information, whether an atom occurs positively or negatively in a clause, is not present in the graph encoding.

DECOMPOSITION After the problem has been encoded as a graph, we decompose it into a tree decomposition of some width. It is preferable to find the optimal tree decomposition of width equal to the treewidth of the graph, but for the correctness of the algorithms, any tree decomposition will do. Methods for finding a tree decomposition have been shown in Chapter 3.

NORMALIZATION Our algorithms work on extended normalized tree decompositions, so an additional normalization step is required. Transforming a tree decomposition into an (extended) normalized tree decomposition can be done in linear time, see Chapter 3.

Example 4.2.3. *To continue our example, we decompose and normalize the graph* G *into a extended normalized tree decomposition* $\mathcal{T}$ *shown in Figure 4.3.*

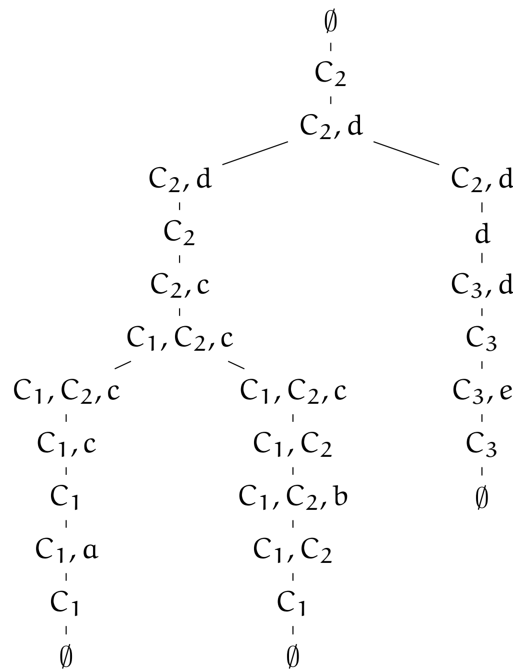


Figure 4.3: Tree decomposition $\mathcal{T}$ with a width of 2 of graph G.

4.3 NODE-TYPES

In the following we describe the possible node-types that may occur in our tree decomposition.

- A *leaf node* (LN) is an empty node with no children.
- A *root node* is an empty node with no parent.
- An *atom introduction node* (AI) has exactly one child and contains exactly one atom more than its child node.
- A *clause introduction node* (CI) has exactly one child and contains exactly one clause more than its child node.
- An *atom removal node* (AR) has exactly one child and contains exactly one atom less than its child node.
- A *clause removal node* (CR) has exactly one child and contains exactly one clause less than its child node.
- A *branch node* (BN) is a node with exactly two children where the bags of the children equal the bag of the branch node.

Being a "root node" is an additional property of one of the removal nodes, since we consider only empty root nodes.

The atom removal and introduction nodes respectively the clause removal and introduction nodes are often augmented with the element that was added or removed. For example, an (a-AI) node introduces the atom a.

Example 4.3.1. *In our example tree decomposition $\mathcal{T}$ of G, we have all node types. See Figure 4.4 for an annotated version.*

4.4 TREE TRAVERSALS

The logic of the algorithms is very problem specific and differs largely from problem to problem. Yet, the basic structure of these algorithms is very similar.

In a dynamic programming fashion, the tree decomposition is traversed from the leaf nodes to the root node (bottom-up), and each node maintains a table of local context information.

Each row of these tables is linked to one or more rows in the child node (except, of course, for leaf nodes).

Depending on the node-type, the table is created in a different way. Thus, the algorithms usually have a large case distinction over the node-type.

For decision problems, there is often no need for a second pass. It is sufficient to check whether there exists a row in the root node that satisfies certain problem specific conditions.

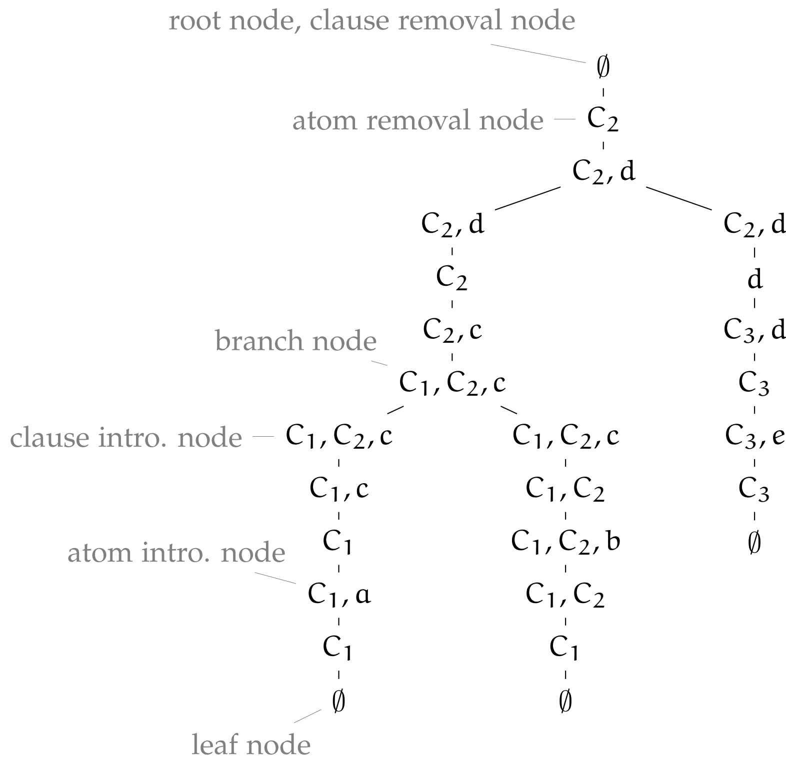


Figure 4.4: Annotated tree decomposition $\mathcal{T}$ with explicit node types.

For counting and enumeration problems, a second pass is needed. The second pass starts at the root node, and computes the number of solutions respectively the solutions itself from the rows of the child nodes. That is, the second pass is a combined top-down and bottom-up pass. Sometimes more than one row of the child node "contributes" to a row in the parent node, we have to compute the solutions of each contributing row, and combine the solutions.

4.4.1 *General Structure*

We will now formalize the tree traversals. We assume the use of an extended normalized tree decomposition $\mathcal{T} = (T, \beta)$.

First, we define $\mathcal{T}$-interpretations, these are the objects we store at each node in T. $\mathcal{T}$-interpretations will then be used to compute the answers for the original problem.

Definition 4.4.1. *A* $\mathcal{T}$-interpretation*, is a tuple* (n, ξ) *where* $n \in T$ *is a node, and* ξ *some problem specific context information bounded in size by the treewidth.*

Next, we need a function to transform the $\mathcal{T}$-interpretation into a solution of the original problem. For this purpose, we define the function $\mathcal{E}(\theta)$.

Definition 4.4.2. *A function* $\mathcal{E}(\theta)$ *is called* $\mathcal{T}$*-interpretation-mapping if it maps a* $\mathcal{T}$*-interpretation* $\theta = (n, \xi)$ *to solutions of the encoded problem, and for each two distinct* $\mathcal{T}$*-interpretations* θ_1 *and* θ_2*, the condition*

$$\mathcal{E}(\theta_1) \cap \mathcal{E}(\theta_2) = \emptyset$$

holds.

Definition 4.4.3. *A* $\mathcal{T}$*-interpretation* $\theta = (n, \xi)$ *is called* $\mathcal{T}$-model *iff* $\mathcal{E}(\theta) \neq \emptyset$*. If* n *is the root node of* T*, the* $\mathcal{T}$*-model is called* $\mathcal{T}$-root-model.

For the counting problem we define the function $\#(\theta)$ to be the cardinality of the set $\mathcal{E}(\theta)$.

$$\#(\theta) = |\mathcal{E}(\theta)|$$

To compute the $\mathcal{E}(\cdot)$, or $\#(\cdot)$ efficiently, we introduce the so called *contributes-relation*, $\prec_{\mathcal{T}}$, between $\mathcal{T}$-interpretations. These $\mathcal{T}$-interpretations can then be used to compute the $\mathcal{E}(\cdot)$ or $\#(\cdot)$ in a bottom-up manner.

Definition 4.4.4. *A* $\mathcal{T}$*-relation,* $\prec_{\mathcal{T}}$*, is a relation between two (respectively three)* $\mathcal{T}$*-interpretations* $\theta' \prec_{\mathcal{T}} \theta$ *(respectively* $(\theta_1, \theta_2) \prec_{\mathcal{T}} \theta$*) if* $\theta = (n, \xi)$ *and* $\theta' = (n', \xi')$ *(respectively* $\theta_1 = (n_1, \xi_1)$ *and* $\theta_2 = (n_2, \xi_2)$*) and* n' *is a child of* n *(respectively* n_1 *and* n_2 *are children of* n*) in* T *of* $\mathcal{T}$*.*

Definition 4.4.5. *A* $\mathcal{T}$*-relation,* $\prec_{\mathcal{T}}$*, is called* contributes-relation*, if the following conditions hold.*
Let θ *and* θ' *(respectively* θ_1 *and* θ_2*) be* $\mathcal{T}$*-interpretations, such that* $\theta' \prec_{\mathcal{T}} \theta$ *(respectively* $(\theta_1, \theta_2) \prec_{\mathcal{T}} \theta$*). Then,* θ *is a* $\mathcal{T}$*-model iff* θ' *is* $\mathcal{T}$*-model (respectively both* θ' *and* θ'' *are* $\mathcal{T}$*-models).*

The main part of each algorithm following this structure is to define the contributes-relation and to find a way to compute $\mathcal{E}(\cdot)$ respectively $\#(\cdot)$ using $\prec_{\mathcal{T}}$ in an efficient manner.

The computation of the mapping $\#(\theta)$ where $\theta = (n, \xi)$ should follow a similar form.

4.4.2 *Counting and Enumerating Models for Propositional Logic*

Before we dive into ASP, we will informally apply the general algorithm structure to the problems discussed by Samer and Szeider (2007) [70], namely #SAT and extend it through our algorithm to the enumeration of all models.

We assume the use of an extended normalized tree decomposition $\mathcal{T} = (T, \beta)$, hence we deal with the node types described in Section 4.3.

We will use A_φ to refer to the atoms in a propositional formula φ, and C_φ to refer to the clauses of φ.

Definition 4.4.6. *The* $\mathcal{T}$-interpretation *is a tuple* (n, M) *where* $n \in T$ *is a node and* $M \subseteq \beta(n)$ *is called* assignment*.*

Let M be a set of clauses and atoms, $M \subseteq C_\varphi \cup A_\varphi$. We introduce some shortcuts:

$$A_{\beta(n)} = A_\varphi \cap \beta(n) \quad \text{set of atoms of the bag } \beta(n)$$
$$C_{\beta(n)} = C_\varphi \cap \beta(n) \quad \text{set of clauses of the bag } \beta(n)$$
$$A_M = A_\varphi \cap M \quad \text{set of atoms in } M$$
$$C_M = C_\varphi \cap M \quad \text{set of clauses in } M$$

To refer to certain (sub-)sets of the tree T, we define the following abbreviations, let n be a node in T:

$$A_{(n)} = A_\varphi \cap \bigcup_{m \in T_n} \beta(m) \quad \text{set of atoms in the subtree } T_n$$
$$C_{(n)} = C_\varphi \cap \bigcup_{m \in T_n} \beta(m) \quad \text{set of clauses in the subtree } T_n$$
$$A_{[n]} = A_{(n)} \setminus \beta(n) \quad \text{set of atoms below the node } n \text{ in } T_n$$
$$C_{[n]} = C_{(n)} \setminus \beta(n) \quad \text{set clauses below the node } n \text{ in } T_n$$

For the enumeration problem of propositional logic, we have the following mapping between an $\mathcal{T}$-interpretation and the set interpretations.

Definition 4.4.7. *For a* $\mathcal{T}$*-interpretation* $\theta = (n, M)$*, let the set of interpretations* $\mathcal{E}(\theta)$ *be defined as*

$$\mathcal{E}(\theta) = \{A_M \cup K \mid K \subseteq A_{[n]}, \mathit{SAT}_n(A_M \cup K) = C_M \cup C_{[n]}\}$$

where

$$\mathit{SAT}_n(I) = \{c \mid c \in C_{(n)}, I \models_{A_{(n)}} c\}$$

$$\begin{aligned}
(a\text{-AR}) &: M = M' \setminus \{a\} \\
(c\text{-CR}) &: c \in M', M = M' \setminus \{c\} \\
(a\text{-AI}) &: M = M' \cup \{c \in C_{\beta(n)} \mid \overline{a} \in c\} \\
&\quad \text{or} \\
&\quad M = M' \cup \{a\} \cup \{c \in C_{\beta(n)} \mid a \in c\} \\
(c\text{-CI}) &: M = \begin{cases} N \cup \{c\} & \text{if } c \cap (A_N \cup \overline{(A_{\beta(n)} \setminus A_N)}) \neq \emptyset \\ N & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 4.5: Conditions for $(n', M') \prec_{\mathcal{T}} (n, M)$ for the various node-types of n.

Theorem 4.4.8. *For the root-model θ of $\mathcal{T}$, the following holds.*

$$I \models \varphi \iff I \in \mathcal{E}(\theta)$$

We define the contributes-relation $\prec_{\mathcal{T}}$ on $\mathcal{T}$-interpretations.

Definition 4.4.9. *For $\mathcal{T}$-interpretations $\theta = (n, M)$ and $\theta' = (n', M')$, we have $\theta' \prec_{\mathcal{T}} \theta$ iff n has a single child n', and (depending on the node type of n) the conditions as depicted in the table of Figure 4.5 are fulfilled.*

Note that in case n is an (AI)-node, there are two ways how θ' and θ can be related to each other.

For branch nodes, we partially extend (with a slight abuse of notation) $\prec_{\mathcal{T}}$ to a ternary relation as follows.

Definition 4.4.10. *For $\mathcal{T}$-interpretations of the parent $\theta = (n, M)$, and its children $\theta_1 = (n_1, M_1)$, $\theta_2 = (n_2, M_2)$ we have $(\theta_1, \theta_2) \prec_{\mathcal{T}} \theta$ iff the following conditions hold:*

1. *n_1 and n_2 are the two children of n;*
2. *$A_{M_1} = A_{M_2}$ and $M = M_1 \cup M_2$;*

We will now show how to compute the $\mathcal{T}$-interpretation-mapping $\mathcal{E}(\cdot)$ following the contributes-relation.

The following theorems are now immediate, Theorem 4.4.12 has already been shown in Samer and Szeider (2007) [70], Theorem 4.4.11 is an easy extension to it. Note however, that Samer and Szeider carry the clauses *not* satisfied whereas we carry the satisfied clauses up the tree.

Theorem 4.4.11. *Let* θ *be a* $\mathcal{T}$*-interpretation for node* n*. The* $\mathcal{T}$*-interpretation-mapping* $\mathcal{E}(\theta)$ *is equivalent to the following bottom-up definition.*

For leaf nodes $n \in T$*, tree models can be determined trivially. Since we have no objects to talk about, the only valid* $\mathcal{T}$*-model is* $M = \emptyset$*, and hence* $\mathcal{E}(\theta) = \{\emptyset\}$*.*

$$\mathcal{E}(\theta) = \begin{cases} \{\emptyset\} & (1) \\ \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta') & (2) \\ \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \{I \cup \{a\} \mid I \in \mathcal{E}(\theta')\} & (3) \\ \bigcup_{(\theta_1, \theta_2) \prec_{\mathcal{T}} \theta} \{I_1 \cup I_2 \mid I_1 \in \mathcal{E}(\theta_1), I_2 \in \mathcal{E}(\theta_2)\} & (4) \end{cases}$$

1. n *is an (L) node.*
2. n *is a (CR), (CI), (AR) node, or an (*a*-AI) node and* $a \notin M$*.*
3. n *is an (*a*-AI) node and* $a \in M$*.*
4. n *is a (B) node.*

We can establish a similar equivalence for $\#(\cdot)$.

Theorem 4.4.12. *Let* θ *be a* $\mathcal{T}$*-interpretation for node* n*. Given that* $\mathcal{E}(\theta_1) \cap \mathcal{E}(\theta_2) = \emptyset$ *for two distinct* $\mathcal{T}$*-interpretations* θ_1 *and* θ_2*, the following holds. If* θ *is not a* $\mathcal{T}$*-model, let* $\#(\theta) = 0$*, otherwise*

$$\#(\theta) = \begin{cases} 1 & \text{if } n \text{ is leaf node,} \\ \sum_{\theta' \prec_{\mathcal{T}} \theta} \#(\theta') & \text{if } n \text{ has one child,} \\ \sum_{(\theta', \theta'') \prec_{\mathcal{T}} \theta} \#(\theta') \cdot \#(\theta'') & \text{if } n \text{ is branch node.} \end{cases}$$

5 ANSWER SET PROGRAMMING WITH BOUNDED TREEWIDTH

This chapter presents the new algorithms for ASP consistency, enumeration and counting. We will use a running example, shown in Listing 5.1 (an abstract version of lines 1-5 of Listing 2.2). We refer to the rules by their line number, that is, to the rule in line 1 of Listing 5.1 as rule 1 (or r_1 for short).

Listing 5.1: Program P.

```
1 a.
2 b :- a, c.
3 e :- a, d.
4 c v d :- a, not f.
```

The program P in Listing 5.1 has two answer sets $\{a, d, e\}$ and $\{a, b, c\}$.

We proceed as sketched in Chapter 4. First, we show how to build a tree decomposition, $\mathcal{T}$, from a given ASP program R. Next, we define the mathematical objects ($\mathcal{T}$-*interpretations*) which will underly our algorithms.

We construct a mapping $\mathcal{E}(\cdot)$ from $\mathcal{T}$-interpretations to (standard) interpretations and observe that a certain subset S of the $\mathcal{T}$-interpretations characterizes the answer sets of the program R. However, we never compute $\mathcal{E}(\cdot)$ explicitly. Instead, we define a relation, the *contributes relation* ($\prec_{\mathcal{T}}$) in Definitions 5.2.8 and 5.2.10, along the structure of $\mathcal{T}$, in order to efficiently compute S in a bottom-up manner via so-called $\mathcal{T}$-*models*, a subset of the $\mathcal{T}$ interpretations, which is shown in Lemma 5.2.12.

Finally, we show how to use this method to decide ASP consistency (Section 5.2.3), and to count (Section 5.2.4) and enumerate (Section 5.2.5) the answer sets of program R from the tree decomposition $\mathcal{T}$ of R.

5.1 TREE DECOMPOSITIONS OF ASP PROGRAMS.

We use an adapted version of the incidence graphs shown in Chapter 3 to represent the programs. Given a program R, such a graph has $R \cup At(R)$ as vertices, and as edges all pairs (a, r) where atom a appears in a rule r of R.

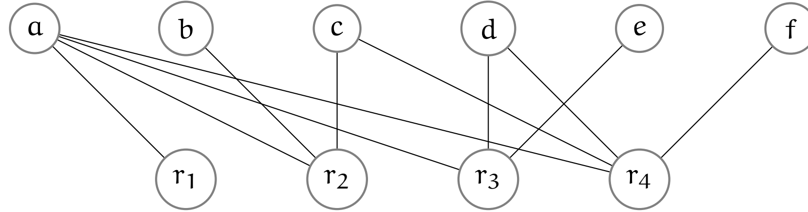


Figure 5.1: Incidence graph G_P of example program P.

From the incidence graph G we build the extended normalized tree decomposition $\mathcal{T}$. The node-types also have to be adapted for the context of ASP programming. That is, we distinguish the following six types of nodes.

- Atom introduction (a-AI)
- Rule introduction (r-RI)
- Atom removal (a-AR)
- Rule removal (r-RR)
- Branch node (B)
- Leaf nodes (L)

The "atom" and "rule" types are augmented with the element (either an atom a or a rule r) which is removed or added compared to the bag of the child node. For example, (a-AI) refers to the node that introduces the atom a compared to its child.

Example 5.1.1. *Figure 5.1 shows the incidence graph G_P of program P and Figure 5.2 shows the extended normalized tree decomposition $\mathcal{T}$ of G_P having width 3. Indeed, we have $tw(G_P) = 3$, so the tree decomposition $\mathcal{T}$ is optimal.*

Examples for node types are n_{18} as (L) node, n_{15} as (b-AR) node, n_{17} as (b-AI) node, n_2 as (r_2-RR) node, n_{16} as (r_2-RI) node, and n_7 as (B) node.

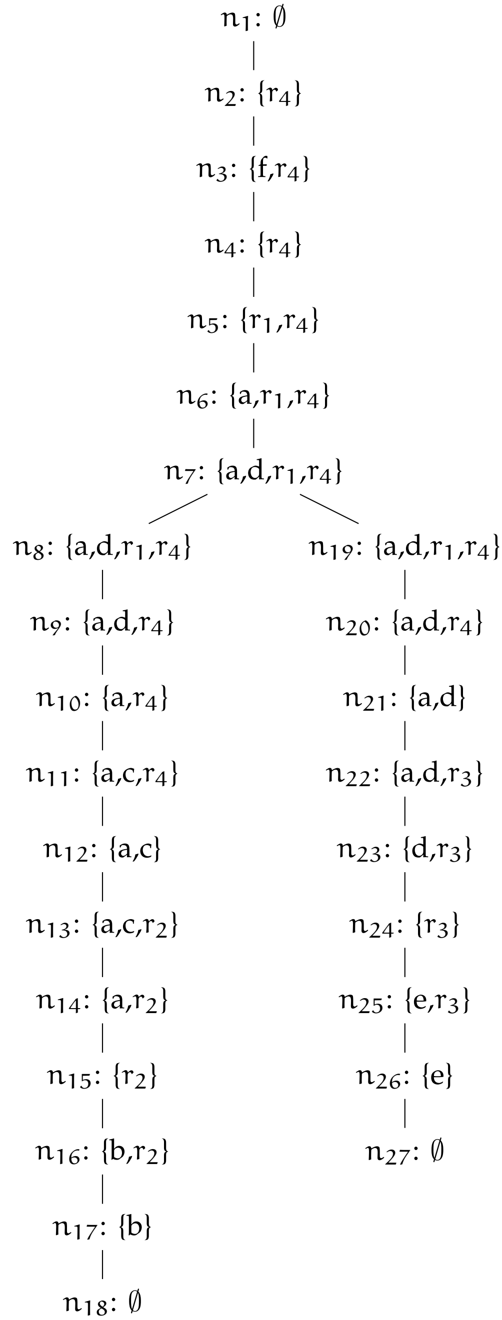


Figure 5.2: The *extended normalized tree decomposition* $\mathcal{T}$ of G_P (Figure 5.1).

5.2 THE DYNAMIC PROGRAMMING APPROACH FOR ASP

Let n be a node in T of $\mathcal{T}$ and M be a set of rules and atoms, $M \subseteq R \cup At(R)$. We introduce some shortcuts:

$$\begin{aligned} A_{\beta(n)} &= At(R) \cap \beta(n) && \text{set of atoms of the bag } \beta(n) \\ R_{\beta(n)} &= R \cap \beta(n) && \text{set of rules of the bag } \beta(n) \\ A_M &= At(R) \cap M && \text{set of atoms in } M \\ R_M &= R \cap M && \text{set of rules in } M \end{aligned}$$

We refer to the root node of T as *rt*. For a node $n \in T$, we denote the subtree rooted at n as T_n.

To refer to certain (sub-)sets of the tree T, we define the following abbreviations, let n be a node in T:

$$\begin{aligned} A_{(n)} &= At(R) \cap \bigcup_{m \in T_n} \beta(m) && \text{set of atoms in the subtree } T_n \\ R_{(n)} &= R \cap \bigcup_{m \in T_n} \beta(m) && \text{set of rules in the subtree } T_n \\ A_{[n]} &= A_{(n)} \setminus \beta(n) && \text{set of atoms no more visible} \\ R_{[n]} &= R_{(n)} \setminus \beta(n) && \text{set of rules no more visible} \end{aligned}$$

5.2.1 *Tree Interpretations*

Definition 5.2.1. *A* $\mathcal{T}$-interpretation *is a tuple* $(n, M, \mathcal{C})$ *where* $n \in T$ *is a node,* $M \subseteq \beta(n)$ *is called* assignment*, and* $\mathcal{C} \subseteq 2^{\beta(n)}$ *is a set of* certificates*.*

The basic intuition behind $\mathcal{T}$-interpretations is that the assignment M of a $\mathcal{T}$-interpretation $(n, M, \mathcal{C})$ contains an interpretation A_M over $A_{\beta(n)}$, that implicitly refers to interpretations I over $A_{(n)}$, together with rules $r \in R_{\beta(n)}$ satisfied by I, i.e., $I \models_{A_{(n)}} r$.

A set of certificates $\mathcal{C}$ can be understood as a set of assignments and it carries interpretations (together with satisfied rules in $(R_{\beta(n)})^I$) which are in a certain subset-relation to M. The following definitions make this more precise.

Definition 5.2.2. *Given a node* $n \in T$ *and two sets* $I, J \subseteq A_{(n)}$*, define*

$$\begin{aligned} \mathit{SAT}_n(I) &= \{r \mid r \in R_{(n)}, I \models_{A_{(n)}} r\} \\ \mathit{RSAT}_n(J, I) &= \{r \mid r \in R_{(n)}, J \models_{A_{(n)}} r \text{ or } B^-(r) \cap I \neq \emptyset\}. \end{aligned}$$

Intuitively, $\mathsf{SAT}_n(I)$ yields those rules of R which occur in bags of the subtree T_n and are satisfied by I. Analogously, $\mathsf{RSAT}_n(J, I)$

yields such rules which are either satisfied by J or not contained in the reduct R^I, thus we can view them as satisfied by J in a trivial way.

Definition 5.2.3. *Let* $\theta = (n, M, \mathcal{C})$ *be a* $\mathcal{T}$*-interpretation, and* $I \subseteq A_{(n)}$ *a set of atoms. We define*

$$\begin{aligned} e_n(M) = \{&A_M \cup K \mid K \subseteq A_{[n]}, \\ &SAT_n(A_M \cup K) = R_M \cup R_{[n]}\} \\ re_n(M, I) = \{&A_M \cup K \mid K \subseteq A_{[n]}, \\ &RSAT_n(A_M \cup K, I) = R_M \cup R_{[n]}\}. \end{aligned}$$

The rationale behind $e_n(M)$ is to yield those extensions of the interpretation A_M stored in M of a $\mathcal{T}$-interpretation $\theta = (n, M, \mathcal{C})$ (i.e., over all atoms occurring in bags of T_n), such that the rules R_M plus *all* rules in $R_{[n]}$ (i.e., all rules occurring in bags of T_n, but below n) are satisfied by I. A similar idea is followed by $re_n(M, I)$ which additionally takes the concept of reduct into account.

We are now ready to define the mapping $\mathcal{E}(\cdot)$ and shall see that for certain $\mathcal{T}$-interpretations θ the relation $\mathcal{E}(\theta) \subseteq \mathcal{AS}(R)$ holds.

Definition 5.2.4. *For a* $\mathcal{T}$*-interpretation* $\theta = (n, M, \mathcal{C})$*, let the set of interpretations* $\mathcal{E}(\cdot)$ *be defined as*

$$\begin{aligned} \mathcal{E}(\theta) =&\{I \mid I \in e_n(M) \text{ and} \\ &\forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in re_n(C, I)]\}. \end{aligned}$$

In the following we will refer to $I \in e_n(M)$ as the assignment condition, Condition $\mathcal{A}$, and to $\forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in re_n(C, I)]$ as the certificate condition wrt. I, Condition $\mathcal{C}$, for a $\mathcal{T}$-interpretation $\theta = (n, M, \mathcal{C})$.

Definition 5.2.5. *A* $\mathcal{T}$*-interpretation* $(n, M, \mathcal{C})$ *is called* root model *for* $\mathcal{T}$ *iff* $n = rt$*,* $M = \emptyset$*, and* $\mathcal{C} = \emptyset$*.*

Note that, since the root node has no elements in its bag, we can only have at most one root model, but a second $\mathcal{T}$-interpretation. This second $\mathcal{T}$-interpretation, $\theta = (rt, \emptyset, \{\emptyset\})$, does not satisfy the definition of an answer set given in Definition 2.3.8 since there exists a $C \in \mathcal{C}$, there exists a $J \subset I$ such that $J \models R^I$, hence I is no answer set for R.

