FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

dynASP
A Dynamic Programming-based Answer Set Programming Solver

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Michael Morak
Matrikelnummer 0627699

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Privatdoz. Dipl.-Ing. Dr.techn. Stefan Woltran
Mitwirkung: Proj.Ass. Dipl.-Ing. Stefan Rümmele

Wien, 31.01.2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

MASTERS THESIS

dynASP

A Dynamic Programming-based Answer Set Programming Solver

Author:
Michael MORAK

Supervisor:
Dr. Stefan WOLTRAN

January 31, 2011

Abstract

Answer Set Programming (ASP) is nowadays a well known and well acknowledged paradigm for declarative problem solving as can be seen by many successful applications in the areas of artificial intelligence and knowledge-based reasoning. Evaluating logic programs that follow the ASP paradigm is usually implemented as a two-step procedure: First, the grounding step (i.e. the instantiation of variables occurring in the program) and second, the actual solving process which works on ground logic programs. In this thesis we introduce a novel approach for dealing with the latter.

Solving ground logic programs is, in general, still an intractable problem. Therefore most standard ASP solvers rely on techniques originating in SAT solving or constraint satisfaction problem solving. In contrast, the algorithm presented in this thesis was developed with techniques that stem from parameterized complexity theory. The idea here is to consider a certain problem parameter as bounded by a constant and thereby obtain a tractability result for the given problem. One such parameter which has lead to many interesting results is treewidth which represents the "tree-likenes" of a graph. Treewidth is defined in terms of tree decompositions which in turn can be used by dynamic programming algorithms to solve the problem under consideration.

We introduce a novel dynamic programming algorithm based on the above approach that is specifically tailored for solving head-cycle free logic programs. Using a purpose-built framework for algorithms on tree decompositions, an actual implementation of the algorithm is presented, carefully evaluated and compared to existing ASP solvers. Experimental results show significant potential for problem instances of low treewidth.

Kurzfassung

Answer Set Programming (ASP) ist heutzutage ein bekanntes und etabliertes Paradigma für deklarative Wissensverarbeitung und wurde bereits in mehreren Gebieten (z.B. im Bereich der künstliche Intelligenz oder der wissensbasierten Systeme) erfolgreich eingesetzt. Grundsätzlich ist die Auswertung von logischen Programmen ein zweistufiges Verfahren: In einem ersten Schritt werden alle Regeln des Programmes grundiert (d.h. falls Variablen in diesen Regeln vorhanden sind, werden Sie durch konkrete Werte ersetzt). Erst im zweiten Schritt erfolgt dann die eigentliche Auswertung des Programmes, da Algorithmen für diesen Schritt nur auf grundierten Programmen arbeiten. In dieser Arbeit stellen wir einen neuen Ansatz für letzteren Schritt vor.

Das Auswerten von Logikprogrammen ist grundsätzlich eine aufwendige Aufgabe. Für ähnlich schwere Probleme im Bereich des SAT-Solvings bzw. im Bereich von Constraint Satisfaction wurden bereits erfolgreich effiziente Algorithmen gefunden, weswegen die meisten modernen Algorithmen für ASP auf Techniken aus diesem Bereich aufbauen. Im Gegensatz dazu setzt der in dieser Arbeit präsentierte Algorithmus auf ein neues Prinzip, das auf Ergebnissen der parametrisierten Komplexitätstheorie basiert. Hierbei wird ein bestimmter Problemparameter fixiert, wodurch das Problem im Allgemeinen leichter lösbar wird. Ein solcher Parameter ist die sg. “Treewidth”, die, grob gesprochen, die “Baumähnlichkeit” eines Graphen beschreibt. Sie wird mit Hilfe von Tree Decompositions definiert, auf welchen der hier vorgestellte Algorithmus aufbaut.

Dieser Algorithmus, der auf obigem Prinzip und dynamischer Programmierung basiert, ist speziell auf die Klasse der sg. “head-cycle-freien” logischen Programme zugeschnitten. Mit Hilfe eines eigens entwickelten Frameworks für Algorithmen und Tree Decompositions wurde der Algorithmus implementiert und anschließend ausführlich getestet. Experimentelle Ergebnisse zeigen großes Potential für Probleme mit kleinen Parameterwerten (d.h. mit kleiner Treewidth).

Michael Morak
Turracherstraße 31
9560 Feldkirchen

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

_____

(Datum, Ort, Unterschrift)

Contents

I Introduction

1 Introduction

Answer Set Programming (ASP) [Marek and Truszczyński, 1999, Niemelä, 1999] today is a well known and well acknowledged paradigm for declarative problem solving in the scientific community as can be seen by many successful applications in the areas of artificial intelligence [Novák, 2009, Köster et al., 2009] and knowledge-based reasoning (cf. for instance [Grasso et al., 2009]). Evaluating logic programs that follow the ASP paradigm is usually implemented as a two-step procedure: First, the grounding step (i.e. the instantiation of variables occurring in the program) and second, the actual solving process itself which works on ground (that is, propositional) logic programs. For the latter task, many different solvers exist today, see for instance [Lin and Zhao, 2004, Leone et al., 2006, Gebser et al., 2007]. In this thesis we introduce a novel approach for dealing with the second step of this process.

The idea of answer set programming in particular and logic programming in general is to follow a declarative programming style. Contrary to imperative programming languages such as C, C++, Java etc., that deal with states and sequences of operations to change them, a declarative programming language tries to describe the problem in a more flexible way.

Example 1.1. *The following example encodes knowledge about birds in an imperative style:*

```
boolean bird(x)
{
    if(x == "tweety") return true;
    if(penguin(x)) return true;
    return false;
}
```

The corresponding logic program would look like this:

$bird(tweety)$.
$bird(X) \leftarrow penguin(X)$.

The imperative procedure represents the knowledge in a rather static way, as in order to extend it with new species of birds or new individual birds would require the addition

of many if-else constructs in the program which, for more complicated examples, can easily get confusing. On the other hand the logic programming approach can simply be extended by adding additional facts or rules to the program.

Answer set programming as a paradigm was defined in the late 1980s and early 1990s with stable model semantics that were first defined for normal logic programs [Gelfond and Lifschitz, 1988] and later on extended also to disjunctive logic programs [Gelfond and Lifschitz, 1991]. In this work we use answer set semantics synonymously with stable model semantics and only deal with programs containing negation-as-failure but no strong negation. Negation-as-failure is closely related to Reiter's *Default Logic* [Reiter, 1980] and hence also often called *default negation*. An overview of the different semantics for logic programs can e.g. be found in [Dix, 1995].

Deciding consistency for ground logic programs under answer set semantics is, in general, an intractable problem. For disjunctive logic programs, decision problems reside on the second level of the polynomial hierarchy (i.e. being complete for Σ_P^2, see [Eiter and Gottlob, 1995]), and even for disjunction-free logic programs (usually called normal logic programs) these decision problems are generally still NP-complete [Marek and Truszczynski, 1991]. If certain restrictions are placed on the use of disjunctions (that is, to head-cycle free disjunctive logic programs) the complexity can be lowered from the second level of the polynomial hierarchy to the same level as that of normal logic programs as shown in [Ben-Eliyahu and Dechter, 1994]. For a more extensive overview on complexity results in this area, see for instance [Dantsin et al., 2001]. In light of these intractability results, most standard ASP solvers rely on techniques originating in SAT solving [Eén and Sörensson, 2003] or constraint satisfaction problem solving [Badros et al., 2001], where such intractability results have already been successfully tackled for practical applications. A good overview of current SAT solvers can for example be found in [Kullmann, 2009] and the accompanying SAT competition.

The algorithm presented in this thesis was developed with techniques that stem from a more theoretical point of view, namely parameterized complexity theory (see e.g. [Downey and Fellows, 1999]) and fixed-parameter algorithms [Niedermeier, 2006]. The idea here is to consider a certain problem parameter as bounded by a constant and thereby obtain a tractability result (i.e. polynomial runtime) for the given problem. A fixed-parameter algorithm is one where the runtime can be described as a function

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

where $f(k)$ is a computable function solely depending on the parameter k.

One such parameter which has lead to many interesting results is called treewidth [Robertson and Seymour, 1984, Kloks, 1994]. Treewidth represents, informally speaking, the "tree-likeness" of a graph. For disjunctive logic programs, the consistency problem (i.e. checking whether such a program has an answer set) has been shown to become tractable when considering only programs whose incidence graphs have bounded

treewidth (cf. [Gottlob et al., 2010]). Treewidth is defined in terms of tree decompositions which in turn can be used by dynamic programming algorithms to solve the problem under consideration. A survey on such applications can for instance be found in [Bodlaender, 1997].

Main Contributions In this thesis we present a novel dynamic programming-based algorithm working on tree decompositions. This fixed-parameter algorithm is specifically tailored for solving head-cycle free answer set programs. The main idea is to first obtain a tree decomposition representation of such a logic program and subsequently evaluate the logic program set by step, by doing a bottom-up traversal of the tree decomposition. In addition to the answer set characterization of head-cycle free programs given in [Ben-Eliyahu and Dechter, 1994], we provide an alternative characterization which forms the basis of our algorithm. This alternative characterization provides a new way of interpreting answer sets for head-cycle free disjunctive logic programs.

In order to realize the algorithm, a general-purpose framework for working with tree decompositions has been implemented. Using this framework, various algorithms based on tree decompositions can be developed. Currently under development are algorithms for argumentation [Dvorák et al., 2010] or multi-cut in graphs [Pichler et al., 2010]. An implementation of the algorithm described in this thesis was created using said framework and the implementation was thoroughly benchmarked and evaluated against existing answer set programming solvers.

Benchmark results show significant potential for instances of low treewidth. Due to a linear runtime behavior for a bounded parameter, the overall runtime of the algorithm is very competitive when compared to existing systems such as DLV [Leone et al., 2006] and depending on the program size a huge performance increase can be observed. As however good runtime results can only be obtained for low parameter values (i.e. low treewidth), we see this approach–albeit able to evaluate every head-cycle free disjunctive logic program–as an augmentation and benchmarking tool for current solvers that could be used as a component in existing solvers to speed up the computations of low-treewidth programs, whereas for high treewidth, the original solver would be used.

Organization This thesis is organized into four parts, namely the Introduction, Preliminaries, Main Results and Final Thoughts.

The preliminaries part introduces the basic concepts needed for our algorithm and is split into three chapters: In chapter 2, the foundations of mathematical logic and logic programming under various semantics are laid out. Chapter 3 deals with the definition of treewidth, tree decompositions and heuristics used to calculate one. Chapter 4 introduces basic classical notions of complexity theory and then deals with parameterization and parameterized complexity theory. In that chapter, also the use of treewidth

as a parameter is discussed and its use for directly classifying fixed-parameter tractable problems is explained.

The main results comprise two chapters: In chapter 5, the theoretical foundations of the algorithm, including the alternative characterization for answer sets in head-cycle free programs is laid out, and the formal definition of the algorithm is given, including an example evaluation of a logic program. Chapter 6 then deals with the general-purpose framework for working with tree decompositions including a short tutorial on how such algorithms can be implemented. In section 6.2, the actual algorithm implementation (dubbed "dynASP") is discussed. Finally, section 6.3 details the benchmark setup and benchmarking results of our algorithm when compared to the DLV system [Leone et al., 2006].

In the final thoughts part, in chapter 7 we discuss the obtained results, and finally provide some concluding remarks and future work ideas in chapter 8.

II Preliminaries

2 Answer Set Programming

This chapter introduces the necessary theoretical basis of mathematical logic and answer set programming. In the first two sections, basic notions from mathematical logic are briefly reviewed and notational agreements are set.

Formal logic systems, as defined in the following sections, consist of the following three components:

- *Grammar $\mathcal{G}$:* A tuple $\langle V, T, P, S \rangle$, consisting of the set of non-terminal symbols (or variables) V, the set of terminal symbols T, the set of productions P and the starting symbol S. For logic systems the set T is usually called the alphabet and consists of the signature of the program, notational symbols and symbols for the logical connectives. The productions in P define the actual syntax of the language of the logic system. A sentence that is produced by the grammar is called a *formula* in the context of a logic system.
- *Interpretation $\mathcal{I}$*: A function that provides a way to interpret the syntax of the language defined by the grammar. Using an interpretation the semantics of a sentence in the language can be determined.
- *Satisfaction relation $\models_\Sigma$:* A relation between interpretations and formulas that determines whether a given interpretation function (or interpretation, for short) satisfies a given formula with respect to the language signature Σ.

In the following sections we will establish the foundations of two formal logic systems, namely propositional logic and predicate logic. This introduction to answer set programming roughly follows the same structure as [Beierle and Kern-Isberner, 2008], therefore we refer to that book for a more thorough introduction to the subject. The interested reader may also find more in-depth information on the subjects of this chapter in e.g. [Baral, 2003, Huth and Ryan, 2004].

2.1 Propositional Logic

Propositional logic is a formal system as described at the beginning of this chapter. A sentence (or formula) in propositional logic represents a proposition (hence the name). Throughout this thesis, propositional logic and predicate logic (see section 2.2) are the

fundamental systems used to formalize propositions in a uniform way, so that they can be processed by a computer system. The following definitions lay out the syntax and semantics.

Definition 2.1. *The **syntax of PL0** is defined by a grammar* $G_{PL0} = \langle V, \mathcal{A} = \Sigma_{PL0} \cup \mathrm{Conn} \cup \mathrm{Sym}, P_{PL0}, S \rangle$*, with the set of non-terminals (variables)* V*, the alphabet* $\mathcal{A}$ *and the starting symbol* S *of the grammar.*

- $V = \{\varphi\}$
- $\mathcal{A}$ *consists of the following disjoint components:*
  - *The signature* Σ_{PL0}*, a set of propositional variables.*
  - *The set of logical connectives* $\mathrm{Conn} = \{\neg, \vee, \wedge, \Rightarrow\}$.
  - *The set of auxiliary symbols* $\mathrm{Sym} = \{(,)\}$.
- P*, the set of productions of the grammar consists of the following single rule:*
  - $\varphi \rightarrow (\neg\varphi) \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \Rightarrow \varphi) \mid a$*, for* $a \in \Sigma_{PL0}$
- $S = \varphi$.

Definition 2.2. *An **interpretation** of a PL0 formula is an interpretation function* I *which is defined in the following way:*

- $I(a) \in \{\mathbf{true}, \mathbf{false}\}$ *for every* $a \in \Sigma_{PL0}$
- $I(F \wedge G), I(F \vee G), I(\neg F), I(F \Rightarrow G)$*: After evaluation of* $I(F)$ *and* $I(G)$*, the result is given by the truth tables in 2.1.*

$\neg$	
false	**true**
true	**false**

(a) not

$\wedge$	**false**	**true**
false	**false**	**false**
true	**false**	**true**

(b) and

$\vee$	**false**	**true**
false	**false**	**true**
true	**true**	**true**

(c) or

$\Rightarrow$	**false**	**true**
false	**true**	**true**
true	**false**	**true**

(d) implies

Table 2.1: Interpretation function for logical connectives in PL0

Definition 2.3. *The **satisfaction relation** $\models_{\Sigma_{PL0}}$ of PL0 is defined as follows:*

$$I \models_{\Sigma_{PL0}} F \text{ if and only if } I(F) = \mathbf{true}$$

where F *is a PL0 formula. An interpretation that satisfies* F *under above relation is called a **model** of* F.

Definition 2.4. *A formula F is **satisfiable** if it has at least one model and F is **valid** if every interpretation is a model, whereas it is **unsatisfiable** if it has no model.*

Definition 2.5. *A **literal** is either an atom or the negation of an atom. A formula is in **conjunctive normal form** (CNF), if it is a conjunction of clauses, where a clause consists of a disjunction of literals.*

Example 2.6. *Consider the formula $\varphi = a \wedge (b \vee c) \wedge d$. φ is in CNF whereas the formula $\psi = a \vee (b \wedge c) \vee d$ is not.*

Remark 2.7. *Every formula can be converted to a logically equivalent formula in conjunctive normal form at the cost of an (in the worst case) exponential increase in the size of the formula (see e.g. [Jackson and Sheridan, 2004]). However, with only a polynomial blow-up of the formula size, a SAT-equivalent formula in conjunctive normal form can be constructed (i.e. a CNF-formula that is satisfiable if and only if the original formula is satisfiable).*

2.2 Predicate Logic

Predicate logic (PL1) is an extension of propositional logic whereby in PL1 it is possible to form sentences that quantify over single objects: Where in propositional logic it was impossible to say that every car is red, this is now possible in PL1. The following definitions set forth the syntax and semantics of PL1:

Definition 2.8. *The **syntax of PL1** is defined by a grammar $G_{PL1} = \langle V, \mathcal{A} = \Sigma_{PL1} \cup \text{Conn} \cup \text{Var} \cup \text{Sym}, P_{PL1}, S \rangle$, with the set of non-terminals (variables) V, the alphabet $\mathcal{A}$ and the starting symbol S of the grammar.*

- $V = \{\tau, \varphi\}$
- *$\mathcal{A}$ consists of the following disjoint components:*
  - *The signature* $\Sigma_{PL1} = \text{Func} \cup \text{Pred}$ *itself consisting of the sets of function symbols and predicate symbols and their respective arities.*
  - *The set of logical connectives* $\text{Conn} = \{\neg, \vee, \wedge, \Rightarrow, \forall, \exists\}$.
  - *The set of variables* Var.
  - *The set of auxiliary symbols* $\text{Sym} = \{\underline{(}, \underline{)}\}$.
- *P, the set of productions of the grammar consists of the following rules:*
  - $\tau \rightarrow x \mid f(\tau, \ldots, \tau)$ *for* $x \in \text{Var}$, $f/n \in \text{Func}$
  - $\varphi \rightarrow p(\tau \ldots \tau) \mid (\neg\varphi) \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \Rightarrow \varphi) \mid (\exists x\varphi) \mid (\forall x\varphi)$ *for* $x \in \text{Var}$, $p/n \in \text{Pred}$

- $S = \varphi$.

The intuitive meaning of all words produced by the non-terminal τ is that they represent basic objects or facts and are called *terms*. A term is called *ground* if it contains no variables. A function of arity 0 is called a *constant*. All words produced by φ are called *formulas* that can be either **true** or **false**. The production $\varphi \to p(\tau \dots \tau)$ with p being a predicate is called an *atom*. An atom is called *ground* if all the terms contained in it are ground.

Definition 2.9. *An **interpretation** of a PL1 formula is a tuple $\mathcal{I} = \langle D, I_\mu, \mu \rangle$, where D is an arbitrary domain, I_μ is the interpretation function and $\mu : \mathrm{Var} \to D$ is a variable assignment.*

The interpretation function I_μ is defined in the following way:

- $I_\mu(c) \in D$ *for every function symbol c of arity 0*
- $I_\mu(x) = \mu(x)$ *for every variable* $x \in \mathrm{Var}$
- $I_\mu(f) : D \times \dots \times D \to D$ *for every function symbol f of arity* > 0
- $I_\mu(p) \subseteq D \times \dots \times D$ *for every predicate symbol p*
- $I_\mu(F \wedge G), I_\mu(F \vee G), I_\mu(\neg F), I_\mu(F \Rightarrow G)$*: See definition of PL0*
- $I_\mu(\forall x F) =$ **true** *if and only if* $\forall c \in D : I_{\mu \cup \{x \mapsto c\}}(F) =$ **true**
- $I_\mu(\exists x F) =$ **true** *if and only if* $\exists c \in D : I_{\mu \cup \{x \mapsto c\}}(F) =$ **true**

Note that the above definition does not deal with unquantified variables and double quantified variables. W.l.o.g. we may assume that all variables are quantified (i.e. unquantified variables are implicitly universally quantified) and no variable is quantified over twice, which can be achieved by introducing additional variables.

Definition 2.10. *The **satisfaction relation** $\models_{\Sigma_{PL1}}$ of PL1 is defined as in PL0, with $\mathcal{I} = \langle D, I_\mu, \mu \rangle$, as defined in 2.9.*

$$\mathcal{I} \models_{\Sigma_{PL1}} F \text{ if and only if } I_\mu(F) = \textbf{true}$$

*where F is a PL1 formula. An interpretation that satisfies F under above relation is called a **model** of F.*

Definition 2.11. *The definitions of **satisfiability, validity** and **unsatisfiability** are equivalent to PL0.*

It can now be easily seen that PL0 is a subset of PL1 where the set of variables and the set of function symbols is simply empty (which implies that only predicates of arity 0 can exist). Every PL0 formula is also a PL1 formula. Therefore PL0 $\subseteq$ PL1 can be easily established. It can be shown that this inclusion is, in fact, strict: We look at the complexity of the satisfiability problem: The problem of checking whether a given

propositional formula is valid (i.e. true in all possible interpretations) is known to be coNP-complete, as shown in e.g. [Cook, 1971]. However, checking whether a given PL1 formula is valid is an undecidable problem, following from the Church-Turing thesis ([Church, 1936, Turing, 1937]) and therefore PL0 $\subset$ PL1.

2.3 Monadic Second-Order Logic

Monadic second-order logic (MSO, MSOL) is in turn an extension of predicate logic whereby in MSOL it is not only possible to form sentences that quantify over single objects as in PL1, it is also possible to quantify over sets of objects. The following definitions set forth the syntax and semantics of MSOL:

Definition 2.12. *The **syntax of MSOL** is defined by a grammar* $G_{MSO} = \langle V, \mathcal{A} = \Sigma_{MSO} \cup \text{Conn} \cup \text{Var} \cup \text{SVar} \cup \text{Sym}, P_{MSO}, S \rangle$*, with the set of non-terminals (variables)* V*, the alphabet* $\mathcal{A}$ *and the starting symbol* S *of the grammar.*

- $V = \{\tau, \varphi\}$
- $\mathcal{A}$ *consists of the following disjoint components:*
  - *The signature* $\Sigma_{PL1} = \text{Func} \cup \text{Pred}$ *itself consisting of the sets of function symbols and predicate symbols and their respective arities.*
  - *The set of logical connectives* $\text{Conn} = \{\neg, \vee, \wedge, \Rightarrow, \forall, \exists, \in\}$.
  - *The set of variables* Var.
  - *The set of set variables* SVar.
  - *The set of auxiliary symbols* $\text{Sym} = \{\underline{(}, \underline{)}\}$.
- P*, the set of productions of the grammar consists of the following rules:*
  - $\tau \rightarrow x \mid f(\tau, \ldots, \tau)$ *for* $x \in \text{Var}$, $f/n \in \text{Func}$
  - $\varphi \rightarrow p(\tau, \ldots, \tau) \mid (\neg\varphi) \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \Rightarrow \varphi) \mid (\exists x\varphi) \mid (\forall x\varphi) \mid (\exists X\varphi) \mid (\forall X\varphi) \mid (x \in X)$*, for* $x \in \text{Var}, X \in \text{SVar}$, $p/n \in \text{Pred}$.
- $S = \varphi$.

The semantics are essentially those of PL1 except that now it is also possible to quantify over sets by means of the set variables. These set variables represent a subset of the domain that is part of an interpretation.

