



UNIVERSITÀ DELLA CALABRIA



# UNIVERSITÀ DELLA CALABRIA

Dipartimento di Matematica e Informatica

## Dottorato di Ricerca in Matematica e Informatica

*con il contributo del*

Fondo Sociale Europeo - POR Calabria FSE 2007/2013


XXVII CICLO

---

### COMPUTATIONAL TASKS IN ANSWER SET PROGRAMMING: ALGORITHMS AND IMPLEMENTATION

Settore Disciplinare INF/01 – INFORMATICA

**Coordinatore:** Chia.mo Prof. Nicola Leone

  
\_\_\_\_\_

**Supervisor:** Prof. Francesco Ricca

  
\_\_\_\_\_

Prof. Mario Alviano

  
\_\_\_\_\_

**Dottorando:** Dott. Carmine Dodaro

  
\_\_\_\_\_



UNIVERSITÀ DELLA CALABRIA



# UNIVERSITÀ DELLA CALABRIA

Dipartimento di Matematica e Informatica

## Dottorato di Ricerca in Matematica e Informatica

*con il contributo del*

**Fondo Sociale Europeo - POR Calabria FSE 2007/2013**

XXVII CICLO

---

## COMPUTATIONAL TASKS IN ANSWER SET PROGRAMMING: ALGORITHMS AND IMPLEMENTATION

Settore Disciplinare INF/01 – INFORMATICA

**Coordinatore:** Chia.mo Prof. Nicola Leone

---

**Supervisor:** Prof. Francesco Ricca

---

Prof. Mario Alviano

---

**Dottorando:** Dott. Carmine Dodaro

---

# Ringraziamenti

La presente tesi è cofinanziata con il sostegno della Commissione Europea, Fondo Sociale Europeo e della Regione Calabria. L'autore è il solo responsabile di questa tesi e la Commissione Europea e la Regione Calabria declinano ogni responsabilità sull'uso che potrà essere fatto delle informazioni in essa contenute.



*Sors salutis et virtutis mihi nunc contraria,  
est affectus et defectus semper in angaria.  
Hac in hora sine mora corde pulsum tangite;  
quod per sortem sternit fortem, mecum  
omnes plangite!*

# Acknowledgements

I want to say thanks:

- To Prof. **Francesco Ricca** and To Prof. **Mario Alviano**, for their attention and their patience during these years. The time spent with you was a great pleasure.
- To Prof. **Nicola Leone**, for his interest in my work and his precious suggestions.
- To Prof. **Joao Marques-Silva** and his research group, for their hospitality and their advices when I was in Dublin.
- To the anonymous reviewers for their suggestions.
- To all of my friends and colleagues.
- To my parents **Mario** and **Tina**, my sisters **Giulia** e **Rita** and my grandmother **Rita**, for their support in my life.
- Last but not least to **Susanna**, for being the milestone of my life.

*Carminé*



## Sommario

L'Answer Set Programming (ASP) è un paradigma di programmazione dichiarativa basato sulla semantica dei modelli stabili. L'idea alla base di ASP è di codificare un problema computazionale in un programma logico i cui modelli stabili, anche detti answer set, corrispondono alle soluzioni del problema. L'espressività di ASP ed il numero crescente delle sue applicazioni hanno reso lo sviluppo di nuovi sistemi ASP un tema di ricerca attuale ed importante.

La realizzazione di un sistema ASP richiede di implementare soluzioni efficienti per vari task computazionali. Questa tesi si occupa delle problematiche relative alla valutazione di programmi proposizionali, ed in particolare affronta i task di model generation, answer set checking, optimum answer set search e cautious reasoning. La combinazione dei primi due task corrisponde alla computazione degli answer set. Infatti, il task di model generation consiste nel generare dei modelli del programma in input, mentre il task di answer set checking ha il compito di verificare che siano effettivamente modelli stabili. Il primo task è correlato alla risoluzione di formule SAT, ed è implementato -nelle soluzioni moderne- con un algoritmo di backtracking simile al Conflict-Driven Clause Learning (CDCL); il secondo è risolto applicando una riduzione al problema dell'insoddisfacibilità di una formula SAT. In presenza di costrutti di ottimizzazione l'obiettivo di un sistema ASP è l'optimum answer set search, che corrisponde a calcolare un answer set che minimizza il numero di violazioni dei cosiddetti weak constraint presenti nel programma. Il cautious reasoning è il task principale nelle applicazioni data-oriented di ASP, e corrisponde a calcolare un sottoinsieme degli atomi che appartengono a tutti gli answer set di un programma. Si noti che tutti questi task presentano una elevata complessità computazionale.

I contributi di questa tesi sono riassunti di seguito:

- (I) è stato studiato il task di model generation ed è stata proposta per la sua risoluzione una combinazione di tecniche che sono state originariamente utilizzate per risolvere il problema SAT;
- (II) è stato proposto un nuovo algoritmo per l'answer set checking che minimizza l'overhead dovuto all'esecuzione di chiamate multiple ad un oracolo co-NP. Tale algoritmo si basa su una strategia di valutazione incrementale ed euristiche progettate specificamente per migliorare l'efficienza della risoluzione di tale problema;
- (III) è stata proposta una famiglia di algoritmi per il calcolo di answer set ottimi di programmi con weak constraint. Tali soluzioni sono state ottenute adattando algoritmi proposti per risolvere il problema MaxSAT;



(IV) è stato introdotto un nuovo framework di algoritmi anytime per il cautious reasoning in ASP che estende le proposte esistenti ed include un nuovo algoritmo ispirato a tecniche per il calcolo di backbone di teorie proposizionali.

Queste tecniche sono state implementate in WASP 2, un nuovo sistema ASP per programmi proposizionali. L'efficacia delle tecniche proposte e l'efficienza del nuovo sistema sono state valutate empiricamente su istanze utilizzate nella competizioni per sistemi ASP e messe a disposizione sul Web.



## Abstract

Answer Set Programming (ASP) is a declarative programming paradigm based on the stable model semantics. The idea of ASP is to encode a computational problem into a logic problem, whose stable models correspond to the solution of the original problem. The high expressivity of ASP, combined with the growing number of applications, made the implementation of new ASP solvers a challenging and crucial research topic.

The implementation of an ASP solver requires to provide solutions for several computational tasks. This thesis focuses on the ones related to reasoning with propositional ASP programs, such as model generation, answer set checking, optimum answer set search, and cautious reasoning. The combination of the first two tasks is basically the computation of answer sets. Indeed, model generation amounts to generating models of the input program, whose stability is subsequently verified by calling an answer set checker. Model generation is similar to SAT solving, and it is usually addressed by employing a CDCL-like backtracking algorithm. Answer set checking is a co-NP complete task in general, and is usually reduced to checking the unsatisfiability of a SAT formula. In presence of optimization constructs the goal of an ASP solver becomes optimum answer set search, and requires to find an answer set that minimizes the violations of the so-called weak constraints. In data-oriented applications of ASP cautious reasoning is often the main task, and amounts to the computation of (a subset of) the certain answers, i.e., those that belong to all answer sets of a program.

These tasks are computationally very hard in general, and this thesis provides algorithms and solutions to solve them efficiently. In particular the contributions of this thesis can be summarized as follows:

1. The task of generating model candidates has been studied, and a combination of techniques, which were originally introduced for SAT solving has been implemented in a new ASP solver.
2. A new algorithm for answer set checking has been proposed that minimizes the overhead of executing multiple calls to a co-NP oracle by resorting to an incremental evaluation strategy and specific heuristics.
3. A family of algorithms for computing optimum answer sets of programs with weak constraints has been implemented by porting to the ASP setting several algorithms introduced for MaxSAT solving.

4. A new framework of anytime algorithms for computing the cautious consequences of an ASP knowledge base has been introduced, that extends existing proposals and includes a new algorithm inspired by techniques for the computation of backbones of propositional theories.

These techniques have been implemented in WASP 2, a new solver for propositional ASP programs. The effectiveness of the proposed techniques and the performance of the new system have been validated empirically on publicly-available benchmarks taken from ASP competitions and other repositories of ASP applications.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>SAT Solving</b>	<b>5</b>
2.1	The Satisfiability Problem . . . . .	5
2.2	Davis Putnam Logemann Loveland Algorithm . . . . .	6
2.3	Conflict-Driven Clause Learning . . . . .	8
2.3.1	Learning . . . . .	9
2.3.2	Branching Heuristics . . . . .	10
2.3.3	Learned Clauses Deletion . . . . .	12
2.3.4	Restarts . . . . .	12
2.4	Incremental SAT . . . . .	13
2.5	Maximum Satisfiability . . . . .	15
<b>3</b>	<b>Answer Set Programming</b>	<b>17</b>
3.1	Syntax . . . . .	17
3.2	Semantics . . . . .	19
3.3	Properties of ASP programs . . . . .	21
3.4	Knowledge Representation And Reasoning . . . . .	25
3.5	Architecture of an ASP System . . . . .	27
<b>4</b>	<b>The ASP Solver WASP 2</b>	<b>28</b>
4.1	The Architecture of WASP 2 . . . . .	28
4.2	Input Processor . . . . .	29
4.3	Simplifications of the Input Program . . . . .	29
4.4	Model Generator . . . . .	30
4.4.1	Propagation . . . . .	32
4.4.2	Learning . . . . .	36
4.5	Answer Set Checker . . . . .	36
4.5.1	Unfounded-free Check for non-HCF components . . . . .	37
4.5.2	Partial Checks . . . . .	40
4.6	Experiments . . . . .	42

4.6.1	Model Generator Evaluation . . . . .	42
4.6.2	Answer Set Checker Evaluation . . . . .	45
<b>5</b>	<b>Optimum Answer Set Search</b>	<b>47</b>
5.1	Preliminaries . . . . .	47
5.2	Algorithm OPT . . . . .	48
5.2.1	Algorithm BASIC. . . . .	49
5.3	Algorithm MGD . . . . .	50
5.4	Algorithm OLL . . . . .	51
5.5	Algorithm PMRES . . . . .	53
5.6	Algorithm BCD . . . . .	55
5.7	Implementation . . . . .	57
5.8	Experiments . . . . .	58
<b>6</b>	<b>Query Answering</b>	<b>62</b>
6.1	Computation of Cautious Consequences . . . . .	63
6.2	Correctness . . . . .	65
6.3	Experiments . . . . .	67
6.3.1	Implementation . . . . .	67
6.3.2	Benchmark Settings . . . . .	67
6.3.3	Discussion of the Results . . . . .	68
<b>7</b>	<b>Related Work</b>	<b>73</b>
7.1	Relations with SAT, MaxSAT and SMT . . . . .	73
7.2	ASP Solvers . . . . .	74
7.2.1	Solvers Based on Translations . . . . .	74
7.2.2	Native Solvers . . . . .	75
<b>8</b>	<b>Conclusion</b>	<b>78</b>

# Chapter 1

## Introduction

During the recent years the number of computer applications have grown exponentially and most of the computational problems of our lives are handled automatically. However, many of these problems are not easily solvable by a computer, especially those for which a deterministic polynomial time algorithm is unknown. The traditional approach for solving this kind of problems is based on the imperative programming paradigm. The major drawback of this approach is that high-level programming skills as well as deep domain knowledge are required in order to find a good algorithm for a hard problem. Moreover, usually small changes in the specification of the problem require a lot of effort for adapting the implementation to the new specifications. An alternative approach to solve these problems is based on declarative programming. In this case, the problem and its solutions are stated in the form of an executable specification, i.e. the problem is solved by stating the features of its solution, rather than specifying how a solution has to be obtained.

**Answer Set Programming.** One of the major declarative programming paradigms based on logic is Answer Set Programming (ASP) [1, 2, 3, 4, 5, 6, 7]. In ASP, knowledge concerning an application domain is encoded by a logic program whose semantics is given by a set of stable models [6], also referred to as answer sets. The core language of ASP, which features disjunction in rule heads and nonmonotonic negation in rule bodies, can express all problems in the second level of the polynomial hierarchy [2]. Therefore, ASP is strictly more expressive than SAT (unless the polynomial hierarchy collapses). Nonetheless, several extensions to the original language were proposed over the years to further improve ASP modeling capabilities, such as aggregates [8] for concise modeling of properties over sets of data, and weak constraints [9] for modeling optimization problems.



**Motivations and Contributions.** The high expressivity of the original language combined with its extensions made ASP a powerful tool for developing advanced applications in the areas of Artificial Intelligence, Information Integration, and Knowledge Management; for example, ASP has been used in industrial applications [10], team-building [11], semantic-based information extraction [12], linux package configuration [13], bioinformatics [14], assisted living [15] and e-tourism [16]. These applications have confirmed the viability of the use of ASP, and at the same time outlined the need for efficient ASP implementations. After twenty years of research many efficient ASP systems have been developed [17, 18, 19, 20, 21], and the improvements obtained in this respect are witnessed by the results of the ASP Competition series [22, 23]. Nonetheless, ASP applications demand better and better performance in hard-to-solve problems, and thus the development of more effective and faster ASP systems remains a crucial and challenging research topic.

The implementation of an efficient ASP solver requires to provide effective solutions for several computational tasks. In particular, this thesis focuses on those related to reasoning with propositional ASP programs that have great impact in applications, such as model generation, answer set checking, optimum answer set search, and cautious reasoning.

The combination of the first two tasks basically corresponds to the computation of answer sets. Indeed, the model generation task has the goal of computing model candidates of the input program, whose stability is subsequently verified by a co-NP check performed by the answer set checker. The generation of answer set candidates is a hard task performed by applying techniques introduced for SAT solving, such as *learning* [24], *restarts* [25] and *conflict-driven heuristics* [26]. Answer set checking is a co-NP complete task in general, and it is usually reduced to checking the unsatisfiability of a SAT formula [27]. Optimum answer set search is the main task of an ASP solver in presence of optimization constructs, and requires to find an answer set that minimizes the violations of the so-called weak constraints. Moreover, cautious reasoning corresponds to the computation of a (subset) of the certain answers, i.e., those that belong to all answer sets of a program. The latter is often the main task in data-oriented applications of ASP [28, 29, 30].

These computational tasks are very hard in general, and this thesis provides algorithms and solutions to solve them efficiently. In particular the contributions of this thesis can be summarized as follows:

- The task of generating model candidates has been studied, and a combination of techniques that were originally introduced for SAT solving, such as *learning* [24], *restarts* [25] and *conflict-driven heuristics* [26]

has been implemented in the new ASP solver WASP 2.

- A new algorithm for answer set checking has been proposed that minimizes the overhead due to multiple calls to an external oracle [27] by resorting to an incremental solving strategy.
- A family of algorithms for computing optimum answer sets of programs with weak constraints [9] has been studied and implemented; these algorithms were obtained by porting to the ASP setting several solutions proposed for MaxSAT solving [31, 32, 33].
- A new framework of anytime algorithms for computing the cautious consequences of an ASP knowledge base has been proposed, that extends existing proposals and includes a new algorithm inspired by techniques for the computation of backbones of propositional theories [34].

**Model Generation.** The model generation task is related to SAT solving, and modern ASP solvers are actually based on CDCL-like algorithms [20] properly adapted in order to take into account the properties and the extensions of ASP programs. Thus, our system extends the CDCL algorithm by implementing several techniques for handling efficiently specific features of ASP programs, such as minimality of answer sets and inference via aggregates. The performances of our implementation have been assessed by an experimental analysis on the instances of the Fourth ASP Competition [23]. The results show that our system is competitive compared with alternative solutions.

**Answer Set Checking.** The second contribution regards answer set checking, which is a co-NP-complete problem for disjunctive logic programs. In fact, a polynomial algorithm for reducing the answer set checking problem into the unsatisfiability problem has been proposed in [27]. However, practical applications have shown the drawbacks of such approach, mostly related to the creation of a new SAT formula for each stability check. We have improved the original algorithm by exploiting a strategy based on incremental solving which minimizes the overhead due to multiple calls to an external oracle. An experimental analysis confirms the viability of our algorithm, which is already comparable with alternative solutions.

**Optimum Model Search.** Optimum model search is the main task in case of optimization programs in ASP. The task is addressed in this thesis by adapting some of the techniques introduced for MaxSAT solving. In

particular, several algorithms introduced for MaxSAT solving, named MGD [31], OPTSAT [32], PMRES [35] and BCD [36] have been properly adapted to the ASP setting. Moreover, the algorithm OLL [33] and BASIC have been also implemented. The former has been introduced for ASP solving and then successfully applied to MaxSAT solving [37], while the latter has been implemented in the ASP solvers SMOBELS [18], DLV [17] and CLASP [20].

**Cautious Reasoning.** Concerning cautious reasoning, we designed and implemented a new framework of *anytime* algorithms for cautious reasoning in ASP. An algorithm is said to be anytime if it produces valid, intermediate solutions, during its execution, thus it can be safely terminated before the end if the quality of the latest found solution is satisfactory. Our algorithms produce certain answers during the computation of the complete solution. The computation can thus be stopped either when a sufficient number of cautious consequences have been produced, or when no new answer is produced after a specified amount of time. Since cautious consequences computation is very hard, anytime property of our algorithms is crucial for real world applications. In fact, we empirically verified that a large number of certain answers can be produced after a few seconds of computation even when the full set of cautious consequences is not computable in reasonable time.

**Organization of the Thesis.** The remainder of the thesis is organized as follows: In Chapter 2 we describe the techniques originally introduced for SAT solving, which we have been extended to adapt them in a native ASP solver. In Chapter 3 we describe the syntax and semantics of ASP and the use of ASP as a powerful knowledge representation and reasoning tool. In Chapter 4 we describe the solving strategy of new ASP solver and the new algorithm for stable model checking. We also present the results of the experiments conducted for assessing the performance of our implementation. In Chapter 5 we compare several strategies for solving ASP optimization problems by introducing several MaxSAT algorithms in the context of ASP. In Chapter 6 we describe algorithms and implementation of cautious reasoning and we show anytime variants of the algorithms reported in literature. In Chapter 7 we describe the work related to this thesis. Finally, Chapter 8 draws the conclusions of the thesis.

# Chapter 2

## SAT Solving

The Satisfiability (SAT) problem [38] and the Maximum Satisfiability problem are well-known hard problems. In the recent years effective techniques for solving SAT and MaxSAT have been proposed, and successfully applied to obtain efficient ASP implementations. This chapter provides an overview on the solving techniques used in modern SAT and MaxSAT solvers. In particular, the classical Davis Putnam Logemann Loveland (DPLL) algorithm [39] is presented. After that, its evolution, called Conflict-Driven Clause Learning (CDCL) [40], is described together with some of the techniques that are the core of CDCL, such as branching heuristics, learning and restarts. Finally, incremental SAT solving and its usage for solving MaxSAT are recalled.

### 2.1 The Satisfiability Problem

Let  $V = \{v_1, \dots, v_n\}$  be a finite set of Boolean variables. A *literal*  $\ell$  is a Boolean variable  $v$  or its negation  $\neg v$ . Given a literal  $\ell$ , its negation  $\neg\ell$  is  $\neg v$  if  $\ell = v$  and it is  $v$  if  $\ell = \neg v$ . A *clause*  $\{\ell_1, \dots, \ell_n\}$  is a disjunction of literals. A SAT instance expressed in *Conjunctive Normal Form (CNF)* is a conjunction of clauses. A set of literals  $L$  is said to be *consistent* if, for every literal  $\ell \in L$ ,  $\neg\ell \notin L$  holds. An interpretation is a consistent set of literals. Given an interpretation  $I$  and a literal  $\ell$ :

- $\ell$  is *true* w.r.t.  $I$  if  $\ell \in I$ ;
- $\ell$  is *false* w.r.t.  $I$  if  $\neg\ell \in I$ ;
- $\ell$  is *undefined* if it is neither true nor false w.r.t.  $I$ .

A clause  $\{\ell_1, \dots, \ell_n\}$  is said to be *satisfied* if at least one literal among  $\ell_1, \dots, \ell_n$  is true. A clause is said to be *violated* if all literals  $\ell_1, \dots, \ell_n$  are false. A clause is *undefined* if it is neither satisfied nor violated. An interpretation  $I$  is a *satisfying variable assignment* (or *model*) for a CNF formula  $\varphi$  if all clauses of  $\varphi$  are satisfied w.r.t.  $I$ . In this case  $\varphi$  is said to be *satisfiable*. Otherwise, if no interpretation is a model of  $\varphi$  then  $\varphi$  is said to be *unsatisfiable*. The SAT problem consists of checking whether a given CNF formula  $\varphi$  is satisfiable.

**Example 1** (Satisfiable CNF formula). Let  $V = \{v_1, v_2, v_3\}$  be a set of Boolean variables. Consider the following CNF formula  $\varphi$ :

$$\begin{array}{cc} \{v_1, v_2\} & \{v_2, v_3\} \\ \{v_1, v_3\} & \{\neg v_2, v_3\} \end{array}$$

Interpretation  $I = \{v_1, \neg v_2, v_3\}$  is a model for  $\varphi$ .  $\triangleleft$

**Example 2** (Unsatisfiable CNF formula). Let  $V = \{v_1, v_2, v_3\}$  be a set of Boolean variables. Consider the following CNF formula  $\varphi$ :

$$\begin{array}{ccc} \{v_1, v_2\} & \{v_2, v_3\} & \{v_2, \neg v_3\} \\ \{v_1, \neg v_2\} & \{\neg v_2, v_3\} & \{\neg v_2, \neg v_3\} \end{array}$$

The formula  $\varphi$  is unsatisfiable because there is no variable assignment that satisfies all the clauses.  $\triangleleft$

## 2.2 Davis Putnam Logemann Loveland Algorithm

SAT solving has recently obtained an increasing interest since both industrial applications and effective SAT solvers are available. Those results have been obtained as improvements of the Davis Putnam Logemann Loveland (DPLL) algorithm [39], which is a complete, backtracking-based algorithm for deciding whether a CNF formula is satisfiable or not. DPLL is reported in pseudo-code in Algorithm 1. In a nutshell, the algorithm runs by setting as true an undefined literal  $\ell$ , simplifies the formula accordingly, and then recursively checks whether the simplified formula is satisfiable or not; in the latter case, the same recursive check is done assuming the literal  $\neg\ell$  as true. In more detail, DPLL takes a formula  $\varphi$  as input and starts from an assignment in which all literals are undefined. Function *UnitPropagation* (line 2) is invoked in order to satisfy unit clauses. An undefined clause is a *unit clause* if it contains only one undefined literal. Intuitively, a unit clause can be

---

**Algorithm 1: DPLL**

---

**Input** : A CNF formula  $\varphi$

**Output**: True, if  $\varphi$  is SAT. False, otherwise.

```
1 begin
2    $\varphi := \text{UnitPropagation}(\varphi);$ 
3    $\varphi := \text{PureLiteralElimination}(\varphi);$ 
4   if all clauses in  $\varphi$  are satisfied then
5     return True;
6   if  $\varphi$  contains an empty clause then
7     return False;
8    $\ell = \text{ChooseLiteral}();$ 
9   return DPLL( $\varphi \wedge \ell$ ) OR DPLL( $\varphi \wedge \neg\ell$ );
```

---

satisfied only if its unique undefined literal is inferred as true. In practice, unit propagation has a tremendous effect on pruning the search space.

**Example 3** (Unit clause). Consider the CNF formula  $\varphi$  of Example 1 and the interpretation  $I = \{\neg v_1\}$ . Clause  $\{v_1, v_2\}$  is *unit* since there is only one undefined literal  $v_2$ . The same consideration holds for clause  $\{v_1, v_3\}$ .  $\triangleleft$

After that, function *PureLiteralElimination* (line 3) is called. This function simplifies clauses containing so-called *pure literals*. A literal  $\ell$  is *pure* if  $\neg\ell$  does not occur in the formula. All clauses in which a pure literal  $\ell$  appears can be satisfied, and thus deleted, by inferring  $\ell$  as true. Since  $\neg\ell$  does not appear anyhow in the formula, satisfiability of the remaining clauses is not affected by this operation. However, albeit this optimization is part of the original DPLL algorithm, modern SAT solvers do not implement pure literal checking due to the computational overhead.

**Example 4** (Pure literal). Consider the CNF formula  $\varphi$  of Example 1. Literals  $v_1$  and  $v_3$  are pure since  $\neg v_1$  and  $\neg v_3$  do not occur in  $\varphi$ .  $\triangleleft$

After calling *UnitPropagation* and *PureLiteralElimination*, satisfiability of the formula is detected (line 5) if all clauses in  $\varphi$  are satisfied; otherwise, if all literals of a clause are false (line 7) then unsatisfiability is detected; finally, if no inferences are possible, an undefined literal  $\ell$ , whose variable is called *branching variable*, is selected according to a heuristic criterion (line 8), and DPLL is called recursively.

---

**Algorithm 2:** Conflict-Driven Clause Learning

---

**Input** : A CNF formula  $\varphi$

**Output:** SAT if  $\varphi$  is satisfiable, UNSAT otherwise

```
1 begin
2   while there are undefined clauses in  $\varphi$  do
3      $\ell := \text{ChooseLiteral}()$ ;
4      $\varphi := \varphi \cup \{\ell\}$ ;
5      $\varphi := \text{UnitPropagation}(\varphi)$ ;
6     while there are violated clauses in  $\varphi$  do
7        $\varphi := \text{LearnClause}(\varphi)$ ;
8       if not  $\text{RestoreConsistency}(\varphi)$  then
9         return UNSAT;
10       $\varphi := \text{UnitPropagation}(\varphi)$ ;
11 return SAT;
```

---

## 2.3 Conflict-Driven Clause Learning

The original DPLL algorithms described in the previous section has been extended during recent years. In particular, almost all modern SAT solvers are based on a new algorithm, called Conflict-Driven Clause Learning (CDCL) [40]. CDCL is reported in pseudo-code in Algorithm 2. CDCL takes a formula  $\varphi$  as input and starts from an assignment in which all literals are undefined. An undefined literal  $\ell$  is selected according to a branching heuristic criterion. A unit clause  $\{\ell\}$  is added to the formula  $\varphi$ , and  $\text{UnitPropagation}(\varphi)$  is performed as in the standard DPLL. After that, if there are violated clauses in  $\varphi$ , function  $\text{LearnClause}(\varphi)$  is called. This function analyzes violated clauses in order to produce a clause modeling the violation (or conflict) which is added to  $\varphi$  (learning). This learned clause is computed in such a way that the variables assignments leading to the violation of the clauses are prohibited. Then, the algorithm calls function  $\text{RestoreConsistency}(\varphi)$  which undoes previous computation until no clauses are violated. If  $\varphi$  cannot be consistent undoing previous computation, function  $\text{RestoreConsistency}(\varphi)$  returns false, and the algorithm terminates returning UNSAT. Otherwise, since the learned clause may be *unit*, function  $\text{UnitPropagation}(\varphi)$  is invoked, and the learning process is repeated until there are no violated clauses (line 6).

One of the most important features of algorithm CDCL is learning. However, the number of learned clauses can grow exponentially, thus an heuristic for controlling the number of learned clauses is also usually implemented.

Moreover, another important technique usually employed in CDCL is called *restarts*. Intuitively, some heuristic criteria are used to stop the computation in order to restart the search by taking into account the learned clauses since the first nondeterministic choices.

