The Birth of a WASP: Preliminary Report on a New ASP Solver*

Carmine Dodaro, Mario Alviano, Wolfgang Faber, Nicola Leone,
Francesco Ricca, and Marco Sirianni

Dipartimento di Matematica, Università della Calabria, 87030 Rende, Italy
carminedodaro@gmail.com,
{alviano,faber,leone,ricca,sirianni}@mat.unical.it

Abstract. We present 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. We describe the key components of this solving strategy, namely: learning, restarts, heuristics based on look-back concepts, and backjumping. At the same time, we introduce a new heuristics based on a mixed approach between look-back and look-ahead techniques. Moreover, we present the results of preliminary experiments that we conducted in order to assess the impact of these techniques on both random and structured instances (used also in the last ASP Competition 2011). In particular, we compared our system with both DLV and ClaspD.

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 [3]. 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. [4–11]). 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 [12], semantic-based information extraction [13], and e-tourism [14]. 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 Contests series [15–17].

* Partly supported by Regione Calabria and EU under POR Calabria FESR 2007-2013 and within the PIA project of DLVSYSTEM s.r.l., and by MIUR under the PRIN project LoDeN. We also thank the anonymous reviewers for their valuable comments.

This paper provides a contribution in the aforementioned context. In particular, we provide a preliminary report on the development of 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 [18], *clause learning* [19, 20], *backjumping* [21, 22], *restarts* [23], and *conflict-driven heuristics* [24] in the style of Berkmin [25]. The mentioned SAT-solving methods have been adapted and combined with state-of-the-art pruning techniques adopted by modern native disjunctive ASP systems [4]. 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., [26]). Moreover, $wasp$ uses a new branching heuristics tailored for ASP programs, which is based on a mixed approach between Berkmin-like heuristics and look-ahead, which takes into account minimality of answer sets (a requirement not present in SAT solving). Finally, stable model checking, which is a co-NP-complete problem for disjunctive logic programs, is efficiently implemented relying on the rewriting method of [27], by calling Minisat [28] as suggested by [29].

In the following, after briefly introducing ASP, we describe the new system $wasp$. We start from the solving strategy and present the design choices regarding propagation, constraint learning, restarts, and the new heuristics. Moreover, we present the results of some experiments conducted for assessing the impact of these techniques, on both random and structured instances; some of these instances had been used in the last ASP Competition [17]. In particular, we compared our system with both DLV and ClaspD. The obtained results are encouraging: the new prototype system is already competitive with state-of-the-art solvers, even if there is still room for improvements in both the implementation (e.g., through the optimization and tuning of data structures and heuristic parameters), and in the supported language features (notably aggregates and weak constraints).

2 Preliminaries

In this paper we consider propositional programs, so an atom p is a member of a countable set $\mathcal{A}$. A *literal* is either an atom p (a positive literal), or an atom preceded by the *negation as failure* symbol not (a negative literal). A *rule* r is of the form

$$p_1 \vee \cdots \vee p_n \texttt{ :- } q_1, \ldots, q_j, \mathrm{not}\ q_{j+1}, \ldots, \mathrm{not}\ q_m \tag{1}$$

where $p_1, \ldots, p_n, q_1, \ldots, q_m$ are atoms and $n \geq 0, m \geq j \geq 0$. The disjunction $p_1 \vee \cdots \vee p_n$ is the *head* of r, while the conjunction $q_1, \ldots, q_j, \mathrm{not}\ q_{j+1}, \ldots, \mathrm{not}\ q_m$ is the *body* of r. Moreover, $H(r)$ denotes the set of head atoms, while $B(r)$ denotes the set of body literals. We also use $B^+(r)$ and $B^-(r)$ for denoting the set of atoms appearing in positive and negative body literals, respectively, and $At(r)$ for the set $H(r) \cup B^+(r) \cup B^-(r)$. A rule r is normal (or disjunction-free) if $|H(r)| \leq 1$, positive (or negation-free) if $B^-(r) = \emptyset$, a *fact* if both $B(r) = \emptyset$ and $|H(r)| = 1$, a *constraint* if $|H(r)| = 0$.

A *program* $\mathcal{P}$ is a finite set of rules; if all rules in it are positive (resp. normal), then $\mathcal{P}$ is a positive (resp. normal) program.