Theorem 5.2.6. *For the root model* $\theta = (rt, \emptyset, \emptyset)$ *of* $\mathcal{T}$*, the equivalence* $\mathcal{AS}(R) = \mathcal{E}(\theta)$ *holds.*

$$
\begin{aligned}
(a\text{-AR}): &\ M = M' \setminus \{a\} \\
&\ \mathcal{C} = \{C \setminus \{a\} \mid C \in \mathcal{C}'\} \\
(r\text{-RR}): &\ r \in M', M = M' \setminus \{r\} \\
&\ \mathcal{C} = \{C \setminus \{r\} \mid C \in \mathcal{C}', r \in C\} \\
(a\text{-AI}): &\ M = M' \times_n a \\
&\ \mathcal{C} = \{C \times_n a \mid C \in \mathcal{C}'\}, \\
&\ \text{ or} \\
&\ M = M' +_n a \\
&\ \mathcal{C} = \{(M' \times_n a) \cup R^{B^-}_{a,n}\} \cup \\
&\ \quad \{(C +_n a) \cup R^{B^-}_{a,n}, (C \times_n a) \cup R^{B^-}_{a,n} \mid C \in \mathcal{C}'\} \\
(r\text{-RI}): &\ M = M' \uplus_n r \\
&\ \mathcal{C} = \begin{cases} \{C \cup \{r\} \mid C \in \mathcal{C}'\} & \text{if } \textsc{b}^-(r) \cap A_M \neq \emptyset \\ \{C \uplus_n r \mid C \in \mathcal{C}'\} & \text{otherwise} \end{cases}
\end{aligned}
$$

where

$$
\begin{aligned}
N +_n a &= N \cup \{a\} \cup R^+_{a,n} \\
N \uplus_n r &= \begin{cases} N \cup \{r\} & \text{if } r \cap (A_N \cup \overline{(A_{\beta(n)} \setminus A_N)}) \neq \emptyset \\ N & \text{otherwise} \end{cases} \\
N \times_n a &= N \cup R^+_{\overline{a},n} \\
R^{B^-}_{a,n} &= \{r \in R_{\beta(n)} \mid a \in \textsc{b}^-(r)\} \\
R^+_{a,n} &= \{r \in R_{\beta(n)} \mid a \in r\}
\end{aligned}
$$

Figure 5.3: Conditions for $(n', M', \mathcal{C}') \prec_{\mathcal{T}} (n, M, \mathcal{C})$ for the various node-types of n.

Proof. Suppose $\theta = (rt, \emptyset, \emptyset)$ is the root model of $\mathcal{T}$. Then, by substituting the definitions we have the following equivalence.

$$\begin{aligned}
\mathcal{E}(\theta) = &\{I \mid I \in \mathsf{e}_{rt}(\emptyset), \forall C \subseteq \emptyset(C \in \emptyset \iff \\
&\exists J \subset I \text{ s.t. } J \in \mathsf{re}_n(\emptyset, I))\} \\
= &\{I \mid I \in \{K \mid K \subseteq At(R), \mathsf{SAT}_{rt}(K) = R\}, \\
&\forall J \in \{K \mid K \subseteq At(R), \mathsf{RSAT}_{rt}(K, I) = R\} \implies J \not\subset I\} \\
= &\{I \mid I \subseteq At(R), \mathsf{SAT}_{rt}(I) = R, \\
&\forall J \subseteq At(R)[\mathsf{RSAT}_{rt}(J, I) = R \implies J \not\subset I]\} \\
= &\{I \mid I \subseteq At(R), I \models R, \\
&\forall J \subseteq At(R)(\{r \mid r \in R, \\
&J \models r \vee B^-(r) \cap I \neq \emptyset\} = R \implies J \not\subset I)\} \\
= &\{I \mid I \subseteq At(R), I \models R, \\
&\forall J \subseteq At(R)(J \models R^I \implies J \not\subset I)\} \\
= &\mathcal{AS}(R)
\end{aligned}$$

□

5.2.2 *Tree Models*

Definition 5.2.7. *A $\mathcal{T}$-interpretation θ is called* $\mathcal{T}$-model *iff $\mathcal{E}(\theta) \neq \emptyset$.*

For leaf nodes n, tree models can be determined trivially. Since we have no objects to talk about, the only valid $\mathcal{T}$-model is $M = \emptyset$ and $\mathcal{C} = \emptyset$, resulting in $\mathcal{E}(\theta) = \{\emptyset\}$. For the $\mathcal{T}$-interpretation $\theta' = (n, \emptyset, \{\emptyset\})$, Condition $\mathcal{C}$ wrt. I, where $I = \emptyset$, does not hold since there exists no $J \subset I$, but a certificate.

Next, we define the relation $\prec_{\mathcal{T}}$ between $\mathcal{T}$-interpretations. The concrete definition depends on the node type.

Definition 5.2.8. *For $\mathcal{T}$-interpretations $\theta = (n, M, \mathcal{C})$ and $\theta' = (n', M', \mathcal{C}')$, we have $\theta' \prec_{\mathcal{T}} \theta$ iff n has a single child n', and (depending on the node type of n) the conditions as depicted in the table of Figure 5.3 are fulfilled.*

Note that, if n is an (a-AI)-node, there are two ways how θ' and θ can be related to each other, once a is true, and once where a is false.

Example 5.2.9. *We will show how to construct one $\mathcal{T}$-model for each node type. The full tree decomposition with the dynamic tables is shown in Figure 5.4.*

(L) NODE *The only viable $\mathcal{T}$-model for the node n_{18} is*

$$\theta_{n_{18}} = (n_{18}, \emptyset, \emptyset).$$

(b-AI) NODE *For the node n_{17}, a (b-AI) node, we have two ways to construct a $\mathcal{T}$-model. The atom b can either be added positively or negatively.*

$$\theta_{n_{17a}} = (n_{17}, \emptyset, \emptyset)$$
$$\theta_{n_{17b}} = (n_{17}, \{b\}, \{\emptyset\})$$

The relation $\theta_{n_{18}} \prec_{\mathcal{T}} \theta_{n_{17a}}$ holds since $\emptyset \times_{n_{17}} b = \emptyset$ and there is no certificate in $\theta_{n_{18}}$.

The relation $\theta_{n_{18}} \prec_{\mathcal{T}} \theta_{n_{17b}}$ holds since $\emptyset +_{n_{17}} b = \{b\}$ and the only certificate to be added is $\emptyset \times_{n_{17}} b = \emptyset$.

(r_2-RI) NODE *Node n_{16} introduces the rule r_2. Since r_2 is satisfied by b, we add it to the assignment of $\theta_{n_{16b}}$, the rule r_2 is not satisfied by $\emptyset$, so it is not added to the assignment of $\theta_{n_{16a}}$. Further the rule r_2 is added to those assignments and certificates that satisfy it in some way.*

$$\theta_{n_{16a}} = (n_{16}, \emptyset, \emptyset)$$
$$\theta_{n_{16b}} = (n_{16}, \{b, r_2\}, \{\emptyset\})$$

The relation $\theta_{n_{17a}} \prec_{\mathcal{T}} \theta_{n_{16a}}$ holds since $\emptyset \uplus_{n_{16}} r_2 = \emptyset$ and there is no certificate in $\theta_{n_{17a}}$.

The relation $\theta_{n_{17b}} \prec_{\mathcal{T}} \theta_{n_{16b}}$ holds since $\{b\} +_{n_{16}} r_2 = \{b, r_2\}$ because $b \in r_2$. The certificate $\emptyset$ does not satisfy r_2, so $\emptyset \uplus_{n_{16}} r_2 = \emptyset$.

(b-AR) NODE *The next node is n_{15}, a b-AR node. The atom is simply removed from all assignments and certificates.*

$$\theta_{n_{15a}} = (n_{15}, \emptyset, \emptyset)$$
$$\theta_{n_{15b}} = (n_{15}, \{r_2\}, \{\emptyset\})$$

An atom removal node is very simple and drops all occurrences of the removed atom. Thus, it is easy to see that $\theta_{n_{16a}} \prec_{\mathcal{T}} \theta_{n_{15a}}$ and $\theta_{n_{16b}} \prec_{\mathcal{T}} \theta_{n_{15b}}$ hold.

(r_2-RR) NODE *We jump up to node n_{12}, an (r_2-RR) node. Let $\theta_{n_{13}} = (n_{13}, \{a, r_2\}, \{\{r_2\}\})$ be a $\mathcal{T}$-model of n_{13}. The rule is simply removed from the assignment and certificates.*

$$\theta_{n_{12}} = (n_{12}, \{a\}, \{\emptyset\})$$

Only the $\mathcal{T}$*-models that include* r_2 *in the assignment are contained in the relation* $\prec_{\mathcal{T}}$*. Let*

$$\theta_{n_{13b}} = (n_{13}, \{a, c\}, \{\{r_2\}, \{a, r_2\}, \{c, r_2\}\}),$$

since the assignment $\{a, c\}$ *does not include* r_2*, the* $\mathcal{T}$*-model* $\theta_{n_{13b}}$ *is not in relation with any* $\mathcal{T}$*-model of* n_{12}*.*

If a certificate does not include the rule r_2*, only the certificate is removed, not the whole* $\mathcal{T}$*-model. Let*

$$\theta_{n_{13c}} = (n_{13}, \{a, c, r_2\}, \{\{r_2\}, \{a, r_2\}, \{a, c\}, \{c, r_2\}\}),$$

we remove r_2 *from all assignments and certificates that include* r_2*, and drop those which do not include* r_2*. In this case the certificate* $\{a, c\}$ *is dropped.*

$$\theta_{n_{12b}} = (n_{12}, \{a, c\}, \{\emptyset, \{a\}, \{c\}\})$$

Thus $\theta_{n_{13c}} \prec_{\mathcal{T}} \theta_{n_{12b}}$ *holds.*

For branch nodes, we partially extend (with a slight abuse of notation) $\prec_{\mathcal{T}}$ to a ternary relation as follows.

Definition 5.2.10. *For* $\mathcal{T}$*-interpretations of the* $\theta = (n, M, \mathcal{C})$*,* $\theta_1 = (n_1, M_1, \mathcal{C}_1)$*, and* $\theta_2 = (n_2, M_2, \mathcal{C}_2)$ *we have* $(\theta_1, \theta_2) \prec_{\mathcal{T}} \theta$ *iff the following conditions hold:*

1. n_1 *and* n_2 *are the two children of* n*;*
2. $A_{M_1} = A_{M_2}$ *and* $M = M_1 \cup M_2$*;*
3. $\mathcal{C}$ *is given by the set* $(\mathcal{C}_1 \bowtie \mathcal{C}_2) \cup (\{M_1\} \bowtie \mathcal{C}_2) \cup (\mathcal{C}_1 \bowtie \{M_2\})$*; where* $\mathcal{C} \bowtie \mathcal{C}'$ *is defined as*

$$\{C \cup C' \mid C \in \mathcal{C}, C' \in \mathcal{C}', A_C = A_{C'}\}.$$

Example 5.2.11. *We will show the construction of a* $\mathcal{T}$*-model for the branch node* n_7*.*

Consider two $\mathcal{T}$*-models, one from* n_8 *and one from* n_{19}*, both nodes are children of* n_7*.*

$$\theta_{n_8} = (n_8, \{a, r_1, r_4\}, \{\{r_4\}, \{a, r_1\}\})$$
$$\theta_{n_{19}} = (n_{19}, \{a, r_1\}, \{\{r_4\}\})$$

We construct now the combined $\mathcal{T}$*-model* $\theta_{n_7} = (n_7, M, \mathcal{C})$ *for* n_7*. Note that the atom* a *is the only atom involved, and it is true in the assignment of both* $\mathcal{T}$*-models, thus condition 2 of Definition 5.2.10 is fulfilled. The new assignment is thus* $M = \{a, r_1, r_4\}$*.*

For the certificates, we first join the certificates of θ_{n_8} *with the certificates of* $\theta_{n_{19}}$. *The only certificates for which the atoms match, are* $\{r4\}$ *from* θ_{n_8} *and* $\{r4\}$ *from* $\theta_{n_{19}}$, *thus the combined certificate looks like this:*

$$\mathcal{C}_{n_8} \bowtie \mathcal{C}_{n_{19}} = \{r4\}$$

The assignment M_{n_8} *of* θ_{n_8} *has no matching atoms in a certificate of* n_{19}*:*

$$\{M_{n_8}\} \bowtie \mathcal{C}_{n_{19}} = \emptyset.$$

The assignment $M_{n_{19}} = \{a, r_1\}$ *of* θ_{n_8} *has a matching certificate in the set of certificates* $\mathcal{C}_{n_{19}}$ *of* $\theta_{n_{19}}$, *namely* $\{a, r_1\}$*:*

$$\mathcal{C}_{n_8} \bowtie \{M_{n_{19}}\} = \{a, r_1\}.$$

Thus, the combined certificate for the branch node n_8 *has the following form:*

$$\begin{aligned}\mathcal{C} &= (\mathcal{C}_{n_8} \bowtie \mathcal{C}_{n_{19}}) \cup (\{M_{n_8}\} \bowtie \mathcal{C}_{n_{19}}) \cup (\mathcal{C}_{n_8} \bowtie \{M_{n_{19}}\}) \\ &= \{\{r4\}, \{a, r_1\}\}\end{aligned}$$

That is, a $\mathcal{T}$*-model for* θ_{n_7} *has the following form:*

$$\theta_{n_7} = (n_7, \{a, r_1, r_4\}, \{\{r_4\}, \{a, r_1\}\}).$$

We are now ready to establish the bottom-up construction for $\mathcal{E}(\cdot)$. This construction depends only on the $\mathcal{T}$-interpretations of the current node and its child nodes.

Lemma 5.2.12. *Let* $\theta = (n, M, \mathcal{C})$ *be a* $\mathcal{T}$*-interpretation, then the following recursive construction is equivalent with Definition 5.2.4.*

$$\mathcal{E}(\theta) = \begin{cases} \{\emptyset\} & (1) \\ \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta') & (2) \\ \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \{I \cup \{a\} \mid I \in \mathcal{E}(\theta')\} & (3) \\ \bigcup_{(\theta_1, \theta_2) \prec_{\mathcal{T}} \theta} \{I_1 \cup I_2 \mid I_1 \in \mathcal{E}(\theta_1), I_2 \in \mathcal{E}(\theta_2)\} & (4) \end{cases}$$

1. n *is an (L) node.*
2. n *is an (RR), (RI), (AR) node, or an (*a*-AI) node and* $a \notin M$.
3. n *is an (*a*-AI) node and* $a \in M$.
4. n *is a (B) node.*

Proof. We show by induction that the above construction is equivalent to $\mathcal{E}(\theta)$.

Base case: Let $\theta = (n, M, \mathcal{C})$ where n is a leaf node, it is easy to see that $\mathcal{E}(\theta) = \{\emptyset\}$ holds for leaf nodes.

Induction step: Let $\theta = (n, M, \mathcal{C})$ where n is a node in $\mathcal{T}$. We assume that the equivalence holds for all $\theta' = (n', M', \mathcal{C}')$ where n' is a node below n in $\mathcal{T}$. We will now look at each possible node type in detail.

CASE 1, n is of type (r-RR). Let the $\mathcal{T}$-interpretations be defined as $\theta = (n, M, \mathcal{C})$ and $\theta' = (n', M', \mathcal{C}')$ where $\theta' \prec_{\mathcal{T}} \theta$ holds.

From the definition of the (r-RR) node, we know that $r \notin M$ and

$$M \cup \{r\} = M' \qquad \mathcal{C} = \{C \setminus \{r\} \mid C \in \mathcal{C}', r \in C\},$$

and thus

$$\begin{aligned} A_{M'} &= A_M & R_{M'} &= R_M \cup \{r\} \\ A_{(n')} &= A_{(n)} & R_{(n')} &= R_{(n)} \\ A_{[n']} &= A_{[n]} & R_{[n']} \cup \{r\} &= R_{[n]} \end{aligned}$$

holds.

Let us now establish some equivalences based on these observations. For the following, let $I = A_M \cup K$ where $K \subseteq A_{[n]}$. Since the rules and atoms in the subtree of n and n' are identical, that is $R_{(n')} = R_{(n)}$ as well as $A_{(n')} = A_{(n)}$, we can conclude

$$\begin{aligned} &\mathsf{SAT}_{n'}(I) \\ &= \{p \mid p \in R_{(n')}, I \models_{A_{(n')}} p\} \\ &= \{p \mid p \in R_{(n)}, I \models_{A_{(n)}} p\} \\ &= \mathsf{SAT}_n(I). \end{aligned} \tag{5.1}$$

The set of satisfied rules for θ' equals the set of satisfied rules for θ, thus $R_{M'} \cup R_{[n']} = R_M \cup R_{[n]}$. The set of atoms $A_{[n']}$ and $A_{[n]}$, as well as $A_{M'}$ and A_M are identical. Together with the already established equivalence (5.1) the following holds.

$$\begin{aligned} &\mathsf{e}_{n'}(M') \\ &= \{A_{M'} \cup K \mid K \subseteq A_{[n']}, \mathsf{SAT}_{n'}(A_{M'} \cup K) = R_{M'} \cup R_{[n']}\} \\ &= \{A_M \cup K \mid K \subseteq A_{[n]}, \mathsf{SAT}_n(A_M \cup K) = R_M \cup R_{[n]}\} \\ &= \mathsf{e}_n(M). \end{aligned} \tag{5.2}$$

Let $J \subseteq A_{(n)}$, then a similar argumentation holds for the sets $\mathsf{RSAT}_{n'}(J, I)$ and $\mathsf{RSAT}_n(J, I)$.

$$\begin{aligned} &\mathsf{RSAT}_{n'}(J, I) \\ &= \{p \mid p \in R_{(n')}, J \models_{A_{(n')}} p \text{ or } \text{B}^-(p) \cap I \neq \emptyset\} \\ &= \{p \mid p \in R_{(n)}, J \models_{A_{(n)}} p \text{ or } \text{B}^-(p) \cap I \neq \emptyset\} \\ &= \mathsf{RSAT}_n(J, I). \end{aligned} \tag{5.3}$$

Let $C \subseteq \beta(n)$ and $C' = C \cup \{r\}$, that is $A_C = A_{C'}$, then $R_{C'} \cup R_{[n']} = R_C \cup R_{[n]}$ holds and thus

$$\begin{aligned} &\mathsf{re}_{n'}(C', I) \\ &= \{A_{C'} \cup K \mid K \subseteq A_{[n']}, \mathsf{RSAT}_{n'}(A_{C'} \cup K, I) = R_{C'} \cup R_{[n']}\} \\ &= \{A_C \cup K \mid K \subseteq A_{[n]}, \mathsf{RSAT}_n(A_C \cup K, I) = R_C \cup R_{[n]}\} \\ &= \mathsf{re}_n(C, I) \end{aligned} \tag{5.4}$$

holds.

We show now that $\mathcal{E}(\theta) = \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$ for a given $\theta = (n, M, \mathcal{C})$.

Let us first recall the definition of $\mathcal{E}(\theta)$.

$$\begin{aligned} \mathcal{E}(\theta) = \{I \mid\ & I \in \mathsf{e}_n(M) \text{ and } \\ & \forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in \mathsf{re}_n(C, I)]\} \end{aligned}$$

Also recall that the condition $I \in \mathsf{e}_n(M)$ is called Condition $\mathcal{A}$ of θ, and $\forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in \mathsf{re}_n(C, I)]$ is called Condition $\mathcal{C}$ of θ with respect to I.

We start by showing $\mathcal{E}(\theta) \supseteq \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$.

Suppose that $I \in \mathcal{E}(\theta')$ for some $\theta' \prec_{\mathcal{T}} \theta$ with $\theta' = (n', M', \mathcal{C}')$.

We show that Condition $\mathcal{C}$ holds for θ wrt. I.

Suppose that $C \in \mathcal{C}$, then there exists a $C' \in \mathcal{C}'$ where $C' = C \cup \{r\}$ by definition of $\prec_{\mathcal{T}}$. Condition $\mathcal{C}$ holds for θ' wrt. I, thus we know that there exists a $J \subset I$ s.t. $J \in \mathsf{re}_{n'}(C', I)$ and from (5.4) we know that $J \in \mathsf{re}_n(C, I)$.

Now, let C be an arbitrary subset of $\beta(n)$ such that $J \in \mathsf{re}_n(C, I)$ where $J \subset I$. Let C' be defined as $C' = C \cup \{r\}$. From (5.4) we know that $J \in \mathsf{re}_{n'}(C', I)$. Since Condition $\mathcal{C}$ holds for θ' wrt. I, we know that $C' \in \mathcal{C}'$, and we can conclude $C \in \mathcal{C}$.

Hence, Condition $\mathcal{C}$ holds for θ wrt. I, and together with (5.2), we know that Condition $\mathcal{A}$ holds for θ. Thus, by definition of $\mathcal{E}(\theta)$, we have the desired $I \in \mathcal{E}(\theta)$.

Next we show $\mathcal{E}(\theta) \subseteq \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$.

Suppose that $I \in \mathcal{E}(\theta)$. We show that there exists a θ' such that $I \in \mathcal{E}(\theta')$ and $\theta' \prec_{\mathcal{T}} \theta$ holds. Let $\theta' = (n', M', \mathcal{C}')$ where $M' = M \cup \{r\}$, and the corresponding set $\mathcal{C}'$ of certificates is defined as follows.

$$\mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \text{ where}$$
$$\mathcal{C}_1 = \{C \cup \{r\} \mid C \in \mathcal{C}\} \text{ and}$$
$$\mathcal{C}_2 = \{C \subseteq \beta(n') \mid r \notin C, \exists J \subset I \text{ s.t. } J \in \mathsf{re}_{n'}(C, I)\}$$

Thus, $\theta' \prec_{\mathcal{T}} \theta$ holds.

We show that Condition $\mathcal{C}$ holds for θ' wrt. I.

Let $C' \in \mathcal{C}_1$, then there exists a $C \in \mathcal{C}$ where $C = C' \setminus \{r\}$. Since Condition $\mathcal{C}$ holds for θ wrt. I, we know that there exists a $J \subset I$ s.t. $J \in \mathsf{re}_n(C, I)$. From (5.4) we can now conclude $J \in \mathsf{re}_{n'}(C', I)$.

Let $C' \in \mathcal{C}_2$, then, per definition of $\mathcal{C}_2$, there exists a $J \subset I$ s.t. $J \in \mathsf{re}_{n'}(C', I)$.

Let C' be an arbitrary subset of $\beta(n')$ such that $J \in \mathsf{re}_{n'}(C', I)$ where $J \subset I$.

First suppose $r \in C'$, then from (5.4) we know that $J \in \mathsf{re}_n(C' \setminus \{r\}, I)$. Since Condition $\mathcal{C}$ holds for θ wrt. I, we can now conclude $C' \in \mathcal{C}_1$.

Now suppose $r \notin C'$, then, per definition of $\mathcal{C}_2$, it holds that $C' \in \mathcal{C}_2$.

Hence, Condition $\mathcal{C}$ holds for θ' wrt. I, and together with (5.2), we know that Condition $\mathcal{A}$ of θ' holds. By the definition of $\mathcal{E}(\theta')$, we have the desired $I \in \mathcal{E}(\theta')$.

Thus, we have shown for a given θ that

- for each θ' with $\theta' \prec_{\mathcal{T}} \theta$, and each $I \in \mathcal{E}(\theta')$ also $I \in \mathcal{E}(\theta)$ holds, and
- for all $I \in \mathcal{E}(\theta)$ there exists a θ' with $\theta' \prec_{\mathcal{T}} \theta$ such that $I \in \mathcal{E}(\theta')$.

Hence $\mathcal{E}(\theta) = \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$ holds.

.

CASE 2, n is of type (r-RI). We distinguish between three cases:

- r is satisfied by atoms in the negative body,
- r is satisfied by atoms in the positive body or the head, or

- r is not satisfied.

Note since we only add a rule, the sets of atoms are not changed in either case, that is for all $\theta = (n, M, \mathcal{C})$ and $\theta' = (n', M', \mathcal{C}')$ with $\theta' \prec_{\mathcal{T}} \theta$ the following equivalences hold.

$$
\begin{aligned}
A_M &= A'_M \\
A_{(n')} &= A_{(n)} \\
A_{[n']} &= A_{[n]}
\end{aligned}
$$

.

CASE 2A. Let $\theta = (n, M, \mathcal{C})$ and $\theta' = (n', M', \mathcal{C}')$ where $\theta' \prec_{\mathcal{T}} \theta$ holds.

Suppose that r is satisfied by A_M, and $B^-(r) \cap A_M \neq \emptyset$ holds, thus $r \cap (A_M \cup \overline{(A_{\beta(n)} \setminus A_M)}) \neq \emptyset$ holds implicitly.

From the definition of the (r-RI) node, we know that

$$
M = M' \cup \{r\} \qquad\qquad \mathcal{C} = \{C \cup \{r\} \mid C \in \mathcal{C}'\},
$$

and thus

$$
\begin{aligned}
R_{M'} \cup \{r\} &= R_M \\
R_{(n')} \cup \{r\} &= R_{(n)} \\
R_{[n']} &= R_{[n]}
\end{aligned}
$$

holds.

Let us now establish some equivalences based on these observations. For the following, let $I = A_M \cup K$ where $K \subseteq A_{[n]}$.

By assumption, we know that $A_M \models_{A_{(n)}} r$ and $A_{M'} \models_{A_{(n')}} r$, and since $R_{(n')} \cup \{r\} = R_{(n)}$, we can conclude

$$
\begin{aligned}
&\mathsf{SAT}_{n'}(I) \cup \{r\} \\
&= \{p \mid p \in R_{(n')}, I \models_{A_{(n')}} p\} \cup \{r\} \\
&= \{p \mid p \in R_{(n')} \cup \{r\}, I \models_{A_{(n')}} p\} \\
&= \{p \mid p \in R_{(n)}, I \models_{A_{(n)}} p\} \\
&= \mathsf{SAT}_n(I).
\end{aligned}
\tag{5.5}
$$

Further, since the rule r is included in the new assignment M, we have $R_{M'} \cup \{r\} = R_M$, and from (5.5) we can conclude the following equivalence.

$$
\begin{aligned}
& \mathsf{e}_{n'}(M') \\
& = \{A_{M'} \cup K \mid K \subseteq A_{[n']}, \mathsf{SAT}_{n'}(A_{M'} \cup K) = R_{M'} \cup R_{[n']}\} \\
& = \{A_{M'} \cup K \mid K \subseteq A_{[n']}, \\
& \quad \mathsf{SAT}_{n'}(A_{M'} \cup K) \cup \{r\} = R_{M'} \cup \{r\} \cup R_{[n']}\} \\
& = \{A_M \cup K \mid K \subseteq A_{[n]}, \mathsf{SAT}_n(A_M \cup K) = R_M \cup R_{[n]}\} \\
& = \mathsf{e}_n(M)
\end{aligned} \tag{5.6}
$$

Let $J \subseteq A_{(n)}$.

Since $B^-(r) \cap A_M \neq \emptyset$ holds by assumption, $B^-(r) \cap I \neq \emptyset$ holds by the connectedness condition and the definition of I. Thus we conclude

$$
\begin{aligned}
& \mathsf{RSAT}_{n'}(J, I) \cup \{r\} \\
& = \{p \mid p \in R_{(n')}, J \models_{A_{(n')}} p \text{ or } B^-(p) \cap I \neq \emptyset\} \cup \{r\} \\
& = \{p \mid p \in R_{(n')} \cup \{r\}, J \models_{A_{(n')}} p \text{ or } B^-(p) \cap I \neq \emptyset\} \\
& = \{p \mid p \in R_{(n)}, J \models_{A_{(n)}} p \text{ or } B^-(p) \cap I \neq \emptyset\} \\
& = \mathsf{RSAT}_n(J, I).
\end{aligned} \tag{5.7}
$$

Let $C' \subseteq \beta(n')$ and $C = C' \cup \{r\}$, then

$$
\begin{aligned}
& \mathsf{re}_{n'}(C', I) \\
& = \{A_{C'} \cup K \mid K \subseteq A_{[n']}, \mathsf{RSAT}_{n'}(A_{C'} \cup K, I) = R_{C'} \cup R_{[n']}\} \\
& = \{A_{C'} \cup K \mid K \subseteq A_{[n']}, \\
& \quad \mathsf{RSAT}_{n'}(A_{C'} \cup K, I) \cup \{r\} = R_{C'} \cup \{r\} \cup R_{[n']}\} \\
& = \{A_C \cup K \mid K \subseteq A_{[n]}, \mathsf{RSAT}_n(A_C \cup K, I) = R_C \cup R_{[n]}\} \\
& = \mathsf{re}_n(C, I)
\end{aligned} \tag{5.8}
$$

holds.

We show now that $\mathcal{E}(\theta) = \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$ for a given $\theta = (n, M, \mathcal{C})$ with $B^-(r) \cap I \neq \emptyset$.

Let us first recall the definition of $\mathcal{E}(\theta)$.

$$
\begin{aligned}
\mathcal{E}(\theta) = \{I \mid & I \in \mathsf{e}_n(M) \text{ and} \\
& \forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in \mathsf{re}_n(C, I)]\}
\end{aligned}
$$

Also recall that the condition $I \in \mathsf{e}_n(M)$ is called Condition $\mathscr{A}$ of θ, and $\forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in \mathsf{re}_n(C, I)]$ is called Condition $\mathscr{C}$ of θ with respect to I.

We start by showing $\mathcal{E}(\theta) \supseteq \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$.

