Università della Calabria

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea Magistrale in Informatica

WASP: A New Model Generator

Relatori:
Prof. Wolfgang **Faber**

Prof. Francesco **Ricca**

Dr. Mario **Alviano**

Dr. Marco **Sirianni**

Studente:
Carmine **Dodaro**
Matricola 138100

Anno Accademico 2010/2011

"Anyone who has never made a mistake has never tried anything new." A. Einstein

Dedicated to my parents, *Mario and Tina*, to my sisters, *Giulia and Rita* and to my grandmother *Rita*.

Abstract

We present a new ASP solver called WASP for ground ASP programs, which builds upon related techniques originally introduced for SAT solving and that have been extended to cope with disjunctive logic programs under the stable model semantics. We describe the key components of this solving strategy, namely learning, restarts, look-back-based heuristics, and backjumping. Moreover, we introduce a new heuristic based on a mixed approach between look-back and look-ahead techniques. Moreover, we present the results of an experimental analysis that we conducted in order to assess the impact of these techniques on both random and structured instances. In particular, we compared our system with the state-of-the-art ASP systems DLV and ClaspD, and WASP resulted to be performance-wise competitive, and even faster in various cases, than existing systems.

Acknowledgements

I want to say thanks:

- To Prof. **Francesco Ricca**, who always supported me during these months. I could not ask for a better supervisor.
- To Prof. **Wolfgang Faber**, who helped me during my stay in Vienna.
- To Dr. **Mario Alviano** and Dr. **Marco Sirianni**: the time spent with you was a great pleasure and helped me to take this thesis more easy.
- To Prof. **Nicola Leone**, for his interest in my work and his precious suggestions.
- To Prof. **Thomas Eiter**, **Eva Nodoma**, Dr. **Thomas Krennwallner** and Dr. **Peter Scüller** for their hospitality and their kindness when I was in Vienna.
- To all friends met in Vienna, especially, to **Tran Trung Kien**, **Tri Kurniawanwijaya** and **Seif El-Din Bairakdar**. You made me feel very comfortable in the lab, even when the work was very hard.
- To all of my friends and colleagues for their support during these months.

C. D.

Contents

Chapter 1

Introduction

Answer Set Programming (ASP) [1] is a declarative programming paradigm which has been proposed in the area of non-monotonic reasoning and logic programming. The idea of ASP is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use a solver to find them [2].

The ASP language considered here allows disjunction in rule heads and nonmonotonic negation in rule bodies. These features make ASP very expressive; all problems in the second level of the polynomial hierarchy are indeed expressible in ASP [58]. Therefore, ASP is strictly more expressive than SAT (unless $P = NP$). Despite the intrinsic complexity of the evaluation of ASP, after twenty years of research many efficient ASP systems have been developed (e.g. [3, 4, 5, 6, 7, 8, 9, 10]). The availability of robust implementations made ASP a powerful tool for developing advanced applications in the areas of Artificial Intelligence, Information Integration, or Knowledge Management; for example, ASP has been used in applications for team-building [61], semantic-based information extraction [62], and e-tourism [63]. These applications of ASP have confirmed the viability of the use of ASP. Nonetheless, the interest in developing more effective and faster systems is still a crucial and challenging research topic, as witnessed by the results of the ASP Contest series [11, 12, 13].

This thesis provides a contribution in the aforementioned context. We present WASP, a new ASP solver for ground ASP programs that builds upon related techniques, originally introduced for SAT solving, which have been extended to cope with disjunctive logic programs under the stable model semantics. In particular, the new system employs an adapted version of the Davis Putnam Logemann Loveland (DPLL) backtracking search algorithm, clause learning, restarts and conflict-driven heuristics in the style of Berk-Min [34]. The mentioned SAT-solving methods have also been combined

with state-of-the-art pruning techniques adopted by modern native disjunctive ASP systems. In particular, the role of Boolean Constraint Propagation in SAT-solvers (based on the simple *unit propagation* inference rule) is taken by a procedure combining a set of inference rules. Those rules combine an extension of the well-founded operator for disjunctive programs with a number of techniques based on ASP program properties (see, e.g., [37]). Moreover, WASP uses a new branching heuristics tailored for ASP programs, which is based on a mixed approach between BerkMin-like and look-ahead-based heuristics, taking into account the minimality of answer sets (a requirement not present in SAT solving).

We also present the results of an experimental analysis that we conducted in order to assess the performance of WASP on both random and structured instances. The obtained results are encouraging: the new system is already competitive and on several domains even faster than the existing state-of-the-art ASP systems DLV [3] and ClaspD [7]. This result is particularly promising since there is still room for improvements in the implementation of WASP, e.g., through the optimization and tuning of data structures and heuristic parameters.

It is worth mentioning that an extract of this thesis has been published in [55], and that a preliminary version of this work has been presented in the XXVI Italian conference on Computational Logic held in Pescara from 31 August 2011 to 2 September 2011 [57].

This 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 WASP starting from the solving strategy and then we present the design choices regarding propagation, constraint learning, restarts, and the new heuristic. Moreover, in Chapter 5 we present the results of the experiments conducted for assessing the impact of these techniques, on instances also used in the last ASP Competition 2011 [13]. Chapter 6 concludes this thesis.

Chapter 2

Propositional Solving Techniques in SAT

The SAT problem is a well-known NP-complete problem. The SAT problem consists of checking whether a satisfying variable assignment V exists for a propositional formula ϕ or determining that the formula is unsatisfiable. The formula ϕ is usually expressed in Conjunctive Normal Form (CNF) format, that is, formula ϕ is composed by a conjunction of clauses, each of which is a disjunction of variables. In a CNF formula, in order to satisfy the formula ϕ, each clause must be satisfied. In the following, we will assume that all formulae are in CNF format.

In this chapter we provide a description of the solving techniques used in SAT solvers, since efficient techniques employed by modern SAT solvers were ported to ASP for devising efficient systems such as ClaspD. We first describe the Davis Putnam Logemann Loveland (DPLL) algorithm, which is the kernel procedure of many SAT solvers; after that, we describe a technique called clause learning (or simply learning) and some effective heuristics based on this technique.

2.1 DPLL

The DPLL algorithm is a complete, backtracking-based algorithm for deciding the satisfiability of propositional logic formulae in conjunctive normal form, i.e. for solving the CNF-SAT problem. The basic algorithm runs by choosing an undefined literal, assigning a truth value to it, simplifying the formula accordingly, and, then recursively checking whether the simplified formula is satisfiable; if this is the case, the original formula is satisfiable; otherwise, the same recursive check is done assuming the complementary

truth value. The DPLL algorithm enhances over the basic backtracking algorithm by the use of both *unit propagation* and *pure literal elimination* at each step.

Unit propagation. If a clause is a unit clause, i.e. it contains only a single unassigned literal, this clause can only be satisfied by assigning the necessary truth value to make this literal true. In practice, this derivation is done frequently and it often leads to deterministic cascades of units derivations with the effect of pruning the search space.

Pure literal elimination. If a propositional variable occurs with only one polarity in the formula, it is called pure. Pure literals can always be assigned in a way that makes all clauses containing them true. Thus, these clauses do not constrain the search anymore and can be deleted. While this optimization is part of the original DPLL algorithm, most current implementations omit it, as the effect for efficient implementations now is negligible or, due to the overhead for detecting purity, even negative.

function DPLL(ϕ)

  for every unit clause c in ϕ
    ϕ = *unit_propagation*(c, ϕ);
  for every literal ℓ^p that occurs pure in ϕ
    ϕ = *pure_literal_elimination*(ℓ^p, ϕ);

  if ϕ "is a consistent set of literals"
    then **return** true;

  if ϕ "contains an empty clause"
    then **return** false;

  ℓ = *choose_literal(ϕ)*;
  return DPLL($\phi \wedge \ell$) OR DPLL($\phi \wedge \neg\ell$);

Figure 2.1: DPLL algorithm

DPLL starts from a formula ϕ and first calls a function *unit propagation* for inferring a truth value for unassigned literals in unit clauses. After that, the function *pure literal elimination* is called for assigning a truth value for all pure literals. Satisfiability of the formula is detected either when all variables are assigned without generating the empty clause, or, in modern implementations, if all clauses are satisfied. Unsatisfiability of a given partial assignment is detected if one clause becomes empty, i.e. if all its variables

have been assigned in a way that makes the corresponding literals false. Unsatisfiability of the complete formula can only be detected after exhaustive search. If satisfiability or unsatisfiability cannot be determined at this point of the computation, an undefined literal ℓ, called *branching variable*, is selected according to a branching heuristic criterion and DPLL is called recursively.

2.2 Learning

In addition to basic DPLL, some algorithms utilize a pruning technique called learning. Learning is the process that adds some clauses to the original formula, which are deduced from a conflict to avoid following the same dead-end path in the future search. These clauses contain a certain combination of variable assignments that are not valid because they will force the conflicting variable to assume both the value true and false [15]. Some SAT solvers add to the basic learning another technique called *restarts*, that consist of a halt in the solution process, and a restart of the search, in order to further exploit the information obtained from the learned clauses. To perform learning an additional structure called Implication Graph is built during the unit propagation. This data structure is exploited for defining a number of different learning schemas, which correspond to different clause learning strategies.

2.2.1 Implication Graph

The implication relationships of variable assignments during the solving process can be expressed as an Implication Graph. An Implication Graph is a directed graph where each vertex represents a variable assignment. An edge from a vertex V1 to a vertex V2 means that V1 is the reason for the assignment V2. Each variable has a decision level associated to it.

In an Implication Graph with no conflict there is at most one vertex for each variable. A conflict occurs when there is a vertex for the true assignment and a vertex for the false assignment of the same variable (called the conflicting variable).

A relevant concept employed in several learning strategies is the one of unique implication point. A **Unique Implication Point (UIP)** is a vertex at the current decision level that dominates both vertices corresponding to the conflicting variable. A vertex V1 of the decision level d is said to *dominate* a vertex V2 of the same decision level d if and only if V1 is contained in all paths connecting the decision variable of the decision level d to V2. It is important to note that 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 variable is always a UIP).

2.2.2 Different Learning Schemes

Rel_Sat

The Rel_Sat engine puts all variables assigned prior to the current level and the current decision variable in the learned clause. The maximum decision level of the variables, except the current decision level variable, is the decision level to backtrack. After backtracking, the conflict clause will become a unit clause, where the decision variable is the unit literal.

First UIP

In this learning scheme, we order the UIPs starting from the conflict. We learn a clause that is composed by the first UIP in this order and by the variables from a smaller decision level that imply the conflict. The first UIP usually allows to learn the smallest constraints implying the conflict. The first UIP is shown in dark gray in Figure 2.2 reporting an example Implication Graph.

GRASP

GRASP, in addition to the first UIP learning scheme, adds other clauses to the input formula. In particular, the GRASP learning engine will add clauses representing each reconvergence between the various UIPs for the current decision level, and adds specific clauses for forcing backjumping (see [15, page 4] for details). This schema introduces many additional constraints w.r.t. the first UIP schema and this may result in poor efficiency in some cases.

Others Learning Schemes

In addition to these three learning schemes, many more options exist. One naive learning scheme is to add to the clause only the decision variable involved in the conflict. It is not a good solution because such a combination of decisions will never occur again in the future unless we restart. Another one is called *All UIP* scheme. In this case the solver has to find the first UIP for each decision level from the current one until reaching decision level 1, learning a constraint for each of them.

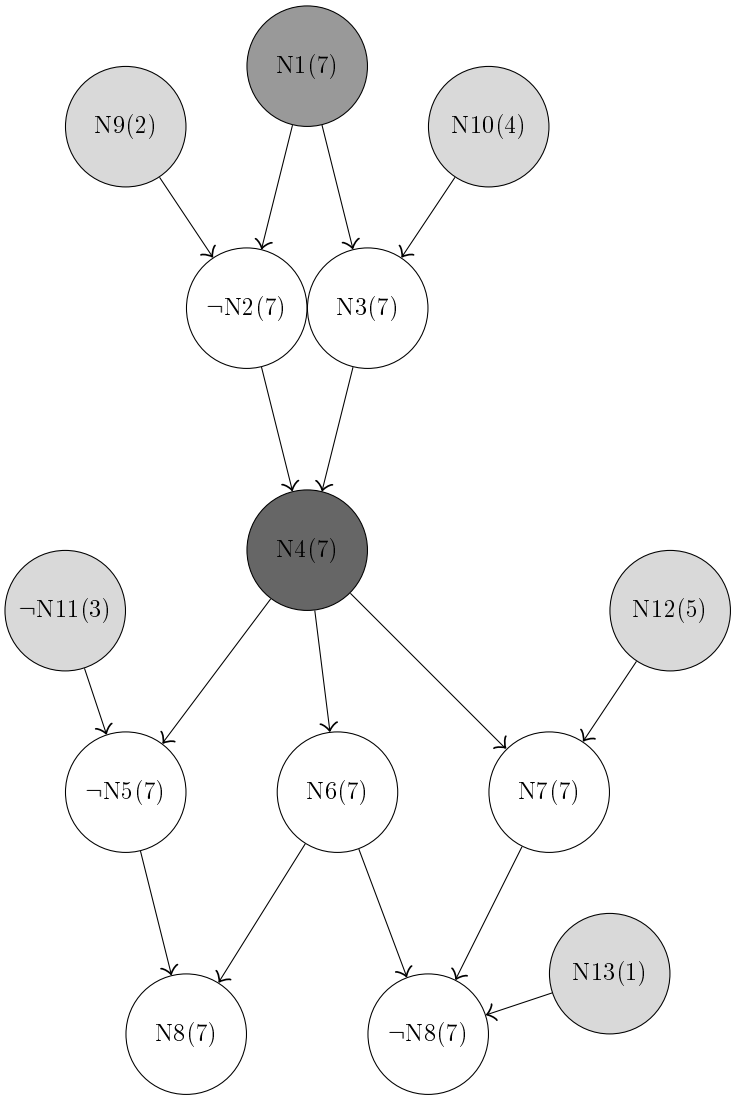


Figure 2.2: Implication Graph (First UIP: N4; Level in parentheses)
N1 is the decision variable of the current decision level (7) which led to the derivation of a truth value for variables N2 to N8. Variables N9 to N13 have been assigned a truth value in a previous decision level. N8 is conflictual because there are two nodes for it in the graph. Note also that N1 and N4 are contained in all paths from N1 to either N8 or ¬N8. Moreover, N1 and N4 are UIPs. Since N4 is the UIP closest to the conflict, it is the first UIP.