Let $\overline{L}$ denote the complement of a literal L, i.e., $\overline{a} = \text{not } a$ and $\overline{\text{not } a} = a$ for an atom a. We extend this to sets of literals and will use $\overline{S}$ for denoting $\{\overline{L} \mid L \in S\}$. An *interpretation* I is a subset of $\mathcal{A} \cup \overline{\mathcal{A}}$. An interpretation I is total if for each $a \in \mathcal{A}$ either $a \in I$ or $\text{not } a \in I$; otherwise, I is partial. An interpretation I is inconsistent if there exists $a \in \mathcal{A}$ such that $\{a, \text{not } a\} \subseteq I$; otherwise, I is consistent. An interpretation thus associates each ASP structure (atom, literal, head or body) with a truth value in the set $\{\mathcal{T}, \mathcal{F}, \mathcal{U}\}$, which extends to $H(r)$ and $B(r)$ in the standard way.

An interpretation I *satisfies* a rule $r \in \mathcal{P}$ if $H(r)$ is true w.r.t. I whenever $B(r)$ is true w.r.t. I, while I *violates* r if $H(r)$ is false but $B(r)$ is true. A total interpretation I is a *model* of a program $\mathcal{P}$ if I satisfies all the rules in $\mathcal{P}$. Given an interpretation I for a program $\mathcal{P}$, the reduct of $\mathcal{P}$ w.r.t. I, denoted by $\mathcal{P}^I$, is obtained by deleting from $\mathcal{P}$ all the rules r with $B^-(r) \cap I \neq \emptyset$, and then by removing all the negative literals from the remaining rules. The semantics of a program $\mathcal{P}$ is given by the set $\mathcal{AS}(\mathcal{P})$ of the answer sets (or stable models) of $\mathcal{P}$, where a total interpretation M is an answer set (or stable model) for $\mathcal{P}$ if and only if M is a subset-minimal model of $\mathcal{P}^M$.

3 Model Generator

In this section we sketch the main model generator function MG (cf. Fig. 1), which is able to perform learning and restart techniques. MG is similar to the Davis-Putnam procedure in SAT solvers. For reasons of presentation, we have considerably simplified the procedure in order to focus on its main ideas. 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, $\mathcal{P}$ 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 $\mathcal{P}$ has an answer set, and 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 [24]. Propagate is similar to unit propagation as employed by SAT solvers, but exploits the peculiarities 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 3.1, 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 to verify that the current total interpretation is also an answer set; the CheckModel function implements the techniques described in [27]. If the stability check succeeds, MG returns true. [1] 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.

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

If Propagate returns false, function ResolveConflict is called, which calculates the Unique Implication Point (UIP) of the implication graph (see Section 3.1), and exploits it to *learn* a constraint representing the inconsistency (see Section 3.2), 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 in the first branch of the search that is free of the just-detected conflict.

After a certain number of conflicts, ResolveConflict may decide to 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 [30].

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.

```
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 ∪ {not 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;
```

Fig. 1. Computation of answer sets

3.1 Propagation

WASP implements a number of deterministic inference rules for pruning the search space during the computation of stable models. These inference rules are named *forward inference*, *Kripke-Kleene negation*, *contraposition for true heads*, *contraposition for false heads* and *well-founded negation*. All of these inference rules are briefly described in this section.

During the propagation of deterministic inferences, implication relationships among atoms are stored in a graph $\mathcal{G}$ named Implication Graph. This graph has a node $\langle a, t\rangle$ for each atom a and truth value t such that a has been assigned t. Each node of the graph is associated with a *decision level*, which is set to the level of the backtracking tree when t is assigned to a. Moreover, $\mathcal{G}$ has a directed arc connecting a node $\langle a, t\rangle$ to a node $\langle a', t'\rangle$ whenever $\langle a, t\rangle$ is one of the reasons that lead to the derivation of the truth value t' for the atom a'. Note that $\mathcal{G}$ will contain at most one node for each atom of the program, unless a conflict is derived. The way of building $\mathcal{G}$ is described below.

Forward Inference. This is essentially modus ponens. When the body of a rule r is true w.r.t. the current partial interpretation, and all but one of the head atoms of r are false and the remaining one is undefined, then there is only one way to satisfy r, by deriving the remaining head atom as true.