In the following, we first detail the learning procedure by describing the most used learning scheme. Then, two effective branching heuristics are described in Section 2.3.2. Finally, in Sections 2.3.3 and 2.3.4, some heuristics for deletion of learned clauses and restarts are described, and details are given for those employed by two modern SAT solvers, i.e. MINISAT [41] and GLUCOSE [42].

### 2.3.1 Learning

Learned clauses forbid literal assignments that are not valid because they lead to a conflict. To perform learning the *implication graph* is built during unit propagation and a learning scheme is applied [24].

**Implication Graph.** The implication graph is a directed graph where each node represents a literal that is true with respect to the current assignment. An arc from a node  $N_1$  to a node  $N_2$  represents that the assignment of  $N_1$  participated to the inference of  $N_2$ , i.e. if  $N_2$  is inferred by applying unit propagation on a clause containing  $\neg N_1$ . Hence, branching literals have no incoming edges. When a literal  $\ell$  is selected by the branching heuristic, a node is added to the implication graph. Each literal  $\ell$  appearing in the implication graph is associated with a nonnegative integer called *decision level*, defined as the number of branching literals in the implication graph after literal  $\ell$  is added. A *conflict* occurs when the implication graph contains a node for a literal  $\ell$  and a node for the complementary literal  $\neg\ell$  is added. In this case, literals  $\ell$  and  $\neg\ell$  are called *conflicting literals*.

**First UIP Learning Scheme.** A node  $N_1$  with decision level  $d$  is said to *dominate* a node  $N_2$  with the same decision level  $d$  if and only if  $N_1$  is contained in all paths connecting the branching literal with decision level  $d$  to  $N_2$ . A *Unique Implication Point (UIP)* in conflictual implication graph is a node at the current decision level that dominates both vertices corresponding to the conflicting literals. It is important to note that a UIP represents a unique reason of the current decision level that implies the conflict, and thus, there may be more than one UIP for a certain conflict (e.g. the decision literal is always a UIP). The UIPs are usually ordered starting from the conflict. In the *first UIP* [26] learning scheme, the learned clause is composed by the



first UIP in the above-mentioned order and by the literals from a smaller decision level that imply the conflict. The first UIP learning scheme usually allows to learn the smallest clauses implying the conflict [26]. The learning algorithm is the following:

1. Let  $u$  be the first UIP, and  $d$  be its decision level.
2. Let  $L$  be the set of literals with decision level  $d$  occurring in a path from  $u$  to the conflicting literals.
3. Add a literal  $\ell$  to the learned clause if there is an arc  $(\neg\ell, \ell')$  in the implication graph,  $\ell' \in L$  and the decision level of  $\ell$  is lower than  $d$ .
4. Add  $\neg u$  to the learned clause.

**Example 5** (Learning). Consider the following CNF formula  $\varphi$ :

$$\begin{array}{ll} \{\neg v1, \neg v9, \neg v2\} & \{\neg v1, \neg v10, \neg v14, v3\} \\ \{v2, \neg v3, v4\} & \{\neg v4, v11, \neg v5\} \\ \{v6, \neg v4\} & \{\neg v4, \neg v12, v7\} \\ \{v5, \neg v6, v8\} & \{\neg v6, \neg v7, \neg v13, \neg v8\} \end{array}$$

A possible Implication Graph associated to  $\varphi$  is shown in Figure 2.1. Decision levels are reported in parentheses. Literal  $v1$  is the branching literal with decision level 7 and leads to the derivation of the literals with decision level 7. The remaining literals  $v9$  to  $v14$  have been assigned as true in a previous decision level. UIPs are  $v1$  and  $v4$ , because they are part of all paths from  $v1$  to either  $v8$  or  $\neg v8$ . The first UIP is  $v4$ , since it is the UIP closest to the conflict. The learned clause is  $\{\neg v4, v11, \neg v12, \neg v13\}$ .  $\triangleleft$

### 2.3.2 Branching Heuristics

An important role in CDCL, as well as in DPLL, is played by the branching heuristics. In the following we focus on two heuristics that are employed with success in CDCL solvers, namely the Variable State Independent Decaying Sum heuristic and the MINISAT heuristic.

**Variable State Independent Decaying Sum.** The Variable State Independent Decaying Sum (VSIDS) [26] heuristic maintains a counter  $cl(\ell)$  for each literal  $\ell$ . Counters are initialized to 0 and a counter  $cl(\ell)$  is increased when a literal  $\ell$  appears in a learned clause. Periodically, all counters are divided by a constant. At each decision, the next branching literal is the

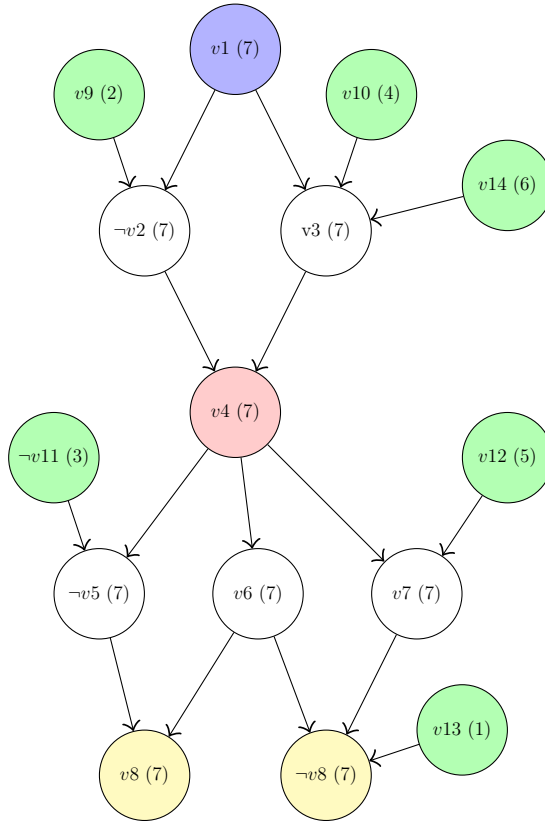


Figure 2.1: Implication Graph (First UIP:  $v_4$ ; Level in parentheses)

undefined literal  $\ell$  with the highest value of  $cl(\ell)$ . If two or more literals have the same highest value of the heuristic counter, the next branching literal is chosen randomly among them.

**MINISAT.** The branching heuristic of the SAT solver MINISAT [41] is an enhancement of the original VSIDS. MINISAT maintains an activity counter for each variable. The activity counter is initialized to 0. When a variable  $v$  is involved in the learning process, i.e. it is responsible of some conflictual assignments, its activity counter of  $v$  is increased by an *increment* value. The *increment* is initialized to the value 1.0 and it is multiplied by  $1.0/0.95$  after each conflict. Thus, counters of the variables involved in future conflict will be increased by a higher value. This has the effect to give more importance to variables involved in recent conflicts. At each decision, the variable with the highest activity is chosen as false. Ties are broken randomly.

### 2.3.3 Learned Clauses Deletion

The number of learned clauses can grow exponentially, and this may cause a performance degradation of propagation. A heuristic is usually employed for deleting some of them. We present the heuristic employed by MINISAT and GLUCOSE, since their effectiveness has been proved in practical applications.

**MINISAT.** The SAT solver MINISAT [41] implements a heuristic that removes learned clauses not involved often in recent conflicts. Each learned clause is associated with activity counters measuring the number of times a clause was involved in the derivation of a conflict and are considered *locked* if binary or when participated to the inference of some literal. Clauses are then deleted performing the following algorithm:

1. Sort all clauses by increasing activity;
2. Remove the first half of the learned clauses that are not locked;
3. Remove all learned clauses that are not locked and have an activity counter smaller than a threshold.

**Glucose.** The deletion heuristic of GLUCOSE [42] is based on the concept of *Literals Blocks Distance (lbd)*. Given a learned clause  $c$ , the *lbd* of  $c$  is the number of different decision levels associated with the literals in  $c$ . The *lbd* is computed when  $c$  is learned and it is updated in specific instants of the search. In more detail, the *lbd* of a clause  $c$  is updated when  $c$  is involved in the derivation of a new conflict, i.e. when a literal  $\ell \in c$ , inferred by unit propagation through  $c$ , is navigated in the implication graph during the computation of a new learned clause.

Clauses are deleted performing the following algorithm:

1. Sort all clauses by increasing *lbd*.
2. Remove the first half of the learned clauses that are not locked.

Thus, clauses with an higher value of *lbd* are more likely to be deleted in the GLUCOSE heuristic.

### 2.3.4 Restarts

In addition to basic learning, many SAT solvers exploit another technique called *restarts*, that consists of a halt in the solution process, and a restart of the search. Essentially, the solution process is interrupted, and the search of

---

**Function** ChooseLiteral(Set  $assum_{\wedge}$ , Set  $assum_{\vee}$ )

---

```
1 if  $\exists$  an undefined literal  $\ell \in assum_{\wedge}$  then  
2   return  $\ell$ ;  
3 if  $\exists$  a true literal  $\ell \in assum_{\vee}$  then  
4   return ChooseLiteral();  
5 if  $\exists$  an undefined literal  $\ell \in assum_{\vee}$  then  
6   return  $\ell$ ;  
7 return ChooseLiteral();
```

---

a model is restarted from scratch. We present the restart strategies employed by MINISAT and GLUCOSE.

**MINISAT.** The SAT solver MINISAT performs restarts depending on the number of encountered conflicts and according to a heuristic sequence. In particular, MINISAT uses the following sequence of conflicts (32, 32, 64, 32, 32, 64, 128, 32, 32, 64, 128, 256, ...) which is based on the Luby series introduced in [43].

**Glucose.** The SAT solver GLUCOSE implements a new strategy for restarts considering the *lbd* scores of learned clauses [44]. In particular, the idea is to restart when learned clauses of recent conflicts are increasing the *lbd* scores. In more detail, this heuristic has two parameters  $k$  and  $x$ , which in GLUCOSE are set to 0.7 and 100, respectively. The latest  $x$  conflicts are considered, and the average of their *lbd* scores is multiplied by  $k$ . A restart occurs when this value is greater than the average *lbd* of all clauses.

## 2.4 Incremental SAT

SAT solvers take a CNF formula  $\varphi$  as input, then look for a variable assignment that satisfies  $\varphi$ , and then return *SAT* if such variable assignment exists, and *UNSAT* otherwise. Albeit a single call to a SAT solver is sufficient for many applications, many problems are not efficiently solvable in a unique call. When multiple calls are needed the following naive algorithm can be applied:

1. Create a CNF formula  $\varphi$  modeling the problem.
2. Invoke a SAT solver on the CNF formula  $\varphi$ .

3. Analyze the results.
4. If a solution to the problem is found then stop.
5. Otherwise create a new CNF formula and go to step 2.

However, this algorithm has shown to be inadequate in many practical cases. The first drawback is that at each iteration the formula is rebuilt. Moreover, the learned clauses cannot be exploited in different iterations. For this reason, modern SAT solvers add an incremental interface for allowing multiple calls. The key idea is to modify the formula  $\varphi$  in order to be shared by all calls, and then solve the problem by using a set of *literal assumptions* [45], which model a particular instance of the problem. In more detail, an incremental variant of Algorithm 2 can be implemented by introducing two sets of assumptions,  $assum_{\wedge}$  and  $assum_{\vee}$ . The incremental algorithm checks whether a formula  $\varphi$  is satisfiable under the literal assumptions, that is, if there exists a model of  $\varphi$  such that  $assum_{\wedge}$  and  $assum_{\vee}$  are satisfied. Given an interpretation  $I$ :

- $assum_{\wedge}$  is satisfied if  $assum_{\wedge} \subseteq I$ , i.e. if each literal  $\ell \in assum_{\wedge}$  is true w.r.t.  $I$ ;
- $assum_{\vee}$  is satisfied if  $assum_{\vee} \cap I \neq \emptyset$ , i.e. if at least one literal  $\ell \in assum_{\vee}$  is true w.r.t.  $I$ .

The satisfiability test under assumptions can be implemented by modifying the function *ChooseLiteral* of Algorithm 2 in the following way:

1. If  $assum_{\wedge}$  is satisfied go to step 2. Otherwise, pick the first undefined literal among the literals in  $assum_{\wedge}$ .
2. If  $assum_{\vee}$  is satisfied go to step 3. Otherwise, pick the first undefined literal among the literals in  $assum_{\vee}$ .
3. Pick a branching literal as in the non-incremental algorithm.

The search then runs as in the non-incremental version. If during the search the assumptions  $assum_{\wedge}$  or the assumptions  $assum_{\vee}$  are violated then the algorithm undoes the computation until the assumptions are not violated anymore.

An important advantage of this incremental strategy is its capability of reusing learned clauses from previous invocations. In a CDCL solver this has a great impact on the performance, since all of the previous computation is used for pruning the search space in future calls. The incremental interface provided by many modern SAT solvers has been used to solve several problems, ranging from MaxSAT to QBF. In the following we show an algorithm for solving the MaxSAT problem based on iterative calls to a SAT solver.

---

**Algorithm 3:** Fu&Malik algorithm

---

**Input** : A WCNF formula  $\Phi = \varphi_H \cup \varphi_S$

**Output:** An optimum model for  $\Phi$

```
1 begin
2   while true do
3      $(res, \varphi_C, M) := \text{SATSolver}(\Phi);$ 
4     if  $res = SAT$  then return  $M;$ 
5      $S := \emptyset;$ 
6     foreach clause  $c \in \varphi_C \cap \varphi_S$  do
7       let  $v$  be a fresh variable;
8        $c := c \cup \{v\};$ 
9        $S := S \cup \{v\};$ 
10     $\varphi_H := \varphi_H \cup \text{CNF}(\#AtMostOne(S));$ 
```

---

## 2.5 Maximum Satisfiability

The Maximum Satisfiability (MaxSAT) problem is the optimization variant of SAT where clauses are replaced by weighted clauses. A weighted clause is a pair  $(c, w)$ , where  $c$  is a clause and  $w$ , called *weight*, is either a positive integer or  $\top$ . A weighted clause  $(c, w)$  is said to be a *hard* clause if  $w = \top$ , otherwise the clause is *soft* and  $w$  represents the cost of violating the clause. A formula in weighted conjunctive normal form (WCNF)  $\Phi = \varphi_H \cup \varphi_S$  is a set of weighted clauses, where  $\varphi_H$  is the set of hard clauses, and  $\varphi_S$  is the set of soft clauses. A model for  $\Phi$  is a variable assignment  $I$  that satisfies all hard clauses. The cost of a model  $I$  is the sum of weights of the soft clauses that are violated w.r.t.  $I$ . Given a WCNF  $\Phi$ , the MaxSAT problem is to find a model that minimizes the cost of violated soft clauses in  $\Phi$ . Several variants of the MaxSAT problem can be obtained by applying constraints to the sets of hard and soft clauses. In particular, given a weighted formula  $\Phi = \varphi_H \cup \varphi_S$ :

- If  $\varphi_H = \emptyset$  and  $\forall (c, w) \in \varphi_S, w = 1$ , the problem is referred to as (unweighted) MaxSAT.
- If  $\varphi_H = \emptyset$  and  $\exists (c, w) \in \varphi_S : w > 1$ , the problem is referred to as weighted MaxSAT.
- If  $\varphi_H \neq \emptyset$  and  $\forall (c, w) \in \varphi_S, w = 1$ , the problem is referred to as (unweighted) partial MaxSAT.

- If  $\varphi_H \neq \emptyset$  and  $\exists (c, w) \in \varphi_S : w > 1$ , the problem is referred to as weighted partial MaxSAT.

During recent years, several algorithms have been proposed for solving the above variants of the MaxSAT problem. Modern and effective MaxSAT solvers implement algorithms that call iteratively a SAT solver and use the incremental interface described in Section 2.4. Many of those algorithms are based on the concept of *unsatisfiable core* (or simply *core*) of an unsatisfiable CNF formula  $\varphi$ , that is, a subset of  $\varphi$  which is still unsatisfiable. Core-based algorithms have been successfully applied for solving industrial instances as witnessed by the results of the MaxSAT evaluations (see <http://maxsat.ia.udl.cat/>) and they represent the state-of-the-art of MaxSAT solving. In the following we present the first core-based algorithm for partial MaxSAT problems, which has been proposed in [46] and it is called Fu&Malik algorithm. The Fu&Malik algorithm is important for historical reasons since it introduced the concept of unsatisfiable cores for solving the MaxSAT problem. The pseudo code of Fu&Malik algorithm is sketched in Algorithm 3. The algorithm takes as input a WCNF formula  $\Phi = \varphi_H \cup \varphi_S$  and returns as output an optimum model for  $\Phi$ . The algorithm uses a modified SAT solver that takes as input a formula  $\Phi$  and returns as output a triple  $(res, \varphi_C, M)$ , where  $res$  is a string,  $\varphi_C$  a set of clauses and  $M$  an interpretation. The SAT solver searches for a model of  $\Phi$ . If one is found, say  $M$ , the function returns  $(SAT, \emptyset, M)$ . Otherwise, the function returns  $(UNSAT, \varphi_C, \emptyset)$ , where  $\varphi_C$  is an unsatisfiable core of  $\varphi_C$ . In the following, without loss of generality, we assume that  $\varphi_H$  is satisfiable. The Fu&Malik algorithm invokes the internal SAT solver on the formula  $\Phi$ . If a model is found, then the algorithm terminates returning it (line 4). Otherwise, an unsatisfiable core  $\varphi_C$  is found. For each soft clause  $c \in \varphi_C$ , a fresh variable  $v$  is added to the clause  $c$ . The variable  $v$  is called *relaxation variable*, and, intuitively, it can be used for satisfying a soft clause without leading to a conflict. Moreover, the algorithm adds to the formula  $\Phi$  an additional set of clauses enforcing that at most one of the relaxation variables is assigned to true (line 10). This set of clauses assures to relax selectively at most one soft clause for each core. The algorithm then iterates until an optimum model is found.

# Chapter 3

## Answer Set Programming

Answer Set Programming (ASP) [1, 2, 3, 4, 5, 6, 7] is a declarative programming approach that provides a simple formalism for knowledge representation. ASP is based on the stable model semantics of logic programs [1] and allows for expressing all problems in the second level of the polynomial hierarchy [2]. In this chapter, we first introduce the syntax and the semantics of ASP as background for motivating its applications and to show how it can be used as a powerful knowledge representation and reasoning tool. An architecture of a general system for evaluating ASP programs is then presented.

### 3.1 Syntax

By convention, strings starting with uppercase letters refer to first-order variables, while strings starting with lower case letters refer to constants. A *term* is either a variable or a constant. Predicates are strings starting with lowercase letters. An arity (non-negative integer) is associated with each predicate. A *standard atom* is an expression  $p(t_1, \dots, t_n)$ , where  $p$  is a predicate of arity  $n$  and  $t_1, \dots, t_n$  are terms. A standard atom  $p(t_1, \dots, t_n)$  is ground if  $t_1, \dots, t_n$  are constants.

A *set term* is either a symbolic set or a ground set. A *symbolic set* is a pair  $\{Terms : Conj\}$ , where *Terms* is a list of terms (variables or constants) and *Conj* is a conjunction of standard atoms, that is, *Conj* is of the form  $a_1, \dots, a_n$  and each  $a_i$  ( $1 \leq i \leq n$ ) is a standard atom. A *ground set* is a set of pairs of the form  $\langle \bar{t} : conj \rangle$ , where  $\bar{t}$  is a list of constants and *conj* is a conjunction of ground atoms. An *aggregate function* is of the form  $f(S)$ , where  $S$  is a ground set and  $f \in \{\#count, \#sum\}$  is an aggregate function symbol. An *aggregate atom* is of the form  $f(S) \prec T$ , where  $f(S)$  is an



aggregate function,  $\prec \in \{=, <, \leq, >, \geq\}$  is a predefined comparison operator, and  $T$  is a constant referred to as *guard*.

An *atom* is either a standard atom, or an aggregate atom. A *literal* is either an atom  $a$ , or its default negation *not*  $a$ . Given a literal  $\ell$  we will use  $\sim\ell$  for denoting its complement, that is *not*  $a$  if  $\ell = a$  and  $a$  if  $\ell = \text{not } a$ , where  $a$  is an atom. This notation extends to sets of literals, i.e.,  $\sim L := \{\sim\ell \mid \ell \in L\}$  for a set of literals  $L$ .

**Definition 1.** A (disjunctive) rule is of the following form:

$$a_1 \vee \cdots \vee a_m \leftarrow \ell_1, \dots, \ell_n \quad (3.1)$$

where  $a_1, \dots, a_m$  are standard atoms and  $\ell_1, \dots, \ell_n$  are literals,  $m \geq 0$  and  $n \geq 0$ .

For a rule  $r$  of the form (3.1), disjunction  $a_1 \vee \cdots \vee a_m$  is called the *head* of  $r$  and conjunction  $\ell_1, \dots, \ell_n$  is named the *body* of  $r$ . The set of head atoms is denoted by  $H(r)$ . The set of body literals is denoted by  $B(r)$ , while the set of positive and negated literals in  $B(r)$  are denoted by  $B^+(r)$  and by  $B^-(r)$ , respectively. Moreover  $C(r) := H(r) \cup \sim B(r)$  is the *clause representation* of  $r$ . A rule  $r$  is *positive* if  $B^-(r) = \emptyset$ . A *normal* rule is a rule of the form (3.1) such that  $m \leq 1$ . An *integrity constraint*, or simply *constraint*, is a rule of the form (3.1) such that  $m = 0$ . A *weak constraint* is a constraint which is associated with a positive integer by the partial function *weight*. For a compact representation, the weight will be sometimes indicated near the implication arrow, e.g.,  $\leftarrow_3 a, \text{not } b$  is a constraint of weight 3. A *fact* is a rule of the form (3.1) such that  $m = 1$  and  $n = 0$ .

A program  $\Pi_R$  is a set of rules. The set of constraints in  $\Pi_R$  is denoted  $\text{constraints}(\Pi_R)$ , while the remaining rules are denoted by  $\text{rules}(\Pi_R)$ . A program with weak constraints  $\Pi$  is a pair  $(\Pi_R, \Pi_W)$ , where  $\Pi_R$  is a program and  $\Pi_W$  is a subset of  $\text{constraints}(\Pi_R)$ .  $\Pi_W$  is the set of *weak constraints*, while  $\text{constraints}(\Pi_R) \setminus \Pi_W$  is the set of *hard constraints*. The set of atoms appearing in  $\Pi_R$  is denoted by  $\text{atoms}(\Pi_R)$ . A program  $\Pi$  is said to be *disjunctive* if  $\Pi$  contains at least one disjunctive rule. Otherwise,  $\Pi$  is said to be *normal*. A term, an atom, a literal, a rule, or a program is called *ground* if no variables appear in it. A local variable of a rule  $r$  is a variable appearing only in sets terms of  $r$ ; a variable of  $r$  is global if it is not local. In ASP, rules in programs are required to be safe. A rule  $r$  is *safe* if both the following conditions hold: (i) for each global variable  $X$  of  $r$  there is a positive standard literal  $\ell \in B^+(r)$  such that  $X$  appears in  $\ell$ ; (ii) each local variable of  $r$  appearing in a symbolic set  $\{\text{Terms} : \text{Conj}\}$  also appears in  $\text{Conj}$ . A program is safe if each of its rules is safe. In the following, we will only consider safe programs.

**Example 6** (Disjunctive logic program). Consider the following disjunctive logic program:

$$\begin{aligned} r_1 : & \quad a(X) \vee b(X) \leftarrow c(X), \text{ not } d(X) \\ r_2 : & \quad \quad \quad \quad \quad \leftarrow c(X), f(X) \\ r_3 : & \quad \quad \quad \quad \quad \quad c(1) \leftarrow \end{aligned}$$

- Rule  $r_1$  is a disjunctive rule with  $H(r_1) = \{a(X), b(X)\}$ ,  $B^+(r_1) = \{c(X)\}$ , and  $B^-(r_1) = \{d(X)\}$ .
- Rule  $r_2$  is a constraint with  $B^+(r_2) = \{c(X), f(X)\}$  and  $B^-(r_2) = \emptyset$ .
- Rule  $r_3$  is a fact. Note that every fact must be ground in order to be safe.  $\triangleleft$

## 3.2 Semantics

The answer set semantics is defined on ground programs and it is given by its stable models. Given a program  $\Pi$ , the *Herbrand universe*  $U_\Pi$  is the set of all constants appearing in  $\Pi$ . If there are no constants in  $\Pi$ , then  $U_\Pi$  contains an arbitrary constant  $c$ . Given a program  $\Pi$ , the *Herbrand base*  $B_\Pi$  is the set of all possible ground atoms which can be constructed from the predicate symbols appearing in  $\Pi$  with the constants of  $U_\Pi$ .

**Example 7** (Herbrand universe and Herbrand base). Consider the following program  $\Pi_0$ :

$$\begin{aligned} r_1 : & \quad a(X) \vee b(X) \leftarrow c(X) \\ r_2 : & \quad \quad \quad \quad \quad \leftarrow d(X), c(X) \\ r_3 : & \quad \quad \quad \quad \quad \quad c(1) \leftarrow \\ r_4 : & \quad \quad \quad \quad \quad \quad c(2) \leftarrow \end{aligned}$$

Then,  $U_{\Pi_0} = \{1, 2\}$  and  $B_{\Pi_0} = \{a(1), a(2), b(1), b(2), c(1), c(2), d(1), d(2)\}$ .  $\triangleleft$

For any rule  $r$ ,  $Ground(r)$  denotes the set of rules obtained by replacing each variable in  $r$  by constants in  $U_\Pi$  in all possible ways. For any program  $\Pi$ , its ground instantiation is the set  $Ground(\Pi) = \bigcup_{r \in \Pi} Ground(r)$ <sup>1</sup>.

<sup>1</sup>We refer the reader to [8, 17, 47] for an accurate description of the grounding.

**Example 8** (Ground instantiation). Consider the program  $\Pi_0$  in Example 7. Its ground instantiation is the following:

$$\begin{aligned}
g_1 : & a(1) \vee b(1) \leftarrow c(1) \\
g_2 : & a(2) \vee b(2) \leftarrow c(2) \\
g_3 : & d(1) \leftarrow b(1), c(1) \\
g_4 : & d(2) \leftarrow b(2), c(2) \\
g_5 : & c(1) \leftarrow \\
g_6 : & c(2) \leftarrow
\end{aligned}$$

Note that the atoms  $c(1)$  and  $c(2)$  are already ground in  $\Pi_0$ , while the rules  $g_1$  and  $g_2$  are obtained from  $r_1$  and the rules  $g_3$  and  $g_4$  are obtained from  $r_2$ .  $\triangleleft$

A set  $L$  of ground literals is said to be *consistent* if, for every literal  $\ell \in L$ , its complementary literal  $\sim\ell$  does not belong to  $L$ . An *interpretation*  $I$  for  $\Pi$  is a consistent set of ground literals over atoms in  $B_\Pi$ . A ground literal  $\ell$  is interpreted as follows:

- $\ell$  is *true* w.r.t.  $I$  if  $\ell \in I$ .
- $\ell$  is *false* w.r.t.  $I$  if  $\sim\ell \in I$ .
- $\ell$  is *undefined* w.r.t.  $I$  if it is neither true nor false w.r.t.  $I$ .