2.3 Heuristics

There are two main types of branching heuristics in the SAT literature: look-ahead and look-back. The look-ahead based heuristics estimate, for any unassigned variable and truth value, the effects of setting that variable to that truth value. The look-back based heuristics use learning techniques for recovering information from a failure in the search tree. Since the most modern and effective solvers adopt a look-back heuristics, in this thesis we focus only on this type of branching heuristics, describing the two most relevant proposals.

VSIDS. The Variable State Independent Decaying Sum (VSIDS) heuristic maintains a counter cl(L) (for each literal L), which is increased when a new clause containing L is added to the formula. Counters are initialized to 0. Periodically, all counters are halved. The next literal to be picked is the one with the highest counter values [18]. The two essential features of VSIDS are the following:

1. the cost to calculate the counters is negligible;
2. VSIDS gives preference to literals that are involved in recent conflicts, in this sense it is dynamic.

BerkMin. The heuristic used in the BerkMin SAT solver is partially or fully adopted by almost all modern SAT solvers. It differs from VSIDS as follows: Learned clauses are organized in a list, and with every new conflict the new clause is appended to the head of the list. The next decision variable is picked from the top-most unsatisfied clause. If no such clause exists or if all the learned clauses are satisfied, then a variable with the highest cv counter is chosen. The counter cv(L) is obtained by summing cl(L) and cl($\neg$ L), where the cl counter is lightly different from the one having the same name in the VSIDS heuristics. In fact, in BerkMin this counter is increased also for the variables encountered during the Implication Graph visit done during the learning of the first UIP conflict clause. At regular intervals, BerkMin divides all the counters by 4. BerkMin decides which literal to pick using a global counter, gcl(L), measuring the global contribution of each literal to the conflicts. The counter gcl is initialized to 0 and increased whenever cl is increased, but it is never divided. If a top-most unsatisfied clause exists, BerkMin picks the literal with the highest gcl counter. If there is no unsatisfied top-most clause, then BerkMin picks the literal with the highest value of two(L), where two(L) calculates the number of binary clauses in the neighborhood of literal L [18][34].

Chapter 3

Answer Set Programming

Computer applications pervade in our life, and many problems of everyday life are handled in an automated way. However, many of these problems, especially those for which a deterministic polynomial time algorithm is unknown, are not easy to solve by a computer. In fact, the traditional problem solving approach is based on the well-known imperative programming paradigm. According to this paradigm, high-level programming skills as well as deep domain knowledge are required in order to find a good algorithm for a hard problem. Stated differently, solving hard problems using the imperative programming approach is often troublesome and inefficient. Moreover, even small changes in the specification of a hard problem typically require a lot of effort for re-engineering the algorithms that solve the problem, as well as for the implementations of such algorithms. The explanation is that the knowledge regarding the problem and its solutions is represented implicitly by representing a specific way of solving the problem rather than by the problem itself. An alternative approach to problem solving is based on declarative programming. In this case, the problem and its solutions are stated explicitly in the form of an executable specification: the problem is solved by stating the features of its solution, rather than specifying how a solution has to be obtained.

Answer Set Programming (ASP) [54] is a declarative programming approach that provides a simple formalism for knowledge representation. ASP is based on the stable model semantics of logic programming [56] and allows for expressing all problems in the second level of the polynomial hierarchy [58]. The idea is to represent a given computational problem by means of a logic program the answer sets of which correspond to solutions, and then use an answer set solver to find such solutions.

In this chapter, after introducing the syntax in Section 3.1 and the semantics of ASP in Section 3.2, we show how ASP can be used as a powerful

knowledge representation and reasoning tool in Section 3.3; after that, in Section 3.4, we show the architecture of a general system for evaluating ASP programs.

3.1 Syntax

The syntax of Answer Set Programming is described in the following. By convention, strings starting with uppercase letters refer to logical variables, while strings starting with lower case letters refer to constants. A **term** is either a variable or a constant. A **standard atom** is an expression $p(t_1, \ldots, t_n)$, where p is a **predicate** of arity n and $t_1, \ldots, t_n$ are terms. 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 ground conjunction of atoms. An **aggregate function** is of the form $f(\mathrm{S})$, 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 $f(\mathrm{S})$ is an aggregate function, $\prec \in \{=, <, \leq, >, \geq\}$ is a predefined comparison operator, and T is a costant referred to as **guard**.

A **literal** is either an atom a or its default negation $not\ a$. Given a literal ℓ we will use $\neg\ell$ for denoting its complement, that is $not\ a$ if $\ell = a$ and a if $\ell = not\ a$, where a is an atom.

Definition 1. A **disjunctive rule r** is a formula

$$a_1 \vee \ldots \vee a_n :- b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_m. \tag{3.1}$$

where $a_1, \ldots, a_n$ are **atoms**, $b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_m$ are **literals** and $n \geq 0$, $m \geq k \geq 0$.

Definition 2. Given a rule r,

- let $H(r) = \{a_1, \ldots, a_n\}$ denote the set of **head atoms**,
- let $B^+(r) = \{b_1, \ldots, b_k\}$ denote the set of **positive body literals**,
- let $B^-(r) = \{not\ b_{k+1}, \ldots, not\ b_m\}$ denote the set of **negative body literals**,
- let $B(r) = B^+(r) \cup B^-(r)$ denote the set of **body literals**.

Definition 3. A **positive rule** r is a rule where $B^-(r) = \emptyset$.

Definition 4. An **integrity constraint**, or simply **constraint**, is a rule r where $H(r) = \emptyset$.

Definition 5. A **fact** is a rule r where $B(r) = \emptyset$ and $|H(r)| = 1$. In this case, we usually omit the :- sign.

Definition 6. A **disjunctive logic program**, or simply **program**, is a finite set of rules.

Definition 7. A program Π is called **positive** if all rules in Π are positive.

In ASP, rules in programs are usually required to be safe.

Definition 8. A rule r is **safe** if each variable appearing in r also appears in at least one positive literal in the body of that rule.

A program is safe, if each of its rules is safe. In the following, we will only consider safe programs.

Definition 9. A term, an atom, a literal, a rule, or a program is called **ground** if no variables appear in it.

Example 1 (Disjunctive logic program)**.** Consider the following disjunctive logic program:

```
r1: a(T) v b(T) :- c(T,Z), d(Z).
r2: :- c(T,Z), f(T), not b(T).
r3: c(1,2).
```

- *r1* is a disjunctive positive rule with $H(r1) = \{a(T), b(T)\}$, $B^+(r1) = \{c(T, Z), d(Y)\}$, and $B^-(r1) = \emptyset$.

- *r2* is an integrity constraint with $B^+(r2) = \{c(T, Z), f(T)\}$, and $B^-(r2) = \{\neg b(T)\}$.

- *r3* is a fact. Moreover, rule *r3* is ground, because no variables appear in it.

3.2 Semantics

The Answer Set Semantics is defined on ground programs and it is given by its stable models.

Definition 10. Given a program Π, the **Herbrand Universe** U_Π is the set of all constants appearing in Π.

Definition 11. Given a program Π, the **Herbrand Base** B_Π is the set of all possible ground atoms which can be constructed from the predicate symbols appearing in Π with the constants of U_Π.

Example 2 (Herbrand Universe and Herbrand Base.)**.** Consider the following program Π_0:

```
r1: a(X) v b(X) :- c(X).
r2: d(X) :- c(X), b(X).
r3: c(1).
r4: c(2).
```

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)$\}$.

Definition 12. For any rule r, $Ground(r)$ denotes the set of rules obtained by replacing each variable in r by constants in U_Π in all possible ways. For any program Π, its ground instantiation is the set $Ground(\Pi) = \bigcup_{r\in\Pi} Ground(r)$.

Example 3 (Ground Instantiation)**.** Consider the program Π_0 in the example 2. Its ground instantion is the following:

```
g1: a(1) v b(1) :- c(1).
g2: a(2) v b(2) :- c(2).
g3: d(1) :- c(1), b(1).
g4: d(2) :- c(2), b(2).
g5: c(1).
g6: c(2).
```

Note that the atoms *c(1)* and *c(2)* are already ground in Π_0, while the rules *g1* and *g2* are obtained by replacing the variables in *r1* with constants in U_{Π_0} and the rules *g3* and *g4* are obtained by replacing the variables in *r2* with constants in U_{Π_0}.

Definition 13. A set L of ground literals is said to be **consistent** if, for every literal $\ell \in$ L, its complementary literal $\neg\ell \notin$ L.

Definition 14. An **interpretation** I for Π is a consistent set of ground literals over atoms in B_Π.

A ground literal ℓ is interpreted as follows:

- ℓ is **true** w.r.t. I if $\ell \in$ I;
- ℓ is **false** w.r.t. I if its complementary literal is in I;

- ℓ is **undefined** if it is neither true nor false w.r.t. I.

Let r be a rule in Π. 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 ℓ is false w.r.t. I.

Definition 15. An interpretation I is **total** if and only if for each literal $l \in B_\Pi$, $l \in$ I or $\neg l \in$ I, otherwise I is **partial**.

Definition 16. 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.

Definition 17. Let I(S) denote the multiset $[\ t_1 | \langle t_1, \ldots, t_n \rangle : conj \in S \wedge conj \text{ is true w.r.t. } I\ ]$.

Definition 18. The valutation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. I is the the result of the application of f on $I(S)$.

An aggregate atom A of the form $f(S) \prec k$ is **true** w.r.t. I if $I(f(S)) \prec k$ holds, otherwise, A is **false**.

Definition 19. 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.

Definition 20. A total interpretation M is a **model** for Π if and only if for each rule $r \in \Pi$, $B(r)$ is true w.r.t M whenever $H(r)$ is true w.r.t. M.

Stated differently, a total interpretation M is a model for Π if, for each $r \in \Pi$, r is satisfied with respect to M.

Definition 21. A set X $\subseteq B_\Pi$ is a **stable model** (or **answer set**) for a **positive program** Π if it is a minimal set (w.r.t. set inclusion) among the models for Π.

Definition 22. The **FLP-reduct** of a ground program Π w.r.t. an Interpretation X is the ground program Π^X obtained from Π by deleting each rule $r \in \Pi$ whose body is not satisfied w.r.t. X.

Definition 23. A **stable model** (or **answer set**) of a **program** Π is a model X of Π such that X is a stable model of Π^X.

One important property of answer sets is supportedness. In fact, given an interpretation I, a positive literal a is supported in I if and only if there is a rule r such that B(r) is true, $a \in$ H(r) and a is the only true atom in H(r).

3.3 Knowledge Representation and Reasoning in ASP

ASP can be used to encode problems in a simple, declarative way. Indeed, the expressive power of disjunctive rules allows for expressing problems which belong to complexity classes harder than NP, and the separation of the encoding program from an input database permits obtaining uniform solutions over a wide variety of instances. For those reasons, ASP has been exploited in several domains, ranging from classical deductive databases to artificial intelligence. In the following sections, we illustrate the usage of ASP as a tool for knowledge representation and reasoning by example. In particular, we present three well-known problems: *Reachability*, *Hamiltonian Path* and *Ramsey Numbers*. The problem *Reachability* shows how ASP can deal with a problem motivated by classical deductive database applications; the problems *Hamiltonian Path* and *Ramsey Numbers* show how harder problems can be easily encoded in ASP. Note that some of the problems described in the following were also used as benchmarks for the system that has been developed for this thesis (see Chapter 5).

3.3.1 Reachability

Given a finite directed graph $G = (V, A)$, we want to compute all pairs of nodes $(a, b) \in V \times V$ such that b is reachable from a through a nonempty sequence of arcs in A. In different terms, the problem amounts to computing the transitive closure of the relation A.

An arc of G is encoded by using the predicate $arc(X, Y)$, which means that X and Y are two nodes and an arc from X to Y exists. It is important to note that the set of nodes V is not explicitly represented. Hence, we encode the graph G only by enumerating the arcs.

The following program then defines a relation `reachable(X,Y)` containing all atoms of the form `reachable(a,b)` such that b is reachable from a through the arcs of the input graph G:

```
r1: reachable(X,Y) :- arc(X,Y).
r2: reachable(X,Y) :- arc(X,U), reachable(U,Y).
```

The first rule states that node Y is reachable from node X if there is an arc in the graph from X to Y, while the second rule reprents the transitive closure by stating that node Y is reachable from node X if there exists a node U such that U is directly reachable from X (there is an arc from X to U) and Y is reachable from U.

Example 4 (Reachability)**.** Consider a graph G represented by the following facts:

```
arc(1,2).    arc(2,3).    arc(3,4).
```

The rule *r1* infers the truth of the literals `reachable(1,2)`, `reachable(2,3)` and `reachable(3,4)`. The rule *r2* infers the truth of the literals `reachable(1,3)`, `reachable(2,4)` and `reachable(1,4)`. Hence, the only answer set of the program is
{ `reachable(1,2)`, `reachable(2,3)`, `reachable(3,4)`, `reachable(1,3)`, `reachable(2,4)`, `reachable(1,4)`, `arc(1,2)`, `arc(2,3)`, `arc(3,4)` }.

3.3.2 Hamiltonian Path

Given a finite directed graph $G = (V, A)$ and a node $a \in V$ of this graph, does there exist a path in G starting at a and passing through each node in V exactly once?