Definition 2.13. *An **interpretation** of an MSO formula is a tuple* $\mathcal{I} = \langle D, I_\mu, \mu, \nu \rangle$*, where* D *is an arbitrary domain,* $I_{\mu,\nu}$ *is the interpretation function,* $\mu : \text{Var} \rightarrow D$ *is a variable assignment and* $\nu : \text{SVar} \rightarrow 2^D$ *is a set assignment.*

The interpretation function $I_{\mu,\nu}$ *is defined in the following way:*

- $I_{\mu,\nu}(c) \in D$ *for every function symbol* c *of arity 0*
- $I_{\mu,\nu}(x) = \mu(x)$ *for every variable* $x \in \text{Var}$
- $I_{\mu,\nu}(f) : D \times \cdots \times D \to D$ *for every function symbol* f *of arity* > 0
- $I_{\mu,\nu}(p) \subseteq D \times \cdots \times D$ *for every predicate symbol* p
- $I_{\mu,\nu}(F \wedge G), I_{\mu,\nu}(F \vee G), I_{\mu,\nu}(\neg F), I_{\mu,\nu}(F \Rightarrow G), I_{\mu,\nu}(\forall x F), I_{\mu,\nu}(\exists x F)$ *with* $x \in \text{Var}$*: See definition of PL1*
- $I_{\mu,\nu}(x \in X) =$ **true** *if and only if* $\mu(x) \in \nu(X)$*, with* $x \in \text{Var}$*,* $X \in \text{SVar}$
- $I_{\mu,\nu}(\forall X F) =$ **true** *if and only if* $\forall C \in 2^D : I_{\mu,\nu\cup\{X\mapsto C\}}(F) =$ **true***, with* $X \in \text{SVar}$
- $I_{\mu,\nu}(\exists X F) =$ **true** *if and only if* $\exists C \in 2^D : I_{\mu,\nu\cup\{X\mapsto C\}}(F) =$ **true***, with* $X \in \text{SVar}$

We deal with unquantified and multiply quantified variables as in PL1.

Definition 2.14. *The* ***satisfaction relation*** $\models_{\Sigma_{MSO}}$ *of MSOL is defined as in PL1*

$$\mathcal{I} \models_{\Sigma_{MSO}} F \text{ if and only if } I_{\mu,\nu}(F) = \textbf{true}$$

where F *is an MSO formula. An interpretation that satisfies* F *under above relation is called a* ***model*** *of* F*.*

Definition 2.15. *The definitions of* ***satisfiability, validity*** *and* ***unsatisfiability*** *are equivalent to PL1.*

Monadic second-order logic distinguishes itself from "full" second-order logic only in the fact, that the function ν in an MSO-interpretation maps a set variable X to a subset of the domain, and therefore, MSO-formulas can only include sentences of the form $x \in X$ for establishing relations between variables and set variables. In full second-order logic, set variables can also be mapped to sets of function symbols or predicates in the language, where instead of $x \in X$ one would write $X(x)$ to establish relationships between variables and set variables. If X ranges over a subset of the domain, $X(x)$ has the same semantics as $x \in X$. If, however, X ranges over a set of predicates (say of arity one), then $X(x)$ says that for every predicate p in X, $p(x)$ must be provable.

2.4 Logic Programs and Answer Set Semantics

In this section the basic concepts of logic programs and logic programming under answer-set semantics are introduced.

2.4.1 Logic Programs

Logic programming follows a central paradigm: The declarative encoding of knowledge and problems as a logical formula whereby the resulting model(s) represent solutions.

Traditional programs on computers are procedural: They execute a fully determined sequence of commands that calculate the solutions to a given problem. In logic programming one only states, in a declarative way, what the problem is. A logic programming solver can then obtain a solution without knowing any procedural, problem-specific routines. Figure 2.1 shows the relation between real-world problems and their declarative encoding as a logic program in a graphical way:

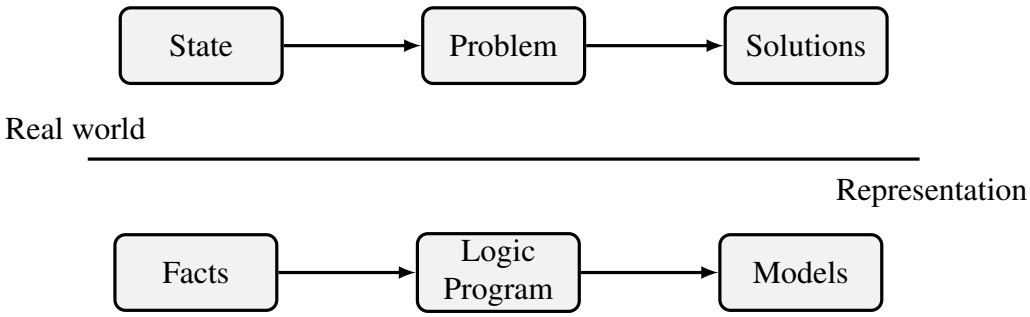


Figure 2.1: The relation between real-world problems and logic programs.

This relation is also illustrated in the following example that encodes a graph three-coloring problem as a logic program. Graph three-coloring takes a graph and as input and tries to color its vertices with tree colors in such a way that no two adjacent vertices have the same color.

Example 2.16. *Let P be a logic program consisting of the following four clauses:*

$V(a).V(b).V(c).$
$E(a,b).E(b,c).E(c,a).$
$R(X) \vee G(X) \vee B(X) \leftarrow V(X).$
$\bot \leftarrow E(X,Y), R(X), R(Y).$
$\bot \leftarrow E(X,Y), G(X), G(Y).$
$\bot \leftarrow E(X,Y), B(X), B(Y).$

Example 2.16 describes a graph with three vertices represented by V, three edges between them, represented by E, the condition that every vertex must have a color (being a vertex implies that either R, G or B holds for that vertex), and the conditions that if there is an edge between two vertices, then those two cannot have the same color.

We will see later on when defining the semantics for such logic programs, that the solutions of Example 2.16 correspond exactly to the solutions of the graph three-colorability problem for a graph with three vertices and three edges.

We begin with a few basic definitions concerning logic programs:

Definition 2.17. *A **Horn clause** is a rule of the form*

$$H \leftarrow B_1, \ldots, B_n$$

where H, B_i are atoms as defined in predicate logic. In such a rule, H is called the ***head*** *and $B_1, \ldots, B_n$ is called the* ***body****. If $n = 0$ then such a rule is called a* ***fact****.*

A ***Horn*** *or* ***classical logic program*** *is a finite set of Horn clauses. If none of these clauses contain any variables, the program is called* ***ground****.*

The ***Herbrand universe*** *$U_{\mathcal{L}}$ of a PL1-language $\mathcal{L}$ is the set of all ground terms that can be formed in $\mathcal{L}$.*

The ***Herbrand base*** *$\mathcal{H}(\mathcal{L})$ of a PL1-language $\mathcal{L}$ is the set of all ground atoms that can be formed in $\mathcal{L}$.*

With each logic program P we associate the PL1-language $\mathcal{L}_P$ that generates it. The signature of $\mathcal{L}_P$ consists exactly of the predicates and functions occurring in P.

A ***Herbrand interpretation*** *of a program P is a subset $M \subseteq \mathcal{H}(\mathcal{L}_P)$. M is called a* ***Herbrand model*** *of P if $\langle \mathcal{H}(\mathcal{L}_P), M, \emptyset \rangle \models_{\mathcal{L}_P} P$, i.e. if it satisfies for every rule in P the formula $\forall \mathbf{x}(B_1 \wedge \cdots \wedge B_n \Rightarrow H)$ where $\mathbf{x}$ contains all the variables occurring in the rule. We write $M \models_{\mathcal{L}_P} P$, as the domain and μ function are implicitly given by P and M.*

We consider in the following only programs that are function-less, i.e. that have only function symbols of arity 0.

The following example shows a simple such logic program and subsequently its Herbrand universe and base as well as a Herbrand interpretation that is obviously also a model.

Example 2.18. *Let P be a logic program consisting of the following four clauses:*

B(*tweety*).
P(*skipper*).
$B(X) \leftarrow P(X)$.
$F(X) \leftarrow B(X)$.

$U_{\mathcal{L}_P}$ = {*tweety, skipper*}

$\mathcal{H}(\mathcal{L}_P)$ = {*B(tweety), B(skipper), P(tweety), P(skipper), F(tweety), F(skipper)*}

A Herbrand model: M = {*B(tweety), B(skipper), P(skipper), F(tweety), F(skipper)*}

Notice that in the above example, as we associate with program P the language $\mathcal{L}_P$ as defined above, the set of function and predicate symbols is finite (as the program is function-less) and implicitly given by the program itself. As this holds for every (finite) classical logic program, instead of we can simply view clauses with variables as "abbreviations" for all the ground clauses that they represent, e.g. the rule B(X) ← P(X) in the above example actually represents the two clauses B(tweety) ← P(tweety) and B(skipper) ← P(skipper). From this point onward we consider only ground programs and non-ground rules used as abbreviations for the corresponding ground program.

Semantics of classical logic programs

Deciding the smallest Herbrand model for a given ground (Horn) logic program is P-complete [Dantsin et al., 2001]. This tractability result is due to the following property:

Proposition 2.19. *Let M_1 and M_2 be Herbrand models of a program P. Then also $M = M_1 \cap M_2$ is a Herbrand model of P.*

The set of all Herbrand models $\mathcal{M}(P)$ of a program P is therefore said to be *closed under intersection*. From Proposition 2.19 the following corollary immediately follows:

Corollary 2.20. *P has a unique smallest Herbrand model given by the following formula:*

$$LM(P) = \bigcap_{M \in \mathcal{M}(P)} M$$

2.4.2 Types of Logic Programs

In the last section we already introduced the concept of Horn (or classical) logic programs. However this notion seems not to be expressive enough in certain cases. Therefore, other types of logic programs exist that extend the classical logic programs. We first take a look at *normal logic programs*.

Definition 2.21. *A **normal logic program** P is a set of rules of the following form:*

$$H \leftarrow P_1, \ldots, P_n, not\ N_1, \ldots, not\ N_m$$

*where H, P_i and N_i are atoms (called literals) and $P_1, \ldots, P_n$ (resp. $N_1, \ldots, N_m$) are called the **positive body** (resp. **negative body**) of the rule.*

The not operator is a unary logical connective, called the *negation as failure* operator or, alternatively, *default negation*. Its semantics is the following: not A is **true** if A cannot be proved and **false** otherwise. The not operator therefore also in a sense represents a closed-world assumption: If some fact A cannot be shown to hold, the fact not$A$ is assumed.

Lets take another look at Example 2.18, interpreting the predicates in the following way:

- $B(X)$ represents that X is a bird.
- $P(X)$ represents that X is a penguin.
- $F(X)$ represents the fact that X can fly.

The intuition of the two non-ground rules in the program is then the following: Every bird flies and every penguin is a bird. However in Example 2.18 above we could not represent an exception to the first statement: Every bird flies, except when it is a penguin. Example 2.22 shows how to implement this particular kind of exception:

Example 2.22. *Let P be a logic program consisting of the following clauses:*

$B(tweety)$.
$P(skipper)$.
$B(X) \leftarrow P(X)$.
$F(X) \leftarrow B(X), not\ P(X)$.

It is evident that the above program now regards the case that penguins are birds but cannot fly. However we need to define a meaningful semantics for normal logic programs, which will be done collectively for normal, disjunctive and head-cycle free logic programs in subsection 2.4.3.

Definition 2.23. ***Disjunctive logic programs*** *are programs that consist of rules of the form*

$$H_1 \vee \cdots \vee H_k \leftarrow P_1, \ldots, P_n, notN_1, \ldots, notN_m$$

where H_i, P_i and N_i are atoms and all atoms $H_1, \ldots, H_k$ are called the head atoms, and $P_1, \ldots, P_n$ (resp. $N_1, \ldots, N_m$) are called the positive body (resp. negative body) of the rule.

To simplify matters in this section we introduce the following notational aid: For a rule r in a logic program, by $H(r)$ we denote the set of head atoms of the formula (i.e. $H(r) = \{H_1, \ldots, H_k\}$), the positive body we denote by $B^+(r) = \{P_1, \ldots, P_n\}$ and the negative body by $B^-(r) = \{N_1, \ldots, N_n\}$.

The algorithm presented in this thesis specifically deals with head-cycle free disjunctive logic programs (cf. [Ben-Eliyahu and Dechter, 1994]). This type of logic program is defined as follows:

Definition 2.24. *A disjunctive logic program P is called* ***head-cycle free*** *if there exists a function $l : \mathcal{H}(\mathcal{L}_P) \rightarrow \mathbb{N}$ that maps every atom in the Herbrand base to an integer and fulfills the following properties for every rule r in P:*

1. $\forall a \in H(r), b \in B^+(r) : l(a) \geq l(b)$
2. $\forall a \in H(r), b \in H(r) : a \neq b \Rightarrow l(a) \neq l(b)$

The function l in the above definition defines a level mapping of the atoms occurring in the program. If such a level mapping can be found, the *dependency graph* of the program does not contain a directed cycle that contains two distinct atoms occurring together in the head of some rule in the program (hence the name head-cycle free). The dependency graph simply contains an edge (a, b) if a and b are atoms that occur together in the same rule, where b is contained in the head and a is contained in the positive body.

Lemma 2.25. *[Ben-Eliyahu and Dechter, 1994, Dix et al., 1996] Every head-cycle free disjunctive logic program can in polynomial time be reduced to an equivalent normal logic program .*

2.4.3 Stable Model Semantics

Stable model semantics (introduced in [Gelfond and Lifschitz, 1988]) gained much traction in the answer-set programming community (see [Marek and Truszczyński, 1999, Provetti and Son, 2001, Gelfond and Leone, 2002] for examples) in the following years and, in the logic programming field, is by now a generally accepted semantics for evaluating normal (and in a slightly extended form also disjunctive) logic programs with many applications, e.g. in the field of planning [Lifschitz, 2002], constraint satisfaction [Niemelä, 1999], or ontological reasoning [Eiter et al., 2006]. The basic definitions for stable model semantics (which in this thesis we use synonymously with "answer set semantics") are laid out in the following paragraphs.

Definition 2.26. *A **reduct** P^S of a normal logic program P with respect to a set S of atoms is defined as:*

$$P^S = \{H(r) \leftarrow B^+(r) | r \in P, B^-(r) \cap S = \emptyset\}$$

Example 2.27. *Let a program P be given by the following two rules:*

$$r_1 : a \leftarrow not\ b$$

$$r_2 : b \leftarrow not\ a$$

$$S_1 = \{a\} : P^{S_1} = \{a \leftarrow\}$$

$$S_2 = \{b\} : P^{S_2} = \{b \leftarrow\}$$

The reduct P^S is computed in two steps:

1. Remove all rules where some atom in S occurs in the negative body of the rule.
2. In all remaining rules, remove all default-negated literals.

P^S depends heavily on S but it still contains all facts and all rules that did not have any default-negated literals in P. We can now define a stable model:

Definition 2.28. *S is a **stable model** of a logic program P if*

- *for P without default-negation: S is the minimal Herbrand model of P.*
- *for P with default-negation: S is the minimal Herbrand model of P^S*

Example 2.29. *Consider the program*

$$a \leftarrow not\ a$$

This program has no stable model. Consider on the other hand the following program:

$$a \leftarrow not\ b.\ b \leftarrow not\ a.$$

This program has two stable models, namely $\{a\}$ *and* $\{b\}$ *as can easily be seen from Definition 2.28 and Example 2.27.*

If we take another look at Example 2.22 we can see that the stable model semantics provide us with a model that reflects our intentions: The atom F(skipper) is not included in the model.

2.4.4 Alternative characterization of stable models for head-cycle free logic programs

In [Ben-Eliyahu and Dechter, 1994], a different characterization of stable models is given for head-cycle free problems. For the following theorem, let $\mathcal{A}(P)$ denote the set of atoms and $\mathcal{R}(P)$ the set of rules occurring in a logic program P.

Theorem 2.30. *[Ben-Eliyahu and Dechter, 1994] Let P be a grounded head-cycle free disjunctive logic program then a set $M \subseteq \mathcal{A}(P)$ is a stable model if and only if the following conditions hold:*

1. *M satisfies all the rules in P*
2. *There exists a function $f : \mathcal{A}(P) \rightarrow \mathbb{N}^+$ such that $\forall a \in M \exists r \in \mathcal{R}(P)$:*
   - $B^+(r) \subseteq M \wedge B^-(r) \cap M = \emptyset$
   - $H(r) \cap M = \{a\}$
   - $\forall b \in B^+(r) : f(b) < f(a)$

Informally the above theorem says that for every atom a in a stable model there must exist a sequence of rules such that every rule derives exactly one atom, the first rule in the sequence is a fact and the last rule in the sequence derives exactly atom a. All the bodies of the rules must be satisfied under the condition that no rule derives an atom used to satisfy a body of a rule that occurs earlier in the sequence.

2.4.5 Computational complexity

Usually solvers for logic programs only work on ground programs and therefore follow a two-step approach to handle non-ground programs: In the first step, compute the explicit ground representation of the non-ground program (by instantiating the variables in the non-ground rules with all possible combinations of constants). Secondly, solve the original problem on the grounded program. Note that the grounded program can

be exponentially larger than the non-ground version considering the (implicit) domain size.

Taking another look at Example 2.16, we can, from the non-ground rules in the program now calculate all the ground rules that they represent. The grounded program is shown in Example 2.31.

Example 2.31. *Let P be the logic program of Example 2.16. Then the grounding of P (denoted ground(P)) is the following program:*

$V(a).V(b).V(c).$
$E(a,b).E(b,c).E(c,a).$
$R(a) \vee G(a) \vee B(a) \leftarrow V(a).$
$R(b) \vee G(b) \vee B(b) \leftarrow V(b).$
$R(c) \vee G(c) \vee B(c) \leftarrow V(c).$
$\bot \leftarrow E(a,a), R(a), R(a).\bot \leftarrow E(b,b), R(b), R(b).$
$\bot \leftarrow E(c,c), R(c), R(c).$
$\bot \leftarrow E(a,b), R(a), R(b).\bot \leftarrow E(b,a), R(b), R(a).$
$\bot \leftarrow E(a,c), R(a), R(c).\bot \leftarrow E(c,a), R(c), R(a).$
$\bot \leftarrow E(b,c), R(b), R(c).\bot \leftarrow E(c,b), R(c), R(b).$
$\bot \leftarrow E(a,a), G(a), G(a).\bot \leftarrow E(b,b), G(b), G(b).$
$\bot \leftarrow E(c,c), G(c), G(c).$
$\bot \leftarrow E(a,b), G(a), G(b).\bot \leftarrow E(b,a), G(b), G(a).$
$\bot \leftarrow E(a,c), G(a), G(c).\bot \leftarrow E(c,a), G(c), G(a).$
$\bot \leftarrow E(b,c), G(b), G(c).\bot \leftarrow E(c,b), G(c), G(b).$
$\bot \leftarrow E(a,a), B(a), B(a).\bot \leftarrow E(b,b), B(b), B(b).$
$\bot \leftarrow E(c,c), B(c), B(c).$
$\bot \leftarrow E(a,b), B(a), B(b).\bot \leftarrow E(b,a), B(b), B(a).$
$\bot \leftarrow E(a,c), B(a), B(c).\bot \leftarrow E(c,a), B(c), B(a).$
$\bot \leftarrow E(b,c), B(b), B(c).\bot \leftarrow E(c,b), B(c), B(b).$

It can bee seen that the program size increases very quickly if a rule with more than one variable is encoutered. The algorithm for solving head-cycle free programs that is presented later on is designed to work with ground programs, therefore we will also restrict our complexity analysis to ground programs. Making this restriction, the complexity of such a program is often called the *data complexity* the program.

By proposition 2.19 we have already established that, given a ground Horn logic program, checking whether it has a model is P-complete. However, stable model semantics and the smallest Herbrand model coincide for these types of programs. Therefore we now look at how the complexity changes under stable model semantics for normal and disjunctive logic programs in the ground case.

The sensible candidate set for models of normal logic programs is still the Herbrand base of the program as no strongly negated atoms occur (only positive or default-

negated atoms). Dealing with negation as failure however require techniques different from simple Herbrand model semantics. Proposed approaches include stratification [Apt et al., 1988], well-founded semantics [Gelder et al., 1991] and stable model/answer set semantics [Gelfond and Lifschitz, 1988, Gelfond and Lifschitz, 1990]. For the last, the following theorem establishes NP-completeness for the consistency problem (i.e. checking whether a stable model exists):

Theorem 2.32. *[Marek and Truszczynski, 1991, Dantsin et al., 2001] Given a ground normal logic program P, deciding whether P has a stable model is NP-complete.*

When disjunctions are added to the head of rules, the expressiveness of the programs increases. Therefore, deciding the consistency problem for disjunctive logic programs in the ground case is Σ_P^2-complete as the following theorem states:

Theorem 2.33. *[Eiter and Gottlob, 1995, Dantsin et al., 2001] Given a grounded disjunctive logic program P, deciding whether P has a stable model is Σ_P^2-complete.*

As this thesis deals mostly with head-cycle free disjunctive logic programs as defined before, we also investigate the complexity of this class of logic programs. The following theorem gives us an NP-completeness result for stable model semantics.

Theorem 2.34. *[Ben-Eliyahu and Dechter, 1994] Given a grounded disjunctive logic program P where P is head-cycle free, then deciding whether P has a stable model is NP-complete.*

Proof. The theorem follows directly from lemma 2.25 and theorem 2.32. □

Another example for a head-cycle free ground logic program is the one presented in Example 2.35. Note that in this example we use only predicate symbols of arity 0, basically making the program an instance of PL0. Note also that this program is trivially head-cycle free as there are no disjunctions in the program.

Example 2.35. *Let P be the following program:*

$$r_1 = u \leftarrow v, y; \quad r_2 = z \leftarrow u; \quad r_3 = v \leftarrow w;$$
$$r_4 = w \leftarrow x; \quad r_5 = x \leftarrow \neg y, \neg z.$$

The only stable model of P is the set v, w, x.

In the future (and also as the algorithm presented here works on these kinds of programs) we will henceforth only consider programs of the kind of Example 2.35. Note that it is easy to transform a ground disjunctive logic program to a program that only contains predicates of arity 0: We can represent each atom in the original program by some new predicate of arity 0.

3 Treewidth

Hard problems on graphs often become very easy when being restricted to trees. For instance, the VERTEX COVER or DOMINATING SET problems, that are in general NP-hard, can be solved in linear time when restricted to trees. Also many constraint satisfaction problems, when their graph-representation is restricted to trees, become easy to solve, e.g. deciding boolean conjunctive query containment becomes LOGCFL-complete when restricted to acyclic queries, as shown in [Gottlob et al., 2001].

Looking at these results it is natural to ask what the properties of trees are that make them nice to work with and whether they can be generalized to graphs or, in other words, we would like to have a measure of the "tree-likeness" of a graph.

3.1 Tree Decompositions

The notion of *tree decompositions* tries to address this matter. In their 1984 paper, Robertson and Seymour introduce the concepts of tree decompositions and treewidth which nowadays play an important role in the field of graph theory but also in the field of fixed-parameter algorithms (see [Robertson and Seymour, 1984]). In this section the basic definitions about tree decompositions and treewidth are given. Many survey papers (e.g. [Bodlaender, 1993b, Bodlaender, 1997, Bachoore and Bodlaender, 2005]) exist that give good overviews over the material. We refer the interested reader to those and to [Kloks, 1994] for details.

