WASP: A Native ASP Solver Based on Constraint Learning*

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

Department of Mathematics and Computer Science,
University of Calabria, 87036 Rende, Italy
{alviano,dodaro,faber,leone,ricca}@mat.unical.it

Abstract. This paper introduces WASP, an ASP solver handling disjunctive logic programs under the stable model semantics. WASP implements techniques originally introduced for SAT solving that have been extended to cope with ASP programs. Among them are restarts, conflict-driven constraint learning and backjumping. Moreover, WASP combines these SAT-based techniques with optimization methods that have been specifically designed for ASP computation, such as source pointers enhancing unfounded-sets computation, forward and backward inference operators based on atom support, and techniques for stable model checking. Concerning the branching heuristics, WASP adopts the BerkMin criterion hybridized with look-ahead techniques. The paper also reports on the results of experiments, in which WASP has been run on the system track of the third ASP Competition.

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.

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 [2]. 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. [3–5]).The availability of robust implementations made ASP a powerful tool for developing advanced applications in the areas of Artificial Intelligence, Information Integration, and Knowledge Management; for example, ASP has been used in industrial applications [6], and for team-building [7], semantic-based information extraction [8], and e-tourism [9]. These applications of ASP have confirmed the viability of the use of ASP. Nonetheless, the interest in developing more

* This research has been partly supported by project PIA KnowRex POR FESR 2007- 2013 BURC n. 49 s.s. n. 1 16/12/2010, by MIUR project FRAME PON01_02477/4, and by the European Commission, European Social Fund and Regione Calabria.

P. Cabalar and T.C. Son (Eds.): LPNMR 2013, LNAI 8148, pp. 54–66, 2013.

effective and faster systems is still a crucial and challenging research topic, as witnessed by the results of the ASP Competition series (see e.g. [10]).

This paper provides a contribution in the aforementioned context. In particular, we present 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 [11], *clause learning* [12], *backjumping* [13], *restarts* [14], and *conflict-driven heuristics* [15] in the style of BerkMin [16]. The mentioned SAT-solving methods have been adapted and combined with state-of-the-art pruning techniques adopted by modern native disjunctive ASP systems [3–5]. In particular, the role of Boolean Constraint Propagation in SAT-solvers (based only on *unit propagation* inference rule) is taken by a procedure combining several 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., [17]). In particular, WASP implements techniques specifically designed for ASP computation, such as source pointers [18] enhanced unfounded-set computation, *native* forward and backward inference operators based on atom support [17]. Moreover, WASP uses a branching heuristics 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 implemented relying on the rewriting method of [19] and by calling MiniSAT [20].

In the following, after briefly introducing ASP, we describe the new system WASP, whose source available at `http://www.mat.unical.it/ricca/wasp`. We start from the solving strategy and present the design choices regarding propagation, constraint learning, restarts, and the heuristics. We also report on an experiment in which we have run WASP on all instances used in the third ASP Competition [10]. In particular, we compare our system with all participants and analyze in detail the impact of our design choices. Finally, we discuss related work and draw the conclusion.

2 Preliminaries

Let $\mathcal{A}$ be a countable set of propositional atoms. A *literal* is either an atom (a positive literal), or an atom preceded by the *negation as failure* symbol not (a negative literal). A *program* is a finite set of rules of the following form:

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

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 called head, and the conjunction $q_1, \ldots, q_j, \mathrm{not}\ q_{j+1}, \ldots, \mathrm{not}\ q_m$ is referred to as body. For a rule r of the form (1), the following notation is also used: $H(r)$ denotes the set of head atoms; $B(r)$ denotes the set of body literals; $B^+(r)$ and $B^-(r)$ denote the set of atoms appearing in positive and negative body literals, respectively; $C(r) := \overline{H(r)} \cup B(r)$ is the nogood representation of r [4]. In the following a rule r is said to be regular if $|H(r)| \geq 1$, and a constraint if $|H(r)| = 0$. Moreover, the complement of a literal ℓ is denoted $\overline{\ell}$, i.e., $\overline{a} = \mathrm{not}\ a$ and $\overline{\mathrm{not}\ a} = a$ for an atom a. This notation extends to sets of literals, i.e., $\overline{L} := \{\overline{\ell} \mid \ell \in L\}$ for a set of literals L.

An *interpretation* I is a set of literals, i.e., $I \subseteq \mathcal{A} \cup \overline{\mathcal{A}}$. Intuitively, literals in I are true, literals whose complements are in I are false, and all other literals are undefined. I is total if there are no undefined literals, and I is inconsistent if there is $a \in \mathcal{A}$ such that $\{a, \text{not } a\} \subseteq I$. An interpretation I satisfies a rule r if $C(r) \cap \overline{I} \neq \emptyset$, while I violates r if $C(r) \subseteq I$. A *model* of a program $\mathcal{P}$ is a total interpretation satisfying all rules of $\mathcal{P}$. The semantics of a program $\mathcal{P}$ is given by the set of its answer sets (or stable models) [1], 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 the reduct $\mathcal{P}^M$ obtained by deleting from $\mathcal{P}$ each rule r such that $B^-(r) \cap I \neq \emptyset$, and then by removing all the negative literals from the remaining rules.