Suppose that $I \in \mathcal{E}(\theta')$ for some $\theta' \prec_{\mathcal{T}} \theta$. Let $\theta' = (n', M', \mathcal{C}')$.

We show that Condition $\mathcal{C}$ holds for θ wrt. I.

Suppose that $C \in \mathcal{C}$. We know from the definition of $\prec_{\mathcal{T}}$ that $r \in C$ holds, thus we know $C \setminus \{r\} \in \mathcal{C}'$. Further, since Condition $\mathcal{C}$ holds for θ' wrt. I, we know that there exists a $J \subset I$ s.t. $J \in \mathtt{re}_{n'}(C \setminus \{r\}, I)$, and from (5.8) we can conclude that $J \in \mathtt{re}_n(C, I)$.

Let C be an arbitrary subset of $\beta(n)$.

First suppose that $r \in C$.

Suppose that $J \in \mathtt{re}_n(C, I)$ where $J \subset I$. From (5.8) we know that $J \in \mathtt{re}_{n'}(C \setminus \{r\}, I)$, and since Condition $\mathcal{C}$ holds for θ' wrt. I, we know that $C \setminus \{r\} \in \mathcal{C}'$. From the definition of $\prec_{\mathcal{T}}$ we can now conclude that $C \in \mathcal{C}$.

Now suppose that $r \notin C$ and towards a contradiction that $C \in \mathcal{C}$. Suppose that $J \in \mathtt{re}_n(C, I)$ where $J \subset I$, thus $\mathsf{RSAT}(J, I) = R_C \cup R_{[n]}$ holds. We know by assumption of $\text{B}^-(r) \cap I \neq \emptyset$ that $r \in \mathsf{RSAT}(J, I)$, but since $r \notin R_C$ by assumption, and $r \notin R_{[n]}$ because of the connectedness condition, we have reached a contradiction. Thus, we conclude if $C \notin \mathcal{C}$ then there exists no $J \subset I$ such that $J \in \mathtt{re}_n(C, I)$.

Hence, Condition $\mathcal{C}$ holds for θ wrt. I, and together with (5.6), we know that Condition $\mathcal{A}$ holds for θ. Thus, by definition of $\mathcal{E}(\theta)$, we have the desired $I \in \mathcal{E}(\theta)$.

Next we show $\mathcal{E}(\theta) \subseteq \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$.

Suppose that $I \in \mathcal{E}(\theta)$. We show that there exists a θ' with $\theta' \prec_{\mathcal{T}} \theta$ such that $I \in \mathcal{E}(\theta')$. Let θ' be defined as $\theta' = (n', M \setminus \{r\}, \mathcal{C}')$ where $\mathcal{C}' = \{C \setminus \{r\} \mid C \in \mathcal{C}\}$.

We show that Condition $\mathcal{C}$ holds for θ' wrt. I.

Suppose that $C' \in \mathcal{C}'$. From the definition of $\prec_{\mathcal{T}}$, we know that then $C' \cup \{r\} \in \mathcal{C}$, and since Condition $\mathcal{C}$ holds for θ wrt. I, we know that there exists a $J \subset I$ s.t. $J \in \mathtt{re}_n(C' \cup \{r\}, I)$. From (5.8) we can now conclude $J \in \mathtt{re}_{n'}(C', I)$.

Let C' be an arbitrary subset of $\beta(n')$.

Suppose that $J \in \mathtt{re}_{n'}(C', I)$ where $J \subset I$, from (5.8) we know that then $J \in \mathtt{re}_n(C' \cup \{r\}, I)$ holds. Since Condition $\mathcal{C}$ holds for θ wrt. I, we know that $C' \cup \{r\} \in \mathcal{C}$, and hence $C' \in \mathcal{C}'$.

Hence, Condition $\mathcal{C}$ holds for θ' wrt. I, and together with (5.6), we know that Condition $\mathcal{A}$ of θ' holds. By the definition of $\mathcal{E}(\theta')$, we have the desired $I \in \mathcal{E}(\theta')$.

Thus, we have shown for a given θ that

- for each θ' with $\theta' \prec_{\mathcal{T}} \theta$, and each $I \in \mathcal{E}(\theta')$ also $I \in \mathcal{E}(\theta)$ holds, and
- for all $I \in \mathcal{E}(\theta)$ there exists a θ' with $\theta' \prec_{\mathcal{T}} \theta$ such that $I \in \mathcal{E}(\theta')$.

Hence $\mathcal{E}(\theta) = \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$ holds.

.

CASE 2B. Let $\theta = (n, M, \mathcal{C})$ and $\theta' = (n', M', \mathcal{C}')$ where $\theta' \prec_{\mathcal{T}} \theta$ holds. Moreover, suppose that $B^-(r) \cap A_M = \emptyset$, but $r \cap (A_M \cup \overline{(A_{\beta(n)} \setminus A_M)}) \neq \emptyset$.

From the definition of the (r-RI) node, we know that

$$M = M' \cup \{r\} \qquad \mathcal{C} = \{C \uplus_n \{r\} \mid C \in \mathcal{C}'\},$$

where

$$C \uplus_n r = \begin{cases} C \cup \{r\} & \text{if } r \cap (A_C \cup \overline{(A_{\beta(n)} \setminus A_C)}) \neq \emptyset \\ C & \text{otherwise} \end{cases}$$

and thus

$$\begin{aligned} R_{M'} \cup \{r\} &= R_M \\ R_{(n')} \cup \{r\} &= R_{(n)} \\ R_{[n']} &= R_{[n]} \end{aligned}$$

holds.

Let us now establish some equivalences based on these observations. For the following, let $I = A_M \cup K$ where $K \subseteq A_{[n]}$.

By assumption, we know that $A_M \models_{A_{(n)}} r$ and $A_{M'} \models_{A_{(n')}} r$, and hence $I \models_{A_{(n')}} r$ and $I \models_{A_{(n)}} r$ by the connectedness condition. Further since $R_{(n')} \cup \{r\} = R_{(n)}$, we can conclude

$$\begin{aligned} &\mathsf{SAT}_{n'}(I) \cup \{r\} \\ &= \{p \mid p \in R_{(n')}, I \models_{A_{(n')}} p\} \cup \{r\} \\ &= \{p \mid p \in R_{(n')} \cup \{r\}, I \models_{A_{(n')}} p\} \\ &= \{p \mid p \in R_{(n)}, I \models_{A_{(n)}} p\} \\ &= \mathsf{SAT}_n(I). \end{aligned} \tag{5.9}$$

Further, since the rule r is included in the new assignment M, we have $R_{M'} \cup \{r\} = R_M$, and from (5.9) we can conclude the following equivalence.

$$\begin{aligned}
&\mathsf{e}_{n'}(M') \\
&= \{A_{M'} \cup K \mid K \subseteq A_{[n']}, \mathsf{SAT}_{n'}(A_{M'} \cup K) = R_{M'} \cup R_{[n']}\} \\
&= \{A_{M'} \cup K \mid K \subseteq A_{[n']}, \\
&\quad \mathsf{SAT}_{n'}(A_{M'} \cup K) \cup \{r\} = R_{M'} \cup \{r\} \cup R_{[n']}\} \\
&= \{A_M \cup K \mid K \subseteq A_{[n]}, \mathsf{SAT}_n(A_M \cup K) = R_M \cup R_{[n]}\} \\
&= \mathsf{e}_n(M)
\end{aligned} \tag{5.10}$$

Let $J \subseteq A_{(n)}$.
Suppose that $J \models_{A_{(n)}} r$, then the following equality holds.

$$\begin{aligned}
&\mathsf{RSAT}_{n'}(J, I) \cup \{r\} \\
&= \{p \mid p \in R_{(n')}, J \models_{A_{(n')}} p \text{ or } B^-(p) \cap I \neq \emptyset\} \cup \{r\} \\
&= \{p \mid p \in R_{(n')} \cup \{r\}, J \models_{A_{(n')}} p \text{ or } B^-(p) \cap I \neq \emptyset\} \\
&= \{p \mid p \in R_{(n)}, J \models_{A_{(n)}} p \text{ or } B^-(p) \cap I \neq \emptyset\} \\
&= \mathsf{RSAT}_n(J, I)
\end{aligned} \tag{5.11}$$

Suppose that $J \not\models_{A_{(n)}} r$. From the connectedness condition of a tree decomposition and the definition of I, we can conclude that if $B^-(r) \cap A_M = \emptyset$ holds, then $B^-(r) \cap I = \emptyset$ holds as well. Thus, $r \notin \mathsf{RSAT}_n(J, I)$ and we obtain

$$\mathsf{RSAT}_{n'}(J, I) = \mathsf{RSAT}_n(J, I). \tag{5.12}$$

Let $C' \subseteq \beta(n')$.
First suppose that $C' \models_{A_{(n')}} r$ holds, and let $C = C' \cup \{r\}$, thus together with (5.11) the following equivalence holds.

$$\begin{aligned}
&\mathsf{re}_{n'}(C', I) \\
&= \{A_{C'} \cup K \mid K \subseteq A_{[n']}, \mathsf{RSAT}_{n'}(A_{C'} \cup K, I) = R_{C'} \cup R_{[n']}\} \\
&= \{A_{C'} \cup K \mid K \subseteq A_{[n']}, \\
&\quad \mathsf{RSAT}_{n'}(A_{C'} \cup K, I) \cup \{r\} = R_{C'} \cup \{r\} \cup R_{[n']}\} \\
&= \{A_C \cup K \mid K \subseteq A_{[n]}, \mathsf{RSAT}_n(A_C \cup K, I) = R_C \cup R_{[n]}\} \\
&= \mathsf{re}_n(C, I)
\end{aligned} \tag{5.13}$$

Now suppose that $C' \not\models_{A_{(n')}} r$ holds, then let $C = C'$, thus together with (5.12) the following equivalence holds.

$$\begin{aligned}
&\mathsf{re}_{n'}(C', I) \\
&= \{A_{C'} \cup K \mid K \subseteq A_{[n']}, \mathsf{RSAT}_{n'}(A_{C'} \cup K, I) = R_{C'} \cup R_{[n']}\} \\
&= \{A_C \cup K \mid K \subseteq A_{[n]}, \mathsf{RSAT}_n(A_C \cup K, I) = R_C \cup R_{[n]}\} \\
&= \mathsf{re}_n(C, I)
\end{aligned} \tag{5.14}$$

We show now that $\mathcal{E}(\theta) = \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$ for a given $\theta = (n, M, \mathcal{C})$.

Let us first recall the definition of $\mathcal{E}(\theta)$.

$$\begin{aligned}
\mathcal{E}(\theta) = \{I \mid\ & I \in \mathsf{e}_n(M) \text{ and} \\
& \forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in \mathsf{re}_n(C, I)]\}
\end{aligned}$$

The condition $I \in \mathsf{e}_n(M)$ is called Condition $\mathcal{A}$ of θ, and $\forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in \mathsf{re}_n(C, I)]$ is called Condition $\mathcal{C}$ of θ with respect to I.

We start by showing $\mathcal{E}(\theta) \supseteq \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$.

Suppose that $I \in \mathcal{E}(\theta')$ for some $\theta' \prec_{\mathcal{T}} \theta$. Let $\theta' = (n', M', \mathcal{C}')$.

We show that Condition $\mathcal{C}$ holds for θ wrt. I.

Let $C \in \mathcal{C}$.

First suppose that $A_C \models_{A_{(n)}} r$. We know from the definition of $\prec_{\mathcal{T}}$ that $r \in C$ and that there exists a $C' \in \mathcal{C}'$ where $C' = C \setminus \{r\}$. Since Condition $\mathcal{C}$ holds for θ' wrt. I, we know that there exists a $J \subset I$ s.t. $J \in \mathsf{re}_{n'}(C', I)$, and from (5.13) we can conclude that $J \in \mathsf{re}_n(C, I)$.

Now suppose that $A_C \not\models_{A_{(n)}} r$. We know from the definition of $\prec_{\mathcal{T}}$ that $r \notin C$ and that $C \in \mathcal{C}'$ holds. Since Condition $\mathcal{C}$ holds for θ' wrt. I, we know that there exists a $J \subset I$ s.t. $J \in \mathsf{re}_{n'}(C, I)$, and from (5.14) we can conclude that $J \in \mathsf{re}_n(C, I)$.

Let C be an arbitrary subset of $\beta(n)$. Suppose that $J \in \mathsf{re}_n(C, I)$ where $J \subset I$.

First suppose that $A_C \models_{A_{(n)}} r$, then from (5.13) we know that $J \in \mathsf{re}_{n'}(C \setminus \{r\}, I)$. Since Condition $\mathcal{C}$ holds for θ' wrt. I, we know that $C \setminus \{r\} \in \mathcal{C}'$. From the definition of $\mathcal{C}$ we can now conclude that $C \in \mathcal{C}$.

Now suppose that $A_C \not\models_{A_{(n)}} r$, then from (5.14) we know that $J \in \mathsf{re}_{n'}(C, I)$ holds, and since Condition $\mathcal{C}$ holds for θ' wrt. I, we know that $C \in \mathcal{C}'$ holds. From the definition of $\mathcal{C}$ we can now conclude that $C \in \mathcal{C}$.

Hence, Condition $\mathcal{C}$ holds for θ wrt. I, and together with (5.10), we know that Condition $\mathcal{A}$ holds for θ. Thus, by definition of $\mathcal{E}(\theta)$, we have the desired $I \in \mathcal{E}(\theta)$.

Next we show $\mathcal{E}(\theta) \subseteq \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$.

Suppose that $I \in \mathcal{E}(\theta)$, we show that there exists a θ' with $\theta' \prec_{\mathcal{T}} \theta$ such that $I \in \mathcal{E}(\theta')$, where $\theta = (n, M, \mathcal{C})$.

Let θ' be defined as $\theta' = (n', M \setminus \{r\}, \mathcal{C}')$ where $\mathcal{C}' = \{C \setminus \{r\} \mid C \in \mathcal{C}\}$. Thus $\theta' \prec_{\mathcal{T}} \theta$ holds.

We show that Condition $\mathcal{C}$ holds for θ' wrt. I.

Let $C' \in \mathcal{C}'$.

First suppose that $A_{C'} \models_{A_{(n')}} r$. From the definition of $\prec_{\mathcal{T}}$, we know that there exists a $C \in \mathcal{C}$ where $C = C' \cup \{r\}$, and since Condition $\mathcal{C}$ holds for θ wrt. I, there exists a $J \subset I$ s.t. $J \in \mathtt{re}_n(C, I)$. From (5.13) we can now conclude $J \in \mathtt{re}_{n'}(C', I)$.

Now suppose that $A_{C'} \not\models_{A_{(n')}} r$. From the definition of $\prec_{\mathcal{T}}$, we know that $C' \in \mathcal{C}$ holds. Since Condition $\mathcal{C}$ holds for θ wrt. I, we know that there exists a $J \subset I$ s.t. $J \in \mathtt{re}_n(C', I)$. From (5.14) we can now conclude $J \in \mathtt{re}_{n'}(C', I)$.

Let C' be an arbitrary subset of $\beta(n')$. Suppose that $J \in \mathtt{re}_{n'}(C', I)$ where $J \subset I$.

First suppose that $A_{C'} \models_{A_{(n')}} r$. From (5.13) we know that $J \in \mathtt{re}_n(C, I)$ holds for $C = C' \cup \{r\}$. Since Condition $\mathcal{C}$ holds for θ wrt. I, we know that $C \in \mathcal{C}$, and hence, from the definition of $\prec_{\mathcal{T}}$, that $C' \in \mathcal{C}'$ holds.

Now suppose that $A_{C'} \not\models_{A_{(n')}} r$. From (5.14) we know that $J \in \mathtt{re}_n(C', I)$ holds. Since Condition $\mathcal{C}$ holds for θ wrt. I, we know that $C' \in \mathcal{C}$, and hence $C' \in \mathcal{C}'$.

Hence, Condition $\mathcal{C}$ holds for θ' wrt. I, and together with (5.10), we know that Condition $\mathcal{A}$ for θ' holds. By the definition of $\mathcal{E}(\theta')$, we have the desired $I \in \mathcal{E}(\theta')$.

Thus, we have shown for a given θ that

- for each θ' with $\theta' \prec_{\mathcal{T}} \theta$, and each $I \in \mathcal{E}(\theta')$ also $I \in \mathcal{E}(\theta)$ holds, and

- for all $I \in \mathcal{E}(\theta)$ there exists a θ' with $\theta' \prec_{\mathcal{T}} \theta$ such that $I \in \mathcal{E}(\theta')$.

Hence $\mathcal{E}(\theta) = \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$ holds.

.

CASE 2C. Let $\theta = (n, M, \mathcal{C})$ and $\theta' = (n', M', \mathcal{C}')$ where $\theta' \prec_{\mathcal{T}} \theta$ holds.

Suppose that $r \cap (A_M \cup \overline{(A_{\beta(n)} \setminus A_M)}) = \emptyset$, thus $\text{B}^-(r) \cap A_M = \emptyset$ holds implicitly.

From the definition of the (r-RI) node, we know that

$$M = M' \qquad\qquad \mathcal{C} = \{C \uplus_n \{r\} \mid C \in \mathcal{C}'\},$$

where

$$C \uplus_n r = \begin{cases} C \cup \{r\} & \text{if } r \cap (A_C \cup \overline{(A_{\beta(n)} \setminus A_C)}) \neq \emptyset \\ C & \text{otherwise} \end{cases}$$

and thus

$$\begin{aligned} R_{M'} &= R_M \\ R_{[n']} &= R_{[n]} \\ R_{(n')} \cup \{r\} &= R_{(n)} \end{aligned}$$

holds.

Let us now establish some equivalences based on these observations. For the following, let $I = A_M \cup K$ where $K \subseteq A_{[n]}$.

By assumption, we know that $A_M \not\models_{A_{(n)}} r$ and $A_{M'} \not\models_{A_{(n')}} r$, and hence $I \not\models_{A_{(n')}} r$ and $I \not\models_{A_{(n)}} r$ by the connectedness condition. Further since $R_{(n')} \cup \{r\} = R_{(n)}$, we can conclude

$$\begin{aligned} &\mathsf{SAT}_{n'}(I) \\ &= \{p \mid p \in R_{(n')}, I \models_{A_{(n')}} p\} \\ &= \{p \mid p \in R_{(n')} \cup \{r\}, I \models_{A_{(n')}} p\} \\ &= \{p \mid p \in R_{(n)}, I \models_{A_{(n)}} p\} \\ &= \mathsf{SAT}_n(I). \end{aligned} \tag{5.15}$$

Further, since the rule r is not included in M, we have $R_{M'} = R_M$, and from (5.15) we can conclude the following equivalence.

$$\begin{aligned} &\mathsf{e}_{n'}(M') \\ &= \{A_{M'} \cup K \mid K \subseteq A_{[n']}, \mathsf{SAT}_{n'}(A_{M'} \cup K) = R_{M'} \cup R_{[n']}\} \\ &= \{A_M \cup K \mid K \subseteq A_{[n]}, \mathsf{SAT}_n(A_M \cup K) = R_M \cup R_{[n]}\} \\ &= \mathsf{e}_n(M) \end{aligned} \tag{5.16}$$

We show now that $I \in \mathcal{E}(\theta) \iff I \in \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$ for a given $\theta = (n, M, \mathcal{C})$.

The proof that Condition $\mathcal{C}$ holds for θ wrt. I, resp. that Condition $\mathcal{C}$ holds for all θ' with $\theta' \prec_{\mathcal{T}} \theta$ wrt. I is analogous to Case 2B. From (5.16) we can conclude Condition $\mathcal{A}$ for θ resp. for all θ' with $\theta' \prec_{\mathcal{T}} \theta$.

Hence $\mathcal{E}(\theta) = \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$ holds.

.

CASE 3, n is of type (a-AR). Let $\theta = (n, M, \mathcal{C})$ and $\theta' = (n', M', \mathcal{C}')$ where $\theta' \prec_{\mathcal{T}} \theta$. From the definition of the (a-AR) node, we know that

$$M = M' \setminus \{a\} \qquad \mathcal{C} = \{C \setminus \{a\} \mid C \in \mathcal{C}'\},$$

thus

$$\begin{aligned} A_{M'} \setminus \{a\} &= A_M & R_{M'} &= R_M \\ A_{(n')} &= A_{(n)} & R_{(n')} &= R_{(n)} \\ A_{[n']} \cup \{a\} &= A_{[n]} & R_{[n']} &= R_{[n]}. \end{aligned}$$

Let us now establish some equivalences based on these observations. We first show for a $M \subseteq \beta(n)$ that

$$e_n(M) = e_{n'}(M \cup \{a\}) \cup e_{n'}(M) \tag{5.17}$$

holds.

Let us first recall the definition of $e_n(M)$.

$$e_n(M) = \{A_M \cup K \mid K \subseteq A_{[n]}, \mathsf{SAT}_n(A_M \cup K) = R_M \cup R_{[n]}\}$$

For the following, let $I \subseteq A_{(n)}$. Since $A_{(n')} = A_{(n)}$ and $R_{(n')} = R_{(n)}$, the following equivalence is immediate.

$$\begin{aligned} &\mathsf{SAT}_{n'}(I) \\ &= \{p \mid p \in R_{(n')}, I \models_{A_{(n')}} p\} \\ &= \{p \mid p \in R_{(n)}, I \models_{A_{(n)}} p\} \\ &= \mathsf{SAT}_n(I) \end{aligned} \tag{5.18}$$

We show $e_n(M) \subseteq e_{n'}(M \cup \{a\}) \cup e_{n'}(M)$. Let $I \in e_n(M)$.

First suppose $a \in I$. By definition of $e_n(M)$, we know that there exists a $K \subseteq A_{[n]}$ such that $I = A_M \cup K$ and $\mathsf{SAT}_n(I) = R_M \cup R_{[n]}$. Since $a \notin A_M$ by definition, $a \in K$ holds, and since $A_{[n']} = A_{[n]} \setminus \{a\}$ holds, $K' \subseteq A_{[n']}$ where $K' = K \setminus \{a\}$. Hence that

$I = A_M \cup \{a\} \cup K'$. Together with (5.18) and since $R_{[n']} = R_{[n]}$ and $R_M = R_{M \cup \{a\}}$, we can conclude $\mathsf{SAT}_{n'}(A_M \cup \{a\} \cup K') = R_{M \cup \{a\}} \cup R_{[n']}$, and thus that $I \in \mathsf{e}_{n'}(M \cup \{a\})$.

Now suppose $a \notin I$. By definition of $\mathsf{e}_n(M)$, we know that there exists a $K \subseteq A_{[n]}$ such that $I = A_M \cup K$ and $\mathsf{SAT}_n(I) = R_M \cup R_{[n]}$. Further we know that $K \subseteq A_{[n']}$ since $A_{[n]} \setminus \{a\} = A_{[n']}$. Together with (5.18) and since $R_{[n']} = R_{[n]}$, we can conclude that $\mathsf{SAT}_{n'}(A_M \cup K) = R_M \cup R_{[n']}$, and thus that $I \in \mathsf{e}_{n'}(M)$.

Next we show $\mathsf{e}_n(M) \supseteq \mathsf{e}_{n'}(M \cup \{a\}) \cup \mathsf{e}_{n'}(M)$.

Let $I \in \mathsf{e}_{n'}(M)$. From the definition of $\mathsf{e}_{n'}(M)$ we know that $I = A_M \cup K$ where $K \subseteq A_{[n']}$ and $\mathsf{SAT}_{n'}(I) = R_M \cup R_{[n']}$ holds. Since $A_{[n]} = A_{[n']} \cup \{a\}$, we know that $K \subseteq A_{[n]}$. Together with (5.18) and since $R_{[n']} = R_{[n]}$ we can conclude that $\mathsf{SAT}_n(A_M \cup K) = R_M \cup R_{[n]}$, and thus that $I \in \mathsf{e}_n(M)$.

Let $I \in \mathsf{e}_{n'}(M \cup \{a\})$. From the definition of $\mathsf{e}_{n'}(M \cup \{a\})$ we know that $I = A_M \cup \{a\} \cup K$ where $K \subseteq A_{[n']}$ and $\mathsf{SAT}_{n'}(I) = R_M \cup R_{[n']}$ holds. Since $A_{[n]} = A_{[n']} \cup \{a\}$ holds, we know that there exists a K' such that $K' \subseteq A_{[n]}$ where $K' = K \cup \{a\}$ holds. Together with (5.18) and since $R_{[n']} = R_{[n]}$ we can conclude that $\mathsf{SAT}_n(A_M \cup K') = R_M \cup R_{[n]}$, and thus that $I \in \mathsf{e}_n(M)$.

Thus, we have shown that (5.17) holds.

We will now show the following equivalence. Let $C \subseteq \beta(n)$ and $I \subseteq A_{(n)}$, then

$$\mathsf{re}_n(C, I) = \mathsf{re}_{n'}(C \cup \{a\}, I) \cup \mathsf{re}_{n'}(C, I) \tag{5.19}$$

holds.

Let us first recall the definition of $\mathsf{re}_n(M)$.

$$\begin{aligned}\mathsf{re}_n(C, I) = \{&A_C \cup K \mid K \subseteq A_{[n]},\\ &\mathsf{RSAT}_n(A_C \cup K, I) = R_C \cup R_{[n]}\}\end{aligned}$$

Let J an arbitrary subset of $A_{(n)}$, then the following equivalence is immediate, since $A_{(n')} = A_{(n)}$ and $R_{(n')} = R_{(n)}$.

$$\begin{aligned}&\mathsf{RSAT}_{n'}(J, I)\\ &= \{p \mid p \in R_{(n')}, J \models_{A_{(n')}} p \text{ or } B^-(p) \cap I \neq \emptyset\}\\ &= \{p \mid p \in R_{(n)}, J \models_{A_{(n)}} p \text{ or } B^-(p) \cap I \neq \emptyset\}\\ &= \mathsf{RSAT}_n(J, I)\end{aligned} \tag{5.20}$$

We first show that $\mathsf{re}_n(C, I) \subseteq \mathsf{re}_{n'}(C \cup \{a\}, I) \cup \mathsf{re}_{n'}(C, I)$ holds.

Let $J \in \mathsf{re}_n(C, I)$.

First suppose $a \in J$. By definition of $\mathsf{re}_n(C, I)$, we know that there exists a $K \subseteq A_{[n]}$ such that $J = A_C \cup K$ and $\mathsf{RSAT}_n(A_C \cup K, I) = R_C \cup R_{[n]}$. Since $a \notin A_C$ by definition, $a \in K$ holds. Since $A_{[n']} = A_{[n]} \setminus \{a\}$ we know that there exists a $K' \subseteq A_{[n']}$ where $K' = K \setminus \{a\}$ and hence that $J = A_C \cup \{a\} \cup K'$. Together with (5.20) and since $R_C \cup R_{[n]} = R_{C \cup \{a\}} \cup R_{[n']}$, we can conclude that $\mathsf{RSAT}_{n'}(A_C \cup \{a\} \cup K', I) = R_{C \cup \{a\}} \cup R_{[n']}$, and thus that $J \in \mathsf{re}_{n'}(C \cup \{a\}, I)$.

Now suppose $a \notin J$. By definition of $\mathsf{re}_n(C, I)$, we know that there exists a $K \subseteq A_{[n]}$ such that $J = A_C \cup K$ and $\mathsf{RSAT}_n(A_C \cup K, I) = R_C \cup R_{[n]}$. Further we know that $K \subseteq A_{[n']}$ since $A_{[n]} \setminus \{a\} = A_{[n']}$ and $a \notin K$. Together with (5.20) and since $R_C \cup R_{[n]} = R_C \cup R_{[n']}$, we can conclude that $K \subseteq A_{[n']}$ such that $\mathsf{RSAT}_{n'}(A_C \cup K, I) = R_C \cup R_{[n']}$, and thus that $J \in \mathsf{re}_{n'}(C, I)$.

Now we show that $\mathsf{re}_n(C, I) \supseteq \mathsf{re}_{n'}(C \cup \{a\}, I) \cup \mathsf{re}_{n'}(C, I)$ holds.

Let $J \in \mathsf{re}_{n'}(C, I)$. Since $J = A_C \cup K$ where $K \subseteq A_{[n']}$ we know that $\mathsf{RSAT}_{n'}(A_C \cup K, I) = R_C \cup R_{[n']}$. Since $A_{[n]} = A_{[n']} \cup \{a\}$ holds, we know that $K \subseteq A_{[n]}$. Together with (5.20) and since $R_C \cup R_{[n]} = R_C \cup R_{[n']}$ we can conclude that $\mathsf{RSAT}_n(A_C \cup K, I) = R_C \cup R_{[n]}$, and thus that $J \in \mathsf{re}_n(C, I)$.