This is a classical NP-complete problem in graph theory. Suppose that the graph G is specified by using facts over predicates *node* and *arc*, and the starting node a is specified by the predicate *start*. Then, the following program solves the *Hamiltonian Path* problem:

```
r1: inPath(X,Y) v outPath(X,Y) :- arc(X,Y).
r2: reached(X) :- start(X).
r3: reached(X) :- reached(Y), inPath(Y,X).
r4: :- inPath(X,Y), inPath(X,Y1), Y != Y1.
r5: :- inPath(X,Y), inPath(X1,Y), X != X1.
r6: :- node(X), not reached(X), not start(X).
```

This program shows how simple it is to encode a NP-problem with the "Guess&Check" methodology originally introduced in [53] and refined in [3]. 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.

In the example the guessing part is composed by the rule *r1* while the rules *r2*, *r3*, *r4*, *r5*, *r6* compose the checking part. In fact, *r1* guesses a subset

S of arcs belonging to the path and the rest of the program checks whether S is a Hamiltonian Path. The auxiliary predicate `reached` is defined, which specifies the set of nodes which are reached from the starting node. In the checking part, the constraint *r4* ensures that in S there are no two arcs starting at the same node, and the constraint *r5* ensures that in S there are no two arcs ending in the same node. The constraint *r6* enforces that all nodes in the graph are reached from the starting node in the subgraph induced by S.

3.3.3 Ramsey Numbers

The Ramsey number $R(k, m)$ is the smallest integer n such that, no matter how we color the arcs of the complete undirected graph (clique) with n nodes using two colors, say red and blue, there is a red clique with k nodes (a red k-clique) or a blue clique with m nodes (a blue m-clique).

Ramsey numbers exist for all pairs of positive integers k and m [64]. We next show a program Π_{ra} that allows us to decide whether a given integer n is *not* the Ramsey Number $R(3, 4)$. By varying the input number n, we can determine $R(3, 4)$, as described below. Let F_{ra} be the collection of facts for input predicates *node* and *arc* encoding a complete graph with n nodes. Π_{ra} is the following program:

```
r1: blue(X,Y) v red(X,Y) :- arc(X,Y).
r2: :- red(X,Y), red(X,Z), red(Y,Z).
r3: :- blue(X,Y),blue(X,Z), blue(Y,Z), blue(X,W),
       blue(Y,W), blue(Z,W).
```

The guessing part of Π_{ra} is composed by the rule *r1*, which guesses a color for each edge; while the checking part is composed by rules *r2* and *r3*, which eliminate the colorings containing a red clique (i.e., a complete graph) with 3 nodes and the colorings containing a blue clique with 4 nodes, respectively. The program $\Pi_{ra} \cup F_{ra}$ has an answer set if and only if there is a coloring of the edges of the complete graph on n nodes containing no red clique of size 3 and no blue clique of size 4. Thus, if there is an answer set for a particular n, then n is *not* $R(3, 4)$, that is, $n < R(3, 4)$. On the other hand, if $\mathcal{P}_{ra} \cup F_{ra}$ has no answer set, then $n \geq R(3, 4)$. Thus, the smallest n such that no answer set is found is the Ramsey number $R(3, 4)$ [54].

3.4 Architecture of an ASP System and Evaluation Algorithms

The architecture of an answer set solver 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 the Model Generator, computes stable model candidates of the program. Stable model candidates are, in turn checked by the third module, called Model Checker, which verifies that, models are actually stable (i.e., they are answer sets). This three modules are briefly described in this section.

Given an input program Π, the Grounder efficiently generates an intelligent ground instantiation of Π that has the same answer sets of the theoretical instantiation, but is much smaller in general [3]. 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 Π, the Grounder generates ground instances of rules containing only atoms which can possibly be derived from Π, and thus (if possible) avoiding the combinatorial explosion which can be achieved by naively considering all the atoms in the Herbrand Base [65]. 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, which is focused on the Model Generator module. Therefore, we assume that the logic program in input is a ground program and we refer the reader to [3] for an accurate description.

The Model Generator usually implements an enhancement of the DPLL algorithm (see Section 2.1). A pseudo-code description of a generic Model Generator is the following:

```
bool MG( Interpretation I )
{
    if( !propagation( I ) ) then { learning(); return false; }
    if( noUndefinedAtoms( I ) ) then return isStableModel( I );
    Atom A = chooseAtomUsingHeuristic();
    if( MG( I ∪ { A } ) ) then return true;
    else return MG( I ∪ { not A } );
}
```

At the beginning, the MG function is invoked with an empty Interpretation I. MG first calls the propagation function, which extends I with the literals that can be deterministically inferred. Moreover, the propagation function returns true if there are no conflicts, false otherwise. A conflict happens when the interpretation is not consistent, i.e. if there is a literal l such that $l \in$ I and $\neg l \in$ I. If the propagation function returns false, the function learning() is invoked before returning false. The function learning computes the atoms that are the reason of the conflict and insert a constraint into the program to avoid in the future search paths leading to the same conflict. When the interpretation is total, i.e. there are no more undefined atoms, MG invokes the function isStableModel(I), which checks if I is an answer set.

It is worth to note that DPLL and learning techniques were originally introduced by efficient SAT solvers (see Section 2.1 and Section 2.2, respectively). 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. In the following sections we show the particular requirement that the basic components of a Model Generator have to satisfy.

3.4.1 Propagation

The role of propagation is similar to the unit propagation procedure in the DPLL algorithm, but it is more complex than unit propagation because it implements a set of inference rules [14]. We will discuss this later in Section 4.5.

Forward Inference derives an atom as true if it occurs in the head of a rule in which all other head atoms are false and the body is true.

Kripke-Kleene Negation derives an atom as false if no rule can support it.

Contraposition for True Heads applies when a true atom has only one rule that can support it; and in this case derives all other head atoms as false, atoms in the positive body as true and atoms in the negative body as false.

Contraposition for False Heads applies when a rule with false head has only one undefined literal in the body and derives it as false.

Well-founded Negation derives all members of the greatest unfounded set to false.

3.4.2 Learning

Learning is the process that adds some constraints to the program to avoid meeting the conflicts already encountered in the search. It is very important for pruning the backtracking tree and to implement an efficient learning-based heuristic. It is a technique introduced by SAT solvers, and it was described already in detail in Section 2.2.

3.4.3 Heuristics

One of the most important feature in a Model Generator is the selection of a literal when there are no deterministic inferences left but there are still undefined atoms. It is important to point out that there is no overall optimal strategy. In fact, strategies which are particularly performant on some domains may perform very badly for other domains. For this reason, some heuristic must be adopted, the quality of which can only be assessed empirically.

3.4.4 Stable Model Checking

The goal of model checking is to verify whether an input model M is an answer set for an input program Π. This task is very hard in general, because checking the stability of a model is well-known to be co-NP-complete [58]. However, the task is polynomial for the class of Head-Cycle Free (HCF) programs [59]. To this purpose, WASP (as well as the DLV system) adopts a strategy that performs a polynomial time check for easy problems (HCF); in case of hard problems (non-HCF), an additional check is carried out by translating the program into a SAT formula and checking whether it is unsatisfiable. A full description of a Model Checker module is out of the scope of this thesis, interested readers may find a description in [52].

Chapter 4

WASP

In this chapter we report a description of WASP. We start reporting a general description of the architecture in Section 4.1. In Section 4.2 and in Section 4.3 we describe the input format and the data structures, respectively. In Sections 4.4 to 4.7 we explore in detail the Model Generator module, describing the main algorithm and the techniques used by WASP. In particular, we report the description of the learning process in Section 4.6 and of the heuristic used to choose a literal in Section 4.7. Finally, in Section 4.8 we report other solvers related to WASP, pointing out the differences and similarities between our solver and other solvers.

4.1 The Architecture of WASP

In this section we provide a full description of the architecture of WASP, which is shown in Figure 4.1. WASP is composed by three modules. The first module is the input processor, which takes in input a ground program and produces a numerical format, described in detail in Section 4.2. The core of WASP is the Model Generator module, which computes stable model candidates using techniques described in the following of this chapter. The full description of the Model Generator is reported in Section 4.4. When the Model Generator produces an answer set candidate, a stability check is required to verify that the produced model is also an answer set. The stability check is carried out using the techniques traditionally exploited by the ASP solver DLV [52]. In particular, WASP includes an implementation of the techniques originally described in [52] based on the SAT solver minisat and proposed in [60].

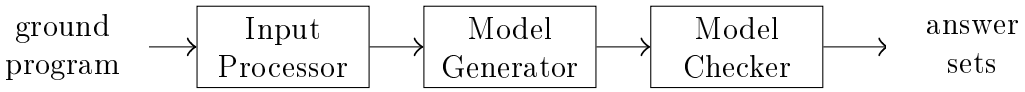


Figure 4.1: WASP architecture

4.2 Input Processor

The input format accepted by the Model Generator of WASP is numeric. To this end, we devised a file format in which each component of a ground program is represented by means of numbers. The Input processor takes as input a ground ASP program and produces a numerically-encoded version of the input, which is compact, non-redundant and can be used to efficiently populate the internal data structures. Each line of an encoded file starts with a number representing the corresponding component of a program. The components are: atoms, bodies, heads, rules and aggregate atoms. The specification of components is followed by the dictionary (*names table*) associating to each atom identifier the corresponding string found in the input program. In particular:

- **0** is a special identifier and it is used only in the first line of each file followed by the version number of the encoding.
- **1** represents a rule, and is followed by two numbers: the numeric head identifier and the body identifier. We distinguish disjunctive heads from non-disjunctive ones. A disjunctive head is represented by a negative number, while a single-atom head is represented by a positive number that is the ID of the atom in the head. At last we have 0 for an empty head. A similar approach is used for bodies. We distinguish unary bodies from "complex" ones indicating a single positive body by the ID of the atom in the body and a "complex" body by a negative body identifier. A body identifier of 0 denotes an empty body.
- **4** represents a disjunctive head. After its identifier we find another number that indicates of how many atoms the head is composed. Then, the list of atom IDs follows.
- **5** represents a "complex" body. The first number is its identifier, followed by the number of atoms in the body. After that, we find a list of positive and negative atom IDs. The sign of the number is used to indicate the atom polarity.

- **10** represents the aggregate *#count*. The first number after 10 is the ID of the aggregate atom, followed by two numbers representing the range in which the aggregate function holds. After that, we find the number of atoms in the aggregate and then we have a list of atom IDs.
- **11** represents the aggregate *#sum*. The first number after 11 is the ID of the aggregate atom, followed by two numbers representing the range in which the aggregate function holds. After that, we find the number of atoms in the aggregate and then we have a list of pairs representing the atom ID and the value associated to it.
- **-1** represents a comment.
- **-2** is used to denote the start of the dictionary(*names table*) that assign strings to IDs representing atoms. Each line of the table contains an ID and a string separated by two tabular spaces.

Example 5 (A program encoded in the numeric format)**.** The encoding of the following program:

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

is reported below.

```
0 1 0                                  1.0 is the current version
-1 Parsed program:                     This is a comment
1 0 -1                                 Encoding of rule 1
5 1 2 -2 1                             The body of the rule 1
1 2 5                                  Encoding of rule 3
1 -1 0                                 Encoding of rule 2
4 1 2 3 4                              The head of the rule 2
1 5 0                                  Encoding of rule 4, it is a fact
-1 Names Table
-2                                     Starting the names table
0 #false                               The id 0 is not used
1 c                                    Id and name of the atom c
2 d                                    Id and name of the atom d
3 a                                    Id and name of the atom a
4 b                                    Id and name of the atom b
5 e                                    Id and name of the atom e
-1 Totale execution time:  0.000000s   Execution time
```

4.3 Data Structures

An important issue in a solver are the data structures that are used to store information about rules, atoms, bodies and heads. In particular, it is important to have bidirectional access from a component to another. For this reason we decided to have separated tables for rules, atoms, bodies and heads. In fact, we store a list of the program rules, each rule contains a pointer to the head and a pointer to the body. Moreover, we store a list of all atoms appearing in the program, each atom contains a list of rules, positive and negative bodies, disjunctive heads and unary heads where it appears. For each body and head we store the information on the atoms that appear in it. Finally, we have a list of atom truth values representing the current interpretation. All atoms in the interpretation are initialized to the undefined truth value.

During the computation (and in particular during the propagation step; this will be clarified next) it is necessary to update information about truth values of atoms and their consequences in terms of rules satisfaction. In particular, often we need to know if either a body or a head is true or false. To do that, we exploit the concept of watch.

A watch is a pointer to an undefined atom. For each body and head, we store two watches pointing, respectively, to the first and last undefined atom. Those watches are useful for several reasons. The first one is that, during propagation, we have to update a body or a head only if the current atom is watched. Moreover, they are useful to determine easily whether bodies and heads are true or false. Finally, if there is only one undefined atom in the structure, we can access it directly.

Watches are also used in the interpretation to index the first and last undefined atom. They are useful to recognize easily when there are no undefined atoms and to restrict the range between the first and the last undefined atom. These watches are updated only if the truth value for a watched atom changes during the computation.

4.4 Model Generator

In this section we sketch the main model generator function MG (shown in Figure 4.2), which performs learning and restart techniques. MG is similar to the DPLL procedure in SAT solvers. For the sake of clarity we considered a simplified procedure. For example, the version described here computes only one answer set, but modifying it to compute all or n stable models is straightforward.

In the sequel, Π will refer to the input program. Initially, the MG function is invoked with $I = \emptyset$, and bj_level = -1 (but it will become 0 immediately), and the global variable numberOfConflicts is set to 0. MG returns true if the program Π has an answer set, and in this case sets I to the computed answer set; otherwise it returns false.

MG first calls a function Propagate, which extends I with those literals that can be deterministically inferred, and keeps track of the reason of each inference by building a representation of the so-called implication graph [33]. Propagate is similar to unit propagation as employed by SAT solvers, but exploits the particulars of ASP for making further inferences (e.g., it uses the knowledge that every answer set is a minimal model). Propagate, described in more detail in Section 4.5, returns false if an inconsistency (or conflict) is detected (i.e., the complement of a true literal is inferred to be true), true otherwise.