3 Answer Set Computation

In this section we review the algorithms and the heuristics implemented in WASP. For reasons of presentation, we have considerably simplified the procedures in order to focus on the main principles.

3.1 Main Algorithm

An answer set of a given propositional program $\mathcal{P}$ is computed in WASP by using Algorithm 1, which is similar to the Davis-Putnam procedure in SAT solvers. The process starts with an empty interpretation I in input. Function Propagate extends I with those literals that can be deterministically inferred (line 2) and keeps track of the reasons of each inference by building a representation of the so-called *implication graph* [15]. This function is similar to unit propagation as employed by SAT solvers, but also uses 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 3.2, returns false if an inconsistency (or conflict) is detected, true otherwise. If Propagate returns true and I is total (line 3), *CheckModel* is invoked (line 4) to verify that I is an answer set by using the techniques described in [19]. In particular, for non head-cycles-free programs the check is co-NP-complete [21] and implemented by a call to the SAT solver MiniSAT [20]. If the stability check succeeds, I is returned; otherwise, I contains some unfounded sets which are analyzed by the procedure *AnalyzeConflictAndLearnConstraints* (described later). Otherwise, if there are undefined literals in I, a heuristic criterion is used to chose one, say ℓ. Then computation proceeds with a recursive call to *ComputeAnswerSet* on $I \cup \{\ell\}$ (lines 6–7). In case the recursive call returns an answer set, the computation ends returning it (lines 8–9). Otherwise, the algorithm unrolls choices until consistency of I is restored (backjumping; lines 10–11), and the computation resumes by propagating the consequences of constraints learned by the conflict analysis. Conflicts detected during propagation are analyzed by procedure *AnalyzeConflictAndLearnConstraints* (line 12; described in Section 3.3).

This general procedure is usually complemented with some heuristic techniques that control the number of learned constraints (which may be exponential in number), and possibly restart the computation to explore different branches of the search tree. Our restart policy is based on the sequence of thresholds introduced in [22], while our learned constraint elimination policy is described in Section 3.4.

Algorithm 1. Compute Answer Set

```
   Input  : An interpretation I for a program P
   Output: An answer set for P or Incoherent
1  begin
2      while Propagate(I) do
3          if I is total then
4              if CheckModel(I) then return I;
5              break;  // goto 12
6          ℓ := ChooseUndefinedLiteral();
7          I' := ComputeAnswerSet(I ∪ {ℓ});
8          if I' ≠ Incoherent then
9              return I';
10         if there are violated learned constraints then
11             return Incoherent;
12     AnalyzeConflictAndLearnConstraints(I);
13     return Incoherent;
```

3.2 Propagation

WASP implements a number of deterministic inference rules for pruning the search space during answer set computation. These propagation rules are named *unit*, *support*, and *well-founded*. During the propagation of deterministic inferences, implication relationships among literals are stored in the implication graph. Each node ℓ in the implication graph is labelled with a *decision level* representing the number of nested calls to AnswerSetComputation at the point in which ℓ has been derived. Note that the implication graph contains at most one node for each atom unless a conflict is derived, in which case for some atom a both a and its negation are in the graph. In the following, we describe the propagation rules and how the implication graph is updated in WASP.

Unit Propagation. An undefined literal ℓ is inferred by unit propagation if there is a rule r that can be satisfied only by ℓ, i.e., r is such that $\overline{\ell} \in C(r)$ and $C(r) \setminus \{\overline{\ell}\} \subseteq I$. In the implication graph we add node ℓ, and arc (ℓ', ℓ) for each literal $\ell' \in C(r) \setminus \{\overline{\ell}\}$.

Support Propagation. Answer sets are supported models, i.e., for each atom a in an answer set there is a (supporting) rule r such that $a \in H(r)$, $B(r) \subseteq I$ and $H(r) \cap I = \{a\}$. Support is on the basis of two propagation rules named *forward* and *backward*.

Forward propagation derives as false all atoms for which there are no candidate supporting rules. More formally, literal $\mathrm{not}\ a$ is derived if for each rule r having a in the head $\overline{I} \cap C(r) \setminus \{\mathrm{not}\ a\} \neq \emptyset$ holds. In the implication graph, a node $\mathrm{not}\ a$ is introduced, and for each rule r having a in the head an arc $(\overline{\ell}, \mathrm{not}\ a)$, where $\ell \in C(r) \setminus \{\mathrm{not}\ a\}$, is added. Within WASP, literal ℓ is the first literal that satisfied r in chronological order of derivation, which is called *first satisfier* of r in the following.