Let $J \in \mathsf{re}_{n'}(C \cup \{a\}, I)$. Since $J = A_C \cup \{a\} \cup K$ where $K \subseteq A_{[n']}$ we know that $\mathsf{RSAT}_{n'}(A_C \cup \{a\} \cup K, I) = R_{C \cup \{a\}} \cup R_{[n']}$ holds. Since $A_{[n]} = A_{[n']} \cup \{a\}$ holds, we know that there exists a K' such that $K' \subseteq A_{[n]}$ where $K' = K \cup \{a\}$. Together with (5.20) and since $R_C \cup R_{[n]} = R_C \cup R_{[n']}$ we can conclude that $\mathsf{RSAT}_n(A_C \cup K', I) = R_C \cup R_{[n]}$, and thus that $J \in \mathsf{re}_n(C, I)$.

Thus, we have shown that (5.19) holds.

We show now that $\mathcal{E}(\theta) = \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$ for a given $\theta = (n, M, \mathcal{C})$.

Let us first recall the definition of $\mathcal{E}(\theta)$.

$$\begin{aligned}\mathcal{E}(\theta) = \{I \mid\ & I \in \mathsf{e}_n(M) \text{ and} \\ & \forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in \mathsf{re}_n(C, I)]\}\end{aligned}$$

Also recall that the condition $I \in \mathsf{e}_n(M)$ is called Condition $\mathcal{A}$ of θ, and $\forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in \mathsf{re}_n(C, I)]$ is called Condition $\mathcal{C}$ of θ with respect to I.

We start by showing $\mathcal{E}(\theta) \supseteq \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$. Let $I \in \mathcal{E}(\theta')$ for some $\theta' \prec_{\mathcal{T}} \theta$ with $\theta' = (n', M', \mathcal{C}')$.

We show that Condition $\mathcal{C}$ holds for θ wrt. I.

Suppose that $C \in \mathcal{C}$. From the definition of $\prec_{\mathcal{T}}$, we know that there exists a $C' \in \mathcal{C}'$ where $C = C' \setminus \{a\}$ holds. Since Condition $\mathcal{C}$ holds for θ' wrt. I, we know that there exists a $J \subset I$ such that $J \in \mathsf{re}_{n'}(C', I)$. From (5.19) we can conclude that then $J \in \mathsf{re}_{n}(C, I)$.

Suppose that $J \in \mathsf{re}_{n}(C, I)$ where $J \subset I$. From (5.19) we know that $J \in \mathsf{re}_{n'}(C', I)$ where $C = C' \setminus \{a\}$ holds. Since Condition $\mathcal{C}$ holds for θ' wrt. I, we know that $C' \in \mathcal{C}'$. Now we can conclude, from the definition of $\prec_{\mathcal{T}}$, that $C \in \mathcal{C}$.

Hence, Condition $\mathcal{C}$ holds for θ wrt. I, and together with (5.17), we know that Condition $\mathcal{A}$ holds for θ. Thus, by definition of $\mathcal{E}(\theta)$, we have the desired $I \in \mathcal{E}(\theta)$.

Next we show $\mathcal{E}(\theta) \subseteq \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$. Let $I \in \mathcal{E}(\theta)$, we show that there exists a θ' with $\theta' \prec_{\mathcal{T}} \theta$ such that $I \in \mathcal{E}(\theta')$.

Suppose that $a \in I$. Let $\theta' = (n', M', \mathcal{C}')$ where $M' = M \cup \{a\}$ and

$$\mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \text{ with}$$
$$\mathcal{C}_1 = \{C \cup \{a\} \mid C \in \mathcal{C}, \exists J \subset I \text{ s.t. } J \in \mathsf{re}_{n'}(C \cup \{a\}, I)\}$$
$$\mathcal{C}_2 = \{C \mid C \in \mathcal{C}, \exists J \subset I \text{ s.t. } J \in \mathsf{re}_{n'}(C, I)\}.$$

Thus $\theta' \prec_{\mathcal{T}} \theta$ holds.

We show that Condition $\mathcal{C}$ holds for θ' wrt. I.

Suppose that $C' \in \mathcal{C}'$, then by definition of $\mathcal{C}'$ it holds that $J \in \mathsf{re}_{n'}(C', I)$.

Now let C' be an arbitrary subset of $\beta(n')$ and suppose that $J \in \mathsf{re}_{n'}(C', I)$ where $J \subset I$. From (5.19) we know that $J \in \mathsf{re}_{n}(C, I)$ where $C = C' \setminus \{a\}$. Since Condition $\mathcal{C}$ holds for θ wrt. I, we know that $C \in \mathcal{C}$, and thus we know, from the definition of $\prec_{\mathcal{T}}$, that

- $C' \in \mathcal{C}_1$ if $C' = C \cup \{a\}$, and
- $C' \in \mathcal{C}_2$ if $C' = C$.

Hence, Condition $\mathcal{C}$ holds for θ' wrt. I, and together with (5.17), we know that Condition $\mathcal{A}$ for θ' holds. By the definition of $\mathcal{E}(\theta')$, we have the desired $I \in \mathcal{E}(\theta')$.

Suppose that $a \notin I$, and let $\theta' = (n', M, \mathcal{C})$, thus $\theta' \prec_{\mathcal{T}} \theta$ holds.

Condition $\mathcal{C}$ holds for θ wrt. I, and for each $C \in \mathcal{C}$ there exists a $J \subset I$ such that $J \in \mathsf{re}_{n}(C, I)$. From (5.19) we thus can conclude that $J \in \mathsf{re}_{n'}(C, I)$.

Now let C be an arbitrary subset of $\beta(n')$ and suppose that $J \in \mathsf{re}_{n'}(C, I)$ where $J \subset I$. From (5.19) we know that $J \in \mathsf{re}_{n}(C, I)$. Since Condition $\mathcal{C}$ holds for θ wrt. I, we know that $C \in \mathcal{C}$.

Hence, Condition $\mathcal{C}$ holds for θ' wrt. I, and together with (5.17), we know that Condition $\mathcal{A}$ for θ' holds. By the definition of $\mathcal{E}(\theta')$, we have the desired $I \in \mathcal{E}(\theta')$.

Thus, we have shown for a given θ that

- for each θ' with $\theta' \prec_{\mathcal{T}} \theta$, and each $I \in \mathcal{E}(\theta')$ also $I \in \mathcal{E}(\theta)$ holds, and
- for all $I \in \mathcal{E}(\theta)$ there exists a θ' with $\theta' \prec_{\mathcal{T}} \theta$ such that $I \in \mathcal{E}(\theta')$.

Hence $\mathcal{E}(\theta) = \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$ holds.

.

CASE 4, n is of type (a-AI). Recall that in the case of an atom introduction, we have two ways how a θ' and a θ can relate to each other. One is by adding a negatively to the assignment of θ (covered in Case 4A), and one by adding a positively to the assignment of θ (covered in Case 4B).

CASE 4A. Let $\theta = (n, M, \mathcal{C})$ and $\theta' = (n', M', \mathcal{C}')$ where $\theta' \prec_{\mathcal{T}} \theta$. From the definition of the (a-AI) node (part 1, the atom a is set to false), we know that

$$M = M' \cup R^{+}_{a,n} \qquad \mathcal{C} = \{C \cup R^{+}_{a,n} \mid C \in \mathcal{C}'\}.$$

Recall that

$$R^{+}_{a,n} = \{r \in R_{\beta(n)} \mid \overline{a} \in r\}.$$

That is, we add all rules to the assignment that are true if the atom a is set to false. Since

$$\begin{aligned} A_{M'} &= A_M & R_{M'} \cup R^{+}_{a,n} &= R_M \\ A_{(n')} \cup \{a\} &= A_{(n)} & R_{(n')} &= R_{(n)} \\ A_{[n']} &= A_{[n]} & R_{[n']} &= R_{[n]}, \end{aligned}$$

and by assumption, we know that $A_M \models_{A(n)} R^{+}_{a,n}$, thus $I \models_{A(n)} R^{+}_{a,n}$ for all $I = A_M \cup K$ where $K \subseteq A_{[n]}$.

Let us now establish some equivalences based on these observations. For the following, let $I = A_M \cup K$ where $K \subseteq A_{[n]}$.

$$\begin{aligned} & \mathsf{SAT}_{n'}(I) \cup R^{+}_{\overline{a},n} \\ & = \{p \mid p \in R_{(n')}, I \models_{A_{(n')}} p\} \cup R^{+}_{\overline{a},n} \\ & = \{p \mid p \in R_{(n)}, I \models_{A_{(n')} \cup \{a\}} p\} \\ & = \{p \mid p \in R_{(n)}, I \models_{A_{(n)}} p\} \\ & = \mathsf{SAT}_{n}(I) \end{aligned} \tag{5.21}$$

Let $M' \subseteq \beta(n')$ and $M = M' \cup R^{+}_{\overline{a},n}$. Since we consider the atom a, but A_M does not include the atom a, we add the set of rules satisfied by $\overline{a}$ to R_M. Thus we can conclude that the following equivalence holds.

$$\begin{aligned} & \mathsf{e}_{n'}(M') \\ & = \{A_{M'} \cup K \mid K \subseteq A_{[n']}, \mathsf{SAT}_{n'}(A_{M'} \cup K) = R_{M'} \cup R_{[n']}\} \\ & = \{A_{M'} \cup K \mid K \subseteq A_{[n']}, \\ & \quad \mathsf{SAT}_{n'}(A_{M'} \cup K) = R_{M'} \cup R_{[n']}\} \\ & = \{A_{M} \cup K \mid K \subseteq A_{[n]}, \\ & \quad \mathsf{SAT}_{n'}(A_{M} \cup K) \cup R^{+}_{\overline{a},n} = (R_{M'} \cup R^{+}_{\overline{a},n}) \cup R_{[n']}\} \\ & = \{A_{M} \cup K \mid K \subseteq A_{[n]}, \mathsf{SAT}_{n}(A_{M} \cup K) = R_{M} \cup R_{[n]}\} \\ & = \mathsf{e}_{n}(M) \end{aligned} \tag{5.22}$$

Let $J \subseteq A_{(n')}$, then we have

$$\begin{aligned} & \mathsf{RSAT}_{n'}(J, I) \cup R^{+}_{\overline{a},n} \\ & = \{p \mid p \in R_{(n')}, J \models_{A_{(n')}} p \text{ or } B^{-}(p) \cap I \neq \emptyset\} \cup R^{+}_{\overline{a},n} \\ & = \{p \mid p \in R_{(n)}, J \models_{A_{(n)}} p \text{ or } B^{-}(p) \cap I \neq \emptyset\} \\ & = \mathsf{RSAT}_{n}(J, I). \end{aligned} \tag{5.23}$$

Let $C' \subseteq \beta(n')$ and $C = C' \cup R^{+}_{\overline{a},n}$, then

$$\begin{aligned} & \mathsf{re}_{n'}(C', I) \\ & = \{A_{C'} \cup K \mid K \subseteq A_{[n']}, \mathsf{RSAT}_{n'}(A_{C'} \cup K, I) = R_{C'} \cup R_{[n']}\} \\ & = \{A_{C'} \cup K \mid K \subseteq A_{[n']}, \\ & \quad \mathsf{RSAT}_{n'}(A_{C'} \cup K, I) \cup R^{+}_{\overline{a},n} = (R_{C'} \cup R^{+}_{\overline{a},n}) \cup R_{[n']}\} \\ & = \{A_{C} \cup K \mid K \subseteq A_{[n]}, \mathsf{RSAT}_{n}(A_{C} \cup K, I) = R_{C} \cup R_{[n]}\} \\ & = \mathsf{re}_{n}(C, I) \end{aligned} \tag{5.24}$$

holds.

We show now that $\mathcal{E}(\theta) = \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$ for a given $\theta = (n, M, \mathcal{C})$.

Let us first recall the definition of $\mathcal{E}(\theta)$.

$$\begin{aligned}\mathcal{E}(\theta) = \{I \mid\ & I \in \mathsf{e}_n(M) \text{ and} \\ & \forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in \mathsf{re}_n(C, I)]\}\end{aligned}$$

Also recall that the condition $I \in \mathsf{e}_n(M)$ is called Condition $\mathcal{A}$ of θ, and $\forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in \mathsf{re}_n(C, I)]$ is called Condition $\mathcal{C}$ of θ with respect to I.

We start by showing $\mathcal{E}(\theta) \supseteq \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$. Let $I \in \mathcal{E}(\theta')$ for some $\theta' \prec_{\mathcal{T}} \theta$ with $\theta' = (n', M', \mathcal{C}')$.

We show that Condition $\mathcal{C}$ holds for θ wrt. I.

Let $C \in \mathcal{C}$. From the definition of $\prec_{\mathcal{T}}$, we know that there exists a $C' \in \mathcal{C}'$ where $C = C' \cup R^{+}_{a,n}$. Since Condition $\mathcal{C}$ holds for θ' wrt. I, we know that there exists a $J \subset I$ s.t. $J \in \mathsf{re}_{n'}(C', I)$. Now we can conclude from (5.24) that $J \in \mathsf{re}_n(C, I)$.

Let C be an arbitrary subset of $\beta(n)$, and suppose that $J \in \mathsf{re}_n(C, I)$ where $J \subset I$. From (5.24) we know that there exists a C' where $C' \cup R^{+}_{a,n} = C$ such that $J \in \mathsf{re}_{n'}(C', I)$ holds. Since Condition $\mathcal{C}$ holds for θ' wrt. I, we know that $C' \in \mathcal{C}'$. From the definition of $\prec_{\mathcal{T}}$ we can now conclude $C \in \mathcal{C}$.

Thus, Condition $\mathcal{C}$ holds for θ wrt. I and together with (5.22), that is Condition $\mathcal{A}$ holds for θ, we can conclude $I \in \mathcal{E}(\theta)$.

Next we show $\mathcal{E}(\theta) \subseteq \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$. Let $I \in \mathcal{E}(\theta)$, we show that there exists a θ' with $\theta' \prec_{\mathcal{T}} \theta$ such that $I \in \mathcal{E}(\theta')$.

Let θ' be defined as $\theta' = (n', M', \mathcal{C}')$ where the following holds.

$$\begin{aligned}M' &= A_M \cup (\mathsf{SAT}_{n'}(I) \cap R_M) \\ \mathcal{C}' &= \{A_C \cup H \mid C \in \mathcal{C}, \\ &\qquad H \subseteq R^{+}_{a,n'}, \exists J \subset I \text{ s.t. } J \in \mathsf{re}_{n'}(C \setminus H, I)\}.\end{aligned}$$

Thus, $\theta' \prec_{\mathcal{T}} \theta$ holds.

We show that Condition $\mathcal{C}$ holds for θ' wrt. I.

Let $C' \in \mathcal{C}'$, then we know that there exists a $C \in \mathcal{C}$ where $C = C' \cup R^{+}_{a,n'}$, and since Condition $\mathcal{C}$ holds for θ wrt. I, we know that there exists a $J \subset I$ s.t. $J \in \mathsf{re}_n(C, I)$. From (5.24) we can now conclude $J \in \mathsf{re}_{n'}(C', I)$.

Let C' be an arbitrary subset of $\beta(n')$, and suppose that $J \in \mathsf{re}_{n'}(C', I)$ where $J \subset I$, from (5.24) we know that then

$J \in \mathsf{re}_n(C, I)$ where $C = C' \cup R^+_{\overline{a},n}$. Since Condition $\mathcal{C}$ holds for θ wrt. I, we also know that $C \in \mathcal{C}$, and hence $C' \in \mathcal{C}'$.

Thus, Condition $\mathcal{C}$ holds for θ' wrt. I and together with (5.22), that is Condition $\mathcal{A}$ holds for θ', we can conclude $I \in \mathcal{E}(\theta')$.

Thus, we have shown for a given θ that

- for each θ' with $\theta' \prec_{\mathcal{T}} \theta$, and each $I \in \mathcal{E}(\theta')$ also $I \in \mathcal{E}(\theta)$ holds, and
- for all $I \in \mathcal{E}(\theta)$ there exists a θ' with $\theta' \prec_{\mathcal{T}} \theta$ such that $I \in \mathcal{E}(\theta')$.

Hence $\mathcal{E}(\theta) = \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')$ holds.

.

CASE 4B. Let $\theta = (n, M, \mathcal{C})$ and $\theta' = (n', M', \mathcal{C}')$ where $\theta' \prec_{\mathcal{T}} \theta$. From the definition of the (a-AI) node (part 2, the atom a is set to true), we know that

$$M = M' \cup \{a\} \cup R^+_{a,n} \qquad \begin{aligned} \mathcal{C} = \{&(M' \cup R^+_{\overline{a},n}) \cup R^{B^-}_{a,n}\} \cup \\ \{&(C \cup \{a\} \cup R^+_{a,n}) \cup R^{B^-}_{a,n}, \\ &(C \cup R^+_{\overline{a},n}) \cup R^{B^-}_{a,n} \mid C \in \mathcal{C}'\}. \end{aligned}$$

Recall that

$$\begin{aligned} R^{B^-}_{a,n} &= \{r \in R_{\beta(n)} \mid a \in B^-(r)\} \\ R^+_{a,n} &= \{r \in R_{\beta(n)} \mid a \in r\}, \\ R^+_{\overline{a},n} &= \{r \in R_{\beta(n)} \mid \overline{a} \in r\}. \end{aligned}$$

thus

$$\begin{aligned} A_{M'} \cup \{a\} &= A_M & R_{M'} \cup R^+_{a,n} &= R_M \\ A_{(n')} \cup \{a\} &= A_{(n)} & R_{(n')} &= R_{(n)} \\ A_{[n']} &= A_{[n]} & R_{[n']} &= R_{[n]} \end{aligned}$$

By assumption, we know that $A_M \models_{A(n)} R^+_{a,n}$.

Let us now establish some equivalences based on these observations. For the following, let $I = A_{M'} \cup K$ where $K \subseteq A_{[n']}$, then

$$\begin{aligned} &\mathsf{SAT}_{n'}(I) \cup R^+_{a,n} \\ &= \{p \mid p \in R_{(n')}, I \models_{A_{(n')}} p\} \cup R^+_{a,n} \\ &= \{p \mid p \in R_{(n)}, I \cup \{a\} \models_{A_{(n)}} p\} \\ &= \mathsf{SAT}_n(I \cup \{a\}) \end{aligned} \tag{5.25}$$

holds.

Let $M' \subseteq \beta(n')$ and $M = M' \cup \{a\} \cup R^{+}_{a,n}$, then

$$
\begin{aligned}
& I \in \mathsf{e}_{n'}(M') \\
& \iff I \in \{A_{M'} \cup K \mid K \subseteq A_{[n']}, \mathsf{SAT}_{n'}(A_{M'} \cup K) = R_{M'} \cup R_{[n']}\} \\
& \iff I \cup \{a\} \in \{A_{M'} \cup \{a\} \cup K \mid K \subseteq A_{[n]}, \\
& \qquad \mathsf{SAT}_{n'}(A_{M'} \cup K) \cup R^{+}_{a,n} = (R_{M'} \cup R^{+}_{a,n}) \cup R_{[n']}\} \\
& \iff I \cup \{a\} \in \{A_{M} \cup K \mid K \subseteq A_{[n]}, \mathsf{SAT}_{n}(A_{M} \cup K) = R_{M} \cup R_{[n]}\} \\
& \iff I \cup \{a\} \in \mathsf{e}_{n}(M)
\end{aligned}
\tag{5.26}
$$

holds.

Let $J \subseteq A_{(n)}$.

Suppose $a \in J$, then

$$
\begin{aligned}
& \mathsf{RSAT}_{n'}(J \setminus \{a\}, I) \cup R^{+}_{a,n} \cup R^{B^-}_{a,n} \\
& = \{p \mid p \in R_{(n')}, J \setminus \{a\} \models_{A_{(n')}} p \text{ or } \textsc{b}^-(p) \cap I \neq \emptyset\} \cup R^{+}_{a,n} \cup R^{B^-}_{a,n} \\
& = \{p \mid p \in R_{(n)}, J \models_{A_{(n)}} p \text{ or } \textsc{b}^-(p) \cap (I \cup \{a\}) \neq \emptyset\} \\
& = \mathsf{RSAT}_{n}(J, I \cup \{a\})
\end{aligned}
\tag{5.27}
$$

holds.

Suppose $a \notin J$, then

$$
\begin{aligned}
& \mathsf{RSAT}_{n'}(J, I) \cup R^{+}_{\bar{a},n} \cup R^{B^-}_{a,n} \\
& = \{p \mid p \in R_{(n')}, J \models_{A_{(n')}} p \text{ or } \textsc{b}^-(p) \cap I \neq \emptyset\} \cup R^{+}_{\bar{a},n} \cup R^{B^-}_{a,n} \\
& = \{p \mid p \in R_{(n)}, J \models_{A_{(n)}} p \text{ or } \textsc{b}^-(p) \cap (I \cup \{a\}) \neq \emptyset\} \\
& = \mathsf{RSAT}_{n}(J, I \cup \{a\})
\end{aligned}
\tag{5.28}
$$

holds.

Let $C' \subseteq \beta(n')$. We distinguish between two cases: Either atom a is contained in the certificate, or not.

Let $C = C' \cup R^{+}_{a,n} \cup R^{B^-}_{a,n} \cup \{a\}$, then

$$\begin{aligned}
& J \setminus \{a\} \in \mathsf{re}_{n'}(C', I) \\
& \quad \iff J \setminus \{a\} \in \{A_{C'} \cup K \mid K \subseteq A_{[n']}, \\
& \quad\qquad \mathsf{RSAT}_{n'}(A_{C'} \cup K, I) = R_{C'} \cup R_{[n']}\} \\
& \quad \iff J \setminus \{a\} \in \{A_{C'} \cup K \mid K \subseteq A_{[n']}, \\
& \quad\qquad \mathsf{RSAT}_{n'}(A_{C'} \cup K, I) \cup R^{+}_{a,n} \cup R^{B^-}_{a,n} = \\
& \quad\qquad (R_{C'} \cup R^{+}_{a,n} \cup R^{B^-}_{a,n}) \cup R_{[n']}\} \\
& \quad \iff J \in \{A_C \cup K \mid K \subseteq A_{[n]}, \\
& \quad\qquad \mathsf{RSAT}_{n}(A_C \cup K, I \cup \{a\}) = R_C \cup R_{[n]}\} \\
& \quad \iff J \in \mathsf{re}_n(C, I \cup \{a\})
\end{aligned} \tag{5.29}$$

holds.

Let $C = C' \cup R^{+}_{a,n} \cup R^{B^-}_{a,n}$, then

$$\begin{aligned}
& \mathsf{re}_{n'}(C', I) \\
& = \{A_{C'} \cup K \mid K \subseteq A_{[n']}, \mathsf{RSAT}_{n'}(A_{C'} \cup K, I) = R_{C'} \cup R_{[n']}\} \\
& = \{A_{C'} \cup K \mid K \subseteq A_{[n']}, \\
& \quad \mathsf{RSAT}_{n'}(A_{C'} \cup K, I) \cup R^{+}_{a,n} \cup R^{B^-}_{a,n} = (R_{C'} \cup R^{+}_{a,n} \cup R^{B^-}_{a,n}) \cup R_{[n']}\} \\
& = \{A_C \cup K \mid K \subseteq A_{[n]}, \mathsf{RSAT}_{n}(A_C \cup K, I \cup \{a\}) = R_C \cup R_{[n]}\} \\
& = \mathsf{re}_n(C, I \cup \{a\})
\end{aligned} \tag{5.30}$$

holds.

We show now that $\mathcal{E}(\theta) = \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \{I \cup \{a\} \mid I \in \mathcal{E}(\theta')\}$ for a given $\theta = (n, M, \mathcal{C})$.

Let us first recall the definition of $\mathcal{E}(\theta)$.

$$\begin{aligned}
\mathcal{E}(\theta) = \{I \mid\ & I \in \mathsf{e}_n(M) \text{ and} \\
& \forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in \mathsf{re}_n(C, I)]\}
\end{aligned}$$

Also recall that the condition $I \in \mathsf{e}_n(M)$ is called Condition $\mathcal{A}$ of θ, and $\forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in \mathsf{re}_n(C, I)]$ is called Condition $\mathcal{C}$ of θ with respect to I.

We start by showing $\mathcal{E}(\theta) \supseteq \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \{I \cup \{a\} \mid I \in \mathcal{E}(\theta')\}$. Let $I \in \mathcal{E}(\theta')$ for some $\theta' \prec_{\mathcal{T}} \theta$ with $\theta' = (n', M', \mathcal{C}')$.

We show that Condition $\mathcal{C}$ holds for θ wrt. I.

Let C be an arbitrary subset of $\beta(n)$.

Suppose that $C \neq M' \cup R^{+}_{a,n} \cup R^{B^-}_{a,n}$.

Suppose that $C \in \mathcal{C}$ where $a \in C$. From the definition of $\prec_{\mathcal{T}}$, we know that there exists a $C' \in \mathcal{C}'$ such that $C = C' \cup R^{+}_{a,n} \cup \{a\} \cup R^{B^-}_{a,n}$. Since Condition $\mathcal{C}$ holds for θ' wrt. I, we know that there exists a $J \subset I$ s.t. $J \in \mathtt{re}_{n'}(C', I)$. Now we can conclude from (5.29) that $J \cup \{a\} \in \mathtt{re}_n(C, I \cup \{a\})$.

Suppose that $C \in \mathcal{C}$ where $a \notin C$. From the definition of $\prec_{\mathcal{T}}$, we know that there exists a $C' \in \mathcal{C}'$ where $C = C' \cup R^{+}_{a,n} \cup R^{B^-}_{a,n}$. Since Condition $\mathcal{C}$ holds for θ' wrt. I, we know that there exists a $J \subset I$ s.t. $J \in \mathtt{re}_{n'}(C', I)$. Now we can conclude from (5.30) that $J \in \mathtt{re}_n(C, I \cup \{a\})$.

Suppose that $J \cup \{a\} \in \mathtt{re}_n(C, I \cup \{a\})$ where $J \subset I$ and $a \in C$. From (5.29) we know that there exists a $J \subset I$ s.t. $J \in \mathtt{re}_{n'}(C', I)$ for some C' where $C' \cup R^{+}_{a,n} \cup R^{B^-}_{a,n} = C$ holds. Since Condition $\mathcal{C}$ holds for θ' wrt. I, we know $C' \in \mathcal{C}'$. From the definition of $\prec_{\mathcal{T}}$ we can now conclude $C \in \mathcal{C}$.

Suppose that $J \in \mathtt{re}_n(C, I \cup \{a\})$ where $J \subset (I \cup \{a\})$ and $a \notin C$. From (5.30) we know that $J \in \mathtt{re}_{n'}(C', I)$ for some C' where $C' \cup R^{+}_{a,n} \cup R^{B^-}_{a,n} = C$ holds. Since Condition $\mathcal{C}$ holds for θ' wrt. I, we know that $C' \in \mathcal{C}'$. From the definition of $\prec_{\mathcal{T}}$ we can now conclude $C \in \mathcal{C}$.

Suppose that $C = M' \cup R^{+}_{a,n} \cup R^{B^-}_{a,n}$ and thus $C \in \mathcal{C}$ by definition. It remains to show that there exists a $J \subset (I \cup \{a\})$ s.t. $J \in \mathtt{re}_n(C, I \cup \{a\})$. Since

- $I \models_{A_{(n)}} R_{[n]}$ all rules no longer visible are satisfied,
- $I \models_{A_{(n)}} R_{M'}$ only satisfied rules are in the assignment,
- $I \models_{A_{(n)}} R^{+}_{a,n}$ per definition of $R^{+}_{a,n}$ since a is not included in I, and
- $I \models_{A_{(n)}} R^{B^-}_{a,n}$ per definition of $R^{B^-}_{a,n}$,

we can conclude that $I \in \mathtt{re}_n(C, I \cup \{a\})$. Thus, since $I \subset (I \cup \{a\})$ holds trivially, we conclude that there exists a $J \subset (I \cup \{a\})$ s.t. $J \in \mathtt{re}_n(C, I \cup \{a\})$.

Thus, Condition $\mathcal{C}$ holds for θ wrt. I and together with (5.26), that is Condition $\mathcal{A}$ holds for θ, we can conclude $I \cup \{a\} \in \mathcal{E}(\theta$.

Next we show $\mathcal{E}(\theta) \subseteq \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \{I \cup \{a\} \mid I \in \mathcal{E}(\theta')\}$. Let $I \in \mathcal{E}(\theta)$, we show that there exists a θ' with $\theta' \prec_{\mathcal{T}} \theta$ such that $I \setminus \{a\} \in \mathcal{E}(\theta')$.

Let θ' be defined as $\theta' = (n', M', \mathcal{C}')$ where the following holds.

$$\begin{aligned} M' &= (A_M \setminus \{a\}) \cup (\mathsf{SAT}_{n'}(I \setminus \{a\}) \cap R_M) \\ \mathcal{C}' &= \{(A_C \setminus \{a\}) \cup H \mid C \in \mathcal{C}, \\ &\qquad H \subseteq R_C, \exists J \subset I \text{ s.t. } J \in \mathsf{re}_{n'}(A_C \setminus \{a\} \cup H, I \setminus \{a\})\}. \end{aligned}$$

Thus, $\theta' \prec_{\mathcal{T}} \theta$ holds.