Concerning the Implication Graph $\mathcal{G}$, it is updated as follows. Let r be of the form (1) and let p_i be the undefined atom in $H(r)$. The following elements are added to $\mathcal{G}$: a node $\langle p_i, \mathcal{T}\rangle$; arcs $(\langle q_k, \mathcal{T}\rangle, \langle p_i, \mathcal{T}\rangle)$ ($k = 1, \ldots, j$); arcs $(\langle q_k, \mathcal{F}\rangle, \langle p_i, \mathcal{T}\rangle)$ ($k = j+1, \ldots, m$); arcs $(\langle p_k, \mathcal{F}\rangle, \langle p_i, \mathcal{T}\rangle)$ ($k = 1, \ldots, n$ and $k \neq i$).

Kripke-Kleene Negation. This derives negative information by using supportedness, the fact that each atom a which is true in a stable model M must occur in at least one rule r such that $B(r)$ is true w.r.t. M and a is the only atom in $H(r)$ which is true w.r.t. M. Hence, atoms with no candidate supporting rules can be derived to be false. So, if all of the rules r such that $a \in H(r)$ are satisfied because of a false body literal or because of a true head atom different from a, atom a is inferred as false.

Concerning $\mathcal{G}$, a node $\langle a, \mathcal{F}\rangle$ is introduced. Moreover, for each rule r with $a \in H(r)$, let L be the first literal (in chronological order of derivation) that satisfied r. If $L \in B^+(r)$, an arc $(\langle L, \mathcal{F}\rangle, \langle a, \mathcal{F}\rangle)$ is added to $\mathcal{G}$; otherwise, if $L \in H(r)$, an arc $(\langle L, \mathcal{T}\rangle, \langle a, \mathcal{F}\rangle)$ is added to $\mathcal{G}$; otherwise, $\overline{L} \in B^-(r)$ and thus an arc $(\langle \overline{L}, \mathcal{T}\rangle, \langle a, \mathcal{F}\rangle)$ is added to $\mathcal{G}$.

Contraposition for True Heads. Supportedness is also used by this inference rule: If an atom a that has been derived as true has only one candidate supporting rule r, the truth of all literals in $B(r)$ and the falsity of all atoms in $H(r)$ different from a are inferred.

Concerning $\mathcal{G}$, the following new nodes and arcs are introduced: $\langle b, \mathcal{T}\rangle$ (for each $b \in B^+(r)$); $\langle b, \mathcal{F}\rangle$ (for each $b \in B^-(r) \cup H(r) \setminus \{a\}$); for each new node $\langle b, v\rangle$ an arc $(\langle a, \mathcal{T}\rangle, \langle b, v\rangle)$. Moreover, for each rule r' such that $a \in H(r')$, let L be the first literal (in chronological order of derivation) that satisfied r'. If $L \in B^+(r')$, an arc $(\langle L, \mathcal{F}\rangle, \langle b, v\rangle)$ is added to $\mathcal{G}$, otherwise, if $L \in B^-(r') \cup H(r') \setminus \{a\}$, an arc $(\langle L, \mathcal{T}\rangle, \langle b, v\rangle)$ is added to $\mathcal{G}$; this is done for each new node $\langle b, v\rangle$ introduced by the application of the inference rule for r.

Contraposition for False Heads. This inference rule is essentially modus tollens. When for a rule r all head atoms are false, the only way to satisfy r is by having a false body. In case all but one body literals of r are true, falsity of the remaining L is inferred.

Concerning $\mathcal{G}$, a node $\langle a, v\rangle$ is added, where a is the atom in L and $v = \mathcal{F}$ if $L = a$ or $v = \mathcal{T}$ if $L = \text{not } a$. Moreover, the following arcs are added to $\mathcal{G}$: $(\langle b, \mathcal{F}\rangle, \langle a, v\rangle)$ (for each $b \in H(r) \cup B^-(r) \setminus \{a\}$; $(\langle b, \mathcal{T}\rangle, \langle a, v\rangle)$ (for each $b \in B^+(r) \setminus \{a\}$).

Well-founded Negation. Unfounded sets are sets of unsupported or self-supporting atoms, that is, atoms that can have supporting rules only if their own truth is assumed. It is well-known that unfounded sets are disjoint from stable models, which allows for assuming the falsity of all the atoms that belong to some unfounded set. Hence, after the propagation process has been carried out, $wasp$ determines the set X of all the atoms belonging to some unfounded set and derives the falsity of these atoms; if this set is empty, the rule does not apply.