If Propagate returns true and no undefined atom is left in I, MG invokes CheckModel [1] to verify that the current total interpretation is also an answer set; the CheckModel function implements the techniques described in [52]. If the stability check succeeds, MG returns true. If Propagate returned true but I is still partial, an undefined literal L is selected according to a heuristic criterion and MG is recursively called. The atom L corresponds to a *branching variable* in SAT solvers.

If Propagate returns false, the function ResolveConflict is called, which calculates the Unique Implication Point (UIP) of the Implication Graph (see Sections 4.6.1 and 4.6.2), and exploits it to *learn* a constraint representing the inconsistency (see Section 4.6.3), which is added to the input program. As a by-product, ResolveConflict returns the recursion level to go back to (backjumping) in order to continue the search at the first branch of the search tree that is free of the just-detected conflict.

After a certain number of conflicts, ResolveConflict may restart the entire search if the total number of conflicts found during the search reached a certain threshold. It is important to note that after each restart MG works on a program composed of the original input program and the learned constraints. Our restart policy is based on the sequence of thresholds (32, 32, 64, 32, 32, 64, 128, ...) introduced in [25].

If the recursive call returned true, MG just returns true as well. If it returned false, the corresponding branch is inconsistent, bj_level is set to the recursion level to backtrack or backjump to. Now, if bj_level is less than the current level, this indicates a backjump, and we return. If not, then we have reached the level to go to, and the search continues.

[1] This is a coNP-complete task in case of general disjunctive ASP programs.

```
bool MG (Interpretation& I, int& bj_level )
    int curr_level = ++ bj_level;

    if ( !Propagate( I ) )
      bj_level = ResolveConflict();
      return false;
    if ( "no atom is undefined in I" )
      if ( CheckModel( I ) ) return true;
      else
        bj_level = ResolveConflict();
        return false;

    Select an undefined atom A using a heuristic;

    if ( MG( I ∪ {A}, bj_level ) ) return true;
    if (bj_level < curr_level) return false;

    if ( MG( I ∪ {¬A}, bj_level ) ) return true;
    if (bj_level < curr_level) return false;

    return false;

int  ResolveConflict()
    int level = calculateFirstUIP();
    learning();
    if(inRestartSequence(numberOfConflict)) return 0;
    return level;
```

Figure 4.2: Computation of answer sets

4.5 Propagation Function

One of the most important features of an ASP solver is a good implementation of a propagation function. Such a function computes the deterministic consequences of a partial Interpretation. The role of a propagation function is similar to that of unit propagation in the DPLL procedure. However, a propagation function is typically more complex than unit propagation. In fact, while unit propagation is based on just one inference rule, a propagation function usually implements a set of such inference rules. Concerning WASP, the rules implemented in its propagation function are the following:

1) Forward Inference;

2) Kripke-Kleene Negation;

3) Contraposition for True Heads;

4) Contraposition for False Heads;

5) Well-founded Negation.

In the following of this chapter we refer to an interpretation I as a partial Interpretation.

4.5.1 Forward Inference

This inference rule derives the truth of an atom a appearing in the head of a rule r such that the body of r is true and all head atoms of r are false, with the exception of a which is undefined. More formally, given a rule r, if there is an undefined atom $a \in H(r)$ such that:

i) for each $b \in B(r) : b \in I$, and

ii) for each $a_i \in (H(r) \setminus \{a\}) : \neg a_i \in I$,

then we infer $a \in I$.

Example 6 (Forward Inference)**.** Consider the following subprogram:

```
r1: a v b :- c.
```

Suppose I = {¬b, c}. After forward inference, we obtain I = I ∪ {a}, that is, atom a is inferred as true. In fact, the body of *r1* is true and the only head atom different from a, namely b, is false.

4.5.2 Kripke-Kleene Negation

In this case, we derive the falsity of an atom A if all rules in which it occurs as a head atom are satisfied w.r.t. I. Formally, given an atom a, if for all rules r such that $a \in H(r)$:

i) there is $b \in B(r)$ such that $\neg b \in I$, or

ii) there is $h \in H(r) \setminus \{a\}$ such that $h \in I$,

then we infer $\neg a \in I$.

Example 7 (Kripke-Kleene Negation)**.** Consider the following subprogram:

```
r1: a v b :- c.
r2: a v d.
r3: a :- b.
```

Suppose I = {¬b, d, ¬c}. By applying Kripke-Kleene Negation, we obtain I = I ∪ {¬a}, that is, atom a is inferred as false. In fact, rule *r1* and rule *r3* cannot support the atom a because their bodies are false, and rule *r2* cannot support the atom a because the atom d is true.

4.5.3 Contraposition for True Heads

This inference rule is based on a well known theorem stating that every true atom in a stable model must be supported. More formally, let Π be a program and M a model for Π. If M is a stable model for Π, then all atoms $a \in M$ are supported w.r.t. M. Thus, if an atom a is true and there is only one rule r that can support a, we infer the truth of all body literals and the falsity of all other head atoms of r. More formally, given an atom $a \in I$ and a rule r such that $a \in H(r)$, if for each rule $r' \neq r$ such that $a \in H(r')$:

i) there is $b' \in B(r') : \neg b' \in I$, or

ii) there is $c' \in H(r') \setminus \{a\} : c' \notin$ I,

then for each $c \in H(r) \setminus \{a\}$ such that $\neg c \notin I$ infer $\neg c \in I$, and for each $b \in B(r)$ such that $b \notin I$ infer $b \in I$.

Example 8 (Contraposition for True Heads)**.** Consider the program:

```
r1: a v b :- not c, d.
r2: a v d.
r3: a :- e.
```

Suppose I = {a, d, ¬e}. By applying Contraposition for True Heads, we obtain I = I ∪ {¬b, ¬c}, that is, b and c are inferred as false. In fact, the only rule that can support a is *r1*, hence we force this rule to support a.

4.5.4 Contraposition for False Heads

This inference rule derives an atom false if it is the only undefined atom in the body of a rule r, the other atoms in the body of r are true and the head of r is false. Formally, given a rule r and an atom $a \in$ B(r) such that:

i) for all $h \in$ H(r) : $\neg h \in I$, and

ii) for all $b \in$ B(r) $\setminus \{a\} : b \in I$,

we infer $\neg a \in I$.

Example 9 (Contraposition for False Heads)**.** Consider the following rule:

```
r1: a v b :- not c, d.
```

Suppose I = {¬a, ¬b, ¬c}. After the propagation, we obtain I = I ∪ {¬d}, that is, d is inferred as false. Indeed, the head of $r1$ is false and the only undefined atom in the body of *r1* is d. Thus, in order to satisfy the rule *r1*, we have to infer d to be false.

4.5.5 Well-founded Negation

In this inference rule, we use the knowledge that each stable model M does not contain any atom which is in some unfounded set w.r.t. the stable model M. During propagation, if we determine that there is a set of atoms that is unfounded w.r.t. the current interpretation, we can infer the falsity of those atoms belonging to the set. It is important to emphasize that, for complexity reasons, we apply this inference rule only on recursive components which are head cycle free.

Definition 24 (Unfounded Set [19])**.** Let I be an interpretation for a ground program Π. A set of atoms $U \subseteq B_\Pi$ is an unfounded set for Π w.r.t. I if, for each $a \in U$ and for each rule $r \in \Pi$ such that $a \in H(r)$, at least one of the following conditions holds:

i) $B(r) \cap \neg I \neq \emptyset$, that is, the body of r is false w.r.t I;

ii) $B^+(r) \cap U \neq \emptyset$, that is, some positive body literal belongs to U;

iii) $(H(r) \setminus U) \cap I) \neq \emptyset$, that is, an atom in the head of r, distinct from a and other elements in U, is true w.r.t. I.

This inference rule derives the falsity of all atoms belonging to some unfounded set.

Example 10 (Well-founded Negation)**.** Consider the following subprogram:

```
r1: a v b.
r2: a :- not c.
r3: a :- d.
r4: d :- a.
```

Suppose I = {b, c}. The set U = {a, d} is unfounded. In fact, the atom a appears in the head of three rules: *r1*, *r2* and *r3*; and for *r1* the condition iii) holds, for *r2* the condition i) holds, for *r3* the condition ii) holds; moreover, the atom d appears in the of the rule *r4* and for this rule the condition ii) holds. Thus, a and d are inferred to be false.

4.5.6 Aggregate Propagation

We next report the propagation for each aggregate supported by WASP. Without loss of generality, we assume that aggregate atoms are of the form $f(A) > k$, where the aggregate set A only contains pairs of the form $\langle \bar{t} : a \rangle$. In fact, WASP internally rewrites the input program Π to obtain this simplified

form. In particular, each aggregate atom $f(S) \prec T$ (with $\prec \in \{=, <, \leq, >, \geq, \}$) occurring in Π is first transformed such that only the operator $>$ is used; then, for each $\langle \bar{t} : conj \rangle \in S$, $conj$ is replaced by a new atom $aux_{f(S)>T}(\bar{v})$, and the rule $aux_{f(S)>T}(\bar{v}) :- conj$ is added to the program, where $\bar{v}$ are the variables occurring in $conj$. This means that conjunctions in S are replaced by freshly introduced auxiliary atoms, along with a rule defining the auxiliary atom by means of the conjunction. This transformation simplifies the implementation of propagation with a negligible overhead. In the remainder of this section, each inference rule regarding aggregates is explained in detail. In particular, WASP implements two inference rules, namely forward inference and backward inference.

Definition 25. Given a ground set A and an interpretation I, let T_A be the set $\{\langle \bar{t}_i : a_i \rangle \in A$ such that $a_i \in I\}$.

Definition 26. Given a ground set A and an interpretation I, let F_A be the set $\{\langle \bar{t}_i : a_i \rangle \in A$ such that $\neg a_i \in I\}$.

Definition 27. Given a ground set A, let $\Phi(A)$ denote $\sum_{[v|\langle v,\bar{t} \,:\, a\rangle \,\in\, A]} v$.

Forward Inference

In this rule, we infer the truth value of an aggregate atom $f(A) > k$ because some atom in A is true or false w.r.t. I.

i) $\#count\{A\} > k$ is inferred to be false if there exists a set $A' \subseteq A$ such that, for each $\langle \bar{t} \,:\, a\rangle \in A'$, a is false in I and $|A| - |A'| \leq k$.

ii) $\#count\{A\} > k$ is inferred to be true if there exists a set $A' \subseteq A$ such that, for each $\langle \bar{t} \,:\, a\rangle \in A'$, a is true in I and $|A'| > k$.

iii) $\#sum\{A\} > k$ is inferred to be false if there exists a set $A' \subseteq A$ such that, for each $\langle v, \bar{t} \,:\, a\rangle \in A'$, a is false in I and $\Phi(A) - \Phi(A') \leq k$.

iv) $\#sum\{A\} > k$ is inferred to be true if there exists a set $A' \subseteq A$ such that, for each $\langle v, \bar{t} \,:\, a\rangle \in A'$, a is true in I and $\Phi(A') > k$.

Example 11 (Aggregate Propagation: Forward inference - Count)**.** Consider the following rule:

```
r1: c :- #count{<1 : a(1)>, <2 : a(2)>, <3 : a(3)>} > 1.
```

Suppose I = {a(1), a(2),¬b(1), ¬b(2), ¬d, ¬e}. By applying this inference rule, we obtain $I = I \cup \{c\}$, that is, c is inferred as true. In fact, we have a set $A' = \{a(1), a(2)\}$ such that *a(1)* and *a(2)* are true and $|A'| > |A|$. Thus, aggregate forward propagation infers c as true.

Example 12 (Aggregate Propagation: Forward inference - Sum)**.** Consider the following rule:

```
r1: c :- not #sum{<1 : a(1)>, <2 : a(2)>, <3 : a(3)>} > 4.
```

Suppose I = {¬a(3), b(3)}. After the propagation, we obtain I = I ∪ {c}, that is, c is inferred as true. In fact, we have a set $A' = \{a(3)\}$ such that $a(3)$ is false. Moreover, we have $\Phi(A) = 6$ and $\Phi(A') = 3$. Since, $\Phi(A) - \Phi(A') = 3 \leq 4$, then aggregate $\#sum\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} > 4$ is false, and aggregate forward propagation infers c as true.

Backward Inference

In this rule, we infer the truth value of some atom in A because an aggregate atom $f(A) > k$ is true or false. This happens when an aggregate atom is derived true or false and there is only one way to satisfy it.

i) Let $\#count\{A\} > k$ be an aggregate. If $\#count\{A\} > k$ is true and $|A| - |F_A| = k + 1$, then each undefined atom a_i such that $\langle \bar{t}_i : a_i\rangle \in A$ is inferred to be true.

ii) Let $\#count\{A\} > k$ be an aggregate. If $\#count\{A\} > k$ is false and $|T_A| = k$, then each undefined atom a_i such that $\langle \bar{t}_i : a_i\rangle \in A$ is inferred to be false.

iii) Let $\#sum\{A\} > k$ be an aggregate. If $\#sum\{A\} > k$ is true and $\Phi(A) - \Phi(F_A) = k + 1$, then each undefined atom a_i such that $\langle \bar{t}_i : a_i\rangle \in A$ is inferred to be true.

iv) Let $\#sum\{A\} > k$ be an aggregate. If $\#sum\{A\} > k$ is false and $\Phi(T_A) = k$, then each undefined atom a_i such that $\langle \bar{t}_i : a_i\rangle \in A$ is inferred to be false.

Example 13 (Aggregates Propagation: Backward inference - Count)**.** Consider the following rule:

```
r1: :- #count{<1 : a(1)>,<2 : a(2)>, <3 : a(3)>} > 1.
```

Suppose I = {a(1), ¬b(1), ¬d} and $\#count\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} > 1$ has to be false. In this case, we have that $T_A = \{a(1)\}$ and $|T_A| = 1 = k$, then we can infer that $a(2)$ and $a(3)$ have to be false.