A ground conjunction of atoms  $conj$  is true w.r.t.  $I$  if all atoms appearing in  $conj$  are true w.r.t.  $I$ . Conversely,  $conj$  is false w.r.t.  $I$  if there is an atom in  $conj$  that is false w.r.t.  $I$ . Let  $I(S)$  denote the multiset  $[t_1 \mid \langle t_1, \dots, t_n \rangle : conj \in S \wedge conj \text{ is true w.r.t. } I]$ . The valuation  $I(f(S))$  of an aggregate function  $f(S)$  w.r.t.  $I$  is the result of the application of  $f$  on  $I(S)$  [8].

Let  $r$  be a rule in  $Ground(\Pi)$ .

- The head of  $r$  is true w.r.t.  $I$  if and only if there is an atom  $a \in H(r)$  such that  $a$  is true w.r.t.  $I$ .
- The body of  $r$  is true w.r.t.  $I$  if and only if each literal  $\ell \in B(r)$  is true w.r.t.  $I$ .
- The head of  $r$  is false w.r.t.  $I$  if and only if each atom  $a \in H(r)$  is false w.r.t.  $I$ .
- The body of  $r$  is false w.r.t.  $I$  if and only if there is a literal  $\ell \in B(r)$  such that  $\ell$  is false w.r.t.  $I$ .

A rule  $r$  is *satisfied* w.r.t.  $I$  if and only if  $H(r)$  is true w.r.t.  $I$  or  $B(r)$  is false w.r.t.  $I$ . For a rule  $r$ ,  $I \models r$  if  $r$  is satisfied w.r.t.  $I$ .

An interpretation  $I$  is *total* if and only if for each literal  $\ell \in B_{\Pi}$ ,  $\ell \in I$  or  $\sim\ell \in I$ , otherwise  $I$  is *partial*. A total interpretation  $M$  is a *model* for  $\Pi$  if and only if for each rule  $r \in \Pi$ ,  $M \models r$ . Stated differently, a total interpretation  $M$  is a model for  $\Pi$  if, for each  $r \in \Pi$ ,  $r$  is satisfied w.r.t.  $M$ . A model  $M$  is a *stable model* (or *answer set*) for a *positive* program  $\Pi$  if it is a minimal set (w.r.t. set inclusion) among the models for  $\Pi$ . The definition of stable models for general programs is based on the *FLP-reduct* [8].

**Definition 2.** The *FLP-reduct* [8] of a ground program  $\Pi$  w.r.t. an interpretation  $I$  is the ground program  $\Pi^I$  obtained from  $\Pi$  by deleting each rule  $r \in \Pi$  whose body is not satisfied w.r.t.  $I$ .

**Definition 3.** A *stable model* (or *answer set*) [1] of a program  $\Pi$  is a model  $I$  of  $\Pi$  such that  $I$  is a stable model of  $\Pi^I$ .

Let  $SM(\Pi)$  denote the set of stable models of  $\Pi$ . If  $SM(\Pi) \neq \emptyset$  then  $\Pi$  is *coherent*, otherwise it is *incoherent*.

**Definition 4** (Optimum Stable Model). For a program with weak constraints  $\Pi = (\Pi_R, \Pi_W)$ , each interpretation  $I$  is associated with a cost:

$$cost(\Pi_W, I) := \sum_{r \in \Pi_W : I \not\models r} weight(r).$$

A stable model  $I$  of  $\Pi_R \setminus \Pi_W$  is optimum for  $\Pi$  if there is no  $J \in SM(\Pi_R \setminus \Pi_W)$  such that  $cost(\Pi_W, J) < cost(\Pi_W, I)$ .

An atom  $a \in atoms(\Pi)$  is a *cautious consequence* of a program  $\Pi$  if  $a$  belongs to all stable models of  $\Pi$ . More formally,  $a \in atoms(\Pi)$  is a cautious consequence of a program  $\Pi$  there is no  $M \in SM(\Pi)$  such that  $a \notin M$ . The set of cautious consequences of  $\Pi$  is denoted  $CC(\Pi)$ .

### 3.3 Properties of ASP programs

In this section we recall some properties of answer sets.

**Definition 5** (Supportedness Property). Given an interpretation  $I$ , a positive literal  $\ell$  is *supported* w.r.t.  $I$  if and only if there exists a rule  $r$  such that for each literal  $\ell_b$  is true with respect to  $I$  and for each atom  $\ell_h \in H(r) \setminus \{\ell\}$ ,  $\ell_h \in I$ .

A model  $M$  is said to be *supported* if for each positive literal  $a \in M$ ,  $a$  is supported. Answer sets are supported models, while the reverse is not necessarily true.

**Definition 6** (Possibly Supporting Rule). Given an interpretation  $I$ , a rule  $r$  is a *possibly supporting rule* for an atom  $a$  if the following conditions hold:

- $a \in H(r)$ ; and
- no literal in  $B(r)$  is false with respect to  $I$ ; and
- $(H(r) \setminus \{a\}) \cap I = \emptyset$ .

Let  $\text{supp}(a, I)$  denotes the set of possibly supporting rules for an atom  $a$  and an interpretation  $I$ .

**Definition 7** (Dependency Graph). Let  $\Pi$  be a program. The *dependency graph* of  $\Pi$ , denoted  $DG_\Pi = (N, A)$ , is a directed graph in which (i) each atom in  $\text{atoms}(\Pi)$  is a node in  $N$  and (ii) there is an arc in  $A$  directed from a node  $a$  to a node  $b$  if there exists a rule  $r$  in  $\Pi$  such that  $a \in H(r)$  and either  $b \in B^+(r)$  or  $b$  occurs in an aggregate atom. It is important to note that negative literals cause no arc in  $DG_\Pi$ . We also safely assume that any rule  $r$  such that  $H(r) \cap B^+(r) \neq \emptyset$  is removed from  $\Pi$ .

**Definition 8** (Component). A *strongly connected component* (or simply *component*)  $C$  of  $DG_\Pi$  is a maximal subset of  $N$  such that each node in  $C$  is connected by a path to all other nodes in  $C$ .

In the following, we assume that for each rule  $r$ , and for each pair of atoms  $(a, b)$  such that  $a \in H(r)$  and  $b$  appears in aggregate atom of  $r$ ,  $a$  and  $b$  are in two different components.

A component  $C$  is *recursive*, or *cyclic*, if  $C$  contains two or more atoms. A component  $C$  is *head-cycle free* (HCF for short) if each rule  $r \in \Pi$  is such that  $|H(r) \cap C| \leq 1$ . Otherwise  $C$  is said to be *non head-cycle free* (non-HCF). Given a program  $\Pi$ , the subprogram  $\text{sub}(\Pi, C)$  corresponding to a component  $C$  of the dependency graph is defined as the set of rules  $r \in \Pi$  such that  $H(r) \cap C \neq \emptyset$ . The rules of  $\Pi$  can be assigned to one or more components. More specifically, a rule  $r$  is assigned to a component  $C$  if  $r \in \text{sub}(\Pi, C)$ . Moreover, a rule  $r \in \text{sub}(\Pi, C)$  is said to be an *external rule* of  $C$  if  $B^+(r) \cap C = \emptyset$ ; otherwise,  $r$  is an *internal rule* of  $C$ . An atom  $a$  is said to be an *external atom* of  $C$  if  $a \notin C$ ; otherwise, the atom is internal. A literal  $\ell = a$  or  $\ell = \sim a$  is said to be an *external literal* if  $a$  is an external atom; otherwise, the literal is internal.

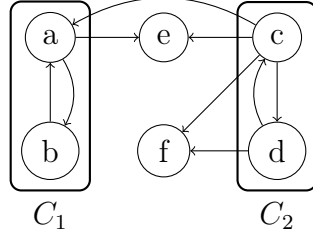


Figure 3.1: Dependency graph of the program  $\Pi$

**Example 9.** Consider the following program  $\Pi$ :

$$\begin{array}{ll}
 r_1 : & a \leftarrow e & r_5 : & c \leftarrow a, e \\
 r_2 : & a \leftarrow b & r_6 : & c \leftarrow d \\
 r_3 : & b \leftarrow a & r_7 : & d \leftarrow c \\
 r_4 : & e \vee f \leftarrow & r_8 : & c \vee d \leftarrow f
 \end{array}$$

The dependency graph  $DG_{\Pi}$  of  $\Pi$  is reported in Fig. 3.1, where we also represented the two recursive components  $C_1 = \{a, b\}$  and  $C_2 = \{c, d\}$ . The component  $C_1$  is HCF;  $r_1$  is an external rule, while rules  $r_2$  and  $r_3$  are internal. The component  $C_2$  is non-HCF;  $r_5$  and  $r_8$  are external rules, while rules  $r_6$  and  $r_7$  are internal.  $\triangleleft$

**Definition 9** (Unfounded Set). A set  $X$  of atoms is *unfounded* w.r.t an interpretation  $I$  if for each rule  $r$  such that  $H(r) \cap X \neq \emptyset$ , at least one of the following conditions is satisfied:

- i) The body of  $r$  is false w.r.t  $I$ .
- ii) The body of  $r$  is false w.r.t.  $(I \setminus X) \cup \sim X$ .
- iii)  $((H(r) \setminus X) \cap I) \neq \emptyset$ , that is, there exists a true atom w.r.t.  $I$  in the head of  $r$  that does not belong to  $X$ .

**Theorem 1** (Theorem 4.6 in [48]). Let  $\Pi$  be a program and  $I$  a supported model for  $\Pi$ ,  $I$  is an answer set iff  $I$  is *unfounded-free*, i.e. if there exists no non-empty  $X \subseteq I$  such that  $X$  is an unfounded set for  $\Pi$  w.r.t.  $I$  [48].

The unfounded-free property can be verified independently for each component  $C$  of the program, i.e. an interpretation  $I$  is unfounded-free w.r.t. a program  $\Pi$  iff  $I$  is unfounded-free w.r.t. each subprogram  $sub(\Pi, C)$ . Checking whether a set of atoms is unfounded is known to be polynomial for HCF components, while it is a co-NP complete problem for non-HCF components [2].

**Definition 10** (Rule shifting). Let  $r$  be a disjunctive rule of the form  $a_1 \vee \dots \vee a_m \leftarrow \ell_1, \dots, \ell_n$ . The shifting of  $r$  consists of replacing  $r$  by  $m$  normal rules, one for each atom in the head. The rule for the  $i$ -th head atom is as follows:

$$a_i \leftarrow \ell_1, \dots, \ell_n, \text{not } a_1, \dots, \text{not } a_{i-1}, \text{not } a_{i+1}, \dots, \text{not } a_m. \quad (3.2)$$

**Definition 11** (Clark's Completion). Given a ground program  $\Pi$ , let  $aux_r$  denotes a fresh atom, i.e., an atom not appearing elsewhere, added for a rule  $r \in \Pi$ . The Clark's completion (or completion) of  $\Pi$ , denoted  $Comp(\Pi)$ , consists of the following set of rules:

$$\begin{aligned} &\leftarrow \text{not } a, aux_{r_1} \\ &\dots \\ &\leftarrow \text{not } a, aux_{r_n} \\ &\leftarrow a, \text{not } aux_{r_1}, \dots, \text{not } aux_{r_n} \end{aligned}$$

for each atom  $a \in atoms(\Pi)$ , where  $r_1, \dots, r_n$  are the rules in  $\Pi$  whose heads contain  $a$ .

$$\leftarrow \text{not } aux_r, B(r)$$

for each rule  $r \in \Pi$ ;

$$\leftarrow aux_r, \sim b_i$$

for each rule  $r \in \Pi$  and for each  $b_i \in B(r)$ ;

**Example 10.** Consider the following program  $\Pi$ :

$$\begin{aligned} r_1 : & a \vee b \leftarrow \\ r_2 : & c \vee d \leftarrow a \end{aligned}$$

The program  $\Pi$  after shifting disjunctive rules is the following:

$$\begin{aligned} r_1 : & a \leftarrow \text{not } b & r_2 : & b \leftarrow \text{not } a \\ r_3 : & c \leftarrow a, \text{not } d & r_4 : & d \leftarrow a, \text{not } c \end{aligned}$$

Finally,  $Comp(\Pi)$  consists of the following set of rules:

$$\begin{aligned} &\leftarrow \text{not } a, aux_{r_1} & &\leftarrow \text{not } b, aux_{r_2} \\ &\leftarrow \text{not } c, aux_{r_3} & &\leftarrow \text{not } d, aux_{r_4} \\ &\leftarrow a, \text{not } aux_{r_1} & &\leftarrow b, \text{not } aux_{r_2} \\ &\leftarrow c, \text{not } aux_{r_3} & &\leftarrow d, \text{not } aux_{r_4} \\ &\leftarrow \text{not } aux_{r_1}, \text{not } b & &\leftarrow \text{not } aux_{r_2}, \text{not } a \\ &\leftarrow \text{not } aux_{r_3}, a, \text{not } d & &\leftarrow \text{not } aux_{r_4}, a, \text{not } c \\ &\leftarrow aux_{r_1}, b & &\leftarrow aux_{r_2}, a \\ &\leftarrow aux_{r_3}, \text{not } a & &\leftarrow aux_{r_4}, \text{not } a \\ &\leftarrow aux_{r_3}, d & &\leftarrow aux_{r_4}, c. \triangleleft \end{aligned}$$

**Proposition 1.** If  $M$  is a model of  $Comp(\Pi)$  then  $M|_{\Pi}$  is a supported model of  $\Pi$ , where  $M|_{\Pi}$  is the restriction of  $M$  to the symbols of  $\Pi$ , that is,  $M|_{\Pi} := M \cap atoms(\Pi)$ .

### 3.4 Knowledge Representation And Reasoning

ASP can be used to encode problems in a simple and declarative way. Moreover, ASP is very expressive by allowing to represent problems that belong to the complexity class  $\Sigma_2^P$  (i.e.  $NP^{NP}$ ). The simplicity of encodings and the high expressivity of the language are the keys of the success of ASP. In fact, ASP has been used in several domains, such as artificial intelligence, deductive databases, bioinformatics, and also on industrial problems.

In this section, we show the usage of ASP as a tool for knowledge representation and reasoning by examples. In particular, we present three well-known problems: *3-Colorability*, *Consistent Query Answering* and *Maximum Clique*. The problem 3-Colorability shows how hard problems can be easily encoded in ASP; Consistent Query Answering shows an application of cautious reasoning in ASP; and the problem Maximum Clique shows how ASP can deal with optimization problems by means of weak constraints.

NP-problems are usually encoded in ASP by using the ‘‘Guess & Check’’ methodology originally introduced in [49] and refined in [17]. The idea behind this method can be summarized as follows: a set of facts (input database) is used to specify an instance of the problem, while a set of rules (guessing part), is used to define the search space; solutions are then identified in the search space by another set of rules (checking part), which impose some admissibility constraints. In other words, the guessing part with the input database defines the set of all possible solutions, which are then filtered by the checking part to guarantee that the answer sets of the resulting program represent precisely the admissible solutions for the input instance.

**3-Colorability.** Given a finite undirected graph  $G = (V, E)$  and a set of three colors  $C$ , does there exist a color assignment for each vertex such that there are no adjacent vertices sharing the color assignment?

This is a typical NP-complete problem in graph theory. Suppose that the graph  $G$  is specified by using facts over predicates *vertex* and *edge*. Then, the following program solves the *Graph Coloring* problem:

$$\begin{array}{ll} r1: & col(X, "blue") \vee col(X, "red") \vee col(X, "green") \quad \leftarrow vertex(X) \\ r2: & \quad \quad \quad \leftarrow edge(X, Y), col(X, C), col(Y, C) \end{array}$$

This NP-complete problem is encoded in a simple way by using the ‘‘Guess & Check’’ methodology. In the example, the guessing part is composed by



the rule  $r_1$  while the rule  $r_2$  composes the checking part. In fact,  $r_1$  guesses a color assignment for a vertex  $v$  and  $r_2$  checks whether the color assignment is valid assuring that two adjacent vertices have different color assignments.

**Consistent Query Answering.** Consistent Query Answering (CQA) is a well-known application of ASP [28, 50]. Consider an inconsistent database  $D$  where in relation  $R = \{\langle 1, 1, 1 \rangle, \langle 1, 2, 1 \rangle, \langle 2, 2, 2 \rangle, \langle 2, 2, 3 \rangle, \langle 3, 2, 2 \rangle, \langle 3, 3, 3 \rangle\}$  the second argument is required to functionally depend on the first. Given a query  $q$  over  $D$ , CQA amounts to computing answers of  $q$  that are true in all repairs of the original database. Roughly, a repair is a revision of the original database that is maximal and satisfies its integrity constraints. In the example, repairs can be modeled by the following ASP rules:

$$\begin{aligned} r_1 : \quad R_{out}(X, Y_1, Z_1) &\leftarrow R(X, Y_1, Z_1), R(X, Y_2, Z_2), Y_1 \neq Y_2, \text{not } R_{out}(X, Y_2, Z_2) \\ r_2 : \quad R_{in}(X, Y, Z) &\leftarrow R(X, Y, Z), \text{not } R_{out}(X, Y, Z) \end{aligned}$$

Rule  $r_1$  detects inconsistent pairs of tuples and guesses tuples to remove in order to restore consistency, while  $r_2$  defines the repaired relation as the set of tuples that have not been removed. The first and third arguments of  $R$  can thus be retrieved by means of the following query rule:

$$q_1 : \quad Q(X, Z) \leftarrow R_{in}(X, Y, Z)$$

The consistent answers of  $q_1$  are tuples of the form  $\langle x, z \rangle$  such that  $Q(x, z)$  belongs to all stable models. In this case the answer is  $\{\langle 1, 1 \rangle, \langle 2, 2 \rangle, \langle 2, 3 \rangle\}$ .

**Maximum Clique.** Given a finite undirected graph  $G = (V, E)$ , a *clique*  $C$  is a subset of its vertices such that for each pair of vertices in  $C$  an edge connects the two vertices. The Maximum Clique problem is to find a clique with the greatest cardinality. That is, for each other clique  $C'$  in  $G$ , the number of nodes in  $C$  should be larger than or equal to the number of nodes in  $C'$ .

Suppose that the graph  $G$  is specified by using facts over predicates *vertex* and *edge*. Then, the following program solves the *Maximum Clique* problem:

$$\begin{aligned} r_1 : \quad in(X) \vee out(X) &\leftarrow vertex(X) \\ r_2 : \quad &\leftarrow in(X), in(Y), \text{not } edge(X, Y), \text{not } edge(Y, X), X < Y \\ r_3 : \quad &\leftarrow_1 out(X) \end{aligned}$$

This program shows the encoding of optimization problems by means of weak constraints. In the example, rule  $r_1$  represents the guessing part while the rule  $r_2$  composes the checking part. In fact,  $r_1$  guesses a subset of nodes candidates to be in a clique and rule  $r_2$  checks whether the subset of nodes is a clique assuring that every vertices in the clique are connected by an edge. Finally, rule  $r_3$  minimizes the number of nodes which are not in the clique.

### 3.5 Architecture of an ASP System

The architecture of an answer set system is usually composed by three modules. The first module is the *Grounder*, which is responsible of the creation of a ground program equivalent to the input one. After the grounding process, the next module, usually called *Model Generator*, computes stable model candidates of the program. Stable model candidates are, in turn, checked by the third module, called *Answer Set Checker*, which verifies that candidates are actually stable models. These three modules are briefly described in this section.

**Grounder.** Given an input program  $\Pi$ , the Grounder efficiently generates an intelligent ground instantiation of  $\Pi$  that has the same answer sets of the theoretical instantiation, but is usually much smaller [17]. Note that the size of the instantiation is a crucial aspect for efficiency, since the answer set computation takes exponential time (in the worst case) in the size of the ground program received as input (i.e., produced by the Grounder). In order to generate a small ground program equivalent to  $\Pi$ , the Grounder generates ground instances of rules containing only atoms which can possibly be derived from  $\Pi$ , and thus (if possible) avoiding the combinatorial explosion which can be achieved by naively considering all the atoms in the Herbrand base [51]. This is obtained by taking into account some structural information of the input program concerning the dependencies among predicates, and applying sophisticated deductive database evaluation techniques. An in-depth description of a Grounder module is out of the scope of this thesis. Therefore, we refer the reader to [17, 47] for an accurate description.

**Model Generator.** The Model Generator takes as input a propositional ASP program and returns as output answer set candidates. The Model Generator usually implements techniques introduced for SAT solving, such as learning, restarts and conflict-driven heuristics. Those techniques match the working principle of a Model Generator but require quite a lot of adaptation to deal with disjunctive logic programs under the stable model semantics.

**Answer Set Checker.** The goal of the Answer Set Checker is to verify whether a model is an answer set for an input program  $\Pi$ . This task is very hard in general, because checking the stability of a model is well-known to be co-NP-complete [2] in the worst case. In case of hard problems, this check can be carried out by translating the program into a SAT formula and checking whether it is unsatisfiable.

# Chapter 4

## The ASP Solver WASP 2

This chapter describes the ASP solver for propositional programs WASP 2 [52]. WASP 2 is inspired by several techniques that were originally introduced for SAT solving, like the CDCL algorithm [40], *learning* [24], *restarts* [25] and *conflict-driven heuristics* [26]. The mentioned SAT solving methods have been adapted and combined with state-of-the-art pruning techniques adopted by modern native ASP solvers [53, 54]. In particular, the role of Boolean Constraint Propagation in SAT solvers is taken by a procedure combining the *unit propagation* inference rule with inference techniques based on ASP program properties. In fact, support inferences are implemented via Clark’s completion, and the implementation of the polynomial unfounded-free checks is based on source pointers [18]. In WASP 2 stability of answer sets is checked by means of a reduction to the unsatisfiability problem as described in [27]. In this chapter, after introducing the architecture of the new solver, we detail the techniques for addressing the tasks of Model Generation and Answer Set Checking as implemented in WASP 2. Finally, we report on an experimental analysis in which we assess the performance of our solutions compared to the state-of-the-art alternatives.

### 4.1 The Architecture of WASP 2

The architecture of WASP 2 is composed by four modules, and it is shown in Figure 4.1. The first module is the *Input Processor*, which takes as input a ground program encoded in the numeric format of gringo [47]. The *Input Processor* applies some preliminary program transformations, and creates the data structures that are used by the subsequent modules. The *Simplifications* step further modifies the input program by removing redundant clauses and variables. The simplified program is fed as input to the *Model Generator*

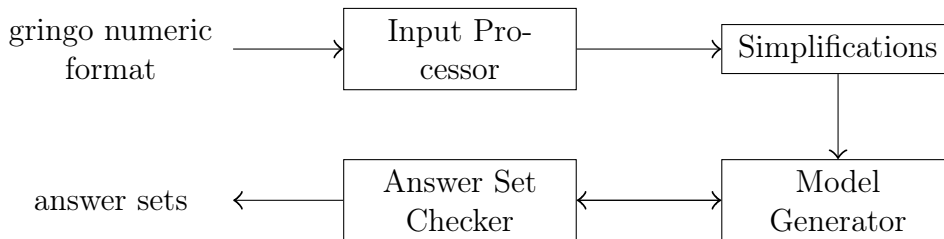


Figure 4.1: Architecture of WASP 2

module, which computes answer set candidates. The subsequent *Answer Set Checker* module implements a stability check to verify that the produced candidates are in fact answer sets.

## 4.2 Input Processor

The Input Processor takes as input a ground program encoded in the numeric format of gringo and creates the data structures that are employed by the subsequent simplification step. In particular, the Input Processor simplifies its input by removing duplicated and trivial rules, determines a splitting of the program in components (or program modules) and, then, applies two transformations, namely *program shifting* and *Clark's completion* as described in Chapter 3.

Program shifting allows to handle disjunctive programs, and Clark's completion is applied to take into account the supportedness property of answer sets. Indeed, it follows from Proposition 1 that given an input program  $\Pi$  the models of  $Comp(\Pi)$  are supported models of  $\Pi$ . This is done for simplifying the architecture of WASP 2 as it is explained in the following.

## 4.3 Simplifications of the Input Program

Program shifting and Clark's completion may cause a quadratic blow-up in the size of the input program. Thus, reducing the size of the resulting program can be crucial for the performance of ASP solvers. The simplifications employed by WASP 2 consist of polynomial algorithms for strengthening and for removing redundant rules, and also include atoms elimination by means of rewriting in the style of SATELITE [55]. Albeit the size of a program is not always related to the easiness of producing answer sets, program simplifications have a great impact in performance as it happens in SAT solving [55].

In the following we detail the simplifications applied by WASP 2.

**Subsumption.** Given a program  $\Pi$  and the program  $Comp(\Pi)$  representing the Clark's completion of  $\Pi$ , WASP 2 implements two types of rule simplifications: *subsumption* and *self-subsumption*.

**Definition 12.** A rule  $r_1 \in Comp(\Pi)$  *subsumes* a rule  $r_2 \in Comp(\Pi)$  if  $C(r_1) \subseteq C(r_2)$ , where  $C(r_1)$  and  $C(r_2)$  denote the clause representation of  $r_1$  and  $r_2$ , respectively.

A subsumed rule in  $Comp(\Pi)$  is redundant and can be safely removed.

**Definition 13.** A rule  $r_1 \in Comp(\Pi)$  *self-subsumes* a rule  $r_2 \in Comp(\Pi)$  if there is a literal  $\ell$  such that  $\ell \in C(r_1)$ ,  $\sim\ell \in C(r_2)$  and  $C(r_1) \setminus \{\ell\} \subseteq C(r_2) \setminus \{\sim\ell\}$ .

Stated differently a rule  $r_1 \in Comp(\Pi)$  self-subsumes a rule  $r_2 \in Comp(\Pi)$ , if  $r_1$  almost subsumes  $r_2$  except for one literal  $\sim\ell \in C(r_2)$  and  $\ell$  appears in  $C(r_1)$ . In this case, the rule  $r_2$  can be strengthened by removing  $\sim\ell$ .

Thus, WASP 2 removes subsumed rules of  $Comp(\Pi)$  and then applies self-subsumption to the remaining rules. Subsumption and self-subsumption are then alternated until no other simplifications can be applied.

**Literals Elimination.** WASP 2 also implements a procedure for eliminating literals through rule distribution. The procedure detects a definition of a literal  $\ell$ , that is  $\ell \iff \ell_1 \wedge \dots \wedge \ell_n$ . In particular, a literal  $\ell$  is eliminated by rule distribution if the following set of rules exists:

$$\begin{aligned} &\leftarrow \sim\ell, \ell_1, \dots, \ell_n \\ &\leftarrow \ell, \sim\ell_1 \\ &\dots \\ &\leftarrow \ell, \sim\ell_n \end{aligned}$$

Each occurrence of  $\ell$  is substituted by  $\ell_1 \wedge \dots \wedge \ell_n$ , and each occurrence of  $\sim\ell$  is substituted by  $\ell_1 \vee \dots \vee \ell_n$ . However, WASP 2 actually eliminates literals only if the number of rules after the simplification is less than the original number of rules. Moreover, a literal  $\ell = a$  or  $\ell = \text{not } a$  is not eliminated if  $a$  is in a cyclic component or  $a$  is an aggregate atom.