We show that Condition $\mathcal{C}$ holds for θ' wrt. I.

Let $C' \in \mathcal{C}'$, then we know that there exists a $C \in \mathcal{C}$ where $C = C' \cup \{a\} \cup R^{+}_{a,n} \cup R^{B^-}_{a,n}$ (resp. $C = C' \cup R^{+}_{a,n} \cup R^{B^-}_{a,n}$) holds. Since Condition $\mathcal{C}$ holds for θ wrt. I, we know that there exists a $J \subset I$ s.t. $J \in \mathsf{re}_n(C, I)$. From (5.29) (resp. (5.30)) we can now conclude $J \in \mathsf{re}_{n'}(C', I \setminus \{a\})$.

Let C' be an arbitrary subset of $\beta(n')$, i.e., $C' \subseteq \beta(n')$. Suppose that $J \in \mathsf{re}_{n'}(C', I \setminus \{a\})$ where $J \subset I \setminus \{a\}$, from (5.29) (resp. (5.30)) we know that then $J \cup \{a\} \in \mathsf{re}_n(C, I)$ (resp. $J \in \mathsf{re}_n(C, I)$) where $C = C' \cup \{a\} \cup R^{+}_{a,n} \cup R^{B^-}_{a,n}$ (resp. $C = C' \cup R^{+}_{a,n} \cup R^{B^-}_{a,n}$) holds. Since Condition $\mathcal{C}$ holds for θ wrt. I, we also know that $C \in \mathcal{C}$, and hence $C' \in \mathcal{C}'$.

Thus, Condition $\mathcal{C}$ holds for θ' wrt. $I \setminus \{a\}$ and together with (5.26), that is Condition $\mathcal{A}$ holds for θ', we can conclude $I \setminus \{a\} \in \mathcal{E}(\theta')$.

Thus, we have shown for a given θ that

- for each θ' with $\theta' \prec_{\mathcal{T}} \theta$, and each $I \in \mathcal{E}(\theta')$ also $I \cup \{a\} \in \mathcal{E}(\theta)$ holds, and
- for all $I \in \mathcal{E}(\theta)$ there exists a θ' with $\theta' \prec_{\mathcal{T}} \theta$ such that $I \setminus \{a\} \in \mathcal{E}(\theta')$.

Hence $\mathcal{E}(\theta) = \bigcup_{\theta' \prec_{\mathcal{T}} \theta} \{I \cup \{a\} \mid I \in \mathcal{E}(\theta')\}$ holds.

.

CASE 5, n is a branch node. Let $\theta = (n, M, \mathcal{C})$, further let $\theta_1 = (n_1, M_1, \mathcal{C}_1)$ and $\theta_2 = (n_2, M_2, \mathcal{C}_2)$ where $(\theta_1, \theta_2) \prec_{\mathcal{T}} \theta$. Hence $A_{M_1} = A_{M_2}$ holds. From the definition of the branch node, we know that

$$\begin{aligned} M &= A_{M_1} \cup R_{M_1} \cup R_{M_2} \\ \mathcal{C} &= \{C_1 \cup C_2 \mid C_1 \in \mathcal{C}_1, C_2 \in \mathcal{C}_2, A_{C_1} = A_{C_2}\} \\ &\quad \cup \{M_1 \cup C_2 \mid C_2 \in \mathcal{C}_2, A_{M_1} = A_{C_2}\} \\ &\quad \cup \{C_1 \cup M_2 \mid C_1 \in \mathcal{C}_1, A_{C_1} = A_{M_2}\}. \end{aligned}$$

That is, in M are all rules combined that are true under the current assignment of the atoms.

Further, we know that the following equivalences hold.

$$\begin{aligned} A_{M_1} &= A_{M_2} = A_M & R_{M_1} \cup R_{M_2} &= R_M \\ A_{(n_1)} \cup A_{(n_2)} &= A_{(n)} & R_{(n_1)} \cup R_{(n_2)} &= R_{(n)} \\ A_{(n_1)} \cap A_{(n_2)} &= A_{\beta(n)} & R_{(n_1)} \cap R_{(n_2)} &= R_{\beta(n)} \\ A_{[n_1]} \cup A_{[n_2]} &= A_{[n]} & R_{[n_1]} \cup R_{[n_2]} &= R_{[n]} \\ A_{[n_1]} \cap A_{[n_2]} &= \emptyset & R_{[n_1]} \cap R_{[n_2]} &= \emptyset \end{aligned}$$

Let us now establish some equivalences based on these observations.

We first show for $M, M_1, M_2 \subseteq \beta(n)$ where $A_M = A_{M_1} = A_{M_2}$ and $R_M = R_{M_1} \cup R_{M_2}$ that the following holds.

$$\mathsf{e}_n(M) = \{I_1 \cup I_2 \mid I_1 \in \mathsf{e}_{n_1}(M_1), I_2 \in \mathsf{e}_{n_2}(M_2)\} \tag{5.31}$$

Let us recall the definition of $\mathsf{e}_n(M)$.

$$\mathsf{e}_n(M) = \{A_M \cup K \mid K \subseteq A_{[n]}, \mathsf{SAT}_n(A_M \cup K) = R_M \cup R_{[n]}\}$$

For the following, let $I = A_M \cup K_1 \cup K_2$ where $K_1 \subseteq A_{[n_1]}$ and $K_2 \subseteq A_{[n_2]}$, further let $I_1 = A_M \cup K_1$ and $I_2 = A_M \cup K_2$.

We start by establishing that

$$\mathsf{SAT}_n(I) = \mathsf{SAT}_{n_1}(I_1) \cup \mathsf{SAT}_{n_2}(I_2) \tag{5.32}$$

holds. Recall the definition

$$\mathsf{SAT}_n(I) = \{p \mid p \in R_{(n)}, I \models_{A_{(n)}} p\}.$$

We first show that $\mathsf{SAT}_n(I) \subseteq \mathsf{SAT}_{n_1}(I_1) \cup \mathsf{SAT}_{n_2}(I_2)$ holds. Let $r \in \mathsf{SAT}_n(I)$. From the definition of $\mathsf{SAT}_n(I)$, it holds that $r \in R_{(n)}$ and since $R_{(n)} = R_{\beta(n)} \cup R_{[n_1]} \cup R_{[n_2]}$, it holds that $r \in R_{\beta(n)} \cup R_{[n_1]} \cup R_{[n_2]}$.

Suppose that $r \in R_{[n_1]}$. From the shape of the tree decomposition we know that only atoms in $A_{(n_1)}$ occur in r. Since $I_1 = I \cap A_{(n_1)}$, we can conclude that $I_1 \models_{A_{(n_1)}} r$, and thus $r \in \mathsf{SAT}_{n_1}(I_1)$ holds. The case for $r \in R_{[n_2]}$ is symmetric.

Suppose that $r \in R_{\beta(n)}$. Since $I = A_M \cup K_1 \cup K_2$ where $K_1 \subseteq A_{[n_1]}$ and $K_2 \subseteq A_{[n_2]}$, we know that at least one of the following holds.

- $A_M \models_{A_M} r$

- $K_1 \models_{A_{[n_1]}} r$
- $K_2 \models_{A_{[n_2]}} r$

Suppose that $A_M \models_{A_M} r$, then $I_1 \models_{A_{(n_1)}} r$ and $I_2 \models_{A_{(n_2)}} r$ since $A_M \subseteq I_1$ and $A_M \subseteq A_{(n_1)}$ respectively $A_M \subseteq I_2$ and $A_M \subseteq A_{(n_2)}$. Thus $r \in \mathsf{SAT}_{n_1}(I_1)$ and $r \in \mathsf{SAT}_{n_2}(I_2)$ holds.

Suppose that $K_1 \models_{A_{[n_1]}} r$, then $I_1 \models_{A_{(n_1)}} r$ holds since $K_1 \subseteq I_1$ and $A_{[n_1]} \subseteq A_{(n_1)}$. Thus $r \in \mathsf{SAT}_{n_1}(I_1)$. The case for $K_2 \models_{A_{[n_2]}} r$ is symmetric.

Next, we show that $\mathsf{SAT}_n(I) \supseteq \mathsf{SAT}_{n_1}(I_1) \cup \mathsf{SAT}_{n_2}(I_2)$ holds. Let $r \in \mathsf{SAT}_{n_1}(I_1)$. From the definition of $\mathsf{SAT}_{n_1}(I_1)$, it holds that $r \in R_{(n_1)}$, and since $R_{(n)} = R_{(n_1)} \cup R_{(n_2)}$, we know that $r \in R_{(n)}$. Since $A_{(n_1)} \subseteq A_{(n)}$ and $I_1 \subseteq I$ we can conclude that $I \models_{A_{(n)}} r$, and hence that $r \in \mathsf{SAT}_n(I)$.

The case for $r \in \mathsf{SAT}_{n_2}(I_2)$ is symmetric.

Thus, (5.32) holds.

Next we show that

$$\mathsf{e}_n(M) \subseteq \{I_1 \cup I_2 \mid I_1 \in \mathsf{e}_{n_1}(M_1), I_2 \in \mathsf{e}_{n_2}(M_2)\}.$$

Suppose that $I \in \mathsf{e}_n(M)$, we show that $I_1 \in \mathsf{e}_{n_1}(M_1)$ and $I_2 \in \mathsf{e}_{n_2}(M_2)$ where $I = I_1 \cup I_2$. By definition, $I = A_M \cup K$ where $K \subseteq A_{[n]}$. Since $A_{[n]} = A_{[n_1]} \cup A_{[n_2]}$ we know that there exists a $K_1 \subseteq A_{[n_1]}$ and a $K_2 \subseteq A_{[n_2]}$ such that $I = A_M \cup K_1 \cup K_2$. Let $I_1 = A_M \cup K_1$ and $I_2 = A_M \cup K_2$. By definition $\mathsf{SAT}_n(I) = R_M \cup R_{[n]}$, further $R_M = R_{M_1} \cup R_{M_2}$ where $R_{M_1} = R_M \cap R_{(n_1)}$ and $R_{M_2} = R_M \cap R_{(n_2)}$. Since $R_{[n]} = R_{[n_1]} \cup R_{[n_2]}$ and from (5.32), we can conclude that $\mathsf{SAT}_{n_1}(I_1) = R_{M_1} \cup R_{[n_1]}$ and $\mathsf{SAT}_{n_2}(I_2) = R_{M_2} \cup R_{[n_2]}$. Thus, we can conclude that $I_1 \in \mathsf{e}_{n_1}(M_1)$ and $I_2 \in \mathsf{e}_{n_2}(M_2)$.

Now we show that

$$\mathsf{e}_n(M) \supseteq \{I_1 \cup I_2 \mid I_1 \in \mathsf{e}_{n_1}(M_1), I_2 \in \mathsf{e}_{n_2}(M_2)\}.$$

Suppose that $I_1 \in \mathsf{e}_{n_1}(M_1)$ and $I_2 \in \mathsf{e}_{n_2}(M_2)$. By definition $I_1 = A_{M_1} \cup K_1$ where $K_1 \subseteq A_{[n_1]}$ and $I_2 = A_{M_2} \cup K_2$ where $K_2 \subseteq A_{[n_2]}$. Let $I = A_M \cup K_1 \cup K_2$, and since $A_M = A_{M_1} = A_{M_2}$ it holds that $I = I_1 \cup I_2$. Further $\mathsf{SAT}_{n_1}(I_1) = R_{M_1} \cup R_{[n_1]}$ and $\mathsf{SAT}_{n_2}(I_2) = R_{M_2} \cup R_{[n_2]}$, by (5.32) we know that $\mathsf{SAT}_n(I) = R_{M_1} \cup R_{M_2} \cup R_{[n_1]} \cup R_{[n_2]}$. Since $R_M = R_{M_1} \cup R_{M_2}$ and $R_{[n]} = R_{[n_1]} \cup R_{[n_2]}$ we can conclude that $\mathsf{SAT}_n(I) = R_M \cup R_{[n]}$, and since $A_{[n]} = A_{[n_1]} \cup A_{[n_2]}$, we have $I \in \mathsf{e}_n(M)$.

Thus, we have shown that (5.31) holds.

Next, we show that for a given $I_1 = A_M \cup K_1$ where $K_1 \subseteq A_{[n_1]}$ and $I_2 = A_M \cup K_2$ where $K_2 \subseteq A_{[n_2]}$ and $I = I_1 \cup I_2$,

and given $C, C_1, C_2 \subseteq \beta(n)$ such that $A_C = A_{C_1} = A_{C_2}$ and $R_C = R_{C_1} \cup R_{C_2}$ that

$$\mathsf{re}_n(C, I) = \{J_1 \cup J_2 \mid J_1 \in \mathsf{re}_{n_1}(C_1, I_1), J_2 \in \mathsf{re}_{n_2}(C_2, I_2)\} \tag{5.33}$$

holds.

Let us first recall the definition of $\mathsf{re}_n(C, I)$.

$$\mathsf{re}_n(C, I) = \{A_C \cup K \mid K \subseteq A_{[n]}, \mathsf{RSAT}_n(A_C \cup K, I) = R_C \cup R_{[n]}\}$$

We first establish that for $J_1 = A_M \cup K_1$ where $K_1 \subseteq A_{[n_1]}$ and $J_2 = A_M \cup K_2$ where $K_2 \subseteq A_{[n_2]}$ and $J = J_1 \cup J_2$ the following holds.

$$\mathsf{RSAT}_n(J, I) = \mathsf{RSAT}_{n_1}(J_1, I_1) \cup \mathsf{RSAT}_{n_2}(J_2, I_2) \tag{5.34}$$

Let us recall the definition of $\mathsf{RSAT}_n(J, I)$.

$$\mathsf{RSAT}_n(J, I) = \{r \mid r \in R_{(n)}, J \models_{A_{(n)}} r \text{ or } \mathrm{B}^-(r) \cap I \neq \emptyset\}$$

We show that $\mathsf{RSAT}_n(J, I) \subseteq \mathsf{RSAT}_{n_1}(J_1, I_1) \cup \mathsf{RSAT}_{n_2}(J_2, I_2)$. Let $r \in \mathsf{RSAT}_n(J, I)$. From the definition of $\mathsf{RSAT}_n(J, I)$, it holds that $J \models_{A_{(n)}} r$, that $\mathrm{B}^-(r) \cap I \neq \emptyset$, or both.

Suppose that $J \models_{A_{(n)}} r$. From the definition of $\mathsf{RSAT}_n(J, I)$, it holds that $r \in R_{(n)}$ and since $R_{(n)} = R_{\beta(n)} \cup R_{[n_1]} \cup R_{[n_2]}$, it holds that $r \in R_{\beta(n)} \cup R_{[n_1]} \cup R_{[n_2]}$.

Suppose that $r \in R_{[n_1]}$. From the shape of the tree decomposition we know that only atoms in $A_{(n_1)}$ occur in r. Since $J_1 = J \cap A_{(n_1)}$, we can conclude that $J_1 \models_{A_{(n_1)}} r$, and thus $r \in \mathsf{RSAT}_{n_1}(J_1, I_1)$ holds. The case for $r \in R_{[n_2]}$ is symmetric.

Suppose that $r \in R_{\beta(n)}$. Since $J = A_M \cup K_1 \cup K_2$ where $K_1 \subseteq A_{[n_1]}$ and $K_2 \subseteq A_{[n_2]}$, we know that at least one of the following holds.

- $A_M \models_{A_M} r$
- $K_1 \models_{A_{[n_1]}} r$
- $K_2 \models_{A_{[n_2]}} r$

Suppose that $A_M \models_{A_M} r$, then $J_1 \models_{A_{(n_1)}} r$ and $J_2 \models_{A_{(n_2)}} r$ since $A_M \subseteq J_1$ and $A_M \subseteq A_{(n_1)}$ respectively $A_M \subseteq J_2$ and $A_M \subseteq A_{(n_2)}$. Thus $r \in \mathsf{RSAT}_{n_1}(J_1, I_1)$ and $r \in \mathsf{RSAT}_{n_2}(J_2, I_2)$ holds.

Suppose that $K_1 \models_{A_{[n_1]}} r$, then $J_1 \models_{A_{(n_1)}} r$ holds since $K_1 \subseteq J_1$ and $A_{[n_1]} \subseteq A_{(n_1)}$. Thus $r \in \mathsf{RSAT}_{n_1}(J_1, I_1)$. The case for $K_2 \models_{A_{[n_2]}} r$ is symmetric.

Suppose that $\text{B}^-(r) \cap I \neq \emptyset$. Since $I = I_1 \cup I_2$, it holds that $\text{B}^-(r) \cap I_1 \neq \emptyset$ or $\text{B}^-(r) \cap I_2 \neq \emptyset$, and hence $r \in \mathsf{RSAT}_{n_1}(J_1, I_1)$ or $r \in \mathsf{SAT}_{n_2}(J_2, I_2)$.

Next, we show that $\mathsf{RSAT}_n(J, I) \supseteq \mathsf{RSAT}_{n_1}(J_1, I_1) \cup \mathsf{RSAT}_{n_2}(J_2, I_2)$ holds. Let $r \in \mathsf{RSAT}_{n_1}(J_1, I_1)$. From the definition of $\mathsf{RSAT}_{n_1}(J_1, I_1)$, it holds that $J_1 \models_{A_{(n_1)}} r$ or $\text{B}^-(r) \cap I_1 \neq \emptyset$. Further, it holds that $r \in R_{(n_1)}$, and since $R_{(n)} = R_{(n_1)} \cup R_{(n_2)}$, we know that $r \in R_{(n)}$. Since $A_{(n_1)} \subseteq A_{(n)}$ and $J_1 \subseteq J$ we can conclude that $J \models_{A_{(n)}} r$ if $J_1 \models_{A_{(n_1)}} r$ or since $I_1 \subseteq I$ it holds that $\text{B}^-(r) \cap I \neq \emptyset$ if $\text{B}^-(r) \cap I_1 \neq \emptyset$. In either case $r \in \mathsf{RSAT}_n(J, I)$ holds. The case for $r \in \mathsf{RSAT}_{n_2}(J_2, I_2)$ is symmetric.

Thus, (5.34) holds.

Next we show that

$$\mathsf{re}_n(C, I) \subseteq \{J_1 \cup J_2 \mid J_1 \in \mathsf{re}_{n_1}(C_1, I_1),\\ J_2 \in \mathsf{e}_{n_2}(C_2, I_2)\}.$$

Suppose that $J \in \mathsf{re}_n(C, I)$, we show that $J_1 \in \mathsf{re}_{n_1}(C_1, I_1)$ and $J_2 \in \mathsf{re}_{n_2}(C_2, I_2)$ where $J = J_1 \cup J_2$. By definition, $J = A_C \cup K$ where $K \subseteq A_{[n]}$. Since $A_{[n]} = A_{[n_1]} \cup A_{[n_2]}$ we know that there exists a $K_1 \subseteq A_{[n_1]}$ and a $K_2 \subseteq A_{[n_2]}$ such that $J = A_C \cup K_1 \cup K_2$. Let $J_1 = A_M \cup K_1$ and $J_2 = A_C \cup K_2$. By definition $\mathsf{RSAT}_n(J, I) = R_C \cup R_{[n]}$, further $R_C = R_{C_1} \cup R_{C_2}$ where $R_{C_1} = R_C \cap R_{(n_1)}$ and $R_{C_2} = R_C \cap R_{(n_2)}$. Since $R_{[n]} = R_{[n_1]} \cup R_{[n_2]}$ and from (5.34), we can conclude that $\mathsf{RSAT}_{n_1}(J_1, I_1) = R_{C_1} \cup R_{[n_1]}$ and $\mathsf{RSAT}_{n_2}(J_2, I_2) = R_{C_2} \cup R_{[n_2]}$. Thus, we can conclude that $J_1 \in \mathsf{re}_{n_1}(C_1, I_1)$ and $J_2 \in \mathsf{re}_{n_2}(C_2, I_2)$.

Now we show that

$$\mathsf{re}_n(C, I) \supseteq \{J_1 \cup J_2 \mid J_1 \in \mathsf{re}_{n_1}(C_1, I_1),\\ J_2 \in \mathsf{e}_{n_2}(C_2, I_2)\}.$$

Suppose that $J_1 \in \mathsf{re}_{n_1}(C_1, I_1)$ and $J_2 \in \mathsf{re}_{n_2}(C_2, I_2)$. By definition $J_1 = A_{C_1} \cup K_1$ where $K_1 \subseteq A_{[n_1]}$ and $J_2 = A_{C_2} \cup K_2$ where $K_2 \subseteq A_{[n_2]}$. Let $J = A_C \cup K_1 \cup K_2$, and since $A_C = A_{C_1} = A_{C_2}$ it holds that $J = J_1 \cup J_2$. Further $\mathsf{RSAT}_{n_1}(J_1, I_1) = R_{C_1} \cup R_{[n_1]}$ and $\mathsf{RSAT}_{n_2}(J_2, I_2) = R_{C_2} \cup R_{[n_2]}$, by (5.34) we know that $\mathsf{RSAT}_n(J, I) = R_{C_1} \cup R_{C_2} \cup R_{[n_1]} \cup R_{[n_2]}$. Since $R_C = R_{C_1} \cup R_{C_2}$ and $R_{[n]} = R_{[n_1]} \cup R_{[n_2]}$ we can conclude that $\mathsf{RSAT}_n(J, I) = R_C \cup R_{[n]}$, and since $A_{[n]} = A_{[n_1]} \cup A_{[n_2]}$, we have $J \in \mathsf{re}_n(C, I)$.

Thus, we have shown that (5.33) holds.

We show now that $\mathcal{E}(\theta) = \bigcup_{(\theta_1,\theta_2)\prec_{\mathcal{T}}\theta}\{I_1 \cup I_2 \mid I_1 \in \mathcal{E}(\theta_1), I_2 \in \mathcal{E}(\theta_2)\}$ for a given $\theta = (n, M, \mathcal{C})$.

Let us first recall the definition of $\mathcal{E}(\theta)$.

$$\begin{aligned}\mathcal{E}(\theta) = \{I \mid\ & I \in \mathsf{e}_n(M) \text{ and} \\ & \forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in \mathsf{re}_n(C, I)]\}\end{aligned}$$

Also recall that the condition $I \in \mathsf{e}_n(M)$ is called Condition $\mathcal{A}$ of θ, and $\forall C \subseteq \beta(n)[C \in \mathcal{C} \iff \exists J \subset I \text{ s.t. } J \in \mathsf{re}_n(C, I)]$ is called Condition $\mathcal{C}$ of θ with respect to I.

We start by showing

$$\mathcal{E}(\theta) \supseteq \bigcup_{(\theta_1,\theta_2)\prec_{\mathcal{T}}\theta} \{I_1 \cup I_2 \mid I_1 \in \mathcal{E}(\theta_1), I_2 \in \mathcal{E}(\theta_2)\}.$$

Let $I_1 \in \mathcal{E}(\theta_1)$ and $I_2 \in \mathcal{E}(\theta_2)$, for some $(\theta_1, \theta_2) \prec_{\mathcal{T}} \theta$ where $\theta_1 = (n_1, M_1, \mathcal{C}_1)$, $\theta_2 = (n_2, M_2, \mathcal{C}_2)$. We show that then $I \in \mathcal{E}(\theta)$ where $I = I_1 \cup I_2$.

We show that Condition $\mathcal{C}$ holds for θ wrt. I.

Suppose that $C \in \mathcal{C}$, we show that there exists a $J \subset I$ such that $J \in \mathsf{re}_n(C, I)$. From the definition of $\prec_{\mathcal{T}}$, we know that C is either

1. $C = C_1 \cup C_2$ where $C_1 \in \mathcal{C}_1$ and $C_2 \in \mathcal{C}_2$,
2. $C = M_1 \cup C_2$ where $C_2 \in \mathcal{C}_2$, or
3. $C = C_1 \cup M_2$ where $C_1 \in \mathcal{C}_1$.

Further we know that in Case 1 $A_{C_1} = A_{C_2}$ and $A_{M_1} = A_{C_2}$ in Case 2 (resp. $A_{C_1} = A_{M_2}$ in Case 3) holds. In the following paragraphs, we will look at these cases in detail.

Suppose that $C = C_1 \cup C_2$. Since Condition $\mathcal{C}$ holds for θ_1 wrt. I and θ_2 wrt. I, we know that there exists a $J_1 \subset I_1$ s.t. $J_1 \in \mathsf{re}_{n_1}(C_1, I_1)$ and that there exists a $J_2 \subset I_2$ s.t. $J_2 \in \mathsf{re}_{n_2}(C_2, I_2)$. From (5.33) we can now conclude $J_1 \cup J_2 \in \mathsf{re}_n(C, I_1 \cup I_2)$ where $J_1 \cup J_2 \subset I_1 \cup I_2$.

Suppose that $C = C_1 \cup M_2$. We know that Condition $\mathcal{C}$ holds for θ_1 wrt. I, and thus there exists a $J_1 \subset I_1$ such that $J_1 \in \mathsf{re}_{n_1}(C_1, I_1)$. Further, we know that $I_2 \in \mathsf{re}_{n_2}(M_2, I_2)$ since $I_2 \in \mathsf{e}_{n_2}(M_2)$. From (5.33) we can now conclude $J_1 \cup I_2 \in \mathsf{re}_n(C, I_1 \cup I_2)$ where $J_1 \cup I_2 \subset I_1 \cup I_2$.

The case for $C = M_1 \cup C_2$ is symmetric.

Suppose that $J \in \mathsf{re}_n(C, I_1 \cup I_2)$ where $J \subset I_1 \cup I_2$ holds, we show that $C \in \mathcal{C}$. Again, we have three cases,

- $J = J_1 \cup J_2$ where $J_1 \subset I_1$ and $J_2 \subset I_2$,
- $J = J_1 \cup I_2$ where $J_1 \subset I_1$, or
- $J = I_1 \cup J_2$ where $J_2 \subset I_2$.

Suppose that $J_1 \cup J_2 \in \mathsf{re}_n(C, I_1 \cup I_2)$ where $J_1 \subset I_1$ and $J_2 \subset I_2$. Suppose that $C = C_1 \cup C_2$ where $C_1 \in \mathcal{C}_1$ and $C_2 \in \mathcal{C}_2$ such that $A_{C_1} = A_{C_2}$. From (5.33) we know that $J_1 \in \mathsf{re}_{n_1}(C_1, I_1)$ and $J_2 \in \mathsf{re}_{n_2}(C_2, I_2)$. Since Condition $\mathcal{C}$ holds for θ_1 wrt. I and θ_2 wrt. I, we know that $C_1 \in \mathcal{C}_1$ and $C_2 \in C_2$. From the definition of $\prec_{\mathcal{T}}$ we can now conclude $C \in \mathcal{C}$.

Suppose that $J_1 \cup I_2 \in \mathsf{re}_n(C, I_1 \cup I_2)$ where $J_1 \subset I_1$. Suppose that $C = C_1 \cup M_2$ where $C_1 \in \mathcal{C}_1$ such that $A_{C_1} = A_{M_2}$. From (5.33) we know that $J_1 \in \mathsf{re}_{n_1}(C_1, I_1)$ and $I_2 \in \mathsf{re}_{n_2}(M_2, I_2)$. Since Condition $\mathcal{C}$ holds for θ_1 wrt. I, we know that $C_1 \in \mathcal{C}_1$. From the definition of $\prec_{\mathcal{T}}$ we can now conclude $C \in \mathcal{C}$.

The case for $J = I_1 \cup J_2$ is symmetric.

Thus, Condition $\mathcal{C}$ holds for θ wrt. I and together with (5.31) we can conclude that Condition $\mathcal{A}$ for θ holds as well, thus $I_1 \cup I_2 \in \mathcal{E}(\theta)$.

Next we show

$$\mathcal{E}(\theta) \subseteq \bigcup_{(\theta_1,\theta_2) \prec_{\mathcal{T}} \theta} \{I_1 \cup I_2 \mid I_1 \in \mathcal{E}(\theta_1), I_2 \in \mathcal{E}(\theta_2)\}.$$

Let $I \in \mathcal{E}(\theta)$, further let $I_1 \subseteq I \cap A_{(n_1)}$ and $I_2 \subseteq I \cap A_{(n_2)}$, hence $I = I_1 \cup I_2$ holds. We show that there exist a θ_1 and a θ_2 such that $(\theta_1, \theta_2) \prec_{\mathcal{T}} \theta$ where $I_1 \in \mathcal{E}(\theta_1)$ and $I_2 \in \mathcal{E}(\theta_2)$. Let the $\mathcal{T}$-interpretations be defined as follows: $\theta_1 = (n_1, M_1, \mathcal{C}_1)$, $\theta_2 = (n_2, M_2, \mathcal{C}_2)$, where

$$\begin{aligned}
M_1 &= A_M \cup (R_M \cap R_{(n_1)})\\
\mathcal{C}_1 &= \{A_C \cup (R_C \cap R_{(n_1)}) \mid C \in \mathcal{C}\}\\
&\quad \cup \{C \subseteq \beta(n_1) \mid \nexists C' \in \mathcal{C} \text{ s.t. } A_{C'} = A_C,\\
&\qquad \exists J \subset I_1 \text{ s.t. } J \in \mathsf{re}_{n_1}(C, I_1)\}\\
M_2 &= A_M \cup (R_M \cap R_{(n_2)})\\
\mathcal{C}_2 &= \{A_C \cup (R_C \cap R_{(n_2)}) \mid C \in \mathcal{C}\}\\
&\quad \cup \{C \subseteq \beta(n_2) \mid \nexists C' \in \mathcal{C} \text{ s.t. } A_{C'} = A_C,\\
&\qquad \exists J \subset I_2 \text{ s.t. } J \in \mathsf{re}_{n_2}(C, I_2)\}.
\end{aligned}$$

We show that Condition $\mathcal{C}$ holds for θ_1 and θ_2 wrt. I.