Example 14 (Aggregates Propagation: Backward inference - Sum)**.** Consider the program:

```
r1: a(1) v b(1).
r2: a(2) v b(2).
r3: a(3) v b(3).
r3: a(1) :- not d.
r4: a(2) :- e.
r5: :- not #sum{<1 : a(1)>,<2 : a(2)>, <3 : a(3)>} > 3.
```

Suppose I = {¬a(2), b(2), ¬e} and $\#sum\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} > 3$ has to be true. We have $F_A = a(2)$, $\Phi(A) = 1 + 2 + 3 = 6$ and $\Phi(F_A) = 2$. Since $\Phi(A) - \Phi(F_A) = 6 - 2 = k + 1 = 4$, there is a unique way to satisfy the aggregate, that is, $a(1)$ and $a(3)$ must be true. Thus, $a(1)$ and $a(3)$ are inferred as true.

4.6 Learning

Learning means acquiring information that avoids arriving again at a conflict that was already encountered during the search. Our learning schema is based on the concept of the first Unique Implication Point (UIP). In the following we describe the Implication Graph that is useful in order to calculate the first UIP.

4.6.1 Implication Graph

During the propagation of deterministic inferences, implication relationships among atoms are stored in a graph $\mathcal{G}$ called Implication Graph. This graph has a node a for each atom a such that a has been assigned the truth value true, and a node $\neg a$ for each atom a such that a has been assigned the truth value false. Each node of the graph is associated with a *decision level*, which is set to the level of the backtracking tree when a is inferred. Moreover, $\mathcal{G}$ has a directed arc connecting a node a to a node a' whenever a is one of the reasons that lead to the derivation of a'. Note that $\mathcal{G}$ will contain at most one node for each literal of the program, unless a conflict is derived, and that $\mathcal{G}$ is built during the propagation. To improve readability in the following examples we omit decision levels of atom in the Implication Graph.

Forward Inference

Recall that forward inference derives the truth of an atom a appearing in the head of a rule r such that the body of r is true and all head atoms of r but a are false. The Implication Graph $\mathcal{G}$ is updated as follows. Let r be of the

form (3.1) and let a_i be the undefined atom in $H(r)$. The following elements are added to $\mathcal{G}$:

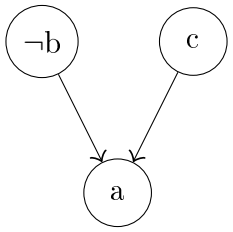
i) a node a_i;

ii) arcs (b_j, a_i) for all $j = 1, \ldots, k$;

iii) arcs $(\neg b_j, a_i)$ for all $j = k+1, \ldots, m$;

iv) arcs $(\neg a_j, a_i)$ for all $j = 1, \ldots, n$ such that $j \neq i$.

Example 15 (Forward inference: Implication Graph for Example 6)**.** In Example 6 we considered a rule `r1: a v b :- c.` and a partial interpretation I = {¬b, c}, and we inferred the atom a as true. The elements added to the Implication Graph are the following:



Kripke-Kleene Negation

Recall that this inference rule derives the falsity of an atom a if no rules can support it. The Implication Graph $\mathcal{G}$ is updated by introducing a node $\neg a$. Moreover, an arc is introduced in $\mathcal{G}$ for each rule r with $a \in H(r)$ as follows: let a_1 be the first atom (in chronological order of derivation) that satisfied r. The arc introduced for r is:

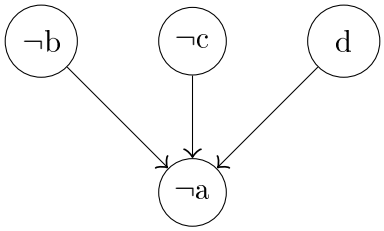
1) $(\neg a_1, \neg a)$ if $a_1 \in B(r)$; otherwise,

2) $(a_1, \neg a)$ if $a_1 \in H(r)$.

Example 16 (Kripke-Kleene Negation: Implication Graph for Example 7)**.** In example 7 we considered the following subprogram:

```
r1: a v b :- c.
r2: a v d.
r3: a :- b.
```

and a partial interpretation I = {¬b, d, ¬c} and we inferred the atom a as false. The elements added to the Implication Graph are the following:



Contraposition for True Heads

Recall that this inference rule imposes the truth or the falsity of the atoms in a rule r, in order to support an atom a in the head of r, if r is the only rule that can support a. another atom a, if there is only one way for the atom a to be supported. Concerning the Implication Graph $\mathcal{G}$, the following new nodes and arcs are introduced:

i) b, for each $b \in B(r)$;

ii) $\neg b$, for each $b \in H(r) \setminus \{a\}$;

iii) (a, b), for each new node b added in i) and ii).

Moreover, for each rule r' such that $a \in H(r')$, let a_1 be the first literal (in chronological order of derivation) that satisfied r'. Let b be a node added in i) and ii).

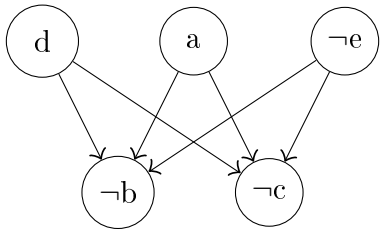
i) If $a_1 \in B(r')$, an arc $(\neg a_1, b)$ is added to $\mathcal{G}$; otherwise,

ii) if $a_1 \in H(r')$, an arc (a_1, b) is added to $\mathcal{G}$.

Example 17 (Contraposition for True Heads: Implication Graph in Example 8)**.** In Example 8 we considered the following subprogram:

```
r1: a v b :- not c, d.
r2: a v d.
r3: a :- e.
```

and a partial interpretation I = {a, d, ¬e} and we inferred the atoms b and c as false. The elements added to the Implication Graph are the following:



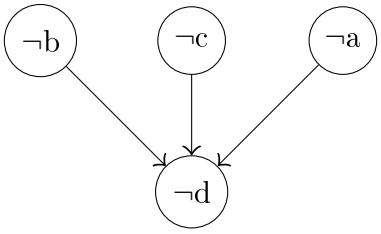
Contraposition for False Heads

Recall that this inference rule derives an atom false if it is the only undefined atom in the body of a rule r, the other literals in the body of r are true and the head of r is false. Concerning the Implication Graph $\mathcal{G}$, a node $\neg a$ is added, where a is the false literal in the body. Moreover, the following arcs are added to $\mathcal{G}$:

i) $(b, \neg a)$ for each $b \in B(r) \setminus \{a\}$;

ii) $(\neg b, \neg a)$ for each $b \in H(r)$.

Example 18 (Contraposition for False Heads: Implication Graph in Example 9)**.** In Example 9 we considered a rule `r1: a v b :- not c, d.` and a partial interpretation I = {¬a, ¬b, ¬c} and we inferred the atom d as false. The elements added to the Implication Graph are the following:



Well-founded Negation

Remind that this inference rule derives the falsity of all atoms belonging to some unfounded set with respect to the current Interpretation. Concerning the Implication Graph $\mathcal{G}$, for each $a \in X$, where a is an undefined atom and X is the greatest unfounded set, a node $\neg a$ is added. Arcs are introduced according to the following schema: for each atom $a \in X$ and for each rule r where $a \in H(r)$, if r is satisfied, let $a_1 \neq a$ be the first atom (in chronological order of derivation) that satisfied r.

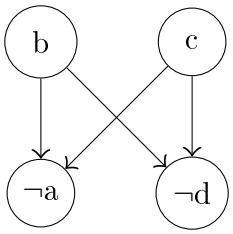
i) If $a_1 \in B(r)$, an arc $(\neg a_1, \neg a)$ is added to $\mathcal{G}$.

ii) If $a_1 \in H(r)$, an arc $(a_1, \neg a)$ is added to $\mathcal{G}$.

Example 19 (Well-founded Negation: Implication Graph in Example 10)**.** In Example 10 we considered the following subprogram:

```
r1: a v b.
r2: a :- not c.
r3: a :- d.
r4: d :- a.
```

and a partial interpretation I = {b, c}, and we inferred atoms a and d as false because they belong to the unfounded set U = {a, d}. The elements added to the Implication Graph are the following:



Aggregates Forward Inference

We recall that this inference rule assigns a truth value for an aggregate atom. Given an aggregate atom $f(A) > k$, if the condition of the aggregate function holds, we infer the truth value of $f(A) > k$; otherwise, if it cannot hold, we infer the falsity of $f(A) > k$. Concerning the Implication Graph $\mathcal{G}$, we have two different cases:

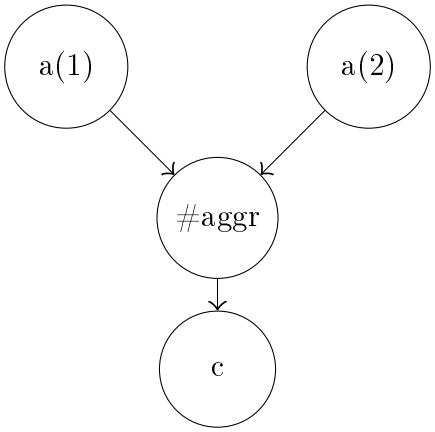
i) *Aggregate function holds.* In this case, a node $\#aggr$ is added, where $\#aggr$ is an auxiliary atom representing the aggregate. Moreover, for each true atom a_i such that $\langle \bar{t}_i : a_i \rangle \in A$, an arc $(a_i, \#aggr)$ is added to $\mathcal{G}$.

ii) *Aggregate function does not hold.* In this case, a node $\neg\#aggr$ is added, where $\#aggr$ is an auxiliary atom representing the aggregate. Moreover, for each false atom a_i such that $\langle \bar{t}_i : a_i \rangle \in A$, an arc $(\neg a_i, \neg\#aggr)$ is added to $\mathcal{G}$.

Example 20 (Aggregates Forward Inference: Implication Graph for Example 11)**.** . In Example 11 we considered the following rule:

```
r1:  c :- #count{<1 :  a(1)>, <2 :  a(2)>, <3 :  a(3)>} > 1.
```

and a partial interpretation I = {a(1), a(2), ¬b(1), ¬b(2), ¬d, ¬e}, and the atom c was inferred as true. The elements added to the Implication Graph are the following:



Aggregates Backward Inference

We recall that this propagation rule applies when an aggregate atom $f(A) > k$ has been derived true or false and there is a unique way to satisfy it by assigning a truth value to some atoms belonging to A. Concerning the Implication Graph $\mathcal{G}$, we have four different cases:

i) $\#count\{A\} > k$ is true and $|A| - |F_A| = k + 1$.
In this case a node a_i is added for each undefined atom a_i such that $\langle \bar{t}_i : a_i \rangle \in A$, and an arc $(\#aggr, a_i)$ is added to $\mathcal{G}$, where $\#aggr$ is an auxiliary atom representing $\#count\{A\} > k$. Finally, for each false atom a_j such that $\langle \bar{t}_j : a_j \rangle \in A$, an arc $(\neg a_j, a_i)$ is added to $\mathcal{G}$.

ii) $\#count\{A\} > k$ is false and $|T_A| = k$.
In this case a node $\neg a_i$ is added for each undefined atom a_i such that $\langle \bar{t}_i : a_i \rangle \in A$, and an arc $(\neg\#aggr, \neg a_i)$ is added to $\mathcal{G}$, where $\#aggr$ is an auxiliary atom representing $\#count\{A\} > k$. Finally, for each true atom a_j such that $\langle \bar{t}_j : a_j \rangle \in A$, an arc $(a_j, \neg a_i)$ is added to $\mathcal{G}$.

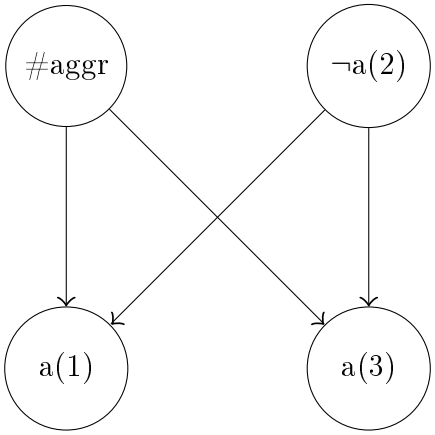
iii) $\#sum\{A\} > k$ is true and $\Phi(A) - \Phi(F_A) = k + 1$.
In this case a node a_i is added for each undefined a_i such that $\langle \bar{t}_i : a_i \rangle \in A$, and an arc $(\#aggr, a_i)$ is added to $\mathcal{G}$. Finally, for each false atom a_j such that $\langle \bar{t}_j : a_j \rangle \in A$, an arc $(\neg a_j, \neg a_i)$ is added to $\mathcal{G}$.

iv) $\#sum\{A\} > k$ is false and $\Phi(T_A) = k$.
In this case a node $\neg a_i$ is added for each undefined atom a_i such that $\langle \bar{t}_i : a_i \rangle \in A$, and an arc $(\#aggr, a_i)$ is added to $\mathcal{G}$. Finally, for each true atom a_j such that $\langle \bar{t}_j : a_j \rangle \in A$, an arc (a_j, a_i) is added to $\mathcal{G}$.

Example 21 (Aggregates Backward Inference: Implication Graph for Example 14)**.** In Example 14 we considered the following program:

```
r1: a(1) v b(1).
r2: a(2) v b(2).
r3: a(3) v b(3).
r3: a(1) :- not d.
r4: a(2) :- e.
r5: :- not #sum{<1 : a(1)>,<2 : a(2)>, <3 : a(3)>} > 3.
```

and a partial interpretation I = {¬a(2), b(2), ¬e}; $\#sum\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} > 3$ was true and $a(1)$ and $a(3)$ were inferred as true. The elements added to the Implication Graph are the following, where #aggr represents the aggregate atom $\#sum\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle, \langle 3 : a(3)\rangle\} > 3$:



4.6.2 First UIP

A node n in the Implication Graph is a UIP for a decision level d if and only if all paths from the choosen literal at the level d to the conflict atom pass through n. Intuitively, a UIP is the most concise reason of the conflict for a certain decision level. By definition, the chosen literal is always a UIP, but since several UIPs may exist, we calculate the UIP closest to the conflict, the first UIP. In particular, we calculate the first UIP only for the decision level of the conflict.