Definition 3.1. *An (undirected)* ***graph*** G *is a pair* $G = (V, E)$ *of sets, where the set* V *contains all the* ***vertices*** *in the graph and* E *consists of the* ***edges****. An edge is identified by an unordered pair* $(v_1, v_2), v_1 \in V, v_2 \in V$ *of adjacent vertices.*

A graph is called ***simple*** *if it only contains edges between distinct vertices and there exists at most one edge between any two vertices.*

In the following we only deal with simple, undirected graphs and therefore use the word "graph" in that sense. Note that, as in a simple graph every edge is uniquely identified by the two distinct vertices that it connects, we simply refer to an edge connecting vertices v_1 and v_2 as (v_1, v_2) instead of assigning each edge a label. Therefore, $(v_1, v_2) \in E$ represents the fact that graph $G = (V, E)$ contains an edge between vertices v_1 and v_2.

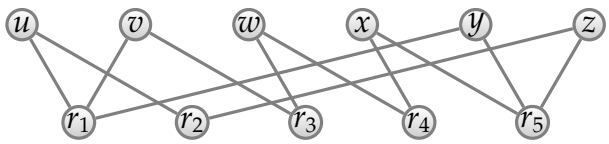
Definition 3.2. *A **tree decomposition** of a graph $G = (V, E)$ is a pair $(\chi, T = (N, F))$ where T is a (rooted) tree with nodes N and edges F and $\chi : N \to 2^V$ is a labeling function assigning for each node $n \in N$ a set $\chi(n) \subseteq V$, such that they satisfy the following properties:*

- $\bigcup_{n \in N} \chi(n) = V$
- $(v_1, v_2) \in E \Rightarrow \exists n \in N : \{v_1, v_2\} \subseteq \chi(n)$
- $v \in \chi(n_1) \wedge v \in \chi(n_2) \wedge n_3 \in \text{path}(n_1, n_2) \Rightarrow v \in \chi(n_3)$

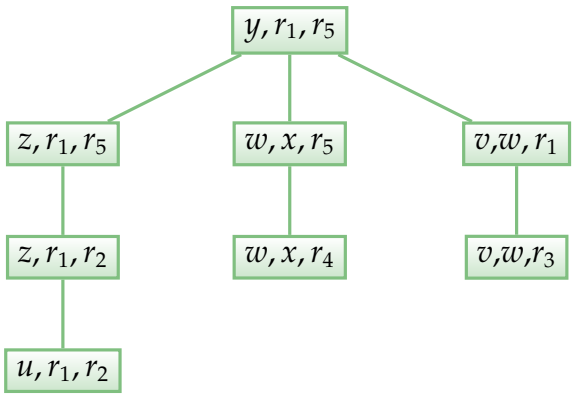
*The set $\chi(n)$ is also called the **bag** of node n.*

The third of these conditions is referred to as the *connectedness condition*. The $\text{path}(n_1, n_2)$ function used in that condition returns the set of all tree nodes along the unique path from n_1 to n_2.

Example 3.3. *Let G be the following graph:*



A tree decomposition for G may look like this:



The tree decomposition above has a width *of 2.*

Definition 3.4. *The **width** of a tree decomposition is the size of the largest bag occurring in it, minus one.*

*The **treewidth** of a graph G is the minimum width of all possible tree decompositions of G.*

An intuitive characterization of tree decompositions and treewidth is given by the so-called *robber-cop game* as established in [Seymour and Thomas, 1993]: In the graph the robber occupies a vertex and wants to avoid the cops. To that end he can, at any time, run from his current vertex to any other vertex in the graph as long as there is a path connecting the two. However he is not permitted to pass a vertex occupied by a cop. Cops either occupy a vertex or they fly to another vertex at any given time. The goal for the cops is to fly to the vertex that the robber currently occupies. However when the robber sees the cops approaching he will run to another vertex, therefore the only way for the cops to catch the robber is to occupy all adjacent vertices and then, with another cop, fly to the vertex that the robber occupies. The treewidth of a graph is then the minimum number of cops needed to do this, minus one.

3.1.1 Normalization

A distinctly "nice" class of tree decompositions are so-called *normalized* tree decompositions. These exhibit a particularly simple structure and are therefore easy to work with and simplify the task of developing dynamic-programming algorithms for solving problems based on such tree decompositions.

Definition 3.5. *[Kloks, 1994] A tree decomposition* (χ, T) *is called* ***normalized*** *or* ***nice*** *if it satisfies the following properties:*

- *Every node of the tree decomposition has at most two children.*
- *If a node* n *has two children* m *and* p*, then* $\chi(n) = \chi(m) = \chi(p)$*. Node* n *is then called a* JOIN *or* BRANCH NODE*.*
- *If* n *has only one child* m*, then* $\mathrm{abs}(|\chi(n)| - |\chi(m)|) = 1$*.*
  - *If* $\chi(n) \subset \chi(m)$*, then* n *is called a* FORGET *or* REMOVAL NODE*.*
  - *If* $\chi(m) \subset \chi(n)$*, then* n *is called a* INTRODUCTION NODE*.*
- *If* n *has no child nodes, then* n *is called a* LEAF NODE*.*

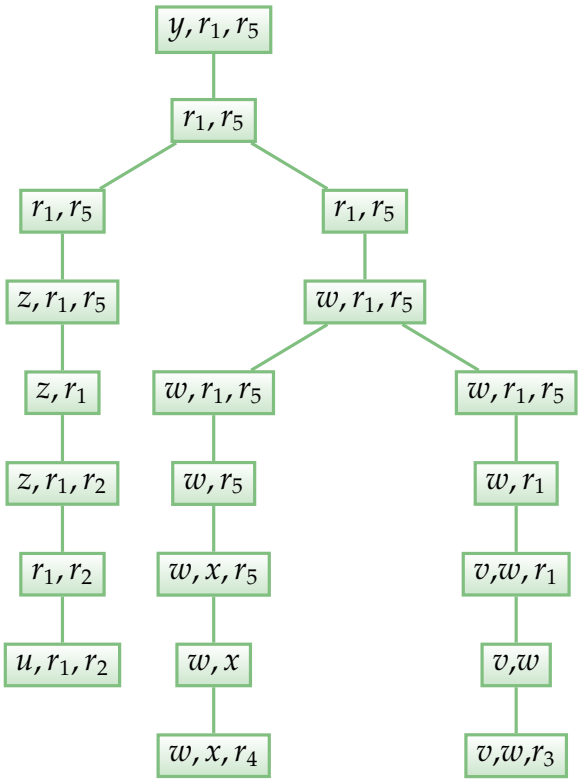
Conveniently, normalizing a tree decomposition does not incur any penalties regarding its width:

Lemma 3.6. *For a graph* G*, given a tree decomposition of width* k *and with* n *nodes, one can in* $O(n)$ *time find a normalized tree decomposition of width* k *for graph* G *that has* $O(n)$ *nodes.*

Proof. (Idea) It can easily be seen that, given a tree decomposition, the properties defined in Definition 3.5 can be satisfied by firstly modifying branch nodes in such a way that they only have two children (by creating new child branch nodes as long as there are too many child nodes), and inserting new removal and introduction nodes between

nodes whose bags differ by more than one node. This simple procedure yields a normalized tree decomposition. □

Example 3.7. *Revisiting the tree of Example 3.3, we can easily adapt the tree decomposition to the following:*



As the tree decomposition in Example 3.3, also this tree decomposition has width 2.

3.2 Tree Decomposition Algorithms

In order to find an optimal tree decomposition for a given graph, one can make use of a number of algorithms. One such approach, capable of computing an optimal tree decomposition is *bucket elimination*. A survey and implementation of this approach has been done in [McMahan, 2004]. Originally based on the *Adaptive Consistency* algorithm in [Dechter, 2003], it stems from solving constraint satisfaction problems (CSPs). These problems, represented by a hypergraph, are solved by computing a generalized hypertree decomposition (which generalizes tree decompositions) and then solving the original CSP problem on the decomposition. The central idea to obtain a tree decomposition by bucket elimination is the notion of an elimination ordering:

Definition 3.8. *An **elimination ordering** for a given graph $G = (V, E)$ is an ordering $\sigma = (v_1, \ldots, v_n)$ of the n vertices in V.*

Given a graph and an elimination ordering, the bucket elimination algorithm described in [McMahan, 2004] returns a tree decomposition. The general idea is to remove a vertex from the graph (placing it and its neighbors in a bag) and then connecting all the neighbors in the graph, in the end yielding a valid tree decomposition, after appropriate connections between the bags are made.

In general one tries to compute tree decompositions of small width. Therefore a tree decomposition whose width equals the treewidth of the graph would be optimal. The formal decision problem regarding treewidth is defined as follows:

Input: A graph $G = (V, E)$ and an integer $k \geq 1$.

Question: Does there exist a tree decomposition for graph G whose width is less than or equal to k?

Unfortunately the decision problem above is NP-complete [Arnborg et al., 1987] and therefore it is very hard–and therefore slow–to algorithmically obtain an "optimal" tree decomposition (with optimality being defined in terms of smallest width). From [Rose, 1972, Koster et al., 2001] it follows that there exists an elimination ordering such that the bucket elimination algorithm returns a tree decomposition of optimal width. However, as the optimal tree decomposition problem is NP-complete and, given an elimination ordering, the bucket elimination algorithm is polynomial, finding such an optimal elimination ordering must itself be NP-complete. This has also been shown in e.g. [Arnborg et al., 1987].

In the light of these negative complexity results, heuristic methods are usually used to efficiently compute a near-optimal tree decomposition. Such heuristic methods can among other papers be found in [Bachoore and Bodlaender, 2005, Schafhauser, 2006, Dermaku et al., 2008, Bodlaender and Koster, 2010].

3.2.1 Elimination Orderings and the Bucket Elimination Algorithm

The heuristics described here make use of the concept of *elimination orderings* as discussed above. To give a more accurate description over the bucket elimination algorithm that lies at the heart of obtaining a tree decomposition as discussed earlier, we need an alternative (but equivalent) definition of elimination orderings.

Definition 3.9. *Given a graph $G = (V, E)$ with (ordered) vertices $\{v_1, \ldots, v_n\}$, an **elimination ordering** is a bijective function $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$.*

That is, an elimination ordering is a permutation function, resulting in a reordering of the vertices of a graph G. We naturally extend the function σ to vertices, where, if $\sigma : i \mapsto j$, then $\sigma : v_i \mapsto v_j$. If for two vertices v_i and v_j we have that $\sigma(i) < \sigma(j)$, then we say that v_i is *lexicographically smaller* than v_j and v_j is *lexicographically bigger* than v_i.

The algorithm that creates a tree decomposition given an elimination ordering is called *bucket elimination* and works in the following way:

Initially a bucket (of vertices) is created for each vertex in the graph G. For each edge in G, we put both incident vertices into the bag of the lexicographically smaller one.

After this preparatory step the buckets are processed in the order given by the elimination ordering σ. When the bucket for a node v_i (lets call it B_{v_i}) is processed, the set $R = B_{v_i} \setminus \{v_i\}$ is copied to the bucket of the lexicographically smallest vertex (say v_j) in R. Finally the buckets B_{v_i} and B_{v_j} are connected by an edge.

After processing every bucket according to the elimination ordering, a tree decomposition is obtained from the buckets as bags and the edges between the buckets as the edges in the tree decomposition.

Finding an elimination ordering that, with the algorithm outlined above, yields an optimal tree decomposition with respect to width is always possible according to [Clautiaux et al., 2004], as at least one such optimal elimination ordering must always exist.

Other methods than the bucket elimination algorithm are known and can be used to obtain a tree decomposition given an elimination ordering, but subsequently we assume that bucket elimination is used.

In the following we give a short overview over heuristics that have proven to be usable in practice.

3.2.2 Heuristics for Elimination Orderings

In light of the intractability result for obtaining a "perfect" elimination ordering, we resort to approximation in order to guarantee good runtime behavior. A multitude of heuristics (and also exact algorithms) are available to obtain good elimination orderings for a given graph. An overview of these methods can be found in [Schafhauser, 2006]. The algorithm presented in chapter 6 does not itself specify how a tree decomposition shall be computed, several existing implementations of heuristic methods have been used for the implementation described in chapter 6.2. Therefore we will at this point give a quick overview of the methods used. A more accurate description can be found in e.g. [Schafhauser, 2006, Dermaku et al., 2008, Bodlaender and Koster, 2010].

Min-Fill Heuristics

The Min-Fill (MF) heuristics obtains an elimination ordering for graph G in the following way:

- Make a copy of G, call it G'.
- Order the vertices $v_1, \ldots, v_n$ as follows:
    1. Select the vertex v_i that introduces the minimum number of edges into G' when eliminated, and add it to the elimination ordering.
    2. Eliminate v_i from G' and introduce as many edges into G' as are needed so that all the neighbors of v_i form a clique in G'.
    3. If there are vertices left in G', go to step 1.
- Return the thus obtained elimination ordering for G.

Min-Induced-Width Heuristics

The Min-Induced-Width heuristics obtains an elimination ordering for G based on the following steps:

- Make a copy of G, name it G'.
- Order the vertices $v_1, \ldots, v_n$ as follows:
    1. In G', contract the edge between a minimum degree vertex v_i and one of its neighbors (say v_j) that has minimum degree with respect to the neighborhood of v_i.
    2. Add vertex v_i to the elimination ordering.
    3. If there are vertices left in G', go to step 1.
- Return the thus obtained elimination ordering for G.

Maximum Cardinality Search Heuristics

The Maximum Cardinality Search (MCS) heuristics obtains an elimination ordering for G based on the following steps:

- Make a copy of G, name it G'.
- Order the vertices $v_1, \ldots, v_n$ as follows:

1. In G', add the vertex v_i to the elimination ordering that has maximum cardinality in G'.
2. Eliminate v_i from G' using the same method as in the MF heuristics.
3. If there are vertices left in G', go to step 1.

- Return the thus obtained elimination ordering for G.

3.3 Working with Tree Decompositions

Once a tree decomposition for a given graph is obtained, one can then easily solve many problems while only considering the information currently available at the given node (or bag) in the tree decomposition. Usually a uniform approach is used (for normalized tree decompositions):

1. Process the nodes of the tree decomposition in a bottom-up manner, starting at the leaf nodes.
2. For each leaf node, consider all possible partial solutions that can be obtained from the vertices in the corresponding bag. Check that for each of these partial solutions, no problem constraint is violated, otherwise abandon the partial solution.
3. For each introduction or removal node, check whether the corresponding vertex can be introduced/removed, without violating problem constraints. If so, calculate all new partial solutions from the old one, otherwise abandon the partial solution.
4. For each branch node, merge matching partial solutions, that is, combine the partial solutions of the left and right subtree, if no problem constraints are violated by doing so.
5. At the root node, check all partial solutions for constraint violations. If there is one or more of them left after this final check, one or more solutions have been found.

The approach described above works mainly because of the defining conditions of tree decompositions: Because every two adjacent vertices must occur in some bag together, at some point the consequence of their relation in the graph will influence the (partial) solutions. Because of the connectedness condition, one knows that a vertex, once removed, will never be encountered again and therefore checking constraint violations for that vertex is already completed when the vertex is removed.

Based on this approach we get to the possibility of deriving fixed-parameter algorithms based on tree decompositions.

4 Parameterized Complexity

In this chapter the basic notions of parameterized complexity are introduced. The first section briefly introduces a few basic classical complexity classes. The following section deals with the motivation behind parameterizing problems and the implications that these parameters have for the runtime of algorithms. Later on the notion of fixed-parameter tractability is defined and fixed-parameter algorithms are briefly discussed.

4.1 Complexity Theory

In this section we briefly introduce the most basic notions of classical complexity theory in order to have the basis of a comparison to parameterized classes of complexity, as introduced in the following sections. A more comprehensive overview can be found in e.g. [Papadimitriou, 1993]. We start of with the defintion of the TIME and NTIME complexity classes, which are defined by the widely known and acknowledged concept of deterministic (resp. non-deterministic) Turing machines, which for brevity we do not restate here.

Definition 4.1. *For languages L and a function $t : \mathbb{N}^+ \to \mathbb{N}^+$, the complexity class* $\text{TIME}(t(n))$ *is defined as follows:*

$$\text{TIME}(t(n)) = \{L \mid \exists \textit{ deterministic Turing machine deciding } L \textit{ in } O(t(n)) \textit{ time.}\}$$

Definition 4.2. *For languages L and a function $t : \mathbb{N}^+ \to \mathbb{N}^+$, the complexity class* $\text{NTIME}(t(n))$ *is defined as follows:*

$$\text{NTIME}(t(n)) = \{L \mid \exists \textit{ non-deterministic Turing machine deciding } L \textit{ in } O(t(n)) \textit{ time.}\}$$

From these definitions we can now define the complexity classes P and NP:

Definition 4.3. *The complexity class **P** is defined as follows:*

$$P = \bigcup_{k=0}^{\infty} \text{TIME}(n^k)$$

where n represents the length of the input.

A language or (decision) problem that can be decided in deterministic polynomial time is therefore said to be a member of the complexity class P. Such a problem is called *tractable*.

Definition 4.4. *The complexity class **NP** is defined as follows:*

$$NP = \bigcup_{k=0}^{\infty} \text{NTIME}(n^k)$$

where n represents the length of the input.

A language or (decision) problem that can be decided in non-deterministic polynomial time is said to belong to the complexity class NP. If the problem can only be solved in non-deterministic polynomial time (or more), the problem is said to be *intractable*. If in finite time the problem cannot be decided at all it is said to be *undecidable*. The fundamental difference between non-deterministic and deterministic polynomial time is that in non-deterministic polynomial time, a multitude of polynomial-length computation paths can be explored at the same time, whereas in deterministic polynomial time only one such path can be explored. Therefore, in order to explore all the paths in deterministic Turing machines, one generally needs exponential time, as a non-deterministic computation path can, at every computation step, potentially split into multiple new computation paths.

One usually distinguishes between "easy-to-solve" (tractable) and "easy-to-verify" (intractable) problems, where "easy" stands for "in less than non-deterministic polynomial time". For example, given a propositional logic formula in conjunctive normal form, it is easy to verify that a given interpretation is indeed a model. However, calculating a model is not.

Note that by definition of the complexity classes P and NP, the following theorem clearly holds:

Theorem 4.5. $P \subseteq NP$

Proof. A deterministic Turing machine running in $O(t(n))$ time has an equivalent non-deterministic Turing machine running in $O(t(n))$ time. □

Definition 4.6. *The complexity class Σ_i^P is defined inductively as follows:*

$$\Sigma_0^P = P$$

$$\Sigma_{i+1}^P = NP^{\Sigma_i^P}$$

where NP^C is the set of decision problems decidable in polynomial time by a non-deterministic Turing machine that has access to an oracle for complexity class C.

A Turing machine with an oracle for some complexity class C can in one computation step obtain an answer for a decision problem in class C.

In order to define the relation between complexity classes, we define here the concept of reducibility.

Definition 4.7. *Let $L_1, L_2 \subseteq \Sigma^*$ be two languages (or problems). We say that L_1 **reduces** to L_2 by a polynomial (many-one-)reduction if and only if there is a function $R : \Sigma^* \to \Sigma^*$ such that*

- *For any $x \in \Sigma^*$, $R(x)$ is computable in polynomial time in the size of x*
- *$x \in L_1$ if and only if $R(x) \in L_2$*

Definition 4.8. *A language or problem L is said to be **hard** for a complexity class C if every problem $X \in C$ can be reduced to L.*

Starting with the complexity class P, for a complexity class C it follows that, by the above definition, no problem in C can be harder than L, as an algorithm for L can solve any problem in C. We now define the central concept of completeness:

Definition 4.9. *If a language or problem L is a member of a complexity class C (i.e. $L \in C$), and if L is also hard for C, then L is said to be **complete** for C.*

4.2 Fixed-Parameter Tractability

In this section we give a short overview over parameterized complexity theory and its core concepts. The interested reader can find good disquisitions on the matter in [Downey and Fellows, 1999, Flum and Grohe, 2006, Niedermeier, 2006].

4.2.1 Why parameterization?

In traditional complexity theory, the single property that is examined about a problem is its input size: For a problem in NP, a problem instance of size n can be decided by a nondeterministic Turing machine in an amount of time that is polynomial in n. Therefore the only "parameter" that classical complexity theory knows is the problem size. In practice there are many computationally hard (i.e. intractable) problems that have to be solved. It turns out however, that, even though the problem is known to be intractable in general, there are a number of instances for which solving the problem is easy. But still classical complexity theory tells us that the problem is hard. Parameterized complexity tries to deal with this seemingly contradictory phenomenon.

Unless $P = NP$, in order to solve NP-hard problems algorithms need exponential running time. The idea of parameterized complexity is to find problem-specific parameters of problems, such that the running time of a suitable algorithm can be expressed (in

big-O notation) as a function of the parameter and the input size, with the underlying idea that the exponentiality of the running time can be restricted to the parameter and only the parameter. For relatively "small" values of the parameter, solving the problem can then be very efficiently done. If such a parameterization exists the problem is said to be *fixed-parameter tractable*, as bounding the parameter by some constant yields a polynomial-time algorithm.

4.2.2 A Parameterized Complexity Class

Fixed-parameter tractability (or FPT, for short) is a complexity class defined as follows (cf. [Downey and Fellows, 1999]):

Definition 4.10.

$$\text{FPT} = \{L \mid \exists \textit{ deterministic Turing machine deciding } L \textit{ in } f(k) \cdot n^{O(1)} \textit{ time}\}$$

where n is the input size, k is a problem-specific parameter and $f(k)$ is some computable function depending only on k.

The main idea of this definition is to exclude problems with runtime n^k, for a parameter k: The runtime in terms of the input size n only depends on the input size and not on the parameter.

Consider the problem SATISFIABILITY:

Input: A boolean formula F.
Question: Does there exist a truth assignment for the variables in F so that F evaluates to true?

Example 4.11. *The formula*

$$(x \vee y) \wedge (x \vee \neg y)$$

is satisfiable, e.g. by a truth assignment assigning x the value **true** *and y the value* **false**.

It is known by Cook's theorem (cf. [Cook, 1971]), that the SATISFIABILITY problem is NP-complete and therefore intractable in general. However in practice it turns out that many instances of the problem are much easier than one would expect in light of this intractability result. Therefore it is natural to ask if more about the complexity of the problem can be learned by studying different parameterizations.

One such parameter is the number of variables k used in the formula of length $n = |F|$. Independent of the length of the formula, there can only be a maximum number of 2^k truth assignments for the k variables in the formula. This immediately leads to an

algorithm that checks (in the worst case) for all 2^n truth assignments whether F evaluates to true. Such an algorithm has a runtime of $O(2^k \cdot n)$ which gives us fixed-parameter tractability for the SATISFIABILITY problem. Thus, if we consider the number of variables in a formula as fixed, we essentially obtain a linear-time algorithm for the problem (similar results can be obtained for parameterization by the length of the formula, or, if the formula is in conjunctive normal form, by the number of clauses). Note however that the above fixed-parameter tractability result for parameterization by the number of variables is only an illustrative example as it is extremely trivial.

Fixed-parameter algorithms are thus developed to exploit the parameterization of a problem in such a way that, when the parameter is of small value, the runtime of the algorithms are favorable. This means that efficient algorithms for solving the problems can be found, as long as the parameter is small. Another well-known approach to get efficient algorithms for problems is the use of approximation algorithms or heuristics. However, there are advantages to using fixed-parameter algorithms:

- Fixed-parameter algorithms always yield optimal solutions.
- Fixed-parameter algorithms have provable upper bounds on the computational complexity.