In order to model such a lack of external supporting rules, a number of nodes and arcs is added to $\mathcal{G}$. For each $a \in X$, a node $\langle a, \mathcal{F}\rangle$ is added. Arcs are introduced according to the following schema: Let C be the set of atoms in X that were previously derived as true, and let c be a randomly selected atom in C. For each $a \in X \setminus C$, an arc $(\langle a, \mathcal{F}\rangle, \langle c, \mathcal{F}\rangle)$ is added to $\mathcal{G}$. Moreover, for each $b \in C \setminus \{c\}$ and for each rule r such that $b \in H(r)$, let L be the first literal (in chronological order of derivation) that satisfied r. If $L \in B^+(r)$, an arc $(\langle L, \mathcal{F}\rangle, \langle c, \mathcal{F}\rangle)$ is added to $\mathcal{G}$; otherwise, if $L \in H(r)$, an arc $(\langle L, \mathcal{T}\rangle, \langle c, \mathcal{F}\rangle)$ is added to $\mathcal{G}$; otherwise, $\overline{L} \in B^-(r)$ and thus an arc $(\langle \overline{L}, \mathcal{T}\rangle, \langle c, \mathcal{F}\rangle)$ is added to $\mathcal{G}$.

3.2 Constraint Learning

Constraint 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) [24]. A node n in the Implication Graph is a UIP for a decision level d iff all paths from the literal chosen at the level d to a conflict atom pass through n. Intuitively, a UIP is the most concise reason for the conflict of a certain decision level. We calculate the first UIP only for the decision level of the conflict. 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. After each conflict at the decision level d, a constraint is learned that contains the first UIP and all atoms of lower levels that are connected to a node between the first UIP and the conflict.

Since the number of learned constraints may become exponential in the size of the program, we adopt the standard technique of expiring learned constraints. Our policy is similar to Minisat's [28]: Each learned constraint has an activity value, measuring how much it is involved in conflicts. If a learned constraint has recently been used for propagation, we do not delete it. If the number of learned constraints is greater than one third of the input program, then we delete half of the learned constraints. Moreover, we also delete all learned constraints with an activity value lower than a threshold value.

4 Heuristics

Clearly, a crucial issue in the Model Generator function in Fig. 1 is the selection of a literal when all inferences have been made and there are still undefined atoms. It is clear that the correctness of the algorithm reported in Fig. 1 does not depend on the strategy in which this selection is made, but making a "good" choice is very important for practical efficiency. However, strategies which perform very well on some domains may perform very bad for other domains, and of course an optimal strategy seems unlikely to be found. For this reason, some heuristic must be adopted; the quality of the adopted heuristic can often only be assessed empirically.

Heuristics 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 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. This approach has already been adopted in the context of SAT, for example in the BerkMin solver [31]. In this sense, our heuristic could be seen as an extension of the heuristic implemented in BerkMin to the framework of ASP.

In the remainder of this section, we will provide a few additional details on the strategy adopted by $wasp$ for selecting undefined atoms and truth values to be assumed during the computation of stable models.

A counter $cl(L)$ is associated with each literal L. Initially, all of these counters are set to zero. When a new constraint is learned, counters for all literals occurring in the constraint are increased by one. In this way, $wasp$ keeps track of those literals occurring more frequently in learned constraints. Moreover, counters are also updated during the computation of the First UIP: If a literal L is traversed in the implication graph, the associated counter $cl(L)$ is increased by one. In this way, those literals that mainly caused the derivation of a conflict are identified. Finally, every 100 conflicts, all these counters are divided by 4 (this is an experimentally determined parameter), which gives more importance to recently active literals. Our heuristic will first select an atom and then a truthvalue for this atom. To this end, we will use $cv(a) := cl(a) + cl(\text{not } a)$, for each propositional atom a.

Learned constraints are stored in chronological order. The atom selection is first restricted to those undefined atoms that occur in the first (if any) learned constraint r with undefined body. Among those, the atom with the highest $cv(\cdot)$ value is chosen. In case of ties, the atom removing the highest number of supporting rules is selected[2].

[2] An atom removes a supporting rule if it makes the body of r false or the head of r true

If two or more atoms remove the same number of supporting rules, the first processed atom is chosen. In this way, the chances of achieving a conflict increases, and this may help the learning process. If no learned constraints with undefined body exist, the undefined atom with the highest $cv(\cdot)$ value is selected. In case of ties, the first processed atom is selected. If there are no learned constraints, e.g. in the beginning of the solving process, the atom occurring in most rules is picked.