Backward propagation occurs when for a true atom there is only one candidate supporting rule. More in detail, if there are an atom $a \in I$ and a rule r such that $a \in H(r)$ and for each other rule r' having a in the head $\overline{I} \cap C(r') \setminus \{\mathrm{not}\ a\} \neq \emptyset$ holds, then all literals in $C(r) \setminus I$ are inferred. Concerning the implication graph, we add node ℓ and

arc (a, ℓ) for each $\ell \in C(r) \setminus I$. Moreover, for each $\ell \in C(r) \setminus I$ and r' having a in the head and different from r, we add an arc (ℓ', ℓ), where ℓ' is the first satisfier of r'.

Well-Founded Propagation. Self-supporting truth is not admitted in answer sets. According to this property, a set X of atoms is *unfounded* if for each r such that $H(r) \cap X \neq \emptyset$ at least one of the following conditions is satisfied: (i) $B(r) \cap \overline{I} \neq \emptyset$; (ii) $B^+(r) \cap X \neq \emptyset$; (iii) $I \cap H(r) \setminus X \neq \emptyset$. Intuitively, atoms in X can have support only by themselves, and can thus be derived false.

To compute unfounded sets we adopted *source pointers* [18]. Roughly, for each atom we set a rule r as its candidate supporting rule, referred to as its source pointer. Source pointers are constrained to not introduce self-supporting atoms, and are updated during the computation. Atoms without source pointers form an unfounded set and are thus derived false. Concerning the implication graph, for each atom $a \in X$, node $\mathrm{not}\ a$ is added. Moreover, for each $a \in X$ and for each rule r having a in the head and first satisfier ℓ of r ($\ell \notin X$), arc $(\ell, \mathrm{not}\ a)$ is added. Note that since $\ell \notin X$ the implication graph is acyclic.

3.3 Constraint Learning

Constraint learning acquires information from conflicts in order to avoid exploring the same search branch several times. In WASP there are two causes of conflicts: failed propagation and stability check failures.

Learning from Propagation. In this case, our learning schema is based on the concept of the first Unique Implication Point (UIP) [15]. A node n in the implication graph is a UIP for a decision level d if all paths from the literal chosen at the level d to the conflict literals pass through n. We calculate UIPs only for the decision level of conflicts, and more precisely the one closest to the conflict, which is called the first UIP. Our learning schema is as follows: Let u be the first UIP. Let L be the set of literals different form u occurring in a path from u to the conflict literals. The learned constraint comprises u and each literal ℓ such that the decision level of ℓ is lower than the one of u and there is an arc (ℓ, ℓ') in the implication graph for some $\ell' \in L$.

Learning from Model Check Failure. Answer set candidates are checked for stability by function *CheckModel* in Algorithm 1. If a model M is not stable, an unfounded set $X \subseteq M$ is computed. X represents the reason for the stability check failure. Thus, we learn a constraint c containing all atoms from X and first satisfiers of possible supporting rules for atoms in X. More formally, a literal ℓ is in $B(c)$ if either $\ell \in X$ or ℓ is the first satisfier of some rule r s.t. $H(r) \cap X \neq \emptyset$ and $B^+(r) \cap X = \emptyset$.

3.4 Heuristics

A crucial role is played by the heuristic criteria used for both selecting branching literals and removing learned constraints.

Branching Heuristic. Concerning the branching heuristics, implemented by function ChooseUndefinedLiteral in Algorithm 1, we adopt a mixed approach between look-back and look-ahead techniques. The idea is to record statistics on atoms involved in

```
Function ChooseUndefinedLiteral
    Output: A branching literal
1   begin
2       if there is no learned constraint then
3           a := MostOccurrentAtom();
4           return MostOccurrentPolarity(a);
5       if there is an undefined learned constraint then
6           c := MostRecentUndefinedLearnedConstraint() ;
7           Candidates := AtomsWithHighestCV(c);
8           if |Candidates| = 1 then
9               return HighestGCVPolarity(Candidates);
10          a := AtomCancellingMoreRules(Candidates);
11          return PolarityCancellingMoreRules(a);
12      a := AtomWithHighestCV();
13      return LookAhead(a) ;
```

constraint learning so to prefer those involved in most recent conflicts (look-back), and in some case the branching literal is chosen by estimating the effects of its propagation (look-ahead). More in detail, WASP implements a variant of the criterion used in the BerkMin SAT solver [23]. In this technique each literal ℓ is associated with counters $cv(\ell)$ and $gcv(\ell)$, initially set to zero. When a new constraint is learned, counters for all literals occurring in the constraint are increased by one. Moreover, counters are also updated during the computation of the first UIP: If a literal ℓ is traversed in the implication graph, the associated counters are increased by one, and counters $cv(\cdot)$ are divided by 4 every 100 conflicts. Thus, literals that are often involved in conflicts will have larger values of $cv(\cdot)$ and $gcv(\cdot)$, where counters $cv(\cdot)$ give more importance to literals involved in recent conflicts.