Both of the above statements are, in general, not true for heuristics-based algorithms. However, fixed-parameter algorithms also have a disadvantage and that is the exponential runtime with regard to the parameter. Therefore, fixed-parameter algorithms are only efficient as long as the parameter is of a small value, otherwise the exponentiality quickly becomes prohibitive.

One well-known and very natural parameter for optimization problems is the size of the solution. For example the classical definition of the VERTEX COVER problem is the following:

Input: A graph $G = (V, E)$.

Task: Find a minimal subset of vertices $C \subseteq V$ such that each edge in E is incident to at least one vertex in C.

We can now parameterize the VERTEX COVER problem by its solution set size k. That means that we consider the size of the solution as part of the input of the problem. The definition of the parameterized version of the VERTEX COVER problem then becomes the following:

Input: A graph $G = (V, E)$ and a non-negative integer k.

Task: Find a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E is incident to at least one vertex in C.

Observe that the difference to the classical definition is that the parameter (i.e. in this case the intended solution set size) is known beforehand. With this parameterization, VERTEX COVER is known to become fixed-parameter tractable (as originally shown in [Nemhauser and Trotter, 1975]). Note however that this parameterization assumes that some a-priori knowledge about the solution of the problem is available (i.e. the solution set size). We will explore another wide-spread parameterization in section 4.3.

By a well-known result in [Bodlaender, 1993a], finding a tree decomposition with minimal treewidth is itself fixed-parameter tractable when parameterized by solution size (i.e. bu the treewidth). Recall that this is in general an NP-complete problem. However, this result does not provide us with an efficient algorithm, as the constant factors hidden in the big-O notation are quite big. Therefore, simply establishing fixed-parameter tractability does not necessarily lead to an efficient algorithm.

4.3 Parameterization by Treewidth

As we have seen in chapter 3, given a tree decomposition T, we can for certain problems obtain algorithms that only use local information at the individual nodes in T to decide the problem. Using a dynamic programming approach on the tree decomposition (that is, discarding computation paths as early as possible–i.e. as soon as a problem constraint violation is discovered) the solution to a (decision) problem can be obtained on the tree decomposition because of its defining properties.

A simple example (based on a normalized tree decomposition) in this context would be VERTEX COVER:

- At each leaf node containing the vertices $v_1, \ldots, v_l$ where k is the treewidth and $l \leq k + 1$ consider all the possible combinations of these vertices (i.e. "guess") which vertices to place in the vertex cover and which ones not. If there is an edge between any of the vertices v_i in the current bag, discard any combination of vertices that do not cover that edge. For each combination count the vertices in the vertex cover and associated that count with the combination.

- At each introduction node (introducing vertex v), for each combination of its child node create two new copies. To one copy, add v and increase the count by 1. For both cases individually, check that all edges in the bag are covered.

- At each removal node, simply remove the node from all combinations. If two combinations coincide, associate the minimum count of the two with the resulting combination.

- At each join node, simply join coinciding combinations of the two child nodes. Associate with the resulting combination the sum of the two counters minus the

overlapping vertices (that is, the vertices of the combination that are currently visible in the bag).

- After processing the root node, check if there is a combination with its count smaller than the requested size of the vertex cover. If so, answer "yes", otherwise answer "no".

The above algorithm is a fixed-parameter algorithm for vertex cover when the problem is parameterized by treewidth: Given a tree decomposition of bounded width k, it takes a linear amount of time (in the size of the graph) to visit each node of the tree decomposition. At every node an exponential number of combinations has to be generated and checked, however the number of combinations is only exponential in the width of the tree decomposition, giving us an algorithm with runtime $2^k \cdot O(n)$, which shows fixed-parameter tractability.

For many problems that can be represented as a whole or in part as a graph such an approach is feasible. Examples include many problems on graphs (e.g. VERTEX COVER, DOMINATING SET, MULTICUT, see [Niedermeier, 2006, Pichler et al., 2010] for more) and other problems like answer set programming (as presented in this thesis and also in e.g. [Jakl et al., 2009, Morak et al., 2010]), argumentation problems (as found for example in [Dvorák et al., 2010]), belief revision (e.g. [Pichler et al., 2009]) and many more. Research in this area is very active and there surely will be more interesting results in this area in the future.

4.3.1 Courcelle's Theorem

One very important aspect of parameterization by treewidth is that, in conjunction with monadic second order logic (as introduced in section 2.3), fixed-parameter tractability can be shown fairly easily for this parameter. First, we extend the interpretations used for MSO formulas to graphs.

Making a slight extension to MSO syntax we provide the definition below. We introduce the formula $xy \in E$ to express that $(x, y) \in E$, i.e. that there is an edge between x and y.

Definition 4.12. *We say that a graph $G = (V, E)$ satisfies a given MSO formula, or*

$$G \models \varphi$$

*if and only if all interpretations whose domains D consist of all the vertices and edges in the graph (i.e. $D = V \cup E$) and whose interpretation functions reflect the edges in G (i.e. $xy \in E =$ **true** if and only if $(x, y) \in E$) satisfy φ.*

Given this definition we can now state the core result in this field due to Bruno Courcelle:

Theorem 4.13. *[Courcelle, 1990] Let $k \geq 1$ and let φ be an MSO formula. Then, given a graph G and a tree decomposition of width at most k, there is linear-time algorithm that decides whether*

$$G \models \varphi.$$

MSO formulas in conjunction with a graph structure can be used to establish fixed-parameter tractability with respect to treewidth as the parameter. However this result is seemingly only of theoretical importance as a classification tool due to the fact that algorithms derived directly from this approach usually have very big factors hidden in the big-O notation and are therefore unusable in practice.

In order to illustrate this, we provide the following example:

Example 4.14. *Given a graph $G = (V, E)$, the fact that G is bipartite can be expressed using the following MSO-formula:*

$$\exists X \exists Y \forall x (x \in V \rightarrow (x \in X \vee x \in Y)) \wedge$$

$$\forall x \forall y ((xy \in E) \rightarrow \neg((x \in X \wedge y \in X) \vee (x \in Y \wedge y \in Y)))$$

and thereby establish fixed-parameter tractability for deciding whether a given graph is bipartite or not, with respect to the parameter treewidth.

III Main Results

5 The Algorithm

In this chapter we outline a novel algorithm ("dynASP") for solving head-cycle free disjunctive logic programs, using a dynamic-programming approach based on a tree decomposition of the logic program.

The first section outlines methods to represent logic programs as graphs. Section 5.2 contains the formal definition of the algorithm and the outline of the corresponding correctness proof.

5.1 Tree Decompositions of Answer Set Programs

In order to utilize tree decompositions for dynamic-programming algorithms, the problem (or a core part of it) has to be representable as a graph. For logic programs, we already informally examined the concept of the dependency graph in section 2.4.2 in order to provide an alternative definition of head-cycle freeness in disjunctive logic programs. However this concept turns out to be too weak to use for a tree decomposition-based algorithm. This is due to the fact that program rules are only implicitly represented in the dependency graph. However in order to evaluate logic programs, it is convenient to have an explicit representation of the rules as entities. To include these, we here define the concept of *incidence graphs*.

Definition 5.1. *The **incidence graph** of a disjunctive logic program $\Pi = (\mathcal{A}, \mathcal{R})$ with atoms $\mathcal{A}$ and rules $\mathcal{R}$ is a bipartite graph $G = (V, E)$. G is constructed in the following way:*

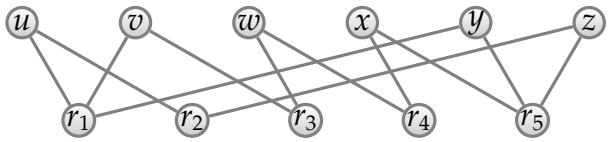
- *For every atom $a \in \mathcal{A}$, add a vertex a to V.*
- *For every rule $r \in \mathcal{R}$, add a vertex r to V.*
- *For every rule $r \in \mathcal{R}$ and atom a occurring in the rule, add an edge (a, r) to E.*

Example 5.2 illustrates the concept of a incidence graph for a specific program. Therefore we take another look at the program in Example 2.35.

Example 5.2. *Let P be the program from Example 2.35. We re-state it here for convenience:*

$$r_1 = u \leftarrow v, y; \quad r_2 = z \leftarrow u; \quad r_3 = v \leftarrow w;$$
$$r_4 = w \leftarrow x; \quad r_5 = x \leftarrow \neg y, \neg z.$$

From this program we construct the incidence graph as defined in Definition 5.1:



With the concept of the incidence graph in hand, we can now create a tree decomposition representing a disjunctive logic program. An example for such a tree decomposition of the program in Example 5.2 is already provided in Example 3.3.

Definition 5.3. *A tree decomposition $\mathcal{T} = (\chi, T)$ of a logic program Π is a tree decomposition as defined in Definition 3.2, whereby $\mathcal{T}$ is a tree decomposition of the incidence graph of Π.*

Given such a tree decomposition, it can now intuitively be seen that the principles for developing algorithms on tree decompositions as laid out in section 4.3 can now be applied to logic programs with a tree decomposition. Informally, the algorithm follows the following line of thought:

For each node in the tree decomposition, only consider a part of the program, namely the part that is induced by the subtree rooted at that node. Consider all possible partial models in the bag. For each of them, check whether it is an answer set of the reduced program. If not, disregard that computation path. If yes, continue with this computation path to the next tree node. Note that partial models are represented only by the atoms and rules occurring in the bag of a node. However, implicitly they also contain knowledge about the subtree rooted at that node, hence they represent a partial model of that whole subtree.

In normalized tree decompositions of logic programs, we now distinguish between ATOM INTRODUCTION and RULE INTRODUCTION and respectively, ATOM REMOVAL and RULE REMOVAL nodes, depending on what type of vertex gets introduced or removed.

5.2 Formal Definition and Correctness

The algorithm described in this section is based on an alternative characterization of answer sets for head-cycle free disjunctive logic programs. We therefore begin with this new characterization.

5.2.1 Alternative Characterization of Answer Sets

In the following, let $\Pi = (\mathcal{A}, \mathcal{R})$ be a head-cycle free disjunctive logic program with atoms $\mathcal{A}$ and rules $\mathcal{R}$. Recall Theorem 2.30 which provided a characterization of answer sets for head-cycle free disjunctive logic programs. From this characterization, the following corollary characterization immediately follows:

Corollary 5.4. *Let $\Pi = (\mathcal{A}, \mathcal{R})$ be a grounded head-cycle free disjunctive logic program. Then, a set $M \subseteq \mathcal{A}$ is an answer set of Π if and only if the following conditions hold:*

1. *S satisfies each rule $r \in \mathcal{R}$,*
2. *There exists a partial order $\trianglelefteq \subseteq (\mathcal{A} \cup \mathcal{R})^2$ such that for each atom $a \in M$, there exists a rule $r \in \mathcal{R}$ where*
   a) *M satisfies $B(r)$,*
   b) *$H(r) \cap M = \{a\}$,*
   c) *for each atom $b \in B^+(r)$, $b \trianglelefteq r$, and*
   d) *$r \trianglelefteq a$.*

Proof. Let $M \subseteq \mathcal{A}$ satisfy both conditions above and let $\trianglelefteq$ be the partial order from Condition 2. By removing all rules from $\trianglelefteq$, we construct a new partial order $\trianglelefteq' \subseteq \mathcal{A} \times \mathcal{A}$. Then, M clearly satisfies the new condition

2'. there exists a partial order $\trianglelefteq' \subseteq \mathcal{A} \times \mathcal{A}$ such that, for each atom $a \in M$, there exists a rule $r \in \mathcal{R}$ such that
   a) M satisfies $B(r)$,
   b) $H(r) \cap M = \{a\}$,
   c) for each atom $b \in B^+(r)$, $b \trianglelefteq' a$.

Now let the total order $\leq$ on set $\mathcal{A}$ be any linear extension of $\trianglelefteq'$. It is then clearly possible to find a function $f : \mathcal{A} \to \{0, \ldots, |\mathcal{A}|\}$ such that $\forall a, b \in \mathcal{A} : f(a) \leq f(b) \Leftrightarrow a \leq b$. Then f satisfies Condition 2 of Theorem 2.30 and therefore M is an answer set of Π.

Conversely, suppose that M is an answer set of Π. By Theorem 2.30 there exists a function f satisfying Condition 2. This function gives rise to a total order $\trianglelefteq'$ on $\mathcal{A}$. Since every total order is also a partial order, $\trianglelefteq'$ satisfies Condition 2' above.

We now construct the partial order $\trianglelefteq \subseteq (\mathcal{A} \cup \mathcal{R})^2$ by extending $\trianglelefteq'$ in the following way: For each $a \in M$ let $r \in \mathcal{R}$ be one of the rules satisfying Condition 2'. Then we add (r, a) and for all $b \in B^+(r)$ the pair (b, r) to $\trianglelefteq$. Now $\trianglelefteq$ satisfies Condition 2 of this corollary and therefore M satisfies all conditions, which clearly follows from Theorem 2.30. □

Given this corollary, the following clearly follows as well:

Corollary 5.5. *Let $\Pi = (\mathcal{A}, \mathcal{R})$ be a grounded head-cycle free disjunctive logic program. Then, a set $M \subseteq \mathcal{A}$ is an answer set of Π if and only if the following conditions hold:*

1. *M satisfies each rule $r \in \mathcal{R}$,*
2. *There exists a set $\rho \subseteq \mathcal{R}$ and a partial order $\trianglelefteq \subseteq (M \cup \rho)^2$ such that, $M = \bigcup_{r \in \rho}(H(r) \cap M)$ and for all $r \in \rho$*
   a) *M satisfies $B(r)$,*
   b) *$|H(r) \cap M| = 1$,*
   c) *for each atom $b \in B^+(r)$, $b \trianglelefteq r$, and*
   d) *for each atom $a \in H(r) \cap S$, $r \trianglelefteq a$.*

Proof. Let $M \subseteq \mathcal{A}$ satisfy both conditions above and let ρ be the set of rules and $\trianglelefteq$ be the partial order of Condition 2. $\trianglelefteq$ is also a partial order over the set $\trianglelefteq \subseteq (\mathcal{A} \cup \mathcal{R})^2$ which is needed in Condition 2 of Corollary 5.4. Since $M = \bigcup_{r \in \rho}(H(r) \cap M)$ there clearly exists for each $a \in M$ at least one rule $r \in \rho$ such that $a \in H(r)$. From $|H(r) \cap M| = 1$ it follows that $H(r) \cap M = \{a\}$ and therefore r satisfies Condition 2 of Corollary 5.4. Hence, M is an answer set of Π.

Now suppose that M is an answer set of Π. By Corollary 5.4 there exists a partial order $\trianglelefteq$ satisfying Condition 2 of Corollary 5.4. Furthermore, for each atom $a \in S$ there exists a rule $r \in \mathcal{R}$ satisfying this condition. Let $\rho \subseteq \mathcal{R}$ be the set containing exactly these rules (i.e. $\left|\rho\right| = |M|$). Hence, for each $a \in M$ there exists exactly one $r \in \rho$ with $H(r) \cap M = \{a\}$ and therefore $M = \bigcup_{r \in \rho}(H(r) \cap S)$. Now let $\trianglelefteq'$ be the partial order obtained by restricting $\trianglelefteq$ to the set $M \cup \rho$. Then ρ and $\trianglelefteq'$ clearly satisfy Condition 2 of Corollary 5.5 and hence, M satisfies all conditions of this corollary. □

For the characterization of answer sets that we provide in Theorem 5.7, we first need the following definition:

Definition 5.6. *For a given head-cycle free disjunctive logic program* $\Pi = (\mathcal{A}, \mathcal{R})$*, let* $M \subseteq \mathcal{A}$ *and* $\rho \subseteq \mathcal{R}$*. Then the* ***derivation graph*** $G = (V, E)$ *induced by* M *and* ρ *is given by* $V = M \cup \rho$ *and* E *is the transitive closure of the edge set* $E' = \{(b, r) : r \in \rho, b \in B^+(r) \cap M\} \cup \{(r, a) : r \in \rho, a \in H(r) \cap M\}$.

With this definition, we can now state a novel, alternative characterization for answer sets in head-cycle free disjunctive logic programs:

Theorem 5.7. *Let* $\Pi = (\mathcal{A}, \mathcal{R})$ *be a grounded head-cycle free disjunctive logic program. Then, a set* $M \subseteq \mathcal{A}$ *is an answer set of* Π *if and only if the following conditions hold:*

1. *M satisfies each rule $r \in \mathcal{R}$,*

2. *There exists a set $\rho \subseteq \mathcal{R}$ such that,*

   - $M = \bigcup_{r \in \rho}(H(r) \cap M)$,
   - *the derivation graph induced by M and ρ is acyclic,*

   and for all $r \in \rho$

   a) *M satisfies $B(r)$, and*

   b) $|H(r) \cap M| = 1$.

Proof. Let $M \subseteq \mathcal{A}$ satisfy the conditions above. Let ρ be the set of rules of Condition 2 and let $G = (V, E)$ be the derivation graph induced by M and ρ. Since G is acyclic and E is transitively closed, it gives rise to a partial order $\trianglelefteq = \subseteq V^2 = (S \cup \rho)^2$ with $a \trianglelefteq b \Leftrightarrow (a, b) \in E$. From the construction of E in Definition 5.6, it immediately follows that ρ and $\trianglelefteq$ satisfy Condition 2 of Corollary 5.5. Hence, M satisfies both conditions of that corollary and is therefore an answer set of Π.

Now suppose that M is an answer set. By Corollary 5.5 there exists a set ρ and a partial order $\trianglelefteq$ satisfying Condition 2 of that corollary. Let $G' = (V', E')$ be given by $V' = M \cup \rho$ and $E' = \trianglelefteq$. Since $\trianglelefteq$ is a partial order, G' is acyclic. Let $G = (V, E)$ be the derivation graph induced by M and ρ. Then clearly $V = V'$ and $E \subseteq E'$, therefore G is acyclic as well. Hence, ρ satisfies Condition 2 of Theorem 5.7 and M, by Corollary 5.5, satisfies both conditions. □

From this point on, we shall refer to the set ρ as the *proof set* for a given answer set M.

5.2.2 Algorithm Description

The algorithm for evaluating a head-cycle free disjunctive logic program is given below in form of a rule-based program. In order to simplify notation we extend the *element-of* relation $\in$ to graphs in the following way: $(x, y) \in G$ is true if graph G contains the edge (x, y). $x \in G$ is true if graph G contains the vertex x.

Before restating the definition in full we briefly sketch the semantics of the used predicates in the rule-based program that describe the semantics and represent the tree decomposition.

Without loss of generality, we assume here that the tree decomposition is normalized (see Definition 3.5) and has empty leaf and empty root nodes. This can easily be achieved by append a sufficient number of removal nodes to the leafs and prepend a sufficient number of removal nodes to the root node. Clearly this does not lead to an increase in width.

- The $\mathfrak{T}(n, G, S, D_A, D_R)$ predicate represents a tuple for a given node n. Each tuple represents a possible partial interpretation (i.e. a partial answer set) and consists of a derivation graph G, a set of satisfied rules S, a set of derived atoms D_A and a set of rules that have already derived an atom D_R. The partial interpretation is implicitly represented by the graph: Each atom that is a vertex in the graph is true in the interpretation.
- The $*\mathrm{err}(n, G, S, \dots)$ predicates represent an error/abort condition. If such a predicate can be derived for a tuple, the tuple is discarded (i.e. the partial interpretation represented by the tuple turned out not to be a valid partial answer set).
- $[\![G]\!]$ represents the transitive closure of a graph G and $G_1 \cup G_2$ is a graph union, where the resulting graph consists of the vertices and edges of both G_1 and G_2.
- The $\mathrm{child}(n, n')$ and $\mathrm{children}(n, n_1, n_2)$ predicates represent the structure of the tree decomposition, where n is the parent and n', n_1, n_2 are its immediate children.
- The $\mathrm{bag}(n, x)$ predicate represents the bags of a given node n in the tree decomposition. It can be derived if x is contained in the bag of node n.
- The $\mathrm{acyclic}(G)$ predicate can be derived, if the graph G is acyclic.
- The notational aids $H(r), B^+(r), B^-(r)$ for rules that were introduced in section 2.4.1 correspond directly to the predicates H, B^+ and B^- in the logic program.

The following paragraphs state the definition of the dynASP algorithm in detail. The rule-based program is split into six parts, one for each node type in the tree decomposition.

As input, the algorithm accepts a tuple $\langle \mathcal{T}, \Pi \rangle$ consisting of a head-cycle free disjunctive logic program Π and a tree decomposition $\mathcal{T}$ of that program. Tree Decomposition and logic program are encoded as ground facts using the predicates described above.