Suppose that $C_1 \in \mathcal{C}_1$ and $C_2 \in \mathcal{C}_2$ where $A_{C_1} = A_{C_2}$, we show that Condition $\mathcal{C}$ holds for θ_1 wrt. I and θ_2 wrt. I. From

the definition of $\prec_{\mathcal{T}}$ we know that there exists a $C \in \mathcal{C}$ where $C = C_1 \cup C_2$. Since Condition $\mathcal{C}$ holds for θ wrt. I, we also know that there exists a $J \subset (I_1 \cup I_2)$ s.t. $J \in \mathtt{re}_n(C, I_1 \cup I_2)$. From (5.33) we can now conclude that $J_1 \in \mathtt{re}_{n_1}(C_1, I_1)$ and $J_2 \in \mathtt{re}_{n_2}(C_2, I_2)$ where $J = J_1 \cup J_2$.

Suppose that $C_1 \in \mathcal{C}_1$ and $A_{M_2} = A_{C_1}$ we show that Condition $\mathcal{C}$ holds for θ_1 wrt. I. From the definition of $\prec_{\mathcal{T}}$ we know that there exists a $C \in \mathcal{C}$ where $C = C_1 \cup M_2$. Since Condition $\mathcal{C}$ holds for θ wrt. I, we also know that there exists a $J \subset (I_1 \cup I_2)$ s.t. $J \in \mathtt{re}_n(C, I_1 \cup I_2)$. From (5.33) we can now conclude that $J_1 \in \mathtt{re}_{n_1}(C_1, I_1)$ where $J = J_1 \cup I_2$, and thus $J_1 \subset I_1$.

The case for $C_2 \in \mathcal{C}_2$ and $A_{M_1} = A_{C_2}$ is symmetric.

For all other cases of $C_1 \in \mathcal{C}_1$ respectively $C_2 \in \mathcal{C}_2$, there exists a $J_1 \subset I_1$ such that $J_1 \in \mathtt{re}_{n_1}(C_1, I_1)$ respectively a $J_2 \subset I_2$ such that $J_2 \in \mathtt{re}_{n_2}(C_2, I_2)$ per definition of $\mathcal{C}_1$ respectively $\mathcal{C}_2$.

Suppose that $J_1 \in \mathtt{re}_{n_1}(C_1, I_1)$ where $J_1 \subset I_1$ holds and $J_2 \in \mathtt{re}_{n_2}(C_2, I_2)$ where $J_2 \subset I_2$ holds. From (5.33) we know that $J_1 \cup J_2 \in \mathtt{re}_n(C, I_1 \cup I_2)$ where $J_1 \cup J_2 \subset I_1 \cup I_2$ and $C = C_1 \cup C_2$. Since Condition $\mathcal{C}$ holds for θ wrt. I, we also know that $C \in \mathcal{C}$. From the definition of $\prec_{\mathcal{T}}$ we can now conclude $C_1 \in \mathcal{C}_1$ and $C_2 \in \mathcal{C}_2$.

Suppose that $J_1 \in \mathtt{re}_{n_1}(C_1, I_1)$ where $J_1 \subset I_1$ and $A_{M_2} = A_{C_1}$. From (5.33) we know that $J_1 \cup I_2 \in \mathtt{re}_n(C, I_1 \cup I_2)$ where $J_1 \cup I_2 \subset I_1 \cup I_2$ and $C = C_1 \cup M_2$. Since Condition $\mathcal{C}$ holds for θ wrt. I, we also know that $C \in \mathcal{C}$. From the definition of $\prec_{\mathcal{T}}$ we can now conclude $C_1 \in \mathcal{C}_1$.

The case for $J_2 \in \mathtt{re}_{n_2}(C_2, I_2)$ where $J_2 \subset I_2$ and $A_{M_1} = A_{C_2}$ is symmetric.

For all other cases of $J_1 \in \mathtt{re}_{n_1}(C_1, I_1)$ where $J_1 \subset I_1$ respectively $J_2 \in \mathtt{re}_{n_2}(C_2, I_2)$ where $J_2 \subset I_2$ we know that $C_1 \in \mathcal{C}_1$ respectively $C_2 \in \mathcal{C}_2$ per definition of $\mathcal{C}_1$ respectively $\mathcal{C}_2$.

Thus, Condition $\mathcal{C}$ holds for θ_1 wrt. I and θ_2 wrt. I. Together with (5.31), that is Condition $\mathcal{A}$ of θ_1 and θ_2, we can conclude $I_1 \in \mathcal{E}(\theta_1)$ and $I_2 \in \mathcal{E}(\theta_2)$.

Thus, we have shown for a given θ that

- for each θ_1 and θ_2 with $(\theta_1, \theta_2) \prec_{\mathcal{T}} \theta$, and each $I_1 \in \mathcal{E}(\theta_1)$ and $I_2 \in \mathcal{E}(\theta_2)$ also $I_1 \cup I_2 \in \mathcal{E}(\theta)$ holds, and

- for all $I \in \mathcal{E}(\theta)$ there exists a θ_1 and a θ_2 with $(\theta_1, \theta_2) \prec_{\mathcal{T}} \theta$ such that $I_1 \in \mathcal{E}(\theta_1)$ and $I_2 \in \mathcal{E}(\theta_2)$ where $I = I_1 \cup I_2$.

Hence $\mathcal{E}(\theta) = \bigcup_{(\theta_1, \theta_2) \prec_{\mathcal{T}} \theta} \{I_1 \cup I_2 \mid I_1 \in \mathcal{E}(\theta_1), I_2 \in \mathcal{E}(\theta_2)\}$ holds.

.

Thus, the recursive production of $\mathcal{E}(\theta)$ is correct and equivalent to Definition 5.2.4. □

The following corollary follows directly from Lemma 5.2.12.

Corollary 5.2.13. *Let $\theta, \theta', \theta''$ be $\mathcal{T}$-interpretations, such that $\theta' \prec_{\mathcal{T}} \theta$ (resp. $(\theta', \theta'') \prec_{\mathcal{T}} \theta$). Then, θ is a $\mathcal{T}$-model iff θ' is $\mathcal{T}$-model (resp. both θ' and θ'' are $\mathcal{T}$-models).*

5.2.3 ASP *Consistency*

In this section we provide the algorithm for ASP consistency together with a runtime analysis.

Corollary 5.2.13 suggests the following algorithm. First, we establish the $\mathcal{T}$-models of leaf nodes, then we compute all remaining $\mathcal{T}$-models via $\prec_{\mathcal{T}}$ in a bottom-up manner. As soon as we have the $\mathcal{T}$-models for the root node, we check whether they include also a root model for $\mathcal{T}$.

Theorem 5.2.14. *Deciding $\mathcal{AS}(R) \neq \emptyset$ can be done in time $O(f(k) \cdot |R|)$, where k denotes the treewidth of R and f is a function that only depends on k but not on $|R|$.*

Proof. Computing the $\mathcal{T}$-models of a leaf node: since we have no objects to talk about, there exists exactly one $\mathcal{T}$-model ($\theta = (n, \emptyset, \emptyset)$) which can be computed in constant time, i.e., $O(1)$.

Computing the $\mathcal{T}$-models for the inner nodes: in the following let $g(k) = 2^{2^{k+1}}$ and $h(k) = 2^{k+1}$.

We first establish the time required by the helper functions and each node-type per $\mathcal{T}$-model (see Figure 5.3). Let the function $\Gamma_{node-type}(M', \mathcal{C}')$ (resp. $\Gamma_B(M', \mathcal{C}', M'', \mathcal{C}'')$ for a branch node) be the function to compute a new θ from the given assignments M' and M'' as well as the given certificates $\mathcal{C}'$ and $\mathcal{C}''$. We use $\| \circ (\cdot) \|$ to denote the number of steps required to compute the

result of $\circ(\cdot)$. We assume the use of a hash set with a perfect hashing function leading to $O(1)$ runtime for lookups and inserts.

$$\begin{aligned}
\|\alpha \cap \beta\| &= O(|\beta|) \\
\|\alpha \cup \beta\| &= O(|\alpha| + |\beta|) \\
\|\alpha \setminus \beta\| &= O(|\alpha|) \\
\|e \in \beta\| &= O(1) \\
\|\alpha = \beta\| &= O(|\alpha| + |\beta|) \\
\|M +_n a\| &= O(k) \\
\|M \times_n a\| &= O(k) \\
\|M \uplus_n r\| &= O(k) \\
\|R^{B^-}_{a,n}\| &= O(k)
\end{aligned}$$

Note that $\alpha \cap \beta$ is defined to traverse through the elements of β and check if they are contained in α.

From these runtimes, we can now establish the time required to compute a new $\mathcal{T}$-model from a given assignment a set of certificates.

$$\begin{aligned}
\|\Gamma_{a-\text{AR}}(M', \mathcal{C}')\| &= O(g(k)) \\
\|\Gamma_{r-\text{RR}}(M', \mathcal{C}')\| &= O(g(k)) \\
\|\Gamma_{a-\text{AI}}(M', \mathcal{C}')\| &= O(g(k)) \\
\|\Gamma_{a-\text{AI}}(M', \mathcal{C}')\| &= O(g(k)) \\
\|\Gamma_{r-\text{RI}}(M', \mathcal{C}')\| &= O(g(k)) \\
\|\Gamma_{\text{B}}(M', \mathcal{C}', M'', \mathcal{C}'')\| &= O(g(k)^2)
\end{aligned}$$

A node has at most $O(g(k) \cdot h(k))$ $\mathcal{T}$-models. Thus, computing all $\mathcal{T}$-models for a node takes time $O(g(k)^4 \cdot h(k)^2)$ in the case of a branch-node since we have to consider the left and right child of the branch node, and $O(g(k)^2 \cdot h(k))$ for any other inner node.

To check if the root node contains a root-model, we have to check if the model $\theta = (rt, \emptyset, \emptyset)$ is present. This can be done in time $O(1)$.

The size of $\mathcal{T}$ is linearly bounded by the size of R, which follows from Bodlaender (1996) [6]. Since no operation depends on $|R|$, this algorithm has the desired time bound

$$O(f(k) \cdot |R|)$$

where $f(k) = g(k)^4 \cdot h(k)^2$, and hence only depends on k.

The correctness of this algorithm immediately follows from Theorem 5.2.6, i.e., $\mathcal{AS}(R) \neq \emptyset$ holds iff there exists a root-model. □

Theorem 5.2.14 is the desired FPT result for the ASP consistency problem. Indeed, if the treewidth k is bounded by a constant, then $\mathcal{AS}(R) \neq \emptyset$ can be decided in linear time.

Example 5.2.15. *Using the algorithm given above and the tree decomposition with the dynamic tables of Figure 5.4 we solve the ASP consistency problem.*

A solution to the ASP consistency problem is immediately recognizable at the root node. According to Theorem 5.2.14, we have to find a $\mathcal{T}$-model of this form: $\theta = (n_1, \emptyset, \emptyset)$, which is present. That is, the program P of Listing 5.1 is consistent and has at least one solution.

5.2.4 *Counting Answer Sets*

The following observation is important and with Lemma 5.2.12 it lays the foundation for our counting algorithm.

Lemma 5.2.16. *For two distinct $\mathcal{T}$-interpretations $\theta_1 = (n, M_1, \mathcal{C}_1)$ and $\theta_2 = (n, M_2, \mathcal{C}_2)$, the following holds.*

$$\mathcal{E}(\theta_1) \cap \mathcal{E}(\theta_2) = \emptyset$$

Proof. Suppose to the contrary that there exists an interpretation $I \in \mathcal{E}(\theta_1) \cap \mathcal{E}(\theta_2)$, we show that $\theta_1 = \theta_2$.

By Lemma 5.2.12 $I \cap \beta(n) = A_{M_1} = A_{M_2}$ holds. Moreover, by the definition of $e_n(M_1)$, there exists a $K \subseteq A_{[n]}$ such that $\mathsf{SAT}_n(A_{M_1} \cup K) = R_{M_1} \cup R_{[n]}$, and by the definition of $e_n(M_2)$, there exists a $K \subseteq A_{[n]}$ such that $\mathsf{SAT}_n(A_{M_2} \cup K) = R_{M_2} \cup R_{[n]}$. Since $R_{M_1} \cap R_{[n]} = \emptyset$ respectively $R_{M_2} \cap R_{[n]} = \emptyset$ and $R_{M_1} \cup R_{[n]} = R_{M_2} \cup R_{[n]}$ we conclude $R_{M_1} = R_{M_2}$, and thus $M_1 = M_2$. Finally,

$$\mathcal{C}_1 = \{C \subseteq \beta(n) \mid \exists J \in \mathsf{re}_n(C, I), \text{ s.t. } J \subseteq I\} = \mathcal{C}_2$$

follows by definition. □

Next, we recursively define a mapping from $\mathcal{T}$-interpretations to numbers.

Definition 5.2.17. *Let θ be a $\mathcal{T}$-interpretation for node n. If θ is not a $\mathcal{T}$-model, let $\#(\theta) = 0$, otherwise let*

$$\#(\theta) = \begin{cases} 1 & \text{if } n \text{ is leaf node,} \\ \sum_{\theta' \prec_{\mathcal{T}} \theta} \#(\theta') & \text{if } n \text{ has one child,} \\ \sum_{(\theta',\theta'') \prec_{\mathcal{T}} \theta} \#(\theta') \cdot \#(\theta'') & \text{if } n \text{ is branch node.} \end{cases}$$

Using Theorem 5.2.6 and Lemma 5.2.12 and 5.2.16, we obtain the following Theorem.

Theorem 5.2.18. *Let θ be the root model of $\mathcal{T}$. Then, $|\mathcal{AS}(R)| = \#(\theta)$.*

Proof. We first prove $|\mathcal{E}(\theta)| = \#(\theta)$ for every model θ by induction.

Base case: It is easy to see that $|\mathcal{E}(\theta)| = \#(\theta)$ for leaf nodes, i.e., $|\{\emptyset\}| = 1$.

Induction step: Let n be a node in $\mathcal{T}$ and assume the claim holds for all nodes below n in $\mathcal{T}$.

CASE 1, n is of type (RR), (RI), (AR) or (a-AI) with $a \notin M$ where $\theta = (n, M, \mathcal{C})$. Let n' be the child node of n. By Lemma 5.2.12, we know that

$$|\mathcal{E}(\theta)| = |\bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')|.$$

Furthermore, from Lemma 5.2.16 we know that

$$|\bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')| = \sum_{\theta' \prec_{\mathcal{T}} \theta} |\mathcal{E}(\theta')|.$$

By induction hypothesis, we have $|\mathcal{E}(\theta')| = \#(\theta')$, and thus we have the desired

$$\#(\theta) = \sum_{\theta' \prec_{\mathcal{T}} \theta} \#(\theta').$$

CASE 2, n is of type (a-AI) with $a \in M$ where $\theta = (n, M, \mathcal{C})$. Let n' be the child node of n. By Lemma 5.2.12, we know that

$$|\mathcal{E}(\theta)| = |\bigcup_{\theta' \prec_{\mathcal{T}} \theta} \{I \cup \{a\} \mid I \in \mathcal{E}(\theta')\}|.$$

Due to the connectedness condition, this is the first time we see the atom a, i.e., there exists no I s.t. $I \cup \{a\} \in \mathcal{E}(\theta')$. Together with Lemma 5.2.16 we know that

$$|\bigcup_{\theta' \prec_{\mathcal{T}} \theta} \{I \cup \{a\} \mid I \in \mathcal{E}(\theta')\}| = |\bigcup_{\theta' \prec_{\mathcal{T}} \theta} \mathcal{E}(\theta')| = \sum_{\theta' \prec_{\mathcal{T}} \theta} |\mathcal{E}(\theta')|.$$

By induction hypothesis, we have $|\mathcal{E}(\theta')| = \#(\theta')$, and thus we have the desired

$$\#(\theta) = \sum_{\theta' \prec_{\mathcal{T}} \theta} \#(\theta').$$

CASE 3, n is a branch node. Let n_1 and n_2 be the child nodes of n and let θ be the $\mathcal{T}$-model of n. By Lemma 5.2.12, we know that

$$|\mathcal{E}(\theta)| = |\bigcup_{(\theta_1,\theta_2) \prec_{\mathcal{T}} \theta} \{I_1 \cup I_2 \mid I_1 \in \mathcal{E}(\theta_1), I_2 \in \mathcal{E}(\theta_2)\}|.$$

Due to the connectedness condition of a tree decomposition, we know that $A_{[n_1]} \cap A_{[n_2]} = \emptyset$, thus for each $I_1 \in \mathcal{E}(\theta_1)$ and $I_2 \in \mathcal{E}(\theta_2)$, $I_1 \cap I_2 = A_M \subseteq A_{\beta(n)}$. Further, we know that $\mathcal{E}(\theta_1)$ and $\mathcal{E}(\theta_2)$ share either exactly one model ($I_1 = I_2 = A_M \subseteq A_{\beta(n)}$), or they do not share anything at all. Together with Lemma 5.2.16, the following equation holds in both cases

$$|\mathcal{E}(\theta)| = \sum_{(\theta_1,\theta_2) \prec_{\mathcal{T}} \theta} |\{I_1 \cup I_2 \mid I_1 \in \mathcal{E}(\theta_1), I_2 \in \mathcal{E}(\theta_2)\}|.$$

By induction hypothesis we have $|\mathcal{E}(\theta_1)| = \#(\theta_1)$ and $|\mathcal{E}(\theta_2)| = \#(\theta_2)$. Hence, the following equation holds:

$$|\{I_1 \cup I_2 \mid I_1 \in \mathcal{E}(\theta_1), I_2 \in \mathcal{E}(\theta_2)\}| = |\mathcal{E}(\theta_1)| \cdot |\mathcal{E}(\theta_2)| = \#(\theta_1) \cdot \#(\theta_2).$$

Now we have the desired

$$\#(\theta) = \sum_{(\theta_1,\theta_2) \prec_{\mathcal{T}} \theta} \#(\theta_1) \cdot \#(\theta_2).$$

Thus, $|\mathcal{E}(\theta)| = \#(\theta)$ for all models θ, so the equation $|\mathcal{AS}(R)| = |\mathcal{E}(\theta)| = \#(\theta)$ holds in particular for the root model θ. □

Using the same algorithm as sketched in the proof of Theorem 5.2.14, plus keeping track of the # values for $\mathcal{T}$-models, we immediately obtain the following result.

Theorem 5.2.19. *$|\mathcal{AS}(R)|$ can be computed in time $O(f(k) \cdot |R|^2 \cdot \log |R|)$, where $k = tw(R)$ and f is a function depending on k but not on $|R|$. Unit cost for arithmetic leads to time $O(f(k) \cdot |R|)$.*

Proof. We deal with numbers as high as $O(2^{|R|})$, and thus need $O(|R|)$ bits to represent them. The time required for integer multiplication is conjectured to be optimal with $\Theta(b \cdot \log b)$, where b is the number of bits required to represent the number.

The most prominent practical integer multiplication algorithm is due to Schönhage and Strassen (1971) [73] and runs in time $O(b \cdot \log b \cdot \log\log b)$. We take integer multiplication as the most expensive operation, thus we need $O(|R| \cdot \log |R|)$ time for each $\mathcal{T}$-model. Together with the algorithm presented in proof of Theorem 5.2.14 we have a total runtime of $O(f(k) \cdot |R|^2 \cdot \log |R|)$. □

Example 5.2.20. *We will now apply this algorithm to our example in Figure 5.4. That is, after we created the dynamic programming tables, we follow the contributes relation of the $\mathcal{T}$-model of the root node (marked with $+$ and $\times$, the different marks are used for the enumeration example below).*

When we reach the leaf nodes, we start with a count of 1. This count is propagated from the leafs up since no $\mathcal{T}$-models are combined. At the branch node n_7, the count of the left and right branches is multiplied, resulting in a count of 1 for both $\mathcal{T}$-models (line 5 and 9 of node n_7). In node n_6 these two $\mathcal{T}$-models collapse into one, thus the count is added. From node n_6 up until the root node n_1 does not change, so the final count is 2, and thus we can conclude that our program of Listing 5.1 has two answer sets.

5.2.5 *Enumerating Answer Sets*

Theorem 5.2.21. *Computing $\mathcal{AS}(R)$ works in space $O(f(k) \cdot |R|)$ and outputs all elements of $\mathcal{AS}(R)$ with delay $O(f(k) \cdot |R|^2)$, where k denotes the treewidth of R and f is a function that only depends on k but not on $|R|$. Unit cost for set-operations leads to time $O(f(k) \cdot |R|)$.*

Proof. After the computation of the $\mathcal{T}$-models (takes $O(f(k) \cdot |R|)$ time and space by Theorem 5.2.14), the algorithm proceeds by traversing the tree top-down. Starting at the root-model, we follow the *contributes relation* ($\prec_{\mathcal{T}}$) until we reach the leaf-nodes. A $\mathcal{T}$-model can have more than one contributor, so the algorithm has to choose one for each iteration and node. The combination of all choices uniquely defines one interpretation. Choosing a combination of $\mathcal{T}$-models to consider can be done in time $O(|R|)$.

From the leaf-nodes upward, we collect all atoms set to true in an (a-AI) node of the currently selected $\mathcal{T}$-models. We consider a linked-list as data-structure for collecting the elements of the interpretation, so inserting an element can be done in $O(1)$.

At a branch-node, we build the union and connect the left and right list, taking time $O(1)$. Since both lists potentially contain

the atoms of the branch-node, we have to remove them first from one of the lists, taking time $O(k \cdot |R|)$.

When we reach the root node, we print all elements of the list, taking time $O(|R|)$.

Thus, traversing the tree to compute one interpretation requires $O(k \cdot |R|^2)$ steps, but only $O(|R|)$ space. □

Example 5.2.22. *Similarly to Example 5.2.20, we follow the contributes relation (shown in the second column in Figure 5.4) from the $\mathcal{T}$-model of the root node down to the leafs. Whenever we reach a $\mathcal{T}$-model with more than one "contributor", we choose one.*

Suppose that we chose the path marked with $+$ to reach the leaf nodes, for the upward direction, we collect the atoms of the (AI) nodes. That is, at the node n_6, we have $\{a, b, c\}$, at the node n_{19}, we have $\{a\}$. At the branch node, we build the union of the two, resulting in $\{a, b, c\}$. From the branch node upwards, we have no more (AI) nodes, so the first answer set is $\{a, b, c\}$.

Suppose that we chose the path marked with $\times$ for the second traversal. At the node n_6, we have $\{a, d\}$ and at the node n_{19} we have $\{a, d, e\}$. Again, at the branch node the sets are combined resulting in $\{a, d, e\}$. Since there are no more (AI) nodes until we reach the root node n_1, this is indeed our second answer set.

n_1: $\emptyset$

ID	Contr.	Ass.	Cert.
$+, \times$ 1	3	$\emptyset$	$\emptyset$
2	4,5	$\emptyset$	$\{\emptyset\}$

n_2: $\{r_4\}$

ID	Contr.	Ass.	Cert.
1	1	$\emptyset$	$\emptyset$
2	2	$\emptyset$	$\{\emptyset\}$
$+, \times$ 3	3	$\{r_4\}$	$\{\emptyset\}$
4	4	$\{r_4\}$	$\{\emptyset,\{r_4\}\}$
5	5,6	$\{r_4\}$	$\{\{r_4\}\}$

n_3: $\{f,r_4\}$

ID	Contr.	Ass.	Cert.
1	1	$\emptyset$	$\emptyset$
2	2	$\emptyset$	$\{\emptyset\}$
$+, \times$ 3	3	$\{r_4\}$	$\{\emptyset\}$
4	4	$\{r_4\}$	$\{\emptyset,\{r_4\}\}$
5	1	$\{f,r_4\}$	$\{\{r_4\}\}$
6	2,3,4	$\{f,r_4\}$	$\{\{r_4\},\{f,r_4\}\}$

n_4: $\{r_4\}$

ID	Contr.	Ass.	Cert.
1	2	$\emptyset$	$\emptyset$
2	1	$\emptyset$	$\{\emptyset\}$
$+, \times$ 3	4	$\{r_4\}$	$\{\emptyset\}$
4	3	$\{r_4\}$	$\{\emptyset,\{r_4\}\}$

n_5: $\{r_1,r_4\}$

ID	Contr.	Ass.	Cert.
1	4	$\{r_1\}$	$\{\{r_1\},\{r_4\}\}$
2	3	$\{r_1\}$	$\{\{r_4\}\}$
3	6	$\{r_1,r_4\}$	$\{\{r_1\},\{r_1,r_4\},\{r_4\}\}$
$+, \times$ 4	5	$\{r_1,r_4\}$	$\{\{r_1\},\{r_4\}\}$
5	1	$\{r_4\}$	$\emptyset$
6	2	$\{r_4\}$	$\{\{r_4\}\}$

n_6: $\{a,r_1,r_4\}$

ID	Contr.	Ass.	Cert.
1	1	$\{r_4\}$	$\emptyset$
2	2,10,11	$\{r_4\}$	$\{\{r_4\}\}$
3	3	$\{a,r_1\}$	$\{\{r_4\}\}$
4	4	$\{a,r_1\}$	$\{\{r_4\},\{a,r_1\}\}$
$+, \times$ 5	5,9	$\{a,r_1,r_4\}$	$\{\{r_4\},\{a,r_1\}\}$
6	6,7,8	$\{a,r_1,r_4\}$	$\{\{r_4\},\{a,r_1\},\{a,r_1,r_4\}\}$

n_7: $\{a,d,r_1,r_4\}$

ID	Contr.	Ass.	Cert.
1	1+1	$\{r_4\}$	$\emptyset$
2	1+2,2+1,2+2	$\{r_4\}$	$\{\{r_4\}\}$
3	3+3	$\{a,r_1\}$	$\{\{r_4\}\}$
4	3+4,4+3,4+4	$\{a,r_1\}$	$\{\{r_4\},\{a,r_1\}\}$
$+$ 5	5+3	$\{a,r_1,r_4\}$	$\{\{r_4\},\{a,r_1\}\}$
6	5+4	$\{a,r_1,r_4\}$	$\{\{r_4\},\{a,r_1\},\{a,r_1,r_4\}\}$
7	6+5	$\{a,d,r_1,r_4\}$	$\{\{r_4\},\{a,r_1\},\{a,r_1,r_4\},\{a,d,r_1,r_4\},\{d,r_4\}\}$
8	7+5	$\{a,d,r_1,r_4\}$	$\{\{r_4\},\{a,r_1\},\{a,d,r_1,r_4\},\{d,r_4\}\}$
$\times$ 9	8+5	$\{a,d,r_1,r_4\}$	$\{\{r_4\},\{a,r_1\},\{d,r_4\}\}$
10	9+6	$\{d,r_4\}$	$\{\{r_4\}\}$
11	9+7,10+6,10+7	$\{d,r_4\}$	$\{\{r_4\},\{d,r_4\}\}$

Figure 5.4: The tree decomposition with all $\mathcal{T}$-models and their relations, see Figure 5.5 and Figure 5.6 for the left and right branches of node n_7.