After selecting an atom a according to the strategy described above, $wasp$ chooses a truth value for a. For this purpose, we only distinguish two cases, namely whether a learned constraint r with undefined body exists or not. If a learned constraint r with undefined body exists, additional counters are considered for choosing a truth value for a. In particular, a counter $gcl(L)$ is associated with each literal L for estimating the global contribution of L to all of the conflicts derived during the computation. For each literal L, $gcl(L)$ is initially set to zero and increased whenever $cl(L)$ is increased. The difference to $cl(L)$ is that $gcl(L)$ is never decreased, that is, $gcl(L)$ is unchanged when $cl(L)$ is divided by 4. Thus, in this case $wasp$ assumes the truth of a if $gcl(a) > gcl(\text{not } a)$; otherwise, if $gcl(a) \leq gcl(\text{not } a)$, the falsity of a is assumed. It is important to emphasize that this counter is not used when the atom removing the highest number of supporting rules was chosen. In fact, in this case the literal removing the highest number of supporting rules is picked. In the other case, that is, if all learned constraints have false bodies, a look-ahead step is performed and both a and $\text{not } a$ are propagated (i.e., the function Propagate is invoked). The literal appearing in more rules is propagated before the other one. During these propagations, $wasp$ estimates the impact of the two assumptions on the computation of answer sets. In particular, $wasp$ counts the number of inferred atoms and the number of rules that have been satisfied by the two propagations. The truth of a is then assumed if the impact of the propagation of a is greater than the impact of the propagation of $\text{not } a$, while a is assumed to be false in other case, that is, if the impact of the propagation of $\text{not } a$ is greater than the impact of the propagation of a. If the impact is equal then a is assumed to be false. It is important to note that when a conflict is derived in one of the two propagations, a deterministic inference is determined. That is, if a conflict is derived during the propagation of a, the falsity of a is determined, while the truth of a is determined whenever a conflict is derived during the propagation of $\text{not } a$.

Example 1. We will now provide an example of the way our heuristic works. In the example, we will consider the following rules r_1–r_4 and learned constraints c_1–c_2 (listed in chronological order):

$$\begin{array}{llll} r_1: & a \texttt{ :- } c. & r_3: & a \vee c \texttt{ :- } e. \\ r_2: & a \vee b \texttt{ :- } d. & r_4: & e \vee b \texttt{ :- } c. \end{array} \qquad \begin{array}{ll} c_1: & \texttt{:- } a,\ b. \\ c_2: & \texttt{:- } a,\ \text{not } c,\ d. \end{array}$$

Moreover, let us assume a partial interpretation $I_1 = \{a,\ \text{not } b\}$ and the following counter values: $cl(a) = 2$, $cl(\text{not } a) = 2$, $cl(b) = 1$, $cl(\text{not } b) = 0$, $cl(c) = 1$, $cl(\text{not } c) = 2$, $cl(d) = 3$ and $cl(\text{not } d) = 0$.

Note that constraint c_1 is satisfied because b is false. Thus, the first learned constraint (according to the chronological order) which is not satisfied is c_2. Indeed, two undefined literals occur in the body of c_2, namely $\text{not } c$ and d. We then consider the counters

$cv(c) = cl(c) + cl(\text{not } c)$ and $cv(d) = cl(d) + cl(\text{not } d)$, which are both equal to 3. The heuristics then examines the removal of supporting rules:

Two supporting rules would be removed (r_1 and r_4) by setting c false, and one supporting rule (r_3) would be removed by setting c true, for a total of 3 supporting rules removed. Concerning d, one supporting rule (r_2) would be removed by setting d false, and no rules would be removed by setting d true, for a total of 1 supporting rule removed. Therefore c removes more supporting rules than d, and therefore our heuristic will choose c and it first will be set to false.

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;[3] a discussion on the difference between $wasp$ and these two systems is provided in Section 6.

The machine used for the experiments is a two-processor Intel Xeon "Woodcrest" (quad core) 3GHz machine 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 [17] and other problems which have already been employed for assessing performance of the ASP solver DLV [4]. 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 pushes of the labyrinth's rows and columns such that the avatar can reach the goal field (which changes its location when pushed) from its starting field (which also changes 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 move 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.

[3] Winners of the disjunctive tracks in the last ASP Competitions [15–17].

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 [32].