The branching criterion is reported in function ChooseUndefinedLiteral. Initially, there is no learned constraint (line 2), and the algorithm selects the atom, say a, occurring most frequently in rules. Then, the most occurrent literal of a and $\mathrm{not}\ a$ is returned. After learning some constraints, two possible scenarios may happen. If there are undefined learned constraints (line 5), the one that was learned more recently, say c, is considered, and the atoms having the highest value of $cv(\cdot)$ are *candidate* choices. If there is only one candidate, say a, then the literal between a and $\mathrm{not}\ a$ having the maximum value of $gcv(\cdot)$ is returned. (If $gcv(a) = gcv(\mathrm{not}\ a)$ then $\mathrm{not}\ a$ is returned.) If there are several candidates, an ASP specific criterion is used for estimating the effect of the candidates on the number of potentially supporting rules. In particular, let a be an atom occurring most often in unsatisfied regular rules. The heuristic chooses the literal between a and $\mathrm{not}\ a$ that satisfies the largest number of rules.

The second scenario happens when all learned constraints are satisfied. In this case one atom, say a, having the highest value of $cv(\cdot)$ is selected, and a look-ahead procedure is called to determine the most promising polarity (lines 12–13). Actually, a look-ahead step is performed by propagating both a and $\mathrm{not}\ a$, and the impact of the two assumptions on answer set computation is estimated by summing up the number

of inferred atoms and the number of rules that have been satisfied. The literal having greater impact is chosen, and in case of a tie the negative literal is preferred. It is important to note that if one of the two propagations fails, the conflict is analyzed as described in Section 3.3, and a constraint is learned and propagated. Possibly, after the propagation, a new branching literal is selected applying the above criterion.

Deletion of Constraints. The number of learned constraints can grow exponentially, and this may cause a performance degradation. A heuristic is employed for deleting some of them, typically the ones that are not involved often in the more recent conflicts. To this end, learned constraints are associated with activity counters as implemented in the SAT solver MiniSAT [20]. The activity counters measure the number of times a constraint was involved in the derivation of a conflict. Once the number of learned constraints is greater than one third of the size (in number of rules) of the original program, constraint deletion is performed as follows: First, all constraints having an activity counter smaller than a threshold are removed (as in MiniSAT) if they are *unlocked*. A constraint c is *unlocked* if $C(c) \setminus I \neq \emptyset$ (roughly, c is undefined and not directly involved in propagations). If this cancellation step removes less than half of the constraints, an additional deletion step is performed. In particular, unlocked constraints having activity less than the average are removed possibly until the number of constraints halves. Note that the second cancellation step is done differently in MiniSAT; our policy seems to be effective in practice for ASP.

4 Experiments

In this section we report the results of an experiment assessing the performance of WASP. In particular, we first compare WASP with all participants of the System Track of the 3rd ASP Competition. Then, we analyze in detail the behavior of WASP in specific domains that help to understand strengths and weaknesses of our solver. The experiments were run on the very same benchmarks, hardware and execution platform used in the 3rd ASP Competition [10]. In particular, we used a four core Intel Xeon CPU X3430 2.4 GHz, with 4 GB of physical RAM and PAE enabled, running Linux Debian Lenny (32bit). As in the competition settings, WASP was benchmarked with just one of the four processors enabled, and time and memory limits set to 600 seconds and 3 GiB (1 GiB = 2^{30} bytes), respectively. Execution times and memory consumptions were measured by the same programs and scripts employed in the competition. In particular, we used the Benchmark Tool Run (`http://fmv.jku.at/run/`).

We have run WASP on the official instances of the System Track of the 3rd ASP Competition [10]. In this paper we consider only problems featuring unstratified or disjunctive encodings, thus avoiding instances that are already solved by the grounders. More in detail, we consider all problems in the NP and Beyond-NP categories, and the polynomial problems Stable Marriage and Partners Unit Polynomial. The competition suite included planning domains, temporal and spatial scheduling problems, combinatorial puzzles, graph problems, and a number of real-world domains in which ASP was applied. (See [10] for an exhaustive description of the benchmarks.)