n_8: $\{a,d,r_1,r_4\}$

ID	Contr.	Ass.	Cert.
1	1	$\{r_4\}$	$\emptyset$
2	2	$\{r_4\}$	$\{\{r_4\}\}$
3	3	$\{a,r_1\}$	$\{\{r_4\}\}$
4	4	$\{a,r_1\}$	$\{\{r_4\},\{a,r_1\}\}$
$+$ 5	5	$\{a,r_1,r_4\}$	$\{\{r_4\},\{a,r_1\}\}$
6	6	$\{a,d,r_1,r_4\}$	$\{\{r_4\},\{a,r_1\},\{a,r_1,r_4\},\{a,d,r_1,r_4\},\{d,r_4\}\}$
7	7	$\{a,d,r_1,r_4\}$	$\{\{r_4\},\{a,r_1\},\{a,d,r_1,r_4\},\{d,r_4\}\}$
$\times$ 8	8	$\{a,d,r_1,r_4\}$	$\{\{r_4\},\{a,r_1\},\{d,r_4\}\}$
9	9	$\{d,r_4\}$	$\{\{r_4\}\}$
10	10	$\{d,r_4\}$	$\{\{r_4\},\{d,r_4\}\}$

n_9: $\{a,d,r_4\}$

ID	Contr.	Ass.	Cert.
1	1	$\{r_4\}$	$\emptyset$
2	2	$\{r_4\}$	$\{\{r_4\}\}$
3	3	$\{a\}$	$\{\{r_4\}\}$
4	4	$\{a\}$	$\{\{r_4\},\{a\}\}$
$+$ 5	5	$\{a,r_4\}$	$\{\{r_4\},\{a\}\}$
6	5	$\{a,d,r_4\}$	$\{\{r_4\},\{a\},\{a,r_4\},\{a,d,r_4\},\{d,r_4\}\}$
7	4	$\{a,d,r_4\}$	$\{\{r_4\},\{a\},\{a,d,r_4\},\{d,r_4\}\}$
$\times$ 8	3	$\{a,d,r_4\}$	$\{\{r_4\},\{a\},\{d,r_4\}\}$
9	1	$\{d,r_4\}$	$\{\{r_4\}\}$
10	2	$\{d,r_4\}$	$\{\{r_4\},\{d,r_4\}\}$

n_{10}: $\{a,r_4\}$

ID	Contr.	Ass.	Cert.
1	1	$\{r_4\}$	$\emptyset$
2	2,6,7	$\{r_4\}$	$\{\{r_4\}\}$
$\times$ 3	3	$\{a\}$	$\{\{r_4\}\}$
4	4	$\{a\}$	$\{\{r_4\},\{a\}\}$
$+$ 5	5	$\{a,r_4\}$	$\{\{r_4\},\{a\}\}$

n_{11}: $\{a,c,r_4\}$

ID	Contr.	Ass.	Cert.
1	1	$\{r_4\}$	$\emptyset$
2	2	$\{r_4\}$	$\{\{r_4\}\}$
$\times$ 3	3	$\{a\}$	$\{\{r_4\}\}$
4	4	$\{a\}$	$\{\{r_4\},\{a\}\}$
$+$ 5	5	$\{a,c,r_4\}$	$\{\{r_4\},\{a\},\{c,r_4\}\}$
6	6	$\{c,r_4\}$	$\{\{r_4\}\}$
7	7	$\{c,r_4\}$	$\{\{r_4\},\{c,r_4\}\}$

n_{12}: $\{a,c\}$

ID	Contr.	Ass.	Cert.
1	1	$\emptyset$	$\emptyset$
2	2	$\emptyset$	$\{\emptyset\}$
$\times$ 3	3	$\{a\}$	$\{\emptyset\}$
4	4	$\{a\}$	$\{\emptyset,\{a\}\}$
$+$ 5	6	$\{a,c\}$	$\{\emptyset,\{a\},\{c\}\}$
6	7	$\{c\}$	$\{\emptyset\}$
7	8	$\{c\}$	$\{\emptyset,\{c\}\}$

n_{13}: $\{a,c,r_2\}$

ID	Contr.	Ass.	Cert.
1	1	$\{r_2\}$	$\emptyset$
2	2	$\{r_2\}$	$\{\{r_2\}\}$
$\times$ 3	3	$\{a,r_2\}$	$\{\{r_2\}\}$
4	4	$\{a,r_2\}$	$\{\{r_2\},\{a,r_2\}\}$
5	3	$\{a,c\}$	$\{\{r_2\},\{a,r_2\},\{c,r_2\}\}$
$+$ 6	4	$\{a,c,r_2\}$	$\{\{r_2\},\{a,r_2\},\{a,c\},\{c,r_2\}\}$
7	1	$\{c,r_2\}$	$\{\{r_2\}\}$
8	2	$\{c,r_2\}$	$\{\{r_2\},\{c,r_2\}\}$

n_{14}: $\{a,r_2\}$

ID	Contr.	Ass.	Cert.
1	1	$\{r_2\}$	$\emptyset$
2	2	$\{r_2\}$	$\{\{r_2\}\}$
$\times$ 3	1	$\{a\}$	$\{\{r_2\}\}$
$+$ 4	2	$\{a,r_2\}$	$\{\{r_2\},\{a\}\}$

n_{15}: $\{r_2\}$

ID	Contr.	Ass.	Cert.
$\times$ 1	1	$\emptyset$	$\emptyset$
$+$ 2	2	$\{r_2\}$	$\{\emptyset\}$

n_{16}: $\{b,r_2\}$

ID	Contr.	Ass.	Cert.
$\times$ 1	1	$\emptyset$	$\emptyset$
$+$ 2	2	$\{b,r_2\}$	$\{\emptyset\}$

n_{17}: $\{b\}$

ID	Contr.	Ass.	Cert.
$\times$ 1	1	$\emptyset$	$\emptyset$
$+$ 2	1	$\{b\}$	$\{\emptyset\}$

n_{18}: $\emptyset$

ID	Contr.	Ass.	Cert.
$+,\times$ 1		$\emptyset$	$\emptyset$

Figure 5.5: Left branch of branch node n_7.

n_{19}: {a,d,r_1,r_4}

ID	Contr.	Ass.	Cert.
1	1	{r_4}	∅
2	2	{r_4}	{{r_4}}
+ 3	3	{a,r_1}	{{r_4}}
4	4	{a,r_1}	{{r_4},{a,r_1}}
× 5	5	{a,d,r_1,r_4}	{{r_4},{a,r_1},{d,r_4}}
6	6	{d,r_4}	{{r_4}}
7	7	{d,r_4}	{{r_4},{d,r_4}}

n_{20}: {a,d,r_4}

ID	Contr.	Ass.	Cert.
1	1	{r_4}	∅
2	2	{r_4}	{{r_4}}
+ 3	3	{a}	{{r_4}}
4	4	{a}	{{r_4},{a}}
× 5	5	{a,d,r_4}	{{r_4},{a},{d,r_4}}
6	6	{d,r_4}	{{r_4}}
7	7	{d,r_4}	{{r_4},{d,r_4}}

n_{21}: {a,d}

ID	Contr.	Ass.	Cert.
1	1	∅	∅
2	2	∅	{∅}
+ 3	3	{a}	{∅}
4	4	{a}	{∅,{a}}
× 5	6	{a,d}	{∅,{a},{d}}
6	7	{d}	{∅}
7	8	{d}	{∅,{d}}

n_{22}: {a,d,r_3}

ID	Contr.	Ass.	Cert.
1	1	{r_3}	∅
2	2	{r_3}	{{r_3}}
+ 3	1	{a,r_3}	{{r_3}}
4	2	{a,r_3}	{{r_3},{a,r_3}}
5	3	{a,d}	{{r_3},{a,r_3},{d,r_3}}
× 6	4	{a,d,r_3}	{{r_3},{a,r_3},{a,d},{d,r_3}}
7	3	{d,r_3}	{{r_3}}
8	4	{d,r_3}	{{r_3},{d,r_3}}

n_{23}: {d,r_3}

ID	Contr.	Ass.	Cert.
+ 1	1	{r_3}	∅
2	2	{r_3}	{{r_3}}
3	1	{d}	{{r_3}}
× 4	2	{d,r_3}	{{r_3},{d}}

n_{24}: {r_3}

ID	Contr.	Ass.	Cert.
+ 1	1	∅	∅
× 2	2	{r_3}	{∅}

n_{25}: {e,r_3}

ID	Contr.	Ass.	Cert.
+ 1	1	∅	∅
× 2	2	{e,r_3}	{∅}

n_{26}: {e}

ID	Contr.	Ass.	Cert.
+ 1	1	∅	∅
× 2	1	{e}	{∅}

n_{27}: ∅

ID	Contr.	Ass.	Cert.
+, × 1		∅	∅

Figure 5.6: Right branch of branch node n_7.

6 IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this chapter we will discuss the implementation of the ASP algorithm presented in Chapter 5. We also provide experimental results for performance measurements and benchmarks against a leading answer-set solver.

Instead of using pseudo code, we will use Haskell[1], a compiled, functional language with lazy semantics [49].

A programming language is "lazy" if it executes statements only when absolutely necessary to perform an action (print the result, for example). For the following exposition, it is critical to understand the concept of lazy evaluation. The concepts needed to follow the examples will be introduced where necessary.

Example 6.0.23. *Listing 6.1 shows a minimal Haskell program. The function* `main` *is called when starting a program from the command line, a* `do` *block is a set of sequential statements. In line 2, we set* a *to refer to an array of three elements, in line 3, we print the third element of that array. The operator* (!!) *is used to access a certain element of an array, starting at* 0*. Lazy evaluation ensures that the more expensive operations* 1000^{1000} *and* 2^{23232} *are not executed. Computations are only executed if we need the result of them, for example if line 3 would read* `print (a!!1)`*, the result of* 2^{23232} *would be calculated and printed.*

This holds also if we nest the computations. For example, if we insert
`let b = a!!1 * a!!2`
between line 2 and 3, the program does not execute any of the expensive operations, it registers only how to compute the value of `b`*, for the case that it needs the result.*

Listing 6.1: Example of lazy evaluation.

```
1 main = do
2   let a = [1000^1000,2^23232,3]
3   print (a!!2)
```

For a thorough treatment of Haskell, see "The Haskell Report" [65].

1 http://www.haskell.org

6.1 ANSWER SET PROGRAMMING SYSTEM IN HASKELL

We call our prototype LAPS (lazy answer-set programming system).

The evaluation of a disjunctive logic program is split into four steps (Figure 6.1).

1. Parse a disjunctive logic program and generate lookup tables for the formulae in our target language.
2. Build the incidence graph of the program and decompose the graph using heuristic methods [16]. The decomposition is then provided as a data structure for the target language.
3. All parts are merged with the algorithm and compiled.
4. The result is a compiled and executable version of the disjunctive logic program.

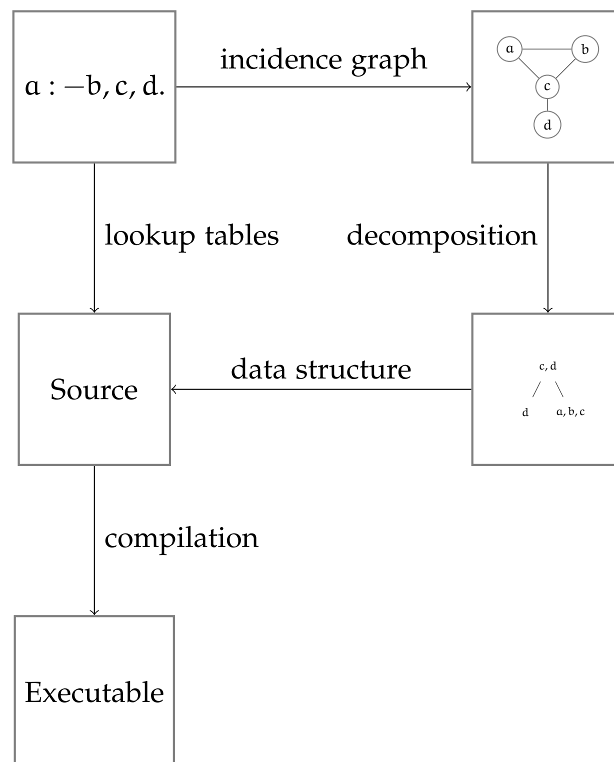


Figure 6.1: Compilation of a disjunctive logic program into an executable, as performed by LAPS.

6.1.1 *Data Structures*

We have two essential data structures within our program. There are the lookup tables for the rules, and the data structure for the tree decomposition.

Rules are stored as triples of sets: head, positive body, and negative body. Rules and atoms are encoded as simple integers, we will refer to the rule R_1 as rule 1, and the atom a_1 as atom 1.

Example 6.1.1. *The code shown in Listing 6.2 shows the definition of two rules. Rule 1 is to be read as* $R_1 : a_1 \vee a_2 \leftarrow a_3, a_4, \neg a_5, \neg a_6$*, similarly, rule 2 reads as* $R_2 : a_2 \leftarrow a_4, a_5$.

Listing 6.2: Rules in Haskell.

```
1  rule 1 = ([1,2], [3,4], [5,6])
2  rule 2 = ([2], [4,5], [])
```

The tree decomposition is a recursive data structure. Listing 6.4 shows the complete definition of it. For a better understanding, we first skim over the basic data types used throughout the program.

Listing 6.3: Basic types used in the program.

```
1  type Atom = Int
2  type Rule = Int
3  type Atoms = [Atom]
4  type Rules = [Rules]
5  type Bag = (Atoms, Rules)
6  data Theta = { assignment   :: Atoms
7               , certificates :: [Atoms]
8               , solution     :: [Atoms]
9               , count        :: Integer
10              }
```

In Listing 6.3 we define the basic types, they are mere names for the data types defined by Haskell, hence the keyword `type`. For example, a type named `Atom` is just another name for `Int` (as mentioned before, we encode atoms and rules as integers).

A bag (data type `Bag`, shown in line 5) is a tuple consisting of a set of atoms and a set of rules. We do not care for the exact position of an atom or rule within a bag, all we need to know is which atoms and which rules are present in a given bag. Since both, atoms and rules, are stored as simple integers, we keep them in separate sets to prevent mix-ups.

Finally, the data type `Theta` is an encoding of the θ used throughout Chapter 5 plus a set of solutions (answer sets) and a count. It is a named tuple consisting of a set of atoms, the assignment. Second, it has a set of sets of atoms, the certificates. The third and fourth positions are for the solutions (a set of sets of atoms) and the count of solutions. The solutions represent the function $\mathcal{E}(\cdot)$ of Chapter 5, that is, a set of sets of atoms. Remember that we do not compute the set $\mathcal{E}(\cdot)$ explicitly in Chapter 5, we do not do it here either! Due to the lazy semantics of Haskell, the contributes relation ($\prec$) is implicitly stored in the solutions (remember Example 6.1). A solution is only computed if we request one, thereby implicitly traversing the tree. The same holds for the count, represented as an `Integer`[2].

The `TreeDecomposition` data type definition in Listing 6.4 is to be read as "A tree decomposition is either a branch node, or an atom introduction node, or an atom removal node etc.". Each node defines whether it has children (of type `TreeDecomposition`), and if so, how many.

Let us start with the simplest node type for the recursive definition of the data structure of the tree decomposition, the leaf. The definition for a `LeafNode`, which is shown in line 27 of Listing 6.4, is empty since leaf nodes do not contain any children, atoms or rules.

The definitions for a rule removal, rule introduction, atom removal, and atom introduction nodes are very similar, they consist of a `Bag`, exactly one `child` which is again the root of a tree decomposition (note the recursion of the data type) and a field called `delta` whose data type varies between rule and atom nodes. The `delta` contains the element that is introduced or removed for faster lookup (it could be computed from the set difference between the bag of the child and the bag of the current node).

The `BranchNode` is defined to consist of a `Bag` and exactly two children, `child1` and `child2` of the tree decomposition type.

Listing 6.4: Data structure for the tree decomposition.

```
1 data TreeDecomposition =
2    BranchNode
3        { bag    :: Bag
4        , child1 :: TreeDecomposition
5        , child2 :: TreeDecomposition
```

2 The difference between `Int` and `Integer` is, that `Integer` is unbounded. `Int` is a machine specific type (currently 32 or 64 bit in size).

```
6         }
7     | AtomIntroductionNode
8         { bag     :: Bag
9         , child   :: TreeDecomposition
10        , delta   :: Atom
11        }
12    | AtomRemovalNode
13        { bag     :: Bag
14        , child   :: TreeDecomposition
15        , delta   :: Atom
16        }
17    | RuleIntroductionNode
18        { bag     :: Bag
19        , child   :: TreeDecomposition
20        , delta   :: Rule
21        }
22    | RuleRemovalNode
23        { bag     :: Bag
24        , child   :: TreeDecomposition
25        , delta   :: Rule
26        }
27    | LeafNode
28        {
29        }
```

Example 6.1.2. *In Listing 6.5, we have a sample encoding of the tree decomposition shown in Figure 4.3. The definition of the root node is to be read as follows. The node* `root` *is a rule removal node and has no rules, and no atoms in its bag. The child of* `root` *is node* `n1` *and rule 2 was removed.*

Listing 6.5: Some nodes of tree decomposition $\mathcal{T}_I$ encoded as Haskell program.

```
1 root = RuleRemovalNode ([],[]) n1 2
2 n1 = AtomRemovalNode ([2],[]) n2 4
3 n2 = BranchNode ([2],[4]) n3 n4
4 n3 = AtomIntroductionNode ([2],[4]) n5 4
```

The general structure for our FPT algorithms, as described in Chapter 5 is prevalent in the structure of the Haskell program. Listing 6.7 can be read like a function definition in mathematics.

To fully understand the structure shown in Listing 6.7, we have to introduce *lists* as they are provided by Haskell. The data type

`[Theta]` stands for a list of objects of the type `Theta`. Lists offer similar convenient notations as their mathematical counterpart: sets. For example, the following set

$$\{a \cdot b \mid a \in \{1..10\}, b \in \{1..10\}, a + b < 10\}$$

can be written as a Haskell function like this
`f = [ a * b | a <- [1..10], b <- [1..10], (a+b) < 10]`.
We have to note that the Haskell list is ordered and allows the same value multiple times, that is, it has not exactly set semantics, but it is sufficient for our purposes. This is only a shortcut for the long form shown in Listing 6.6.

Listing 6.6: Haskell lists.

```
1 f = do
2   a <- [1..10]
3   b <- [1..10]
4   guard ((a+b)<10)
5   return (a*b)
```

The short version is convenient if only a simple list manipulation is required, for longer variants with complex operations involved, the second form is preferred. The `guard` (line 4) deserves special attention, it ensures that only instances of a and b are considered that conform to the constraint $a + b < 10$, other combinations are not considered. The result of the function `f` is thus a list of values, but note that the ordering is not fixed in advance. Haskell lists are the representation of mathematical sets. This is the form of list processing used in Listing 6.7, and throughout the other examples.

The function `asp` is a mapping from a `TreeDecomposition` to a list of `Theta`s. We have a function definition for each node type, thus line 4 of Listing 6.7, defines the steps required for a branch node. Since each node type is dependent on the child nodes, the recursion is implicit (except for leaf nodes).

Listing 6.7: Declarative structure of the algorithm in Haskell.

```
1 asp :: TreeDecomposition -> [Theta]
2 asp (LeafNode) = do
3   ...
4 asp (BranchNode bag child1 child2) = do
5   solution_child1 <- asp child1
6   solution_child2 <- asp child2
7   ...
```

```
8  asp (AtomRemovalNode bag child a) =  do
9    solution_child <- asp child
10   ...
11 asp (RuleRemovalNode bag child c) = do
12   solution_child <- asp child
13   ...
14 asp (AtomIntroductionNode bag child a) = do
15   solution_child <- asp child
16   ...
17 asp (RuleIntroductionNode bag child c) = do
18   solution_child <- asp child
19   ...
```

The root node is the entry point for the algorithm, shown in Listing 6.8. Remember that the execution is triggered when we request the answer, in our case the `print` statements in lines 3 and 4.

Listing 6.8: Call of the `asp` algorithm with the root node as parameter.

```
1 main = do
2   let (asgn, cert, solutions, count) = asp root
3   print count
4   print solutions
```

Example 6.1.3. *In Listing 6.9 we give the definition of the function* `asp` *for a leaf node in* LAPS*. As defined by the conditions for the enumeration problem in Lemma 5.2.12, a leaf node has an empty assignment,* $\emptyset$*, no certificates,* $\emptyset$*, and a single model* $\{\emptyset\}$*.*

For the counting problem, we have only one solution at the leaf node, so the count is 1*.*

Listing 6.9: Example for a leaf node in Haskell.

```
1 asp (LeafNode) = do
2   return (Theta [] [] [[]] 1)
```

Example 6.1.4. *In Listing 6.10 we give the definition of the function* `asp` *for a rule removal node in* LAPS*.*

The removed rule is stored in `delta`*, and for each* `Theta` *from the child, we discard those tuples whose assignment,* `asgn`*, does not contain the removed rule (line 6). The remaining* `Theta`*s are transformed, so that the removed rule is deleted from the assignment (line 5). All certificates,* `cert`*, that do not contain the removed rule are also dismissed. From the*

remaining certificates the removed rule is deleted (line 3 and 4). The `models` *and the* `count` *are not altered according to Figure 5.3.*

Note that in some cases, two `Theta`*s have to be joined if their assignments and certificates are equal, this is not shown here. The models have to be joined for the enumeration problem. For the counting problem, the counts have to be summed. For the consistency problem this is not required since it carries only* `Theta`*s containing an assignment and certificates.*

Listing 6.10: Example for a rule removal node in Haskell.

```
1  asp (RuleRemovalNode bag child delta) = do
2   (Theta asgn cert models count) <- asp child
3   let temp  = filter ruleMember cert
4   let cert' = map removeRule temp
5   let asgn' = removeRule asgn
6   guard (ruleMember asgn)
7   return (Theta asgn' cert' models count)
8   where
9    removeRule (Atoms var rl)
10     = Atoms var (delete delta rl)
11   ruleMember (Atoms var rl)
12     = delta `member` rl
```

We omit the remaining mappings for the `asp` function, since they are straightforward translations of the conditions shown in Figure 5.3.

6.2 PERFORMANCE TESTS AND BENCHMARKS

This section discusses our tests using the prototype introduced in the previous section. Readily available benchmarks, for instance from the ASP system competition Gebser et al. (2007) [33] or Zhao and Lin (2003) [79], could not be used since those have a very large treewidth in general.

For a treewidth higher than six, our algorithms are not practical if we consider the worst-case scenario, but for many cases, programs with a higher treewidth were successfully solved. There is some evidence that many real world problems do have low treewidth. Thorup (1998) [77], for instance, shows that the treewidth of the control-flow graph of structured programs (more precisely, goto-free C programs) is at most six.

TEST SETUP For our tests, we generated random problem instances, and filtered those with treewidth smaller than eight. Instances with treewidth one and two were not generated. The test setup was as follows. A CNF formula with exactly three atoms was generated, this formula was then transformed into an ASP program. The number of atoms to the number of rules-ratio was fixed at three, that is, we had three times more atoms than rules, to keep the treewidth within our range. Only larger instances with hundreds of rules tended to have a treewidth higher than eight. We generated programs with increasing numbers of rules (1 to 150 rules). For each number of rules, we generated 15 instances. For each instance, we built the tree decomposition, and dismissed those with a treewidth larger than 7. The remaining instances were compiled with GHC 6.10.1[3] according to the structure shown in Figure 6.1 and executed.

The execution step was timed with Unix' `time` command. Only the CPU time was taken into account (leaving out time used for I/O and time used by other threads interrupting the execution). The programs ran on an Intel Xeon E5345 multiprocessor machine (2 CPUs, each with 4 cores) at 2.33 GHz with 4 MB cache and 48 GB of shared main-memory. Note that our implementation does not use more than one core.

Instances with a high treewidth also tend to have a higher number of models, see Figure 6.2. The plot is a standard boxplot with removed outliers. As we can see, the randomly generated programs cover a wide spectrum of possible models — from zero, that is inconsistent (or unsatisfiable) instances, to instances with more than 2.5 million models.

In the following, we analyze the runtime used by LAPS for counting and enumeration problems and draw a comparison to DLV[4] [21, 22], a state of the art ASP solver.

Figure 6.3 summarizes the runtime behavior of LAPS and DLV for counting, assuming the tree decomposition is already given, Figure 6.4 shows the data of the enumeration problem. Each graph shows the same 305 instances.

COUNTING The upper graph of Figure 6.3 shows the time required to count the number of answer sets with increasing number of answer sets for a fixed treewidth of five. LAPS' runtime is very stable, it stays within a range of half a second. It includes some variance between the individual instances. This "jitter" is

3 http://www.haskell.org/ghc
4 http://www.dlvsystem.com

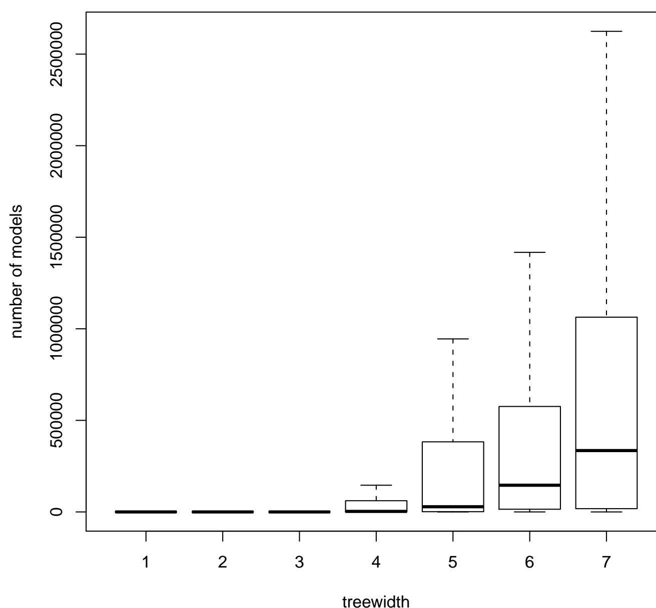


Figure 6.2: Plot showing the number of models against the treewidth (1628 instances).

caused by different shapes of the tree decompositions. It turns out, that the fewer branch nodes a tree decomposition has, the faster the algorithm can compute the answer. This is in line with the theoretical results given in Chapter 5. The time to handle very large integers is not visible for these instances. Since DLV, shown in the lower graph, has no special support for counting, its runtime depends directly on the number of answer sets. Note that the scales of the graphs are different.

ENUMERATION The graphs in Figure 6.4 show the time required to enumerate the number of answer sets with increasing number of answer sets for a fixed treewidth of five. The time is normalized to the time per 100 answer sets. The enumeration time quickly approaches almost zero for larger instances. For the very first instances, the cost for producing the dynamic tables can be seen (up to four seconds for 100 answer sets in LAPS' case), yet this "setup cost" is quickly amortized by the easy model generation. DLV does not require a noteworthy preprocessing time.

INFLUENCE OF THE TREEWIDTH The last set of graphs shown in Figure 6.5 shows the runtime for counting, respectively enumerating, answer sets by LAPS plotted against the treewidth (1628 instances). We can see the exponential increase in runtime with the treewidth. This phenomenon is best seen in the counting case (upper graph). The increasing number of possible models reduces the effect at a higher treewidth in the graph that shows the timings for enumeration of 100 answer sets. For a lower treewidth, that is, three or four, the setup cost is still visible.

DLV is not affected by the treewidth, so the graph is omitted.

Tests using `smodels`[5] ([74]) instead of DLV resulted in a very similar runtime behavior as DLV.

Depending on the requirements, we conclude that for our prototype implementation a treewidth of six is currently the highest practical treewidth to work with. For the counting problem, there may be cases where a higher treewidth is still manageable. For the enumeration problem, if only some answer sets are required, LAPS is still a good choice, especially with the proved linear delay between two solutions. If all, or most, answer sets are needed, other approaches are preferable if the treewidth is larger than six.