## 4.4 Model Generator

The Model Generator implements a CDCL-like algorithm (see Section 2.3). A pseudo-code description of the Model Generator of WASP 2 is shown in

---

**Algorithm 4:** ComputeAnswerSet

---

**Input** : A program  $\Pi$   
          An interpretation  $I$  for  $\Pi$   
**Output:** An answer set for  $\Pi$  or *Incoherent*

```
1 begin
2   while Propagate( $I$ ) do
3     if  $I$  is total then
4       return  $I$ ;
5      $\ell :=$  ChooseUndefinedLiteral();
6      $I' :=$  ComputeAnswerSet( $\Pi, I \cup \{\ell\}$ );
7     if  $I' \neq$  Incoherent then
8       return  $I'$ ;
9     if there are violated (learned) clauses then
10      return Incoherent;
11   AnalyzeConflictAndLearnClauses( $I$ );
12  return Incoherent;
```

---

Algorithm 4, which takes as input a program  $\Pi$  and an interpretation  $I$  for  $\Pi$  and searches for an answer set of  $\Pi$ . Initially, interpretation  $I$  is set to  $\emptyset$ . Function Propagate (line 2), detailed in the next section, extends  $I$  with those literals that can be deterministically inferred. This function returns false if an inconsistency (or conflict) is detected, true otherwise. When no inconsistency is detected, interpretation  $I$  is returned if total (lines 3–4). Otherwise, an undefined literal, say  $\ell$ , is chosen according to some heuristic criterion (line 5). Then computation proceeds with a recursive call to ComputeAnswerSet on  $I \cup \{\ell\}$  (line 6). In case the recursive call returns an answer set, the computation ends returning it (lines 7–8). Otherwise, the algorithm unrolls choices until consistency of  $I$  is restored (backjumping; lines 9–10), and the computation resumes by propagating the consequences of the clause learned by the conflict analysis. Conflicts detected during propagation are analyzed by procedure *AnalyzeConflictAndLearnClauses* (line 11).

The main algorithm is usually complemented with some heuristic techniques that control the number of learned constraints (which may be exponential in number), and possibly restart the computation to explore different branches of the search tree. Concerning deletion of learned constraints as well as restarts policy, WASP 2 implements both the heuristics of the SAT solvers MINISAT and GLUCOSE, as described in Sections 2.3.3 and 2.3.4. The default heuristics of WASP 2 are the ones implemented in GLUCOSE. Moreover, a

---

**Function** Propagate( $I$ )

---

```
1 while UnitPropagation( $I$ )  $\wedge$  AggregatesPropagation( $I$ ) do
2   if Unfounded-free( $I$ ) then
3     return true;
4 return false;
```

---

crucial role is played by the heuristic criteria used for selecting branching literals. WASP 2 implements the branching heuristic of the SAT solver MINISAT. Propagation and constraint learning are described in more detail in the following sections.

#### 4.4.1 Propagation

The function Propagate extends the interpretation with the literals that can be deterministically inferred. The role of propagation is similar to the unit propagation procedure in the DPLL/CDCL algorithm, but it is more complex than unit propagation because it implements a set of inference rules for taking in account the properties of ASP programs. In particular, WASP 2 implements three deterministic inference rules for pruning the search space during answer set computation. These propagation rules are named *unit*, *aggregates* and *unfounded-free*. Unit propagation infers literals appearing in unsatisfied rules containing only one undefined literal. Aggregates propagation concerns the propagation of the truth of aggregate atoms that can be deterministically inferred. Unfounded-free propagation infers the falsity of atoms appearing in an unfounded set.

Unit propagation and aggregates propagation are applied first (line 1 of function Propagate). They return false if an inconsistency arises. Otherwise, unfounded-free propagation is applied (line 2). Function unfounded-free propagation may learn an implicit clause in  $\Pi$ , in which case true is returned and unit propagation is applied on the new clause. When no new clause can be learned by unfounded-free propagation, function Propagate returns true to report that no inconsistency has been detected.

During the propagation of deterministic inferences, implications relationships among literals are stored in the implication graph. Recall that each literal  $\ell \in I$  is associated with a *decision level*, corresponding to the depth nesting level of the recursive call to ComputeAnswerSet on which  $\ell$  is added to  $I$ .

In the following we detail the propagation rules applied during the answer set computation and their effects on the implication graph.

**Unit Propagation.** Unit propagation operates on the completion of the input program  $\Pi$ , thus it is as in SAT solvers. An undefined literal  $\ell$  is inferred by unit propagation if there is a rule  $r$  that can be satisfied only by  $\ell$ , i.e.,  $r$  is such that  $\ell \in C(r)$  and for each  $\ell' \in C(r) \setminus \{\ell\}$ ,  $\sim\ell' \in I$ . In the implication graph we add node  $\ell$ , and arc  $(\sim\ell', \ell)$  for each literal  $\ell' \in C(r) \setminus \{\ell\}$ .

**Aggregates Propagation.** We now detail aggregates propagation, which consists of using aggregates for determining further deterministic consequences of an interpretation. For simplifying the presentation, in the following we adopt an alternative syntax for aggregate atoms introduced in [18]. In particular, an aggregate atom can be denoted as follows:

$$\{b_1 = w_1, \dots, b_m = w_m, \text{not } b_{m+1} = w_{m+1}, \dots, \text{not } b_n = w_n\} \geq \text{bound} \quad (4.1)$$

where  $b_i$  ( $i = 1, \dots, n$ ) are atoms, *bound* and each  $w_i$  ( $i = 1, \dots, n$ ) are positive integers. Each  $w_i$  ( $i = 1, \dots, n$ ) is the *weight* associated with the  $i$ -th literal in the aggregate. Given an interpretation  $I$ , an aggregate atom  $A$  of the form (4.1) is *true* w.r.t.  $I$  if

$$\sum_{i \in [1..m]: b_i \in I} w_i + \sum_{j \in [m+1..n]: \sim b_j \in I} w_j \geq \text{bound}$$

holds.

Given an aggregate atom of the form (4.1) an additional data structure is maintained, whose syntax is the following:  $[\text{not } b_0 = w_0, b_1 = w_1, \dots, b_m = w_m, \text{not } b_{m+1} = w_{m+1}, \dots, \text{not } b_n = w_n]$ , where  $b_0$  represents the aggregate atom and  $w_0$  is a positive integer computed as follows:  $w_0 = \max\{\text{bound}, 1 - \text{bound} + \sum_{i=1..n} w_i\}$ . Two counters  $C_1$  and  $C_2$  are also maintained. Counters are updated when an atom is inferred as true or false and they model when the inference rule can be applied. The algorithm for propagating the aggregate is the following:

1. Initialize  $C_1 = w_0 + \text{bound} - 1$  and  $C_2 = \sum_{i=0..n} w_i - \text{bound}$ .
2. Set  $C_1 = C_1 - w_i$ , when an atom  $b_i$  ( $i = 1, \dots, m$ ) becomes true or an atom  $b_i$  ( $i = 0, m + 1, \dots, n$ ) becomes false.
3. Set  $C_2 = C_2 - w_i$ , when an atom  $b_i$  ( $i = 1, \dots, m$ ) becomes false or an atom  $b_i$  ( $i = 0, m + 1, \dots, n$ ) becomes true.
4. Infer an undefined atom  $b_i$  ( $i = 1, \dots, m$ ) as true if  $w_i > C_2$  and as false if  $w_i > C_1$ .



5. Infer an undefined atom  $b_i (i = 0, m + 1, \dots, n)$  as true if  $w_i > C_1$  and as false if  $w_i > C_2$ .

Concerning the implication graph, given an aggregate of the form (4.1) and an atom  $a$ , a literal  $\ell = a$  or  $\ell = \text{not } a$  is inferred as true when the weight associated to  $a$  is greater than a particular counter  $C$ . In the implication graph we add the node  $\ell$ , and arcs  $(\sim\ell', \ell)$  for each  $\ell'$  that is responsible of the updating of  $C$ . Note that this inference rule is implemented in WASP 2 as described in [56].

**Example 11** (Aggregates propagation). Consider the following aggregate atom  $\{a_1 = 1, a_2 = 2, a_3 = 3, \text{not } a_4 = 4\} \geq 2$ . The corresponding data structure is the following  $[\text{not } a_0 = 9, a_1 = 1, a_2 = 2, a_3 = 3, \text{not } a_4 = 4]$ , where  $a_0$  represents the aggregate atom and the weight 9 is the maximum value between the bound of the aggregate (2) and the sum of the weights  $w_1, \dots, w_4$  plus one minus the bound (9). Counters  $C_1$  and  $C_2$  are initially set to 10 and 17, respectively. Suppose that  $a_1$  is inferred as true. The counter  $C_1$  is then updated. The new value of  $C_1 = 9$ , i.e. the old value minus the weight associated to  $w_1$ . Next, assume that  $a_3$  is set as false, thus  $C_2$  is updated and the new value is 14. Finally, suppose that  $\text{not } a_4$  becomes true. Counter  $C_1$  is updated and the new value is 5. At this point, atom  $w_0$  is inferred as true because its weight (9) is greater than the counter  $C_1$  (5). In the implication graph we add the node  $w_0$ , and arcs  $(\text{not } a_1, w_0)$  and  $(a_4, w_0)$ .  $\triangleleft$

**Unfounded-free Propagation.** This inference rule detects the sets of atoms that are unfounded and then propagates unfounded atoms as false. Checking whether a set of atoms is unfounded is known to be polynomial for HCF components, while it is a co-NP complete problem for non-HCF components [2]. The following polynomial procedure for unfounded-free propagation is thus applied to cyclic HCF components, whereas unfounded sets of non-HCF components are detected during answer set checking (see Section 4.5).

Algorithm 5 reports the polynomial algorithm for finding unfounded sets of a HCF component by means of source pointers [18]. Each atom in a given cyclic and HCF-component is associated with a rule modeling founded support. Such rules are referred to as *source pointers*. The unfounded set check is performed after the invalidation of any source pointer of a set of atoms  $S$ . Briefly, the idea is to find a new source pointer for each atom in  $S$ . If the algorithm fails to find a new source pointer for an atom  $a$  in  $S$ , then  $a$  is considered as unfounded.

---

**Algorithm 5:** Unfounded-free Check for HCF components

---

**Input** : An HCF component  $C$   
A set  $S \subseteq C$  of atoms with no source pointers  
An interpretation  $I$

**Output:** *true* if the interpretation  $I$  is unfounded-free, *false* otherwise

```
1 begin
2   foreach  $a \in S$  do
3     if  $\text{FindSourcePointer}(a, C, S, I)$  then
4        $S := S \setminus \{a\}; Q := \{a\};$ 
5       foreach  $a' \in Q$  do
6          $T := \{r : a' \in B^+(r), H(r) \cap S \neq \emptyset, B^+(r) \cap S = \emptyset\};$ 
7         foreach rule  $r \in T$  do
8           Let  $a''$  be the unique atom in  $H(r) \cap S$ ;
9           if  $r \in \text{supp}(a'', I)$  then
10             $S := S \setminus \{a''\}; Q := Q \cup \{a''\};$ 
11 return  $S = \emptyset$  ;
```

---

In more detail, the algorithm is invoked when atoms in a set  $S$  in a HCF component  $C$  are not associated to any source pointers. For each atom  $a$  in  $S$  the function `FindSourcePointer` searches for a source pointer for  $a$ . If  $a$  has a source pointer then  $a$  is removed from  $S$  and  $a$  is added to the set of founded atoms  $Q$ . Then for each founded atom  $a' \in Q$  a set  $T$  is computed. A rule  $r$  is in the set  $T$  if  $a'$  appears in the positive body of  $r$ , the head of  $r$  contains an atom that is in  $S$  and there is no atom in the positive body of  $r$  that is in  $S$ . For each rule  $r$  in  $T$ , at most one atom in the head of  $r$  is in the component  $C$  (otherwise the component would be non-HCF). Let  $a''$  be this atom if  $r$  is a possible supporting rule for  $a''$ , then  $a''$  is founded and thus is removed from  $S$  and added to  $Q$ . At the end of the algorithm all atoms in  $S$  are unfounded.

When an unfounded set  $S$  is found, our learning scheme adds one learned constraint for each atom  $a \in S$ . Constraints for other atoms in  $S$  will be learned on subsequent calls to the function, unless an inconsistency arises during unit propagation. In case of inconsistencies, indeed, the unfounded set  $S$  is recomputed. Given an atom  $a \in S$  the added constraint is computed as follows:

1.  $R := \{r : H(r) \cap S \neq \emptyset, B^+(R) \cap S = \emptyset\}$ .
2. For each  $r \in R$ , let  $\ell_r$  be the first literal satisfying  $r$ .

---

**Function** FindSourcePointer(Atom  $a$ , HCF Component  $C$ ,  
SetOfAtoms  $S$ , Interpretation  $I$ )

---

```

1 begin
2    $Ext := \{r : r \in \text{supp}(a, I), r \text{ is external of } C\};$ 
3   if  $Ext \neq \emptyset$  then
4     return true;
5    $Int := \{r : r \in \text{supp}(a, I), r \text{ is internal of } C, B^+(r) \cap S = \emptyset\};$ 
6   if  $Int \neq \emptyset$  then
7     return true;
8   return false;

```

---

3. The constraint is  $\leftarrow a, \{\sim \ell_r : r \in R\}$ .

#### 4.4.2 Learning

Learning acquires information from conflicts in order to avoid exploring the same conflictual search branch several times. Learning is very important for pruning the backtracking tree and to implement an efficient learning-based heuristic. WASP 2 adopts a learning scheme based on the concept of the first Unique Implication Point (UIP) (see Section 2.3.1), which is computed by analyzing the implication graph. A node  $n$  in the implication graph is a UIP for a decision level  $d$  if all paths from the choice of level  $d$  to the conflict literals pass through  $n$ . The first UIP is the UIP for the decision level of the conflict that is closest to the conflict. The learning scheme add a constraint built as follows: Let  $u$  be the first UIP. Let  $L$  be the set of literals different from  $u$  occurring in a path from  $u$  to the conflict literals. The learned constraint comprises  $u$  and each literal  $\ell$  such that the decision level of  $\ell$  is lower than the one of  $u$  and there is an arc  $(\ell, \ell')$  in the implication graph for some  $\ell' \in L$ .

### 4.5 Answer Set Checker

Answer Set Checker is the module that verifies whether a supported model  $M$  is unfounded-free, i.e.  $M$  is an answer set. Answer set checking is a co-NP-complete problem in case of non-HCF components and it is usually addressed by reducing the answer set checking (or stable model checking) problem to the unsatisfiability problem, which is the problem to decide whether a CNF formula is unsatisfiable, as described in [27] and implemented in the ASP

solver DLV. In this section, we first describe the algorithm proposed in [27], and then we present a new algorithm addressing the drawbacks related to the previous approach.

**DLV algorithm [27].** Given a program  $\Pi$  and an interpretation  $I$  to be checked whether it is unfounded-free, the DLV algorithm rewrites the program into a CNF formula. The interpretation  $I$  is unfounded-free iff the corresponding CNF formula is unsatisfiable, otherwise  $I$  contains an unfounded set.

Given a program  $\Pi$ , a non-HCF component  $C$  and an interpretation  $I$  a CNF formula  $\varphi_C^I$ , is built in the following way:

1.  $\varphi_C^I := \emptyset$ ;  $\Pi_{aux} := \emptyset$ .
2.  $\Pi_{aux} := \{r \in \Pi : H(r) \cap I \subseteq C \neq \emptyset\}$ .
3. Remove all rules in  $\Pi_{aux}$  whose body is false w.r.t.  $I$ .
4. For each rule  $r \in \Pi_{aux}$ , remove all negative literals and all positive external literals from the body of  $r$  and all false atoms w.r.t.  $I$  from the head of  $r$ .
5.  $\varphi_C^I := \{C(r) : r \in \Pi_{aux}\}$ , where  $C(r)$  is the clause representation of  $r$ .
6.  $\varphi_C^I := \varphi_C^I \cup \{\{\neg a : a \in (C \cap I)\}\}$ .

The major drawback of this algorithm is that the CNF formula  $\varphi_C^I$  is built every time an interpretation  $I$  has to be checked. This leads to a computational overhead. Moreover, since the CNF formula is created at each check, the solver cannot benefit of the knowledge of previous computations. We will now show an alternative strategy that creates once a different CNF formula, which does not depend on an interpretation, and it is thus suitable for being reused in different calls to the answers set checker.

#### 4.5.1 Unfounded-free Check for non-HCF components

The WASP 2 algorithm for checking whether an interpretation is unfounded-free addresses the weakness of the DLV algorithm by using the incremental interface of modern SAT solvers. In particular, it creates a CNF formula for each component  $C$  only once, and then to reuses it in every check.

Given a program  $\Pi$  and a non-HCF component  $C$ , the CNF formula  $\varphi_C$  is built by WASP 2 in the following way:

1.  $\varphi_C := \emptyset$ ;  $\Pi_{aux} := \emptyset$ .

2.  $\Pi_{aux} := \{r \in \Pi : H(r) \cap C \neq \emptyset\}$ .
3. For each atom  $a \notin C$ , let  $aux_a$  be a fresh variable not appearing in  $\Pi_{aux}$ , replace each occurrence of  $a$  in all positive bodies of  $\Pi_{aux}$  with  $\sim aux_a$ .
4. For each atom  $a \in C$ , let  $aux_a$  be a fresh variable not appearing in  $\Pi_{aux}$ , replace each occurrence of  $\sim a$  in all negative bodies of  $\Pi_{aux}$  with  $\sim aux_a$ .
5.  $\varphi_C := \{C(r) : r \in \Pi_{aux}\}$ , where  $C(r)$  is the clause representation of  $r$ .

**Example 12.** Consider the program  $\Pi_1$  in Example 9:

$$\begin{array}{ll}
r_1 : & a \leftarrow e & r_5 : & c \leftarrow a, e \\
r_2 : & a \leftarrow b & r_6 : & c \leftarrow d \\
r_3 : & b \leftarrow a & r_7 : & d \leftarrow c \\
r_4 : & e \vee f \leftarrow & r_8 : & c \vee d \leftarrow f
\end{array}$$

The component  $C_2 = \{c, d\}$  is non-HCF and the program  $\Pi_{aux}$  is the following:

$$\begin{array}{ll}
r_1 : & c \leftarrow a, e & r_2 : & c \leftarrow d \\
r_3 : & d \leftarrow c & r_4 : & c \vee d \leftarrow f
\end{array}$$

Steps 3 and 4 then modify  $\Pi_{aux}$  in the following way:

$$\begin{array}{ll}
r_1 : & c \leftarrow \text{not } aux_a, \text{ not } aux_e & r_2 : & c \leftarrow d \\
r_3 : & d \leftarrow c & r_4 : & c \vee d \leftarrow \text{not } aux_f
\end{array}$$

Finally,  $\varphi_C$  is created and consists of the following set of clauses:

$$\{c, aux_a, aux_e\} \quad \{c, \neg d\} \quad \{d, \neg c\} \quad \{c, d, aux_f\}. \triangleleft$$

Note that, for each atom in  $\Pi$  this procedure introduces at most one fresh atom. Moreover, a CNF formula  $\varphi_C$  is associated to exactly one non-HCF component  $C$ .

A pseudo-code description of the algorithm for the unfounded-free check of non-HCF components implemented in WASP 2 is shown in Algorithm 6. In a nutshell, the idea is to simulate the DLV algorithm by using two set of assumptions  $assum_{\wedge}$  and  $assum_{\vee}$  (see Section 2.4). The set  $assum_{\wedge}$  is populated in order to simulate steps 3 and 4 of the DLV algorithm. While, the set  $assum_{\vee}$  is populated by adding the literals contained in the clause added at step 6 of the DLV algorithm. In more detail, the algorithm takes as input a non-HCF component  $C$  and an interpretation  $I$  and returns as output

*true* if  $I$  is unfounded-free, *false* otherwise. In the beginning the Function  $\text{ComputeAssumptions}(C, I)$  computes the set of assumptions  $assum_{\wedge}$  and  $assum_{\vee}$  (initially set to  $\emptyset$ ).  $\text{ComputeAssumptions}(C, I)$  uses two sets of atoms  $C_{in}$  and  $C_{out}$ , containing all atoms in  $C$  that are false w.r.t.  $I$  and all atoms in the  $C$  that are not false w.r.t.  $I$ , respectively. Moreover,  $E_p$  represents a set of positive external literals of the component and  $E_n$  represents a set of negative external literals of the component. The set  $assum_{\vee}$  is extended by adding  $\{\sim a\}$ , for each atom  $a \in C_{out}$  (line 2). While  $assum_{\wedge}$  is extended by the following literals:

- $\{\sim a\}$  and  $\{\sim aux_a\}$ , for each atom  $a \in C_{in}$ .
- $\{aux_a\}$ , for each atom  $a$  in  $C_{out}$ .
- $\{aux_{\ell}\}$ , for each literal  $\ell \in E_p$  such that  $\ell$  is true w.r.t.  $I$ .
- $\{\sim aux_{\ell}\}$ , for each literal  $\ell \in E_p$  such that  $\ell$  is not true w.r.t.  $I$ .
- $\{\ell\}$ , for each literal  $\ell \in E_n$  such that  $\ell$  is true w.r.t.  $I$ .
- $\{\sim \ell\}$ , for each literal  $\ell \in E_n$  such that  $\ell$  is not true w.r.t.  $I$ .

The SAT solver is then invoked. If the formula  $\varphi_C$  is unsatisfiable under the assumptions  $assum_{\wedge}$  and  $assum_{\vee}$ , then the interpretation  $I$  is unfounded-free. Otherwise, the satisfying assignment  $I'$  for  $\varphi_C$  is returned and an unfounded set  $X$  can be computed. In fact, the unfounded set  $X$  is composed by all non-false atoms in the component  $C$  that are false in  $I'$ . The falsity of all atoms in the  $X$  can be inferred by adding a constraint to the program. However, we heuristically limit the number of added constraints by adding a constraint only for the first atom in the unfounded set. This constraint is computed as in the HCF case.

Note that the algorithm is defined for working on a generic SAT solver. In our implementation, we used a specialized instance of the same algorithm employed for model generation which takes as input a CNF formula and does not invoke ASP specific inference rules, controlled by using the incremental interface (see Section 2.4) of WASP 2.

**Example 13.** Consider the following program  $\Pi$ :

$$\begin{array}{ll} r_1 : a \vee b \leftarrow c & r_3 : b \leftarrow a \\ r_2 : a \leftarrow b & r_4 : c \vee d \leftarrow \end{array}$$

The component  $C = \{a, b\}$  is non-HCF and the CNF formula  $\varphi_C$  is the following:

$$\{a, b, aux_c\} \quad \{a, \neg b\} \quad \{b, \neg a\}$$

---

**Algorithm 6:** Unfounded-free Check for non-HCF components

---

**Input** : A non-HCF component  $C$  and an interpretation  $I$

**Output:** *true* if the interpretation  $I$  is unfounded-free, *false* otherwise

```
1 begin
2    $assum_{\wedge} := \emptyset; \quad assum_{\vee} := \emptyset; \quad I' := \emptyset;$ 
3    $(assum_{\wedge}, assum_{\vee}) := \text{ComputeAssumptions}(C, I);$ 
4   //let  $\varphi_C$  be the CNF formula associated to  $C$ 
5    $(res, I', Core) := \text{SATSolver}_{assum}(\varphi_C, assum_{\wedge}, assum_{\vee});$ 
6   if  $res = UNSAT$  then
7     return true;
8   else
9     //unfounded set  $U := \{a : \sim a \in assum_{\vee}, \sim a \in I'\};$ 
10    return false;
```

---

---

**Function**  $\text{ComputeAssumptions}(C, I)$ 

---

```
1  $C_{in} := \{a \in C : \sim a \in I\}; \quad C_{out} := \{a \in C : \sim a \notin I\};$ 
2  $assum_{\vee} := \{\sim a : a \in C_{out}\};$ 
3  $assum_{\wedge} := \{\sim a : a \in C_{in}\} \cup \{aux_a : a \in C_{out}\} \cup \{\sim aux_a : a \in C_{in}\};$ 
4  $E_p := \{\ell : \ell \text{ is external, } \ell \text{ is positive}\};$ 
5  $E_n := \{\ell : \ell \text{ is external, } \ell \text{ is negative}\};$ 
6  $assum_{\wedge} := assum_{\wedge} \cup \{aux_{\ell} : \ell \in E_p, \ell \in I\} \cup \{\sim aux_{\ell} : \ell \in E_p, \ell \notin I\};$ 
7  $assum_{\wedge} := assum_{\wedge} \cup \{\ell : \ell \in E_n, \ell \in I\} \cup \{\sim \ell : \ell \in E_n, \ell \notin I\};$ 
8 return  $(assum_{\wedge}, assum_{\vee});$ 
```

---

Suppose that the interpretation to check is  $I = \{a, b, \sim c, d\}$ . The sets used by the algorithm are the following:  $C_{in} := \emptyset$ ,  $C_{out} := \{a, b\}$ ,  $E_p := \{c, d\}$  and  $E_n := \{\text{not } c, \text{not } d\}$ . The set  $assum_{\vee} := \{\text{not } a, \text{not } b\}$  and the set  $assum_{\wedge} := \{aux_a, aux_b, \text{not } aux_c, aux_d, \text{not } c, d\}$ . Note that, since atoms  $aux_a, aux_b, aux_d, c$  and  $d$  do not appear in the CNF formula  $\varphi_C$ , in this case their value is in fact irrelevant for the correctness of the algorithm. The SAT solver is then invoked on the CNF formula  $\varphi_C$ , which is unsatisfiable under the assumptions  $assum_{\vee}$  and  $assum_{\wedge}$ , and thus the interpretation  $I$  is unfounded-free.  $\triangleleft$

## 4.5.2 Partial Checks

WASP 2 implements an optimization of the main algorithm described in [57] motivated by the following observation. When an unfounded set check fails

Table 4.1: Number of solved instances, average running time and average memory usage

Problem	#	CLASP 3.0.1			WASP 2		
		sol.	t	mem.	sol.	t	mem.
BottleFillingProblem	30	30	7.2	302.3	30	6.5	616.6
GracefulGraphs	30	15	103.1	17.3	8	74.7	65.5
GraphColouring	30	13	129.6	6.8	8	85.3	20.1
HanoiTower	30	28	52.4	22.1	30	38.2	61.5
IncrementalScheduling	30	2	210.3	2961.6	3	164.2	1304.0
Labyrinth	30	26	57.3	79.8	24	149.3	694.2
NoMystery	30	9	73.5	159.9	6	74.2	391.0
PermutationPatternMatching	30	18	16.2	339.7	26	48.8	324.9
QualitativeSpatialReasoning	30	30	44.8	431.7	23	56.5	1184.8
RicochetRobot	30	30	92.5	25.6	30	159.9	116.2
Sokoban	30	11	37.5	14.4	12	73.4	60.2
Solitaire	27	22	13.9	11.1	22	11.9	29.0
StableMarriage	30	17	145.3	1468.1	30	228.5	314.2
VisitAll	30	19	37.1	17.1	19	90.0	90.5
Weighted-Sequence Problem	30	25	65.2	236.5	26	140.6	883.7
<b>Total</b>	<b>447</b>	<b>295</b>	<b>59.1</b>	<b>240.3</b>	<b>297</b>	<b>93.8</b>	<b>399.4</b>

the solver backtracks until it is deactivated. However, the interpretation after backtracking takes place may still contain another unfounded set. The basic strategy would wait until a new answer set candidate is found for calling the Answer Set Checker and detecting that actually the candidate is unfounded. This can be avoided if an additional unfounded-free check is performed on a partial interpretation after backtracking from answer set checking failures. This smarter strategy is called partial answer set checking, and was already used also in DLV [58]. We observe that each partial check is usually similar to its antecedent check since the interpretation to check usually changes slightly. Thus, the ability of reusing the computation of antecedent checks is crucial for performing fast checks. However, we also observed that when the number of answer set candidates is high the number of partial checks shows a strong growth and they are not helpful for pruning the search space. Thus, albeit partial checks ideally improve the basic algorithm [58], they can lead to a deterioration of the performances in specific benchmarks. Thus, WASP 2 uses an heuristic for enabling and disabling partial checks during the computation. In particular, partial checks are disabled when the number of learned constraints from unfounded sets is greater than half of the other learned constraints; and are enabled otherwise. The idea is to *balance* the number of learned constraints from unfounded sets and from conflicts. This heuristic has been empirically proved to be effective (see Section 4.6).