WASP was coupled with a custom version of the DLV grounder properly adapted to work with our solver. We report the results in Table 1 together with the official results

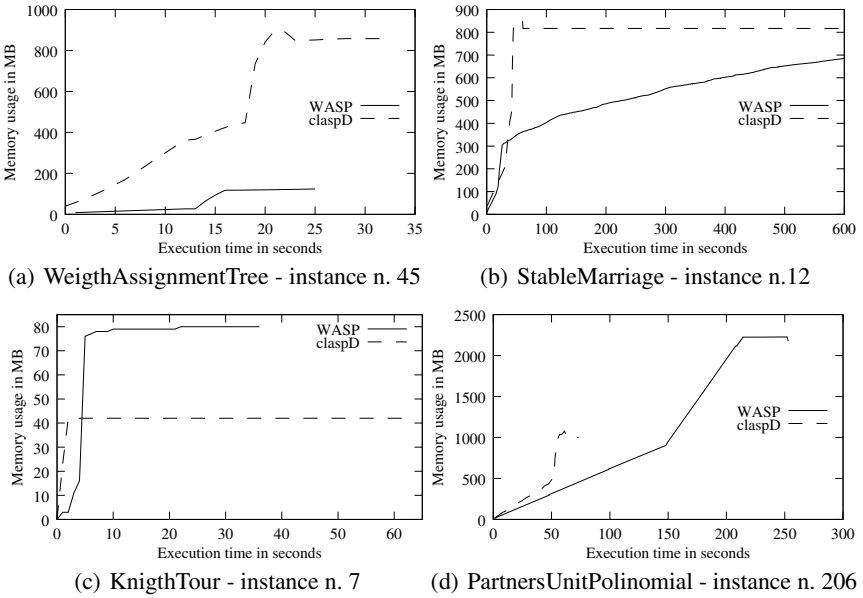
Table 1. Scores on the 3rd ASP Competition benchmark

		Cumulative				P		NP												Bnd-NP	
System		Total	P	NP	BNP	StableMarriage	PartnerUnitsPolynomial	SokobanDecision	KnightTour	DisjunctiveScheduling	PackingProblem	Labyrinth	MCS Querying	Numberlink	HanoiTower	GraphColouring	Solitaire	Weight-AssignmentTree	MazeGeneration	StrategicCompanies	MinimalDiagnosis
claspd	**Score**	**668**	**13**	**552**	**103**	0	13	68	68	30	0	65	75	69	31	19	11	20	96	12	91
	Inst	425	10	355	60	0	10	45	40	25	0	45	50	40	25	10	10	15	50	10	50
	Time	243	3	197	43	0	3	23	28	5	0	20	25	29	6	9	1	5	46	2	41
wasp	**Score**	**663**	**46**	**553**	**64**	40	6	32	68	34	0	64	73	59	36	15	37	54	81	0	64
	Inst	465	40	380	45	35	5	25	40	25	0	45	50	35	30	10	25	45	50	0	45
	Time	198	6	173	19	5	1	7	28	9	0	19	23	24	6	5	12	9	31	0	19
claspfolio	**Score**	**627**	**18**	**609**	**-**	5	13	66	65	37	0	63	75	64	47	55	21	21	95	-	-
	Inst	400	15	385	-	5	10	45	35	25	0	40	50	35	35	40	15	15	50	-	-
	Time	227	3	224	-	0	3	21	30	12	0	23	25	29	12	15	6	6	45	-	-
clasp	**Score**	**617**	**20**	**597**	**-**	6	14	78	63	38	0	78	75	65	39	23	21	21	96	-	-
	Inst	385	15	370	-	5	10	50	35	25	0	50	50	35	30	15	15	15	50	-	-
	Time	232	5	227	-	1	4	28	28	13	0	28	25	30	9	8	6	6	46	-	-
idp	**Score**	**597**	**0**	**597**	**-**	0	0	64	74	38	0	52	75	70	65	18	38	8	95	-	-
	Inst	370	0	370	-	0	0	45	45	25	0	30	50	40	45	10	25	5	50	-	-
	Time	227	0	227	-	0	0	19	29	13	0	22	25	30	20	8	13	3	45	-	-
cmodels	**Score**	**582**	**0**	**510**	**72**	0	0	67	56	21	0	62	75	30	51	29	18	6	95	0	72
	Inst	380	0	335	45	0	0	45	30	20	0	45	50	20	35	20	15	5	50	0	45
	Time	202	0	175	27	0	0	22	26	1	0	17	25	10	16	9	3	1	45	0	27
lp2diffz3	**Score**	**394**	**0**	**394**	**-**	0	0	42	55	0	0	0	70	45	47	27	25	0	83	-	-
	Inst	270	0	270	-	0	0	30	35	0	0	0	50	30	35	20	20	0	50	-	-
	Time	124	0	124	-	0	0	12	20	0	0	0	20	15	12	7	5	0	33	-	-
sup	**Score**	**357**	**11**	**346**	**-**	0	11	52	40	37	0	58	72	0	31	16	15	25	0	-	-
	Inst	250	10	240	-	0	10	35	25	25	0	40	50	0	25	10	10	20	0	-	-
	Time	107	1	106	-	0	1	17	15	12	0	18	22	0	6	6	5	5	0	-	-
lp2sat2gmsat	**Score**	**321**	**11**	**310**	**-**	0	11	36	10	32	0	46	71	22	47	17	29	0	0	-	-
	Inst	235	10	225	-	0	10	30	5	25	0	35	50	15	35	10	20	0	0	-	-
	Time	86	1	85	-	0	1	6	5	7	0	11	21	7	12	7	9	0	0	-	-
lp2sat2msat	**Score**	**307**	**5**	**302**	**-**	0	5	39	0	32	0	52	71	15	47	17	29	0	0	-	-
	Inst	225	5	220	-	0	5	30	0	25	0	40	50	10	35	10	20	0	0	-	-
	Time	82	0	82	-	0	0	9	0	7	0	12	21	5	12	7	9	0	0	-	-
lp2sat2lmsat	**Score**	**301**	**0**	**301**	**-**	0	0	35	0	32	0	53	71	17	47	17	29	0	0	-	-
	Inst	220	0	220	-	0	0	30	0	25	0	40	50	10	35	10	20	0	0	-	-
	Time	81	0	81	-	0	0	5	0	7	0	13	21	7	12	7	9	0	0	-	-
smodels	**Score**	**269**	**0**	**269**	**-**	0	0	0	55	36	0	9	53	27	0	0	0	0	89	-	-
	Inst	165	0	165	-	0	0	0	30	25	0	5	35	20	0	0	0	0	50	-	-
	Time	104	0	104	-	0	0	0	25	11	0	4	18	7	0	0	0	0	39	-	-