5 `http://www.tcs.hut.fi/Software/smodels/index.html`

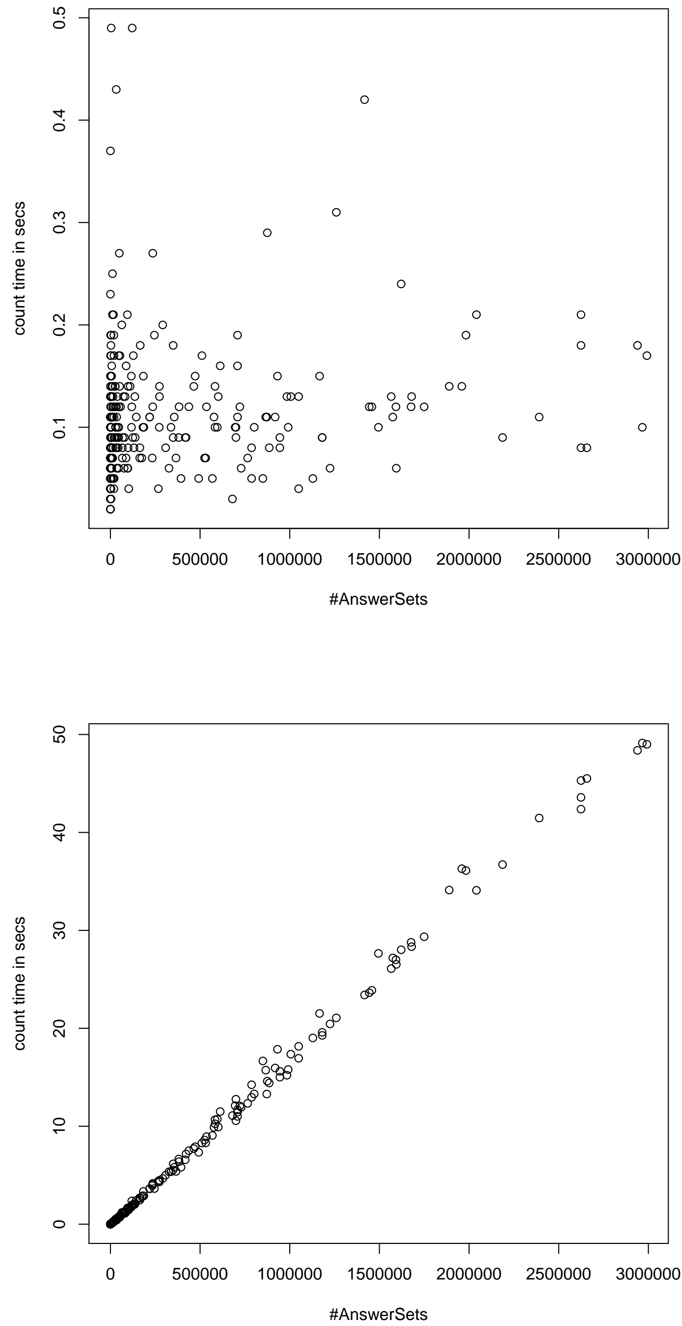
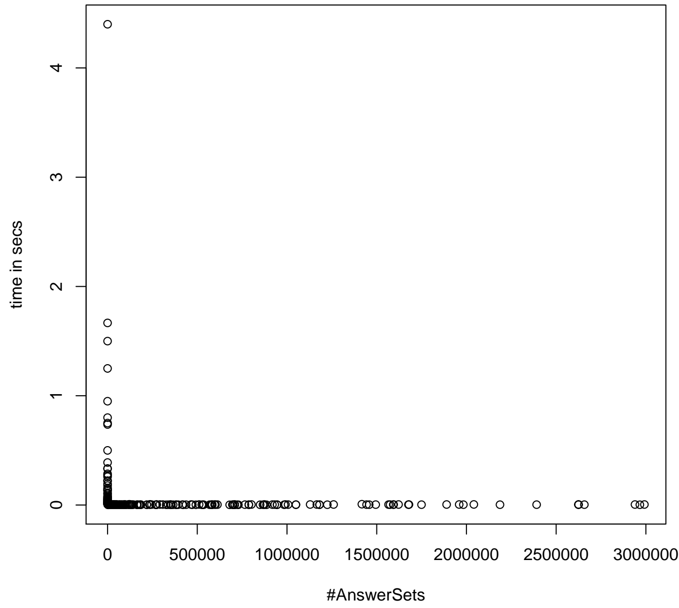


Figure 6.3: Time required by LAPS (upper graph) and DLV (lower graph) for counting of all answer sets (305 instances).

Enumeration



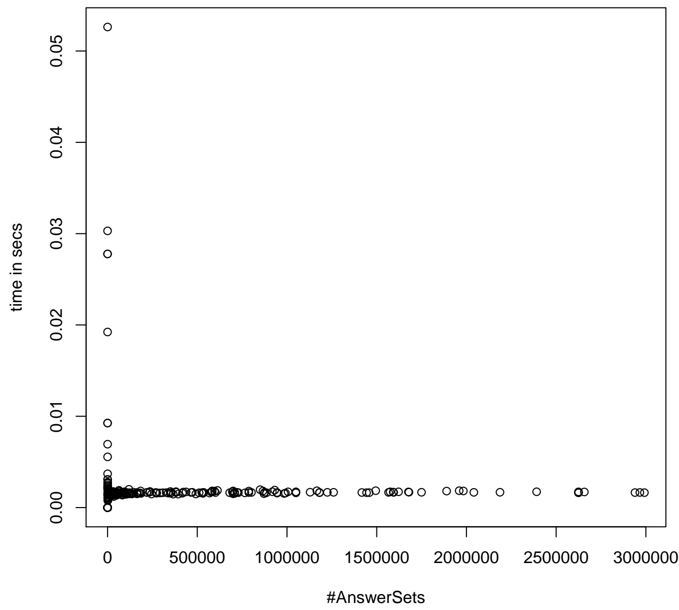


Figure 6.4: Time required by LAPS (upper graph) and DLV (lower graph) for enumeration of all answer sets—normalized to time per 100 answer sets (305 instances).

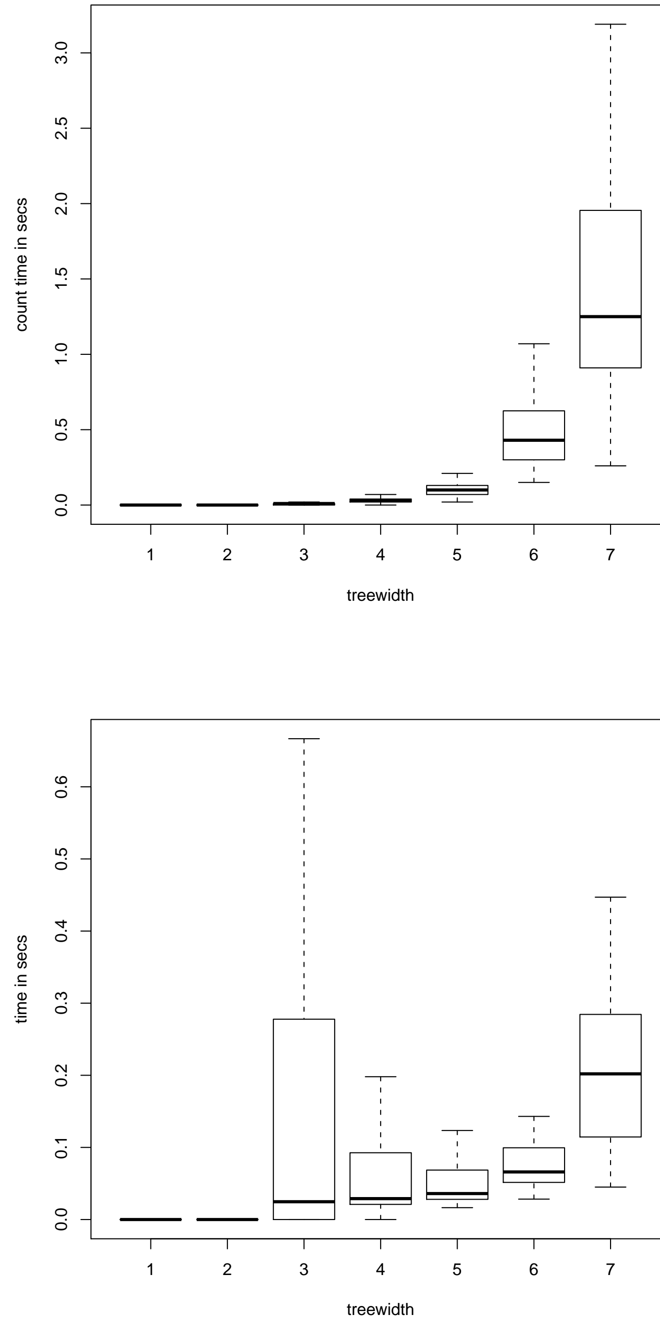


Figure 6.5: Comparison of the runtime behavior for counting (upper graph) and enumeration of 100 answer sets (lower graph) plotted against the treewidth (1628 instances).

7 CONCLUSION

To summarize, we introduced novel algorithms for the decision problem of ASP as well as for counting and enumerating answer sets. We have shown a general framework for fixed parameter tractable problems using treewidth as parameter, and have reported on a prototype implementation.

Our experiments show unprecedented performance on the counting problem, and very competitive performance on the enumeration problem for a low treewidth (up to six). For the ASP consistency problem, the preprocessing required by our algorithms is too high compared to state of the art solvers.

On the theoretical side we proved the correctness of our algorithms and several time bounds. The decision and the counting problems are proved to be fixed parameter linear, and the enumeration problem has been show to have linear delay.

Taking the time required to build the tree decomposition into account, our approach is not yet "production ready". The ASP algorithms run sufficiently fast on a given tree decomposition, but the time required to build a tree decomposition is currently the main obstacle in our approach.

RELATED WORK The work most closely related to ours is by Samer and Szeider (2007) [70], where the #SAT problem in case of bounded treewidth was solved by dynamic programming. Compared to the ASP problems, #SAT is #P complete, compared to the harder #NP problems of ASP. The difference becomes visible by comparing the data being propagated up the tree structure as shown in Chapter 4.

Two related problems are constraint satisfaction problems (CSP) and conjunctive query (CQ) evaluation, for which, apart from treewidth, further methods based on structural decomposition have been used to construct efficient algorithms by Gottlob et al. (1999) and Chekuri and Rajaraman (1998) [39, 10]. These methods also work with a bottom-up traversal of a tree structure. As with #SAT, the data propagated up the tree structure is much simpler than in case of ASP solving. The idea of post-processing by a top-down traversal in order to compute all solutions is also present in the context of CQ evaluation.

Another FPT result for ASP is due to Lin and Zhao (2004) [55], who use the number of cycles in the (directed) dependency graph as parameter. An interesting parameter in this context is the number of loops of a program, as proposed by Ferraris, Lee, and Lifschitz (2006) [28].

Recently, dynamic programming has also been applied to logic programming in the context of query answering over Semantic-Web data by Ruckhaus et al. (2008) [69]. In their work, dynamic programming is applied to the computation of an optimal join order for CQ evaluation over deductive databases.

FUTURE WORK Our prototype implementation has its limit at treewidth six, a fine tuned implementation using specially crafted data structures could increase that bound. The data stored at each node of the tree decomposition, for example, differs only slightly to that of the child node, this could be exploited.

Another possible way to improve the presented prototype, is to distribute the computation to many processors, or even many computers connected by a high speed network. The branch nodes split the computation into two independent sub-trees, a parallel implementation could use that fact and speed the computation up by computing each branch on its own processor. Another way of exploiting parallelism is at the tree traversal step, instead of computing one solution after the other, several processors could compute the solutions with a small coordination overhead simultaneously.

Propositional disjunctive logic programs are the foundation of ASP, but there are many more aspects to be explored in the context of ASP and bounded treewidth. The grounding step, described in Section 2.3, could be extended towards the use of structures of bounded treewidth. For example by building the tree decomposition for small treewidth and if successful solve the grounded program using our new algorithm, if not successful, solve it by the standard algorithms. Another interesting approach would be to build the tree decomposition of the ungrounded program to see if this early grounding leads to faster decomposition times.

Since the performance of our algorithms depends on finding a tree decomposition of the smallest possible width, our approach will directly benefit from future progress in the research for more efficient tree decomposition algorithms. Especially further research in fast and optimal tree decomposition methods for small treewidth would greatly enhance the usefulness of the presented FPT algorithms.

Finally, as the SAT competitions lead to comparability between SAT solvers and in effect to high performance SAT solvers, we would like to see similar competitions in the context of treewidth, or general fixed parameter algorithms. Current platforms, like Asparagus[1] aim at general solvers, but more competition for special case solvers could be a catalyst for new ideas and further improvements in this highly promising approach.

1 http://asparagus.cs.uni-potsdam.de

BIBLIOGRAPHY

[1] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, 1988. (Cited on page 27.)

[2] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987. ISSN 0196-5212. doi: http://dx.doi.org/10.1137/0608024. (Cited on page 37.)

[3] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12 (2):308–340, 1991. (Cited on pages 17 and 51.)

[4] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA, 2003. ISBN 0521818028. (Cited on pages 17 and 26.)

[5] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11(1-2):1–22, 1993. (Cited on page 37.)

[6] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996. (Cited on pages 45, 47, and 106.)

[7] Hans L. Bodlaender and Ton Kloks. Better algorithms for the pathwidth and treewidth of graphs. In *Proceedings of the 18th International Colloquium on Automata, Languages and Programming*, pages 544–555, New York, NY, USA, 1991. Springer-Verlag New York, Inc. ISBN 0-387-54233-7. (Cited on page 47.)

[8] Hans L. Bodlaender, John R. Gilbert, Hjlmtr Hafsteinsson, and Ton Kloks. Approximating treewidth, pathwidth, and minimum elimination tree height. *Journal of Algorithms*, 18: 238–255, 1991. (Cited on page 47.)

[9] Julius Richard Büchi. Weak second order logic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 5:66–92, 1960. (Cited on page 36.)

[10] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2): 211–229, 1998. (Cited on page 129.)

[11] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, New York, NY, USA, 1971. ACM. doi: http://doi.acm.org/10.1145/800157.805047. (Cited on page 25.)

[12] Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In *Handbook of Theoretical Computer Science, Volume B*, pages 193–242. Elsevier Science Publishers, 1990. (Cited on pages 17 and 51.)

[13] Bruno Courcelle, Johann A. Makowsky, and Udi Rotics. On the fixed parameter complexity of graph enumeration problems definable in monadic second order logic. *Discrete Applied Mathematics*, 108:23–52, 2002. (Cited on page 51.)

[14] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. GASP: Answer set programming with lazy grounding. *Fundamenta Informaticae*, 96(3):297–322, 2009. ISSN 0169-2968. (Cited on page 31.)

[15] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/502807.502810. (Cited on page 27.)

[16] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Ben Mcmahan, Nysret Musliu, and Marko Samer. Heuristic methods for hypertree decomposition. In *MICAI '08: Proceedings of the 7th Mexican International Conference on Artificial Intelligence*, pages 1–11, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-88635-8. doi: http://dx.doi.org/10.1007/978-3-540-88636-5_1. (Cited on pages 49 and 116.)

[17] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer, New York, 1999. (Cited on pages 17 and 51.)

[18] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite Model Theory, 2nd edition*. Springer Monographs in Mathematics. Springer, 1999. (Cited on page 33.)

[19] Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15(3/4):289–323, 1995. (Cited on pages 17, 27, and 31.)

[20] Thomas Eiter and Georg Gottlob. Expressiveness of stable model semantics for disjunctive logic programs with functions. *The Journal of Logic Programming*, 33(2):167 – 178, 1997. ISSN 0743-1066. doi: DOI:10.1016/S0743-1066(97)00027-7. URL `http://www.sciencedirect.com/science/article/B6V0J-3SP2DBV-6/2/655191b4c2f666d9007415eed4cc0948`. (Cited on page 28.)

[21] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The architecture of a disjunctive deductive database system. In *Proceedings of the Fourth International Conference on Declarative Programming (APPIA-GULP-PRODE 1997)*, pages 141–152, 1997. (Cited on page 123.)

[22] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR system dlv: Progress report, comparisons and benchmarks. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR-98)*, pages 406–417, 1998. (Cited on page 123.)

[23] Thomas Eiter, Wolfgang Faber, Michael Fink, and Stefan Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *Annals of Mathematics and Artificial Intelligence*, 51(2-4):123–165, 2007. ISSN 1012-2443. doi: http://dx.doi.org/10.1007/s10472-008-9086-5. (Cited on page 31.)

[24] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate A. Schmidt, editors, *5th International Reasoning Web Summer School (RW 2009), Brixen/Bressanone, Italy, August 30–September 4, 2009*, volume 5689 of *LNCS*, pages 40–110. Springer, September 2009. ISBN 978-3-642-03753-5. doi:

10.1007/978-3-642-03754-2_2. URL `http://www.kr.tuwien.ac.at/staff/tkren/pub/2009/rw2009-asp.pdf`. (Cited on page 31.)

[25] Calvin C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98(1):21–51, 1961. (Cited on page 36.)

[26] Michael R. Fellows. New directions and new challenges in algorithm design and complexity, parameterized. *Lecture Notes in Computer Science*, 12:505–520, 2003. URL `http://www.springerlink.com/index/Y0MCD383U8J2JF7K.pdf`. (Cited on pages 47, 50, and 51.)

[27] Michael R. Fellows, Jens Gramm, and Rolf Niedermeier. On the parameterized intractability of motif search problems. *Combinatorica*, 26(2):141–167, 2006. ISSN 0209-9683. doi: http://dx.doi.org/10.1007/s00493-006-0011-4. (Cited on page 50.)

[28] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. A generalization of the lin-zhao theorem. *Annals of Mathematics and Artificial Intelligence*, 47(1-2):79–101, 2006. (Cited on page 130.)

[29] Jörg Flum and Martin Grohe. Describing parameterized complexity classes. *Information and Computation*, 187(2):291–319, 2003. ISSN 0890-5401. doi: http://dx.doi.org/10.1016/S0890-5401(03)00161-5. (Cited on page 51.)

[30] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree decompositions. *Journal of the ACM*, 49(6):716–752, 2002. (Cited on pages 17, 51, 52, and 55.)

[31] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Annals of Pure and Applied Logic*, 130(1-3):3–31, 2004. URL `http://linkinghub.elsevier.com/retrieve/pii/S0168007204000612`. (Cited on pages 18 and 53.)

[32] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447. (Cited on page 22.)

[33] Martin Gebser, Lengning Liu, Gayathri Namasivayam, Andre Neumann, Torsten Schaub, and Mirosław Truszczyński.

The first answer set programming system competition. In C. Baral, G. Brewka, and J. Schlipf, editors, *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007)*, volume 4483 of *LNCS*, pages 3–17. Springer, 2007. (Cited on pages 18 and 122.)

[34] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. Engineering an incremental ASP solver. In *ICLP '08: Proceedings of the 24th International Conference on Logic Programming*, pages 190–205, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89981-5. doi: http://dx.doi.org/10.1007/978-3-540-89982-2_23. (Cited on page 31.)

[35] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, (38), 1991. (Cited on page 27.)

[36] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings Fifth International Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press. (Cited on page 29.)

[37] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991. (Cited on page 27.)

[38] Rob Glabbeek and Bas Ploeger. Five determinisation algorithms. In *CIAA '08: Proceedings of the 13th International Conference on Implementation and Applications of Automata*, pages 161–170, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70843-8. doi: http://dx.doi.org/10.1007/978-3-540-70844-5_17. (Cited on page 35.)

[39] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, 1999. (Cited on page 129.)

[40] Georg Gottlob, Reinhard Pichler, and Fang Wei. Bounded treewidth as a key to tractability of knowledge representation and reasoning. In *Proceedings of the 21st National Conference on Artificial Intelligence and the 18th Conference on Innovative Applications of Artificial Intelligence*, pages 250–256. AAAI Press, 2006. (Cited on pages 17, 52, and 55.)

[41] Georg Gottlob, Reinhard Pichler, and Fang Wei. Monadic datalog over finite structures with bounded treewidth. In *PODS '07: Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of database systems*, pages 165–174, 2007. (Cited on page 52.)

[42] Martin Grohe. Descriptive and parameterized complexity. In *CSL '99: Proceedings of the 13th International Workshop and Eighth Annual Conference of the EACSL on Computer Science Logic*, pages 14–31, London, UK, 1999. Springer-Verlag. ISBN 3-540-66536-6. (Cited on page 18.)

[43] Thomas Hammerl. *Ant Colony Optimization for Tree and Hypertree Decompositions*. Master's thesis, TU Wien, 2009. (Cited on page 49.)

[44] Fritz Henglein and Harry G. Mairson. The complexity of type inference for higher-order lambda calculi. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–130, New York, NY, USA, 1991. ACM. ISBN 0-89791-419-8. doi: http://doi.acm.org/10.1145/99583.99602. (Cited on page 50.)

[45] Petr Hlinený, Sang-il Oum, Detlef Seese, and Georg Gottlob. Width parameters beyond tree-width and their applications. *The Computer Journal*, 51(3):326–362, 2007. ISSN 0010-4620. doi: 10.1093/comjnl/bxm052. URL `http://comjnl.oxfordjournals.org/cgi/doi/10.1093/comjnl/bxm052`. (Cited on page 50.)

[46] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, New York, NY, USA, 2004. ISBN 052154310X. (Cited on page 24.)

[47] Michael Jakl, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. Fast counting with bounded treewidth. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 5330 of *LNCS*, pages 436–450. Springer, 2008. ISBN 978-3-540-89438-4. (Cited on pages 7 and 55.)

[48] Michael Jakl, Reinhard Pichler, and Stefan Woltran. Answer-set programming with bounded treewidth. In *IJCAI'09:*

Proceedings of the 21st International Joint Conference on Artifical Intelligence, pages 816–822, 2009. (Cited on page 7.)

[49] Mark B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science*, 68(1):105–111, 1989. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(89)90122-9. (Cited on page 115.)

[50] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994. ISBN 3-540-58356-4. (Cited on page 40.)

[51] Hendrik W. Lenstra Jr. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8: 538–548, 1983. (Cited on page 50.)

[52] Leonid Levin. Universal search problems. In *Problems of Information Transmission*, volume 9, pages 115–116, 1973. (Cited on page 25.)

[53] Leonid Libkin. *Elements Of Finite Model Theory (Texts in Theoretical Computer Science. An Eatcs Series)*. SpringerVerlag, 2004. ISBN 3540212027. (Cited on pages 33 and 52.)

[54] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 97–107, New York, NY, USA, 1985. ACM. ISBN 0-89791-147-4. doi: http://doi.acm.org/10.1145/318593.318622. (Cited on page 50.)

[55] Fangzhen Lin and Xishun Zhao. On odd and even cycles in normal logic programs. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the 19th National Conference on Artificial Intelligence and the 17th Conference on Innovative Applications of Artificial Intelligence*, pages 80–85. AAAI Press / The MIT Press, 2004. ISBN 0-262-51183-5. (Cited on page 130.)

[56] Victor W. Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In K. R. Apt, V. W. Marek, M. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. Springer Verlag, 1999. (Cited on pages 17, 26, and 29.)

[57] Jirí Matousek and Robin Thomas. Algorithms finding tree-decompositions of graphs. *Journal of Algorithms*, 12(1):1–22, 1991. (Cited on page 47.)

[58] Oliver Matz and Wolfgang Thomas. The monadic quantifier alternation hierarchy over graphs is infinite. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, pages 236–244. IEEE, 1997. (Cited on page 33.)

[59] John McCarthy. Circumscription - a form of non-monotonic reasoning. *Artificial Intelligence*, 13(1-2):27–39, 1980. (Cited on page 26.)

[60] Jack Minker. Overview of disjunctive logic programming. *Annals of Mathematics and Artificial Intelligence*, 12:1–24, 1994. (Cited on page 26.)

[61] Jack Minker. Logic and databases: A 20 year retrospective. In *Proceedings of the International Workshop on Logic in Databases (LID'96)*, volume 1154 of *LNCS*, pages 3–57. Springer, 1996. (Cited on page 29.)

[62] Rolf Niedermeier. *Invitation to Fixed Parameter Algorithms (Oxford Lecture Series in Mathematics and Its Applications)*. Oxford University Press, USA, March 2006. ISBN 0198566077. (Cited on page 51.)

[63] Ilkka Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999. (Cited on pages 17, 26, and 29.)

[64] Ilkka Niemelä. Language extensions and software engineering for ASP. Technical Report WP3, Working Group on Answer Set Programming (WASP), IST-FET-2001-37004, September 2005. Available at `http://www.tcs.hut.fi/Research/Logic/wasp/wp3/wasp-wp3-web/`. (Cited on page 27.)

[65] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`. (Cited on page 115.)

[66] Bruce A. Reed. Finding approximate separators and computing tree width quickly. In *STOC '92: Proceedings of the 24th*

Annual ACM Symposium on Theory of Computing, pages 221–228, New York, NY, USA, 1992. ACM. ISBN 0-89791-511-9. doi: http://doi.acm.org/10.1145/129712.129734. (Cited on page 47.)

[67] Neil Robertson and Paul D. Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36 (1):49–64, 1984. (Cited on page 37.)

[68] Donald J. Rose and Robert E. Tarjan. Algorithmic aspects of vertex elimination. In *Proceedings of Seventh Annual ACM Symposium on Theory of Computing*, page 254. ACM, 1975. URL `http://portal.acm.org/citation.cfm?id=803775`. (Cited on pages 15, 47, and 48.)

[69] Edna Ruckhaus, Eduardo Ruiz, and Maria-Esther Vidal. Query evaluation and optimization in the semantic web. *Theory and Practice of Logic Programming*, 8(3):393–409, 2008. (Cited on page 130.)

[70] Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8 (1):50–64, 2010. ISSN 1570-8667. doi: http://dx.doi.org/10.1016/j.jda.2009.06.002. (Cited on pages 18, 56, 61, 62, and 129.)

[71] Daniel P. Sanders. On linear recognition of tree-width at most four. *SIAM Journal on Discrete Mathematics*, 9(1):101–117, 1996. ISSN 0895-4801. doi: http://dx.doi.org/10.1137/S0895480193243043. (Cited on page 47.)

[72] Werner Schafhauser. *New Heuristic Methods for Tree Decompositions and Generalized Hypertree*. Master's thesis, TU Wien, 2006. (Cited on page 49.)

[73] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. (German) [Fast multiplication of large numbers]. *Computing*, 7(3–4):281–292, 1971. ISSN 0010-485X (printed version), 1436-5057 (electronic version). (Cited on page 110.)

[74] Patrik Simons. Extending and implementing the stable model semantics. Research Report A58, Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory for Theoretical Computer Science, Espoo, Finland, April 2000. Doctoral dissertation. (Cited on page 125.)

[75] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, June 2002. (Cited on page 29.)

[76] James W. Thatcher and Jesse B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1): 57–81, 1968. (Cited on page 36.)

[77] Mikkel Thorup. All structured programs have small tree-width and good register allocation. *Information and Computation*, 142(2):159–181, 1998. (Cited on page 122.)

[78] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976. (Cited on page 26.)

[79] Yuting Zhao and Fangzhen Lin. Answer set programming phase transition: A study on randomly generated programs. In *ICLP 2003: Proceedings of the 19th International Conference on Logic Programming*, pages 239–253, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-20642-6. (Cited on page 122.)

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, Mai 2010

Michael Jakl

Michael Jakl
Fasangasse 10/10
A-1030 Wien

Persönliche Daten

Geburtsdatum 02. Mai 1981

Staatsbürgerschaft Österreich

Wehrdienst abgeleistet

Sprachen Deutsch (Muttersprache), Englisch (fließend)

Ausbildung

TU Wien Okt. 2002 – Juni 2010

Doktoratsstudium 7 Semester Doktoratsstudium der technischen Wissenschaften. Voraussichtlicher Abschluss: Juni 2010. Titel der Doktorarbeit: *Answer Set Programming with Bounded Treewidth.* *Feb. 2007 – Juni 2010*

Magisterstudium 3 Semester Magisterstudium Software Engineering & Internet Computing am 09. Jänner 2007 mit Auszeichnung abgeschlossen.
Titel der Diplomarbeit: *Efficient Algorithms through Bounded Treewidth.* *Sept. 2005 – Jan 2007*

Bakkalaureat 6 Semester Bakkalaureat Software & Information Engineering am 07. Juli 2005 abgeschlossen.
Titel der Bakkalaureatsarbeit: *Representational State Transfer.* *Okt.* 2002 – Juli 2005

HTL Villach 5 Jahre Höhere Technische Bundeslehr- und Versuchsanstalt für elektronische Datenverarbeitung und Organisation in Villach. Reifeprüfung am 04. Juni 2000 mit Auszeichnung bestanden. Sept. 1995 – Juni 2000

BERUFSERFAHRUNG

TU Wien

Universitäts-Assistent am Institut für Informationssysteme, Arbeitsgruppe für Datenbanken und Artificial Intelligence. Neben Lehrtätigkeiten für 400-500 Studenten pro Semester in den Themenbereichen
- *Datenbanken* (SQL3, PL/SQL, JDBC) und
- *Semistrukturierte Daten (XML, XSLT, Java+XML),*

war das Entwickeln und Umsetzen diverser Algorithmen in Java, Haskell und Datalog ein zentrales Thema.
Verwendete Datenbanken: Oracle, PostgreSQL
März 2007 – Juni 2010

Knallgrau

Freier Mitarbeiter als Entwickler. Ziel war die Umsetzung eines um semantische Aspekte erweitertes Dokumenten- und Informationsmanagement Systemes.
Verwendete Technologien: Java, JavaScript (Helma), MySQL
Juli 2005 – Sept. 2006

ho.bit Ltd.

Freier Mitarbeiter als Consultant und Entwickler. Ziel war die Umsetzung einer sicheren Netzwerkinfrastruktur basierend auf einer Public-Key Infrastuktur und Browser-Zertifikaten.
Verwendete Technologien: Java, LDAP, RSA/DSA
März 2004 – Nov. 2004

AppCom

Freier Mitarbeiter als Entwickler. Entwicklung von eines Frameworks zur vereinfachten Erstellung von Webapplikationen in Java.
Weitere verwendete Technologien: PostgreSQL, Hibernate, StrutsCX
Okt. 2002 – Okt. 2003

Net4You

Angestellter als Entwickler. Projektverantwortlicher (Konzeption und Umsetzung) für ein Bestellsystem der hogast-Betriebe sowie ein Planungssystem für das Casino Velden. Weitere Aufgabengebiete war die Wartung diverser Linux Systeme und Tooling der Software-Entwicklung.
Verwendete Technologien: Java, CVS, Firebird Datenbank
Mai 2001 – Sept. 2002

Gewerbeschein

Erwerb eines Gewerbescheines für Dienstleistungen in der EDV und IT neben dem Studium.
Jan. 2001 – Sept. 2007