Definition 5.8. *The **dynASP algorithm** does a bottom-up traversal of the tree decomposition in the following manner:*

Leaf node:

$$\mathfrak{T}(n, G, S, D_A, D_R) \leftarrow \text{leaf}(n), G = (\emptyset, \emptyset), S = \emptyset, D_A = \emptyset, D_R = \emptyset.$$

Atom removal node:

$$\begin{array}{lll}
\mathfrak{T}(n, G \setminus \{x\}, S, D_A \setminus \{x\}, D_R) & \leftarrow & \text{ar}(n, x), \text{child}(n, n'), \mathfrak{T}(n', G, S, D_A, D_R), x \in G, x \in D_A. \\
\mathfrak{T}(n, G, S, D_A, D_R) & \leftarrow & \text{ar}(n, x), \text{child}(n, n'), \mathfrak{T}(n', G, S, D_A, D_R), x \notin G.
\end{array}$$

Rule removal node:

$$\begin{array}{lll}
\mathfrak{T}(n, G \setminus \{r\}, S, D_A, D_R \setminus \{r\}) & \leftarrow & \text{rr}(n, r), \text{child}(n, n'), \mathfrak{T}(n', G, S, D_A, D_R), r \in G, r \in D_R. \\
\mathfrak{T}(n, G, S \setminus \{r\}, D_A, D_R) & \leftarrow & \text{rr}(n, r), \text{child}(n, n'), \mathfrak{T}(n', G, S, D_A, D_R), r \in S.
\end{array}$$

Rule introduction node:

$$\begin{array}{lll}
\mathfrak{T}(n, G, S \cup \{r\}, D_A, D_R) & \leftarrow & \text{ri}(n, r), \text{child}(n, n'), \mathfrak{T}(n', G, S, D_A, D_R), \text{bag}(n', x), x \in G, x \in B^-(r). \\
\mathfrak{T}(n, G, S \cup \{r\}, D_A, D_R) & \leftarrow & \text{ri}(n, r), \text{child}(n, n'), \mathfrak{T}(n', G, S, D_A, D_R), \text{bag}(n', x), x \in G, x \in H(r). \\
\mathfrak{T}(n, G, S \cup \{r\}, D_A, D_R) & \leftarrow & \text{ri}(n, r), \text{child}(n, n'), \mathfrak{T}(n', G, S, D_A, D_R), \text{bag}(n', x), x \notin G, x \in B^+(r). \\
\mathfrak{T}(n, [\![G']\!], S, D'_A, D'_R) & \leftarrow & \text{ri}(n, r), \text{child}(n, n'), \mathfrak{T}(n', G, S, D_A, D_R), \text{not rierr}(n', G, S, r), \\
 & & \text{riadapt}(G, D_R, D_A, r, G', D'_A, D'_R), \text{acyclic}(G'). \\
\text{rierr}(n, G, S, r) & \leftarrow & \text{bag}(n, x), x \in G, x \in B^-(r). \\
\text{rierr}(n, G, S, r) & \leftarrow & \text{bag}(n, x), x \notin G, x \in B^+(r). \\
\text{rierr}(n, G, S, r) & \leftarrow & \text{bag}(n, x), \text{bag}(n, y), x \neq y, x \in G, y \in G, x \in H(r), y \in H(r). \\
\text{riadapt}(G, D_A, D_R, r, G', D'_A, D'_R) & \leftarrow & G = (V, E), G' = (V', E'), V' = V \cup \{r\}, \\
 & & E' = E \cup \{(x, r) | x \in V, x \in B^+(r)\} \cup \{(r, x) | x \in V, x \in H(r)\}, \\
 & & D'_R = D_R \cup \{r' | r' = r, (r', x) \in E'\}, D'_A = D_A \cup \{x | x \in V, (r, x) \in E'\}.
\end{array}$$

Atom introduction node:

$$\begin{array}{lll}
\mathfrak{T}(n, G, S', D_A, D_R) & \leftarrow & \text{ai}(n, a), \text{child}(n, n'), \mathfrak{T}(n', G, S, D_A, D_R), \text{not ainerr}(n', G, S, a), \\
 & & \text{ainadapt}(G, S, a, S'). \\
\mathfrak{T}(n, [\![G']\!], S', D'_A, D'_R) & \leftarrow & \text{ai}(n, a), \text{child}(n, n'), \mathfrak{T}(n', G, S, D_A, D_R), \text{not aiperr}(n', G, S, D_R, a), \\
 & & \text{aipadapt}(G, S, D_A, D_R, a, G', S', D'_A, D'_R), \text{acyclic}(G'). \\
\text{ainerr}(n, G, S, a) & \leftarrow & \text{bag}(n, r), r \in G, a \in B^+(r). \\
\text{aiperr}(n, G, S, D_R, a) & \leftarrow & \text{bag}(n, r), r \in G, a \in B^-(r). \\
\text{aiperr}(n, G, S, D_R, a) & \leftarrow & \text{bag}(n, r), r \in G, r \in D_R, a \in H(r). \\
\text{ainadapt}(G, S, a, S') & \leftarrow & S' = S \cup \{r | r \notin G, a \in B^+(r)\}. \\
\text{aipadapt}(G, S, D_A, D_R, a, G', S', D'_A, D'_R) & \leftarrow & G = (V, E), G' = (V', E'), S' = S \cup \{r | r \notin G, a \in B^-(r)\}, V' = V \cup \{a\}, \\
 & & E' = E \cup \{(a, r) | r \in V, a \in B^+(r)\} \cup \{(r, a) | r \in V, a \in H(r)\}, \\
 & & D'_A = D_A \cup \{a' | a' = a, (r, a) \in E'\}, D'_R = D_R \cup \{r | r \in V, (r, a) \in E'\}.
\end{array}$$

Branch node:

$$\begin{array}{lll}
\mathfrak{T}(n, [\![G]\!], S, D^l_A \cup D^r_A, D^l_R \cup D^r_R) & \leftarrow & \text{branch}(n), \text{children}(n, n_l, n_r), \mathfrak{T}(n_l, G_l, S_l, D^l_A, D^l_R), \mathfrak{T}(n_r, G_r, S_r, D^r_A, D^r_R), \\
 & & \text{agree}(G_l, G_r, D^l_R, D^r_R), G = G_l \cup G_r, S = S_l \cup S_r, \text{acyclic}(G). \\
\text{agree}(G_1, G_2, D_1, D_2) & \leftarrow & G_1 = (V_1, E_1), G_2 = (V_2, E_2), V_1 = V_2, \\
 & & \{r | r \in D_1, r \in D_2, (r, a) \in G_1, (r, a) \in G_2, a \in H(r)\} = D_1 \cap D_2.
\end{array}$$

5.2.3 Correctness

In order to prove the correctness of the algorithm given in Definition 5.8, we introduce the following notation: For a tree decomposition $\mathcal{T} = (\chi, T)$ with tree T rooted at node n_{root}, we denote that a node n is part of T by writing $n \in T$. Furthermore we use the shorthands $\chi_{\mathcal{A}}(t) = \chi(t) \cap \mathcal{A}$ and $\chi_{\mathcal{R}}(t) = \chi(t) \cap \mathcal{R}$ to denote the atoms and rules in a bag respectively. We denote by T_n the subtree of T rooted at node $n \in T$. For a logic program Π, we denote by $\Pi_n = (\mathcal{A}_n, \mathcal{R}_n)$ the subprogram induced by T_n, i.e., $\mathcal{A}_n = \bigcup_{t \in T_n}(\chi_{\mathcal{A}}(t))$ and $\mathcal{R}_n = \{r^n : r \in \bigcup_{t \in T_n}(\chi_{\mathcal{R}}(t))\}$, where r^n denotes the rule r after removing every head and body atom not occurring in $\mathcal{A}_n$.

The algorithm contains the crucial predicate $\mathfrak{T}(n, G, S, D_A, D_R)$ with the following intended meaning: For a program $\Pi = (\mathcal{A}, \mathcal{R})$, n denotes a node in the tree decomposition of Π. $G = (V, E)$ is a derivation graph where $V \subseteq \chi(n)$ and V represents a partition of $\chi(n)$, such that every atom in V is part of an answer set and every rule in V is part of a corresponding proof set. S and D_R denote subsets of $\chi_{\mathcal{R}}(n)$. D_A denotes a subset of $\chi_{\mathcal{A}}(n)$. For all values of n, G, S, D_A, D_R, the ground fact $\mathfrak{T}(n, G, S, D_A, D_R)$ shall be true for a given HCF logic program Π, if and only if the following property holds:

Property 1. *There exists an extension V' of the partition V to $\mathcal{A}_n \cup \mathcal{R}_n$ such that $V' \cap \chi(n) = V$, such that the following conditions hold for the sets $M = V' \cap \mathcal{A}_n$ of atoms and $\rho = V' \cap \mathcal{R}_n$ or rules:*

1. *M satisfies each rule $r \in (\mathcal{R}_n \setminus \chi_{\mathcal{R}}(n)) \cup D_R \cup S$*
2. *$D_A \cup (M \setminus \chi_{\mathcal{A}}(n)) = \bigcup_{r \in \rho}(H(r^n) \cap M)$*
3. *The derivation graph G induced by M and ρ is acyclic,*
4. *for all $r \in \rho$*
   - *M satisfies the body of r^n*
   - *$|H(r^n) \cap M| \leq 1$*
5. *for all $r \in D_R \cup (\rho \setminus \chi_{\mathcal{R}}(n))$, $|H(r^n) \cap M| = 1$*

From this point onward, we may for simplicity also refer to the extension defined above also as "an extension of G", instead of "an extension of V".

Lemma 5.9. *The $\mathfrak{T}$-predicate has the intended meaning above, i.e. for all values of n, G, S, D_A, D_R, the ground fact $\mathfrak{T}(n, G, S, D_A, D_R)$ can be derived for a given HCF logic program Π, if and only if Property 1 holds.*

Proof Sketch. We have to show that the $\mathfrak{T}$-predicate computed by the program in Definition 5.8 has the intended meaning. This is shown by structural induction over $\mathcal{T}$. We at this point only sketch the most important details of the proof.

Base case For every (empty) leaf node n exactly the fact $\mathfrak{T}(n, (\emptyset, \emptyset), \emptyset, \emptyset, \emptyset)$ is derived. It is easy to verify, that Property 1 holds for this tuple, as every extension is also empty. Conversely, as the leaf node n is empty (i.e. $\chi(n) = \emptyset$), the only possible value for all of G, S, D_A, D_R is the empty set. If for these values Property 1 holds, the fact $\mathfrak{T}(n, (\emptyset, \emptyset), \emptyset, \emptyset, \emptyset)$ is clearly derived by definition of the algorithm.

Induction step–"only if"-direction: Suppose that for arbitrary values of n, G, S, D_A and D_R, the ground fact $\mathfrak{T}(n, G, S, D_A, D_R)$ was derived. To show that Property 1 then holds, we distinguish between the five different cases of node types in the tree decomposition.

1. *Atom removal node n with removal of atom a:* Let the fact $\mathfrak{T}(n, G, S, D_A, D_R)$ be derived and n' be the child of n. Then also $\mathfrak{T}(n', G', S, D'_A, D_R)$ with $G = G' \setminus \{a\}$ and $D_A = D'_A \setminus \{a\}$ must have been derived in the program. By induction hypothesis, for that fact there must then be an extension $(M \cup \rho)$ of G' as defined in Property 1. As however $\mathcal{A}_n = \mathcal{A}_{n'}$ and $\mathcal{R}_n = \mathcal{R}_{n'}$, this extension then is also a desired extension of G: Conditions 1, 3, 4 and 5 are clearly satisfied, and the set of atoms used in Condition 2 does not change.
2. *Rule removal node n with removal of rule r:* Let the fact $\mathfrak{T}(n, G, S, D_A, D_R)$ be derived and n' be the child of n. Then also $\mathfrak{T}(n', G', S', D_A, D'_R)$ with $G = G' \setminus \{r\}$, $S = S' \setminus \{r\}$ and $D_R = D_R \setminus \{r\}$ must have been derived in the program. By induction hypothesis, for that fact there must then be an extension $(M \cup \rho)$ of G' as defined in Property 1. As however $\mathcal{A}_n = \mathcal{A}_{n'}$ and $\mathcal{R}_n = \mathcal{R}_{n'}$, this extension then is also a desired extension of G: Conditions 2, 3 and 4 are clearly satisfied, and the set of rules used in Conditions 1 and 5 does not change.
3. *Atom introduction node n with introduction of atom a:* Let $\mathfrak{T}(n, G, S, D_A, D_R)$ be derived and n' be the child of n. Then, when compared to n', either atom a has been added to G or not.
   In case of the latter, a fact of the form $\mathfrak{T}(n', G, S', D_A, D_R)$ must also have been derived and by induction hypothesis Property 1 holds for that fact, i.e. there exists an extension $(M \cup \rho)$ of G for that fact. This extension is also a desired extension for $\mathfrak{T}(n, G, S, D_A, D_R)$: Conditions 3, and 5 are clearly satisfied. Because of the connectedness condition of the tree decomposition, the set of atoms in Condition 2 does not change, as a does not occur in any descendant of n, and therefore the condition is satisfied. Condition 1 is satisfied, as, by the definition of the algorithm (i.e. by the definition of the predicate *ainadapt*), S' is extended to S by adding all rules of the current bag that are satisfied by setting a to false (that is, not adding it to G). Finally, Condition 4 is satisfied as the *ainerr* predicate would

have prevented derivation if setting a to false would have resulted in a rule in G not being satisfied.
In the case that a has been added to G, a fact of the form $\mathfrak{T}(n', G', S', D'_A, D'_R)$ must have been derived for which by induction hypothesis Property 1 holds and thus there exists an extension $(M' \cup \rho)$ of G'. From this we construct an extension $M \cup \rho$ of G with $M = M' \cup \{a\}$. This extension indeed satisfies all the conditions of Property 1: By definition of the algorithm, G is constructed from G' by adding those edges of the derivation graph that include atom a. Also, G is acyclic which is asserted by the use of the predicate *acyclic*. Therefore Condition 3 holds. By definition of the predicate *aipadapt*, $D'_A = D_A \setminus \{a\}$, D_R is constructed from D'_R by adding all rules in G' with $a \in H(r^n)$ and S is constructed from S' by adding all rules not in G' whose body is satisfied by a. From this construction it clearly follows that Conditions 1 and 2 are satisfied by M and ρ. Finally, Conditions 4 and 5 are satisfied, as the definition of the *aiperr* predicate asserts that each rule derives at most one atom and all rules in G are still satisfied after setting atom a to true.

4. *Rule introduction node n with introduction of rule r:* The proof for rule introduction nodes generally follows the approach used in the proof for atom introduction nodes, distinguishing between the cases that r is added to the graph or not and making use of the connectedness condition of tree decompositions, by which it can be assured that r does not occur in the subtree rooted at n (excluding n). The proof itself is tedious but straightforward, therefore we omit giving details here.
5. *Branch node n:* Let the fact $\mathfrak{T}(n, G, S, D_A, D_R)$ be derived and n_l, n_r be the children of n. Then the facts $\mathfrak{T}(n_l, G_l, S_l, D^l_A, D^l_R)$ and $\mathfrak{T}(n_r, G_r, S_r, D^r_A, D^r_R)$ must have been derived in a previous step for n_l and n_r. By induction hypothesis, there then exist extensions $(M_l \cup \rho_l)$ of G_l and $(M_r \cup \rho_r)$ of G_r satisfying Property 1. From these two extensions we construct an extension $(M \cup \rho)$ for $\mathfrak{T}(n, G, S, D_A, D_R)$, such that $M = M_l \cup M_r$ and $\rho = \rho_l \cup \rho_r$. This extension also satisfies the property: G is constructed from G_l and G_r by forming a union over those two graphs. Predicate *acyclic* of the algorithm definition asserts that G is acyclic and predicate *agree* asserts that no two atoms are true in the same rule head. Therefore Condition 3 is satisfied. As the algorithm sets $S = S_l \cup S_r$, $D_A = D^l_A \cup D^r_A$ and $D_R = D^l_R \cup D^r_R$, it can be checked that Conditions 1, 2 and 4 also hold for $(M \cup \rho)$. Given the earlier established fact, that it is assured that (as established earlier) no rule head has more than one true atom and the fact that Condition 5 was already true for both $(M_l \cup \rho_l)$ and $(M_r \cup \rho_r)$, it is easy to see that Condition 5 also holds for $(M \cup \rho)$.

Induction Step–"if"-direction (idea): For this direction we start with arbitrary values of n, G, S, D_A, D_R and assume that Property 1 holds. We have to show that then the ground fact $\mathfrak{T}(n, G, S, D_A, D_R)$ is indeed derived in the program. Again this can be done

by distinguish between the five different cases of node types in the tree decomposition. By a number of subsequent case distinctions the "if"-direction can then be shown for each of these nodes. As the proof is straightforward but lengthy we will not state it here explicitly, but state the overall idea:

The general approach for removal nodes n to verify that all the conditions of Property 1 are satisfied for the derived fact of child n' and then by induction hypothesis construct from it the fact $\mathfrak{T}(n, G, S, D_A, D_R)$. For introduction nodes, the same basic principle is used but making use of the connectedness condition of the tree decomposition. □

Theorem 5.10. *The program given in Definition 5.8 decides the consistency problem for head-cycle free answer set programs, i.e. the ground fact* $\mathfrak{T}(\text{root}, (\emptyset, \emptyset), \emptyset, \emptyset, \emptyset)$ *can be derived if and only if the input* $\langle \mathcal{T}, \Pi \rangle$ *encodes a head-cycle free disjunctive logic program with at least one answer set, and a corresponding tree decomposition.*

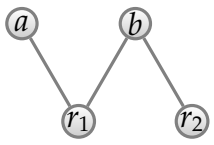
Proof. By Lemma 5.9, the ground fact $\mathfrak{T}(n_{\text{root}}, (\emptyset, \emptyset), \emptyset, \emptyset, \emptyset)$ is derived if and only if it satisfies Property 1. By the definition of the property there then exist sets $M \subseteq \mathcal{A}_{n_{\text{root}}}$ and $\rho \subseteq \mathcal{R}_{n_{\text{root}}}$ that satisfy conditions 1 and 2 of Lemma 5.9. From Theorem 5.7 it then follows immediately that M then is an answer set which proves Theorem 5.10, as Property 1 and the characterization of answer sets in 5.7 coincide for empty root nodes in a tree decomposition. □

5.3 Example

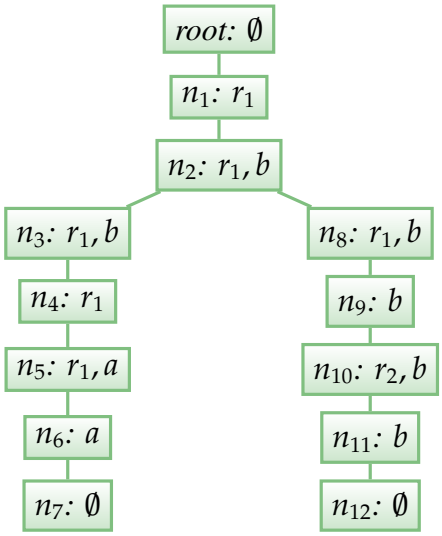
Example 5.11. *Let P be the following logic program:*

$$r_1 \quad = \quad a \leftarrow b; \qquad r_2 \quad = \quad b;$$

From this program we construct the incidence graph as defined in Definition 5.1:



A tree decomposition for this graph is given below:



For this example the algorithm above generates the following predicates $\mathfrak{T}$:

Node	*Derived Predicates*
n_7	$\mathfrak{T}(n_7,(\emptyset,\emptyset),\emptyset,\emptyset,\emptyset)$
n_6	$\mathfrak{T}(n_6,(\emptyset,\emptyset),\emptyset,\emptyset,\emptyset)$, $\mathfrak{T}(n_6,(\{a\},\emptyset),\emptyset,\emptyset,\emptyset)$
n_5	$\mathfrak{T}(n_5,(\emptyset,\emptyset),\emptyset,\emptyset,\emptyset)$, $\mathfrak{T}(n_5,(\{a\},\emptyset),\{r_1\},\emptyset,\emptyset)$, $\mathfrak{T}(n_5,(\{r_1\},\emptyset),\emptyset,\emptyset,\emptyset)$, $\mathfrak{T}(n_5,(\{a,r_1\},\{(r_1,a)\}),\emptyset,\{a\},\{r_1\})$
n_4	$\mathfrak{T}(n_4,(\emptyset,\emptyset),\emptyset,\emptyset,\emptyset)$, $\mathfrak{T}(n_4,(\{r_1\},\emptyset),\emptyset,\emptyset,\emptyset)$, $\mathfrak{T}(n_4,(\{r_1\},\emptyset),\emptyset,\emptyset,\{r_1\})$
n_3	$\mathfrak{T}(n_3,(\emptyset,\emptyset),\{r_1\},\emptyset,\emptyset)$, $\mathfrak{T}(n_3,(\{r_1,b\},\{(b,r_1)\}),\emptyset,\emptyset,\emptyset)$, $\mathfrak{T}(n_3,(\{r_1,b\},\{(b,r_1)\}),\emptyset,\emptyset,\{r_1\})$, $\mathfrak{T}(n_3,(\{b\},\emptyset),\emptyset,\emptyset,\emptyset)$
n_{12}	$\mathfrak{T}(n_{12},(\emptyset,\emptyset),\emptyset,\emptyset,\emptyset)$
n_{11}	$\mathfrak{T}(n_{11},(\emptyset,\emptyset),\emptyset,\emptyset,\emptyset)$, $\mathfrak{T}(n_{11},(\{b\},\emptyset),\emptyset,\emptyset,\emptyset)$
n_{10}	$\mathfrak{T}(n_{10},(\emptyset,\emptyset),\emptyset,\emptyset,\emptyset)$, $\mathfrak{T}(n_{10},(\{b\},\emptyset),\{r_2\},\emptyset,\emptyset)$, $\mathfrak{T}(n_{10},(\{r_2\},\emptyset),\emptyset,\emptyset,\emptyset)$, $\mathfrak{T}(n_{10},(\{b,r_2\},\{(r_2,b)\}),\emptyset,\{b\},\{r_2\})$
n_9	$\mathfrak{T}(n_9,(\{b\},\emptyset),\emptyset,\emptyset,\emptyset)$, $\mathfrak{T}(n_9,(\{b\},\emptyset),\emptyset,\{b\},\emptyset)$
n_8	$\mathfrak{T}(n_8,(\{r_1,b\},\{(b,r_1)\}),\emptyset,\emptyset,\emptyset)$, $\mathfrak{T}(n_8,(\{b\},\emptyset),\emptyset,\emptyset,\emptyset)$, $\mathfrak{T}(n_8,(\{r_1,b\},\{(b,r_1)\}),\emptyset,\{b\},\emptyset)$, $\mathfrak{T}(n_8,(\{b\},\emptyset),\emptyset,\{b\},\emptyset)$
n_2	$\mathfrak{T}(n_2,(\{r_1,b\},\{(b,r_1)\}),\emptyset,\emptyset,\emptyset)$, $\mathfrak{T}(n_2,(\{r_1,b\},\{(b,r_1)\}),\emptyset,\emptyset,\{r_1\})$, $\mathfrak{T}(n_2,(\{r_1,b\},\{(b,r_1)\}),\emptyset,\{b\},\emptyset)$, $\mathfrak{T}(n_2,(\{r_1,b\},\{(b,r_1)\}),\emptyset,\{b\},\{r_1\})$, $\mathfrak{T}(n_2,(\{b\},\emptyset),\emptyset,\emptyset,\emptyset)$, $\mathfrak{T}(n_2,(\emptyset,\emptyset),\emptyset,\emptyset,\emptyset)$
n_1	$\mathfrak{T}(n_1,(\{r_1\},\emptyset),\emptyset,\emptyset,\emptyset)$, $\mathfrak{T}(n_1,(\{r_1\},\emptyset),\emptyset,\emptyset,\{r_1\})$, $\mathfrak{T}(n_1,(\emptyset,\emptyset),\emptyset,\emptyset,\emptyset)$
root	$\mathfrak{T}(\text{root},(\emptyset,\emptyset),\emptyset,\emptyset,\emptyset)$

As the program of this example has an answer set $\{a,b\}$, the empty $\mathfrak{T}$-predicate is derived at the root node, and the algorithm answers the consistency problem with "yes", as expected.

6 From Theory to Practice

In this chapter the implementation of the algorithm described in chapter 5 is laid out. The implementation is done on the basis of a purpose-built framework (called **SHARP**[1]) for working with tree decompositions.

6.1 The SHARP Framework

The basic idea of the **SHARP** framework (or just **SHARP** for short) essentially lies in the fact that to develop paramaterized (or fixed-parameter) algorithms for problems one usually follows a uniform approach:

1. From the input problem, obtain the parameterized representation of the problem.
2. Using the fixed-parameter algorithm, obtain the solution.

This can be narrowed down even further when considering only a specific parameterization: Treewidth. The algorithm then basically follows the following steps:

1. From the input problem, obtain a tree decomposition with a specific treewidth, representing the parameterized version of the problem.
2. On the thus obtained tree decomposition, do a traversal of the tree in order to check for a solution.
3. Depending on what type solution is sought (i.e. in our case enumeration of stable models for a given logic program), do a second traversal of the tree, generating only the relevant solutions in the process.

The algorithm laid out in chapter 5 follows this pattern. Therefore, choosing to implement the algorithm in a purpose-built framework specifically designed for this task was a natural choice. **SHARP** provides a set of base classes that provide the means to focus on implementing the actual algorithm without having to deal with the problems of data management, flow control, etc. which usually is a time-consuming and tedious task.