Strategic Companies. Strategic companies is a well-known NP^{NP}-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', the associated company c_i must belong to C' (for each $i = 1, \ldots, m$). We considered a random instance having 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 [4], which is based on a reduction presented in [33]. 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 [34], 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. 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 [34], by using the function $random\_graph(85, 700, 0, 0, 0, 0, 0, 1, 1, 33)$, having 700 edges and 85 nodes; the other instances 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) sitting on a table. The goal is to build one or more vertical stacks of blocks. The catch is that only one block may be moved at a time: it may either be placed on the table or placed atop 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 $\mathcal{T}$ and $\mathcal{F}$ values 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.

Towers of 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 was considered of 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 is able to capture 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 1, reporting, for each considered instance the execution times in seconds elapsed by each considered system. 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 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.

Table 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**

Overall, the results of the preliminary experimental analysis are encouraging: the performance of $wasp$ is comparable to 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 outperforms the competitor systems, solving the hardest instance (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 remaining 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 [35].

6 Related Work and Conclusion

In this paper we provided a preliminary 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 [18], *clause learning* [19, 20], *backjumping* [21, 22], *restarts* [23], and *conflict-driven heuristics* [24] in the style of Berkmin [25]. Actually, some of the techniques adopted in $wasp$, including *backjumping* and *look back heuristics* were first introduced for Constraint Satisfaction [21, 22, 36] and successively successfully applied in SAT [37, 38, 25, 24] and QBF solving [39–42]. Some of these tech-

niques were already adapted in modern non-disjunctive ASP solvers like Smodels$_{cc}$ [43, 44], Clasp [8], and solvers supporting disjunction like CModels3 [10], GnT [45], and DLV [46, 47].

Concerning other ASP solvers, we differ from non-native solvers like Cmodels3 [10], in the sense that we do not rely on a rewriting into a propositional formula and an external SAT solver, but use native ASP techniques. Among native solvers, similarities with DLV [4] 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 [46] and extended in [47], 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 [45], which, as DLV, implements a systematic backtracking without learning and look-ahead heuristics.

Comparing our system with ClaspD (a disjunction-supporting version built upon Clasp) 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 [25]; 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.

It is worth pointing out that the implementation of $wasp$ is still in a preliminary phase, yet the results obtained up to now are encouraging. Our system is able to compete with the state-of-the-art solvers, and even outperform them in some of the considered benchmarks.

Concerning future work, we plan to extend the prototypical system by introducing new language constructs such as aggregates [48, 49] and weak constraints [50], 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 [35] as well as support for multi-threading are also planned.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
2. Lifschitz, V.: Answer Set Planning. In: ICLP'99, Las Cruces, New Mexico, USA, The MIT Press (1999) 23–37
3. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22** (1997) 364–418

4. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7** (2006) 499–562
5. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. AI **138** (2002) 181–234
6. Lin, F., Zhao, Y.: ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In: AAAI-2002, Edmonton, Alberta, Canada, AAAI Press / MIT Press (2002)
7. Babovich, Y., Maratea, M.: Cmodels-2: Sat-based answer sets solver enhanced to non-tight programs. `http://www.cs.utexas.edu/users/tag/cmodels.html` (2003)
8. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI 2007,(2007) 386–392
9. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.H.: Unfolding Partiality and Disjunctions in Stable Model Semantics. ACM TOCL **7** (2006) 1–37
10. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: LPNMR'05. LNCS 3662, (2005) 447–451
11. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-Driven Disjunctive Answer Set Solving. In: Proc. of KR 2008, Sydney, Australia, AAAI Press (2008) 422–432
12. Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., Leone, N.: Team-building with Answer Set Programming in the Gioia-Tauro Seaport. TPLP. *CUP* (2011) To appear.
13. Manna, M., Ruffolo, M., Oro, E., Alviano, M., Leone, N.: The HiLeX System for Semantic Information Extraction. Transactions on Large-Scale Data and Knowledge-Centered Systems. *Berlin/Heidelberg* (2011) To appear.
14. Ricca, F., Alviano, M., Dimasi, A., Grasso, G., Ielpa, S.M., Iiritano, S., Manna, M., Leone, N.: A Logic-Based System for e-Tourism. FI. *IOS Press* **105** (2010) 35–55
15. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: LPNMR'07. LNCS 4483, (2007) 3–17
16. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming competition. In: Proc. of LPNMR '09, Berlin, Heidelberg, (2009) 637–654
17. Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., Cozza, S., Faber, W., Febbraro, O., Leone, N., Manna, M., Martello, A., Panetta, C., Perri, S., Reale, K., Santoro, M.C., Sirianni, M., Terracina, G., Veltri, P.: The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track. In: Proc. of LPNMR11., LNCS (2003) 388–403
18. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem Proving. Communications of the ACM **5** (1962) 394–397
19. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In: ICCAD 2001. (2001) 279–285
20. Pipatsrisawat, K., Darwiche, A.: On Modern Clause-Learning Satisfiability Solvers. JAIR **44** (2010) 277–301
21. Gaschnig, J.: Performance measurement and analysis of certain search algorithms. PhD thesis, CMU (1979) Tech. Report CMU-CS-79-124.
22. Prosser, P.: Hybrid Algorithms for the Constraint Satisfaction Problem. Computational Intelligence **9** (1993) 268–299
23. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting Combinatorial Search Through Randomization. In: Proceedings of AAAI/IAAI 1998, AAAI Press (1998) 431–437
24. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC 2001 (2001) 530–535
25. Goldberg, E., Novikov, Y.: 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