## 4.6 Experiments

In this section we report the results of an experiment assessing the performance of WASP 2. Model Generator and Answer Set Checker are compared in separate sections. Moreover, we compare WASP 2 with the state of the art ASP solver CLASP version 3.0.1.

The experiments were run on a four core Intel Xeon CPU X3430 2.4 GHz, with 4 GB of physical RAM and running Debian Linux 7.3 (kernel ver. 3.2.0-4-amd64). Only one core was enabled, and time and memory limits were set to 600 seconds and 3 GB, respectively. Performance was measured using the tools `pyrunlim` and `pyrunner` (<https://github.com/alviano/python>). In the experiments Gringo 3.0.5 [47] is used as grounder.

### 4.6.1 Model Generator Evaluation

In this section we report the experiments for evaluating the performances of the Model Generator. Tested instances were taken from the 4th ASP Competition<sup>1</sup> [23], in particular all instances in the NP category and the polynomial problem StableMarriage. This category includes planning domains, temporal and spatial scheduling problems, combinatorial puzzles, graph problems, and a number of real-world domains in which ASP has been applied. (See [23] for an exhaustive description of the benchmarks.) Note that we excluded from our analysis the problem KnightTour because no solver was able to solve instances of the 4th ASP Competition within the memory and the time limit.

**Results.** Table 4.1 summarizes the number of solved instances, the average running times in seconds and the average memory usage in megabytes for each solver. In particular, the first column reports the total number of instances (#); the remaining columns report the number of solved instances within the time-out (sol.), the running times averaged over solved instances ( $t$ ) and the average memory usage over solved instances (mem.).

As a general comment, WASP 2 is comparable with the state of the art solver CLASP 3.0.1. In particular, WASP 2 solves two more instances than CLASP 3.0.1. Analyzing the results in more detail, WASP 2 and CLASP 3.0.1 exhibit significantly different performances in some specific benchmarks. For example in GracefulGraphs, GraphColouring and QualitativeSpatialReasoning, where CLASP 3.0.1 solved 7, 5 and 7 instances more than WASP 2, respectively, and in PermutationPatternMatching and StableMarriage where

---

<sup>1</sup><https://www.mat.unical.it/aspcomp2013/OfficialProblemSuite>

WASP 2 solved 8 and 13 instances more than CLASP 3.0.1, respectively. Those differences are due to the different techniques applied in WASP 2 and CLASP 3.0.1 that are discussed in detail in the following. CLASP 3.0.1 has better performances than WASP 2 in GracefulGraphs and GraphColouring, whose instances show different symmetric solutions. The heuristic of WASP 2 seems to be ineffective when instances have different symmetric solutions, hence in these cases WASP 2 performs poorly compared to CLASP 3.0.1. Moreover, CLASP 3.0.1 performs better than WASP 2 also in QualitativeSpatialReasoning. In this benchmark, CLASP 3.0.1 has a lower memory usage than WASP 2. In particular, WASP 2 exceeds the memory limit in 7 instances while CLASP 3.0.1 has no memory out at all and the average memory usage of WASP 2 is about 3 times greater than the average memory usage of CLASP 3.0.1 on solved instances. On the contrary, WASP 2 has a better memory usage in PermutationPatternMatching and in StableMarriage. Concerning PermutationPatternMatching, WASP 2 exceeds the memory limit in 4 instances while CLASP 3.0.1 exceeds the memory limit in 12 instances. Similar considerations hold for StableMarriage, where CLASP 3.0.1 exceeds the memory limit in 13 instances while WASP 2 is able to solve all instances within the memory limit. Also by looking at solved instances, the average memory usage of CLASP 3.0.1 is about 4 times greater than the average memory usage of WASP 2.

Some additional observations can be made by studying in more detail memory usage of WASP 2 and CLASP 3.0.1. To this end we report in Figure 4.2 four plots depicting the memory consumption during the execution of the solvers. In particular, we focus on two specific instances solved by WASP 2 in which CLASP 3.0.1 exceeds the memory limit and two instances solved by CLASP 3.0.1 in which WASP 2 exceeds the memory limit. Figure 4.2(a) reports the result for an instance of StableMarriage. StableMarriage is a problem in the P category and its encoding is unstratified. In this case WASP 2 performs better than CLASP 3.0.1. We observed that the grounder gringo produces several duplicated rules in the grounding phase (around 85% in some instances). WASP 2 does not store such rules and thus it is able to maintain its memory consumption always under 500 MB, while CLASP 3.0.1 exceeds the memory limit while reading the input. Figure 4.2(b) shows the result for an instance of IncrementalScheduling. Also in this case we observe that WASP 2 is less memory demanding than CLASP 3.0.1, that exceeds the memory limit while reading the input. By our analysis, the data structures of WASP 2 seem to be less memory demanding than the ones of CLASP 3.0.1 in this specific benchmark. In fact, WASP 2 has a slight growth of memory usage during the time, while CLASP 3.0.1 exceeds the time limit after 177 seconds. Figure 4.2(c) depicts the results for an instance of QualitativeS-

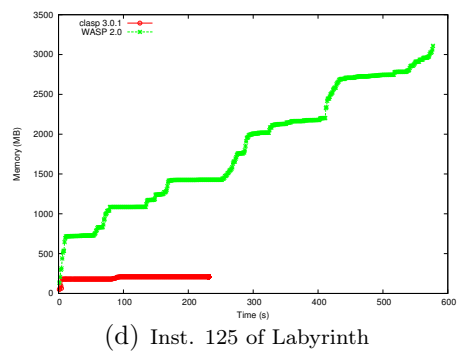
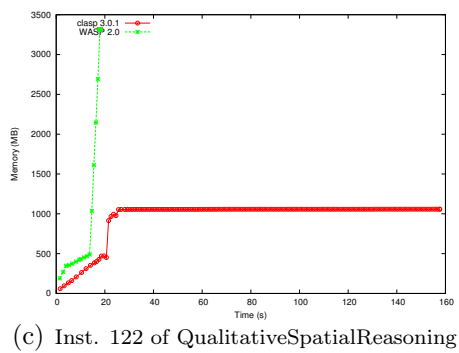
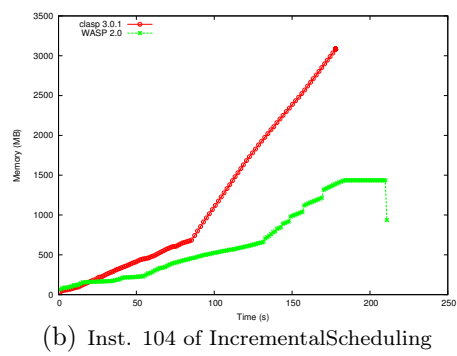
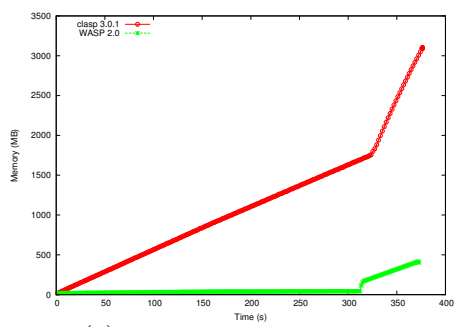


Figure 4.2: Comparison of WASP 2 and CLASP 3.0.1 memory consumption

Table 4.2: Number of solved instances and average running time

Problem	#	CLASP 3.0.1		WASP 2 <sub>BAL</sub>		WASP 2 <sub>PC</sub>		WASP 2 <sub>NOPC</sub>	
		sol.	t	sol.	t	sol.	t	sol.	t
2QBF	65	13	43.63	14	95.63	9	76.22	14	95.56
ConformantPlanning	23	22	9.23	22	48.66	21	20.23	18	51.94
Repair	60	42	3.56	40	30.90	40	41.59	39	30.18
<b>Total</b>	<b>148</b>	<b>77</b>	<b>11.94</b>	<b>76</b>	<b>47.97</b>	<b>70</b>	<b>39.63</b>	<b>71</b>	<b>48.59</b>

patialReasoning, a problem in the NP category whose encoding has several disjunctive rules. In this case WASP 2 has a peak of memory usage around 18 seconds while CLASP 3.0.1 handles efficiently this instance. We found out that the peak of memory usage occurs while shifting the disjunctive rules. Thus, an improvement of the performances could be obtained by optimizing the Clark’s completion in this specific case. Finally, Figure 4.2(d) reports the result for Labyrinth, a problem in the NP category whose encoding is recursive. In this case CLASP 3.0.1 performs better than WASP 2. In fact, WASP 2 has a slight growth of memory usage related to the constraints added during the unfounded-free propagation. On the contrary, CLASP 3.0.1 has a constant memory usage in this benchmark. However, the performances of WASP 2 could be improved by both reducing the number and the size of learned constraints during the unfounded-free propagation.

## 4.6.2 Answer Set Checker Evaluation

In this section we report the experiments for evaluating the performances of the Model Generator. Tested instances include all *2QBF* instances of the application track used in the 2014 QBF Gallery (<http://qbf.satisfiability.org/gallery/>), all the instances of *Conformant Planning* and *Repair* used in [21]. We excluded from our analysis the time of the grounder since instances were publicly available already grounded.

The results are summarized in Table 4.2, that reports the number of solved instances (sol.) and the average time (t) of solved instances in seconds for each solver. The first column (#) reports the total number of instances for each set. We compared three versions of WASP 2: WASP 2<sub>BAL</sub>, i.e. WASP 2 using the balanced heuristic; WASP 2<sub>PC</sub>, i.e. WASP 2 using partial checks; and WASP 2<sub>NOPC</sub>, i.e. WASP 2 with no partial checks.

We observe that WASP 2<sub>BAL</sub> is the best performing version of WASP 2 in the overall. In fact, WASP 2<sub>BAL</sub> solves 6 and 5 more instances than WASP 2<sub>PC</sub> and WASP 2<sub>NOPC</sub>, respectively. Concerning partial checks, we observe that they lead in a degradation of the performances in 2QBF and they are effective in ConformantPlanning. Our explanation is related to the number of answer set candidates. In more detail, if the Model Generator finds a high

number of answer set candidates then partial checks are not effective. In this sense, WASP 2<sub>BAL</sub> is able to perform partial checks when it is really needed. In fact, WASP 2<sub>BAL</sub> behaves the same as WASP 2<sub>PC</sub> in 2QBF. In ConformantPlanning, WASP 2<sub>BAL</sub> improves both WASP 2<sub>PC</sub> and WASP 2<sub>NOPC</sub> by solving more instances. In the overall, WASP 2<sub>BAL</sub> is also comparable with CLASP 3.0.1 on this benchmark set. We observe that CLASP 3.0.1 is faster than WASP 2<sub>BAL</sub> in terms of average time, in ConformantPlanning and in Repair. In the latter, CLASP 3.0.1 solves 2 more instances than WASP 2<sub>BAL</sub>. However, WASP 2<sub>BAL</sub> solves one more instance than CLASP 3.0.1 in 2QBF. As conclusion, WASP 2<sub>BAL</sub> implements a promising heuristic for scheduling partial checks since it improves the performance of both WASP 2<sub>PC</sub> and WASP 2<sub>NOPC</sub>.

# Chapter 5

## Optimum Answer Set Search

Optimization problems in ASP are usually modeled by means of programs with weak constraints [9]. In this chapter we present and compare alternative solutions for evaluating programs with weak constraints, including model-guided and core-guided algorithms. In particular, we consider an algorithm inspired by *optsat* [32] that we call OPT and its variant that we called BASIC; the model-guided algorithm MGD [31] introduced for solving MaxSAT; and the core-guided algorithms OLL [33] that has been introduced in the context of ASP and then successfully ported to MaxSAT [37]; PMRES implemented in the MaxSAT solver EVA [35] and BCD implemented in the MaxSAT solver MSUNCORE [36]. These algorithms were implemented in our solver WASP 2 and tested on publicly-available benchmarks.

### 5.1 Preliminaries

The algorithms considered in this chapter can be classified in two categories, namely *core-guided* and *model-guided* algorithms. These algorithms operate by calling iteratively an ASP solver, thus we introduce in this section some preliminary definitions and a common algorithmic framework. All algorithms considered in this chapter are based on relaxation of constraints.

**Definition 14.** Given a weak constraint  $w$ , we call the constraint obtained by adding to  $w$  a fresh atom  $aux_w$  the *relaxation* of  $w$ .

Core-guided algorithms are based on the concept of unsatisfiable core first introduced in the context of SAT solving [59]. According to the original definition, an unsatisfiable core of an unsatisfiable CNF  $\varphi$  is a subset of  $\varphi$  that is also unsatisfiable. The analogous notion in ASP can be stated as follows:

---

**Function** RelaxWeakConstraint( $r$ : weak constraint, **var**  $R$ : set)

---

```

1 begin
2   Let  $aux$  be a fresh atom;
3    $R := R \cup \{aux = weight(r)\}$ ;
4   return  $\leftarrow B(r), \sim aux$ ;

```

---

**Definition 15.** An unsatisfiable core of an incoherent propositional program  $\Pi_R$  is a set  $\Pi_{core} \subseteq constraints(\Pi_R)$  such that  $rules(\Pi_R) \cup \Pi_{core}$  is incoherent.

**Example 14.** Given program  $\Pi = (\Pi_R, \Pi_W)$ , where  $\Pi_R$  consists of the following rules:

$$\begin{array}{llll}
 r_1 : a \vee b \leftarrow & r_3 : \leftarrow_1 a & r_5 : \leftarrow_1 b \\
 r_2 : c \vee d \leftarrow & r_4 : \leftarrow_2 c & r_6 : \leftarrow_2 d
 \end{array}$$

and  $\Pi_W = \{r_3, r_4, r_5, r_6\}$ . A core of  $\Pi_R$  is  $\{r_3, r_5\}$ .  $\triangleleft$

Core-guided algorithms start by considering weak constraints as hard constraints and then selectively relaxing some of them (i.e., replacing them with their relaxation) until an optimum cost is found. Model-guided algorithms instead start by ignoring weak constraints and then enforce an improvement on the cost of the computed stable models.

In the following if  $\Pi = (\Pi_R, \Pi_W)$  is an input program then  $\Pi_R \setminus \Pi_W$  is assumed to be coherent. Moreover, we use a variant of function ComputeAnswerSet( $\Pi, I$ ) (see Chapter 4) called ComputeAnswerSet<sub>cores</sub>( $\Pi, PrefChoices$ ). ComputeAnswerSet<sub>cores</sub>( $\Pi, PrefChoices$ ) takes as input a program  $\Pi$  and a set of literals  $PrefChoices$ , and returns as output a triple  $(res, \Pi_{core}, I)$ , where  $res$  is a string,  $\Pi_{core}$  a set of rules and  $I$  an interpretation. The function searches for a stable model of  $\Pi$ . If one is found, say  $I$ , the function returns  $(Coherent, \emptyset, I)$ . Otherwise, the function returns  $(Incoherent, \Pi_{core}, \emptyset)$ , where  $\Pi_{core}$  is an unsatisfiable core of  $\Pi$ . During the search, the first choices are those specified by the input parameter  $PrefChoices$ . In the following we describe three model-guided algorithms namely OPT, BASIC, and MGD; and three core-guided algorithms namely OLL, PMRES, and BCD.

## 5.2 Algorithm OPT

The pseudo code of algorithm OPT is reported in Algorithm 7. All weak constraints are relaxed in the beginning and relaxing atoms are added to the set  $PrefChoices$ . Relaxing atoms are added in  $PrefChoices$  in descending order according to weights of the corresponding weak constraints. During the

---

**Algorithm 7:** OPT

---

**Input** : A program  $\Pi = (\Pi_R, \Pi_W)$

**Output:** The optimum cost OPT for  $\Pi$

```
1 begin
2    $(R, OPT) := (\emptyset, 1 + \sum_{r \in \Pi_W} weight(r))$ ;
3   foreach  $r \in \Pi_W$  do
4      $\Pi_R := (\Pi_R \setminus \{r\}) \cup \{RelaxWeakConstraint(r, R)\}$ ;
5      $PrefChoices := \{\sim aux \mid aux = w \in R\}$ ;
6      $(res, \Pi_{core}, I) := ComputeAnswerSet_{cores}(\Pi_R \cup \{\leftarrow R \geq$ 
        $OPT\}, PrefChoices)$ ;
7     if  $res = Incoherent$  then return OPT;
8      $OPT := cost(\Pi_W, I)$ ;
9   goto 6;
```

---

stable model search, the branching heuristic is modified in order to prioritize atoms in *PrefChoices*. Moreover, a constraint with an aggregate atom is used to force an improvement of the solution within each stable model found. The algorithm runs searching for a stable model. If a stable model is found then its cost is computed and an aggregate atom forcing the solver to obtain a solution with a better cost is added to the program. Otherwise, the algorithm terminates returning the optimum cost found so far.

**Example 15.** Consider the program in Example 14. Initially, *OPT* is 7. In the beginning all weak constraints are relaxed as follows:

$$\begin{aligned} r'_3 &:\leftarrow a, \text{ not } aux_3 & r'_5 &:\leftarrow b, \text{ not } aux_5 \\ r'_4 &:\leftarrow c, \text{ not } aux_4 & r'_6 &:\leftarrow d, \text{ not } aux_6. \end{aligned}$$

Moreover,  $R$  is  $\{aux_3 = 1, aux_4 = 2, aux_5 = 1, aux_6 = 2\}$  and *PrefChoices* is  $\{\sim aux_4, \sim aux_6, \sim aux_3, \sim aux_5\}$ . Then, a stable model  $I$  for  $\Pi_R$  is found, say  $\{a, c, aux_3, aux_4\}$  of cost 3. *OPT* is thus updated to 3 and a new stable model is searched, with the additional constraint  $\leftarrow R \geq 3$ . The resulting program is actually incoherent, and the algorithm terminates.  $\triangleleft$

### 5.2.1 Algorithm BASIC.

A variant of Algorithm 7 is implemented as the standard strategy for handling optimization ASP problems in the solvers SMOBELS [18], DLV [17] and CLASP [54]. We will refer to such an algorithm as BASIC. In a nutshell, the only difference with Algorithm 7 is that set *PrefChoices* is empty. Stated differently, to obtain basic from Algorithm 7, line 5 is replaced by



---

**Algorithm 8:** MGD

---

**Input** : A program  $\Pi = (\Pi_R, \Pi_W)$   
**Output**: The optimum cost  $OPT$  for  $\Pi$

```
1 begin
2    $(\Pi_S, \Pi_R, R, OPT) := (\Pi_W, \Pi_R \setminus \Pi_W, \emptyset, 1 + \sum_{r \in \Pi_W} weight(r));$ 
3    $(res, \Pi_{core}, I) := ComputeAnswerSet_{cores}(\Pi_R \cup \{\leftarrow R \geq OPT\}, \emptyset);$ 
4   if  $res = Incoherent$  then return  $OPT$ ;
5    $OPT := min(cost(\Pi_W, I), OPT);$ 
6   foreach  $r \in \Pi_S$  such that  $I \not\models r$  do
7      $\Pi_S := \Pi_S \setminus \{r\};$ 
8      $\Pi_R := \Pi_R \cup \{RelaxWeakConstraint(r, R)\};$ 
9   goto 3;
```

---

$PrefChoices := \emptyset;$

so that the branching heuristic is free to choose any undefined literal even if some relaxation atom is still undefined.

### 5.3 Algorithm MGD

The pseudo code of the model-guided algorithm MGD is reported in Algorithm 8. The algorithm runs as follows. Weak constraints are initially ignored and a stable model is found. Recall that if  $\Pi = (\Pi_R, \Pi_W)$  is an input program then  $\Pi_R \setminus \Pi_W$  is assumed to be coherent. Violated weak constraints are relaxed and considered as hard constraints in the subsequent stable model searches. Moreover, the program is extended by a constraint of the form  $\leftarrow R \geq OPT$ , where  $R$  contains the relaxing atoms and the associated weights, and  $OPT$  is the current optimum cost. This process is iterated until the program becomes incoherent, which means that  $OPT$  is the optimum cost for the original program.

**Example 16.** Consider the program in Example 14. Initially,  $\Pi_S$  contains all weak constraints, i.e.,  $r_3$ – $r_6$ , which are instead removed from  $\Pi_R$ . The optimal value  $OPT$  is initially set to 7, and set  $R$  is empty. A stable model for  $\Pi_R = \{r_1, r_2\}$  is computed, say  $\{a, c\}$ , and stored in variable  $I$ . The cost of this solution is 3, thus  $OPT$  is updated. Weak constraints  $r_3$  and  $r_4$  are then relaxed, i.e., they are removed from  $\Pi_S$ , and  $\Pi_R$  is extended with the following constraints:

$$r'_3 : \leftarrow a, \text{ not } aux_3 \quad r'_4 : \leftarrow c, \text{ not } aux_4$$

---

**Algorithm 9:** OLL

---

**Input** : A program  $\Pi = (\Pi_R, \Pi_W)$   
**Output**: The optimum cost for  $\Pi$

```
1 begin
2    $(\Pi_S, \Pi_{aggr}) := (\Pi_W, \emptyset);$ 
3    $(res, \Pi_{core}, I) := ComputeAnswerSet_{cores}(\Pi_R \cup \Pi_{aggr}, \emptyset);$ 
4   if  $res \neq Incoherent$  then return  $cost(\Pi_W, I);$ 
5   foreach  $\leftarrow R \geq lb \in \Pi_{aggr} \cap \Pi_{core}$  such that  $|R| > lb$  do
6      $\Pi_S := \Pi_S \cup \{\leftarrow_1 R \geq lb\};$ 
7      $\Pi_{aggr} := (\Pi_{aggr} \setminus \{\leftarrow R \geq lb\}) \cup \{\leftarrow R \geq lb + 1\};$ 
8      $minWeight := \min(\{weight(r) \mid r \in \Pi_{core} \cap \Pi_S\});$ 
9      $R := \emptyset;$ 
10    foreach  $r \in \Pi_{core} \cap \Pi_S$  do
11      if  $weight(r) > minWeight$  then
12         $\Pi_S := \Pi_S \cup \{\leftarrow_{weight(r)-minWeight} B(r)\};$ 
13         $\Pi_S := \Pi_S \setminus \{r\};$ 
14         $\Pi_R := (\Pi_R \setminus \{r\}) \cup \{RelaxWeakConstraint(r, R)\};$ 
15     $\Pi_{aggr} := \Pi_{aggr} \cup \{\leftarrow R \geq 2\};$ 
16    goto 3;
```

---

where  $aux_3$  and  $aux_4$  are fresh atoms. After this process, set  $R$  is  $\{aux_3 = 1, aux_4 = 2\}$ , and the subsequent coherence test must satisfy the constraint  $\leftarrow R \geq 3$ . Let us assume that the returned stable model  $I$  is  $\{a, d, aux_3\}$ , with cost 3. Weak constraint  $r_6$  is then relaxed. Again, it is removed from  $\Pi_S$  and  $\Pi_R$  is extended with  $r'_6$ , i.e.,  $\leftarrow d$ , *not*  $aux_6$  where  $aux_6$  is a fresh atom.  $R$  is extended with  $aux_6 = 2$  and a new stable model is searched. Say that  $I = \{b, c, aux_4\}$  is returned, again with cost 3. Weak constraint  $r_5$  is relaxed. It is removed from  $\Pi_S$  and  $\Pi_R$  is extended with  $r'_5$ , i.e.,  $\leftarrow b$ , *not*  $aux_5$ .  $R$  is extended with  $aux_5 = 1$  and a new stable model is searched, but the program is now incoherent. The algorithm thus terminates by returning 3, i.e., the optimum cost for the original program.  $\triangleleft$

## 5.4 Algorithm OLL

The core-guided algorithm OLL [33] is reported in Algorithm 9. OLL is conceived for unweighted ASP optimization problems, i.e., for programs in which all weak constraints have the same weight. However, there are several possibilities for using the algorithm in case of weighted ASP optimization problems

[33]. One strategy considers the replacement of each weak constraint  $r$  by  $weight(r)$  copies of  $r$  of weight 1. We consider a procedure described in [33] that is similar to the one described in [60, 61].

The algorithm runs by considering all weak constraints of the input program  $\Pi = (\Pi_R, \Pi_W)$  as hard constraints and searches for a stable model. If none is found then some weak constraints are relaxed, which means that they can possibly be violated during the search for a stable model. This process is iterated until a stable model is found. The algorithm uses a set  $\Pi_S$  for storing all weak constraints of  $\Pi$  that are not relaxed, so that any weak constraint is relaxed at most once. Initially,  $\Pi_S$  is equal to  $\Pi_W$ . The algorithm also uses a set  $\Pi_{aggr}$  of constraints created by the algorithm, which is initially empty and will store constraints consisting of a unique aggregate atom. If a stable model  $I$  for  $\Pi_R \cup \Pi_{aggr}$  is found then  $I$  is also an optimum solution of  $\Pi$ . Otherwise, an unsatisfiable core  $\Pi_{core}$  is computed and used for relaxing  $\Pi_R$ . In more detail, constraints in  $\Pi_{aggr} \cap \Pi_{core}$  are moved into  $\Pi_S$  and replaced by copies with increased lower bounds, unless the copies are trivially satisfied. The minimum weight  $minWeight$  of the weak constraints in the core is then computed (line 8). Then for each weak constraint  $r$  in  $\Pi_S \cap \Pi_{core}$  with a weight greater than the  $minWeight$  a copy of  $r$  is created. The weight of the new weak constraint is equal to the weight of  $r$  minus the  $minWeight$  and the new weak constraint is added to  $\Pi_S$ . The weak constraint  $r$  is then considered as unweighted and relaxed by procedure *RelaxWeakConstraint*. Finally, an aggregate containing the new relaxing atoms and unitary weights is added to  $\Pi_{aggr}$  (line 15).