4.6.3 Main learning schema

Our learning schema is basically a first UIP learning schema (see Chapter 2). When a conflict is derived at a decision level d a constraint is learned. Such a constraint contains all atoms of a level lower than d which are connected to a node in a path from the first UIP to the conflict atom.

Example 22 (First UIP learning schema)**.** Consider the program:

```
r1: a v not_a.
r2: b v not_b.
r3: c v not_c.
r4: e :- c.
r5: :- c, d.
r6: f :- not d, e, a.
r7: g :- f.
r8: :- b, f, g.
```

Suppose I = {a, b, c} and the decision levels of the atoms a, b and c are 1, 2 and 3, respectively, $dl(a) = 1$, $dl(b) = 2$, $dl(c) = 3$. After the propagation of the atom c, forward inference on rule *r4* infers e as true, contraposition for false heads on rule *r5* infers d as false. By applying forward inference on rule *r6*, f is inferred as true. After that, forward inference on rule *r6* infers g as true and contraposition for false heads on rule *r8* infers g as false. Hence, we have a conflict on the atom g. As we can see in the Implication Graph (Figure 4.3), the first UIP is f. In this case, the learned constraint is the following:

```
:- f, b.
```

In fact, we have that each interpretation in which the atoms f and b are true leads toward a conflict on the atom g. In particular, the atom f infers the truth of the atom g by applying forward inference on the rule *r7*, and the atom f with the atom b infer the falsity of the atom g by applying contraposition for false heads on the rule *r8*. It is important to note that, after backtracking, b will still be true, so this constraint contains only one undefined atom, the first UIP. In this sense, the constraint forces the UIP to flip its truth value.

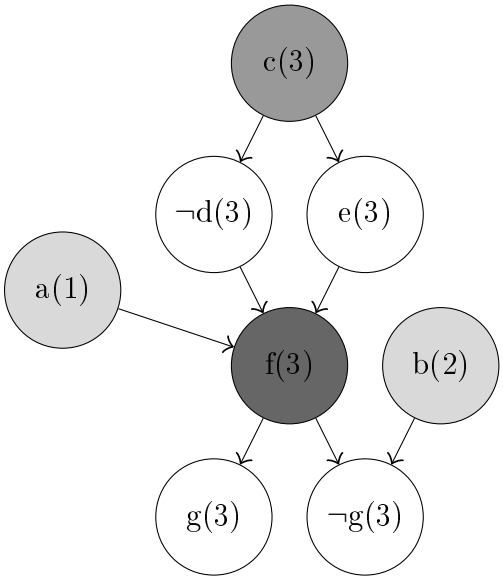


Figure 4.3: Implication Graph (First UIP: f; levels in parentheses)

4.6.4 Learned Constraint Deletion

Since the number of learned constraints may become exponential in the size of the program, we adopt a standard technique for expiring learned constraints. Our policy is similar to that of Minisat [21]: If the number of learned constraints is greater than one third of the input program, then we delete half of the learned constraints that have not recently been used for propagation. Moreover, each learned constraint has an activity value, measuring how frequently the constraint was involved in conflicts and we also delete all learned constraints with an activity value lower than a threshold value.

4.6.5 Restart

After a certain number of conflicts, the algorithm decide to restart the entire search if the total number of conflicts found during the search reaches a certain threshold. It is important to note that after each restart WASP works on a program composed of the original input program and the learned constraints. Our restart policy is based on the sequence of thresholds 32, 32, 64, 32, 32, 64, 128, ... introduced in [25]. This policy can be formally defined as the following sequence:

$$t_i = h * \begin{cases} 2^{k-1} & \text{if } i = 2^k - 1;\ \textit{for some } k \in \mathbb{N}, \\ t_{i-2^{k-1}+1} & \text{if } 2^{k-1} \leq\ i \leq 2^k;\ \textit{for some } k \in \mathbb{N}, \end{cases} \tag{4.1}$$

where h is a heuristic value set to 32. Note that for $h = 1$, the above policy coincides with the one that is proved in [25] to be universally optimal for a particular class of randomized algorithms, called Las Vegas algorithms. In particular, the running time is a logarithmic factor slower than the really optimal policy and any other universal policy can only do better up to a constant factor.

4.7 Heuristics

A crucial issue in the Model Generator function in Figure 4.2 is the selection of a literal when all deterministic inferences have been made and there are still undefined atoms. It is clear that the correctness of the algorithm reported in Figure 4.2 does not depend on the strategy in which this selection is made, but making a "good" choice is very important to achieve efficiency. However, strategies which perform very well on some domains may perform very badly for other domains, and an optimal strategy does not appear to exist. For this reason, some heuristic must be adopted, the quality of which can only be assessed empirically.

As in SAT, heuristics for ASP can be classified in two main classes, *look-ahead* based and *look-back* based. Look-ahead heuristics estimate the effects of assigning a specific truth value to a given undefined atom, for any truth value and for a set of undefined atoms (which might also be the set of all undefined atoms). Once the effects of all candidate assumptions have been estimated, a look-ahead heuristic selects the most promising undefined atom and truth value according to some function. Look-back heuristics, instead, rely on the information on conflicts derived in the computation so far.

The main heuristic implemented in WASP is based on a mixed approach. In fact, a look-back approach is used for selecting an undefined atom and, in some cases, a look-ahead step is performed for choosing the truth value for the selected atom. More specifically, statistics on previously detected conflicts are analyzed and atoms that have caused most conflicts are preferred. Also the "age" of conflicts is taken into account in the selection process, and more recent conflicts are given greater importance. Intuitively, an old conflict is less important than a new one, because it could have been learned very early in the computation and at the time a choice has to be made, we are exploring another area of the search space where that constraint is not used.

This approach has already been adopted in the context of SAT, for example in the BerkMin solver [34]. In this sense, our heuristic could be seen as an extension of the heuristic implemented in BerkMin to the framework of ASP.

4.7.1 Main heuristic

This heuristic maintains, for each literal L:

- a counter *cl(L)* which is increased when a new constraint containing L is added to the database and when L is traversed in the Implication Graph during the first UIP calculus; each of these counters is initialized to *0* and every *100* conflicts it is divided by *4*;
- a counter *cv(L)*, which is obtained by summing *cl(L)* and *cl(¬L)*.
- a global counter, *gcl(L)*, measuring the global contribution of each literal to the conflicts; each of these counters *gcl* is initialized to *0* and increased whenever *cl* is increased, but it is never decreased or divided.

Moreover, all learned constraints are organized in a chronologically ordered list.

A choice

The next decision atom is chosen as described below

i) If some learned constraints are not satisfied yet, consider the newest and choose the undefined atom with the highest value of *cv(L)* appearing in it.

ii) If some learned constraints are not satisfied yet, consider the newest, and if two or more atoms have the same highest value of *cv(L)*, we choose the atom that removes the highest number of supporting rules.

iii) If all the learned constraints are satisfied, then the atom with the highest *cv* counter is chosen.

iv) If no such learned constraint exists, we choose the atom with most occurrences in the program.

Truth Value choice

Once an atom is chosen, we have to select a truth value for it. We adopt the following policy:

1) If some learned constraints are not satisfied yet, we pick the literal with the highest *gcl* counter.

2) If some learned constraints are not satisfied yet, but the atom was chosen by ii), we pick the literal that removes the highest number of supporting rules.

3) If the atom was picked using iii), we use a look-ahead heuristic. In particular, we try to propagate both literals and then choose the literal that has the higher impact on the propagation, in the sense that it does more operations, like moving watches and satisfying rules. It is important to note that if we obtain a conflict in one of the two branches, we can choose the other truth value for the atom deterministically.

4) If we chose the atom with the atom a using iv), the literal in the set $\{a, \neg a\}$ with most occurrences in the program is picked.

Example 23 (Heuristic example)**.** Consider the following subprogram:

```
r1: a :- c.
r2: a v b :- d.
r3: a v c :- e.
r4: e v b :- c.
```

and the following learned constraints (in order of insertion):

```
c1: :- a, b.
c2: :- a, not c, d.
```

Suppose $I = \{a, \neg b\}$ and

$$\begin{array}{llll} cl(a) = 2, & cl(b) = 1, & cl(c) = 1, & cl(d) = 3, \\ cl(\neg a) = 2, & cl(\neg b) = 0, & cl(\neg c) = 2, & cl(\neg d) = 0. \end{array}$$

We have that *c1* is satisfied because b is false, and the newest not satisfied yet constraint is *c2*. In particular, in this constraint we have two undefined atoms, c and d, for which $cv(\mathrm{c}) = cv(\mathrm{d}) = 3$. In this case, we choose the atom that removes the higher number of supporting rules. In fact, if we set c false, we remove two supporting rules (*r1* and *r4*), while setting c true removes one supporting rule (*r3*). Conversely, if we pick d as false, we remove one supporting rule (*r2*), while setting d true removes no rules. Hence, we choose c as false.

4.7.2 "Two Function" Heuristic

We also implemented the *two* function used into the original BerkMin heuristic. The atom is chosen as explained in Section 4.7.1. The *two* function is

used in order to pick a literal only when an atom is chosen using iii). This heuristic picks the literal with the highest value of *two(L)*, where *two(L)* calculates the number of binary rules in the neighborhood of L. A rule is *binary* if it is unsatisfied and it contains two undefined atoms.

First of all, we compute the number of binary rules containing L. Let this number be n_1. Then, for each literal L_1 appearing in a binary rule with L, we compute the number of binary rules containing $\neg L_1$. Let this number be n_2. The value of *two(L)* is obtained by summing n_1 and n_2. If the value of the function *two* is the same for both literals, we assign false to the chosen atom. In order to limit the quantity of time spent to compute the function, we use a threshold value (equal to 100) for the *two* counters. In fact, once the value of *two* exceeds the threshold, the computation is stopped.

4.7.3 "Two Light" Heuristic

Since the computation of function *two* is expensive, we decided to use a variant of the previous heuristic. In fact, we compute the values of the *two* function only for the first conflict after each restart. For all other conflicts, we use the values computed by the last invocation of the *two* function.

4.7.4 Look-Ahead Heuristic

In this section we discuss a variant of the main heuristic that we implemented in WASP. This heuristic is similar to the look-ahead heuristic used in DLV [27].

Definition 28. A Possibly-True (PT) atom of Π w.r.t. I is an undefined atom a such that there exists a rule $r \in \Pi$ for which all of the following conditions hold:

i) $a \in H(r)$;

ii) for all $a_i \in H(r),\ a_i \notin I$, that is, there are no true atoms in the head of r;

iii) for all $a_i \in B(r),\ a_i \in I$, that is, the body of r is true.

Definition 29. An Unsupported-True (UT) atom of Π w.r.t. I is a true atom a such that there is no rule $r \in \Pi$ for which all of the following conditions hold:

i) $a \in H(r)$.

ii) for all $a_i \in H(r) \setminus \{a\}$, $\neg a_i \in I$, that is, all atoms in the head, with the exception of the atom a, are false.

iii) for all $a_i \in B(r)$, $a_i \in I$, that is, the body of the rule r is true.

Hence, if an atom is true but still misses a supporting rule for it, then it is an unsupported-true atom.

For example, the rule `:- not a.` implies that a is true, but it is not a supporting rule for it. In order to obtain a stable model, each true atom must be supported, hence we aim to decrease the number of unsupported true atoms. Given an atom A, we have five counters:

1. Let $UT(A)$ be the number of unsupported-true atoms after the propagation of A.
2. Let $UT_2(A)$ be the number of unsupported-true atoms appearing in the heads of exactly two unsatisfied rules after the propagation of A.
3. Let $UT_3(A)$ be the number of unsupported-true atoms appearing in the heads of exactly three unsatisfied rules after the propagation of A.
4. Let $Sat(A)$ be the number of satisfied rules after the propagation of A.
5. Let $DS(A)$ be the degree of supportedness of the interpretation intended as the ratio between the number of supporting rules and the number of true atoms after the propagation of A.

As described in [27], a good heuristic should take into account the effect of the look-ahead for both the branching literal and its complement. To this end, given an atom A, we have the following heuristic values:

1. $UT'(A) = UT(A) + UT(\neg A)$;
2. $UT'_2(A) = UT_2(A) + UT_2(\neg A)$;
3. $UT'_3(A) = UT_3(A) + UT_3(\neg A)$;
4. $Sat'(A) = Sat(A) + Sat(\neg A)$;
5. $DS'(A) = DS(A) + DS(\neg A)$.

Moreover, define a preference relationship $\succ_{look-ahead}$ such that $A \succ_{look-ahead} B$ if the criteria (described below) prefer choosing the atom A rather than B. More specifically, given two atoms A and B:

1. $UT'(A) < UT'(B)$; otherwise,

2. $UT'(A) = UT'(B)$ and $UT'_2(A) < UT'_2(B)$; otherwise,
3. $UT'(A) = UT'(B)$, $UT'_2(A) = UT'_2(B)$ and $UT'_3(A) < UT'_3(B)$; otherwise,
4. $UT'(A) = UT'(B), UT'_2(A) = UT'_2(B), UT'_3(A) = UT'_3(B)$ and $Sat'(A) > Sat'(B)$; otherwise,
5. $UT'(A) = UT'(B), UT'_2(A) = UT'_2(B), UT'_3(A) = UT'_3(B)$ and $DS'(A) > DS'(B)$.

The next decision atom is the first processed PT atom A such that for each PT atom $A_1 \neq A$, $A_1 \succ_{look-ahead} A$ does not hold.