The following section provides a general overview over how to implement algorithms based on tree decompositions in the **SHARP** framework.

[1] Smart Hypertree-Decomposition-based Algorithm fRamework for Parameterized Problems

6.1.1 Framework Description

Parsing

Parsing the input usually is the first thing that needs to be done. The framework allows much freedom at this point as different problems usually tend to have different input formats therefore not much common ground could be found to provide base functionality out of the box.

The usual method (which is currently used in all existing algorithm implementation) is to write a lexer and parser with lex[2] and yacc[3] set of tools.

Also, if possible one might run certain preprocessing and pre-optimization subroutines at this point (e.g. redundancy reduction/elimination, consolidation, etc.).

Provided Parameters

The parser should be able to start work with two parameters: A stream which yields the entire input when read and a pointer to the `Problem` class (see section 6.1.4).

In order to work with the framework, the parser has to store the input in such a way that later on the graph representation of the problem can be constructed from it. This can either be in form of a graph itself or in the form of an intermediate data structure more suited for working with later on in the algorithm. Usually this is done by implementing the necessary methods in the problem class and calling them from the parser class, whereby all the input data is stored in the `Problem` class.

6.1.2 Tree Decomposition

Once the input is read, the goal is to obtain the parameterized version of the problem or, in other words, to generate a tree decomposition of the graph representation of the input. This is done in three steps:

1. Obtain the graph representation of the input.
2. Decompose the thus created graph into a tree.
3. Normalize the tree decomposition for easier use later on in the algorithm.

Each of these steps is discussed in detail in the following sections.

Graph Representation

In order to use the **SHARP** framework, it must be possible to represent the problem (or at least certain aspects thereof) by a graph. This usually is the case when the problem

[2] `http://dinosaur.compilertools.net/lex/index.html`
[3] `http://dinosaur.compilertools.net/yacc/index.html`

consists of entities and relations between entities. Except the problem of solving answer set programs that is discussed in depth in this thesis, following is a short list of example problems that can be implemented in the framework:

- Argumentation problems: Here the vertices are the arguments and the attack relation directly corresponds to the edges in the graph (see [Dvorák et al., 2010]).
- Multi-Cut problems: Here the input is already a graph and therefore can be directly used for the tree decomposition step (see [Pichler et al., 2010]).
- Satisfiability problems: Given a propositional logic formula in conjunctive normal form, one can compute an incidence graph similar to the one defined in section 5.1.

The graph representation in the framework is represented by the `Hypergraph` class (the framework is actually implemented for hypergraphs and hypertree decompositions). This implementation is a straight-forward approach as the `Hypergraph` class consists of two collections, one for the vertices and one for the edges. In order to fill the `Hypergraph` class so that the framework can work with it, one has to instantiate the `Node` class for each vertex and the `Hyperedge` class for each edge between them. The following is an example, instantiating a graph that has two vertices and one edge that connects them.

Listing 6.1: Example of a graph with two nodes and one edge.

```
Hypergraph *hg = new Hypergraph();
Node *a = new Node(1, 1), *b = new Node(2, 2);
Hyperedge *e = new Hyperedge(1, 1);
e->insNode(a); e->insNode(b); a->insEdge(e); b->insEdge(e);
a->updateNeighbourhood(); b->updateNeighbourhood();
e->updateNeighbourhood();
hg->iMyMaxNrOfNodes = 2;
hg->iMyMaxNrOfEdges = 1;
```

However, in order to avoid having to implement the tedious subroutine of moving vertices and edges into the `Hypergraph` class, the framework provides a ready-to-use method doing exactly that for the case that the graph representation of the problem is indeed a graph (i.e. each edge connects exactly two vertices). This (static) method can be found in the `Problem` class and has the following signature:

Listing 6.2: The createGraphFromSets method signature.

```
Hypergraph *Problem::createGraphFromSets
                (VertexSet, EdgeSet);
```

The `VertexSet` and `EdgeSet` data types are defined as an STL set of integers (i.e. `std::set<int>`) and an STL map from integers to integers (`std::map<int, int>`) whereby each integer represents the internal number of the corresponding vertex

in the graph. This internal number should be defined during the parsing process as it is the only way to map the labels in the tree decomposition back to the corresponding vertices.

Decomposition

The tree decomposition step turns the graph representation of the problem (as provided in the `Hypergraph` class) into a tree, trying to minimize the treewidth in the process. As finding a tree decomposition with minimal width is in itself an NP-hard problem (as discussed in chapter 3), this is done by a heuristic to speed things up. Therefore the generated tree will not necessarily have minimal width. The specific heuristics currently implemented can be found in [Dermaku et al., 2008]. Standalone implementations can be obtained from `http://www.dbai.tuwien.ac.at/proj/hypertree/downloads.html`.

The tree decomposition routine yields an instance of the `Hypertree` class which represents one of the possible trees corresponding to the given graph.

Normalization

In the normalization step the previously obtained `Hypertree` class is transformed into an instance of the `ExtendedHypertree` class which provides additional methods for easily accessing the bags in the tree decomposition. During this conversion the tree decomposition is normalized as discussed in section 3.1.1.

6.1.3 Algorithm Implementation

Walking the Tree

This is where the core algorithm is executed. The `AbstractAlgorithm` class represents a tree decomposition-based algorithm. Recall that in a normalized tree decomposition there are only four types of nodes in the tree decomposition and only four operations need to be implemented for the following types of nodes. See chapter 3.3 for details of the general approach.

The Algorithm Class

The algorithm has to specify actions taking place when each of these node types is encountered. Listing 6.3 shows the declaration of the algorithm class as seen in the corresponding C++ header file. In order to implement an algorithm in the **SHARP** framework, one has to implement the five purely virtual (i.e. abstract) methods as seen in the listing.

Listing 6.3: The `AbstractAlgorithm` class header.

```
class AbstractAlgorithm
{
public:
        AbstractAlgorithm(Problem *problem);
        virtual ~AbstractAlgorithm();

protected:
        Instantiator *instantiator;
        Problem *problem;

public:
        void setInstantiator(Instantiator *instantiator);
        Solution *evaluate(const ExtendedHypertree *root);

protected:
        virtual Solution *selectSolution(
                TupleSet *tuples,
                const ExtendedHypertree *root) = 0;

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

        TupleSet *evaluateNode
                (const ExtendedHypertree *node);
};
```

The `evaluate*Node` methods represent the actions the algorithm takes when encountering one of the four node types. The framework automatically calls the correct method when the `evaluateNode` method is called for a node.

Tuples and Possible Worlds

In order to pass data calculated in one node on to the next one, the concept of tuples is introduced: One `Tuple` instance represents a possible world for a particular node. Basically a bottom-up tree decomposition-based algorithm calculates at each node all the possible worlds for that node by applying incremental operations to all the possible worlds of its child node(s). All "impossible" worlds are simply omitted or eliminated which results in a classical dynamic programming algorithm.

As the `Tuple` data structure varies strongly from algorithm to algorithm, one needs

to derive the `Tuple` class and define the data structures needed for each tuple (i.e. each possible world of the specific algorithm). As an example, one might consider an algorithm for the CNF-SATISFIABILITY problem, given by a PL0-formula in conjunctive normal form, where each conjunct is called a *clause* and represented by its incidence graph. For this problem each tuple just contains the set of positive atoms, the set of negative atoms and the set of clauses, as these three sets suffice to represent a possible world in a node (as the bags of the tree-nodes would contain a subset of the atoms and a subset of the clauses of the original problem). Note that the set of negative variables could also be stored implicitly as the difference of the variables in the Node and the positive variables.

Usually the information that is stored in the `Tuple` class is enough to calculate an answer for the decision problem (i.e. answer "yes" or "no"). However, usually one also wants to actually find one or more solutions to the problem (i.e. in case of the CNF-SATISFIABILITY problem, one wants to not only know that the input formula is satisfiable, but also what all the satisfying truth assignments look like). In order to do just that, the concept of a `Solution` is introduced.

Solution and SolutionContent classes

Each `Tuple` in a node represents a possible world from which one or more partial–partial in the sense that one node only represents a part of the whole problem–solution to the problem can be generated. However calculating all (partial) solutions during the bottom-up traversal would ultimately lead to an exponential running time for hard problems which is what we want to avoid. Also many of these partial solutions would never be needed again as the corresponding `Tuple` may at some point be eliminated. Therefore the **SHARP** framework provides the `Solution` and `SolutionContent` classes. Again the structure of a solution to the problem depends on the problem, therefore one has to define the data structure for the solutions. This is done by deriving the `SolutionContent` class. One instance of the `SolutionContent` class represents one or more partial solutions associated with a `Tuple` instance (i.e. with a possible world).

The framework already provides ready-made derived `SolutionContent` classes for some common solution types (namely Enumeration, Counting and Boolean solution types). However, new derived `SolutionContent` classes may be implemented as needed. Each `SolutionContent` must provide implementations for the three (idempotent) merging operations:

- **Union**: When by modifying the `Tuple` instances from a child node (i.e. in an introduction or removal node) after modification two `Tuple` instances coincide, their associated `SolutionContent` instances are merged using the `calculateUnion` method.

- **CrossJoin**: When in a branch node a `Tuple` instance of the left child and right child are joined, their associated `SolutionContent` instances are merged using the `calculateCrossJoin` method.
- **AddDifference**: When in an introduction node a `Tuple` instance is modified by incorporating the introduced vertex, its associated `SolutionContent` instance incorporates the same vertex using the `calculateAddDifference` method.

All these operations can be triggered by calling the appropriate method of the provided `Instantiator` class instance associated with the algorithm. This class handles `SolutionContent` instantiation and also ensures lazy evaluation of solutions if needed.

Skeleton Implementation

Once the `Tuple` class is derived from and the algorithm-specific `Tuple` is defined, one can start implementing the respective node evaluation methods. For a bottom-up traversal, the implementation usually takes the form of listing 6.4 (here the case of an introduction node is taken as an example).

Listing 6.4: Generic skeleton for implementing the algorithm methods.

```
TupleSet *SomeAlgorithm::
evaluateIntroductionNode(const ExtendedHypertree *node)
{
// call this method first to do a bottom-up traversal
TupleSet *base = this->evaluateNode(node->firstChild());

// instantiate the new Tuple set for this node
TupleSet *ts = new TupleSet();

for(TupleSet::iterator it = base->begin();
        it != base->end(); ++it)
{
        SomeTuple &told = *(SomeTuple *)it->first;

        // calculate the new Tuple based on the
        // old one and the node type/difference
        // NOTE: this is where the actual code goes
        SomeTuple &tnew = modify(told, node);

        // incorporate the change into the solution
        // by calling the appropriate method of the
        // Instantiator instance
        // NOTE: This is only an example, your code
        //       may call other Instantiator methods
        Solution *snew = this->instantiator->
                addDifference(it->second,
```

```
                        node->getDifference());

        // try to insert the new Tuple into the set
        pair<TupleSet::iterator, bool> result =
                ts->insert(TupleSet::value_type(
                        &tnew, snew));

        // if the very same Tuple is already contained
        // in the Tuple set, merge the solutions using
        // the Union operation, then insert it instead
        // of the old one
        if(!result.second)
        {
                Solution *sold = result.first->second;
                ts->erase(result.first);
                ts->insert(TupleSet::value_type(&tnew,
                        this->instantiator->combine
                                (Union, sold, snew)));
        }
}

// free up some memory, old Tuples not needed anymore
delete base;

// return the new Tuple set
return ts;
}
```

After evaluating the last node (i.e. the root node of the decomposition), the method `selectSolutions` is called. This method should simply check all TupleSets and return the Union (using the `Instantiator`) of all solutions that belong to valid tuples.

6.1.4 Pulling it all together

Now that we have all the principal components to run our algorithm, the only thing that still lacks is a class that provides the necessary program flow control (i.e. that determines when to do what and in which order). In order to do this, the framework provides the `Problem` class:

The Problem Class

The `Problem` class is provided as a base class by the **SHARP** framework and handles the earlier mentioned task of flow control. The `Problem` class also acts as the interface to the "outside world" such that it provides simple methods to read a problem instance and generate a solution for it.

For each new problem the framework should handle, the `Problem` class needs to be derived from (i.e. one may have an `AnswerSetProblem` class for answer set programs and a `SatisfiabilityProblem` class for the **SATISFIABILITY** problem). Each one of these derived classes then may use multiple different parsers (i.e. for different input formats of the same problem) and different tree decomposition algorithms

In order to implement a `Problem` class, one needs to implement three methods:

1. The `parse` method: This method should call the parser (if there is one) and store the problem in an internal data format, which can be defined as needed (i.e. private fields in the Problem class, etc.).
2. The `preprocess` method: This method should perform preprocessing tasks on the problem stored in the internal data format.
3. The `buildHypergraphRepresentation` method: This method should convert the internal representation of the problem to an instance of the `Hypergraph` class, as discussed in section 6.1.2.

The `Problem` class must be instantiated by passing as a parameter in the constructor an instance of the `AbstractAlgorithm` class that should be used to solve the problem. Once this is done and the methods discussed above are implemented, the `Problem` class is ready for use and will use the specified algorithm to solve the problem read by the `parse` method. This is done by calling the `calculateSolution` method of the `Problem` class. This method handles the program flow. Its (for the sake of readability simplified) definition is provided in listing 6.5.

Listing 6.5: Definition of the `parse` method of the `Problem` class (simplified).

```
Solution *Problem::calculateSolution(Instantiator *inst)
{
        this->parse();
        this->preprocess();

        Hypergraph *hg =
                this->buildHypergraphRepresentation();

        H_BucketElim be;
        Hypertree *ht =
                be.buildHypertree(hg, BE_MIW_ORDER);

        ht = new ExtendedHypertree(ht);
        ((ExtendedHypertree *)ht)->normalize();

        this->algorithm->setInstantiator(inst);
        return this->algorithm->
                evaluate((ExtendedHypertree *)ht);
}
```

Obviously this method does what is expected in order to run the whole algorithm from parsing to solution generation. First, the parser is started, then, after preprocessing, the hypergraph representation is built, which is then decomposed into a hypertree and subsequently normalized, after which the actual algorithm is run using the provided `Instantiator` instance.

As discussed in section 6.1.3, the `Instantiator` instance that is used by the algorithm determines which `SolutionContent` class will get instantiated. Therefore, by just calling the `calculateSolution` method of the `Problem` class with different `Instantiator` instances, one can easily specify what the solution should look like (i.e. a call with an `Instantiator` that creates boolean `SolutionContent` instances, only a yes/no answer is provided, whereas when the `Instantiator` creates `CountingSolutionContent` instances, the solution yields a number–counting for example the number of answer sets).

6.2 Implementation of the dynASP Algorithm

The "dynASP" algorithm is a novel dynamic programming-based algorithm for solving ground answer-set programs (that is, for determining the stable models of such a program). Based on a tree decomposition of the incidence graph of the program, the algorithm does a bottom-up traversal of said tree decomposition, checking whether a solution to the program exists and if it does, output either just "yes", or count the number of stable models, or enumerate the stable models. A first prototype of this implementation was presented at JELIA 2010 (see [Morak et al., 2010]). Generally, solving disjunctive logic programs under stable semantics is known to be a hard problem (as discussed in chapter 2). However it is known to be fixed-parameter tractable and a first FPT-algorithm was published in [Jakl et al., 2009].

As an example, take the program from Example 2.35.

This program has an incidence graph equal to the one in Example 5.2, and a tree decomposition for this graph is given in Example 3.3. The (simplified) implementation of the algorithm is described in the following sections. Atoms in the logic program are referred to as "variables" in the code.

6.2.1 Implementing the Problem Class

Listing 6.6: Definition of the `DatalogProblem` class (simplified).

```
typedef Vertex Rule;
typedef Vertex Variable;
typedef VertexSet RuleSet;
typedef VertexSet VariableSet;
typedef std::map<Rule, std::map<Variable, bool> > SignMap;
```

```
typedef std::vector<VariableSet> HeadMap;

class DatalogParser;
class DatalogProblem : public Problem
{
public:
        DatalogProblem(std::istream *stream);
        Rule addNewRule();
        Variable addVariable(std::string name);
        void addEdge(Rule rule, Variable variable,
                bool positive, bool head);
protected: ...  // declaration of parse, preprocess
                // and buildHG methods
private:
        SignMap signs;
        HeadMap heads;
        TypeMap types;
        DatalogParser *parser;
};
```

Listing 6.6 shows the (simplified) declaration of the derived `Problem` class. The main points are the following:

- Declaration of the internal data structures to store the answer-set problem. The `SignMap` stores the rules (including body and head atoms) and the `HeadMap` stores the rule heads.
- Declaration of methods called by the parser (`addNewRule`, `addVariable`, `addEdge`) when the respective event is encountered during parsing. These methods then store the data in the `HeadMap` and `SignMap`.
- Stream that supplies the input is used as parameter of the constructor and a pointer to the parser instance (`DatalogParser`) is kept.

The implementation of the various functions is fairly self-explanatory so we omit details here but only try to make the main points clear. The constructor of the `Problem` class above simply initializes the corresponding parser class (i.e. creates an instance of the `DatalogParser`) and passes a pointer to itself and the input stream to the parser.

6.2.2 Implementing the Parser

The parser in this particular program is written with the lex/yacc combination as already discussed previously in section 6.1.1. The parser is written in such a way that it accepts as constructor arguments the stream that yields the input and a pointer to the `Problem` class mentioned in section 6.2.1. Whenever the parser encounters a new rule during parsing, the `addRule` method of the `Problem` class is called. Analogously, for each

variable the `addVariable` method is called and for each occurrence of a variable in a rule the `addEdge` method is called, including information whether the variable occurs negated or in the head of the rule.

6.2.3 Implementing the Algorithm

Implementing the algorithm is fairly straight-forward given the structure that is already provided by the framework. Doing a bottom-up traversal of the tree using the method described in section 6.1.3 and 6.1.3 immediately results in an easy way to implement the algorithm in the framework. There are however a few points worth noting when looking at the declarations in listing 6.7:

Listing 6.7: Definition of the `AnswerSetAlgorithm` class and corresponding tuple (simplified).

```
class AnswerSetTuple : public Tuple
{
public:
        set<Variable> variables;
        set<Rule> rules;
        set<Atom> guards;
};
class AnswerSetAlgorithm : public AbstractAlgorithm
{
public:
        AnswerSetAlgorithm(Problem *problem);
protected:
        virtual Solution *selectSolution
                (TupleSet *tuples, const ExtendedHypertree *node);
        virtual TupleSet *evaluateLeafNode
                (const ExtendedHypertree *node);
        ... // declaration of all other evaluation methods
private:
        DatalogProblem *problem;
};
```

- In the constructor a pointer to the `Problem` instance is taken. This is usually needed, as all the information about the problem (i.e. in this case `HeadMap`, `SignMap`) is stored in the `Problem` instance.
- For the sake of clarity, equality and less-than operators have been omitted in the `Tuple` declaration. These operators are however needed in order to store `Tuple` instances efficiently in sets.
- In the `evaluateLeafNode` method, the `Solution` instances are created using the `Instantiator`'s `createLeafSolutions` method for every partition of the variables in the node.

6.2.4 Implementing the Instantiator

No separate implementation for the `Instantiator` class is needed, as the framework already provides a `GenericInstantiator` class which can be used for all `SolutionContent`-derived classes, as long as they implement the same constructors as the prototype. Also, default-implementations for enumerating solutions, counting solutions and boolean solutions are provided by the framework. The `Instantiator` instance can thus simply be obtained by using the code in listing 6.8.

Listing 6.8: Using the generic `Instantiator` class.

```
Instantiator *inst =
        new GenericInstantiator<EnumerationSolution>(true);
// or...
inst =
        new GenericInstantiator<CountingSolution>(false);
```

The parameter passed to the `GenericInstantiator` constructor is a boolean determining whether to enable lazy evaluation or not. Enable this (i.e. call with "true") when exponential blowup is otherwise to be expected. For example when enumerating stable models, at each step in the tree enumerating all partial solutions would be infeasible. Therefore in this case we enable lazy evaluation so that only the partial solutions that actually contribute to the full solution are enumerated. On the other hand when counting is the objective, we may safely disable lazy evaluation as in each step only a number needs to be updated.

6.2.5 Result

The implementation described above results in a runnable program reading propositional disjunctive logic programs in a DLV-based input format (see [Leone et al., 2006]). Experimental results of the implementation are discussed in section 6.3.

Currently, also other algorithms based on tree decompositions are being implemented, most notably an algorithm to solve argumentation problems under various semantics (see [Dvorák et al., 2010]) and multicut algorithms in graphs as described in [Pichler et al., 2010].

6.3 Experimental Results

Figures 6.1 to 6.6 show experimental runtime results for answer-set programs of constant treewidth and increasing size. As our implementation is designed to work with the DLV syntax (see [Leone et al., 2006]), we simultaneously ran the benchmarks on the DLV system for comparison.

Benchmarks were done on an Intel Core2Duo 2.6GHz, Ubuntu 10.10 32bit machine. Random CNF-SATISFIABILITY programs of constant treewidth were generated, converted to ASP programs and passed to the solver. In order to generate the CNF-SATISFIABILITY programs, the `mkcnf` tool[4] was used. For a sufficiently large number of atoms, the clause size correlates with the treewidth of the incidence graph of a generated program, making it easy to generate a sufficient number of programs with the same treewidth (based on the incidence graph). The programs are then translated to equivalent answer set program.

In the following diagrams, the vertical axis represents the time in seconds to solve the consistency problem (i.e. check, whether an answer set exists) and the horizontal axis the number of clauses (the *size*) of the generated CNF-SAT program.

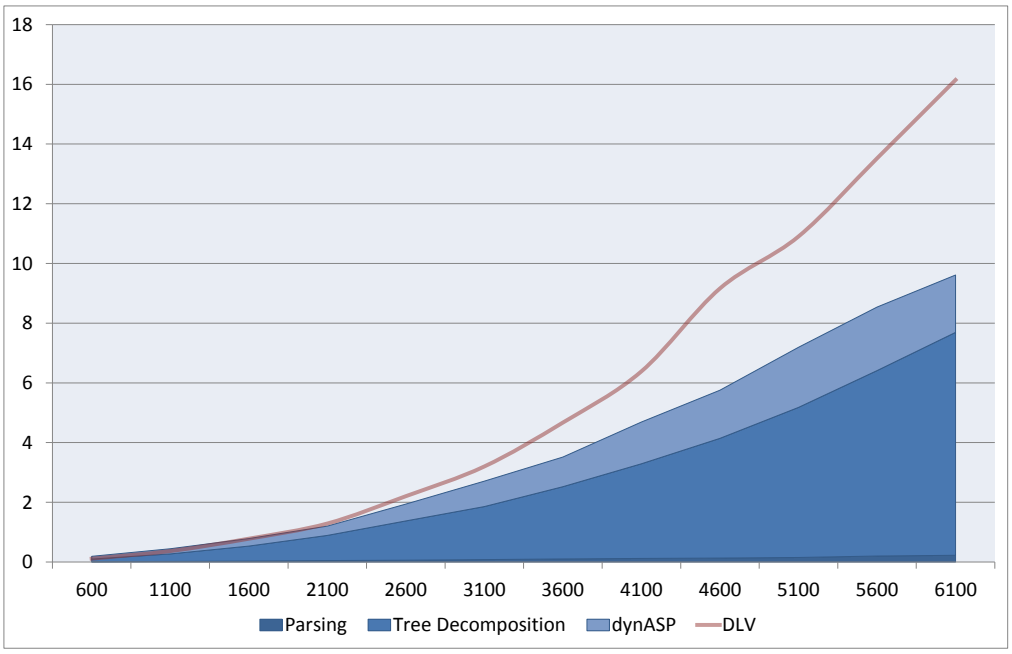


Figure 6.1: Benchmarks: Treewidth 4, average runtime

As Figure 6.1 shows, for small treewidth the system performs fairly well in the average case, beating the DLV system even for problems of small size.