**Example 17.** Consider the program in Example 14. Initially,  $\Pi_S$  contains all weak constraints, i.e.,  $r_3-r_6$ . Program  $\Pi_R = \{r_1, \dots, r_6\}$  is incoherent, and thus an unsatisfiable core  $\Pi_{core}$  is computed, say  $\{r_3, r_4, r_5, r_6\}$ .<sup>1</sup> The minimum cost  $minWeight$  of the weak constraints in the core is 1. A copy of weak constraints  $r_4$  and  $r_6$  is then created, and the new weak constraints are the following:

$$r_7 : \leftarrow_1 c \qquad r_8 : \leftarrow_1 d.$$

The new weak constraints  $r_7$  and  $r_8$  are added to  $\Pi_S$ . All weak constraints in  $\Pi_{core}$  are thus relaxed, i.e., they are replaced by the following constraints:

$$\begin{array}{ll} r'_3 : \leftarrow a, not\ aux_3 & r'_5 : \leftarrow b, not\ aux_5 \\ r'_4 : \leftarrow c, not\ aux_4 & r'_6 : \leftarrow d, not\ aux_6. \end{array}$$

The subsequent coherence test must also satisfy a constraint  $r_{aggr}$  of the form  $\leftarrow \{aux_3 = 1, aux_4 = 1, aux_5 = 1, aux_6 = 1\} \geq 2$ . The processed program is

---

<sup>1</sup>Note that the core is not minimal since  $\{r_3, r_5\}$  and  $\{r_4, r_6\}$  are also cores.

incoherent and an unsatisfiable core  $\{r_7, r_8\}$  is returned. The minimum cost of the weak constraints in the core is 1. Weak constraints  $r_7$  and  $r_8$  are thus relaxed, i.e., they are replaced by the following constraints:

$$r'_7 : \leftarrow c, \text{not } aux_7 \quad r_8 : \leftarrow d, \text{not } aux_8$$

where  $aux_7$  and  $aux_8$  are fresh atoms. The subsequent coherence test must also satisfy a constraint  $r_{aggr}$  of the form  $\leftarrow \{aux_7 = 1, aux_8 = 1\} \geq 2$ . The program is still incoherent and an unsatisfiable core  $\{r_{aggr}, r'_3, \dots, r'_6\}$  is returned. Constraint  $r_{aggr}$  is thus added to  $\Pi_S$  in order to be relaxed, and its bound is increased by 1 in the subsequent coherence check. The relaxed version of  $r_{aggr}$  is  $\leftarrow \{aux_3 = 1, aux_4 = 1, aux_5 = 1, aux_6 = 1\} \geq 2, \text{not } aux_{aggr}$ , where  $aux_{aggr}$  is a fresh atom. (Actually, there is yet another trivial constraint, namely  $\leftarrow \{aux_{aggr} = 1\} \geq 2$ .) The processed program is now coherent and a stable model is computed, say  $\{a, c, aux_3, aux_4, aux_7\}$  of cost 3, which is also optimum.  $\triangleleft$

## 5.5 Algorithm PMRES

Algorithm 10 is based on partial MaxSAT resolution [62], an extension of resolution taking into account weighted clauses. It was first introduced for MaxSAT in the EVA solver [35]. PMRES starts by considering all weak constraints of the input program  $\Pi$  as hard constraints and searches for a stable model (line 3). If none is found then fresh atoms are introduced for each weak constraint in the computed core (line 6) in order to relax them (line 11). The lower bound, which is initially set to zero (line 2), is increased of the minimum weight of the relaxed weak constraints (lines 7–8), and their weights are decreased of the same value (line 12).

At this point, PMRES adds  $n-1$  weak constraints, where the  $i$ -th weak constraint is violated if the formula  $aux_i \vee (aux_{i+1} \wedge \dots \wedge aux_n)$  is false (lines 13–15). As a further optimization, the algorithm introduces fresh atoms  $c_{i+1}$ , for each  $i \in [1..n-2]$ , for modeling the conjunctions  $aux_{i+1} \wedge \dots \wedge aux_n$ .

This process is iterated until a stable model is found, and finally the lower bound, i.e., the optimum cost for  $\Pi$ , is returned.

**Example 18.** Consider the program in Example 14. Initially, weak constraints are considered hard and an unsatisfiable core  $\Pi_{core}$  is returned, say  $\{r_3, r_4, r_5, r_6\}$ , and the lower bound is increased to 1. All weak constraints are thus relaxed, i.e., they are replaced by the following constraints:

$$\begin{aligned} r'_3 : & \leftarrow_1 a, \sim aux_3 & r'_5 : & \leftarrow_1 b, \sim aux_5 \\ r'_4 : & \leftarrow_2 c, \sim aux_4 & r'_6 : & \leftarrow_2 d, \sim aux_6 \end{aligned}$$

---

**Algorithm 10:** PMRES

---

**Input** : A program  $\Pi = (\Pi_R, \Pi_W)$   
**Output**: The optimum cost for  $\Pi$

```
1 begin
2    $(\Pi_S, lower\_bound) := (\Pi_W, 0);$ 
3    $(res, \Pi_{core}, I) := ComputeAnswerSet_{cores}(\Pi, \emptyset);$ 
4   if  $res = Coherent$  then return  $lower\_bound;$ 
5   Let  $\Pi_S \cap \Pi_{core} = \{r_1, \dots, r_n\}$ , where  $n \geq 1;$ 
6   Let  $aux_1, \dots, aux_n, c_2, \dots, c_{n-1}$  be fresh atoms;
7    $m := \min\{weight(r_1), \dots, weight(r_n)\};$ 
8    $lower\_bound := lower\_bound + m;$ 
9   for  $i = 1$  to  $n$  do
10     $\Pi_S := \Pi_S \setminus \{r_i\};$ 
11     $\Pi_R := (\Pi_R \setminus \{r_i\}) \cup \{\leftarrow B(r_i), \sim aux_i\};$ 
12    if  $weight(r_i) > m$  then  $\Pi_S := \Pi_S \cup \{\leftarrow_{weight(r_i)-m} B(r_i)\};$ 
13    if  $i \in [1..n - 2]$  then  $next := c_{i+1};$  else  $next := aux_n;$ 
14    if  $i \in [1..n - 1]$  then  $\Pi_S := \Pi_S \cup \{\leftarrow_m \sim aux_i, \sim next\};$ 
15    if  $i \in [2..n - 1]$  then  $\Pi_R := \Pi_R \cup \{c_i \leftarrow aux_i, next\};$ 
16  goto 3;
```

---

However, two of them, namely  $r_4$  and  $r_6$ , are put back in the program with weight 1. Moreover, fresh atoms  $c_4, c_6$ , and the following rules, are introduced:

$$\begin{aligned} r_7 : \quad & \leftarrow_1 \sim aux_3, \sim c_3 & r_{c_3} : \quad & c_3 \leftarrow aux_5, c_4 \\ r_8 : \quad & \leftarrow_1 \sim aux_5, \sim c_4 & r_{c_4} : \quad & c_4 \leftarrow aux_4, aux_6 \\ r_9 : \quad & \leftarrow_1 \sim aux_4, \sim aux_6 \end{aligned}$$

The processed program is still incoherent and an unsatisfiable core, say  $\{r_4, r_6, r_9\}$ , is returned. Constraint  $r_{11}$  is thus relaxed and replaced by

$$r'_9 : \leftarrow \sim aux_4, \sim aux_6, \sim aux_7.$$

(Note that, according to Algorithm 10,  $r_4$  and  $r_6$  should be relaxed again. However, this is not really required for computing the optimum cost of the input program, and the algorithm can be easily modified to avoid such a redundant operation.) Weak constraints  $r_4, r_6, r_9$  are removed because their weights are equal, but a new fresh atom  $c_9$  and the following rules are added:

$$\begin{aligned} r_{10} : \quad & \leftarrow_1 \sim aux_4, \sim c_9 & r_{c_9} : \quad & c_9 \leftarrow aux_9, aux_6 \\ r_{11} : \quad & \leftarrow_1 \sim aux_9, \sim aux_6 \end{aligned}$$

The lower bound is 2 at this point.

The program is still incoherent. For example,  $\{r'_3, r'_5, r_7, r_8\}$  is an unsatisfiable core. Hence,  $r_7$  and  $r_8$  are relaxed and replaced by the following rules:

$$\begin{aligned} r'_8 &: \leftarrow \sim aux_5, \sim c_4, \sim aux_8 & r'_9 &: \leftarrow \sim aux_4, \sim aux_6, \sim aux_9 \\ r_{12} &: \leftarrow_1 \sim aux_8, \sim aux_9 \end{aligned}$$

where  $aux_8$  and  $aux_9$  are fresh atoms. The lower bound is increased to 3.

The processed program is now coherent and a stable model is computed, say  $\{a, c, aux_3, aux_4, aux_8, aux_9\}$ . The optimum cost 3 is finally returned.  $\triangleleft$

## 5.6 Algorithm BCD

Algorithm 11 is called core-guided binary search with disjoint cores, in short BCD [36] and implements a binary search of the optimal solution. In a nutshell, all weak constraints are initially considered as hard constraints and a stable model is searched. If the processed program is incoherent then an unsatisfiable core is computed and stored in a set *Cores*, which is initially empty. Weak constraints in the computed core are relaxed and the new relaxing atoms are used to build a constraint comprising a single aggregate atom aimed at performing a binary search on the subsequent coherence tests. In fact, each unsatisfiable core is associated with a lower and an upper bound, which are updated during the computation. More in detail, whenever an incoherent program is processed, the new unsatisfiable core  $\Pi_{core}$  is merged with each element in *Cores* intersecting  $\Pi_{core}$ . In this way *Cores* is guaranteed to contain pairwise disjoint unsatisfiable cores, which actually represent disjoint subproblems. When no weak constraint needs to be relaxed, and  $\Pi_{core}$  intersects only one previously computed core  $C$ , the lower bound of  $C$  is increased because the subproblem associated with  $C$  has no solution of cost smaller than  $(C.lb + C.ub)/2$ . In fact, such a subproblem is represented by a constraint added at line 7. Hence, the new lower bound of  $C$  will force the algorithm to search for a solution of higher cost, actually resulting in a binary search of the optimum cost. When instead a stable model  $I$  is found,  $OPT$  as well as the upper bounds of the computed cores are updated according to  $I$ . This process is repeated until *Cores* contains unsolved subproblems.

**Example 19.** Consider the program in Example 14. Initially,  $\Pi_S$  contains all weak constraints, i.e.,  $r_3$ – $r_6$ , while *Cores* is empty (line 2). Program  $\Pi_R = \{r_1, \dots, r_6\}$  is incoherent, and thus an unsatisfiable core  $\Pi_{core}$  is computed, say  $\{r_3, r_5\}$ . Weak constraints  $r_3$  and  $r_5$  are thus relaxed (line 22), i.e., they are replaced by the following constraints:

$$r'_3 : \leftarrow a, \sim aux_3 \quad r'_5 : \leftarrow b, \sim aux_5$$

---

**Algorithm 11:** BCD

---

**Input** : A program  $\Pi = (\Pi_R, \Pi_W)$   
**Output**: The optimum cost for  $\Pi$

```
1 begin
2    $(\Pi_S, Cores, OPT) := (\Pi_W, \emptyset, 0)$ ;
3   repeat
4      $\Pi_{aggr} := \emptyset$ ;
5     foreach  $C \in Cores$  do
6       if  $C.lb + 1 = C.ub$  then  $C.mb := C.ub$ ; else
7          $C.mb := \lfloor \frac{C.ub + C.lb}{2} \rfloor$ ;
8          $\Pi_{aggr} := \Pi_{aggr} \cup \{ \leftarrow C.R \geq C.mb + 1 \}$ ;
9      $(res, \Pi_{core}, I) := ComputeAnswerSet_{cores}(\Pi_R \cup \Pi_{aggr}, \emptyset)$ ;
10    if  $res \neq Incoherent$  then
11       $OPT := cost(\Pi_W, I)$ ;
12      foreach  $C \in Cores$  do
13         $C.ub := \sum_{aux=w \in C.R \wedge I \models aux} w$ ;
14    else
15       $SubCores := \{ C \in Cores \mid C.core \cap \Pi_{core} \neq \emptyset \}$ ;
16      if  $\Pi_{core} \cap \Pi_S = \emptyset$  and  $|SubCores| = 1$  then
17        Let  $SubCores = \{C\}$ ;
18         $C.lb := C.mb$ ;
19      else
20        Let  $C$  be a new structure;
21         $(C.core, C.R) := (\emptyset, \emptyset)$ ;
22        foreach  $r \in \Pi_{core} \cap \Pi_S$  do
23           $r' := RelaxWeakConstraint(r, C.R)$ ;
24           $\Pi_S := \Pi_S \setminus \{r\}$ ;
25           $\Pi_R := (\Pi_R \setminus \{r\}) \cup \{r'\}$ ;
26           $C.core := C.core \cup \{r'\}$ ;
27         $(C.lb, C.ub) := (0, 1 + \sum_{aux=w \in C.R} w)$ ;
28        foreach  $C' \in SubCores$  do
29           $(C.core, C.R) := (C.core \cup C'.core, C.R \cup C'.R)$ ;
30           $(C.lb, C.ub) := (C.lb + C'.lb, C.ub + C'.ub)$ ;
31         $Cores := (Cores \setminus SubCores) \cup \{C\}$ ;
32  until  $\forall C \in Cores \ C.lb + 1 \geq C.ub$ ;
33  return  $OPT$ ;
```

---

where  $aux_3$  and  $aux_5$  are fresh atoms. Rules  $r'_3$  and  $r'_5$  are stored in a new structure  $C_1$  in *Cores*, whose lower and upper bounds are initially set to 0 and 3 (line 26). The subsequent coherence test must also satisfy a constraint obtained from  $C_1$ , that is,  $\leftarrow \{aux_3 = 1, aux_5 = 1\} \geq 2$ . However, the processed program is still incoherent and an unsatisfiable core  $\{r_4, r_6\}$  is returned. Weak constraints  $r_4$  and  $r_6$  are thus relaxed, i.e., they are replaced by the following constraints:

$$r'_4 : \leftarrow c, \sim aux_4 \quad r'_6 : \leftarrow d, \sim aux_6$$

where  $aux_4$  and  $aux_6$  are fresh atoms. Rules  $r'_4$  and  $r'_6$  are stored in a new structure  $C_2$  in *Cores*, whose lower and upper bounds are initially set to 0 and 5, so that the next coherence check must also satisfy  $\leftarrow \{aux_4 = 2, aux_6 = 2\} \geq 3$ . Actually, a stable model is found, say  $\{a, c, aux_3, aux_4\}$ , which means that the current optimal solution has cost 3. Upper bounds of  $C_1$  and  $C_2$  are updated to 1 and 2, respectively. The next coherence check must thus satisfy  $r_1, r_2, r'_3, \dots, r'_6$ , and the additional constraints  $r_{C_1} : \leftarrow \{aux_3 = 1, aux_5 = 1\} \geq 2$  and  $r_{C_2} : \leftarrow \{aux_4 = 2, aux_6 = 2\} \geq 2$ . However, the unsatisfiable core  $\{r'_4, r'_6, r_{C_2}\}$  is returned. In this case the lower bound of  $C_2$  is set to 1 and the algorithm terminates returning 3, i.e., the optimal cost. In fact, the lower and the upper bounds of  $C_1$  are 0 and 1, respectively, and the lower and the upper bounds of  $C_2$  are 1 and 2, respectively. Hence, for each structure in *Cores* the lower bound + 1 is greater than the upper bound (line 31).  $\triangleleft$

## 5.7 Implementation

In the following, we report a few technical details and design choices that enlighten on the implementations of optimal stable model search in WASP 2. Source codes of the implemented algorithms can be downloaded from the branch *optimization* of the repository <https://github.com/alviano/wasp.git>. The incremental interface of WASP 2 is a key component for the implementation of algorithms for optimal stable model search. Indeed, at the beginning of the computation, each weak constraint is considered as hard and one *relaxation literal* is added in its body. After that, the constraints to be considered during each call to the ASP solver are easily selected (depending on the search algorithm of choice) by means of assumptions on the relaxation literals. For example, core guided algorithms start by considering all weak constraints as hard by simply assuming all relaxation literals to be false.

An additional technique implemented in WASP 2 that improves efficiency in weighted problems is called *stratification* [63]. The idea is to start by



considering weak constraints of maximum weight, say  $w$ , and by ignoring the remaining weak constraints. If the resulting program is incoherent, all weak constraints in the detected unsatisfiable core have weight  $w$  (or greater), and therefore the lower bound can be increased significantly. This process is iterated until a stable model is found, which is the case when the processed program has no unsatisfiable core only involving weak constraints of weight  $w$  (or greater). Hence, the maximum weight among the weak constraints of weight smaller than  $w$  is computed, say  $w'$ , and all weak constraints of weight  $w'$  or greater are considered in the next search. This process terminates when all weak constraints are considered and a stable model is found.

## 5.8 Experiments

In this section we report the results of an experiment assessing the performance of WASP 2 implementing the algorithms described in this chapter. We also include in the comparison the ASP solver CLASP. All the solvers used gringo 3.0.5 [47] as grounder. Concerning CLASP we used the version 3.0.1.

**Hardware Setting.** The experiments were run on a four core Intel Xeon CPU X3430 2.4 GHz, with 4 GB of physical RAM and running Debian Linux 7.3 (kernel ver. 3.2.0-4-amd64). Only one core was enabled, and time and memory limits were set to 600 seconds and 3 GB, respectively. Performance was measured using the tools pyrunlim and pyrunner (<https://github.com/alviano/python>).

**Compared methods.** We implemented several algorithms for optimal stable model search in WASP 2, so to minimize the influence of the result returned by function `ComputeAnswerSetcores` (all algorithms will use the same ASP solver). In particular, in our comparison, we considered the following variants of WASP 2:

- OPT: The solver WASP 2 employing Algorithm 7.
- BASIC: The solver WASP 2 employing the BASIC algorithm.
- MGD: The solver WASP 2 employing Algorithm 8.
- OLL: The solver WASP 2 employing Algorithm 9.
- PMRES: The solver WASP 2 employing Algorithm 10.
- BCD: The solver WASP 2 employing Algorithm 11.

We also considered, as reference of the state of the art, two versions of the solver that won the fifth ASP Competition in the optimization track (T3), namely:

- CLASP<sub>BASIC</sub>: The solver CLASP 3.0.1 in its default configuration, which runs algorithm BASIC.
- CLASP<sub>OLL</sub>: The solver CLASP 3.0.1 running algorithm OLL.

**Benchmark Setting.** The experiment was conducted on benchmarks that were already used in the literature for the comparative evaluation of ASP solvers [64, 23] and MaxSAT solvers (<http://www.maxsat.udl.cat/>). In particular, we considered:

- All benchmarks evaluated in Track 3 (Optimization) of the 5th ASP Competition [65], namely: *Crossing Minimization*, *Maximal Clique*, *Still Life*, and *Valves Location*.
- Benchmarks used in [33] namely: *Labyrinth*, *Minimum Postage Stamp Problem*, *Sokoban*, *Weight Bounded Dominating Set Suite*, *Fastfood* and *Open Doors*. We downloaded all the instances available for these problems in the Asparagus repository.
- All the industrial instances from the family *WCSP.spot5* of the MaxSAT Competition 2014 (<http://maxsat.ia.udl.cat/>).

MaxSAT instances were translated from the original formulation in Extended DIMACS CNF format to the lparse numeric format in the straightforward way. All instances used in our experiments are publicly available, and can be downloaded, together with detailed descriptions of benchmark problems, from the following web sites: the 5th ASP competition (<https://www.mat.unical.it/aspcomp2014>), Asparagus (<http://asparagus.cs.uni-potsdam.de>), and the 9th MaxSAT competition (<http://maxsat.ia.udl.cat/>).

Instances have been classified in *unweighted* and *weighted* depending on whether all constraints have associated the same costs or not, respectively. In particular, the weighted set comprises *Fastfood*, *Open Doors*, *Valves Location*, *WCSP.spot5.dir* and *WCSP.spot5.log*; the remaining benchmarks are unweighted.

**Results overview.** We first analyze the performance of the considered algorithms for optimal stable model search in terms of the number of optimal

Table 5.1: Number of solved instances and average running time in seconds

Problem	#	OPT		BASIC		MGD		OLL		PMRES		BCD	
		sol.	time	sol.	time	sol.	time	sol.	time	sol.	time	sol.	time
unweighted	CrossingMinim	30	1 142.43	7 249.71	8 248.53	23 52.47	23 100.08	10 83.99					
	Labyrinth	29	3 117.95	4 162.43	1 482.29	6 58.85	4 13.52	3 18.31					
	MaximalClique	30	0 -	2 549.81	1 452.16	30 31.31	7 278.90	0 -					
	MPSP	6	4 34.49	4 57.07	4 39.36	5 12.35	5 11.54	3 29.98					
	Sokoban	28	28 2.60	28 1.70	28 1.83	28 4.24	28 4.24	28 2.62					
	StillLife	10	4 147.64	3 1.57	3 1.13	5 41.25	5 62.83	3 2.44					
	WBDSS	29	0 -	0 -	0 -	9 65.77	9 49.75	0 -					
weighted	Fastfood	29	20 67.73	26 98.84	26 118.13	17 86.18	18 51.36	19 116.64					
	OpenDoors	31	31 13.66	31 11.69	31 12.64	31 15.26	31 16.96	31 14.04					
	ValvesLocation	30	2 10.31	4 14.62	4 14.08	4 44.33	4 64.23	4 33.43					
	WCSP.spot5.dir	21	3 66.76	3 21.03	3 32.51	15 7.31	14 13.47	6 16.16					
	WCSP.spot5.log	21	3 52.45	3 107.31	3 59.14	14 31.70	13 15.79	5 0.11					
	<b>Total unweighted</b>	<b>162</b>	<b>40 32.44</b>	<b>48 78.71</b>	<b>45 69.66</b>	<b>106 32.8</b>	<b>81 64.77</b>	<b>47 22.67</b>					
<b>Total weighted</b>	<b>132</b>	<b>59 36.55</b>	<b>67 50.39</b>	<b>67 56.63</b>	<b>81 32.95</b>	<b>80 26.26</b>	<b>65 44.35</b>						
<b>Total</b>	<b>294</b>	<b>99 34.89</b>	<b>115 62.21</b>	<b>112 61.87</b>	<b>187 32.87</b>	<b>161 45.64</b>	<b>112 35.25</b>						

solutions found within the timeout. This datum is reported for each benchmark problem in Table 5.1, together with the average running time on solved instances. Benchmarks are properly divided in weighed and unweigted, and total scores of each algorithm for the two categories, as well as total scores overall, are reported in the last three rows of the table.

As a general comment, we note that core-guided algorithms outperform model-guided alternatives in almost all benchmarks. The only exception is Fastfood, in which BASIC and MGD solve 26 instances, whereas core-guided algorithms solve at most 19 instances (BCD). As a possible explanation for this behavior, we observed that for instances of Fastfood the unsatisfiable cores computed by WASP 2 contain relatively many weak constraints (at least 80 weak constraints are involved in all unsatisfiable cores of several instances in this benchmark). On the other hand, this peculiarity cannot affect model-guided algorithms, as they approach the solution from a different direction.

Concerning model-guided algorithms, we observe that they behave similarly in almost all benchmarks, with BASIC leading the category with a total of 115 solved instances, only 3 more than MGD. The distance of OPT is instead higher, as it solves only 99 instances. As the difference with BASIC consists in constraining the branching heuristic with a set of preferred choices, we conclude that such an interference pays off only in few cases.

Among core-guided algorithms, the category is led by OLL, with 187 solved instances, 26 instances more than PMRES. Actually, after observing the similarities of Algorithms 9 and 10, the general behaviors of OLL and PMRES was expected to be similar. However, the native implementation of aggregates in WASP 2 sharpens an advantage of OLL with respect to PMRES, which is particularly evident in the MaximalClique domain, where the

Table 5.2: Number of solved instances and average running time in seconds

	Problem	#	OLL		BASIC		CLASPOLL		CLASPBASIC	
			sol.	time	sol.	time	sol.	time	sol.	time
unweighted	CrossingMinim	30	23	52.47	7	249.71	24	72.63	14	162.02
	Labyrinth	29	6	58.85	4	162.43	17	89.73	4	112.90
	MaximalClique	30	30	31.31	2	549.81	30	61.82	1	370.31
	MPSP	6	5	12.35	4	57.07	5	3.21	4	148.42
	Sokoban	28	28	4.24	28	1.70	28	1.04	28	0.92
	StillLife	10	5	41.25	3	1.57	6	75.70	4	32.60
	WBDSS	29	9	65.77	0	-	10	48.58	0	-
weighted	Fastfood	29	17	86.18	26	98.84	16	30.32	29	13.07
	OpenDoors	31	31	15.26	31	11.69	31	10.65	31	12.46
	ValvesLocation	30	4	44.33	4	14.62	2	12.67	4	9.84
	WCSP.spot5.dir	21	15	7.31	3	21.03	13	15.10	3	2.53
	WCSP.spot5.log	21	14	31.70	3	107.31	6	0.00	3	8.71
	<b>Total unweighted</b>	<b>162</b>	<b>106</b>	<b>32.81</b>	<b>48</b>	<b>78.71</b>	<b>120</b>	<b>50.90</b>	<b>55</b>	<b>69.82</b>
<b>Total weighted</b>	<b>132</b>	<b>81</b>	<b>32.95</b>	<b>67</b>	<b>50.39</b>	<b>68</b>	<b>15.25</b>	<b>70</b>	<b>11.98</b>	
<b>Total</b>	<b>294</b>	<b>187</b>	<b>32.87</b>	<b>115</b>	<b>62.21</b>	<b>188</b>	<b>38.01</b>	<b>125</b>	<b>37.43</b>	

difference of solved instances is 23. Including in this comparison the other core-guided algorithm, BCD, we note that the distance with OLL is 75 instances.

**Comparison with the state of the art.** We run the state of the art solver CLASP on the tested benchmarks, with the default configuration and explicitly selecting OLL. The number of solved instances is reported in Table 5.2, where we also repeat the result obtained by our implementation of OLL and BASIC to ease the comparison. As a first observation, the result confirms that OLL is the best option also for CLASP, with a difference of 63 solved instances. The two implementations of BASIC behave similarly, with CLASP solving 10 instances more than WASP 2. Comparing the two implementations of OLL, we observe that CLASP performs better in unweighted benchmarks, where it solves 14 instances more than WASP 2. On the other hand, the opposite happens for weighted benchmarks, where WASP 2 solves 13 instances more than CLASP. As an explanation for the unweighted case, we note that CLASP is much faster than WASP 2 in Labyrinth, where it appears to handle more efficiently the unfounded-free propagation (see Section 4.4.1). As for the weighted benchmarks, instead, we checked that the stratification technique described in Section 5.7 provides a boost of performance. The better performance of WASP 2 over CLASP is justified by this technique, which is not implemented in CLASP.

# Chapter 6

## Query Answering

An ASP knowledge base can be queried according to two possible modes of reasoning, usually referred to as brave (or credulous) and cautious (or skeptical). Brave reasoning provides answers to the input query that are witnessed by some stable model of the knowledge base. For cautious reasoning, instead, answers have to be witnessed by all stable models. Cautious reasoning over ASP knowledge bases has relevant applications in various fields ranging from Databases to Artificial Intelligence. Among them are consistent query answering [28], data integration [29], and ontology-based reasoning [30]. A common practice in ASP is to reduce query answering to the computation of a subset of the cautious consequences of a logic program [17], where cautious consequences are atoms belonging to all stable models. It is important to note that cautious reasoning is a resource demanding task, which is often not affordable to complete in reasonable time. It is interesting to observe that query answering is addressed differently in other logic programming languages. For example, Prolog queries having infinitely many answers are common due to the presence of uninterpreted function symbols. Prolog systems are thus designed to produce underestimates of the complete, possibly infinite solution, which actually represent sound answers to the input query.