Example 24 (Look-Ahead Heuristic example)**.** Consider the following subprogram:

```
r1: a v n_a.
r2: b v n_b.
r3: c v n_c.
r4: d v n_d.
r3: :- not e.
r4: :- not f.
r5: e :- a.
r6: e :- c.
r7: f :- a.
r8: f :- d.
```

Suppose $I = \{e, f\}$, hence the PT atoms are: {a, n_a, b, n_b, c, n_c, d, n_d}. As shown in [27], a and $n\_\, a$, b and $n\_\, b$, c and $n\_\, c$, d and $n\_\, d$ are each look-ahead equivalent, hence we can just focus on the respective atom. In particular, we have:

$$\begin{array}{llll} UT(a) = 0, & UT(b) = 2, & UT(c) = 1, & UT(d) = 1, \\ UT(\neg a) = 2, & UT(\neg b) = 2, & UT(\neg c) = 2, & UT(\neg d) = 2. \end{array}$$

Moreover, we have $UT'(a) < UT'(c) = UT'(d) < UT'(b)$, hence, the best choice according to the criteria is a.

Optimizing the Computation of the Heuristic

The computation of this heuristic is expensive because the number of PT atoms to be considered is very large in some cases. Thus, we use two techniques that reduce the amount of time needed to evaluate the heuristics by

reducing the number of look-aheads that have to be performed. In particular, we use the look-ahead equivalence result obtained in [27] and we define a threshold value for PT atoms to be considered in the heuristics.

4.8 Related Work

The techniques presented in this thesis were in part already adopted by other existing non-disjunctive ASP solvers like Smodels$_{cc}$ [44, 45], Clasp [7], and solvers supporting disjunction like CModels3 [9], GnT [46], ClaspD [7] and DLV [47, 48]. Concerning those existing ASP solvers, WASP differs from non-native solvers like Cmodels3 [9] in the sense that we do not rely on a rewriting to a propositional formula and an external SAT solver, but use native ASP techniques. Among native solvers, similarities with DLV [3] can be found in the propagation rules, in the computation of the greatest unfounded set, and in the model checking technique. However, we clearly differ from DLV as it does not implement many of the look-back techniques borrowed from CP and SAT. The prototypical version of DLV, presented in [47] and extended in [48], implements backjumping and some forms of look-back heuristics, but it does not include clause learning, restarts, and does not use an Implication Graph for determining the reasons of the conflicts. Similar considerations hold for GnT [46], which, as DLV, implements a systematic backtracking without learning and look-ahead heuristics.

Comparing our system with ClaspD more similarities can be found, as it includes similar techniques, e.g. backjumping, clause learning, restarts, and look-back heuristics. There are nonetheless several differences with WASP. First of all, WASP performs the unfounded set checking by means of the well-founded operator, while ClaspD relies on the computation of loop formulas. Moreover, ClaspD implements an alternative version of the Implication Graph that is more similar to SAT solvers, since it relies on unit propagation of nogoods (minimality is handled via loop formula learning). Furthermore, ClaspD, as WASP, adopts a branching heuristics based on Berkmin [34]; however, WASP extends the original Berkmin heuristics by exploiting a lookahaed technique in place of the *two* function calculating the number of binary clauses in the neighborhood of literal L, together with an additional criterion based on minimality of answer sets. In particular, to deal with the case of two atoms with the same heuristic value, WASP chooses the atom that introduces the maximum number of unsatisfied supporting rules.

Chapter 5

Experiments

In this section we report the results of an experimental analysis we carried out in order to assess the performance of WASP. As a comparison, we also ran the suite of our benchmarks on two state-of-the-art ASP solvers, namely DLV and ClaspD (winners of the disjunctive tracks in the last ASP Competitions [11, 12, 13]). A discussion on the difference between WASP and these two systems is provided in Section 4.8.

The machine used for the experiments is a two-processor Intel Xeon "Woodcrest" (quad core) 3GHz with 4MB of L2 Cache and 4GB of RAM, running Debian GNU Linux 4.0. As our ASP system focuses on the Model Generation phase, only the time for evaluating ground programs (previously produced by the DLV instantiator from the original non-ground instances) have been considered.

In the following, we briefly describe both benchmark problems and data.

5.1 Benchmark Problems and Data

In our experiments, we considered problems from the most recent ASP Competition [13] and other problems which have already been used for assessing performance of the ASP solver DLV [3]. Our experiments consist of 36 instances in 15 different domains. The instances and encodings are those that were used in the competitions or in the other (publicly available) suites. In the following we describe the benchmark problems.

Labyrinth

Ravensburger's Labyrinth game deals with guiding an avatar through a dynamically changing labyrinth to certain fields. A solution is represented by

pushing the labyrinth's rows and columns such that the avatar can reach the goal field (which can change its location during pushes) from its starting field (which can also change its location when pushed) by a move along some path after each push.

Knight-Tour

Given a chessboard, the problem is to find a tour for a knight piece that starts at any square, travels all squares, and comes back to the origin, following the knight motion rules of chess.

Graph coloring

Given an undirected graph and a set of n colors, we are interested in checking whether there is an assignment of colors to nodes such that no adjacent nodes share the same color.

Maze-Generation

A maze is an $m \times n$ grid, in which each cell is empty or a wall and two distinct cells on the edges are indicated as entrance and exit, satisfying the following conditions:

1) each cell on the edge of the grid is a wall, except entrance and exit that are empty;
2) there is no 2×2 square of empty cells or walls;
3) if two walls are on a diagonal of a 2×2 square, then not both of their common neighbors are empty;
4) no wall is completely surrounded by empty cells;
5) there is a path from the entrance to every empty cell.

The problem has been proved to be NP-complete in [22].

Strategic Companies

Strategic companies is a well-known Σ_2^P-complete problem that has often been used for system comparisons, also in the previous ASP Competitions. In the Strategic Companies problem, a collection $C = c_1, \ldots, c_m$ of companies is given, for some $m \geq 1$. Each company produces some goods in a set G,

and each company c_i in C is possibly controlled by a set of owner companies O_i (where O_i is a subset of C, for each $i = 1, \ldots, m$). In this context, a set C' of companies (i.e., a subset of C) is a *strategic set* if it is minimal among all the sets satisfying the following conditions: (i) Companies in C' produce all goods in G; (ii) if O_i is a subset of C' if $c_i \in C'$ (for each $i = 1, \ldots, m$). We considered a random instance involving 7500 companies and 22500 products.

2-QBF

The problem consists of checking the validity of a quantified boolean formula $\Phi = \exists X \forall Y \phi$, where X and Y are disjoint sets of propositional variables and $\phi = C_1 \vee \ldots \vee C_k$ is a DNF on variables X and Y. In our benchmark, we used the transformation from 2-QBF to ASP presented in [3], which is based on a reduction presented in [24]. The instance considered has 1000 universal variables, 20 existential variables, 10000 clauses, and is a 5-DNF.

Prime Implicants

In Boolean logic, an implicant is a "covering" (sum term or product term) of one or more minterms (a product term in which each of the n variables appears once) in a sum of products, or, maxterms (a sum term in which each of the n variables appears once) in a product of sums, of a boolean function. Formally, a product term P in a sum of products is an implicant of the Boolean function F if P implies F. A prime implicant of a function is an implicant that cannot be covered by a more general (more reduced - meaning with fewer literals) implicant. The instance we considered consists of 180 variables and 774 clauses.

3-Colorability

This well-known problem asks for an assignment of three colors to the nodes of a graph, in such a way that adjacent nodes always have different colors. One simplex graph was generated with the Stanford GraphBase library [23], by using the function $simplex(600, 600, -2, 0, 0, 0, 0)$. Another ladder graph was generated having 11998 edges, and 8000 nodes.

Hamiltonian Cycle

This is a classical NP-complete problem in graph theory, which can be expressed as follows: given a directed graph $G = (V, E)$ and a node $a \in V$ of this graph, does there exist a path in G starting at a and passing through each

node in V exactly once? One random graph was generated with the Stanford GraphBase library [23], by using the function $random\_graph(85, 700, 0, 0, 0, 0, 0, 1, 1, 33)$, having 700 edges and 85 nodes; the other instance has been generating using the function $random\_graph(80, 456, 0, 0, 0, 0, 0, 1, 1, 33)$, having 456 edges and 80 nodes.

Blocks World

Blocks world is one of the most famous planning domains in artificial intelligence. We have a set of cubes (blocks) deposited on a table. The goal is to build one or more vertical stacks of blocks. The constraint is that only one block may be moved at a time: it may either be placed on the table or placed on top of another block. Because of this, any blocks that are, at a given time, under another block cannot be moved. The four instances considered are by Esra Erdem and taken from the ccalc homepage (http://www.cs.utexas.edu/users/tag/cc/).

3SAT

The satisfiability problem (SAT) is a decision problem, whose instance is a propositional formula. The question is: given the formula, is there some assignment of the values TRUE and FALSE to the variables that will make the entire expression true? SAT is the best-known NP-complete problem. 3-satisfiability is a special case of SAT, where each formula is a CNF in which each clause contains exactly three literals. We considered two random instances with 280 variables and 1204 clauses.

Hanoi

The Towers of Hanoi (ToH) problem has three pegs and n disks. Initially, all n disks are on the left-most peg. The goal is to move all n disks to the right-most peg with the help of the middle peg. The rules are: (1) move one disk at a time; (2) only the top disk on a peg can be moved; (3) a larger disk cannot be placed on top of a smaller one. The instance we considered has 6 disks, and we check whether a plan of length 64 exists.

Ramsey Numbers

The Ramsey number $ramsey(k, m)$ is the least integer n such that, no matter how the edges of the complete undirected graph (clique) with n nodes are colored using two colors, say red and blue, there is a red clique with k nodes (a

red k-clique) or a blue clique with m nodes (a blue m-clique). The encoding of this problem consists of one rule and two constraints. For the experiments, the problem considered was deciding whether, for $k = 3$, $m = 7$, $n = 21$, and for $k = 4$, $m = 6$, $n = 26$, n is the Ramsey number $ramsey(k, m)$.

n-Queens

The 8-queens puzzle is the problem of putting eight chess queens on an 8x8 chessboard such that none of them threatening any other (using the standard chess queen's moves). The n-queens puzzle is the more general problem of placing n queens on an nxn chessboard ($n \geq 4$). The instance considered is for $n = 23$.

Timetabling

The problem is determining a timetable for some university lectures that have to be given in one week to some groups of students. The timetable must respect a number of given constraints concerning availability of rooms, teachers, and other issues related to the overall organization of the lectures.

5.2 Experimental Results

The results of our experiment are summarized in Table 5.1, reporting the execution times in seconds elapsed by each considered system, for each considered instance. For each instance of the benchmark problems, we allowed a maximum of 600 seconds of execution time. Timeouts are indicated by means of the word TIME in Table 5.1. In the last rows we report, for each system, the total number of solved instances, the average execution time for solving all the 36 considered instances (timeouts are counted 600s each), and the number of instances in which each solver resulted to be the fastest.

Overall, the results of the preliminary experimental analysis are encouraging: the performance of WASP is comparable to that of ClaspD (same number of wins and cumulative average time), and it is often faster than DLV (only 9 wins vs 15 of WASP and ClaspD). In more detail, for the Labyrinth problem WASP was able to solve four instances out of five in the allowed time, while the other systems solved all five instances; the system is always outperformed by the competitors, except for one instance in which it is the best performer. Regarding the Knight-Tour problem, WASP always outperformed the competitor systems, solving the hardest instance

Table 5.1: Benchmark Results on ASP competition suite

Problem	**WASP**	**DLV**	**ClaspD**
LABYRINTH-1	0,39	0,02	0,03
LABYRINTH-2	299,74	3,17	65,84
LABYRINTH-3	415,14	56,19	113,04
LABYRINTH-4	TIME	25,76	561,93
LABYRINTH-5	14,47	29,15	490,04
KNIGHT-TOUR-1	0,07	0,21	0,15
KNIGHT-TOUR-2	0,14	1,64	0,34
KNIGHT-TOUR-3	0,65	14,45	2,84
KNIGHT-TOUR-4	0,67	56,31	10,56
KNIGHT-TOUR-5	7,44	TIME	179,48
GRAPH-COLOURING-1	153,67	TIME	3,05
GRAPH-COLOURING-2	TIME	TIME	TIME
MAZE-GENERATION-1	0,28	0,93	0,79
MAZE-GENERATION-2	46,84	104,47	1,76
MAZE-GENERATION-3	47,37	261,57	3,94
MAZE-GENERATION-4	94,17	TIME	9,64
MAZE-GENERATION-5	123,40	TIME	23,49
STRATCOMP	179,06	2,33	5,71
2QBF	0,11	3,31	0,92
PRIMEIMPL	3,24	1,33	0,21
3COL-SIMPLEX	23,02	33,58	TIME
3COL-LADDER	2,29	91,24	34,08
HAMCYCLE-RANDOM	5,29	1,50	2,52
HAMCYCLE-FREE	106,89	31,37	0,47
BLOCKS-WORLD-1	224,09	6,48	1,92
BLOCKS-WORLD-2	340,84	11,84	1,75
BLOCKS-WORLD-3	0,76	8,87	1,67
BLOCKS-WORLD-4	129,28	11,05	0,83
3SAT-1	78,31	9,59	65,84
3SAT-2	31,07	5,43	0,06
TOWERS-OF-HANOI	3,81	8,46	437,55
RAMSEY-1	3,03	9,84	24,01
RAMSEY-2	4,87	15,74	40,28
23-QUEENS	0,10	41,10	0,54
SCHOOL-TIMETABLING	7,45	61,09	224,93
TOTAL SOLVED	**34**	**31**	**34**
WEIGHTED AVERAGE	**98,08**	**108,87**	**98,17**
WINS	**15**	**9**	**15**

(on which DLV timed out) in only $7,44$ seconds, compared to $179,48$ seconds for ClaspD. Concerning the Graph Coloring problem, WASP was slower than ClaspD, but solved one instance more than DLV. Also for the Maze Generation benchmarks, WASP was slightly slower than ClaspD, but always outperformed DLV. Considering the other benchmarks, WASP outperformed the other two ASP solvers on 2QBF, Ramsey Numbers, N-Queens, School Timetabling, 3Colorability, and Towers of Hanoi. In the other benchmarks, the system remains competitive, with the single exception of Strategic Companies. For this, we hypothesize that a reason might be that WASP does not implement yet a model-checking-driven backjumping technique, which proved to be very effective on this particular benchmark [28].