[4] `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/instances`

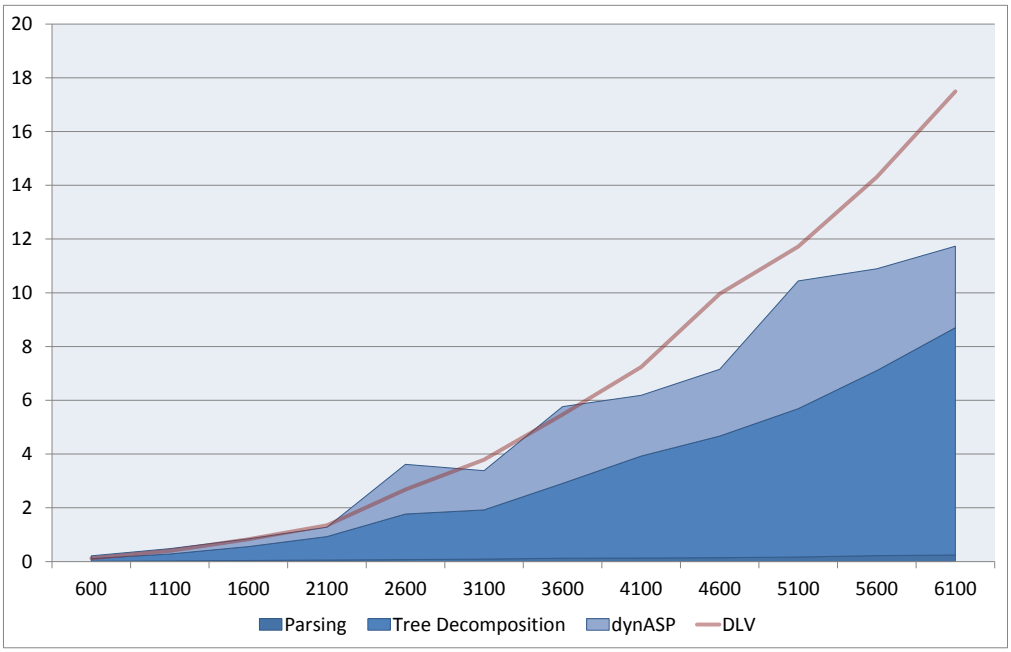


Figure 6.2: Benchmarks: Treewidth 4, worst-case runtime

The worst-case runtime for small treewidth shows a mixed picture: On the one hand, the overall picture is similar to the one for the average runtime, on the other hand, the spikes at problem sizes 2600, 3600 and 5100 show that in certain cases the worst-case runtime is not yet optimal. Further investigations in this area are needed.

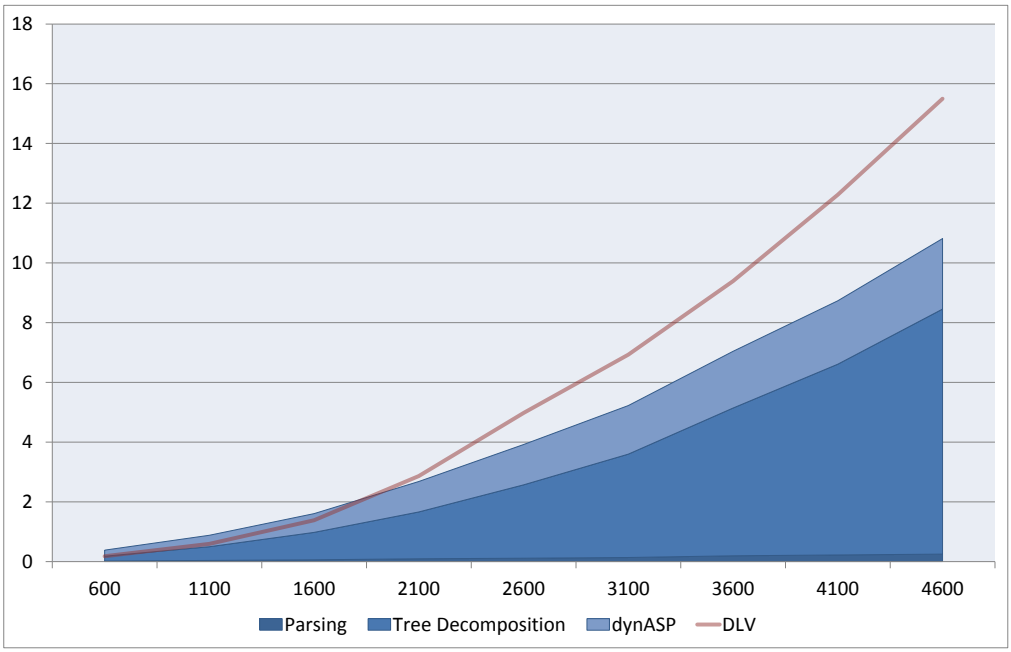


Figure 6.3: Benchmarks: Treewidth 5, average runtime

The average runtime increases overall, once the treewidth is increased from 4 to 5. The overall percentage of time taken up by the dynASP routine increases when compared to the time needed for the tree decomposition step. Also notice that for smaller problems (i.e. size 600 to 2100) DLV is now faster on average.

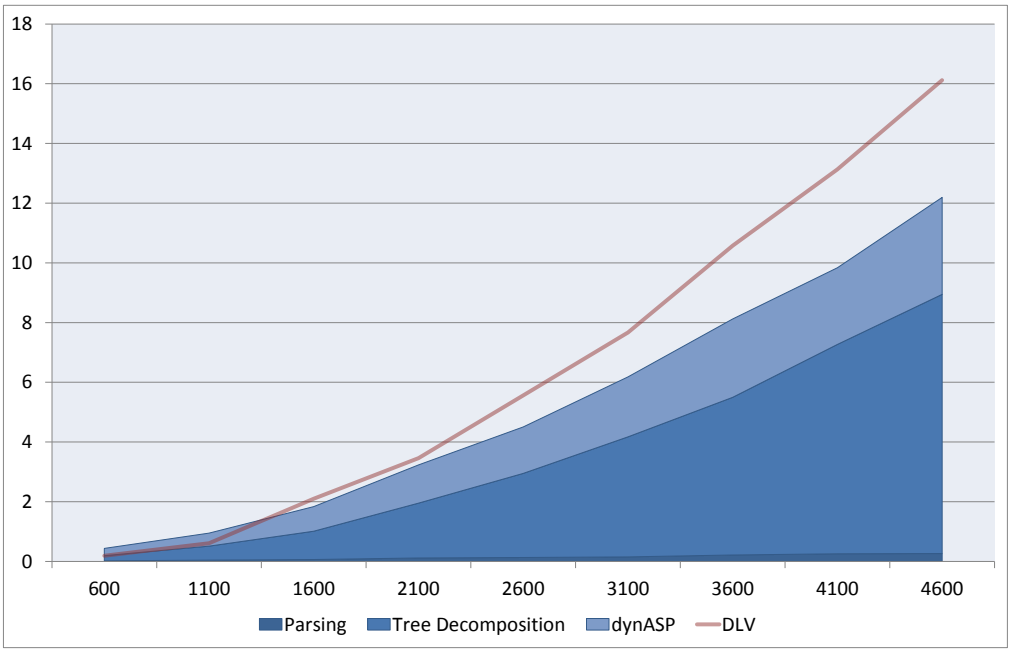


Figure 6.4: Benchmarks: Treewidth 5, worst-case runtime

For treewidth 5, worst-case runtime behavior shows the same picture as for treewidth 4: For e.g. a problem size of 4600 SAT clauses the worst-case runtime is about twice the time it takes on average, whereas DLV only experiences a 6 percent increase in runtime. It is at the moment unclear which parameters determine these runtime fluctuations and this is a topic for further investigation.

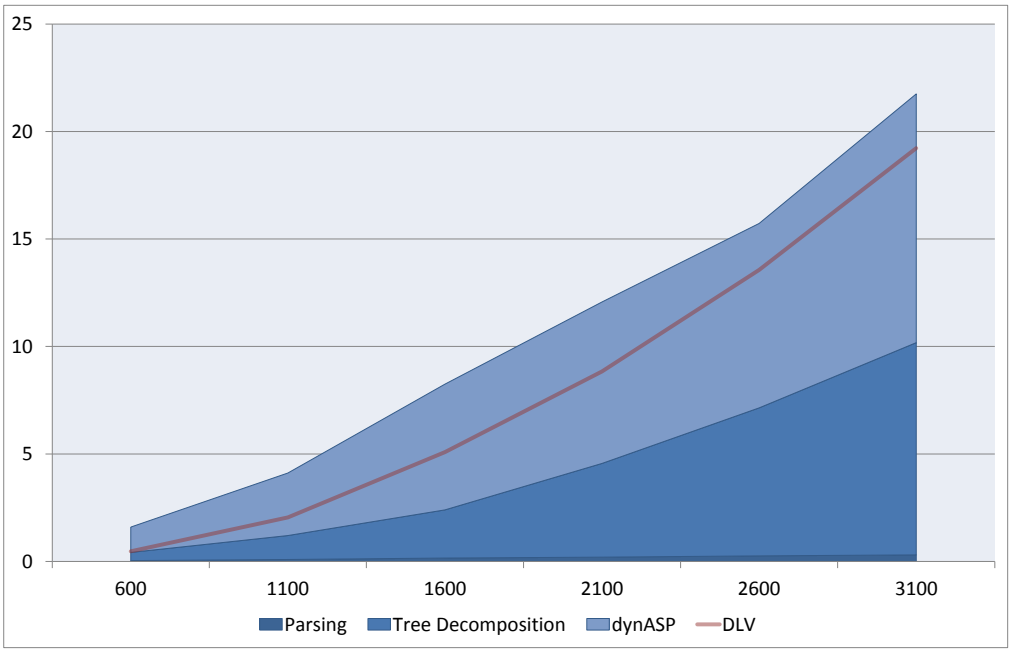


Figure 6.5: Benchmarks: Treewidth 7, average runtime

For treewidth 7, determined by the $f(k) \cdot n$ runtime of the dynASP algorithm, the expected increase in runtime of the dynASP component can clearly be seen. As $f(k)$ is in our case an exponential function, an increase of k from 4 to 7 leads to a significant increase in runtime costs. Due to limitations of the testing environment, it is at the moment impossible to reliably generate big programs with treewidth 7. Therefore benchmark data stops at 3100 SAT clauses at the moment.

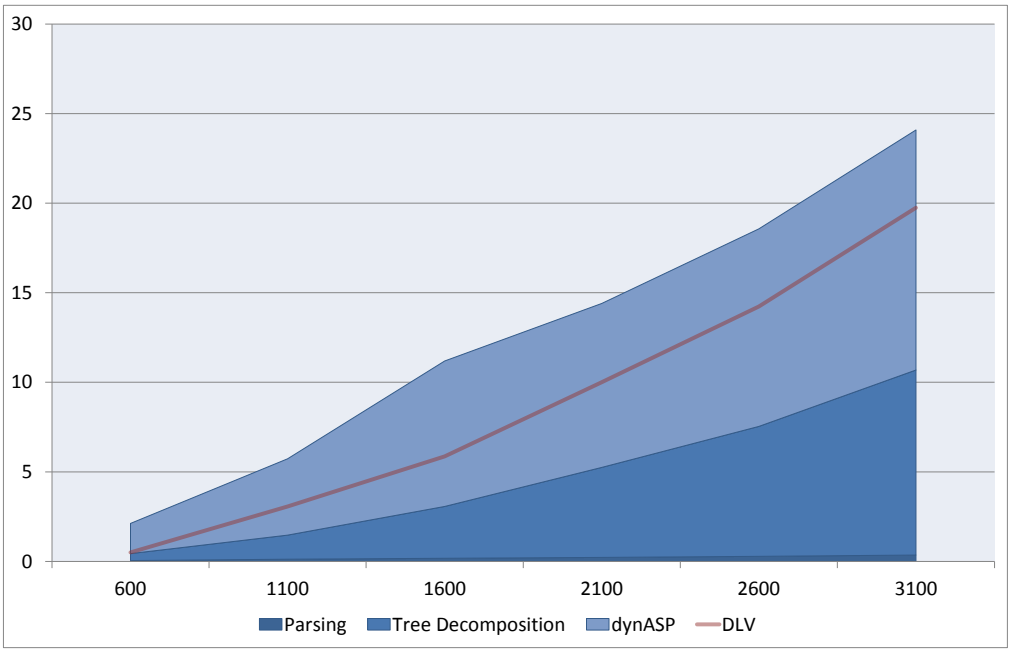


Figure 6.6: Benchmarks: Treewidth 7, worst-case runtime

Considering the linear increase of the dynASP portion of the runtime, it can be expected, that for problems of 5000 SAT clauses and up, the overall runtime of the dynASP algorithm is able to outperform DLV. As the proportionate tree decomposition runtime seems to increase as the problem size increases, finding good and efficient tree decomposition heuristics would significantly improve the overall runtime of our algorithm.

IV Final Thoughts

7 Discussion

We presented a novel algorithm for evaluating head-cycle free disjunctive logic programs. The algorithm was then implemented in C++, carefully evaluated and benchmarked against the DLV system for evaluating general disjunctive logic programs. This implementation of the algorithm we named "dynASP".

First experiments with the dynASP implementations show great potential for programs of low treewidth, as seen in section 6.3, when compared to state-of-the-art answer set programming solvers. Particularly in situations where the task only requires one pass on the tree decomposition, our algorithm outperforms traditional solvers. This is the case e.g. for the decision problem, where the algorithm only has to check, whether at the (empty) root node, a tuple still exists, and therefore no second pass (as in e.g. enumeration) is required to calculate the actual solution.

Also in comparison to a reference implementation of the algorithm described in [Jakl et al., 2009], our algorithm performs better in the average case. Therefore, the optimization for head-cycle free disjunctive logic programs has paid off in the sense that the specially tailored algorithm described in this thesis does not only outperform the more general algorithm of Jakl et. al. in theory, but this could also be verified in practice.

One major performance impact results from slow tree decomposition heuristics. Currently there are (to the best of our knowledge) no implementations available that significantly outperform the library that we used in **SHARP**. Also, depending on the shape of the tree decomposition, the worst-case runtime of the algorithm is sometimes 200 percent of the average case runtime. This is a significant improvement over the initial implementation that was presented in [Morak et al., 2010], where it was almost 1000 percent, however there is still room for improvement. Also the libraries currently in use for representing internal data structures represent a bottleneck in the algorithm as well as in the **SHARP** framework itself.

Related Work In [Jakl et al., 2009, Jakl, 2010], a fixed-parameter tractable algorithm for evaluating general disjunctive answer set programs is introduced. This approach is also based on tree decompositions and treewidth as parameter and the initial implementation was done in Haskell[1]. However this approach required an extensive amount

[1] `www.haskell.org`

of preprocessing (i.e. transforming the program to a graph representation, building the tree decomposition by an external tool, transforming the output into static Haskell code and adding it to the Haskell program, then finally compiling and running the Haskell program). This approach was not competitive when compared to out-of-the-box solvers that simply read the problem description and do all these preprocessing steps directly and in memory. However, for preliminary benchmarking, an implementation of this algorithm in the **SHARP** framework has been created.

Another work that is based on the same principle as the work described in this thesis was presented in [Samer and Szeider, 2010]. In this paper the #SAT problem was solved using dynamic programming based on tree decompositions. However, the structure needed for the tuples in the tree is much simpler than the one used in this thesis.

Related problems of answer set programming are classically the problems of constraint satisfaction (CSP) and conjunctive query (CQ) evaluation. For both, treewidth-based approaches as well as other, structural decomposition-based approaches have been examined (see e.g. [Chekuri and Rajaraman, 2000, Gottlob et al., 2000]). These approaches (when using treewidth) also follow the general approach of propagating data upward by a bottom-up traversal of the tree decomposition. However, also the idea of a top-down traversal for post-processing for conjunctive queries has been examined.

Also for conjunctive queries, treewidth-related notions have been examined. In [Gottlob et al., 2001] the concept of hypertree decompositions is discussed. In a nutshell, hypertree decompositions satisfy all the properties of tree decompositions (extended to hypergraphs and hyperedges), and one tries to minimize the number of hyperedges occurring in a tree bag. This approach has been shown to generalize tree decompositions (see [Gottlob et al., 1999] for the definition of hypertree decompositions). Bounding the hypertree width, conjunctive query evaluation is shown to be in the complexity class LOGCFL which entails that conjunctive query evaluation is a highly parallelizable task.

8 Conclusion

In this thesis we presented a novel algorithm for evaluating head-cycle free disjunctive logic programs. This algorithm is based on results from parameterized complexity theory, using treewidth as a problem parameter to restrict the complexity of evaluating said class of logic programs.

In the course of developing this algorithm, we established an alternative characterization of answer sets in head-cycle free disjunctive logic programs. Given a set of atoms one has to check whether there is a corresponding set of rule-sequences such that every rule derives exactly one atom, all the atoms in the set are derived and each sequence of rules starts with a fact. If the set of atoms then satisfy all the rules of the given program, it represents an answer set. In the characterization, these sequences of rules are represented by means of a derivation graph. If this graph is acyclic, then the rules fulfill the conditions discussed above.

These results lead to an algorithm that, in short, calculates the derivation graph for a given set of atoms and rules and checks whether the appropriate conditions are met. This is done by means of tree decompositions, evaluating in a bottom-up manner for each node in the tree decomposition, whether the derivation graph property is locally satisfied. Making use of the special properties of a tree decomposition, when arriving at the (empty) root node, one can immediately decide whether answer sets for the given program exist or not. This approach then yields an algorithm with runtime of the form

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

where k is the width of the tree decomposition and n is the size of the program. Such an algorithm is called a fixed-parameter algorithm as, when the parameter k is regarded as fixed, the runtime becomes polynomial ($f(k)$ for an NP-complete problem is necessarily exponential, as otherwise it would follow that $P = NP$).

Although this approach yields an algorithm that is able to evaluate *every* possible head-cycle free disjunctive logic program, best performance is achieved for instances with small parameter values, as can easily be seen from the runtime function.

In order to implement the algorithm, a special framework for working with tree decompositions has been designed, yielding a flexible means to rapidly implement various algorithms based on tree decompositions. Using this purpose-built framework, the

dynASP algorithm was then implemented[1] and carefully evaluated and benchmarked against existing systems such as DLV [Leone et al., 2006], where actual runtime results confirm the theoretical runtime behavior, outperforming DLV for instances of small treewidth.

In light of these results we see the future of our solver as a benchmark tool for current answer set programming solvers and also as a possible augmentation of current systems, where e.g. it could after parsing be decided whether an instance of low treewidth is encountered and if yes, our algorithm is executed. Otherwise the original algorithm is used to evaluate the logic program.

8.1 Future Work

In the future we plan to further optimize the implementation of the algorithm in order to be competitive also at higher treewidths. Therefore we plan to optimize data structures used in our algorithm as well as do away with the niceness property (i.e. normalization as laid out in Definition 3.5) of the tree decomposition as handling a non-nice tree decomposition directly would result in a substantial performance increase.

Furthermore, as the tree decomposition heuristics currently in use present a bottleneck in our implementation, we will investigate the area of tree decomposition upper bounds and approximation algorithms for tree decompositions that yield trees of low width (that is, near the actual treewidth of the problem) while still exhibiting a favorable runtime behavior. Furthermore we plan on investigating tree decomposition heuristics for certain restricted classes of graphs. For example, the incidence graph used in this thesis to represent answer set programs is always bipartite. However this fact is not used by the current tree decomposition heuristics, and it would be worth investigating whether such properties can be exploited in order to increase efficiency of the heuristics or approximation algorithm.

In light of the results in [Gottlob et al., 2001], it would be interesting to pinpoint the exact complexity of parameterized answer-set programming with treewidth as its parameter. Tree decompositions make for good parallelization opportunities in actual implementations, however a native approach would require for each sequence of non-branch nodes in the tree decomposition a sequential number of computation steps of the same size (one for each tree node). Investigating the complexity of the parameterized problem, it may be possible to find that the problem is indeed highly parallelizable, that is, no such sequence of computation steps is in fact needed.

Another area worth investigating is the field of testing and benchmark generation. Currently there is a lack of benchmark suites for ground answer set programs in general and ground answer set programs for restricted settings in particular (i.e. instances of

[1] `http://dbai.tuwien.ac.at/proj/dynasp/`

constant, but low treewidth). Therefore it would be interesting to look into meaningful benchmark generation for this area, as has for instance been done for SAT with bounded width resolution refutations in [Atserias et al., 2009].

Finally the area of evaluating non-ground answer set programs directly is an area that we will definitely look into in the future. As discussed earlier in this thesis, virtually all current ASP solvers employ a two-step approach when evaluating answer-set programs: A grounding and subsequently a solving step. Except for [Palù et al., 2009], to the best of our knowledge, no investigations in this area have been undertaken. It would therefore be interesting to combine the idea of lazy grounding with tree or hypertree decompositions. Also a two-step decomposition approach could be interesting where a hypertree decomposition of a static (non-ground) program is generated and can then be evaluated over multiple knowledge bases or domains. This approach would resemble a compiler for non-ground logic programs, where the program is compiled and can then be executed on multiple inputs without having to reconstruct the tree or hypertree decomposition every single time, while at the same time eliminating the time for the (hyper-)tree decomposition step in the runtime.

V Appendices

A Benchmark Data

Find below the raw benchmark data of the dynASP implementation.