of all participants of the competition. The results are expressed in terms of scores computed with the same methods adopted in the competition. Roughly, the instance score can be obtained multiplying by 5 the number of solved instances within the timeout, whereas the time score is computed according to a logarithmic function of the execution times (details can be found in `http://www.mat.unical.it/aspcomp2011/files/scoringdetails.pdf`). The first column contains the total scores, followed by columns containing data aggregated first by benchmark class, and then by problem name. For each solver we report total score, instance score and time score in separate rows. A dash in the table indicates that the corresponding solver cannot handle the corresponding instances of a specific class/problem. We observe that the only solvers able to deal with Beyond-NP problems are claspD, Cmodels and WASP.

As a general comment, by looking at Table 1 we can say that WASP is comparable in performance with the best solvers in the group, and scored just 5 points less than claspD. WASP solved more instances in overall, obtaining 40 points more than claspD for the instance score. However, WASP performs worse than the five best solvers in terms of raw speed, and in particular its time score is 45 points less than claspD. If we analyze the results by problem class, we observe that WASP is the best solver in the P category, and it is comparable to claspD but follows claspfolio, clasp and idp in NP. In Beyond-NP, where claspD is the best solver, WASP solves the same instances as Cmodels but it is slower than this latter. These results already outline some advantages and weaknesses of our implementation. In particular, a weakness of WASP, also affecting claspD when compared with clasp, is that handling unrestricted disjunction can cause a reduction in performance in the NP category, which is however compensated in the total score by the additional points earned in the Beyond-NP category. In this category WASP performs similar to Cmodels, which can be justified by the similar learning strategy in case of model checking failures that the two solvers adopt. Nonetheless, both the efficiency of implementation and the interplay between the main algorithm and model checker has to be significantly improved to fill the gap with claspD. The strength of WASP in P can be explained as a combination of two factors. First, WASP uses the DLV grounder, which in some cases produces a smaller program than Gringo. The second factor is that WASP often requires less memory than the best five alternative solvers. This behavior makes a sensitive difference in this category, where the instances of Stable Marriage are large in size. We will analyze the issue of memory usage in more detail later, but it is worth mentioning that WASP runs out of memory only 23 times in total, which is less than any of the five best systems. In fact, according to the data reported on the Web site of the Competition, claspD, clasp, claspfolio, idp, and Cmodels ran out of memory 63, 61, 56, 63, and 74 times, respectively.

Analyzing the results in more detail, there are some specific benchmark problems where the differences among WASP and the five best participants in the competition are significant. In these cases, the differing behaviors can be explained by different choices made during design and implementation of solvers. Analyzing Table 1 from left to right, the first of these problems is StableMarriage, which belongs to the P category. As previously pointed out, in this category the combination of DLV (grounder) and WASP needs less memory, which explains the result. Then there is SokobanDecision, in which WASP performed worse than several other solvers. To understand the



(a) WeigthAssignmentTree - instance n. 45 (b) StableMarriage - instance n.12

(c) KnigthTour - instance n. 7 (d) PartnersUnitPolinomial - instance n. 206

Fig. 1. Memory usage in WASP and claspD

reason, we ran some additional experiments (not report due to space constraints), from which we observed that both (i) the default heuristic of WASP is not suitable for this problem; and (ii) profiling revealed that the implementation of well-founded propagation in WASP causes considerable overhead. Concerning (i), we verified that selecting a different criterion (e.g., standard BerkMin) can sensibly improve performance. Eventually, WASP solves more instances of Weight-Assignment Tree than any alternative. Here WASP, featuring a native implementation of support propagations, is advantaged over other solvers, like clasp(D), Cmodels and idp, which apply Clark's completion. This is a rewriting technique that adds additional symbols and auxiliary rules to encode the support requirement. It is known from the literature that adding these additional symbols can lead to better performance [24], nonetheless, in this case they seem to cause higher memory usage and slower propagation.