In fact, underestimates are useful in practice, especially in the cases in which waiting for termination is not affordable, and this may be the case even if termination is guaranteed. It is thus natural to ask whether underestimates can be computed also in the context of ASP.

This chapter provides a description of the algorithms employed by ASP systems. Moreover, we have also adapted to ASP the *iterative consistency testing* algorithm for computing backbones of propositional theories [34]. An interesting aspect of the algorithms analyzed in this chapter is that underestimates are produced during the computation of the complete solution. The computation can thus be stopped either when a sufficient number of cautious

consequences have been produced, or when no new answer is produced after a specified amount of time. Such algorithms are referred to as anytime in the literature. Finally, we also discuss the results of an experimental analysis showing that a large percentage of the sound answers can be produced after few minutes of the computation.

## 6.1 Computation of Cautious Consequences

Several strategies for computing cautious consequences of a given program are reported in this section. Some of these strategies aim at solving the problem producing overestimates of the solution, which are improved and eventually result in the set of cautious consequences of the input program. Among them are the algorithms implemented by the ASP solvers DLV [17] and CLASP [20], respectively called *enumeration of models* and *overestimate reduction* in the following. However, another strategy can in addition produce sound answers during the computation of the complete solution, thus providing underestimates also when termination is not affordable in reasonable time. This strategy is *iterative coherence testing*, an adaptation of an algorithm computing backbones of propositional formulas [34]. Finally, a strategy for obtaining underestimates from enumeration of models and overestimate reduction is presented, which can also be used to improve the other algorithms.

In more detail, the algorithms considered here have a common skeleton, reported as Algorithm 12. They receive as input a program  $\Pi$  and a set of atoms  $Q$  representing answer candidates of a query, and produce as output either the largest subset of  $Q$  that only contains cautious consequences of  $\Pi$ , in case  $\Pi$  is coherent, or *Incoherent* otherwise. Initially, the underestimate  $U$  and the overestimate  $O$  are set to  $\emptyset$  and  $Q$ , respectively (line 1). A coherence test of  $\Pi$  is then performed (lines 2–4) by calling function `ComputeAnswerSetassum`, which actually is the incremental variant (see Section 2.4) of the stable model search as described in Section 4.4. The first argument of the function is a program  $\Pi$ . The second argument is an interpretation, which is initially empty. The third and the fourth arguments are the set of assumptions  $assum_{\wedge}$  and  $assum_{\vee}$ , described in Section 2.4. The function returns either  $I$  in case a stable model  $I$  of  $\Pi$  is found, or *Incoherent* otherwise. Note that *Incoherent* is returned not only when  $\Pi$  is incoherent, but in general when each stable model  $M$  of  $\Pi$  violates some assumptions. The first stable model found is used to improve the overestimate (line 5). At this point, estimates are improved according to different strategies until they are equal (line 6). The strategy implemented by `EnumerationOf-`

---

**Algorithm 12:** CautiousReasoning

---

**Input** : a program  $\Pi$  and a set of atoms  $Q$

**Output:** atoms in  $Q$  that are cautious consequences of  $\Pi$ , or *Incoherent*

```
1  $U := \emptyset$ ;  $O := Q$ ;  $I := \emptyset$ ;  $assum_{\wedge} := \emptyset$ ;  $assum_{\vee} := \emptyset$ ;  
2  $I := \text{ComputeAnswerSet}_{assum}(\Pi, I, assum_{\wedge}, assum_{\vee})$ ;  
3 if  $I = \text{Incoherent}$  then  
4 | return Incoherent;  
5  $O := O \cap I$ ;  
6 while  $U \neq O$  do  
7 | // EnumerationOfModels or other procedure  
8 return  $U$ ;
```

---

---

**Procedure** EnumerationOfModels

---

(A1)

```
1  $\Pi := \Pi \cup \text{Constraint}(I)$ ;  
2  $I := \text{ComputeAnswerSet}_{assum}(\Pi, I, \emptyset, \emptyset)$ ;  
3 if  $I = \text{Incoherent}$  then  
4 |  $U := O$ ;  
5 else  
6 |  $O := O \cap I$ ;
```

---

---

**Procedure** OverestimateReduction

---

(A2)

```
1  $\Pi := \Pi \cup \text{Constraint}(O)$ ;  
2  $I := \text{ComputeAnswerSet}_{assum}(\Pi, I, \emptyset, \emptyset)$ ;  
3 if  $I = \text{Incoherent}$  then  
4 |  $U := O$ ;  
5 else  
6 |  $\Pi := \Pi \setminus \text{Constraint}(O)$ ;  
7 |  $O := O \cap I$ ;
```

---

---

**Procedure** IterativeCoherenceTesting

---

(A3)

```
1  $a := \text{OneOf}(O \setminus U)$ ;  
2  $I := \text{ComputeAnswerSet}_{assum}(\Pi, I, \{\sim a\}, \emptyset)$ ;  
3 if  $I = \text{Incoherent}$  then  
4 |  $U := U \cup \{a\}$ ;  
5 else  
6 |  $O := O \cap I$ ;
```

---

Models adds to  $\Pi$  a constraint that eliminates the last stable model found (line 1). Function  $\text{Constraint}(\{a_1, \dots, a_n\})$  in fact returns a singleton of the form  $\{\leftarrow a_1, \dots, a_n\}$ . The algorithm then searches for a new stable model

(line 2) to improve the overestimate (line 6). If no new stable model exists, the underestimate is set equal to the overestimate (lines 3–4), thus terminating the computation. OverestimateReduction is similar, but the constraint added is obtained from the current overestimate (line 1). In this way, when a new stable model is found, an improvement of the overestimate is guaranteed, and the constraint can thus be reduced accordingly (lines 6 and 1).

The strategy implemented by IterativeCoherenceTesting can also improve the underestimate many times during its computation. In fact, one cautious consequence candidate is selected by calling function OneOf (line 1). A stable model is searched (line 2) by assuming this candidate as false. If none is found then the underestimate can be increased (lines 3–4). Otherwise, the overestimate can be improved (lines 5–6).

Variants of these algorithms can be obtained by modifying the function ComputeAnswerSet<sub>assum</sub> with another function which actually implements stable model search, but also improves the current underestimate after each restart. This function is referred to as ComputeAnswerSet<sub>assum</sub>\*.

## 6.2 Correctness

In this section we show the correctness of the algorithms presented in the previous section.

**Theorem 2.** Let  $\Pi$  be a program and  $Q \subseteq atoms(\Pi)$  a set of atoms. CautiousReasoning( $\Pi, Q$ ) terminates after finitely many steps and returns  $Q \cap CC(\Pi)$  if  $\Pi$  is coherent; otherwise, it returns *Incoherent*. Moreover,  $U \subseteq Q \cap CC(\Pi) \subseteq O$  holds at each step of computation. The claim holds for all variants of Algorithm 12.

The proof is split into several lemmas using  $\Pi_i, L_i, U_i, O_i, I_i$  to denote the content of variables  $\Pi, L, U, O, I$  at step  $i$  of computation ( $i \geq 0$ ), where  $L$  is the set containing the learned constraints. More in detail, in Lemma 1 we will first show that underestimates form an increasing sequence and, on the contrary, overestimates form a decreasing sequence. Then, in Lemma 2 we will prove properties of stable models of programs  $\Pi_i \cup L_i$  ( $i \geq 0$ ). Correctness of estimates will be shown in Lemmas 3–4, and termination of the algorithms in Lemma 5. Finally, in Lemma 6 we will extend the proof to variants using ComputeAnswerSet<sub>assum</sub>\*.

**Lemma 1.**  $U_i \subseteq U_{i+1}$  and  $O_{i+1} \subseteq O_i \subseteq Q$  for each  $i \geq 0$ .

*Proof.* Variable  $U$  is initially empty. EnumerationOfModels and OverestimateReduction reassign  $U$  only once. IterativeCoherenceTesting always enlarges the set stored in  $U$  by means of set union (line 4). Concerning variable



$O$ , it is initially equal to  $Q$  and restricted at each reassignment by means of set intersection (line 7 for OverestimateReduction; line 6 for the other procedures).  $\square$

**Lemma 2.**  $SM(\Pi_{i+1} \cup L_{i+1}) \subseteq SM(\Pi_i \cup L_i)$  for each  $i \geq 0$ . For IterativeCoherenceTesting we also have  $SM(\Pi_{i+1} \cup L_{i+1}) = SM(\Pi_i \cup L_i)$  for each  $i \geq 0$ .

*Proof.* Variable  $\Pi$  is reassigned only by EnumerationOfModels and OverestimateReduction, where constraints are added to the previous program. Constraints can only remove stable models (as a consequence of the Splitting Set Theorem by [66]). On the other hand, learned constraints stored in variable  $L$  are implicit in the program stored by variable  $\Pi$ , and thus cannot change its semantics.  $\square$

**Lemma 3.**  $O_i \supseteq Q \cap CC(\Pi)$  for each  $i \geq 0$ .

*Proof.* The base case is true because  $O_0 = Q$ . Assume the claim is true for some  $i \geq 0$  and consider  $O_{i+1} = O_i \cap I_{i+1}$ , where  $I_{i+1} \in SM(\Pi_i \cup L_i)$ . By applications of Lemma 2, we obtain  $I_{i+1} \in SM(\Pi_0 \cup L_0)$ , i.e.,  $I_{i+1} \in SM(\Pi)$ . We can thus conclude  $a \in O_i \setminus O_{i+1}$  implies  $a \notin CC(\Pi)$ , and we are done.  $\square$

**Lemma 4.**  $U_i \subseteq Q \cap CC(\Pi)$  for each  $i \geq 0$ .

*Proof.* The base case is true because  $U_0 = \emptyset$ . Assume the claim is true for some  $i \geq 0$  and consider  $U_{i+1}$ . If  $U_{i+1} = U_i$  then the claim is true. Otherwise, we distinguish two cases.

For IterativeCoherenceTesting,  $U_{i+1} = U_i \cup \{a\}$  for some  $a \in O_i \setminus U_i$ . Moreover, there is no  $M \in SM(\Pi_i \cup L_i)$  such that  $a \notin M$  because  $I_{i+1}$  is *Incoherent*. From Lemma 2, we can conclude that there is no  $M \in SM(\Pi)$  such that  $a \notin M$ , i.e.,  $a \in CC(\Pi)$ . Since  $a \in O_i \setminus U_i$ , we have  $a \in O_i$  and thus  $a \in Q$  by Lemma 1. Therefore,  $a \in Q \cap CC(\Pi)$  and we are done.

For EnumerationOfModels and OverestimateReduction,  $U_{i+1} = O_i$  and the algorithm terminates. Exactly  $i + 1$  constraints were added to  $\Pi$ , one for each stable model of  $\Pi$  found, i.e.,  $I_1, \dots, I_i$ . Moreover,  $I_{i+1}$  is *Incoherent*. Assume by contradiction that there is  $a \in O_i \setminus CC(\Pi)$ . Hence, there is  $M \in SM(\Pi)$  such that  $a \notin M$ . Moreover,  $a \in I_j$  ( $j = 1, \dots, i$ ) and thus  $M$  is a model of all constraints added at line 1. Consequently,  $M$  is a stable model of  $\Pi_i \cup L_i$ , which contradicts  $I_{i+1}$  is *Incoherent*.  $\square$

**Lemma 5.** Algorithm 12 terminates after finitely many steps.

*Proof.* When EnumerationOfModels is used, termination is guaranteed because  $\Pi$  has a finite number of stable models. OverestimateReduction either sets  $U$  equal to  $O$ , or reduces  $O$ , which initially is equal to  $Q$ , a finite set. IterativeCoherenceTesting either increases  $U$ , or reduces  $O$ , and thus terminates because  $O$  is finite and  $U_i \subseteq O_i$  holds for each  $i \geq 0$  by Lemmas 3 and 4.  $\square$

**Lemma 6.** Underestimates produced by ComputeAnswerSet<sub>assum</sub>\* are sound.

*Proof.* Follows by the fact that  $L$  contains constraints that are implicit in the program stored by variable  $\Pi$ .  $\square$

## 6.3 Experiments

We implemented the algorithms introduced in the previous section in order to analyze their performances. Details on the implementation, on the tested benchmarks, and on the obtained results are reported in this section.

### 6.3.1 Implementation

The algorithms described in Section 6.1 are implemented in WASP 2. In the following,  $A2$  and  $A3$  will denote WASP 2 running Algorithm 12 with procedures OverestimateReduction and IterativeCoherenceTesting, respectively.  $A2^*$  and  $A3^*$  will instead denote the variants using the function ComputeAnswerSet<sub>assum</sub>\*. Procedure EnumerationOfModels is not considered in the analysis since it is significantly outperformed by the other strategies in general.

### 6.3.2 Benchmark Settings

We compared the implemented algorithms on three benchmarks, corresponding to different applications of cautious reasoning, briefly described below.

*Multi-Context Systems Querying (MCS).* Multi-context systems [67] are a formalism for interlinking heterogeneous knowledge bases, called contexts, using bridge rules that model the flow of information among contexts. Testcases in this benchmark are roughly those of the third ASP competition [22], where each context is modeled by a normal logic program under the stable model semantics. We actually made the testcases harder by requiring the computation of all pairs of the form  $\langle c, a \rangle$  such that atom  $a$  is true in context  $c$ , while in the original testcases a single pair of that form was involved in the query. The benchmark contains 53 of the 73 instances submitted to the third

ASP competition. We in fact excluded instances corresponding to incoherent theories, which are solved by the first coherence test in around 6 seconds on average, and always in less than 14 seconds.

*Consistent Query Answering (CQA)* is a well-known application of ASP [28, 50] that we briefly described in Section 3.4. We considered the benchmark proposed by [68], and in particular query  $Q3$  encoded according to the rewritings by [50]. The benchmark contains 13 randomly-generated databases of increasing size ranging from 1000 to 7000 tuples per relation. Each generated relation contains around 30% of primary key violations.

*SAT Backbones (SBB)*. The backbone of a propositional formula  $\varphi$  is the set of literals that are true in all models of  $\varphi$ . When  $\varphi$  is a set of clauses over variables  $v_1, \dots, v_n$  ( $n \geq 1$ ), satisfiability of  $\varphi$  can be modeled in ASP by rules  $t_i \leftarrow \sim f_i$  and  $f_i \leftarrow \sim t_i$  ( $i = 1, \dots, n$ ), and introducing a constraint for each clause in  $\varphi$ . Backbone computation thus corresponds to the computation of cautious consequences of an ASP program. The benchmark contains 20 industrial instances used in the SAT Challenge 2012 [69].

The experiment was run on a Mac Pro equipped with two 3 GHz Intel Xeon X5365 (quad core) processors, with 4 MB of L2 cache and 16 GB of RAM, running Debian Linux 7.3 (kernel ver. 3.2.0-4-amd64). Binaries were generated with the GNU C++ compiler 4.7.3-4 shipped by Debian. Time and memory limits were set to 600 seconds and 8 GB, respectively. Performance was measured using the tool RunLim (<http://fmv.jku.at/runlim/>). All instances were grounded by gringo 3.0.5 [47], whose execution time is not included in our analysis. We however report that the grounding time was often less than 1 second, with a peak of around 5 seconds for the largest 10 instances of MCS.

### 6.3.3 Discussion of the Results

The performance of the algorithms for computing cautious consequences introduced in Section 6.1 can be studied from several perspectives. On the one hand, we want to know which solution performs better and in which cases. On the other hand, we are interested in analyzing the rate at which each algorithm produces sound answers.

**Overall performance.** Table 6.1 summarizes the number of solved instances and average running times. In particular, the first column reports the total number of instances (#); the remaining columns report the number of solved instances within the time-out (solved) and the average running times on solved instances (avg time).  $A3^*$  outperforms  $A2^*$  in MCS, and is

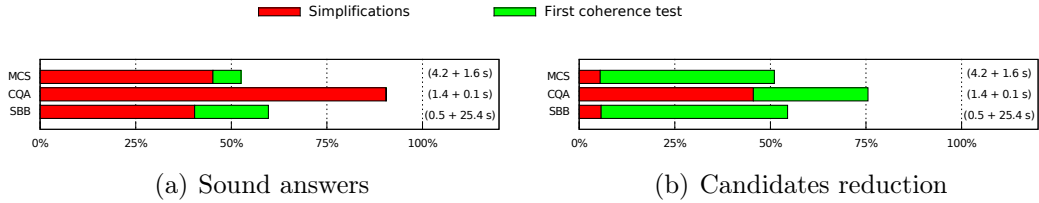


Figure 6.1: Sound answers and candidates reduction from simplifications and from first coherence test

faster also in CQA. On the other hand,  $A2^*$  performs well in SBB, solving one instance more than  $A3^*$ , and being faster on average. Note that, as expected, if one considers both the number of solved instances and running time,  $A2$ ,  $A3$  perform as  $A2^*$ ,  $A3^*$ , respectively.

**Detailed Analysis.** An important feature of the algorithms analyzed in this chapter is the ability to produce both sound answers and overestimates during the computation. Figure 6.1 reports, for each benchmark, the average percentage of (a) sound answers produced and (b) candidates reduction within the initial steps of the computation. In particular, we plot the effects of simplifications and of the first coherence test. The improvement of the overestimate reported in Figure 6.1(b) is significant. The first steps of the computation are able to reduce the number of candidates of at least 51% (in MCS) up to around 75% (in CQA). Simplifications are already very effective in CQA, where candidates are reduced of around 45%. It is important to note that the reduction of candidates at this stage applies to all algorithms, while sound answers are produced only by anytime algorithms. This is effective in practice, as shown in Figure 6.1(a). Indeed, anytime algorithms print from 40% (in SBB) to 90% (in CQA) of sound answers already after simplifications, which requires few seconds on the average. The first coherence test further improves the underestimate, which ranges from 52% (in MCS)

Table 6.1: Average running time and number of solved instances

Problem	#	$A2^*$		$A3^*$	
		solved	avg time	solved	avg time
MCS	53	23	181.9	39	254.1
CQA	13	12	118.8	13	89.5
SBB	20	15	53.4	14	65.7
<b>Total</b>	<b>86</b>	<b>50</b>	<b>118.0</b>	<b>66</b>	<b>136.4</b>

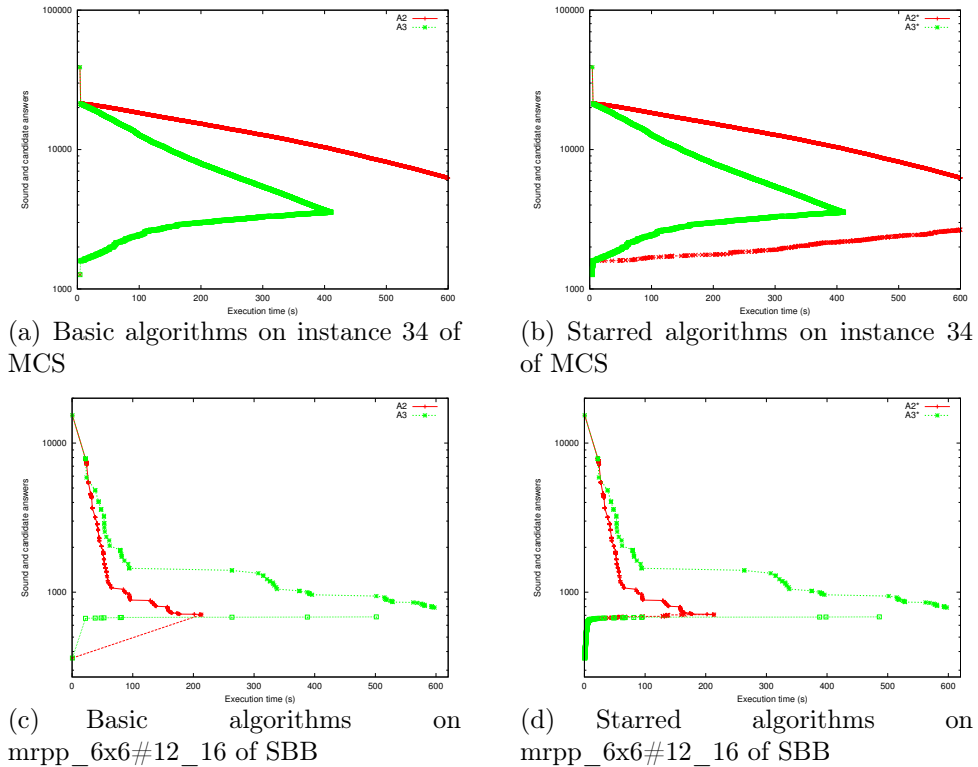


Figure 6.2: Overestimate and underestimate improvement during execution

to around 91% (in CQA). However, we observe that the first coherence test may require some time (25s on the average for SBB instances, with a peak of 193s), which motivated the starred variants. In fact, starred variants can produce underestimates at each restart, not only when a coherence test is completed. Actually, both  $A2^*$  and  $A3^*$  improve progressively the underestimate up to around an additional 30% before the first stable model is found, which is desirable on hard instances.

In order to further confirm the above observations, we analyze in detail the behavior of the algorithms after simplifications. In particular, Figure 6.2 plots both the number of sound answers (line below) and the number of candidate answers (line above) over time. In particular, Figure 6.2(b) is devoted to the starred algorithms on an instance of MCS, whereas Figure 6.2(a) plots the behavior of the basic algorithms on the same instance. First we note that  $A3$  outperforms  $A2$ , which timed out. Notably,  $A2^*$  can produce the underestimate (see the bottom line in Figure 6.2(b)) whereas  $A2$  can only print the overestimate (there is no underestimate line for  $A2$  in Figure 6.2(a)). In general,  $A3$  is able to improve its estimates better than  $A2$ . Note that

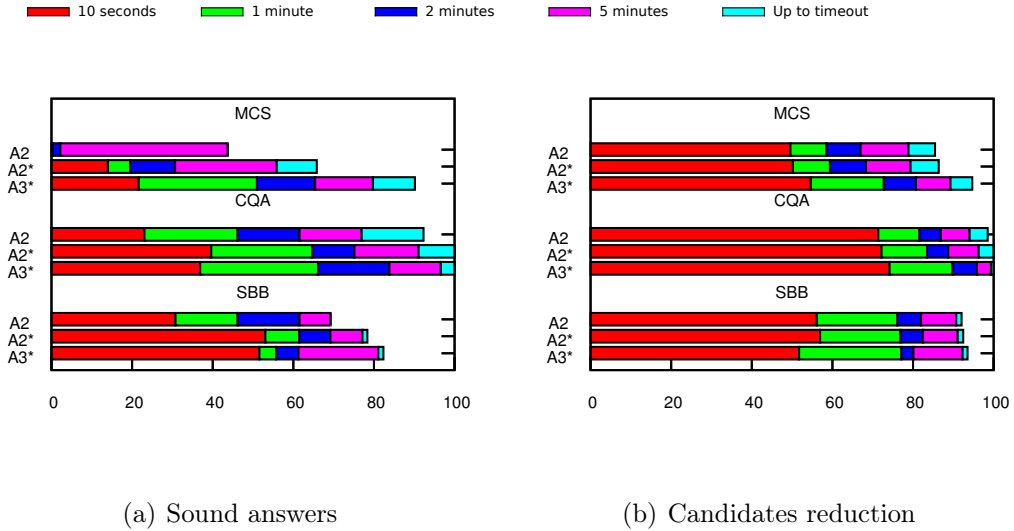


Figure 6.3: Sound answers and candidates reduction after simplification

there is a point in the plots for each improvement of estimates, and lines are very dense on MCS instances. This confirms that MCS instances have a huge number of stable models that can be rapidly computed. We observed an analogous behavior for CQA. Plots for SBB instances on Figure 6.2(c) and Figure 6.2(d) have, instead, sparse lines, confirming that stable model search is harder for this benchmark. Nevertheless, the starred algorithms can rapidly produce most of the sound answers. A deeper look at Figure 6.2(c) suggests that  $A2$  is much faster than  $A3$  in solving this instance. In fact,  $A2$  improves the overestimate faster than  $A3$ .

More insights on the general behavior of the algorithms in the non-deterministic part of the computation can be obtained by looking at Figure 6.3. In particular, Figure 6.3(a) reports the average percentage of sound answers produced *after the simplification step*, while candidates reduction is shown on Figure 6.3(b).  $A3$  is not shown in the figure because it performs similarly to  $A3^*$  in this perspective. We point out that all bars refer to sound answers and candidates remaining after simplifications, also for  $A2$ . As a general observation,  $A2$  prints sound answers only at the end of the computation, while other algorithms are anytime. Consequently,  $A2$  does not provide sound answers as soon as the other algorithms, as shown on Figure 6.3(a). Basically,  $A2$  can print something in the first 10s only for easy instances, while  $A2^*$  improves a lot in this respect. For example,  $A2^*$  outputs around 14% of sound answers already in the first 10s of computation in MCS, while  $A2$  produces no output. Nonetheless,  $A3^*$  performs generally

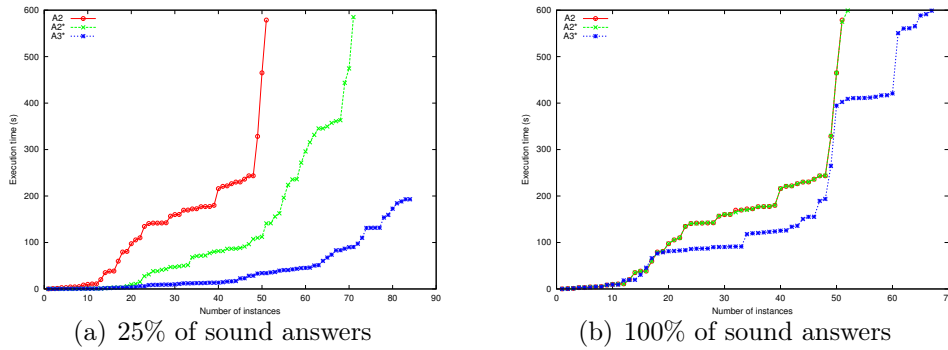


Figure 6.4: Time performance of algorithms for computing underestimates

better than  $A2^*$ . Analogous considerations can be done for the reduction of candidates on Figure 6.3(b).

Another perspective on the behavior of the various algorithms can be obtained by looking at Figure 6.4. Here are reported, for each benchmark, two variants of the classical cactus plot. Recall that, in a cactus plot, the x-axis reports the number of instances that are solved within the time reported on the y-axis. Here we consider variants where on the x-axis is the number of instances for which an algorithm printed 25% (resp. 100%) of sound answers within the time reported on the y-axis. We point out that anytime solvers can print 100% of sound answers before the timeout, even if termination is not reached within the allotted time. Figure 6.4 confirms that  $A2$  is slower than anytime algorithms in printing sound answers. It is interesting to note that the anytime  $A2^*$  improves sensibly  $A2$  in all benchmarks, especially at the beginning of the computation. Note that the differences are more evident in the plots on the left that focus on the first 25% of sound answers. Finally, we confirm that  $A2^*$  is the fastest single-process implementation in SBB. Nonetheless,  $A3^*$  prints more sound answers also in non-terminating instances.

# Chapter 7

## Related Work

In this chapter, we first discuss related work from the SAT area; then we compare existing ASP solvers and their solving approaches with WASP 2. Moreover, the ASP solvers which are more similar to our solver, namely CLASP and WASP, are subject of a detailed comparison with WASP 2.