Benchmark Setup

- Intel Core2Duo P9500 (2.6 GHz, 1066 MHz FSB)
- 4096 MB DDR2-RAM
- Ubuntu 10.10

Benchmark Method

- Random-generated SAT problems of increasing size, constant treewidth
- Conversion to ASP problems
- Single-threaded benchmark, using `/usr/bin/time -a -v`

Size	Parsing	Tree Decomposition	dynASP	DLV
600	0.012	0.080005	0.092006	0.1
600	0.012	0.088006	0.080005	0.11
600	0.012	0.088005	0.100007	0.12
600	0.008	0.080005	0.092006	0.11
600	0.012	0.084005	0.092006	0.11
600	0.012	0.092005	0.100007	0.12
600	0.008	0.088006	0.084005	0.1
600	0.012	0.092006	0.088006	0.11
600	0.012	0.096006	0.088006	0.12
600	0.012	0.088006	0.104006	0.12
600 Average	0.0112	0.0876055	0.092006	0.112
1100	0.024001	0.256016	0.16001	0.34
1100	0.020001	0.248016	0.16401	0.35
1100	0.020001	0.256016	0.16401	0.4
1100	0.020001	0.252016	0.152009	0.36
1100	0.020001	0.248016	0.16801	0.36
1100	0.020001	0.256016	0.192012	0.36
1100	0.008	0.252016	0.176011	0.38
1100	0.020001	0.256016	0.200012	0.38
1100	0.020001	0.248016	0.17601	0.35
1100	0.020001	0.252016	0.176011	0.4
1100 Average	0.0192009	0.252416	0.1728105	0.368
1600	0.024001	0.512032	0.288018	0.83
1600	0.028001	0.492031	0.244015	0.74
1600	0.032002	0.48803	0.264017	0.78
1600	0.032002	0.504032	0.224013	0.82
1600	0.036002	0.492031	0.248015	0.74
1600	0.028001	0.500032	0.232014	0.74
1600	0.036002	0.516032	0.248016	0.8
1600	0.032002	0.504031	0.256016	0.79
1600	0.040002	0.504032	0.264016	0.79
1600	0.036002	0.500031	0.248016	0.78
1600 Average	0.0324017	0.5012314	0.2516156	0.781

Table A.1: Benchmark data, Treewidth 4

Size	Parsing	Tree Decomposition	dynASP	DLV
2100	0.048003	0.836052	0.332025	1.33
2100	0.052003	0.836052	0.316025	1.3
2100	0.040002	0.872055	0.312023	1.34
2100	0.048003	0.828051	0.284016	1.22
2100	0.056003	0.824051	0.292016	1.22
2100	0.056003	0.836052	0.348025	1.3
2100	0.052003	0.860054	0.308023	1.35
2100	0.044002	0.848053	0.284015	1.31
2100	0.044002	0.840053	0.320025	1.22
2100	0.048003	0.844052	0.308025	1.3
2100 Average	0.0488027	0.8424525	0.3104218	1.289
2600	0.052003	1.27608	0.388027	2.35
2600	0.064004	1.26408	0.356016	2
2600	0.064004	1.26808	0.364026	2
2600	0.072004	1.25608	0.456026	2.28
2600	0.068004	1.30408	0.424026	2.15
2600	0.060003	1.29208	0.420027	2.68
2600	0.064004	1.26008	0.724046	1.9
2600	0.040002	1.69611	0.388018	2.05
2600	0.068004	1.26008	0.272016	2.02
2600	0.060004	1.27608	1.848116	2.62
2600 Average	0.0612036	1.315283	0.5640344	2.205
3100	0.084005	1.78011	0.460035	2.87
3100	0.076004	1.76411	1.188076	3.6
3100	0.084005	1.82811	1.276085	2.76
3100	0.080005	1.77611	0.468035	2.8
3100	0.076004	1.72811	1.000056	3.34
3100	0.092005	1.80411	0.456035	2.69
3100	0.084005	1.73611	0.440025	3.7
3100	0.080005	1.78011	1.460095	3.79
3100	0.080005	1.78411	0.504035	2.8
3100	0.076004	1.78011	1.260076	3.66
3100 Average	0.0812047	1.77611	0.8512553	3.201

Table A.2: Benchmark data, Treewidth 4, continued

Size	Parsing	Tree Decomposition	dynASP	DLV
3600	0.104006	2.30414	0.516034	3.58
3600	0.092005	2.34415	1.384085	5.47
3600	0.124007	2.35215	0.560033	3.73
3600	0.104006	2.35615	0.508034	5.19
3600	0.100006	2.63616	1.056074	5.4
3600	0.108006	2.34415	0.524034	3.81
3600	0.112007	2.34815	1.440083	5.32
3600	0.088005	2.34415	2.856175	5.14
3600	0.104006	2.38015	0.580034	5.36
3600	0.100006	2.78417	0.504034	3.71
3600 Average	0.103606	2.419352	0.992862	4.671
4100	0.120007	3.07219	2.260143	7.11
4100	0.120007	3.04019	1.476093	6.86
4100	0.116007	3.79224	1.072063	4.93
4100	0.120007	3.00819	1.496093	6.75
4100	0.124007	3.37221	0.732043	4.94
4100	0.112007	3.05619	1.928123	7.24
4100	0.124008	2.97619	1.660102	7.15
4100	0.120007	3.05619	0.616043	6.98
4100	0.116007	3.05219	1.920123	6.84
4100	0.132008	3.27621	0.824042	5.06
4100 Average	0.1204072	3.170199	1.3984868	6.386
4600	0.120007	3.83224	2.244143	9.16
4600	0.144009	3.84824	1.680101	8.68
4600	0.128008	3.83624	1.424092	9.96
4600	0.136008	4.52428	1.388092	9.44
4600	0.132008	4.06025	1.948122	8.69
4600	0.140009	3.78824	2.136131	8.95
4600	0.124008	3.92024	0.788052	8.8
4600	0.144009	4.50428	2.484151	8.87
4600	0.132008	3.99625	1.420092	9.5
4600	0.132008	3.78024	0.608032	9.58
4600 Average	0.1332082	4.00905	1.6121008	9.163

Table A.3: Benchmark data, Treewidth 4, continued

Size	Parsing	Tree Decomposition	dynASP	DLV
5100	0.152009	4.90431	1.164071	9.88
5100	0.144009	4.66029	3.860241	10.67
5100	0.16801	4.55628	1.78812	10.12
5100	0.152009	4.7723	1.660101	11.59
5100	0.136008	5.52034	1.168082	11.06
5100	0.156009	5.33633	1.088071	10.66
5100	0.148009	5.37234	1.652101	11.72
5100	0.16801	5.31233	1.6441	11.42
5100	0.156009	4.96831	1.352081	10.99
5100	0.144009	4.8603	4.752301	10.92
5100 Average	0.1524091	5.026313	2.0129269	10.903
5600	0.188011	5.99637	2.364149	14.31
5600	0.196012	6.88443	1.096068	13.41
5600	0.204012	5.95237	2.064128	13.53
5600	0.196012	6.3804	3.788188	13.74
5600	0.220014	5.90037	1.960126	12.82
5600	0.208013	6.30039	1.464097	13.93
5600	0.204013	6.32439	1.932127	13.5
5600	0.184011	5.54035	2.928179	13.85
5600	0.208013	6.00438	2.776167	13.03
5600	0.204012	6.81243	0.904048	13.08
5600 Average	0.2012123	6.209588	2.1277277	13.52
6100	0.220013	8.0605	1.204077	15.71
6100	0.228014	7.26445	3.032236	15.16
6100	0.224014	7.28045	2.088136	15.92
6100	0.216013	7.08044	2.264147	16.89
6100	0.236014	7.22445	1.888116	15.66
6100	0.240015	7.53247	1.964125	16.72
6100	0.212013	7.44046	1.820117	16.02
6100	0.204012	7.02844	1.728108	17.49
6100	0.232014	8.45653	1.400056	16.13
6100	0.244015	7.28846	1.828115	15.76
6100 Average	0.2256137	7.465665	1.9217233	16.146

Table A.4: Benchmark data, Treewidth 4, continued

Size	Parsing	Tree Decomposition	dynASP	DLV
600	0.048003	0.388024	1.6841	0.5
600	0.040002	0.372023	1.00406	0.47
600	0.032002	0.368023	1.15207	0.43
600	0.036002	0.388024	1.04407	0.49
600	0.036002	0.376023	0.984062	0.46
600	0.040002	0.384024	0.744046	0.48
600	0.032002	0.380024	1.5361	0.46
600	0.048003	0.388024	1.10007	0.48
600	0.040002	0.372023	0.984061	0.44
600	0.036002	0.384024	1.5561	0.47
600 Worst-Case	0.048003	0.388024	1.6841	0.5
1100	0.096006	1.07207	4.26027	2.11
1100	0.092006	1.10407	3.02419	1.62
1100	0.076005	1.08407	2.96019	1.58
1100	0.088005	1.10407	2.52416	1.59
1100	0.084005	1.07207	2.64017	1.54
1100	0.088005	1.08407	1.89612	3.07
1100	0.084005	1.09607	2.89218	2.12
1100	0.092005	1.08407	2.64817	2.44
1100	0.076004	1.09207	4.19226	2.17
1100	0.124008	1.34408	2.04413	2.21
1100 Worst-Case	0.124008	1.34408	4.26027	3.07
1600	0.16001	2.89218	8.12051	4.79
1600	0.152009	2.10013	6.61641	5
1600	0.16401	2.15213	3.84824	4.12
1600	0.156009	2.09613	7.02044	4.75
1600	0.16401	2.10813	3.30021	5.59
1600	0.17201	2.17214	6.4764	5.54
1600	0.144009	2.26414	4.08026	5.06
1600	0.152009	2.18414	6.23239	5.62
1600	0.176011	2.24014	6.49641	5.86
1600	0.136008	2.19214	6.30039	4.54
1600 Worst-Case	0.176011	2.89218	8.12051	5.86

Table A.5: Benchmark data, Treewidth 7

Size	Parsing	Tree Decomposition	dynASP	DLV
2100	0.17201	4.54828	5.44034	8.67
2100	0.192012	3.65223	5.72436	8.22
2100	0.224014	3.59622	8.59254	8.94
2100	0.196012	4.23627	9.15257	9.06
2100	0.204012	5.02431	8.0045	8.47
2100	0.200012	4.58429	5.77236	8.86
2100	0.192012	4.57629	8.42853	8.51
2100	0.208013	4.52028	8.36852	10
2100	0.204013	4.50428	8.11651	8.82
2100 Worst-Case	0.224014	5.02431	9.15257	10
2600	0.244015	6.82043	10.2566	13.44
2600	0.252015	7.18845	10.9047	13.2
2600	0.272017	6.66042	7.12444	13.23
2600	0.248015	7.02444	6.56841	13.54
2600	0.268016	6.78842	11.0287	13.12
2600	0.244015	7.11244	4.34427	14.23
2600	0.248015	6.65642	7.14845	13.76
2600	0.260016	6.78843	7.06044	13.42
2600	0.260016	6.55641	10.4447	13.67
2600	0.284017	7.25245	10.9207	14.02
2600 Worst-Case	0.284017	7.25245	11.0287	14.23
3100	0.284017	9.69661	12.6008	18.66
3100	0.280017	9.69661	12.9208	18.58
3100	0.312019	9.71661	12.6968	19.55
3100	0.296018	9.6286	8.50453	19.68
3100	0.356022	10.1926	8.13651	19.3
3100	0.32002	10.3286	12.6168	19.16
3100	0.300018	9.81261	13.4008	19.26
3100	0.32002	10.1966	12.8328	19.44
3100	0.260016	9.84461	12.8768	18.9
3100	0.32002	9.6046	9.12857	19.74
3100 Worst-Case	0.356022	10.3286	13.4008	19.74

Table A.6: Benchmark data, Treewidth 7, continued

B SHARP Class List

- `Problem`-the class providing flow control and acting as the interface to the "outside world"
- `AbstractAlgorithm`-the algorithm skeleton for tree-based algorithms
- `Tuple`-skeleton class, represents the data the algorithm needs to represent one possible world in a tree node
- `SolutionContent`-skeleton class, represents the solution data associated with one possible world (i.e. `Tuple`)
- `Solution`-framework helper class that provides lazy solution evaluation support
- `Instantiator`-framework class for instantiating the `Solution` class and the `SolutionContent` classes
- `GenericInstantiator`–framework class that provides a generic instantiator supporting lazy evaluation and supports all derived `SolutionContent` classes that implement the skeleton constructors of the `SolutionContent` class
- `EnumerationSolutionContent`–`SolutionContent` class that a set of sets of vertices (i.e. multiple (partial) solutions per `Tuple` may be possible here as one solution is represented by a set of vertices)
- `ConsistencySolutionContent`–`SolutionContent` class that stores a simple boolean value representing the answers "yes" or "no"
- `CountingSolutionContent`–`SolutionContent` class that stores an arbitrarily large number for counting purposes
- `Hypertree`-represents a tree
- `ExtendedHypertree`-represents a tree with specifically extended functionality (i.e. vertex-aware, normalization capabilities, etc.) for easier use in the algorithm
- `Hypergraph`-represents a graph

- `Hyperedge`-represents an edge in a graph
- `Node`-represents a vertex in a graph

Bibliography

[Apt et al., 1988] Apt, K. R., Blair, H. A., and Walker, A. (1988). Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming.*, pages 89–148. Morgan Kaufmann.

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

[Atserias et al., 2009] Atserias, A., Fichte, J. K., and Thurley, M. (2009). Clause-learning algorithms with many restarts and bounded-width resolution. In [Kullmann, 2009], pages 114–127.

[Bachoore and Bodlaender, 2005] Bachoore, E. H. and Bodlaender, H. L. (2005). New upper bound heuristics for treewidth. In Nikoletseas, S. E., editor, *Experimental and Efficient Algorithms, 4th International Workshop, WEA 2005, Santorini Island, Greece. Proceedings*, volume 3503 of *Lecture Notes in Computer Science*, pages 216–227. Springer.

[Badros et al., 2001] Badros, G. J., Borning, A., and Stuckey, P. J. (2001). The cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction*, 8(4):267–306.

[Baral, 2003] Baral, C. (2003). *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

[Beierle and Kern-Isberner, 2008] Beierle, C. and Kern-Isberner, G. (2008). *Methoden Wissensbasierter Systeme*. Vieweg+Teubner, 4 edition.

[Ben-Eliyahu and Dechter, 1994] Ben-Eliyahu, R. and Dechter, R. (1994). Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12(1-2):53–87.

[Bodlaender, 1993a] Bodlaender, H. L. (1993a). A linear time algorithm for finding tree-decompositions of small treewidth. In *STOC*, pages 226–234.

[Bodlaender, 1993b] Bodlaender, H. L. (1993b). A tourist guide through treewidth. *Acta Cybernetica*, 11(1-2):1–22.

[Bodlaender, 1997] Bodlaender, H. L. (1997). Treewidth: Algorithmoc techniques and results. In Prívara, I. and Ruzicka, P., editors, *Mathematical Foundations of Computer Science 1997, 22nd International Symposium, MFCS'97, Bratislava, Slovakia. Proceedings*, volume 1295 of *Lecture Notes in Computer Science*, pages 19–36. Springer.

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

[Chekuri and Rajaraman, 2000] Chekuri, C. and Rajaraman, A. (2000). Conjunctive query containment revisited. *Journal of Theoretical Computer Science*, 239(2):211–229.

[Church, 1936] Church, A. (1936). A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41.

[Clautiaux et al., 2004] Clautiaux, F., Moukrim, A., Négre, S., and Carlier, J. (2004). Heuristic and metaheuristic methods for computing graph treewidth. In *RAIRO Operations Research*, volume 4, pages 13–26.

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM.

[Courcelle, 1990] Courcelle, B. (1990). Graph rewriting: An algebraic and logic approach. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 193–242. Elsevier and MIT Press.

[Dantsin et al., 2001] Dantsin, E., Eiter, T., Gottlob, G., and Voronkov, A. (2001). Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425.

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

[Dermaku et al., 2008] Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B. J., Musliu, N., and Samer, M. (2008). Heuristic methods for hypertree decomposition. In Gelbukh, A. F. and Morales, E. F., editors, *MICAI 2008: Advances in Artificial Intelligence, 7th Mexican International Conference on Artificial Intelligence, Atizapán de Zaragoza, Mexico. Proceedings*, volume 5317 of *Lecture Notes in Computer Science*, pages 1–11. Springer.

[Dix, 1995] Dix, J. (1995). A classification theory of semantics of normal logic programs: I. Strong properties. *Fundamenta Informaticae*, 22(3):227–255.

[Dix et al., 1996] Dix, J., Gottlob, G., and Marek, V. W. (1996). Reducing disjunctive to non-disjunctive semantics by shift-operations. *Fundamenta Informaticae*, 28(1-2):87–100.

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

[Dvorák et al., 2010] Dvorák, W., Pichler, R., and Woltran, S. (2010). Towards fixed-parameter tractable algorithms for argumentation. In Lin, F., Sattler, U., and Truszczynski, M., editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Canada*. AAAI Press.

[Eén and Sörensson, 2003] Eén, N. and Sörensson, N. (2003). An extensible sat-solver. In Giunchiglia, E. and Tacchella, A., editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy. Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer.

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

[Eiter et al., 2006] Eiter, T., Ianni, G., Polleres, A., Schindlauer, R., and Tompits, H. (2006). Reasoning with rules and ontologies. In Barahona, P., Bry, F., Franconi, E., Henze, N., and Sattler, U., editors, *Reasoning Web, Second International Summer School 2006, Lisbon, Portugal, Tutorial Lectures*, volume 4126 of *Lecture Notes in Computer Science*, pages 93–127. Springer.

[Erdem et al., 2009] Erdem, E., Lin, F., and Schaub, T., editors (2009). *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*. Springer.

[Flum and Grohe, 2006] Flum, J. and Grohe, M. (2006). *Computational Complexity*. Springer.

[Gebser et al., 2007] Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. (2007). *c*lasp : A conflict-driven answer set solver. In Baral, C., Brewka, G., and Schlipf, J. S., editors, *Logic Programming and Nonmonotonic Reasoning, 9th International*

Conference, LPNMR 2007, Tempe, AZ, USA. Proceedings, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer.

[Gelder et al., 1991] Gelder, A. V., Ross, K. A., and Schlipf, J. S. (1991). The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650.

[Gelfond and Leone, 2002] Gelfond, M. and Leone, N. (2002). Logic programming and knowledge representation - the a-prolog perspective. *Artificial Intelligence*, 138(1-2):3–38.

[Gelfond and Lifschitz, 1988] Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080.

[Gelfond and Lifschitz, 1990] Gelfond, M. and Lifschitz, V. (1990). Logic programs with classical negation. In *ICLP*, pages 579–597.

[Gelfond and Lifschitz, 1991] Gelfond, M. and Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computation*, 9(3/4):365–386.

[Gottlob et al., 1999] Gottlob, G., Leone, N., and Scarcello, F. (1999). Hypertree decompositions and tractable queries. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1999, Philadelphia, Pennsylvania*, pages 21–32. ACM Press.

[Gottlob et al., 2000] Gottlob, G., Leone, N., and Scarcello, F. (2000). A comparison of structural csp decomposition methods. *Artificial Intelligence*, 124(2):243–282.

[Gottlob et al., 2001] Gottlob, G., Leone, N., and Scarcello, F. (2001). The complexity of acyclic conjunctive queries. *Journal of the ACM*, 48(3):431–498.

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

[Grasso et al., 2009] Grasso, G., Iiritano, S., Leone, N., and Ricca, F. (2009). Some dlv applications for knowledge management. In [Erdem et al., 2009], pages 591–597.

[Huth and Ryan, 2004] Huth, M. and Ryan, M. (2004). *Logic in Computer Science*. Cambridge University Press, 2 edition.

[Jackson and Sheridan, 2004] Jackson, P. and Sheridan, D. (2004). Clause form conversions for boolean circuits. In Hoos, H. H. and Mitchell, D. G., editors, *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, Canada. Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 183–198. Springer.

[Jakl, 2010] Jakl, M. (2010). *Fixed Parameter Algorithms for Answer Set Programming*. PhD thesis, Faculty of Computer Science, Vienna University of Technology, Austria.

[Jakl et al., 2009] Jakl, M., Pichler, R., and Woltran, S. (2009). Answer-set programming with bounded treewidth. In Boutilier, C., editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA*, pages 816–822.

[Kloks, 1994] Kloks, T. (1994). *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer.

[Koster et al., 2001] Koster, A. M. C. A., Bodlaender, H. L., and van Hoesel, S. P. M. (2001). Treewidth: Computational experiments. *Electronic Notes in Discrete Mathematics*, 8:54–57.

[Köster et al., 2009] Köster, M., Novák, P., Mainzer, D., and Fuhrmann, B. (2009). Two case studies for jazzyk bsm. In Dignum, F., Bradshaw, J. M., Silverman, B. G., and van Doesburg, W. A., editors, *Agents for Games and Simulations, Trends in Techniques, Concepts and Design, AGS 2009, The First International Workshop on Agents for Games and Simulations, Budapest, Hungary. Proceedings*, volume 5920 of *Lecture Notes in Computer Science*, pages 33–47. Springer.

[Kullmann, 2009] Kullmann, O., editor (2009). *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*. Springer.

[Leone et al., 2006] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562.

[Lifschitz, 2002] Lifschitz, V. (2002). Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54.

[Lin and Zhao, 2004] Lin, F. and Zhao, Y. (2004). Assat: computing answer sets of a logic program by sat solvers. *Artificial Intelligence*, 157(1-2):115–137.

[Marek and Truszczynski, 1991] Marek, V. W. and Truszczynski, M. (1991). Autoepistemic logic. *Journal of the ACM*, 38(3):588–619.

[Marek and Truszczyński, 1999] Marek, V. W. and Truszczyński, M. (1999). Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. Springer.

[McMahan, 2004] McMahan, B. J. (2004). Bucket elimination and hypertree decompositions. implementation report. Technical report, Institute for Information Systems, Vienna University of Technology.

[Morak et al., 2010] Morak, M., Pichler, R., Rümmele, S., and Woltran, S. (2010). A dynamic-programming based asp-solver. In Janhunen, T. and Niemelä, I., editors, *Logics in Artificial Intelligence - 12th European Conference, JELIA 2010, Helsinki, Finland. Proceedings*, volume 6341 of *Lecture Notes in Computer Science*, pages 369–372. Springer.

[Nemhauser and Trotter, 1975] Nemhauser, G. L. and Trotter, L. E. (1975). Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8(1):232–248.

[Niedermeier, 2006] Niedermeier, R. (2006). *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics and Its Applications. Oxford University Press.

[Niemelä, 1999] Niemelä, I. (1999). Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273.

[Novák, 2009] Novák, P. (2009). *Behavioural State Machines: Agent Programming and Engineering*. PhD thesis, Faculty of Mathematics/Computer Science and Mechanical Engineering, Clausthal University of Technology, Germany.

[Palù et al., 2009] Palù, A. D., Dovier, A., Pontelli, E., and Rossi, G. (2009). Gasp: Answer set programming with lazy grounding. *Fundamenta Informaticae*, 96(3):297–322.

[Papadimitriou, 1993] Papadimitriou, C. H. (1993). *Parameterized Complexity Theory*. Addison Wesley.

[Pichler et al., 2009] Pichler, R., Rümmele, S., and Woltran, S. (2009). Belief revision with bounded treewidth. In [Erdem et al., 2009], pages 250–263.

[Pichler et al., 2010] Pichler, R., Rümmele, S., and Woltran, S. (2010). Multicut algorithms via tree decompositions. In Calamoneri, T. and Díaz, J., editors, *Algorithms and Complexity, 7th International Conference, CIAC 2010, Rome, Italy. Proceedings*, volume 6078 of *Lecture Notes in Computer Science*, pages 167–179. Springer.

[Provetti and Son, 2001] Provetti, A. and Son, T. C., editors (2001). *Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop, Stanford, March 26-28, 2001*.

[Reiter, 1980] Reiter, R. (1980). A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132.

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

[Rose, 1972] Rose, D. J. (1972). A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In Read, R. C., editor, *Graph Theory and Computing*, pages 183–217. Academic Press.

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

[Schafhauser, 2006] Schafhauser, W. (2006). New heuristic methods for tree decompositions and generalized hypertree decompositions. Master's thesis, Institute for Information Systems, Vienna University of Technology.

[Seymour and Thomas, 1993] Seymour, P. D. and Thomas, R. (1993). Graph searching and a min-max theorem for tree-width. *Journal of Combinatorial Theory*, 58(1):22–33.

[Turing, 1937] Turing, A. M. (1937). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265.