Some additional observations can be made by studying in more detail memory usage of WASP and claspD. To this end we report in Figure 1 four plots depicting the memory consumption during the solvers' execution. In particular, Figure 1(a) reports the result for an instance of WeightAssignmentTree, a problem in the NP category whose encoding is unstratified. In this case WASP performs better than claspD both in memory and time. We observed that the output of DLV is five times smaller than the output of Gringo, which can justify the memory required by claspD up to 18 seconds. At that point of execution, claspD doubles its memory consumption, which could be a side effect of Clark's completion. Figure 1(b) shows the result for an instance of StableMarriage that neither WASP nor claspD solve in the allotted time. StableMarriage is a problem in the P category and its encoding is unstratified. Also in this case we observe

that WASP is less memory demanding than claspD. Figure 1(c) depicts the results for an instance of KnightTour, a problem in the NP category whose encoding is recursive. In this case claspD requires half of the memory consumed by WASP, which is an insight that our current implementation of the well-founded propagation is not optimal in terms of memory consumption. Nonetheless, WASP is faster than claspD for the tested instance. Finally, Figure 1(d) reports the result for PartnersUnitPolinomial, a problem in the P category whose encoding is recursive. In this case claspD performs better than WASP both in memory and size, which is partially due to DLV. In fact, even if DLV and Gringo output programs of the same size for the tested instance, DLV terminated in 150 seconds, while Gringo just requires 50 seconds. Again, we note that WASP requires more than two times the memory used by claspD in this unstratified encoding.

5 Related Work

WASP is inspired by several techniques that were originally introduced for SAT solving, like the DPLL backtracking search algorithm [11], clause learning [12], backjumping [13], restarts [14], and conflict-driven heuristics [15] in the style of BerkMin [16]. Actually, some of the techniques adopted in WASP, including backjumping and look-back heuristics, were first introduced for Constraint Satisfaction, and then successfully applied to SAT and QBF solving. Some of these techniques were already adapted in modern non-disjunctive ASP solvers like Smodels$_{cc}$ [25], clasp [4], Smodels [18], and solvers supporting disjunction like Cmodels3 [5], and DLV [26].

More in detail, WASP differs from non-native solvers like Cmodels3 [5] that are based on a rewriting into a propositional formula and an external SAT solver. Nonetheless, our learning strategy for stability check failures is similar to that of Cmodels3. Concerning native solvers, WASP implements native support propagation rules and model checking techniques similar to DLV [3]. However, we implement look-back techniques borrowed from CP and SAT which are not present in DLV. In fact, DLV implements a systematic backtracking without learning and adopts look-ahead heuristics. We also mention an extension of DLV [26] that implements backjumping and look-back heuristics, which however does not include clause learning, restarts, and does not use an implication graph for determining the reasons of the conflicts. WASP uses an implication graph which is similar to the one implemented in Smodels$_{cc}$ [25]. Nonetheless, there is an important difference between these two implication graphs. In fact, the first one is guaranteed to be acyclic while the latter might be cyclic due to the well founded propagation.

Our solver is more similar to clasp and its extension to disjunctive programs claspD. In fact, source pointers, backjumping, learning, restarts, and look-back heuristics are also used by clasp(D). There are nonetheless several differences with WASP. The first difference is that clasp(D) use Clark's completion for modeling support, while WASP features a native implementation of support propagation (which caused major performance differences in our experiments). Also, minimality is handled by learning no-goods (called loop formulas) in clasp(D). It turns out that clasp(D) almost relies on unit propagation and thus uses an implication graph that is more similar to SAT solvers. Furthermore, there are differences concerning the restart policy, constraint deletion and

branching heuristics. WASP adopts as default a policy based on the sequence of thresholds introduced in [22], whereas clasp(D) employs by default a different policy based on geometric series. Concerning deletion of learned constraints, WASP adopts a criterion inspired by MiniSAT. In clasp(D) a more involved criterion is adopted, where constraints are cancelled when the size of the program grows up to a crescent threshold depending on the number of restarts. Nonetheless, the program can grow in clasp(D) up to three times the size of the original input, while WASP limits the growth of the program to one third. Clasp(D) and WASP adopt a branching heuristics based on BerkMin [16] with differences in the intermediate steps of the selection procedure. WASP extends the original BerkMin heuristics by using a look-ahead technique in place of the *"two"* function calculating the number of binary clauses in the neighborhood of a literal ℓ. Moreover, WASP introduces an additional criterion based on supportedness of answer sets for selecting among heuristically-equivalent candidate literals of the last undefined learned constraint. Clasp(D) instead uses as intermediate step a variant of the MOMS criterion. MOMS estimates the effect of the candidate literals in short clauses and is convenient for clasp(D) because Clark's completion produces many binary constraints.