### 7.1 Relations with SAT, MaxSAT and SMT

In this thesis we proposed several solutions for four different tasks related to propositional answer set solving. The techniques employed for addressing these tasks are related to those proposed in the SAT area.

WASP 2 adopts and extends CDCL backtracking search algorithm [40], learning [24], restarts [25] and conflict-driven heuristics [26] that were first introduced for SAT solving. The mentioned SAT solving methods have been properly adapted and modified for handling efficiently specific properties and language constructs of ASP programs that have no correspondence in SAT formulas, such as the minimality of answer sets and aggregate atoms. These characteristics features of ASP have been dealt with by using a solving architecture that is similar to the concept of propagator introduced in Satisfiability Modulo Theories (SMT) solving [70]. More in detail, WASP 2 implements and extends techniques introduced in the SAT solvers MINISAT [41] and GLUCOSE [42], and combines them with ASP-specific propagators exploiting minimality of answer sets and implementing aggregates.

Optimum answer set search has been addressed in this thesis by adapting core-guided [60, 61, 36] and model-guided algorithms [31] introduced for MaxSAT solving. In particular, we consider an algorithm inspired by *optsat* [32] that we call OPT and its variant called BASIC; the model-guided algorithm MGD [31] introduced for solving MaxSAT; and the core-guided al-



gorithms OLL [33] that has been introduced in the context of ASP and then successfully ported to MaxSAT [37]; PMRES [35] implemented in the MaxSAT solver EVA and BCD [36] implemented in the MaxSAT solver MSUNCORE. It is worth observing that the first attempt of porting MaxSAT algorithms to ASP was described in [33], in which the algorithm OLL was proposed. Nonetheless, to the best of our knowledge, no previous attempt to porting MGD, OPT, BCD and PMRES algorithms to the ASP framework is reported in the literature.

Cautious reasoning is related to the problem of backbone computations of propositional formulas [34, 71]. In fact, the backbone of a propositional formula  $\varphi$  is the set of literals that are true in all models of  $\varphi$ . Several algorithms for computing backbones of propositional formulas are based on variants of the iterative consistency testing algorithm [34, 72], which essentially corresponds to the iterative coherence testing algorithm analyzed in Chapter 6. Backbone search algorithms usually feature additional techniques for removing candidates to be tested, such as *implicant reduction* and *core-based chunking* [73]. Most of the implicant reduction techniques are not applicable to normal ASP programs because of the intrinsic minimality of stable models. For example, backbone search algorithms can reduce their overestimate by removing all unassigned variables when a (partial) model is found; in our setting, ASP solvers always terminate with a complete assignment. Core-based chunking, instead, requires a portfolio of algorithms [72] in order to be effective, which is beyond the scope of this thesis.

## 7.2 ASP Solvers

ASP has obtained growing interest since robust solvers were available. ASP solvers can be classified into *based on translation* (or non-native) and *native* according to the evaluation strategies employed. Solvers based on translation usually perform a translation from ASP to other theories and then use specific solvers for those theories as black box, while native solvers implement specific algorithms and data structures for dealing with ASP programs.

### 7.2.1 Solvers Based on Translations

ASSAT [74] was one of the first non-native solver based on a rewriting of normal ASP programs into propositional formulas and the call of an external SAT solver. Similar techniques were adopted by CMODELS [19], and more recently by the LP2SAT [75] family of solvers. Among these CMODELS is the only one that can deal with disjunctive programs. In particular, CMODELS

performs the task of answer set checking by testing the unsatisfiability of a CNF formula as proposed in [27], and it is able to learn a (loop) formula modeling unfounded sets in case of stability check failure. We also mention other approaches based on translations, like PBMODELS [76] that rewrites ASP to pseudo-Boolean constraints; and LP2DIFF [77] rewriting ASP to SMT. WASP 2 is a native solver, thus, it adopts a very different solving approach with respect to non-native solvers. Nonetheless, some similarities exist, indeed WASP 2 uses Clark’s completion as implemented by ASSAT [74].

### 7.2.2 Native Solvers

Among the first effective native solvers that were proposed we mention DLV [17] and SMOBELS [18].

DLV implements a systematic backtracking without learning and adopts look-ahead heuristics, while WASP 2 is based on CDCL and look-back techniques. Concerning unfounded-free propagation, DLV implements a pruning technique based on finding external supporting rules, while WASP 2 is based on source pointers [18]. Answer set checking is implemented by WASP 2 as an improvement of the algorithm introduced in DLV [27] that avoids the creation of a new formula for each check as detailed in Section 4.5. Concerning optimization problems, both DLV and WASP 2 implement the algorithm BASIC. Nonetheless, WASP 2 implements also several other strategies that are not implemented by DLV. Cautious reasoning is addressed in DLV only by implementing the algorithm *enumeration of models*, and DLV does not print any form of estimation of the result during the computation. We also note that DLV features brave reasoning, which is not currently supported by WASP 2. For the sake of completeness, we mention an extension of DLV [78] that implements backjumping and look-back heuristics, which however does not include learning, restarts, and does not use an implication graph for determining the reasons of conflicts.

SMOBELES implements a DPLL-like algorithm without learning and adopts look-ahead heuristics. It is worth observing that the algorithm based on source pointers as introduced in SMOBELES [18] is also used by WASP 2 for unfounded-free propagation. SMOBELES supports normal logic programs, while disjunctive programs are supported by its extension called GNT [79]. GNT does not perform incremental unsatisfiability tests on CNF formulas as implemented by WASP 2, instead it uses SMOBELES for testing the incoherence of logic programs at each check. The variant of SMOBELES called SMOBELES<sub>CC</sub> [80] features learning and look-back heuristics but do not apply Clark’s completion, resulting in support-related inference rules that are not required in WASP 2. Concerning optimization problems, SMOBELES introduced the

BASIC algorithm that has been implemented also by WASP 2. Nonetheless, WASP 2 supports a portfolio of model-guided and core-guided algorithms that are not implemented by SMODELS. We also mention MINISAT(ID) that extends the SAT solver MINISAT for dealing with ASP programs [81]. However, MINISAT(ID) does not support the full language of ASP as WASP 2.

The native solvers that are more similar to WASP 2 are CLASP [20] and WASP [82]. In the following we detail the differences and similarities of WASP 2 compared with those two solvers.

**Differences with CLASP.** Both WASP 2 and CLASP use source pointers, backjumping, learning, restarts, and look-back heuristics. There are nonetheless several differences with WASP 2 related to data structures and input simplification, for example WASP 2 handles duplicated rules while reading the input program, which results in a lower memory usage of WASP 2 in some benchmarks during our experiments.

WASP 2 differs from CLASP concerning the algorithm for unfounded set check in case of non-HCF components. In particular, the algorithm of WASP 2 is an enhancement of the algorithm of DLV, which is based on the reduction of the unfounded set check problem into the unsatisfiability problem [27]. On the contrary, CLASP creates a set of nogoods encoding unfounded subsets. The unfounded set checks is then performed by extracting nogoods from the interpretation [21]. Both WASP 2 and CLASP uses a technique for performing the unfounded set check under assumptions [21].

Concerning optimization problems, both CLASP and WASP 2 implement the algorithms BASIC and OLL. The latter has been introduced in UNCLASP [33], an experimental branch of CLASP. However, WASP 2 implements MGD, OPT, PMRES and BCD introduced for MaxSAT solving that are not implemented by CLASP. Moreover, WASP 2 improves the performances of the algorithm OLL on weighted benchmarks by implementing the *stratification* strategy.

Concerning cautious reasoning, the algorithm implemented by CLASP is overestimate reduction. WASP 2 differs from this solver especially with respect to the output produced during the computation of cautious consequences. In fact, CLASP only prints overestimates during the computation. Our implementation, instead, is anytime and thus prints both underestimates and overestimates during the computation. Underestimates provide sound answers also when termination is not affordable in reasonable time, and are thus of practical importance for hard problems. It is interesting to observe that among the strategies supported by our implementation there is an anytime variant of the algorithm used by CLASP that performed very well on two

of our three benchmarks. We also note that CLASP features brave reasoning, which is not currently supported by our implementation.

**Differences with WASP.** WASP 2 is a substantially revised version of WASP [82]. First of all we observe that WASP does not implement any program transformation phase, whereas WASP 2 applies both Clark’s completion and program simplification in the style of SATELITE [55]. The Clark’s completion introduces a number of clauses that represent support propagation, which is implemented by a specific propagation procedure in WASP instead. The addition of this preprocessing step brings advantages in both terms of simplifying the implementation of the propagation procedure and in terms of performances. The program simplification step of WASP 2 optimizes the program by eliminating redundant atoms (also introduced by the completion) and shrinking definitions. This results in a program that is usually easier to evaluate. Concerning the unfounded-free propagation both WASP 2 and WASP compute unfounded sets according to the *source pointers* [18] technique. WASP immediately infers unfounded atoms as false, and updates a special implementation of the implication graph. In contrast, WASP 2 learns a rule representing the inference and propagates it with unit propagation. This choice combined with Clark’s completion allows to simplify conflict analysis, learning and backjumping. Indeed, WASP implements specialized variants of these procedures that require the usage of complex data structures that are difficult to optimize. Since in WASP 2 literals are always inferred by the UnitPropagation procedure, we could adopt an implementation of these strategies optimized as in modern SAT solvers. Finally both WASP 2 and WASP implement conflict-driven branching heuristics. WASP 2 uses a branching heuristic inspired to the one of MINISAT, while WASP uses an extension of the BerkMin [83] heuristic extended by adding a look-ahead technique and an additional ASP-specific criterion.

Concerning answer set checking, WASP implements an algorithm similar to the one introduced in [27]. In particular, WASP creates a CNF formula at each stability check and then uses the SAT solver MINISAT in order to check the unsatisfiability of the formula. Optimization problems are handled in WASP by implementing several model-guided and core-guided algorithms, including OPT, MGD, OLL and BCD. However, WASP 2 implements also PMRES and a *stratification* technique for improving the performances on weighted benchmarks. Finally, WASP has no front-end for dealing with cautious reasoning.

# Chapter 8

## Conclusion

This thesis focused on computational tasks related to reasoning with propositional ASP programs, such as model generation, answer set checking, optimum answer set search, and cautious reasoning. These tasks are computationally very hard in general, and this thesis provided algorithms and solutions to solve them efficiently. In particular the contributions of this thesis can be summarized as follows:

1. We studied the task of generating model candidates and we implemented in a new ASP solver a combination of techniques originally introduced for SAT solving, which led to the following publications [52, 82, 84, 85].
2. We proposed a new algorithm for stable model checking that minimizes the overhead of executing multiple calls to a co-NP oracle by resorting to an incremental evaluation strategy and specific heuristics.
3. We implemented a family of algorithms for computing optimum answer sets of programs with weak constraints porting to the ASP setting several algorithms introduced for MaxSAT solving.
4. We introduced a new framework of anytime algorithms for computing the cautious consequences of an ASP knowledge base, that extended existing proposals and included a new algorithm inspired by techniques for the computation of backbones of propositional theories. This work has been presented in [86].

These techniques have been implemented in WASP 2, a new solver for propositional ASP programs. The effectiveness of the proposed techniques and the performance of the new system have been validated empirically on

publicly-available benchmarks taken from ASP competitions and other repositories of ASP applications. Our proposals are effective and WASP 2 represents a valuable addition to the state-of-the-art of ASP solving.

As a final remark, we point out that the source code of WASP 2 has been released under the Apache 2.0 License and can be downloaded at <https://github.com/alviano/wasp>.

# Bibliography

- [1] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R.A., Bowen, K.A., eds.: *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988* (2 Volumes), MIT Press (1988) 1070–1080
- [2] Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. *ACM Trans. Database Syst.* **22** (1997) 364–418
- [3] Apt, K.R., Warren, D.S., Truszczyński, M., eds.: *The Logic Programming Paradigm: A 25-Year Perspective*. 1st edn. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
- [4] Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.* **25** (1999) 241–273
- [5] Lifschitz, V.: Answer set planning. In Schreye, D.D., ed.: *Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999*, MIT Press (1999) 23–37
- [6] Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. *New Generation Comput.* **9** (1991) 365–386
- [7] Baral, C.: *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press (2003)
- [8] Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell.* **175** (2011) 278–298
- [9] Buccafurri, F., Leone, N., Rullo, P.: Enhancing disjunctive datalog by constraints. *IEEE Trans. Knowl. Data Eng.* **12** (2000) 845–860
- [10] Grasso, G., Iiritano, S., Leone, N., Ricca, F.: Some DLV applications for knowledge management. In Erdem, E., Lin, F., Schaub, T., eds.: *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 (2009) 591–597
- [11] Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., Leone, N.: Team-building with answer set programming in the gioia-tauro seaport. *TPLP* **12** (2012) 361–381
- [12] Manna, M., Oro, E., Ruffolo, M., Alviano, M., Leone, N.: The *H<sub>2</sub>L<sub>E</sub>X* system for semantic information extraction. *T. Large-Scale Data- and Knowledge-Centered Systems* **5** (2012) 91–125
- [13] Gebser, M., Kaminski, R., Schaub, T.: *aspcud*: A linux package configuration tool based on answer set programming. In Drescher, C., Lynce, I., Treinen, R., eds.: *Proceedings Second Workshop on Logics for Component Configuration, LoCoCo 2011, Perugia, Italy, 12th September 2011*. Volume 65 of *EPTCS*. (2011) 12–25
- [14] Erdem, E., Erdem, Y., Erdogan, H., Öztok, U.: Finding answers and generating explanations for complex biomedical queries. In Burgard, W., Roth, D., eds.: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*, AAAI Press (2011)
- [15] Mileo, A., Schaub, T., Merico, D., Bisiani, R.: Knowledge-based multi-criteria optimization to support indoor positioning. *Ann. Math. Artif. Intell.* **62** (2011) 345–370
- [16] Ricca, F., Dimasi, A., Grasso, G., Ielpa, S.M., Iiritano, S., Manna, M., Leone, N.: A logic-based system for e-tourism. *Fundam. Inform.* **105** (2010) 35–55
- [17] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.* **7** (2006) 499–562
- [18] Simons, P., Niemelä, I., Soinen, T.: Extending and implementing the stable model semantics. *Artif. Intell.* **138** (2002) 181–234
- [19] Lierler, Y., Maratea, M.: *Cmodels-2*: Sat-based answer set solver enhanced to non-tight programs. In Lifschitz, V., Niemelä, I., eds.: *Logic*



- Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, Proceedings. Volume 2923 of Lecture Notes in Computer Science., Springer (2004) 346–350
- [20] Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. *Artif. Intell.* **187** (2012) 52–89
- [21] Gebser, M., Kaufmann, B., Schaub, T.: Advanced conflict-driven disjunctive answer set solving. In Rossi, F., ed.: *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, Beijing, China, August 3-9, 2013, *IJCAI/AAAI* (2013)
- [22] Calimeri, F., Ianni, G., Ricca, F.: The third open answer set programming competition. *TPLP* **14** (2014) 117–135
- [23] Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., Krennwallner, T., Kronegger, M., Oetsch, J., Pfandler, A., Pührer, J., Redl, C., Ricca, F., Schneider, P., Schwengerer, M., Spindler, L.K., Wallner, J.P., Xiao, G.: The fourth answer set programming competition: Preliminary report. In Cabalar, P., Son, T.C., eds.: *LPNMR*. (2013) 42–53
- [24] Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in Boolean satisfiability solver. In: *ICCAD*. (2001) 279–285
- [25] Gomes, C.P., Selman, B., Kautz, H.A.: Boosting combinatorial search through randomization. In Mostow, J., Rich, C., eds.: *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98*, July 26-30, 1998, Madison, Wisconsin, USA., AAAI Press / The MIT Press (1998) 431–437
- [26] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: *Proceedings of the 38th Design Automation Conference, DAC 2001*, Las Vegas, NV, USA, June 18-22, 2001, ACM (2001) 530–535
- [27] Koch, C., Leone, N., Pfeifer, G.: Enhancing disjunctive logic programming systems by SAT checkers. *Artif. Intell.* **151** (2003) 177–212
- [28] Arenas, M., Bertossi, L.E., Chomicki, J.: Answer sets for consistent query answering in inconsistent databases. *TPLP* **3** (2003) 393–424

- [29] Eiter, T.: Data integration and answer set programming. In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: *Logic Programming and Nonmonotonic Reasoning, 8th International Conference, LPNMR 2005, Diamante, Italy, September 5-8, 2005, Proceedings*. Volume 3662 of *Lecture Notes in Computer Science.*, Springer (2005) 13–25
- [30] Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. *Artif. Intell.* **172** (2008) 1495–1539
- [31] Morgado, A., Heras, F., Marques-Silva, J.: Model-guided approaches for MaxSAT solving. In: *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013, IEEE (2013)* 931–938
- [32] Rosa, E.D., Giunchiglia, E., Maratea, M.: Solving satisfiability problems with preferences. *Constraints* **15** (2010) 485–515
- [33] Andres, B., Kaufmann, B., Matheis, O., Schaub, T.: Unsatisfiability-based optimization in clasp. In Dovier, A., Costa, V.S., eds.: *ICLP (Technical Communications)*. (2012) 211–221
- [34] Marques-Silva, J., Janota, M., Lynce, I.: On computing backbones of propositional theories. In Coelho, H., Studer, R., Wooldridge, M., eds.: *ECAI*. Volume 215 of *Frontiers in Artificial Intelligence and Applications.*, IOS Press (2010) 15–20
- [35] Narodytska, N., Bacchus, F.: Maximum satisfiability using core-guided maxsat resolution. In Brodley, C.E., Stone, P., eds.: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, AAAI Press (2014) 2717–2723
- [36] Heras, F., Morgado, A., Marques-Silva, J.: Core-guided binary search algorithms for maximum satisfiability. In Burgard, W., Roth, D., eds.: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*, AAAI Press (2011)
- [37] Morgado, A., Dodaro, C., Marques-Silva, J.: Core-guided MaxSAT with soft cardinality constraints. In O’Sullivan, B., ed.: *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*. Volume 8656 of *Lecture Notes in Computer Science.*, Springer (2014) 564–573

- [38] Cook, S.A.: The complexity of theorem-proving procedures. In Harrison, M.A., Banerji, R.B., Ullman, J.D., eds.: Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA, ACM (1971) 151–158
- [39] Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. *Commun. ACM* **5** (1962) 394–397
- [40] Silva, J.P.M., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers* **48** (1999) 506–521
- [41] Eén, N., Sörensson, N.: An extensible sat-solver. In Giunchiglia, E., Tacchella, A., eds.: Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. Volume 2919 of Lecture Notes in Computer Science., Springer (2003) 502–518
- [42] Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In Boutilier, C., ed.: IJCAI. (2009) 399–404
- [43] Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. *Inf. Process. Lett.* **47** (1993) 173–180
- [44] Audemard, G., Simon, L.: Refining restarts strategies for SAT and UNSAT. In: Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings. Volume 7514 of Lecture Notes in Computer Science., Springer (2012) 118–126
- [45] Eén, N., Sörensson, N.: Temporal induction by incremental sat solving. *Electr. Notes Theor. Comput. Sci.* **89** (2003) 543–560
- [46] Fu, Z., Malik, S.: On solving the partial max-sat problem. In Biere, A., Gomes, C.P., eds.: SAT. Lecture Notes in Computer Science (2006) 252–265
- [47] Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in *gringo* series 3. In Delgrande, J.P., Faber, W., eds.: LPNMR. Volume 6645 of LNCS., Springer (2011) 345–351
- [48] Leone, N., Rullo, P., Scarcello, F.: Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Inf. Comput.* **135** (1997) 69–112

- [49] Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative problem-solving using the DLV system. In Minker, J., ed.: *Logic-Based Artificial Intelligence*. Volume 597 of *The Springer International Series in Engineering and Computer Science*. Springer US (2000) 79–103
- [50] Manna, M., Ricca, F., Terracina, G.: Consistent query answering via ASP from different perspectives: Theory and practice. *TPLP* **13** (2013) 227–252
- [51] Leone, N., Perri, S., Scarcello, F.: Improving ASP instantiators by join-ordering methods. In Eiter, T., Faber, W., Truszczynski, M., eds.: *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings*. Volume 2173 of *Lecture Notes in Computer Science.*, Springer (2001) 280–294
- [52] Alviano, M., Dodaro, C., Ricca, F.: Preliminary report on WASP 2.0. *CoRR* [abs/1404.6999](https://arxiv.org/abs/1404.6999) (2014)
- [53] Alviano, M., Faber, W., Leone, N., Perri, S., Pfeifer, G., Terracina, G.: The disjunctive datalog system DLV. In de Moor, O., Gottlob, G., Furche, T., Sellers, A.J., eds.: *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*. Volume 6702 of *Lecture Notes in Computer Science.*, Springer (2010) 282–301
- [54] Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In Veloso, M.M., ed.: *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*. (2007) 386
- [55] Eén, N., Biere, A.: Effective preprocessing in sat through variable and clause elimination. In: *SAT*. Volume 3569 of *LNCS.*, Springer (2005) 61–75
- [56] Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: On the implementation of weight constraint rules in conflict-driven ASP solvers. In Hill, P.M., Warren, D.S., eds.: *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*. Volume 5649 of *Lecture Notes in Computer Science.*, Springer (2009) 250–264
- [57] Janhunen, T., Niemelä, I., Simons, P., You, J.: Unfolding partiality and disjunctions in stable model semantics. In Cohn, A.G., Giunchiglia,

- F., Selman, B., eds.: KR 2000, Principles of Knowledge Representation and Reasoning Proceedings of the Seventh International Conference, Breckenridge, Colorado, USA, April 11-15, 2000., Morgan Kaufmann (2000) 411–422
- [58] Pfeifer, G.: Improving the model generation/checking interplay to enhance the evaluation of disjunctive programs. In Lifschitz, V., Niemelä, I., eds.: Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, Proceedings. Volume 2923 of Lecture Notes in Computer Science., Springer (2004) 220–233
- [59] Zhang, L., Malik, S.: Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In: DATE, IEEE Computer Society (2003) 10880–10885
- [60] Manquinho, V.M., Silva, J.P.M., Planes, J.: Algorithms for weighted Boolean optimization. In Kullmann, O., ed.: 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 (2009) 495–508
- [61] Ansótegui, C., Bonet, M.L., Levy, J.: Solving (weighted) partial MaxSAT through satisfiability testing. In Kullmann, O., ed.: 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 (2009) 427–440
- [62] Larrosa, J., Heras, F.: Resolution in max-sat and its relation to local consistency in weighted csps. In Kaelbling, L.P., Saffiotti, A., eds.: IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005, Professional Book Center (2005) 193–198
- [63] Ansótegui, C., Bonet, M.L., Levy, J.: Sat-based maxsat algorithms. *Artif. Intell.* **196** (2013) 77–105
- [64] Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczynski, M.: The second answer set programming competition. In Erdem, E., Lin, F., Schaub, T., eds.: 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 (2009) 637–654
- [65] Calimeri, F., Gebser, M., Maratea, M., Ricca, F.: The design of the fifth answer set programming competition. *CoRR* **abs/1405.3710** (2014)
- [66] Lifschitz, V., Turner, H.: Splitting a logic program. In Hentenryck, P.V., ed.: *Logic Programming, Proceedings of the Eleventh International Conference on Logic Programming, Santa Marherita Ligure, Italy, June 13-18, 1994*, MIT Press (1994) 23–37
- [67] Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: *AAAI*, AAAI Press (2007) 385–390
- [68] Kolaitis, P.G., Pema, E., Tan, W.C.: Efficient querying of inconsistent databases with binary integer programming. *PVLDB* **6** (2013) 397–408
- [69] Jarvisalo, M., Berre, D.L., Roussel, O., Simon, L.: The international sat solver competitions. *AI Magazine* **33** (2012)
- [70] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL( $T$ ). *J. ACM* **53** (2006) 937–977
- [71] Slaney, J.K., Walsh, T.: Backbones in optimization and approximation. In Nebel, B., ed.: *IJCAI*, Morgan Kaufmann (2001) 254–259
- [72] Janota, M., Lynce, I., Marques-Silva, J.: Algorithms for computing backbones of propositional formulae. *AI Commun.* (2014) To appear.
- [73] Ravi, K., Somenzi, F.: Minimal assignments for bounded model checking. In Jensen, K., Podelski, A., eds.: *TACAS*. Volume 2988 of LNCS., Springer (2004) 31–45
- [74] Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. *Artif. Intell.* **157** (2004) 115–137
- [75] Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics* **16** (2006) 35–86
- [76] Liu, L., Truszczyński, M.: Pmodels - software to compute stable models by pseudoboolean solvers. In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: *Logic Programming and Nonmonotonic Reasoning, 8th International Conference, LPNMR 2005, Diamante, Italy, September*

- 5-8, 2005, Proceedings. Volume 3662 of Lecture Notes in Computer Science., Springer (2005) 410–415
- [77] Janhunen, T., Niemelä, I., Sevalnev, M.: Computing stable models via reductions to difference logic. In Erdem, E., Lin, F., Schaub, T., eds.: 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 (2009) 142–154
- [78] Ricca, F., Faber, W., Leone, N.: A backjumping technique for disjunctive logic programming. *AI Commun.* **19** (2006) 155–172
- [79] Janhunen, T., Niemelä, I.: GNT - A solver for disjunctive logic programs. In Lifschitz, V., Niemelä, I., eds.: Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, Proceedings. Volume 2923 of Lecture Notes in Computer Science., Springer (2004) 331–335
- [80] Ward, J., Schlipf, J.S.: Answer set programming with clause learning. In Lifschitz, V., Niemelä, I., eds.: Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, Proceedings. Volume 2923 of Lecture Notes in Computer Science., Springer (2004) 302–313
- [81] Pooter, S.D., Wittocx, J., Denecker, M.: A prototype of a knowledge-based programming environment. In Tompits, H., Abreu, S., Oetsch, J., Pührer, J., Seipel, D., Umeda, M., Wolf, A., eds.: Applications of Declarative Programming and Knowledge Management - 19th International Conference, INAP 2011, and 25th Workshop on Logic Programming, WLP 2011, Vienna, Austria, September 28-30, 2011, Revised Selected Papers. Volume 7773 of Lecture Notes in Computer Science., Springer (2011) 279–286
- [82] Alviano, M., Dodaro, C., Faber, W., Leone, N., Ricca, F.: WASP: A native ASP solver based on constraint learning. In Cabalar, P., Son, T.C., eds.: LPNMR. Volume 8148 of LNCS., Springer (2013) 54–66
- [83] Goldberg, E., Novikov, Y.: Berkmin: A fast and robust sat-solver. *Discrete Applied Mathematics* **155** (2007) 1549–1561
- [84] Dodaro, C.: Engineering an efficient native ASP solver. *TPLP* **13** (2013)

- [85] Alviano, M., Dodaro, C., Ricca, F.: Comparing alternative solutions for unfounded set propagation in ASP. In Baldoni, M., Baroglio, C., Boella, G., Micalizio, R., eds.: *AI\*IA 2013: Advances in Artificial Intelligence - XIIIth International Conference of the Italian Association for Artificial Intelligence*, Turin, Italy, December 4-6, 2013. Proceedings. Volume 8249 of *Lecture Notes in Computer Science.*, Springer (2013) 1–12
- [86] Alviano, M., Dodaro, C., Ricca, F.: Anytime computation of cautious consequences in answer set programming. *TPLP* **14** (2014) 755–770