Chapter 6

Conclusion

In this thesis we provided a report on a new ASP solver for propositional programs called WASP. The new system is inspired by several techniques that were originally introduced for SAT solving, like the *Davis-Putnam-Logemann-Loveland (DPLL) backtracking search algorithm* [29], *clause learning* [30, 31], *restarts* [32], and *conflict-driven heuristics* [33] in the style of Berkmin [34]. Actually, some of the techniques adopted in WASP, including *backjumping* and *look back heuristics*, were first introduced for Constraint Satisfaction [35, 36, 37] and then successfully applied in SAT [38, 39, 34, 33] and QBF solving [40, 41, 42, 43]. It is worth noting that none of the employed techniques can be applied as-is in an ASP solver, and in this thesis we adapted and properly modified them in order to fit the particularities of disjunctive logic programs under the answer set semantics.

The performance of WASP was assessed by an experimental analysis conducted on both random and structured instances, the results of which are also reported in this thesis; the observed result was that WASP is able to compete with the state-of-the-art solvers DLV [3] and ClaspD [7], and even outperform them in several of the considered benchmarks. This result is particularly promising, since there is still room for improvements in the implementation of WASP, e.g. through the optimization and tuning of data structures and heuristic parameters.

Concerning future work, we plan to extend the system by introducing new language constructs such as full support to aggregates [49, 50] and weak constraints [51], which are currently missing from WASP. Moreover, the current implementation can be improved in several respects: parameter tuning of the heuristics, fine tuning of the source code. A model-checking-driven backjumping [28] as well as support for multi-threading are also planned.

It is worth pointing out that a part of this work was developed at the Tech-

nische Universität of Wien, under the Erasmus Placement project. Moreover, a prelimary version of this work has been published in [55] and presented in the XXVI Italian conference on Computational Logic held in Pescara from 31 August 2011 to 2 September 2011 [57].

Bibliography

[1] M. Gelfond, V. Lifschitz. *Classical Negation in Logic Programs and Disjunctive Databases*. In: New Generation Computing, 1991, pp. 365-385.

[2] V. Lifschitz. *Answer Set Planning*. In: Proceedings of the 16th International Conference on logic programming (ICLP '99), D. D. Schreye, Ed. The MIT Press, Las Cruces, New Mexico, USA, 23-37.

[3] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello. *The DLV System for Knowledge Representation and Reasoning*. ACM TOCL 7 (2006) 499-562.

[4] P. Simons, I. Niemelä, T. Soininen. *Extending and Implementing the Stable Model Semantics*. In: AI 138 (2002), 181-234.

[5] F. Lin, Y. Zhao. *ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers*. In: AAAI-2002, Edmonton, Alberta, Canada, AAAI Press / MIT Press (2002).

[6] Y. Babovich, M. Maratea. *Cmodels-2: Sat-based answer sets solver enhanced to non-tight programs*. http://www.cs.utexas.edu/users/tag/cmodels.html (2003).

[7] M. Gebser, B. Kaufmann, A. Neumann, T. Schaub. *Conflict-driven answer set solving*. In: IJCAI 2007,(2007) 386-392.

[8] T. Janhunen, I. Niemelä, D. Seipel, P. Simons, J.H. You. *Unfolding Partiality and Disjunctions in Stable Model Semantics*. ACM TOCL 7 (2006) 1-37.

[9] Y. Lierler. *Disjunctive Answer Set Programming via Satisfiability*. In: LPNMR '05. LNCS 3662, (2005) 447-451.

[10] C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. Konig, M. Ostrowski, T. Schaub. *Conflict-Driven Disjunctive Answer Set Solving*. In:

Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2008), Sydney, Australia, AAAI Press (2008) 422-432.

[11] M. Gebser, L. Liu, G. Namasivayam, A. Neumann, T. Schaub, M. Truszczyński. *The first answer set programming system competition.* LPNMR '07. LNCS 4483, (2007) 3-17.

[12] M. Denecker, J. Vennekens, S. Bond, M. Gebser, M. Truszczyński. *The second answer set programming competition.* Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning. LPNMR '09, Berlin, Heidelberg, (2009) 637-654.

[13] F. Calimeri, G. Ianni, F. Ricca, M. Alviano, A. Bria, G. Catalano, S. Cozza, W. Faber, O. Febbraro, N. Leone, M. Manna, A. Martello, C. Panetta, S. Perri, K. Reale, M.C. Santoro, M. Sirianni, G. Terracina, P. Veltri. *The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track.* Proc. of LPNMR '11, LNCS (2003) 388-403.

[14] F.Ricca, W.Faber, N.Leone. *A Backjumping Technique for Disjunctive Logic Programming.* In: AI Communications, Volume 19 Issue 2, January 2006.

[15] L. Zhang, C. F. Madigan, M. H. Moskewicz, S. Malik. *Efficient Conflict Driving Learning in a Boolean Satisfiability Solver.* In: Proceedings of the International Conference on Computer-Aided Design (ICCAD 2001).

[16] M. Maratea, F. Ricca, W. Faber, N. Leone. *Look-Back Techniques and Heuristics in DLV: Implementation, Evaluation, and Comparison to QBF Solvers.* In: JOACIL, Volume 63 (2008).

[17] E. Giunchiglia, M. Maratea. *An Experimental Study of Search Strategies and Heuristics in Answer Set Programming.* In: Proceedings of ASP'05.

[18] N. Dershowitz, Ziyad Hanna, Alexander Nadel. *A Clause-Based Heuristic for SAT Solvers.* In: International Conference on Theory and Applications of Satisfiability Testing.

[19] N. Leone, P. Rullo, F. Scarcello. *Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics, and Computation.* Christian Doppler Laboratory for Expert Systems, TU Vienna, Austria.

[20] J. P. Marques-Silva, K. A. Sakallah. *"GRASP: A search Algorithm for Propositional Satisfiability"*. IEEE Transactions on Computers, Volume 48, 506-521, 1999.

[21] N. Eén, N. Sörensson. *An Extensible SAT-solver*. In: Theory and Applications of Satisfiability Testing, vol. 2919, 333-336, 2004.

[22] M. Alviano. *The Maze Generation Problem is NP-complete*. In: Proceedings of the 11th Italian Conference on Theoretical Computer Science (ICTCS '09) (2009).

[23] D.E. Knuth. *The Stanford GraphBase*. A Platform for Combinatorial Computing, ACM Press, New York (1994).

[24] T. Eiter, G. Gottlob. *On the Computational Cost of Disjunctive Logic Programming*. Propositional Case, AMAI 15 (1995) 289-323.

[25] M. Luby, A. Sinclair, D. Zuckerman. *Optimal speedup of las vegas algorithms*. Inf. Process. Lett. 47 (1993) 173-180.

[26] W. Faber, N. Leone, M. Maratea, F. Ricca. *Look-back Techniques for ASP Programs with Aggregates*. The 15th RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (RCRA 2008). December 2008.

[27] W. Faber, N. Leone, G. Pfeifer, F. Ricca. *On Look-Ahead Heuristics in Disjunctive Logic Programming*. In: Annals of Mathematics and Artificial Intelligence archive Volume 51 Issue 2-4, December 2007.

[28] C. Koch, N. Leone, G. Pfeifer. *Enhancing Disjunctive Logic Programming Systems by SAT Checkers*. AI 15 (2003) 177-212.

[29] M. Davis, G. Logemann, D. Loveland. *A Machine Program for Theorem Proving*. Communications of the ACM 5 (1962) 394-397.

[30] L. Zhang, C.F. Madigan, M.W. Moskewicz, S. Malik. *Efficient Conflict Driven Learning in Boolean Satisfiability Solver*. In: ICCAD 2001. (2001) 279-285.

[31] K. Pipatsrisawat, A. Darwiche. *On Modern Clause-Learning Satisfiability Solvers*. JAIR 44 (2010) 277-301.

[32] C.P. Gomes, B. Selman, H.A. Kautz. *Boosting Combinatorial Search Through Randomization*. In: Proceedings of AAAI/IAAI 1998, AAAI Press (1998) 431-437.

[33] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik. *Chaff: Engineering an Efficient SAT Solver.* In: DAC 2001 (2001) 530-535.

[34] E. Goldberg, Y. Novikov. *BerkMin: A Fast and Robust Sat-Solver.* In: Design, Automation and Test in Europe Conference and Exposition (DATE 2002), Paris, France, IEEE Computer Society (2002) 142-149.

[35] J. Gaschnig. *Performance measurement and analysis of certain search algorithms.* PhD thesis, CMU (1979) Tech. Report CMU-CS-79-124.

[36] P. Prosser. *Hybrid Algorithms for the Constraint Satisfaction Problem.* Computational Intelligence 9 (1993) 268-299.

[37] W. Faber, N. Leone, G. Pfeifer. *Pushing Goal Derivation in DLP Computations.* In: LPNMR '99. LNCS 1730, (1999) 177-191.

[38] R. Bayardo, R. Schrag. *Using CSP Look-back Techniques to Solve Real-world SAT Instances.* In: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI- 97). (1997) 203-208.

[39] J.P.M. Silva, K.A. Sakallah. *GRASP: A Search Algorithm for Propositional Satisfiability.* IEEE Transaction on Computers 48 (1999) 506-521.

[40] L. Zhang, S. Malik. *Conflict Driven Learning in a Quantified Boolean Satisfiability Solver.* In: Proceedings of the International Conference on Computer-Aided Design (ICCAD 2002). (2002) 442-449.

[41] L. Zhang, S. Malik. *Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation.* In: CP 2002. NY, USA, (2002) 200-215.

[42] E. Giunchiglia, M. Narizzano, A. Tacchella. *Backjumping for Quantified Boolean Logic Satisfiability.* AI 145 (2003) 99-120.

[43] R. Letz. *Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas.* In: TABLEAUX 2002. Denmark, (2002) 160-175.

[44] J. Ward, J.S. Schlipf. *Answer Set Programming with Clause Learning.* In: LPNMR-7. LNCS 2923, (2004) 302-313.

[45] J. Ward. *Answer Set Programming with Clause Learning.* PhD thesis, Ohio State University, Cincinnati, Ohio, USA (2004).

[46] T. Janhunen, I. Niemelä. *Gnt - a solver for disjunctive logic programs.* In: LPNMR-7. LNCS 2923, Fort Lauderdale, Florida, USA, (2004) 331-335.

[47] F. Ricca, W. Faber, N. Leone. *A Backjumping Technique for Disjunctive Logic Programming.* AI Communications 19 (2006) 155-172.

[48] M. Maratea, F. Ricca, W. Faber, N. Leone. *Look-back techniques and heuristics in dlv: Implementation, evaluation and comparison to qbf solvers.* Journal of Algorithms in Cognition, Informatics and Logics 63 (2008) 70-89.

[49] N. Pelov, M. Denecker, M. Bruynooghe. *Well-founded and Stable Semantics of Logic Programs with Aggregates.* TPLP 7 (2007) 301-353.

[50] W. Faber, N. Leone, G. Pfeifer. *Semantics and complexity of recursive aggregates in answer set programming.* AI 175 (2011) 278-298. Special Issue: John McCarthy's Legacy.

[51] F. Buccafurri, N. Leone, P. Rullo. *Enhancing Disjunctive Datalog by Constraints.* IEEE TKDE 12 (2000) 845-860.

[52] C. Koch, N. Leone, G. Pfeifer. *Enhancing Disjunctive Logic Programming Systems by SAT Checkers.* AI 15 (2003) 177-212.

[53] T. Eiter, W. Faber, N. Leone, G. Pfeifer. *Declarative Problem-Solving Using the DLV System.* In: Jack Minker, editor, Logic-Based Artificial Intelligence, pages 79-103. Kluwer Academic Publishers, 2000.

[54] W. Faber, N. Leone, F. Ricca. *Answer Set Programming.* In: Wiley Encyclopedia of Computer Science and Engineering, 2008.

[55] C. Dodaro, M. Alviano, W. Faber, N. Leone, F. Ricca, M. Sirianni. The Birth of a WASP: Preliminary Report on a New ASP Solver. In Fabio Fioravanti, editor, *26th Italian Conference on Computational Logic (CILC 2011)*, volume 810 of *CEUR Workshop Proceedings.* Sun SITE Central Europe, 2011.

[56] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070-1080, Cambridge, Mass., 1988. MIT Press.

[57] http://www.sci.unich.it/cilc2011/

[58] T. Eiter, G. Gottlob, H. Mannila. *Disjunctive Datalog.* In: ACM Transactions on Database Systems, 22(3):364-418, September 1997.

[59] N. Leone, P. Rullo, F. Scarcello. *Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation.* In Information and Computation, 135(2):69-112, June 1997.

[60] M. Maratea, F. Ricca, P. Veltri. *DLVMC: Enhanced Model Checking in DLV.* In JELIA 2010: 365-368.

[61] F. Ricca, G. Grasso, M. Alviano, M. Manna, V. Lio, S. Iiritano, N. Leone. *Team-building with Answer Set Programming in the Gioia-Tauro Seaport.* In: Theory and Practice of Logic Programming. Cambridge University Press, 2011.To appear.

[62] M. Manna, M. Ruffolo, E. Oro, M. Alviano, N. Leone. *The HiLeX System for Semantic Information Extraction.* In: Transactions on Large-Scale Data and Knowledge-Centered Systems. Springer Berlin/Heidelberg, 2011. To appear.

[63] F. Ricca, M. Alviano, A. Dimasi, G. Grasso, S. M. Ielpa, S. Iiritano, M. Manna, N. Leone. *A Logic-Based System for e-Tourism.* In: Fundamenta Informaticae. IOS Press, 105(1–2):35–55, 2010.

[64] S. P. Radziszowski. *Small Ramsey Numbers.* In: The Electronic Journal of Combinatorics, 1, 1994. Revision 9: July 15, 2002.

[65] N. Leone, S. Perri, F. Scarcello. *Improving ASP Instantiators by Join-Ordering Methods.* In: Logic Programming and Nonmonotonic Reasoning - 6th International Conference, LPNMR'01, Vienna, Austria.