6 Conclusion

In this paper we presented a new native ASP solver for propositional programs called WASP. WASP builds upon a number of techniques originally introduced in the neighboring fields of CP and SAT, which are extended and properly combined with techniques specifically defined for solving disjunctive ASP programs. The performance of WASP was assessed and compared with alternative implementations by running the System Track of the 3rd ASP Competition. Our analysis shows that WASP is efficient and can compete with the state-of-the-art solvers on this benchmark. The effects of a native implementation of support propagations in a learning-based ASP solver is also discussed, showing that this design choice pays off in terms of memory usage and time performance in specific benchmark domains. The experiments also outline some specific weakness of the implementation (e.g., in Beyond NP domains), which will be subject of future work. It is worth pointing out that the implementation of WASP is still in an initial phase, yet the results obtained up to now are encouraging.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–385 (1991)
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM Transactions on Database Systems 22, 364–418 (1997)
3. Alviano, M., Faber, W., Leone, N., Perri, S., Pfeifer, G., Terracina, G.: The disjunctive datalog system DLV. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog 2010. LNCS, vol. 6702, pp. 282–301. Springer, Heidelberg (2011)
4. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Twentieth International Joint Conference on Artificial Intelligence, IJCAI 2007, pp. 386–392. Morgan Kaufmann Publishers (2007)
5. Lierler, Y., Maratea, M.: Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR 2004. LNCS (LNAI), vol. 2923, pp. 346–350. Springer, Heidelberg (2003)

6. Grasso, G., Iiritano, S., Leone, N., Ricca, F.: Some DLV applications for knowledge management. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 591–597. Springer, Heidelberg (2009)
7. 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. Theory and Practice of Logic Programming 12, 361–381 (2012)
8. Manna, M., Oro, E., Ruffolo, M., Alviano, M., Leone, N.: The HiLeX system for semantic information extraction. In: Hameurlain, A., Küng, J., Wagner, R. (eds.) TLDKS V. LNCS, vol. 7100, pp. 91–125. Springer, Heidelberg (2012)
9. Ricca, F., Alviano, M., Dimasi, A., Grasso, G., Ielpa, S.M., Iiritano, S., Manna, M., Leone, N.: A Logic-Based System for e-Tourism. Fundamenta Informaticae 105, 35–55 (2010)
10. Calimeri, F., et al.: The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 388–403. Springer, Heidelberg (2011)
11. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem Proving. Communications of the ACM 5, 394–397 (1962)
12. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In: Proceedings of ICCAD 2001, pp. 279–285 (2001)
13. Gaschnig, J.: Performance measurement and analysis of certain search algorithms. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1979) TR CMU-CS-79-124
14. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting Combinatorial Search Through Randomization. In: Proceedings of AAAI/IAAI 1998, pp. 431–437. AAAI Press (1998)
15. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of DAC 2001, pp. 530–535. ACM (2001)
16. 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, pp. 142–149. IEEE Computer Society (2002)
17. Faber, W., Leone, N., Pfeifer, G.: Pushing Goal Derivation in DLP Computations. In: Gelfond, M., Leone, N., Pfeifer, G. (eds.) LPNMR 1999. LNCS (LNAI), vol. 1730, pp. 177–191. Springer, Heidelberg (1999)
18. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence 138, 181–234 (2002)
19. Koch, C., Leone, N., Pfeifer, G.: Enhancing Disjunctive Logic Programming Systems by SAT Checkers. Artificial Intelligence 15, 177–212 (2003)
20. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
21. Ben-Eliyahu, R., Dechter, R.: Propositional Semantics for Disjunctive Logic Programs. Annals of Mathematics and Artificial Intelligence 12, 53–87 (1994)
22. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. Inf. Process. Lett. 47, 173–180 (1993)
23. Goldberg, E., Novikov, Y.: Berkmin: A fast and robust sat-solver. Discrete Appl. Math. 155, 1549–1561 (2007)
24. Gebser, M., Schaub, T.: Tableau calculi for answer set programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 11–25. Springer, Heidelberg (2006)
25. Ward, J., Schlipf, J.: Answer Set Programming with Clause Learning. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR 2004. LNCS (LNAI), vol. 2923, pp. 302–313. Springer, Heidelberg (2003)
26. Ricca, F., Faber, W., Leone, N.: A Backjumping Technique for Disjunctive Logic Programming. AI Communications 19, 155–172 (2006)