26. Faber, W., Leone, N., Pfeifer, G.: Pushing Goal Derivation in DLP Computations. In: LPNMR'99. LNCS 1730, (1999) 177–191
27. Koch, C., Leone, N., Pfeifer, G.: Enhancing Disjunctive Logic Programming Systems by SAT Checkers. AI **15** (2003) 177–212
28. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003., LNCS (2003) 502–518
29. Maratea, M., Ricca, F., Veltri, P.: DLV^C: Enhanced Model Checking in DLV. In: Proceedings of Logics in Artificial Intelligence, JELIA 2010. (2010) 365–368
30. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. Inf. Process. Lett. **47** (1993) 173–180
31. Goldberg, E., Novikov, Y.: Berkmin: A fast and robust sat-solver. Discrete Appl. Math. **155** (2007) 1549–1561
32. Alviano, M.: The Maze Generation Problem is NP-complete. In: Proc. of ICTCS '09. (2009)
33. Eiter, T., Gottlob, G.: On the Computational Cost of Disjunctive Logic Programming: Propositional Case. AMAI **15** (1995) 289–323
34. Knuth, D.E.: The Stanford GraphBase : A Platform for Combinatorial Computing. ACM Press, New York (1994)
35. Pfeifer, G.: Improving the Model Generation/Checking Interplay to Enhance the Evaluation of Disjunctive Programs. In: LPNMR-7. LNCS 2923, (2004) 220–233
36. Dechter, R., Frost, D.: Backjump-based backtracking for constraint satisfaction problems. AI **136** (2002) 147–188
37. Bayardo, R., Schrag, R.: 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
38. Silva, J.P.M., Sakallah, K.A.: GRASP: A Search Algorithm for Propositional Satisfiability. IEEE Transaction on Computers **48** (1999) 506–521
39. Zhang, L., Malik, S.: Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. In: Proc. of ICCAD 2002. (2002) 442–449
40. Zhang, L., Malik, S.: Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In: CP 2002. NY, USA, (2002) 200–215
41. Giunchiglia, E., Narizzano, M., Tacchella, A.: Backjumping for Quantified Boolean Logic Satisfiability. AI **145** (2003) 99–120
42. Letz, R.: Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In: TABLEAUX 2002. Denmark, (2002) 160–175
43. Ward, J., Schlipf, J.S.: Answer Set Programming with Clause Learning. In: LPNMR-7. LNCS 2923, (2004) 302–313
44. Ward, J.: Answer Set Programming with Clause Learning. PhD thesis, Ohio State University, Cincinnati, Ohio, USA (2004)
45. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In: LPNMR-7. LNCS 2923, Fort Lauderdale, Florida, USA, (2004) 331–335
46. Ricca, F., Faber, W., Leone, N.: A Backjumping Technique for Disjunctive Logic Programming. AI Communications **19** (2006) 155–172
47. Maratea, M., Ricca, F., Faber, W., Leone, N.: 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
48. Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and Stable Semantics of Logic Programs with Aggregates. TPLP **7** (2007) 301–353
49. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. AI **175** (2011) 278–298 Special Issue: John McCarthy's Legacy.
50. Buccafurri, F., Leone, N., Rullo, P.: Enhancing Disjunctive Datalog by Constraints. IEEE TKDE **12** (2000) 845–860